

ХОРОШИЙ КОД, ПЛОХОЙ КОД

Паттерны
и антипаттерны
мастерства

Том Лонг



Good Code, Bad Code

THINK LIKE A SOFTWARE ENGINEER

TOM LONG



MANNING
SHELTER ISLAND

ХОРОШИЙ КОД, ПЛОХОЙ КОД

Паттерны
и антипаттерны
мастерства

Том Лонг

Санкт-Петербург
«БХВ-Петербург»
2023

УДК 004.4'236
ББК 32.973.26-018.1
Л76

Лонг Т.

Л76 Хороший код, плохой код: Пер. с англ. — СПб.: БХВ-Петербург, 2023. — 432 с.: ил.

ISBN 978-5-9775-1790-4

В книге изложены приемы и методы, обязательные при грамотном объектно-ориентированном программировании. Следование приведенным паттернам проектирования защитит программу от ошибок, проблем с совместимостью, неожиданных и непонятных отказов, а также обеспечит модульность кода. Рассмотрены примеры на языке Java, легко адаптируемые для C#, пояснены приемы тестирования, развития и адаптации кода с учетом меняющихся требований.

Книга адресована программистам на языках Java и C# и руководителям команд, практикующих объектно-ориентированное программирование.

УДК 004.4'236
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	<i>Олег Сивченко</i>
Зав. редакцией	<i>Людмила Гауль</i>
Перевод с английского	<i>Екатерины Черских</i>
Редактор	<i>Софья Исакова</i>
Компьютерная верстка	<i>Натальи Смирновой</i>
Корректор	<i>Анна Брезман</i>
Оформление обложки	<i>Зои Канторович</i>

Original English language edition published by Manning Publications.

Copyright (©) 2021 by Manning Publications.

Russian-language edition copyright (©) 2023 by BHV. All rights reserved.

Оригинальное издание на английском языке опубликовано Manning Publications.

© 2021 Manning Publications.

Издание на русском языке © 2023 ООО «БХВ». Все права защищены.

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

ISBN 978-1-61729-893-6 (англ.)
ISBN 978-5-9775-1790-4 (рус.)

© Manning Publications, 2021
© Перевод на русский язык, оформление.
ООО "БХВ-Петербург", ООО "БХВ", 2023

Оглавление

Предисловие	13
Благодарности.....	14
Об этой книге	15
Кому следует прочитать эту книгу	15
Как организована эта книга: дорожная карта	15
О коде	17
Дискуссионный форум liveBook	17
Как использовать советы, приведенные в этой книге	18
Дополнительные материалы.....	19
Об авторе.....	20
Об иллюстрации на обложке	21
Контрольный список хорошего кода (что следует учитывать).....	22
Соображения высокого уровня	22
Более конкретные соображения	22
Часть 1. В ТЕОРИИ	26
Глава 1. Качество кода	27
1.1. Как код становится программным обеспечением	29
1.2. Почему код должен быть качественным	32
1.2.1. Код должен работать	33
1.2.2. Код должен продолжать работать	33
1.2.3. Код должен быть адаптируемым к изменяющимся требованиям	33
1.2.4. Не следует изобретать велосипед.....	35
1.3. Столпы качества кода	36
1.3.1. Сделать код читабельным	37
1.3.2. Избегать сюрпризов	38
1.3.3. Сделать код трудным для неправильного использования	40
1.3.4. Сделать код модульным	41

1.3.5. Сделать код многоразовым и обобщаемым.....	43
1.3.6. Сделать код тестируемым и протестировать его должным образом	44
1.4. Замедляет ли нас написание высококачественного кода?	46
Итоги.....	48
Глава 2. Слои абстракции.....	49
2.1. Нулевые значения и соглашение псевдокода в этой книге	50
2.2. Зачем создавать слои абстракции?.....	51
2.3. Слои кода	55
2.3.1. API и детали реализации	56
2.3.2. Функции.....	57
2.3.3. Классы	60
2.3.4. Интерфейсы	69
2.3.5. Когда слои становятся слишком тонкими	73
2.4. Как насчет микросервисов?	75
Итоги.....	77
Глава 3. Другие программисты и соглашения по написанию кода.....	78
3.1. Ваш код и код других программистов.....	78
3.1.1. Вещи, которые очевидны для вас, не очевидны для других	80
3.1.2. Другие программисты непреднамеренно попытаются взломать ваш код.....	81
3.1.3. Со временем вы забудете о своем собственном коде	81
3.2. Как другие поймут, как надо использовать ваш код?	82
3.2.1. Изучение названий объектов	83
3.2.2. Анализ типов данных	83
3.2.3. Чтение документации	83
3.2.4. Личный визит	84
3.2.5. Изучение вашего кода	84
3.3. Соглашения по написанию кода	85
3.3.1. Мелкий шрифт в соглашениях.....	85
3.3.2. Не слишком полагайтесь на мелкий шрифт	88
3.4. Проверки и утверждения	93
3.4.1. Проверки.....	94
3.4.2. Утверждения.....	96
Итоги.....	97
Глава 4. Ошибки.....	99
4.1. Возможность восстановления	99
4.1.1. Ошибки, которые могут быть исправлены	100
4.1.2. Ошибки, которые не могут быть исправлены	100
4.1.3. Часто только вызывающий код знает, можно ли исправить ошибку.....	101
4.1.4. Информировать вызывающих об ошибках, которые они могут захотеть исправить	103
4.2. Надежность в сравнении с отказом.....	104
4.2.1. Быстрая ошибка.....	104
4.2.2. Громкая ошибка	106

4.2.3. Объем восстанавливаемости	106
4.2.4. Не скрывайте ошибки	108
4.3. Способы сигнализации об ошибках.....	113
4.3.1. Резюме: исключения	114
4.3.2. Явный: проверенные исключения	115
4.3.3. Неявный: непроверенные исключения	117
4.3.4. Явный: тип возвращаемого значения, допускающий значение Null.....	119
4.3.5. Явный: возвращаемый тип Result.....	121
4.3.6. Явный: возвращаемый тип Outcome.....	124
4.3.7. Неявный: обещание или будущее.....	126
4.3.8. Неявный: возврат магического значения	129
4.4. Сигнализация ошибок, после которых нельзя восстановиться	130
4.5. Сообщения об ошибках, от которых вызывающий абонент может захотеть восстановиться.....	130
4.5.1. Аргументы в пользу использования непроверенных исключений	131
4.5.2. Аргументы в пользу использования явных методов	134
4.5.3. Мое мнение: используйте явную технику	137
4.6. Не игнорируйте предупреждения компилятора	138
Итоги.....	140

Часть 2. В ПРАКТИКЕ..... 141

Глава 5. Сделайте код читаемым 143

5.1. Используйте описательные имена	143
5.1.1. Неописательные имена затрудняют чтение кода.....	143
5.1.2. Комментарии являются плохой заменой описательных имен	144
5.1.3. Решение: сделайте имена описательными.....	146
5.2. Используйте комментарии надлежащим образом.....	147
5.2.1. Избыточные комментарии могут быть вредными	147
5.2.2. Комментарии являются плохой заменой читаемого кода	148
5.2.3. Комментарии могут быть полезны для объяснения того, почему существует код	149
5.2.4. Комментарии могут содержать полезные аннотации высокого уровня	150
5.3. Не заикливайтесь на количестве строк кода.....	151
5.3.1. Избегайте краткого, но нечитаемого кода	152
5.3.2. Решение: сделайте код читабельным, даже если для этого требуется больше строк	153
5.4. Придерживайтесь последовательного стиля кодирования.....	154
5.4.1. Непоследовательный стиль кодирования может привести к путанице	154
5.4.2. Решение: примите руководство по стилю и следуйте ему.....	155
5.5. Избегайте глубоко вложенного кода	157
5.5.1. Глубоко вложенный код может быть трудночитаемым	157
5.5.2. Решение: реструктуризация для минимизации вложенности.....	158

5.5.3. Вложенность часто является результатом слишком больших усилий.....	159
5.5.4. Решение: разбить код на более мелкие функции	160
5.6. Сделайте вызовы функций читаемыми	161
5.6.1. Аргументы могут быть трудны для расшифровки	161
5.6.2. Решение: используйте именованные аргументы	162
5.6.3. Решение: используйте описательные типы	163
5.6.4. Иногда нет хорошего решения	164
5.6.5. Как насчет IDE?.....	165
5.7. Избегайте использования необъяснимых значений	166
5.7.1. Необъяснимые значения могут сбивать с толку	166
5.7.2. Решение: используйте хорошо названную константу	167
5.7.3. Решение: используйте хорошо названную функцию	168
5.8. Используйте анонимные функции надлежащим образом	170
5.8.1. Анонимные функции могут отлично подходить для небольших задач.....	171
5.8.2. Анонимные функции могут быть трудночитаемыми.....	172
5.8.3. Решение: вместо этого используйте именованные функции	172
5.8.4. Большие анонимные функции могут быть проблематичными.....	174
5.8.5. Решение: разбейте большие анонимные функции на именованные функции	175
5.9. Используйте блестящие новые языковые функции надлежащим образом.....	177
5.9.1. Новые функции могут улучшить код.....	177
5.9.2. Неясные функции могут сбивать с толку	178
5.9.3. Используйте лучший инструмент для работы.....	178
Итого.....	179
Глава 6. Избегайте сюрпризов	180
6.1. Избегайте возврата магических чисел.....	181
6.1.1. Магические числа могут привести к ошибкам.....	181
6.1.2. Решение: вернуть значение null, необязательное значение или ошибку	183
6.1.3. Иногда магические числа могут произойти случайно.....	185
6.2. Используйте шаблон нулевого объекта соответствующим образом.....	188
6.2.1. Возврат пустой коллекции может улучшить код.....	188
6.2.2. Возврат пустой строки иногда может быть проблематичным.....	190
6.2.3. Более сложные нулевые объекты могут привести к сюрпризам	193
6.2.4. Реализация нулевого объекта может привести к сюрпризам.....	194
6.3. Избегайте возникновения неожиданных побочных эффектов.....	196
6.3.1. Побочные эффекты, которые очевидны и преднамеренны, — это нормально.....	197
6.3.2. Неожиданные побочные эффекты могут быть проблемой	197
6.3.3. Решение: избегайте побочного эффекта или сделайте его очевидным	202
6.4. Остерегайтесь изменения входных параметров	203
6.4.1. Изменение входного параметра может привести к ошибкам	203
6.4.2. Решение: скопируйте, прежде чем изменять.....	205
6.5. Избегайте написания вводящих в заблуждение функций	206
6.5.1. Бездействие при отсутствии критических входных данных может вызвать сюрпризы	206

6.5.2. Решение: сделать необходимые критические входные данные обязательными	209
6.6. Обработка перечислений, ориентированная на будущее	211
6.6.1. Неявная обработка будущих перечисляемых значений может быть источником проблем.....	212
6.6.2. Решение: используйте исчерпывающий оператор switch	214
6.6.3. Остерегайтесь результата по умолчанию	216
6.6.4. Предостережение: полагаться ли на перечисление из другого проекта	218
6.7. Нельзя ли все эти проблемы решить тестированием?	218
Итоги.....	219

Глава 7. Постарайтесь затруднить неверное использование кода..... 221

7.1. Подумайте о том, чтобы сделать вещи неизменными	222
7.1.1. Изменяемые классы могут быть легко использованы не по назначению.....	223
7.1.2. Решение: устанавливайте значения только во время проектирования	225
7.1.3. Решение: используйте шаблон проектирования для неизменности.....	227
7.2. Подумайте о том, чтобы сделать вещи глубоко неизменяемыми.....	233
7.2.1. Глубокая изменяемость может привести к неправильному использованию	233
7.2.2. Решение: защитное копирование.....	236
7.2.3. Решение: используйте неизменяемые структуры данных.....	238
7.3. Избегайте слишком общих типов данных.....	239
7.3.1. Чрезмерно общие типы могут использоваться неправильно	239
7.3.2. Парные типы легко использовать неправильно	243
7.3.3. Решение: используйте специальный тип	245
7.4. Работа со временем	247
7.4.1. Представление времени целыми числами может быть проблематичным	247
7.4.2. Решение: используйте соответствующие структуры данных для определения времени.....	251
7.5. Имейте единственные источники достоверных данных.....	253
7.5.1. Вторые источники истины могут привести к неверным состояниям	254
7.5.2. Решение: используйте первичные данные в качестве единственного источника истины	255
7.6. Имейте единственные источники истины для логики	258
7.6.1. Множественные источники истины для логики могут привести к ошибкам	258
7.6.2. Решение: имейте единый источник истины	260
Итоги.....	262

Глава 8. Сделайте код модульным..... 264

8.1. Рассмотрите возможность использования внедрения зависимости	264
8.1.1. Жестко запрограммированные зависимости могут повлечь за собой проблемы	265
8.1.2. Решение: используйте внедрение зависимостей.....	267
8.1.3. Разрабатывайте код с учетом внедрения зависимостей	269

8.2. Предпочитайте зависимость от интерфейсов	271
8.2.1. Зависимость от конкретных реализаций ограничивает адаптивность	271
8.2.2. Решение: по возможности полагаться на интерфейсы	272
8.3. Остерегайтесь наследования классов	273
8.3.1. Наследование классов может приводить к проблемам	274
8.3.2. Решение: использовать композицию	279
8.3.3. Как насчет подлинных отношений «is-a»?	282
8.4. Классы должны заботиться о себе сами	285
8.4.1. Чрезмерная забота о других классах может быть проблематичной	285
8.4.2. Решение: заставить классы заботиться о себе самих	286
8.5. Инкапсулируйте связанные данные вместе	288
8.5.1. Неинкапсулированные данные могут быть трудны в обработке	288
8.5.2. Решение: сгруппировать связанные данные в объекты или классы	290
8.6. Остерегайтесь утечки деталей реализации в типах возвращаемых значений	291
8.6.1. Утечка деталей реализации в типе возвращаемого значения может быть проблематичной	292
8.6.2. Решение: вернуть тип, соответствующий слою абстракции	293
8.7. Остерегайтесь утечки деталей реализации в исключениях	295
8.7.1. Утечка деталей реализации в исключениях может быть проблематичной	295
8.7.2. Решение: сделайте исключения соответствующими слою абстракции	297
Итоги	300
Глава 9. Сделайте код многозадачным и обобщаемым	301
9.1. Остерегайтесь предположений	301
9.1.1. Предположения могут привести к ошибкам при повторном использовании кода	302
9.1.2. Решение: избегайте ненужных предположений	303
9.1.3. Решение: если предположение необходимо, приведите его в исполнение ..	304
9.2. Остерегайтесь глобального состояния	307
9.2.1. Глобальное состояние может сделать повторное использование небезопасным	309
9.2.2. Решение: общее состояние с внедрением зависимостей	312
9.3. Используйте возвращаемые значения по умолчанию надлежащим образом	315
9.3.1. Возвращаемые значения по умолчанию в низкоуровневом коде могут повредить возможности повторного использования	316
9.3.2. Решение: укажите значения по умолчанию в коде более высокого уровня	318
9.4. Сосредоточьтесь на функциональных параметрах	321
9.4.1. Функцию, которая требует больше, чем ей нужно, может быть трудно повторно использовать	321
9.4.2. Решение: заставьте функции принимать только то, что им нужно	323
9.5. Рассмотрите возможность использования универсальных типов	324
9.5.1. В зависимости от конкретного типа ограничивайте обобщаемость	325
9.5.2. Решение: используйте универсальные типы	326
Итоги	327

Часть 3. Модульное тестирование.....	1
Глава 10. Принципы модульного тестирования	331
10.1. Учебник по модульному тестированию	332
10.2. Что делает модульный тест хорошим?	334
10.2.1. Точное обнаружение поломок	335
10.2.2. Не зависит от деталей реализации	336
10.2.3. Хорошо объяснимые сбои	338
10.2.4. Понятный тестовый код	339
10.2.5. Легкий и быстрый запуск	340
10.3. Сосредоточьтесь на общедоступном API, но не игнорируйте важные параметры поведения	340
10.4. Имитированные реализации	347
10.4.1. Причины использования имитированной реализации	347
10.4.2. Имитации	353
10.4.3. Заглушки	355
10.4.4. Имитации и заглушки могут быть проблематичными	357
10.4.5. Подделки	362
10.4.6. Школы мыслей об имитациях	365
10.5. Выбор из основных принципов тестирования	367
Итоги.....	368
Глава 11. Методы модульного тестирования.....	370
11.1. Тестируйте поведение, а не только функции	371
11.1.1. Одного контрольного примера для каждой функции часто недостаточно	371
11.1.2. Решение: сосредоточьтесь на тестировании каждого поведения	373
11.2. Избегайте делать вещи видимыми только для тестирования.....	376
11.2.1. Тестирование приватных функций — это обычно плохая идея	377
11.2.2. Решение: выбирайте тестирование через общедоступный API.....	380
11.2.3. Решение: разделите код на более мелкие модули	381
11.3. Тестируйте одну модель поведения за раз	385
11.3.1. Одновременное тестирование нескольких моделей поведения может привести к некачественным тестам	385
11.3.2. Решение: протестируйте каждую модель поведения в своем собственном контрольном примере	388
11.3.3. Параметризованные тесты	389
11.4. Используйте совместную настройку тестирования надлежащим образом.....	391
11.4.1. Совместное состояние может быть проблематичным	393
11.4.2. Решение: избегайте совместного использования состояния или сбросьте его	396
11.4.3. Совместная конфигурация может быть проблематичной	397
11.4.4. Решение: определить, какая конфигурация важна в контрольных примерах	401
11.4.5. Когда использование совместной конфигурации возможно	403

11.5. Используйте подходящие обнаружители овпадений в утверждениях	405
11.5.1. Неподходящие средства обнаружения совпадений могут привести к плохо объясненным сбоям	405
11.5.2. Решение: используйте подходящий обнаружитель совпадений	407
11.6. Используйте внедрение зависимостей для улучшения тестируемости.....	409
11.6.1. Жестко запрограммированные зависимости могут сделать невозможным тестирование кода	409
11.6.2. Решение: использовать внедрение зависимостей	411
11.7. Несколько заключительных слов о тестировании.....	412
Итоги.....	413
 Часть 4. Приложения	415
Приложение А. Рецепт шоколадных пирожных	417
Приложение В. Null-безопасность и необязательные значения	418
В.1. Использование null-безопасности	418
Проверка на наличие значений null.....	419
В.2. Использование optional.....	420
Приложение С. Дополнительные примеры кода	422
Шаблон проектирования.....	422
 Предметный указатель.....	425

Предисловие

Я занимаюсь программированием в той или иной форме с 11 лет, поэтому к тому времени, когда я получил свою первую работу в качестве программного инженера, я написал довольно много кодов. Несмотря на это, я быстро обнаружил, что кодирование и разработка программного обеспечения — это не одно и то же. Кодирование в качестве разработчика программного обеспечения означало, что мой код должен быть понятен другим людям и не ломаться, когда они что-то меняют. Это также означало, что были реальные люди (иногда много), которые использовали мой код и полагались на него, поэтому последствия того, что что-то пошло не так, были намного серьезнее.

По мере того как программный инженер становится более опытным, он узнает, как решения, которые он принимает в своем повседневном кодировании, могут иметь серьезные последствия для того, будет ли программное обеспечение работать должным образом, продолжать правильно работать и поддерживаться другими. Чтобы научиться писать хороший код (с точки зрения разработки программного обеспечения), может потребоваться много лет. Эти навыки часто приобретаются медленно и произвольно, поскольку инженеры учатся на собственных ошибках или получают частичные советы от более старших инженеров, с которыми они работают.

Эта книга направлена на то, чтобы дать начинающим программистам толчок к приобретению этих навыков. Она дает некоторые из наиболее важных уроков и теоретических основ написания кода, который будет надежным, поддерживаемым и адаптируемым к меняющимся требованиям. Я надеюсь, что вы сочтете ее полезной.

Благодарности

Написание книги — это не личный труд, и я хочу поблагодарить всех, кто приложил руку к воплощению этой книги в жизнь. В частности, я хотел бы поблагодарить своего редактора по развитию Тони Арритолу за терпеливое руководство в процессе написания книги, а также за ее постоянное внимание к читателю и качественное преподавание. Я также хотел бы поблагодарить моего рецензента издательства Эндрю Уолдрона за то, что он в первую очередь поверил в идею книги и за многочисленные бесценные идеи, полученные в процессе. Я также хотел бы поблагодарить моего редактора по технической разработке Майкла Дженсена за его глубокие технические идеи и предложения на протяжении всей книги. И спасибо моему техническому корректору Крису Вильянуэву за внимательное изучение кода и технического содержания книги, а также за все прекрасные предложения.

Я также хотел бы поблагодарить всех рецензентов — Амру Умудлу, Криса Вильянуэву, Дэвида Рэйси, Джорджа Томаса, Гири Свамнатана, Харрисона Масеко, Хоули Уолдман, Хизера Уорда, Генри Линя, Джейсона Тейлора, Джеффа Ноймана, Джо Иванса, Джошуа Сандемана, Кушика Викрама, Марселя ван ден Бринка, Себастьяна Ларссона, Себастьяна Пальма, Срути С., Чарли Римса, Эудженио Маркиори, Цзин Тана, Андрея Молчанова и Сатьяки Упадхьяй — которые нашли время, чтобы прочитать книгу на нескольких этапах в процессе ее разработки и предоставить точную и действенную обратную связь. Трудно переоценить важность и полезность этих отзывов.

Почти все концепции в книге — это хорошо зарекомендовавшие себя идеи и методы в сообществе разработчиков программного обеспечения, поэтому в качестве заключительного признания я хотел бы поблагодарить всех тех, кто на протяжении многих лет вносил свой вклад в совокупность знаний и делился ими.

Об этой книге

«*Хороший код, плохой код*» знакомит с ключевыми концепциями и методами, которые профессиональные программные инженеры регулярно используют для создания надежного и поддерживаемого кода. Вместо того чтобы просто перечислять, что можно и чего нельзя, книга призвана объяснить основные аргументы, лежащие в основе каждой концепции и техники, а также любые компромиссы. Это должно помочь читателям развить фундаментальное понимание того, как мыслить и кодировать как опытный программный инженер.

Кому следует прочесть эту книгу

Эта книга предназначена для людей, которые уже умеют программировать, но хотят улучшить свои навыки программирования в качестве программиста в профессиональной среде. Эта книга будет наиболее полезна любому, у кого есть опыт работы программистом от нуля до трех лет. Более опытные инженеры, вероятно, обнаружат, что они уже знают многое из этой книги, но я надеюсь, что они все равно сочтут ее полезным ресурсом для наставничества других.

Как организована эта книга: дорожная карта

Книга состоит из одиннадцати глав, разбитых на три части. Первая часть знакомит с некоторыми теоретическими концепциями высокого уровня, которые формируют наше представление о коде. Вторая часть переходит к практическим урокам. Каждая глава в *части 2* разделена на серию тем, охватывающих конкретное соображение или методику. Третья и последняя часть книги посвящена принципам и методам создания эффективных и поддерживаемых модульных тестов.

Общая схема в отдельных разделах книги — продемонстрировать сценарий (и некоторый код), который может быть проблематичным, а затем показать альтернативный подход, который устраняет некоторые или все проблемы. В этом смысле разделы имеют тенденцию демонстрировать прогресс от «плохого» кода к «хорошему» с оговоркой, что термины *плохой* и *хороший* субъективны и зависят от контекста. И, как подчеркивается в книге, часто необходимо учитывать нюансы и компромиссы, что означает, что это различие не всегда четко выражено.

Часть 1 «В теории» закладывает основы для некоторых всеобъемлющих и немного более теоретических соображений, которые формируют наш подход к написанию кода как программных инженеров.

- ♦ *Глава 1* знакомит с концепцией *качества кода* и, в частности, с практическим набором целей, которые мы стремимся достичь с помощью высококачественного кода. Затем она расширяет их до шести «столпов качества кода», которые обеспечивают высокоуровневые стратегии, которые можно использовать в повседневном кодировании.
- ♦ *Глава 2* обсуждает *слои абстракции*, фундаментальное соображение, которое определяет, как мы структурируем и разделяем код на отдельные части.
- ♦ *Глава 3* подчеркивает важность размышлений о других инженерах, которым придется работать с нашим кодом. Далее обсуждаются *соглашения по написанию кода* и то, как тщательно продумать их, чтобы предотвратить ошибки.
- ♦ *Глава 4* обсуждает ошибки и почему тщательное обдумывание того, как их определять и обрабатывать, является жизненно важной частью написания хорошего кода.

Часть 2 «В практике» охватывает первые пять столпов качества кода (установленных в *главе 1*) более практичным образом с конкретными методами и примерами.

- ♦ *Глава 5* посвящена тому, как сделать код читаемым и гарантировать, что другие инженеры смогут разобраться в нем.
- ♦ *Глава 6* посвящена предотвращению неожиданностей, что сводит к минимуму вероятность ошибок, гарантируя, что другие инженеры не будут неправильно интерпретировать то, что делает фрагмент кода.
- ♦ *Глава 7* посвящена тому, как сделать код трудным для неправильного использования, что сводит к минимуму вероятность ошибок, затрудняя инженерам случайное создание кода, который логически неверен или нарушает допущения.
- ♦ *Глава 8* посвящена созданию модульного кода, ключевому методу, который помогает гарантировать, что код демонстрирует чистые уровни абстракции и что он будет адаптироваться к изменяющимся требованиям.
- ♦ *Глава 9* посвящена повторному использованию кода и обобщению. Это делает добавление новых функций или создание новых функций проще и безопаснее, предотвращая необходимость изобретать велосипед.

Часть 3 «Модульное тестирование» знакомит с ключевыми принципами и практиками, которые используются при написании эффективных модульных тестов.

- ♦ *Глава 10* знакомит с рядом принципов и соображений более высокого уровня, которые влияют на то, как мы проводим модульное тестирование кода.
- ♦ *Глава 11* основывается на принципах, изложенных в *главе 10*, и дает ряд конкретных практических советов по написанию модульных тестов.

Идеальный способ прочитать эту книгу — от корки до корки, потому что идеи, изложенные в предыдущих частях книги, закладывают основы для последующих час-

тей. Но, несмотря на это, темы в *части 2* (и *глава 11*), как правило, довольно самостоятельны, и каждая занимает всего несколько страниц, поэтому большинство из них будут полезны, даже если читать отдельно. Это сделано намеренно, с целью предоставить эффективный способ быстро объяснить установившуюся передовую практику другому инженеру. Это должно быть полезно для любых инженеров, желающих объяснить конкретную концепцию в обзоре кода или во время наставничества другого инженера.

О коде

Книга предназначена для инженеров, которые пишут код на статически типизированном объектно-ориентированном языке программирования, например, на одном из следующих: Java, C#, TypeScript, JavaScript (ECMAScript 2015 или более поздней версии со статической проверкой типов), C++, Swift, Kotlin, Dart 2 или аналогичный. Концепции, описанные в этой книге, широко применимы при написании кода на языке, подобном одному из перечисленных.

Разные языки программирования имеют разные синтаксисы и парадигмы для выражения логики и структуры кода. Но, для того чтобы предоставить примеры кода в этой книге, необходимо стандартизировать какой-то синтаксис и набор парадигм. Для этого в книге используется псевдокод, который заимствует идеи из нескольких разных языков. Цель псевдокода — быть четким, понятным и легко узнаваемым для наибольшего числа инженеров. Пожалуйста, имейте в виду это утилитарное намерение; книга не ставит своей целью утверждать, что какой-либо один язык лучше или хуже любого другого.

Аналогичным образом, там, где существует компромисс между однозначностью и краткостью, примеры псевдокода имеют тенденцию ошибаться в сторону однозначности. Одним из примеров этого является использование явных типов переменных, в отличие от предполагаемых типов с таким ключевым словом, как `var`. Другим примером является использование операторов `if` для обработки нулей, а не более кратких (но, возможно, менее знакомых) операторов объединения нулей и условных операторов `null` (см. *приложение В*). В реальных кодовых базах (и вне контекста книги) инженеры, возможно, пожелают сделать больший акцент на лаконичности.

Дискуссионный форум liveBook

Приобретение *«Хороший код, плохой код: ремесло и мастерство программиста»* включает бесплатный доступ к частному веб-форуму, управляемому Manning Publications, где вы можете комментировать книгу, задавать технические вопросы и получать помощь от автора и других пользователей. Чтобы получить доступ к форуму, перейдите по адресу <https://livebook.manning.com/#!/book/good-codebad-code/discussion>. Вы также можете узнать больше о форумах Manning и правилах поведения на <https://livebook.manning.com/#!/discussion>.

Обязательство Manning перед нашими читателями состоит в том, чтобы обеспечить место, где может состояться содержательный диалог между отдельными читателями, а также между читателями и автором. Это не является обязательством какого-либо определенного количества участников со стороны автора, чей вклад в форум остается добровольным (и неоплачиваемым). Мы предлагаем вам попробовать задать автору несколько сложных вопросов, чтобы не потерять его интерес! Форум и архивы предыдущих обсуждений будут доступны с веб-сайта издателя, пока книга находится в печати.

Как использовать советы, приведенные в этой книге

Читая любую книгу или статью о разработке программного обеспечения, всегда стоит помнить, что это субъективная тема и что решения реальных проблем редко бывают четкими. По моему опыту лучшие инженеры подходят ко всему, что они читают, со здоровым скептицизмом и желанием понять фундаментальное мышление, стоящее за этим. Мнения расходятся и развиваются, а доступные инструменты и языки программирования постоянно совершенствуются. Понимание причин, лежащих в основе конкретного совета, его контекста и его ограничений, необходимо для того, чтобы знать, когда его применять, а когда игнорировать.

Эта книга призвана собрать ряд полезных тем и методов, которые помогут инженерам в написании лучшего кода. Несмотря на то что, вероятно, разумно учитывать эти вещи, ничто в этой книге не следует считать непогрешимым или применять в качестве жесткого и строгого правила, которое никогда нельзя нарушать. Здравый смысл — важнейший компонент хорошей инженерии.

Дополнительные материалы

Книга призвана стать ступенькой в мир программирования в качестве программного инженера. Она должна дать читателю общее представление о том, как думать о коде, о вещах, которые могут быть проблематичными, и о методах, позволяющих избежать этих проблем. Но путешествие на этом не должно заканчиваться; разработка программного обеспечения — это огромная и постоянно развивающаяся предметная область, и настоятельно рекомендуется толковать широко и быть в курсе событий. Помимо статей и блогов есть книги, которые могут быть полезными читателям:

Refactoring: Improving the Design of Existing Code, second edition, Martin Fowler (Addison-Wesley, 2019)

Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Martin (Prentice Hall, 2008)

Code Complete: A Practical Handbook of Software Construction, second edition, Steve McConnell (Microsoft Press, 2004)

The Pragmatic Programmer: Your Journey to Mastery, 20th anniversary, second edition, David Thomas and Andrew Hunt (Addison-Wesley 2019)

Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1994)

Effective Java, third edition, Joshua Bloch (Addison-Wesley, 2017)

Unit Testing: Principles, Practices and Patterns, Vladimir Khorikov (Manning Publications, 2020)

Об авторе

Том Лонг — инженер-программист в Google. Работает техническим руководителем и помимо прочих задач регулярно обучает новых инженеров-программистов передовым методам профессионального кодирования.

Об иллюстрации на обложке

Рисунок на обложке книги *«Хороший код, плохой код»* озаглавлен «Homme Zantiote», или человек с острова Закинтос в Греции. Иллюстрация взята из коллекции костюмов из разных стран Жака Грассе де Сен-Совера (1757–1810) под названием *«Костюмы разных стран»*, опубликованной во Франции в 1797 году. Каждая иллюстрация тонко прорисована и раскрашена вручную. Богатое разнообразие коллекции Grasset de Saint-Sauveur живо напоминает нам о том, насколько культурно обособленными были города и регионы мира всего 200 лет назад. Изолированные друг от друга люди говорили на разных диалектах и языках. На улицах или в сельской местности было легко определить, где они живут и каково их ремесло или положение в жизни, просто по их одежде.

С тех пор то, как мы одеваемся, изменилось, и разнообразие по регионам, столь богатое в то время, исчезло. Сейчас трудно отличить жителей разных континентов, не говоря уже о разных городах, регионах или странах. Возможно, мы обменяли культурное разнообразие на более разнообразную личную жизнь — безусловно, на более разнообразную и быстро развивающуюся технологическую жизнь.

В то время когда трудно отличить одну компьютерную книгу от другой, Manning отмечает изобретательность и инициативу компьютерного бизнеса с обложками книг, основанными на богатом разнообразии региональной жизни двухвековой давности, оживленной картинами Грассе де Сен-Совера.

Контрольный список хорошего кода (что следует учитывать)

Соображения высокого уровня

Создает ли код чистые уровни абстракции?

- ◆ Детали реализации скрыты? (разд. 2.3.1)
- ◆ Функции соответствующего размера? (разд. 2.3.2)
- ◆ Классы соответствующего размера? (разд. 2.3.3)
- ◆ Интерфейсы, где это уместно? (разд. 3.2.4)

Ясно ли кодовое соглашение?

- ◆ Количество мелкого шрифта сведено к минимуму? (разд. 3.3.2)
- ◆ Применяется мелкий шрифт? (разд. 3.4)

Сигнализируются ли ошибки и обрабатываются ли они надлежащим образом?

- ◆ Можно ли исправить ошибки? (разд. 4.1)
- ◆ Код быстро выходит из строя при возникновении ошибки? (разд. 4.2.1)
- ◆ Ошибки не останутся незамеченными? (разд. 4.2.2–4.2.4)
- ◆ Неявная или явная сигнализация? (разд. 4.3–4.5)

Более конкретные соображения

Читаем ли код?

- ◆ Понятный и последовательный? (разд. 5.1–5.4)
- ◆ Избегает глубокой вложенности (или отступов)? (разд. 5.5)
- ◆ Вызовы функций легко расшифровываются? (разд. 5.6)
- ◆ Объяснены значения и константы? (разд. 5.7)
- ◆ Отсутствуют большие анонимные функции? (разд. 5.8)
- ◆ Блестящие функции используются по правильным причинам? (разд. 5.9)

Позволяет ли код избежать неожиданностей?

- ◆ Избегать неожиданных возвращаемых значений? (разд. 6.1–6.2)
- ◆ Избегать неожиданных побочных эффектов? (разд. 6.3–6.4)
- ◆ Функции всегда делают то, что они утверждают? (разд. 6.5)
- ◆ Обработка перечислений с расчетом на будущее? (разд. 6.6)

Трудно ли неправильно использовать код?

- ◆ Вещи достаточно неизменные? (разд. 7.1–7.2)
- ◆ Типы данных не чрезмерно общие? (разд. 7.3–7.4)
- ◆ Источники истины едины? (разд. 7.5–7.6)

Является ли код достаточно модульным?

- ◆ Используется внедрение зависимостей? (разд. 8.1)
- ◆ Наследования классов не используется чрезмерно? (разд. 8.3)
- ◆ Классы заботятся только о себе? (разд. 8.4)
- ◆ Связанные данные инкапсулированы вместе? (разд. 8.5)
- ◆ Не просочились детали реализации? (разд. 8.6–8.7)

Является ли код многоцветным и обобщаемым?

- ◆ Хрупкие предположения отсутствуют? (разд. 9.1)
- ◆ Глобальное состояние сведено к минимуму? (разд. 9.2)
- ◆ Тщательно продуманы значения по умолчанию? (разд. 9.3)
- ◆ Универсальные типы там, где это уместно? (разд. 9.5)

Правильно ли протестирован код?

- ◆ Точное обнаружение поломок? (разд. 10.2.1)
- ◆ Независим от реализации? (разд. 10.2.2, 10.3)
- ◆ Хорошо объясненные неудачи? (разд. 10.2.3)
- ◆ Имитированные реализации используются надлежащим образом? (разд. 10.4)
- ◆ Каждое поведение проверено? (разд. 11.1)
- ◆ Рассмотрены все уровни тестирования? (разд. 11.7)

Часть 1

В теории

Мир разработки программного обеспечения полон советов и мнений о том, как лучше всего писать код. Но жизнь редко сводится к слепому следованию им. Во-первых, разные советы из разных источников часто могут противоречить друг другу. Как же тогда мы узнаем, каким из них следовать? Еще более важно то, что разработка программного обеспечения не является точной наукой и ее нельзя свести к набору непогрешимых правил (как бы мы ни старались). Каждый проект индивидуален, и почти всегда нужно идти на компромиссы.

Чтобы написать хороший код, нам нужно применять здравый смысл к конкретному сценарию и уметь продумывать последствия (хорошие и плохие) определенного способа выполнения чего-либо. Для этого нам нужно понять главное: чего мы на самом деле хотим достичь, когда пишем код? И каковы соображения высокого порядка, которые помогают нам достичь этого? *Глава 1* направлена на то, чтобы обеспечить прочную основу теоретических аспектов написания хорошего кода.

1 Качество кода

ЭТА ГЛАВА ОХВАТЫВАЕТ

- ◇ Причину того, почему код должен быть качественным
- ◇ Четыре цели, на достижение которых направлен высококачественный код
- ◇ Шесть стратегий высокого уровня, которые мы можем использовать для обеспечения высокого качества кода
- ◇ Как написание высококачественного кода на самом деле экономит время и усилия в среднесрочной и долгосрочной перспективах

Вы, вероятно, использовали сотни, может быть, даже тысячи различных программных продуктов только за последний год. Мы очень много взаимодействуем с программным обеспечением: с программами, установленными на вашем компьютере, приложениями на вашем телефоне, кассами самообслуживания в супермаркетах.

Существует также множество программных продуктов, от которых мы зависим, даже не осознавая этого. Например, мы доверяем нашему банку, чтобы у него была хорошо работающая серверная система, которая не собирается неожиданно передавать содержимое нашего банковского счета кому-то другому или внезапно решать, что у вас миллионы долларов долга.

Иногда мы сталкиваемся с программами, которые доставляют нам массу удовольствия; они делают именно то, что мы хотим, содержат очень мало ошибок и просты в использовании. Но в других случаях мы сталкиваемся с частями программного обеспечения, которые просто ужасны в использовании. Они полны ошибок, постоянно выходят из строя и интуитивно непонятны.

Некоторые части программного обеспечения, очевидно, менее важны, чем другие; приложение на нашем телефоне, содержащее ошибку, вероятно, раздражает, но не станет причиной конца света. С другой стороны, ошибка во внутренней системе банка может привести к разрушению жизни. Даже проблемы в той части программного обеспечения, которая не кажется критичной, могут разрушить бизнес. Если пользователи находят какое-либо программное обеспечение раздражающим или сложным в использовании, то есть большая вероятность, что они переключатся на альтернативу.

Более качественный код, как правило, обеспечивает создание программного обеспечения, которое надежнее, проще в обслуживании и менее глючное. Многие принципы повышения качества кода касаются не только обеспечения того, чтобы

программное обеспечение изначально было хорошим, но и того, чтобы оно оставалось таким на протяжении всего срока службы по мере развития требований и появления новых сценариев. На рис. 1.1 показаны некоторые из способов, с помощью которых качество кода может повлиять на качество программного обеспечения.



Рис. 1.1. Высококачественный код увеличивает вероятность того, что программное обеспечение будет надежным, поддерживаемым и будет соответствовать требованиям. Некачественный код, как правило, делает обратное

Хороший код, очевидно, не является единственным ингредиентом, необходимым для создания хорошего программного обеспечения. Однако он в списке основных. У нас могут работать лучшие в мире команды по продуктам и маркетингу, мы можем разворачиваться на лучших платформах и программировать с использованием лучших фреймворков, но, в конце концов, все, что делает программное обеспечение, происходит потому, что кто-то написал часть кода, необходимую, чтобы это произошло.

Повседневные решения, которые инженеры принимают при написании кода, могут показаться второстепенными по отдельности, а иногда и вообще незначительными, но они в совокупности определяют, будет ли часть программного обеспечения хорошей или плохой. Если код содержит ошибки, неправильно настроен или непра-

вильно обрабатывает случаи ошибок, то программное обеспечение, созданное на его основе, скорее всего, будет глючным и нестабильным и, вероятно, не будет выполнять свою работу должным образом.

В этой главе определены четыре цели, на достижение которых должен быть направлен высококачественный код. Затем цели расширяются до шести высокоуровневых стратегий, которые мы можем использовать в нашей повседневной работе, чтобы гарантировать высокое качество написанного нами кода. В последующих главах книги эти стратегии рассматриваются более подробно, со многими проработанными примерами использования псевдокода.

1.1. Как код становится программным обеспечением

Прежде чем мы углубимся в обсуждение качества кода, стоит кратко обсудить, как код становится программным обеспечением. Если вы уже знакомы с процессом разработки и развертывания программного обеспечения, можете переходить к *разд. 1.2*. Если вы знаете, как программировать, но никогда раньше не имели практического опыта в области разработки программного обеспечения, то этот раздел даст вам обзор информации на хорошем уровне.

Программное обеспечение создано из кода; это очевидно и на самом деле не нуждается в объяснении. Что может быть менее очевидным (если у вас нет опыта работы инженером-программистом), так это процесс, посредством которого код становится программным обеспечением, запущенным в «дикой природе» (в руках пользователей или при выполнении бизнес-задач).

Код, как правило, не становится программным обеспечением, работающим в «дикой природе», в тот момент, когда инженер его пишет. Обычно существуют различные процессы и проверки, чтобы убедиться, что код выполняет то, для чего он предназначен, и ничего не нарушает. Их часто называют процессом разработки и развертывания программного обеспечения.

Нам не обязательно иметь подробные знания о процессе, чтобы спокойно читать эту книгу, но это поможет, по крайней мере, узнать о нем в общем виде. Для начала полезно ввести несколько терминов.

- ◆ *Кодовая база* — хранилище кода, из которого можно создавать части программного обеспечения. Обычно этим управляет система контроля версий, такая, как git, subversion, perforce и т. д.
- ◆ *Отправка кода* — иногда называемая «фиксацией кода» или «объединением запроса на извлечение». Программист обычно вносит изменения в код в локальной копии кодовой базы. Как только он будет доволен изменением, он отправит его в основную кодовую базу. Примечание: в некоторых установках вносить изменения в кодовую базу должен назначенный сопровождающий, а не автор, отправляющий их.

- ◆ *Проверка кода* — многие организации требуют, чтобы код был проверен другим инженером, прежде чем его можно будет отправить в кодовую базу. Это немного похоже на корректуру кода; вторая пара глаз часто обнаруживает проблемы, которые пропустил автор кода.
- ◆ *Проверки перед отправкой* — иногда называемые «хуками предсказания», «проверками предсказания» или «проверками предфиксации», они блокируют отправку изменений в базу кода в случае сбоя тестов или если код не компилируется.
- ◆ *Релиз* — часть программного обеспечения создается на основе моментального снимка кодовой базы. После тщательных проверок качества это затем выпускается в «дикую природу». Вы часто будете слышать фразу «выбраковка релиза» для обозначения процесса внесения определенной редакции в кодовую базу и создания релиза из нее.
- ◆ *Производство* — это правильный термин для «дикой природы», когда программное обеспечение развертывается на сервере или в системе (а не отправляется клиентам). Как только программное обеспечение выпущено и выполняет бизнес-задачи, считается, что оно запущено в производство.

Существует множество вариантов процесса превращения кода в программное обеспечение, работающее в «дикой природе», но ключевыми шагами в этих процессах обычно являются следующие:

1. Инженер работает над локальной копией кодовой базы для внесения изменений.
2. Как только инженер удовлетворен, он отправит эти изменения на проверку.
3. Другой инженер рассмотрит код и, возможно, предложит свои изменения.
4. Как только и автор, и рецензент будут довольны, код будет отправлен в кодовую базу.
5. Релизы будут периодически отбраковываться из кодовой базы. Частота выбраковки может варьироваться в разных организациях и командах (от нескольких минут до нескольких месяцев).
6. Если код провалит любой из тестов или не будет компилироваться, он не сможет быть отправлен в кодовую базу, либо же не состоится релиз.

На рис. 1.2 представлена схема типичного процесса разработки и развертывания программного обеспечения. У разных компаний и команд есть свои собственные варианты этого процесса, и уровень автоматизации отдельных частей процесса может сильно различаться.

Стоит отметить, что процессы разработки и развертывания программного обеспечения сами по себе являются обширными темами; им посвящено много отдельных книг. Вокруг них также существует множество различных структур и идеологий, и, если вам интересно, стоит прочесть об этом больше. Эта книга не посвящена главным образом этим темам, поэтому мы не будем рассматривать их более под-

робно. Все, что вам нужно, чтобы читать эту книгу, — это лишь приблизительное представление о том, как код становится программным обеспечением.

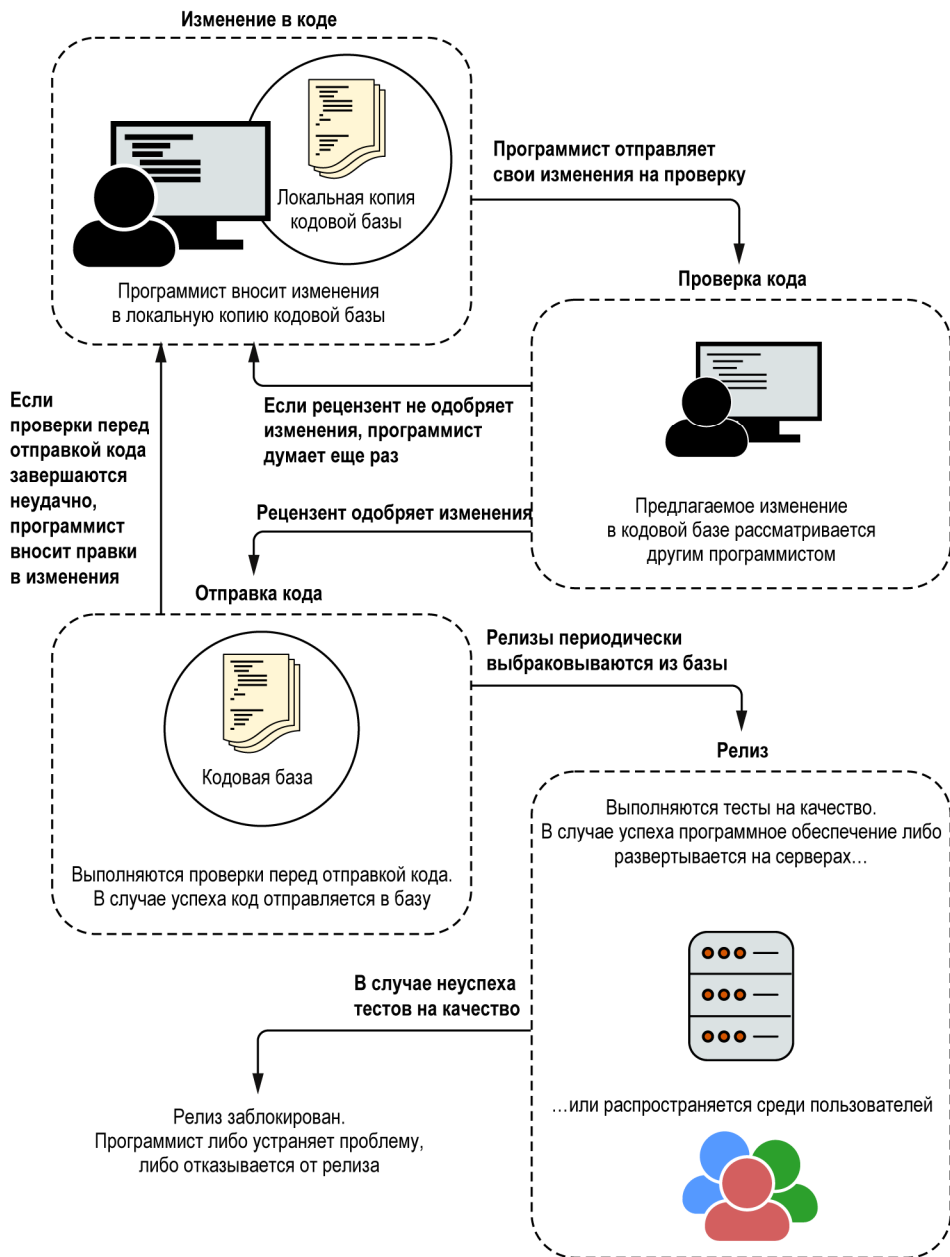


Рис. 1.2. Упрощенная схема типичного процесса разработки и развертывания программного обеспечения. Точные шаги и уровень автоматизации могут сильно различаться в разных организациях и командах

1.2. Почему код должен быть качественным

Если мы покупаем автомобиль, качество, вероятно, будет одним из наших главных соображений. Нам нужна машина, которая

- ◆ безопасна;
- ◆ действительно работает;
- ◆ не ломается;
- ◆ ведет себя предсказуемо: когда мы нажимаем на педаль тормоза, машина должна замедляться.

Если мы спросим кого-нибудь, что делает автомобиль качественным, один из наиболее вероятных ответов, который мы получим, — это то, что он хорошо построен. Это означает, что автомобиль был хорошо спроектирован, что он был протестирован на безопасность и надежность перед запуском в производство и что он был собран правильно. Создание программного обеспечения во многом аналогично: чтобы создавать высококачественное программное обеспечение, мы должны убедиться, что оно хорошо построено. Вот в чем заключается качество кода.

Слова «качество кода» иногда могут вызывать ассоциации с придирчивыми советами о тривиальных и неважных вещах. Вы, без сомнения, будете время от времени сталкиваться с этим, но на самом деле качество кода не об этом. Качество кода основано на исключительно практических соображениях. Иногда это может касаться мелких деталей, а иногда и больших, но цель одна и та же: создание лучшего программного обеспечения.

Несмотря на это, качество кода все еще может быть довольно сложной концепцией, понять которую сложно. Иногда мы можем увидеть определенный код и подумать: «Фу» или «Вау, это выглядит топорно», а в других случаях мы можем наткнуться на код и подумать: «Это превосходно». Не всегда очевидно, почему код вызывает такие реакции, и иногда это может быть просто инстинктивная реакция без реальной причины.

Определение кода как высококачественного или низкокачественного по своей сути является субъективным и основанным на личном суждении. Я попытаюсь быть немного более объективным в этом вопросе; лично я считаю полезным сделать шаг назад и подумать о том, чего я действительно пытаюсь достичь, когда пишу код. Код, который помогает мне достичь этих целей, по моему мнению, является высококачественным, а код, который мешает, является низкокачественным.

Есть четыре цели высокого уровня, которых я стремлюсь достичь при написании кода:

- ◆ Он должен работать.
- ◆ Он должен продолжать работать.
- ◆ Он должен быть адаптируемым к изменяющимся требованиям.
- ◆ Не следует изобретать велосипед.

В следующих нескольких подразделах эти четыре цели объясняются более подробно.

1.2.1. Код должен работать

Это настолько очевидно, что, вероятно, не стоит об этом и говорить, но я все равно скажу. Когда мы пишем код, мы пытаемся решить задачу, такую, как реализация функции, исправление ошибки или выполнение задачи. Основная цель нашего кода состоит в том, чтобы он работал; он должен решать задачу, которую мы намерены решить. Это также подразумевает, что код не содержит ошибок, поскольку наличие ошибок, скорее всего, помешает ему работать должным образом и полностью решить задачу.

При определении понятия «рабочий код» нам необходимо учесть все требования. Например, если задача, которую мы решаем, особенно чувствительна к производительности (например, задержка или загрузка ЦП), то обеспечение адекватной производительности нашего кода подпадает под пункт «код должен работать», потому что это часть требований. То же самое относится и к другим важным соображениям, таким, как конфиденциальность и безопасность пользователей.

1.2.2. Код должен продолжать работать

Код может работать очень быстро; он может работать сегодня, но как мы можем быть уверены, что он все еще будет работать завтра или через год? Это может показаться странным поводом для беспокойства; почему код внезапно может перестать работать? Дело в том, что код не живет изолированно, и, если мы не будем осторожны, он может легко сломаться, когда все вокруг изменится.

- ◆ Код, вероятно, зависит от другого кода, который будет скорректирован, обновлен и изменен.
- ◆ Любая необходимая новая функциональность может означать, что в коде требуются изменения.
- ◆ Задача, которую мы пытаемся решить, может со временем измениться: предпочтения потребителей, потребности бизнеса и технологические соображения могут меняться.

Код, который работает сегодня, но ломается завтра, когда меняется одна из его составляющих, не очень полезен. Часто бывает легко создать работающий код, но гораздо сложнее создать код, который продолжает работать. Обеспечение того, чтобы код продолжал работать, является одним из важнейших соображений, с которыми сталкиваются программисты, и это необходимо учитывать на всех этапах программирования. Нельзя рассматривать это в последнюю очередь, а также нельзя думать, что этого можно достичь, просто проведя дополнительные тесты.

1.2.3. Код должен быть адаптируемым к изменяющимся требованиям

Довольно редко бывает, что фрагмент кода пишется один раз, а затем никогда не изменяется снова. Непрерывная разработка части программного обеспечения

может длиться несколько месяцев, обычно несколько лет, а иногда даже десятилетий. На протяжении всего этого процесса меняются требования:

- ◆ Меняются реалии бизнеса.
- ◆ Меняются предпочтения потребителей.
- ◆ Предположения становятся недействительными.
- ◆ Постоянно добавляются новые функции.

Решение о том, сколько усилий нужно приложить, для того чтобы сделать код адаптируемым, может быть сложной задачей. С одной стороны, мы в значительной степени знаем, что требования к части программного обеспечения будут меняться с течением времени (крайне редко бывает, что этого не происходит). Но, с другой стороны, у нас часто нет уверенности в том, как именно они будут развиваться. Невозможно сделать абсолютно точные прогнозы о том, как часть кода или программного обеспечения будет меняться с течением времени. Но только потому что мы точно не знаем, как что-то будет развиваться, не означает, что мы должны полностью игнорировать тот факт, что оно будет развиваться. Чтобы проиллюстрировать это, давайте рассмотрим два крайних сценария.

- ◆ *Сценарий А.* Мы пытаемся точно предсказать, как требования могут измениться в будущем, и разработать наш код так, чтобы он поддерживал все потенциальные изменения. Скорее всего, мы потратим дни или недели на то, чтобы наметить все возможные пути развития кода и программного обеспечения. Затем нам придется тщательно продумать каждую деталь кода, который мы пишем, чтобы убедиться, что он поддерживает все потенциальные будущие требования. Это сильно замедлит нас; часть программного обеспечения, на разработку которой могло потребоваться три месяца, теперь может занять год или больше. И, в конце концов, это, вероятно, будет пустой тратой времени, потому что конкурент опередит нас на рынке на несколько месяцев и наши прогнозы о будущем, вероятно, в любом случае окажутся неверными.
- ◆ *Сценарий В.* Мы полностью игнорируем тот факт, что требования могут измениться. Мы пишем код, чтобы точно соответствовать текущим требованиям и не прилагаем никаких усилий для адаптации любой части кода. Хрупкие предположения укрепляются, и решения подзадач объединяются в большие неразделимые куски кода. Мы запускаем первую версию программного обеспечения в течение трех месяцев, но отзывы от первоначальной группы пользователей ясно показывают, что нам необходимо изменить некоторые функции и добавить новые, если мы хотим, чтобы программное обеспечение было успешным. Изменения в требованиях не являются массовыми, но, поскольку мы не учитывали адаптивность при написании кода, наш единственный вариант — выбросить все и начать заново. Затем нам придется потратить еще три месяца на переписывание программного обеспечения, и если требования снова изменятся, нам придется потратить еще три месяца на то, чтобы после этого переписать его снова. К тому времени, как мы создали программное обеспечение, отвечающее потребностям пользователей, конкурент в очередной раз опередил нас в этом.

Сценарий А и сценарий В представляют собой две противоположные крайности. Результаты в обоих сценариях довольно плохие, и ни один из них не является эффективным способом создания программного обеспечения. Вместо этого нам нужно найти подход где-то посередине этих двух крайностей. Нет единого ответа на вопрос, какая точка спектра между сценарием А и сценарием В является оптимальной. Это будет зависеть от вида проекта, над которым мы работаем, и от культуры организации, в которой мы работаем. К счастью, существуют некоторые общеприменимые методы, которые мы можем применить, чтобы обеспечить адаптивность кода без необходимости точно знать, как он может быть адаптирован в будущем. Мы рассмотрим многие из этих методов в этой книге.

1.2.4. Не следует изобретать велосипед

Когда мы пишем код для решения задачи, мы обычно берем одну большую задачу и разбиваем ее на несколько более мелких подзадач. Например, если бы мы писали код для загрузки файла изображения, преобразования его в изображение в оттенках серого, а затем сохранения его снова, подзадачи, которые нам нужно решить, заключаются в следующем:

- ◆ Загрузить несколько байтов данных из файла.
- ◆ Пропарсить байты данных в формат изображения.
- ◆ Преобразовать изображение в оттенки серого.
- ◆ Преобразовать изображение обратно в байты.
- ◆ Сохранить эти байты обратно в файл.

Многие из этих задач уже были решены другими; например, загрузка нескольких байтов из файла, скорее всего, является чем-то, для чего язык программирования имеет встроенную поддержку. Мы не стали бы писать собственный код для низкоуровневой связи с файловой системой. Точно так же, вероятно, существует библиотека, которую мы можем использовать для парсинга байтов в изображение.

Если мы все же напишем наш собственный код для низкоуровневой связи с файловой системой или для парсирования нескольких байтов в изображение, то мы фактически изобретаем колесо. Есть несколько причин, по которым лучше использовать существующее решение, чем изобретать его заново:

- ◆ *Это экономит время и усилия* — если мы используем встроенную поддержку для загрузки файла, это, вероятно, займет всего несколько строк кода и несколько минут нашего времени. Напротив, написание собственного кода для этого, скорее всего, потребует чтения многочисленных стандартов файловых систем и написания многих тысяч строк кода. Это, вероятно, заняло бы у нас много дней, если не недель.
- ◆ *Это уменьшает вероятность ошибок* — если где-то существует код для решения данной проблемы, то он уже должен быть тщательно протестирован. Также вероятно, что он уже используется в «дикой природе», поэтому вероятность того, что код содержит ошибки, снижается, потому что, если они и были, они, скорее всего, уже обнаружены и исправлены.

- ◆ *Это использует существующий опыт* — участники команды, поддерживающие код, который парсирует несколько байтов в изображение, вероятно, являются экспертами по кодированию изображений. Если выйдет новая версия JPEG-кодировки, они, скорее всего, узнают об этом и обновят свой код. Повторно используя их код, мы извлекаем выгоду из их опыта и будущих обновлений.
- ◆ *Это облегчает понимание кода* — если есть стандартизированный способ что-то сделать, то есть и разумная вероятность, что другой инженер увидел это раньше. Большинству инженеров, вероятно, в какой-то момент приходилось читать файл, и они мгновенно распознают встроенный способ сделать это и поймут, как он функционирует. Если мы будем использовать для этого свою собственную логику, то другие инженеры не будут знакомы с ней и не сразу поймут, как она функционирует.

Концепция отказа от изобретения колеса применима в обоих направлениях. Если другой программист уже написал код для решения подзадачи, то мы должны вызвать его код, а не писать свой собственный. Аналогично, если мы пишем код для решения подзадачи, то мы должны структурировать наш код таким образом, чтобы другим программистам было легко использовать его повторно, чтобы им не нужно было изобретать велосипед.

Одни и те же классы подзадач часто возникают снова и снова, поэтому преимущества совместного использования кода разными программистами и командами становятся очевидными очень быстро.

1.3. Столпы качества кода

Четыре цели, которые мы только что рассмотрели, помогают нам сосредоточиться на том, чего мы в основном пытаемся достичь, но они не дают конкретных советов о том, что делать в нашей повседневной работе. Полезно попытаться определить более конкретные стратегии, которые помогут нам написать код, соответствующий этим целям. Эта книга будет посвящена шести таким стратегиям, которые я буду называть (в очень широком смысле) «шестью столпами качества кода». Мы начнем с общего описания каждого компонента, а в последующих главах будут приведены конкретные примеры, показывающие, как их применять в повседневном написании программ.

Шесть столпов качества кода:

1. Сделать код читабельным.
2. Избегать сюрпризов.
3. Сделать код трудным для неправильного использования.
4. Сделать код модульным.
5. Сделать код многократно и обобщаемым.
6. Сделать код тестируемым и протестировать его должным образом.

1.3.1. Сделать код читабельным

Рассмотрим следующий отрывок текста. Его намеренно сделали трудночитаемым, так что не тратьте слишком много времени на расшифровку. Бегло прочитайте и попробуйте понять все, что сможете:

Возьмите миску; теперь мы будем называть её А. Возьмите кастрюлю; а её теперь мы будем называть В. Наполните В водой и поставьте на плиту. Возьмите и положите в неё сливочное масло и шоколад, 100 граммов первого, 185 граммов второго. Шоколад должен быть темным с содержанием какао 70%. Поместите А поверх В; оставьте её там, пока содержимое А не растает, затем снимите А с В. Возьмите другую миску; теперь мы будем называть её С. Возьмите С и положите в неё яйца, сахар и ванильную эссенцию, 2 штуки первого, 185 граммов второго и половину чайной ложки третьего. Смешайте содержимое С. Как только содержимое А остынет, добавьте содержимое А в С и перемешайте. Возьмите ещё миску; её мы будем называть D. Возьмите D и положите в неё муку, какао-порошок и соль, 50 граммов первого, 35 граммов второго и половину чайной ложки третьего. Тщательно перемешайте содержимое D, а затем просейте в С. Смешайте содержимое D ровно настолько, чтобы полностью объединить компоненты. Кстати, мы готовим шоколадные пирожные; я что, забыл упомянуть об этом? Возьмите D и добавьте 70 граммов шоколадной стружки, смешайте содержимое D ровно настолько, чтобы объединить компоненты. Возьмите форму для выпечки; мы будем называть её Е. Смажьте Е и застелите бумагой для выпечки. Поместите содержимое D в Е. Мы будем называть вашу духовку F. Кстати, вы должны были предварительно разогреть F до 160 °C. Поместите Е в F на 20 минут, затем удалите Е из F. Дайте Е остыть в течение нескольких часов.

Теперь несколько вопросов. О чем этот отрывок текста? Что мы получим в итоге после выполнения всех инструкций? Какие ингредиенты и в каком количестве нам нужны?

Мы можем найти ответ на все эти вопросы в отрывке текста, но это не просто; текст плохо читается. Есть несколько проблем, которые делают текст менее читабельным, в том числе следующие:

- ◆ Нет заголовка, поэтому мы должны прочитать весь отрывок, чтобы понять, о чем он. Отрывок не представлен красиво в виде серии шагов (или подзадач); вместо этого он представлен в виде одной большой стены текста.
- ◆ Вещи упоминаются с бесполезно расплывчатыми названиями, такими, как «А» вместо «миска с растопленным маслом и шоколадом».
- ◆ Фрагменты информации размещаются далеко от того места, где они необходимы: ингредиенты и их количество разделены, и важная инструкция о том, что духовку нужно предварительно прогреть, упоминается только в конце.

(На случай, если вы не смогли дочитать отрывок, это рецепт шоколадных пирожных. В приложении А есть более читаемая версия на случай, если вы действительно захотите их сделать.)

Чтение фрагмента плохо написанного кода и попытка разобраться в нем не отличается от опыта, который мы только что получили при чтении рецепта брауни. В частности, нам может быть трудно понять следующие вещи о коде:

- ◆ что он делает;
- ◆ как он это делает;
- ◆ какие ингредиенты ему нужны (входные данные или состояние);
- ◆ что мы получим после запуска этого фрагмента кода.

В какой-то момент другому программисту, скорее всего, потребуется прочитать наш код и понять его. Если наш код должен пройти проверку перед отправкой, то это произойдет почти сразу. Но, даже игнорируя проверку кода, однажды кто-то обнаружит, что смотрит на наш код и пытается понять, что происходит. Это может случиться, когда меняются требования или код нуждается в отладке.

Если наш код плохо читаем, другим программистам придется потратить много времени на его расшифровку. Существует также высокая вероятность того, что они могут неправильно истолковать то, что наш код делает, или пропустить важные детали. Если это произойдет, то менее вероятно, что ошибки будут обнаружены во время проверки кода, и более вероятно, что будут допущены новые ошибки во время изменения кода при добавлении новой функциональности. Все, что делает часть программного обеспечения, происходит из-за кода, который заставляет это происходить. Если программисты не могут понять, что делает этот код, то становится почти невозможно убедиться, что программное обеспечение в целом будет выполнять свою работу должным образом. Как и в случае с рецептом: код должен быть читабельным.

В *главе 2* мы увидим, как определение правильных слоев абстракции может помочь в этом. А в *главе 5* мы рассмотрим ряд конкретных методов, позволяющих сделать код более читабельным.

1.3.2. Избегать сюрпризов

Получение подарка на день рождения или выигрыш в лотерею — два примера приятных сюрпризов. Однако когда мы пытаемся выполнить конкретную задачу, сюрпризы, как правило, плохая вещь.

Представьте, что вы голодны, поэтому решили заказать пиццу. Вы достаете свой телефон, находите номер пиццерии и набираете его. Линия молчит подозрительно долгое время, но в конце концов соединяется, и голос на другом конце спрашивает вас, чего вы хотите.

«Я хочу заказать одну большую маргариту, пожалуйста».

«Хорошо, какой у вас адрес?».

Через полчаса ваш заказ доставлен, вы заглядываете в пакет и обнаруживаете следующее (рис. 1.3).

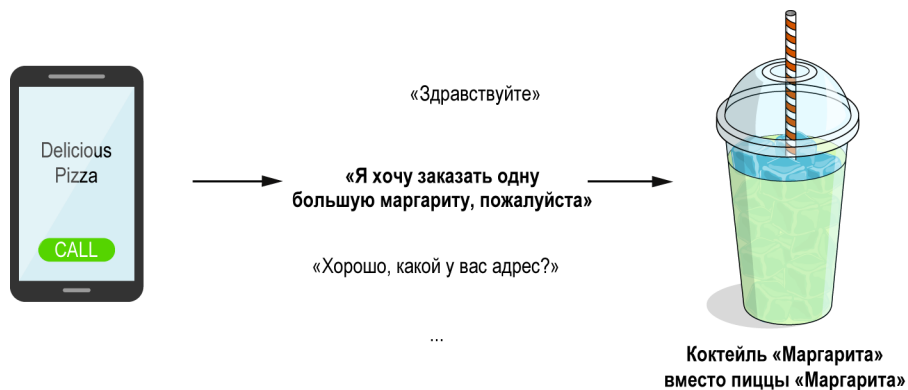


Рис. 1.3. Если вы думаете, что разговариваете с пиццерией, когда на самом деле вы разговариваете с мексиканским рестораном, ваш заказ все еще может иметь смысл, но вы получите сюрприз, когда он будет доставлен

Вот это сюрприз! Очевидно, кто-то перепутал «Маргариту» (разновидность пиццы) с «Маргаритой» (разновидностью коктейля), но это немного странно, потому что в пиццерии не подают коктейли.

Оказывается, пользовательское приложение для набора номера, которое вы используете на своем телефоне, добавило новую «умную» функцию. Разработчики приложения заметили, что, когда пользователи звонят в ресторан и обнаруживают, что линия занята, 80% из них немедленно звонят в другой ресторан, поэтому они создали удобную функцию экономии времени: когда вы звоните по номеру, который приложение распознает как ресторан, и линия занята, он просто набирает следующий номер ресторана в вашем телефоне вместо этого.

В данном случае это оказался ваш любимый мексиканский ресторан, а не пиццерия, в которую вы, как вам казалось, звонили. В мексиканском ресторане определенно подают коктейли «Маргарита», но не пиццу. Разработчики приложения имели благие намерения и думали, что облегчают жизнь пользователям, но они создали систему, которая преподносит сюрприз. Мы полагаемся на нашу ментальную модель телефонного звонка, чтобы определить, что происходит, основываясь на том, что мы слышим. Важно отметить, что если мы слышим голосовой ответ, то наша ментальная модель говорит нам, что мы были подключены к номеру, который мы набрали.

Новая функция в приложении набора номера ломает наши ожидания; это нарушает предположение нашей ментальной модели о том, что, если голос отвечает, мы подключены к набранному номеру. Это вполне может быть полезной функцией, но, поскольку ее поведение выходит за рамки ментальной модели нормального человека, приложение должно четко объяснять, что происходит. Например, можно добавить аудиосообщение, сообщающее нам, что номер, по которому мы звонили, занят, и спрашивающее нас, не хотим ли мы вместо этого подключиться к другому ресторану.

Приложение набора номера аналогично фрагменту кода. Другой программист, использующий наш код, будет использовать такие подсказки, как имена, типы дан-

ных и общие соглашения, чтобы построить ментальную модель того, что они ожидают от нашего кода в качестве входных данных, что он будет делать и что он будет возвращать. Если наш код делает что-то за пределами этой ментальной модели, то это очень часто может привести к появлению ошибок в программном обеспечении.

В примере со звонком в пиццерию казалось, что все работает, даже после того как произошло неожиданное: вы заказали маргариту, и ресторан был рад услужить. Только гораздо позже, когда уже было слишком поздно исправляться, вы обнаружили, что по неосторожности заказали коктейль вместо еды. Это аналогично тому, что часто происходит в программных системах, когда какой-то код делает что-то неожиданное: поскольку вызывающий код не знает, чего ожидать, он продолжает оставаться в неведении. Часто будет казаться, что какое-то время все в порядке, но позже все пойдет катастрофически неверно, когда программа окажется в недопустимом состоянии или пользователю будет возвращено странное значение.

Даже с самыми лучшими намерениями написание кода, который делает что-то полезное или умное, может привести к сюрпризам. Если код делает что-то неожиданное, то программист, использующий этот код, не будет знать или думать, как справиться с этим сценарием. Часто это приводит к тому, что система будет сбоить до тех пор, пока какое-то странное поведение не проявится вдали от рассматриваемого кода. Это может вызвать слегка раздражающую ошибку, но также может привести к катастрофической проблеме, которая приведет к повреждению важных данных. Мы должны остерегаться сюрпризов в коде и стараться по возможности избегать их.

В *главе 3* мы увидим, что контрактное программирование является фундаментальной техникой, которая может помочь в этом. В *главе 4* рассматриваются ошибки, которые могут стать причиной сюрпризов, если их не обнаруживать или не обрабатывать надлежащим образом. А в *главе 6* рассматривается ряд более конкретных приемов, позволяющих избежать сюрпризов.

1.3.3. Сделать код трудным для неправильного использования

Если мы посмотрим на заднюю панель телевизора, он, вероятно, будет выглядеть примерно так, как показано на рис. 1.4. В нем будет куча разных розеток, в которые мы сможем подключить кабели. Важно отметить, что розетки будут иметь разную форму; производитель телевизора сделал невозможным подключение шнура питания к разъему HDMI.



Рис. 1.4. Разъемы на задней панели телевизора намеренно имеют разную форму, чтобы затруднить подключение неправильных кабелей в неправильные отверстия

Представьте себе, если бы производитель так не делал, а вместо этого сделал бы каждую розетку одинаковой формы. Как вы думаете, сколько людей могут случайно подключить кабели не к тем розеткам, когда они возятся с задней панелью своего телевизора? Если бы кто-то подключил кабель HDMI к розетке питания, то ничего, вероятно, не сработало бы. Это было бы досадно, но не слишком катастрофично. Однако если бы кто-то подключил кабель питания к разъему HDMI, это могло бы буквально привести к взрыву.

Код, который мы пишем, часто вызывается другим кодом и немного похож на заднюю панель телевизора. Мы ожидаем, что другой код «подключит» определенные вещи, такие, как входные аргументы или перевод системы в определенное состояние перед вызовом. Если в наш код будут вставлены неправильные вещи, то все может взорваться; сбой системы, необратимое повреждение базы данных или потеря важных данных. Даже если ничего не взорвется, есть большая вероятность, что код не сработает. Была причина, по которой был вызван наш код, и неправильное подсоединение может означать, что важная задача не выполняется или программа ведет себя странно, хотя этого никто не замечает.

Мы можем максимально увеличить вероятность того, что код работает и продолжает работать, делая вещи трудными или невозможными для неправильного использования. Существует множество практических способов сделать это. Глава 3 посвящена соглашениям по написанию кода, которые (так же как с предотвращением сюрпризов) являются фундаментальным методом, затрудняющим неправильное использование кода. В главе 7 рассматривается ряд более конкретных методов, позволяющих затруднить неправильное использование кода.

1.3.4. Сделать код модульным

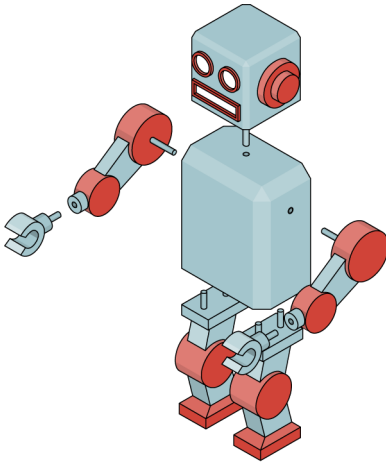
Модульность означает, что объект или система состоят из более мелких компонентов, которые можно независимо менять местами или заменить. Чтобы продемонстрировать это, а также преимущества модульности, рассмотрим две игрушки на рис. 1.5.

Игрушка слева является высокомодульной. Голова, руки, кисти и ноги можно легко и независимо менять местами или заменить, не затрагивая другие части игрушки. Игрушка справа, наоборот, очень немодульна. Не существует простого способа менять местами или заменить голову, руки, кисти или ноги.

Одной из ключевых особенностей модульной системы (такой, как игрушка слева) является то, что различные компоненты имеют четко определенные интерфейсы с как можно меньшим количеством точек взаимодействия. Если рассматривать руку как компонент, то с игрушкой слева есть одна точка взаимодействия с простым интерфейсом: один колышек и одно отверстие, в которое он помещается. Игрушка справа имеет невероятно сложный интерфейс между рукой и остальной частью игрушки: 20 с лишним петель нити на руке, и кисти переплетены друг с другом.

Теперь представьте, что наша работа заключается в обслуживании этих игрушек, и однажды наш менеджер говорит нам, что существует новое требование, согласно

которому руки теперь должны иметь пальцы. С какой игрушкой / системой мы бы предпочли работать?



Модульная игрушка



Немодульная игрушка

Рис. 1.5. Модульную игрушку можно легко реконфигурировать.
Игрушку, которая была сшита вместе, реконфигурировать чрезвычайно трудно

С игрушкой слева мы могли бы изготовить новый дизайн руки, а затем очень легко внедрить его вместо существующего. Если бы наш менеджер затем передумал через две недели, у нас не возникло бы проблем с возвращением игрушки в прежнюю конфигурацию.

С игрушкой справа нам, вероятно, придется достать ножницы, разрезать 20 с лишним нитей, а затем пришить новые руки прямо к игрушке. Скорее всего, мы повредим игрушку в процессе, и, если наш менеджер передумает через две недели, мы столкнемся с таким же трудоемким процессом возврата игрушки к предыдущей конфигурации.

Программные системы и кодовые базы во многом аналогичны этим игрушкам. Часто бывает полезно разбить фрагмент кода на автономные модули, где взаимодействие между двумя соседними модулями происходит в одном месте и использует четко определенный интерфейс. Это помогает гарантировать, что код будет легче адаптироваться к изменяющимся требованиям, поскольку изменения части функций не требуют большого количества изменений повсюду.

Модульные системы также, как правило, легче понять и осмыслить, потому что функциональность разбита на управляемые блоки, а взаимодействия между блоками функциональности четко определены и документированы. Это увеличивает вероятность того, что код будет работать сейчас и продолжит работать в будущем, потому что меньше вероятность того, что программисты неправильно поймут, что делает код.

В главе 2 мы увидим, как создание чистых слоев абстракции является фундаментальным методом, который может привести нас к более модульному коду. А в главе 8 мы рассмотрим ряд конкретных методов, позволяющих сделать код более модульным.

1.3.5. Сделать код многократно и обобщаемым

Возможность повторного использования и обобщаемость — это две схожие, но немного разные концепции:

- ◆ *Возможность повторного использования* означает, что что-то может быть использовано для решения одной и той же задачи, но в нескольких сценариях. Ручная дрель многократно, потому что ее можно использовать для сверления отверстий в стенах, в досках пола и в потолках. Задача та же (нужно просверлить отверстие), но сценарий другой (сверление в стене, а не в полу или в потолке).
- ◆ *Обобщаемость* означает, что что-то может быть использовано для решения нескольких концептуально схожих проблем, которые немного отличаются. Ручная дрель также поддается обобщению, потому что ее можно использовать не только для сверления отверстий, но и для закручивания шурупов. Производитель дрели осознает, что вращение чего-либо является общей задачей, которая относится как к сверлению отверстий, так и к закручиванию шурупов, поэтому они создали инструмент, который обобщает, решая обе задачи.

В случае с дрелью мы можем сразу же признать преимущества вышесказанного. Представьте, если бы нам понадобились четыре разных инструмента:

- ◆ Дрель, которая работает только при удержании вертикально, что означает, что она пригодится только для сверления стен.
- ◆ Дрель, которая работает только тогда, когда она направлена вниз под углом 90°, что делает ее полезной только для сверления в полу.
- ◆ Дрель, которая работает только тогда, когда она направлена вверх под углом 90°, что позволяет сверлить только в потолке.
- ◆ Электрическая отвертка для ввинчивания шурупов в предметы.

Мы потратили бы гораздо больше денег на приобретение этой коллекции из четырех инструментов, нам пришлось бы носить с собой больше вещей и нам пришлось бы заряжать в четыре раза больше аккумуляторов — это настоящее расточительство. К счастью, кто-то создал дрель, которую можно использовать повторно и обобщать, и теперь нам нужен только один инструмент для выполнения всех этих работ. Как вы уже, несомненно, догадались, ручная дрель здесь — это еще одна аналогия для кода.

Создание кода требует времени и усилий, а после его создания также требуется постоянное время и усилия для поддержки. Создание кода также сопряжено с рисками: как бы мы ни были осторожны, некоторая часть кода, который мы пи-

шем, будет содержать ошибки, и чем больше кода мы напишем, тем больше у нас будет ошибок. Суть здесь в том, что чем меньше строк кода у нас в кодовой базе, тем лучше. Возможно, мои слова покажутся вам странными, ведь написание кода — это наша работа. Но на самом деле нам платят за решение задачи, а код — это всего лишь средство для достижения нашей цели. Отлично, если мы сможем решить эту задачу, прилагая меньше усилий, а также уменьшим вероятность того, что мы непреднамеренно создадим новые задачи, внося ошибки.

Создание многократно используемого и обобщаемого кода позволяет нам (и другим) использовать его в нескольких местах по всей кодовой базе, в более чем одном сценарии и для решения более чем одной задачи. Это экономит время и усилия и делает наш код более надежным, потому что мы часто будем повторно использовать логику, которая уже была опробована и протестирована в «дикой природе», а это означает, что любые ошибки, скорее всего, уже обнаружены и исправлены.

Код, который является более модульным, также имеет тенденцию быть более многократно используемым и обобщаемым. Главы, касающиеся модульности, перекликаются с темой возможности повторного использования и генерализации. Кроме того, *глава 9* охватывает ряд методов и соображений, специфичных для того, чтобы сделать код более многократно используемым и обобщаемым.

1.3.6. Сделать код тестируемым и протестировать его должным образом

Как мы видели ранее на схеме процесса разработки и развертывания программного обеспечения (рис. 1.2), тесты являются жизненно важной частью процесса обеспечения того, чтобы ошибки и сбои в работе не запускались в «дикой природе». Они часто являются основной защитой в двух ключевых моментах процесса (рис. 1.6):

- ◆ предотвращение отправки кода с ошибками и сбоями в базу кода;
- ◆ обеспечение того, чтобы релиз с ошибками или сбоями был заблокирован и не попал в «дикую природу».

Вот почему тесты являются важной частью обеспечения того, чтобы код работал и продолжал работать.

Трудно переоценить, насколько важно тестирование в разработке программного обеспечения. Вы, без сомнения, слышали это и раньше, и вам легко отмахнуться от этого как от очередной банальности. Однако тесты действительно важны. Как мы увидим еще не раз далее в книге, программные системы и кодовые базы, как правило, слишком велики и сложны, чтобы один человек мог знать о них каждую мельчайшую деталь, а люди (даже исключительно умные программисты) совершают ошибки.

Это более или менее правда жизни, и если мы не проверяем функциональность нашего кода с помощью тестов, то эта правда жизни сработает против нас (и нашего кода).



Рис. 1.6. Тесты жизненно важны для минимизации попадания ошибок и сбоев в работе в кодовую базу, а также, если это произойдет, для предотвращения попадания ошибок в «дикую природу»

Название этого столпа качества кода содержит две важные концепции: «сделать код тестируемым» и «протестировать его должным образом». Тестирование и тестируемость связаны, но несут в себе разные соображения.

Тестирование — как следует из названия, относится к тестированию нашего кода или программного обеспечения в целом. Тестирование может быть ручным или автоматизированным. Как инженеры, мы обычно стремимся автоматизировать наше тестирование, написав тестовый код, который использует «реальный» код и проверяет, все ли работает так, как должно. Существуют различные уровни тестирования. Три наиболее распространенных, с которыми вы, вероятно, будете работать, заключаются в следующем. (Пожалуйста, обратите внимание, что это не исчерпывающий список; существует множество способов классификации тестов, и разные организации часто используют разные номенклатуры.)

- ◆ *Модульные тесты* — они обычно тестируют небольшие блоки кода, такие, как отдельные функции или классы. Модульное тестирование — это уровень, с которым инженеры по тестированию чаще всего работают в своем повседневном программировании. Это единственный уровень тестирования, который будет подробно описан в этой книге.
- ◆ *Интеграционные тесты* — система обычно состоит из нескольких компонентов, модулей или подсистем. Процесс соединения этих компонентов и подсистем вместе известен как *интеграция*. Интеграционные тесты призваны гарантировать, что эти интеграции работают и продолжают работать.
- ◆ *Сквозные тесты (End-to-end tests, E2E)* — эти тесты проверяют типичные маршруты (или потоки работ) через всю программную систему от начала до конца. Если бы рассматриваемое программное обеспечение было интернет-магазином, то примером теста E2E мог бы быть тест, который автоматически управляет веб-браузером, чтобы гарантировать, что пользователь может пройти весь процесс завершения покупки.

Тестируемость — относится к «реальному» коду (в отличие от тестового кода) и описывает, насколько хорошо этот код поддается тестированию. Концепция того,

что что-то поддается проверке, может также применяться на уровне подсистемы или системы. Тестируемость часто тесно связана с модульностью, при этом более модульный код (или системы) лучше поддается тестированию. Представьте, что производитель автомобилей разрабатывает систему экстренного торможения перед пешеходами. Если система не очень модульная, то единственным способом протестировать ее будет установить систему в реальном автомобиле, поехать в нем по направлению к реальному пешеходу и проверить, сможет ли автомобиль автоматически притормозить. Если это так, то количество сценариев, в которых система может быть протестирована, ограничено, потому что стоимость каждого теста очень высока: нужно создать целый автомобиль, арендовать настоящую трассу и подвергнуть риску живого человека, играющего роль пешехода на дороге. Система экстренного торможения становится намного более проверяемой, если это отдельный модуль, который можно запускать вне реального автомобиля. Теперь его можно протестировать, отправив предварительно записанное видео, на котором пешеход выходит на дорогу, а затем проверив, что система выдает правильный сигнал, предназначенный для тормозной системы. Теперь можно очень легко, дешево и безопасно протестировать многие тысячи различных дорожных сценариев.

Если код не поддается тестированию, то его правильное тестирование может стать невозможным. Чтобы убедиться, что код, который мы пишем, поддается тестированию, полезно постоянно спрашивать себя: «Как мы будем это тестировать?», когда мы пишем код. Поэтому тестирование не следует рассматривать как второстепенную мысль: это неотъемлемая и фундаментальная часть написания кода на всех этапах. *Главы 10 и 11* полностью посвящены тестированию, но поскольку тестирование является неотъемлемой частью написания кода, оно встречается и во многих других местах этой книги.

РАЗРАБОТКА НА ОСНОВЕ ТЕСТОВ

Поскольку тестирование является неотъемлемой частью написания кода, некоторые инженеры выступают за то, чтобы тесты были написаны до кода. Это одна из практик, которую отстаивает процесс *разработки на основе тестирования* (*test-driven development, TDD*). Мы обсудим это подробнее в *главе 10 (разд. 10.5)*.

Тестирование программного обеспечения — это огромная тема, и, скажу честно, эта книга даже не приблизится к тому, чтобы отдать ей должное. В этой книге мы рассмотрим некоторые из наиболее важных и часто упускаемых из виду аспектов модульного тестирования кода, поскольку они обычно наиболее полезны в ходе повседневного программирования. Но, пожалуйста, имейте в виду, что в нашей книге мы лишь прикоснемся к тому, что нужно знать о тестировании программного обеспечения.

1.4. Замедляет ли нас написание высококачественного кода?

Ответ на этот вопрос заключается в том, что в очень краткосрочной перспективе может показаться, что написание высококачественного кода замедлит нас. Написа-

ние высококачественного кода обычно требует немного больше размышлений и усилий, чем просто написание первого, что приходит нам в голову. Но если мы пишем что-то более существенное, чем небольшая утилита, которую можно запустить один раз, а затем выбросить, то написание высококачественного кода обычно ускорит время разработки в среднесрочной и долгосрочной перспективе.

Представьте, что мы вешаем полку дома. Существует «правильный» способ сделать это, а еще есть быстрый, «хакерский» способ:

- ◆ *Правильный способ* — мы прикрепляем кронштейны к стене, сверля и ввинчивая их во что-то прочное, например, в стенные заклепки или каменную кладку. Затем мы устанавливаем полку на эти кронштейны. Затраченное время: 30 минут.
- ◆ *Хитрый хакерский способ* — мы покупаем немного клея и приклеиваем полку к стене. Затраченное время: 10 минут.

Похоже, что хитрый способ установки полки может сэкономить нам 20 минут, а также сэкономить нам усилия по извлечению дрели и отвертки из ящика. Мы выбрали быстрый подход; теперь давайте рассмотрим, что произойдет дальше.

Мы приклеили полку к тому, чем покрыта стена; это, скорее всего, слой штукатурки. Штукатурка непрочная и может легко треснуть и отвалиться большими кусками. Как только мы начнем использовать полку, вес предметов на ней, скорее всего, приведет к тому, что штукатурка треснет, и полка упадет и захватит с собой большой кусок штукатурки. Теперь у нас нет нормальной полки, а также нам нужно отремонтировать и заштукатурить стену (работа, которая займет несколько часов, если не дней). Даже если каким-то чудом полка не упадет, мы создали себе проблемы в будущем, быстро установив ее. Представьте себе еще пару сценариев.

1. Мы поняли, что установили полку недостаточно высоко (ошибка):

- Для полки на кронштейнах мы можем просто установить прокладку между кронштейном и полкой. Затраченное время: 5 минут.
- Для приклеенной полки нам нужно оторвать ее от стены; тогда вместе с ней уйдет большой кусок штукатурки. Теперь нам нужно заново покрасить стену, а затем снова установить полку. Затраченное время: несколько часов, если не дней.

2. Мы решили отремонтировать комнату (новое требование):

- Мы можем снять полку на кронштейнах, выкрутив саморезы. Далее мы делаем ремонт в комнате, а затем возвращаем полку на место. Время, затраченное на работу, связанную с полкой: 15 минут.
- Для приклеенной полки мы либо оставляем полку на месте, и тогда нам придется аккуратно красить или оклеивать обоями стену вокруг полки, рискуя капнуть на полку краской или клеем и сделать ее неопрятной. Или же мы отрываем полку от стены и смиряемся с тем фактом, что нам придется всё переклеивать. Мы можем либо сделать некачественный ремонт, либо потратить несколько часов или дней на то, чтобы опять делать все заново.

Вы поняли идею. Поначалу могло показаться, что все сделано правильно, и установка полки на кронштейнах была ненужной тратой 20 минут, но в долгосрочной перспективе вполне вероятно, что это сэкономит нам много времени и хлопот. В случае будущего проекта по ремонту мы также увидели, как, начав с быстрого, хакерского решения, мы затем идем по пути выполнения большего количества вещей хакерским способом, например, пытаемся покрасить или оклеить обоями стену вокруг полки, вместо того чтобы снять полку на время ремонта.

Программирование очень схоже с этим примером. Написание первого, что приходит нам в голову, без учета качества кода, скорее всего, сэкономит какое-то время на начальном этапе. Но мы быстро получим хрупкую, сложную кодовую базу, которую становится все труднее понимать или обдумывать. Добавление новых функций или исправление ошибок будет становиться все более сложным и медленным, поскольку нам придется иметь дело с поломками и новыми требованиями.

Уверен, вы уже слышали фразу «тише едешь — дальше будешь»: и действительно, во многих жизненных ситуациях слишком поспешные действия без продумывания или правильного выполнения часто приводят к ошибкам, которые снижают нашу общую скорость и качество. Итак, написание высококачественного кода ускоряет нашу работу; не путайте спешку со скоростью.

Итоги

- ◆ Чтобы создать хорошее программное обеспечение, нам нужно написать высококачественный код.
- ◆ Прежде чем код станет программным обеспечением, работающим в «дикой природе», он должен пройти несколько этапов проверок и тестов (иногда вручную, иногда автоматически).
- ◆ Эти проверки помогают удостовериться, что ошибки и сбои в работе не достигли пользователей или важных для бизнеса систем.
- ◆ Полезно помнить о тестировании на каждом этапе написания кода; его не следует рассматривать как нечто необязательное.
- ◆ Может показаться, что написание высококачественного кода замедляет нас поначалу, но это часто ускоряет время разработки в среднесрочной и долгосрочной перспективе.

2 Слои абстракции

ЭТА ГЛАВА ОХВАТЫВАЕТ

- ◇ Как разбить проблему на подзадачи с помощью чистых слоев абстракции
- ◇ Как слои абстракции могут помочь улучшить качество кода
- ◇ API и детали реализации
- ◇ Как разбить код на отдельные слои абстракции с помощью функций, классов и интерфейсов

Написание кода — это решение проблем. Проблем высокого уровня, таких, как «нам нужна функция, позволяющая пользователям обмениваться фотографиями», и проблем низкого уровня, таких, как «нам нужен код для сложения двух чисел». Даже если мы не осознаем того факта, что мы это делаем, когда решаем проблему высокого уровня, мы обычно делаем это, разбивая ее на несколько более мелких подзадач. Формулировка проблемы, такая, как «нам нужна система, позволяющая пользователям обмениваться фотографиями», может означать, что нам необходимо решить такие подзадачи, как хранение фотографий, их привязка к пользователям и их отображение.

Важно то, как мы решаем проблемы и подзадачи, но часто не менее важно то, как мы структурируем код, который их решает. Например, должны ли мы просто свалить все в одну гигантскую функцию или класс, или мы должны попытаться разбить ее на несколько функций или классов? И если да, то как мы должны это сделать?

То, как мы структурируем код, является одним из наиболее фундаментальных аспектов качества кода, и выполнение его часто сводится к созданию чистых слоев абстракции. В этой главе мы объясним, что это значит, и продемонстрируем, как разбивка проблем на отдельные *слои абстракции* и структурирование кода для отражения этого могут значительно улучшить его читаемость, модульность, возможность повторного использования, обобщаемость и тестируемость.

В этой и следующих главах приведено множество примеров псевдокода для демонстрации обсуждаемых тем. Прежде чем углубляться в эти примеры, важно потратить несколько минут на объяснение того, как соглашение псевдокода в этой книге обрабатывает нулевые значения. *Раздел 2.1* будет посвящен этому, в то время как *разд. 2.2* и далее будут охватывать основные темы этой главы.

2.1. Нулевые значения и соглашение псевдокода в этой книге

Прежде чем мы перейдем к рассмотрению примеров кодирования, важно объяснить, как соглашение псевдокода в этой книге обрабатывает нулевые значения.

Во многих языках программирования отсутствует понятие значения (или ссылки/указателя); встроенный способ сделать это часто имеет значение `null`. Исторически нулевые значения находились в дихотомии между тем, чтобы быть невероятно полезными и невероятно проблематичными.

- ◆ Они полезны, потому что очень часто возникает концепция отсутствия чего-либо: значение не было предоставлено или функция не может обеспечить желаемый результат.
- ◆ Они проблематичны, потому что не всегда очевидно, когда значение может/не может быть нулевым, и инженеры очень часто забывают проверить, является ли переменная нулевой, прежде чем обращаться к ней. Это очень часто приводит к ошибкам; вы, вероятно, ранее видели `NullPointerException`, `NullReferenceException` или `Cannot read property of null` больше раз, чем можете вспомнить.

Из-за того, насколько проблематичными могут быть нули, вы иногда увидите советы, которые рекомендуют никогда не использовать их или, по крайней мере, никогда не возвращать их из функции. Это, безусловно, помогает избежать проблем с нулевыми значениями, но может потребоваться много упражнений с кодом, чтобы следовать этому совету на практике.

К счастью, в последние годы идея *нулевой безопасности* — *null safety* (также называемая *безопасностью пустоты* — *void safety*) получила все большее распространение. Это гарантирует, что любые переменные или возвращаемые значения, которые могут быть нулевыми, помечены как таковые, и компилятор гарантирует, что они не будут использоваться без предварительной проверки того, что они не являются нулевыми.

Большинство важных новых языков, появившихся в последние годы, поддерживают нулевую безопасность. Она также может быть дополнительно включена в более поздние версии языков, таких, как C#, и даже существуют способы ее модификации в такие языки, как Java. Если язык, который мы используем, поддерживает нулевую безопасность, то, вероятно, его использование — хорошая идея.

Если язык, который мы используем, не поддерживает безопасность нулей, то хорошей альтернативой является использование опционального типа. Многие языки поддерживают это, включая Java, Rust (называемый `Option`) и C++ (хотя в C++ есть нюансы, которые следует учитывать, они описаны в приложении В). Даже в языках, которые не поддерживают безопасность нулей в качестве стандартной функции, часто существуют сторонние утилиты, которые добавляют его поддержку.

Соглашение псевдокода в этой книге предполагает, что нулевая безопасность существует. По умолчанию переменные, параметры функций и типы возвращаемых

значений не могут быть обнулены. Но если тип имеет суффикс «?», то это означает, что он может быть нулевым, и компилятор будет следить за тем, чтобы он не использовался без предварительной проверки того, что он не равен нулю. Следующий фрагмент демонстрирует, как выглядит некоторый псевдокод, использующий нулевую безопасность:

```
Element? getFifthElement(List<Element> elements) { ❶
    if (elements.size() < 5) {
        return null; ❷
    }
    return elements[4];
}
```

Если язык, который мы используем, не поддерживает нулевую безопасность и мы хотим написать эту функцию с использованием опционального типа, то следующий фрагмент демонстрирует, как код, который мы только что видели, может быть переписан:

```
Optional<Element> getFifthElement(List<Element> elements) { ❸
    if (elements.size() < 5) {
        return Optional.empty(); ❹
    }
    return Optional.of(elements[4]);
}
```

❶ «?» в `Element?` указывает, что тип возвращаемого значения может быть нулевым.

❷ Значение `null` возвращается, если значение не может быть получено.

❸ Тип возвращаемого значения является опциональным элементом.

❹ `Optional.empty()` возвращается вместо `null`.

Если вы хотите узнать больше о нулевой безопасности и дополнительных типах, в приложении В содержится дополнительная информация.

2.2. Зачем создавать слои абстракции?

Написание кода часто сводится к тому, чтобы взять сложную проблему и постоянно разбивать ее на более мелкие подзадачи. Чтобы продемонстрировать это, представьте, что мы пишем фрагмент кода для запуска на устройстве пользователя и хотим отправить сообщение на сервер. Мы, вероятно, надеемся, что сможем написать что-то вроде кода в листинге 2.1. Обратите внимание, насколько прост код; он состоит из трех строк и требует работы только с четырьмя простыми концепциями:

- ◆ URL-адрес сервера;
- ◆ соединение;
- ◆ отправка строки сообщения;
- ◆ закрытие соединения.

Листинг 2.1. Отправка сообщения на сервер

```
HttpConnection connection =  
    HttpConnection.connect("http://example.com/server");  
connection.send("Hello server");  
connection.close();
```

На высоком уровне это кажется довольно простой проблемой, и, действительно, решение ее выглядит довольно простым. Но это, очевидно, непростая проблема: существует огромная сложность, связанная с отправкой строки «Hello server» с клиентского устройства на сервер, включающая следующее:

- ◆ сериализация строки в формат, который может быть передан;
- ◆ все тонкости протокола HTTP;
- ◆ соединения TCP;
- ◆ независимо от того, находится ли пользователь в сети Wi-Fi или сотовой сети;
- ◆ модулирование данных в радиосигнал;
- ◆ ошибки передачи данных и их исправление.

В этом примере существует проблема высокого уровня, которая нас волнует: отправка сообщения на сервер. Но для этого необходимо решить множество подзадач (например, все только что перечисленные). К счастью для нас, другие инженеры уже решили все эти подзадачи, причем таким образом, что нам даже не нужно знать о них.

Мы можем думать о решениях проблем и подзадач как о формировании ряда слоев. На самом верхнем уровне нас волнует отправка сообщения на сервер, и мы можем написать код для этого, не зная ничего о том, как реализован протокол HTTP. Аналогично, инженеру, написавшему код для реализации протокола HTTP, вероятно, не нужно было ничего знать о том, как данные модулируются в радиосигнал. Инженер, который реализовал код `HttpConnection`, смог представить физическую передачу данных как абстрактную концепцию, и, в свою очередь, мы можем представить HTTP-соединение как абстрактную концепцию. Она известна как *слои абстракции*. На рис. 2.1 показаны некоторые слои абстракции, участвующие в отправке сообщения на сервер.

Стоит потратить еще немного времени, чтобы оценить, насколько прост код для отправки сообщения на сервер по сравнению с огромной сложностью, связанной с его фактическим выполнением.

- ◆ Для этого потребовалось три строки простого кода.
- ◆ Это касалось только четырех простых концепций:
 - URL-адрес сервера;
 - связь;
 - отправка строки сообщения;
 - закрытие соединения.

Задача в общем виде

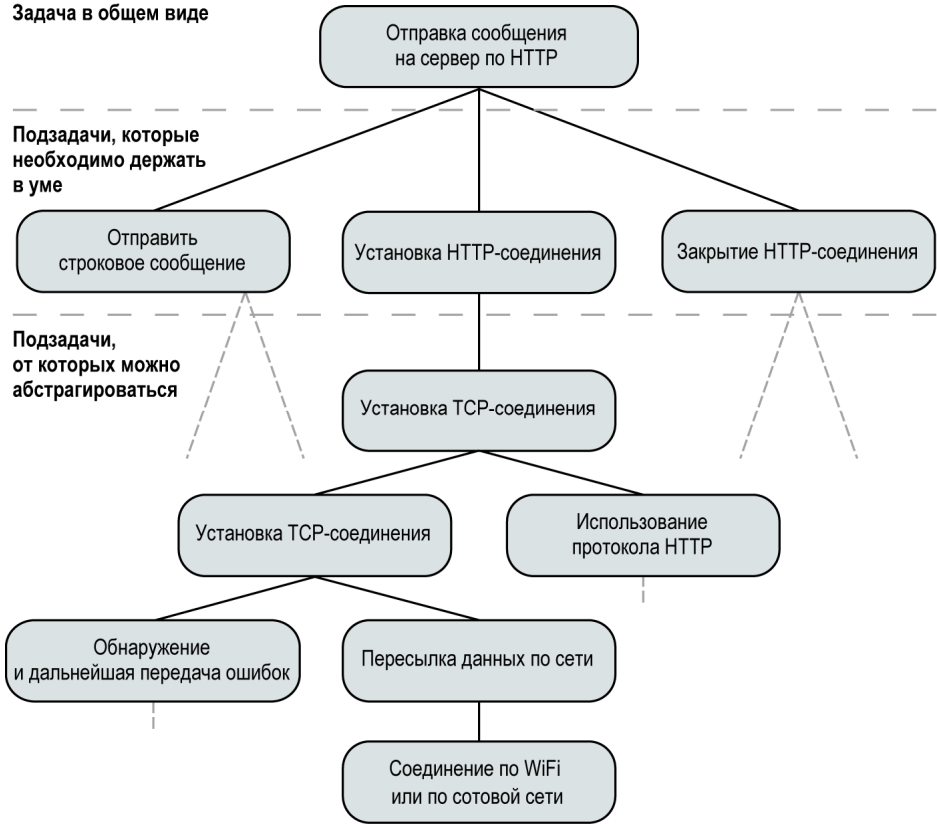
Подзадачи, которые
необходимо держать
в умеПодзадачи,
от которых можно
абстрагироваться

Рис. 2.1. При отправке сообщения на сервер мы можем повторно использовать решения для подзадач, которые уже создали другие. Чистые слои абстракции также означают, что нам нужно знать только несколько концепций, чтобы решить проблему высокого уровня

В более общем плане, если мы проделаем хорошую работу по рекурсивному разбиению проблемы на подзадачи и созданию слоев абстракции, то ни один отдельный фрагмент кода никогда не будет казаться особенно сложным, потому что он будет иметь дело всего с несколькими легко понятными концепциями одновременно. Это должно быть нашей целью при решении проблемы как программиста: даже если проблема невероятно сложна, мы можем справиться с ней, определив подзадачи и создав правильные уровни абстракции.

Слои абстракции и основы качества кода

Построение чистых и четких слоев абстракции имеет большое значение для достижения четырех столпов качества кода, которые мы видели в *главе 1*. В следующих подразделах объясняется почему.

Читабельность

Инженеры не могут понять каждую деталь каждого фрагмента кода в кодовой базе, но им довольно легко понять и использовать несколько абстракций высокого уровня одновременно. Создание чистых и четких слоев абстракции означает, что инженерам нужно иметь дело только с одним или двумя слоями и несколькими концепциями одновременно. Это значительно повышает читабельность кода.

Модульность

Когда уровни абстракции четко разделяют решения на подзадачи и гарантируют, что между ними не будет утечки деталей реализации, тогда становится очень легко поменять местами реализации внутри слоя, не затрагивая другие слои или части кода. В примере `HttpConnection` часть системы, которая обрабатывает физическую передачу данных, скорее всего, будет модульной. Если пользователь подключен к Wi-Fi, то будет использоваться один модуль; если пользователь подключен к сотовой сети, то будет использоваться другой модуль. Нам не нужно делать ничего особенного в нашем коде более высокого уровня, чтобы учесть эти различные сценарии.

Возможность повторного использования и обобщаемость

Если решение подзадачи представлено в виде чистого слоя абстракции, то легко повторно использовать только решение этой подзадачи. И если проблемы разбиты на соответствующие абстрактные подзадачи, то вполне вероятно, что решения будут обобщены, чтобы быть полезными в нескольких различных сценариях. В примере `HttpConnection` большинство частей системы, которые обрабатывают TCP/IP и сетевые подключения, вероятно, также могут быть использованы для решения подзадач, необходимых для других типов соединений, таких, как `WebSockets`.

Тестируемость

Если бы вы покупали дом и хотели убедиться, что он конструктивно надежен, вы бы не просто посмотрели снаружи и сказали: «Да, похоже на дом. Я куплю это». Вы бы хотели, чтобы геодезист проверил, не проседает ли фундамент, не треснули ли стены и не прогнили ли какие-либо деревянные конструкции. Аналогично, если нам нужен надежный код, то нам также необходимо убедиться, что решение каждой подзадачи является надежным и работающим. Если код четко разбит на слои абстракции, то становится намного проще полностью протестировать решение каждой подзадачи.

2.3. Слои кода

На практике мы создаем уровни абстракции, разделяя код на разные блоки, где один блок зависит от другого, создавая график зависимостей (рис. 2.2). В большинстве языков программирования в нашем распоряжении имеется несколько конструкций для разбиения кода на разные блоки. Чаще всего у нас будет следующее:

- ◆ функции;
- ◆ классы (и потенциально другие классоподобные вещи, такие, как структуры и смешивание);
- ◆ интерфейсы (или эквивалентная конструкция);
- ◆ пакеты, пространства имен или модули.

Я упоминаю об этом для полноты картины, но на самом деле мы не будем рассматривать их в этой книге, поскольку эти более высокие уровни структурирования кода часто диктуются в значительной степени организационными и системными соображениями, ни одно из которых не входит в рамки этой книги.

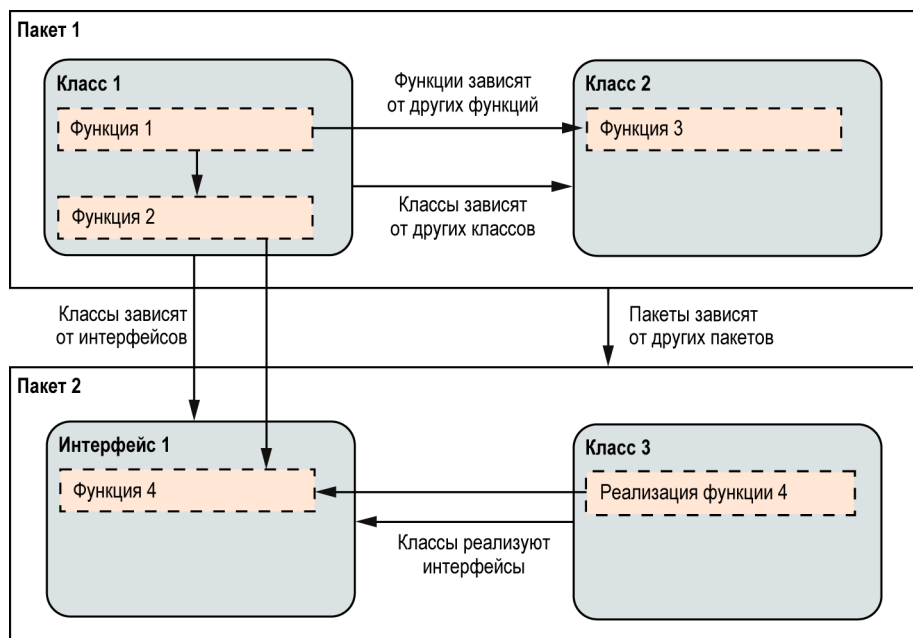


Рис. 2.2. Единицы кода зависят от других единиц кода, образуя график зависимостей

В следующих нескольких разделах будет рассмотрено, как лучше всего разбить код на чистые слои абстракции с помощью функций, классов и интерфейсов.

2.3.1. API и детали реализации

При написании кода часто необходимо учитывать два аспекта.

◆ То, что увидят вызывающие пользователи нашего кода:

- какие классы, интерфейсы и функции мы публикуем;
- какие понятия составляющие раскрывают в своих именах, входных параметрах и типах возвращаемых данных;
- любая дополнительная информация, которую вызывающим необходимо знать для правильного использования кода (например, порядок, в котором нужно совершать вызовы).

◆ То, что не увидят вызывающие пользователи нашего кода: детали реализации.

Если вы когда-либо работали со службами (создавали их или вызывали), то, скорее всего, вам знаком термин *программный интерфейс приложения* (application programming interface, API). Это формализует концепцию вещей, которые должны знать вызывающие службы, и все детали реализации службы остаются скрытыми за API.

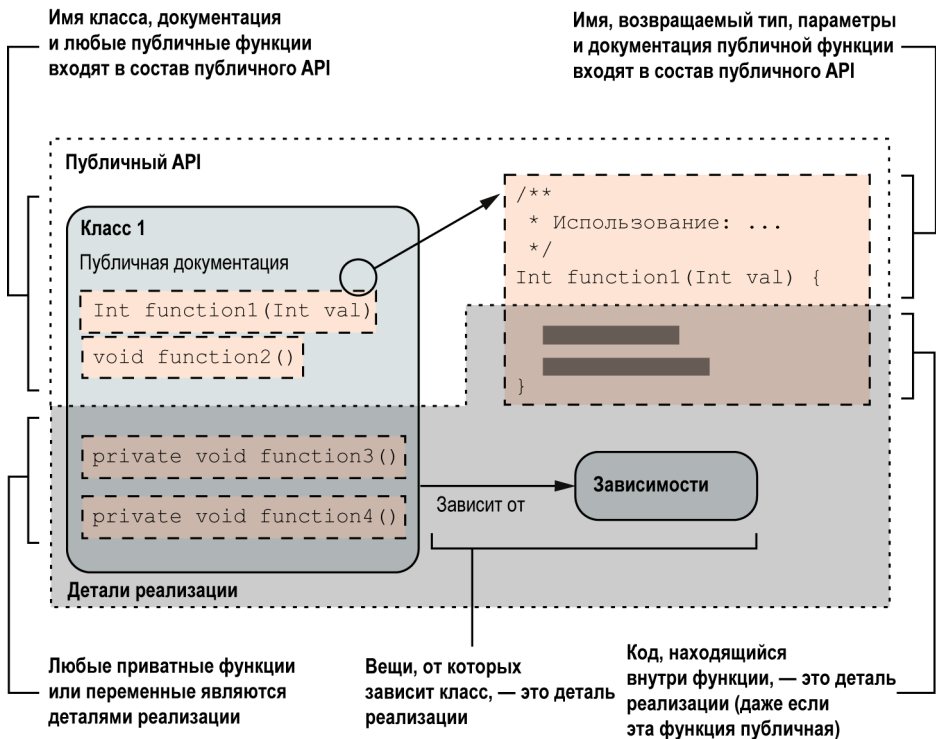


Рис. 2.3. Мы можем думать о тех частях нашего кода, о которых вызывающие пользователи должны знать как о предоставлении общедоступного API. Все, что не представлено в общедоступном API, является деталью реализации

Часто бывает полезно думать о коде, который мы пишем, как о мини-API, который могут использовать другие части кода. Инженеры часто делают это, и вы, вероятно, услышите, как они говорят о классах, интерфейсах и функциях как о «представлении API». На рис. 2.3 приведен пример того, как различные аспекты класса могут быть разделены между тем, чтобы быть частью общедоступного API и деталями реализации.

Размышление о коде в терминах API может помочь создать чистые слои абстракции, потому что API определяет, какие концепции доступны вызывающим, а все остальное — это детали реализации. Если мы пишем или изменяем какой-то код и что-то, что должно быть деталью реализации, просачивается в API (через входной параметр, возвращаемый тип или общедоступную функцию), то очевидно, что слои абстракции не так чисты и различны, как, вероятно, должны быть.

Мы будем использовать эту концепцию кода, раскрывающего API в нескольких местах этой книги, поскольку это полезный и лаконичный способ обратиться к уровню абстракции, предоставляемому фрагментом кода.

2.3.2. Функции

Порог, при котором полезно внедрить некоторую логику в новую функцию, часто бывает довольно низким. Код внутри каждой функции в идеале должен читаться как одно короткое, хорошо написанное предложение. Чтобы продемонстрировать это, рассмотрим пример функции, которая пытается сделать слишком много (листинг 2.2). Функция находит адрес владельца транспортного средства, а затем отправляет ему письмо, если таковой найден. Функция содержит всю основную логику для поиска адреса владельца, а также вызов функции для отправки письма. Это затрудняет понимание, так как в нем рассматривается слишком много понятий одновременно. В листинге 2.2 мы также можем увидеть, как излишек работы внутри одной функции может привести к другим проблемам, которые затрудняют понимание кода, таким, как глубоко вложенные утверждения `if` (в главе 5 это рассмотрено более подробно).

Листинг 2.2. Функция, которая делает слишком много

```
SentConfirmation? sendOwnerALetter(  
    Vehicle vehicle, Letter letter) {  
    Address? ownersAddress = null;  
    if (vehicle.hasBeenScraped()) {  
        ownersAddress = SCRAPYARD_ADDRESS;  
    } else {  
        Purchase? mostRecentPurchase =  
            vehicle.getMostRecentPurchase();  
        if (mostRecentPurchase == null) {  
            ownersAddress = SHOWROOM_ADDRESS;  
        }  
    }  
}
```

```
    } else {  
        ownersAddress =  
            mostRecentPurchase.getBuyersAddress();  
    }  
}  
if (ownersAddress == null) { ❶  
    return null;  
}  
return sendLetter(ownersAddress, letter);  
}
```

❶ Логика условной отправки письма.

Если бы функция `sendOwnerALetter()` была переведена в предложение, она гласила бы: «Найдите адрес владельца (который будет адресом свалки, если транспортное средство было сдано на утилизацию, или адресом выставочного зала, если транспортное средство еще не продано, или адресом зарегистрированного покупателя, если таковой имеется), и, если он найден, отправьте ему письмо». Это не очень приятное предложение: оно требует одновременного рассмотрения нескольких разных концепций, а сам объем слов означает, что нам, вероятно, придется сделать несколько попыток прочесть его, чтобы убедиться, что мы правильно его поняли.

Было бы намного лучше, если бы у нас была функция, которая переводилась бы примерно так: «Найдите адрес владельца (подробнее ниже), и, если он найден, отправьте ему письмо». Хорошая стратегия, чтобы попытаться гарантировать, что функции могут быть переведены в красивые предложения, подобные этому, состоит в том, чтобы ограничить функцию одной из следующих задач.

- ◆ Выполнение одной-единственной задачи.
- ◆ Создание более сложного поведения путем простого вызова других хорошо названных функций.

Это не точная наука, потому что «одна-единственная задача» открыта для интерпретации, и даже при составлении более сложного поведения путем простого вызова других функций нам, вероятно, все равно понадобится поток управления (например, оператор `if` или цикл `for`). Поэтому, как только мы написали функцию, стоит попытаться прочесть ее как предложение. Если это трудно сделать или предложение становится очень громоздким, то, скорее всего, функция слишком длинная, и было бы полезно разбить ее на более мелкие функции.

В случае с `sendOwnerALetter()` мы уже установили, что это не переводится в хорошее предложение, а также явно не соответствует только что упомянутой стратегии. Функция выполняет две задачи: поиск адреса владельца и запуск отправки письма. Но, вместо того чтобы делать это, просто составляя свою функцию из других функций, она содержит основную логику для поиска адреса владельца.

Лучшим подходом было бы разделить логику поиска адреса владельца на отдельную функцию, поэтому функция `sendOwnerALetter()` теперь преобразуется в более идеальное предложение. В листинге 2.3 показано, как это выглядит. Любой,

кто прочитает его после этого изменения, может легко понять, как она решает данную подзадачу.

Узнайте адрес владельца.

Отправьте письмо владельцу, если адрес был найден.

Еще одно преимущество нового кода в листинге 2.3 заключается в том, что логику поиска адреса владельца теперь легче использовать повторно. В будущем может возникнуть запрос на создание функции, которая отображает только адрес владельца без отправки письма. Инженер, создающий это, мог бы повторно использовать функцию `getOwnersAddress()` в том же классе или переместить ее в соответствующий вспомогательный класс и сделать ее общедоступной относительно легко.

Листинг 2.3. Меньшие функции

```
SentConfirmation? sendOwnerALetter(Vehicle vehicle, Letter letter) {  
    Address? ownersAddress = getOwnersAddress(vehicle); ❶  
    if (ownersAddress == null) { ❷  
        return null;  
    }  
    return sendLetter(ownersAddress, letter);  
}  
  
private Address? getOwnersAddress(Vehicle vehicle) { ❸  
    if (vehicle.hasBeenScraped()) {  
        return SCRAPYARD_ADDRESS;  
    }  
    Purchase? mostRecentPurchase =  
        vehicle.getMostRecentPurchase();  
    if (mostRecentPurchase == null) {  
        return SHOWROOM_ADDRESS;  
    }  
    return mostRecentPurchase.getBuyersAddress();  
}
```

❶ Узнайте адрес владельца.

❷ Отправьте письмо владельцу, если адрес был найден.

❸ Функция для поиска адреса владельца. Может быть легко использована повторно.

Создание небольших и сфокусированных функций — один из лучших способов обеспечить читаемость и многократное использование кода. При создании кода довольно легко в итоге написать слишком длинную и не очень читабельную функцию. Поэтому после написания первого фрагмента нашего кода и перед отправкой его на проверку часто стоит критически взглянуть на него. Всякий раз, когда мы видим функцию, которую трудно перевести в предложение, мы должны рассмотреть возможность разделения частей логики на хорошо названные вспомогательные функции.

2.3.3. Классы

Инженеры часто обсуждают, каков идеальный размер для одного класса. Было выдвинуто множество теорий и эмпирических правил, например таких.

- ◆ *Количество строк* — иногда вы будете слышать указания, такие, как «класс не должен быть длиннее 300 строк кода».
- ◆ Очень часто (но не всегда) верно, что класс длиной более 300 строк обрабатывает слишком много концепций и должен быть разбит. Это эмпирическое правило не подразумевает, что класс размером 300 строк или меньше является подходящим размером. Это служит только предупреждением о том, что что-то может быть не так, но не является гарантией того, что все правильно. Поэтому подобные правила часто имеют весьма ограниченное практическое применение.
- ◆ *Связность* ❶ — это показатель того, насколько хорошо вещи внутри класса «принадлежат» друг другу, при этом идея заключается в том, что хороший класс — это тот, который обладает высокой связностью. Есть много способов, с помощью которых вещи можно классифицировать как взаимосвязанные друг с другом. Вот несколько примеров.
- ◆ Последовательная связность означает, что выход одной вещи необходим в качестве входа для другой вещи. Примером этого в реальном мире может быть приготовление чашки свежего кофе. Мы не можем варить кофе, пока не измельчим зерна; результат процесса измельчения зерен является входным сигналом для процесса приготовления кофе. Поэтому мы могли бы сделать вывод, что измельчение и заваривание тесно связаны друг с другом.
- ◆ Функциональная связность означает, что группа вещей способствует достижению одной задачи. Определение одной задачи может быть весьма субъективным; это, примерно, как если бы вы хранили все свое оборудование для приготовления тортов в одном специальном ящике на своей кухне. Вы могли решить, что миски для смешивания, деревянные ложки и формы для тортов являются единым целым и принадлежат друг другу, потому что все они способствуют выполнению одной и той же функциональной задачи: приготовлению тортов.
- ◆ Разделение проблем ❷. Это принцип проектирования, который гласит, что системы должны быть разделены на компоненты, каждый из которых имеет дело с отдельной проблемой (задачей). Примером этого в реальном мире является то, как игровая консоль обычно отделена от телевизора, а не объединена в одно неразделимое устройство. Игровая консоль предназначена для запуска игры, а телевизор — для отображения движущегося изображения. Это обеспечивает большую настраиваемость: один человек, покупающий игровую консоль, может жить в небольшой квартире и иметь место только для небольшого телевизора, в то время как кто-то другой, у кого больше места, может захотеть подключить его к настенному телевизору с диагональю 292 дюйма. Разделение этих элементов также позволяет нам обновлять один без необходимости обновлять другой. Когда появится новая, более быстрая игровая консоль, нам не придется тратить-ся на новый телевизор.

❶ Идея использования связности в качестве показателя для оценки структуры программного обеспечения была впервые предложена Ларри Л. Константином в 1960-х годах, а затем расширена Уэйном П. Стивенсом, Гленфордом Дж. Майерсом и Ларри Л. Константином в 1970-х годах.

❷ Широко распространено мнение, что термин «разделение интересов» был введен Эдсгером У. Дейкстрой в 1970-х годах.

Идеи связности и разделения проблем, как правило, требуют от нас принятия решения о том, на каком уровне полезно рассматривать группу взаимосвязанных вещей как одну вещь. Это часто может быть сложнее, чем может показаться, потому что это может быть очень субъективно. Для одного человека объединение измельчения и приготовления кофе может иметь смысл, но для другого человека, который просто хочет измельчить специи для приготовления пищи, это может показаться очень бесполезным способом группировки вещей, потому что они явно не хотят варить свои специи.

Я не встречал инженеров, которые совершенно не знакомы с этими эмпирическими правилами или которые не согласились бы с таким утверждением, как «класс должен быть связным и в идеале заниматься одной-единственной вещью». Но, несмотря на знание этого совета, многие инженеры все еще пишут слишком большие классы. Классы часто оказываются слишком большими, когда инженеры недостаточно тщательно продумывают, сколько различных концепций они вводят в рамках одного класса и какие элементы логики могут быть подходящими для повторного использования или перенастройки. Иногда это может произойти, когда класс создается впервые, а иногда — когда класс органично растет с течением времени, поэтому важно думать о том, не станет ли класс слишком большим как при изменении существующего кода, так и при написании совершенно нового кода.

Эмпирические правила, такие, как «класс должен заниматься только одной вещью» или «класс должен быть связным», существуют для того, чтобы попытаться помочь инженерам создать код более высокого качества. Но нам все равно нужно хорошо подумать о том, чего мы в принципе пытаемся достичь. Что касается уровней кода и создания классов, четыре основных принципа, которые были определены в *главе 1*, отражают то, чего мы должны попытаться достичь.

- ◆ *Сделайте код читабельным.* Чем больше количество различных концепций, которые мы объединяем в одном классе, тем менее читабельным будет этот класс. Человеческий мозг не способен думать о многих вещах одновременно. Чем большую когнитивную нагрузку мы возлагаем на других инженеров, пытающихся прочесть код, тем больше времени это займет у них и тем больше вероятность того, что они неправильно его истолкуют.
- ◆ *Сделайте код модульным.* Использование классов и интерфейсов является одним из лучших способов сделать код модульным. Если решение подзадачи является автономным внутри своего собственного класса, а другие классы взаимодействуют с ним только с помощью нескольких хорошо продуманных открытых функций, тогда будет легко заменить эту реализацию другой, если нам понадобится.

- ◆ *Сделайте код многоразовым и обобщаемым.* Если для решения проблемы требуется решить две подзадачи, то есть определенная вероятность, что кому-то еще также может потребоваться решить одну из этих подзадач в будущем. Если мы объединим решение обеих подзадач в одном классе, это уменьшит вероятность того, что кто-то другой сможет повторно использовать одну из них.
- ◆ *Сделайте код тестируемым и протестируйте его должным образом.* В предыдущем разделе использовалась аналогия с домом, где мы хотели бы проверить надежность всех частей дома перед его покупкой, а не только его внешний вид. Аналогично, если логика разбита на классы, то становится намного проще правильно протестировать каждую ее часть.

Рис. 2.4 иллюстрирует, как слишком большой класс может в конечном итоге достичь противоположного из этих четырех столпов.

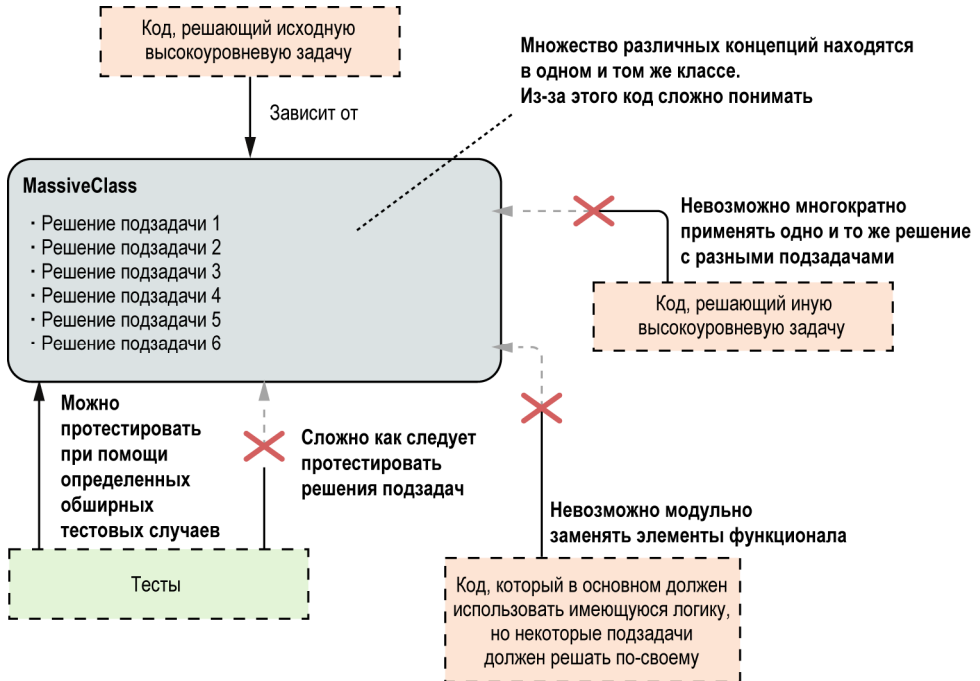


Рис. 2.4. Отсутствие разбивки кода на классы соответствующего размера часто приводит к тому, что код имеет дело со слишком большим количеством концепций одновременно и является менее читаемым, модульным, многоразовым, обобщаемым и тестируемым

Листинг 2.4. Класс, который слишком велик

```
class TextSummarizer {
    ...
    String summarizeText(String text) {
        return splitIntoParagraphs(text)
            .filter(paragraph ->
```



```
        calculateImportance(paragraph) >=
            IMPORTANCE_THRESHOLD)
        .join("\n\n");
    }

    private Double calculateImportance(String paragraph) {
        List<String> nouns = extractImportantNouns(paragraph);
        List<String> verbs = extractImportantVerbs(paragraph);
        List<String> adjectives =
            extractImportantAdjectives(paragraph);
        ... a complicated equation ...
        return importanceScore;
    }

    private List<String> extractImportantNouns(String text) { ... }
    private List<String> extractImportantVerbs(String text) { ... }
    private List<String> extractImportantAdjectives(String text)
    { ... }

    private List<String> splitIntoParagraphs(String text) {
        List<String> paragraphs = [];
        Int? start = detectParagraphStartOffset(text, 0);
        while (start != null) {
            Int? end = detectParagraphEndOffset(text, start);
            if (end == null) {
                break;
            }
            paragraphs.add(text.substring(start, end));
            start = detectParagraphStartOffset(text, end);
        }
        return paragraphs;
    }

    private Int? detectParagraphStartOffset(
        String text, Int fromOffset) { ... }

    private Int? detectParagraphEndOffset(
        String text, Int fromOffset) { ... }
}
```

Если бы мы поговорили с автором этого класса, он вполне мог бы сказать, что класс касается только одного: обобщения отрывка текста. И на высоком уровне он был бы отчасти прав. Но класс явно содержит код, который решает кучу подзадач:

- ◆ разделение текста на абзацы;
- ◆ вычисление оценки важности для строки текста (дополнительно делится на подзадачи поиска важных существительных, глаголов и прилагательных).

Основываясь на этом наблюдении, другой инженер может возразить и сказать: «Нет, этот класс занимается несколькими разными вещами. Он должен быть разбит». В этом сценарии оба инженера согласны с тем, что класс должен быть связным и заниматься одним делом, но они не согласны с тем, считается ли решение связанных с ним подзадач различными проблемами или неотъемлемыми частями основной проблемы. Чтобы лучше судить о том, следует ли разбивать этот класс, возможно, было бы лучше посмотреть, как он складывается в соответствии с теми столпами, о которых мы только что упомянули. Если мы сделаем это, мы, скорее всего, придем к выводу, что класс в его нынешнем виде является кодом низкого качества, основываясь на следующем (также проиллюстрированном на рис. 2.5).

- ◆ *Код не так удобочитаем, как мог бы быть.* Когда мы первоначально читаем код, это стена из нескольких различных концепций, таких, как разделение текста на абзацы, извлечение таких вещей, как важные существительные, и вычисление оценок важности. Требуется некоторое время, чтобы выяснить, какая из этих концепций необходима для решения какой из подзадач.
- ◆ *Код не является особенно модульным.* Это затрудняет перенастройку или изменение кода. Этот алгоритм, без сомнения, является довольно наивным способом обобщения отрывка текста и, вероятно, является тем, что инженеры захотят повторить с течением времени. Трудно перенастроить код, чтобы опробовать что-то новое, не изменяя его для каждого вызывающего абонента. Было бы лучше, если бы код был модульным, чтобы мы могли, например, поменять местами новый способ расчета оценки важности.
- ◆ *Код не может быть повторно использован.* При работе над решением другой проблемы нам может потребоваться решить некоторые из тех же подзадач, которые были решены здесь. Если бы нам нужно было создать функцию для подсчета количества абзацев, содержащихся в отрывке текста, было бы здорово, если бы мы могли просто повторно использовать функцию `splitIntoParagraphs()`. В настоящее время мы не можем этого сделать, и нам пришлось бы заново решать эту подзадачу самостоятельно или же рефакторизовать класс `TextSummarizer`. Сделать функцию `splitIntoParagraphs()` общедоступной, чтобы разрешить повторное использование, может показаться заманчивым, но это не было бы хорошей идеей: это загрязнило бы общедоступный API класса `textsummarizer`, казалось бы, несвязанной концепцией, и было бы неудобно изменять эту функциональность в классе `textsummarizer` в будущем, так как другой внешний код начнет зависеть от функции `splitintoParagraphs()`.
- ◆ *Код не поддается обобщению.* Все решение предполагает, что вводимый текст является обычным текстом. Но мы, возможно, захотим начать обобщать веб-страницы в ближайшем будущем. В этом случае нам может потребоваться ввести фрагмент HTML, а не обычный текст. Если бы код был более модульным, то мы, возможно, могли бы заменить логику, которая разбивает текст на абзацы, чем-то, что могло бы разделить HTML на абзацы.
- ◆ *Код трудно правильно протестировать.* Многие из решаемых подзадач на самом деле имеют довольно сложные решения. Разделение текста на абзацы

выглядит как нетривиальная задача, и вычисление оценки важности является особенно сложным алгоритмом. В настоящее время все, что можно проверить, — это общее поведение с помощью функции `summarizeText()`, но будет сложно правильно проверить все тонкости того, работает ли код оценки важности, просто вызвав `summarizeText()`. Мы могли бы начать публиковать другие функции (такие, как `calculateImportance()`), чтобы мы могли их правильно протестировать, но это затем загромождало бы общедоступный API `textsummarizer`. Мы могли бы добавить комментарий, гласящий: «Только для общего просмотра для тестирования», но это еще больше увеличило бы когнитивную нагрузку на других инженеров.

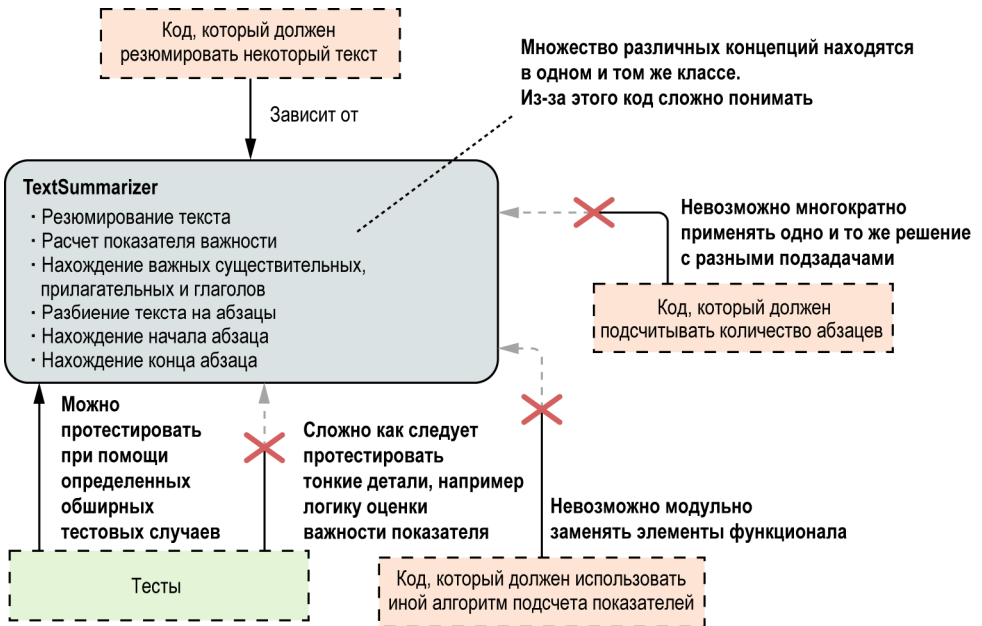


Рис. 2.5. Класс сумматора текста содержит слишком много различных концепций, и в результате код становится менее читаемым, модульным, многоразовым, обобщаемым и тестируемым

Класс `TextSummarizer` явно слишком велик и обрабатывает слишком много различных концепций, и это снижает качество кода. В следующем разделе показано, как можно улучшить этот код.

Как улучшить код

Код можно было бы улучшить, разделив решение каждой подзадачи на свой собственный соответствующий класс, как показано в следующем листинге. Классы, которые решают подзадачи, предоставляются классу `TextSummarizer` в качестве параметров в его конструкторе. Этот шаблон известен как *внедрение зависимостей* и будет более подробно рассмотрен в главе 8.

Листинг 2.5. Один класс на концепцию

```

class TextSummarizer {
    private final ParagraphFinder paragraphFinder;
    private final TextImportanceScorer importanceScorer;

    TextSummarizer(
        ParagraphFinder paragraphFinder, ❶
        TextImportanceScorer importanceScorer) {
        this.paragraphFinder = paragraphFinder;
        this.importanceScorer = importanceScorer;
    }

    static TextSummarizer createDefault() { ❷
        return new TextSummarizer(
            new ParagraphFinder(),
            new TextImportanceScorer());
    }

    String summarizeText(String text) {
        return paragraphFinder.find(text)
            .filter(paragraph ->
                importanceScorer.isImportant(paragraph))
            .join("\n\n");
    }
}

class ParagraphFinder { ❸
    List<String> find(String text) {
        List<String> paragraphs = [];
        Int? start = detectParagraphStartOffset(text, 0);
        while (start != null) {
            Int? end = detectParagraphEndOffset(text, start);
            if (end == null) {
                break;
            }
            paragraphs.add(text.substring(start, end));
            start = detectParagraphStartOffset(text, end);
        }
        return paragraphs;
    }

    private Int? detectParagraphStartOffset(
        String text, Int fromOffset) { ... }
    private Int? detectParagraphEndOffset(
        String text, Int fromOffset) { ... }
}

```

```
class TextImportanceScorer {  
    ...  
    Boolean isImportant(String text) {  
        return calculateImportance(text) >=  
            IMPORTANCE_THRESHOLD;  
    }  
  
    private Double calculateImportance(String text) {  
        List<String> nouns = extractImportantNouns(text);  
        List<String> verbs = extractImportantVerbs(text);  
        List<String> adjectives =  
            extractImportantAdjectives(text);  
        ... a complicated equation ...  
        return importanceScore;  
    }  
    private List<String> extractImportantNouns(String text) { ...  
    }  
    private List<String> extractImportantVerbs(String text) { ...  
    }  
    private List<String> extractImportantAdjectives(String text) {  
        ... }  
}
```

- ❶ Зависимости класса вводятся через его конструктор. Это называется внедрением зависимостей.
- ❷ Статическая фабричная функция, облегчающая вызывающим пользователям создание экземпляра класса по умолчанию.
- ❸ Решения подзадач разбиты на свои собственные классы.

Код теперь намного более удобочитаем, потому что каждый класс требует, чтобы кто-то читал его, чтобы иметь дело только с несколькими концепциями одновременно. Мы можем посмотреть на класс `TextSummarizer` и в течение нескольких секунд узнать все концепции и шаги, которые формируют высокоуровневый алгоритм.

- ◆ Найдите абзацы.
- ◆ Отфильтруйте те, которые не важны.
- ◆ Присоединитесь к оставшимся абзацам.

Если это все, что мы хотели знать, то отлично; работа выполнена. Если бы нам было все равно, как подсчитывались баллы важности, но мы хотели узнать, как были найдены абзацы, тогда мы могли бы перейти в класс `ParagraphFinder` и быстро понять, как решается эта подзадача.

Как показано на рис. 2.6, существует множество других преимуществ.

- ◆ *Код теперь более модульный и перенастраиваемый.* Если бы мы хотели попробовать другой способ оценки текста, то было бы легко извлечь `TextImportanceScorer`

в интерфейс и создать альтернативную его реализацию. Мы обсудим это в следующем разделе.

- ♦ *Код более многообразный.* Теперь мы можем довольно легко использовать класс поиска абзацев в других сценариях, если захотим.
- ♦ *Код более проверяемый.* Легко написать комплексные и целенаправленные тесты для каждого из классов подзадач.

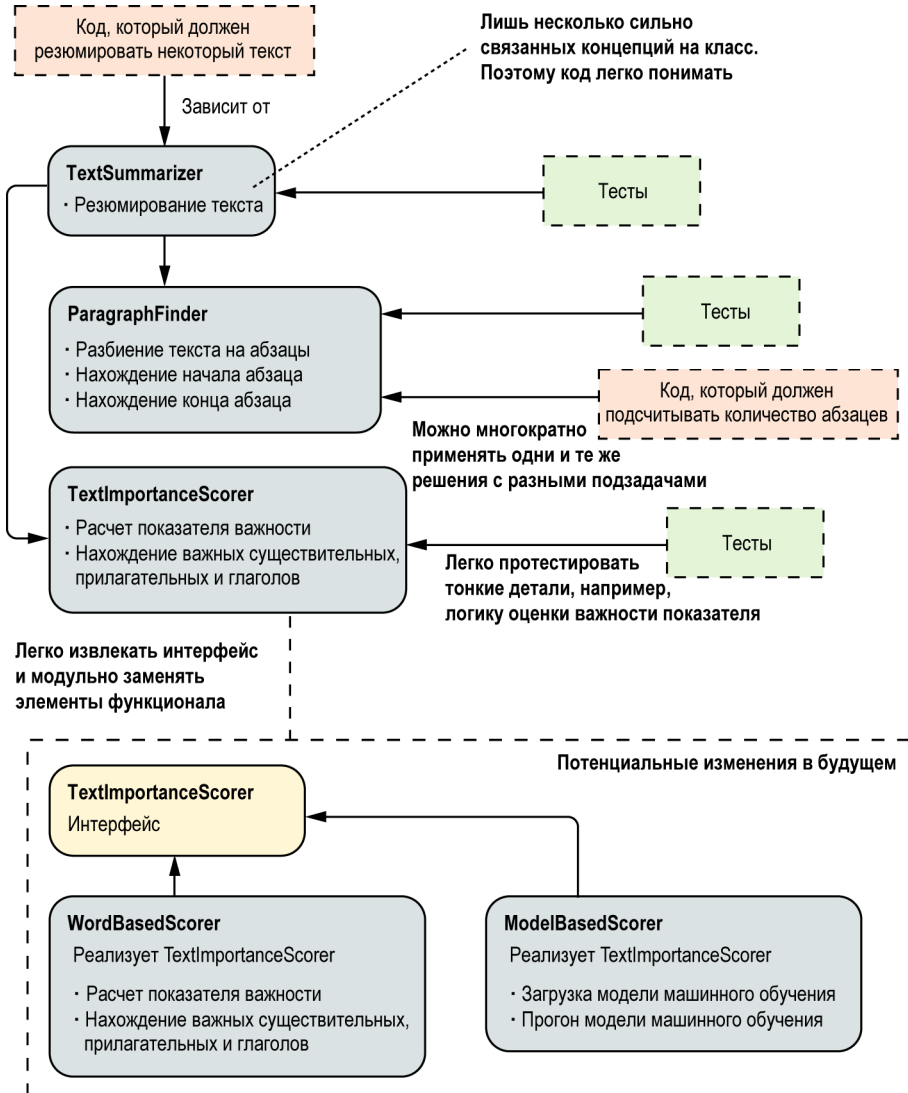


Рис. 2.6. Разбиение кода на уровни абстракции соответствующего размера приводит к тому, что код имеет дело только с несколькими концепциями одновременно.

Это делает код более читаемым, модульным, многообразным, обобщаемым и тестируемым

Большие классы, которые выполняют много функций, слишком распространены во многих кодовых базах, и, как показано в этом разделе, это часто приводит к снижению качества кода. При проектировании иерархии классов полезно тщательно подумать о том, соответствует ли она принципам качества кода, которые только что обсуждались. Классы часто могут расти с течением времени, становясь слишком большими, поэтому полезно думать о перечисленных выше принципах при изменении существующих классов, а также при создании новых. Разбиение кода на классы соответствующего размера является одним из наиболее эффективных инструментов для обеспечения того, чтобы у нас были хорошие уровни абстракции, поэтому стоит потратить время и подумать над тем, чтобы сделать это правильно.

2.3.4. Интерфейсы

Один из подходов, который иногда используется, для того чтобы заставить слои быть отдельными и гарантировать, что детали реализации не будут просачиваться между слоями, заключается в определении интерфейса, который определяет, какие общедоступные функции будут предоставляться этим слоем. Фактический класс, содержащий код для слоя, затем реализует этот интерфейс. Уровни кода выше этого будут зависеть только от интерфейса, а не от конкретного класса, реализующего логику.

Если у нас есть более одной реализации для данного абстрактного слоя или если мы думаем, что в будущем мы добавим больше, то обычно рекомендуется определить интерфейс. Чтобы продемонстрировать это, рассмотрим пример для обобщения текста, который мы видели в предыдущем разделе. Важной подзадачей является оценка фрагментов текста (в данном случае абзацев), чтобы определить, можно ли их исключить из итога. В исходном коде используется довольно наивное решение, основанное на поиске важных слов.

Более надежным подходом может быть использование машинного обучения для обучения модели определению того, важен ли фрагмент текста. Вероятно, это то, с чем мы хотели бы поэкспериментировать, сначала опробовать в режиме разработки и, возможно, выпустить в качестве дополнительной бета-версии. Мы не хотим просто заменить старую логику подходом, основанным на модели, за один раз; нам нужен способ настроить код с помощью любого из подходов.

Один из лучших способов сделать это — извлечь класс `TextImportanceScorer` в интерфейс, а затем создать класс реализации для каждого из подходов к решению этой подзадачи. Класс `TextSummarizer` будет зависеть только от интерфейса `TextImportanceScorer` и никогда ни в одной из конкретных реализаций. На рис. 2.7 показано, как это может выглядеть с точки зрения зависимостей между различными классами и интерфейсом.

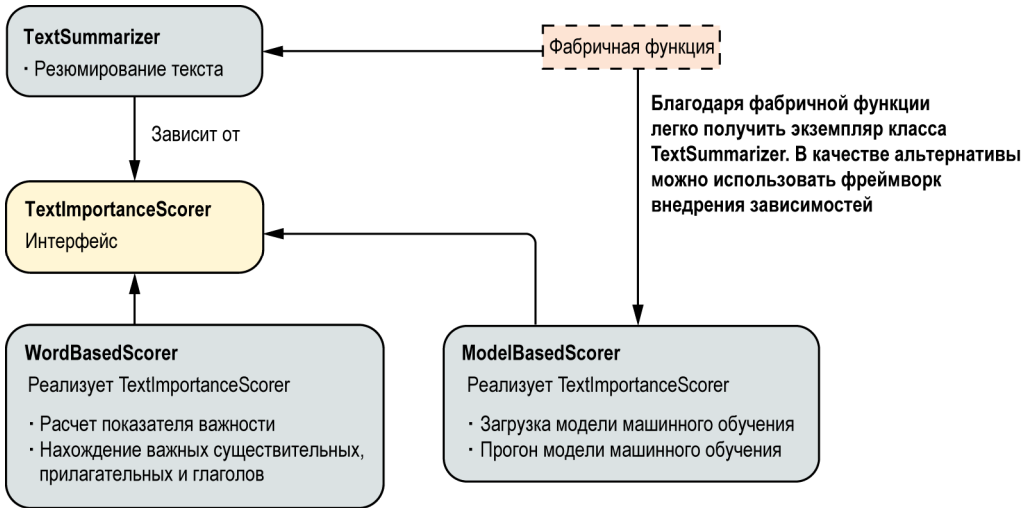


Рис. 2.7. Определив интерфейс для представления слоя абстракции, мы можем легко поменять местами реализации для решения данной подзадачи. Это делает код более модульным и настраиваемым

Код в следующем листинге показывает, как могут выглядеть новые классы интерфейса и реализации.

Листинг 2.6. Интерфейс и реализации

```
interface TextImportanceScorer { ❶
    Boolean isImportant(String text);
}

class WordBasedScorer implements TextImportanceScorer { ❷
    ...
    override Boolean isImportant(String text) { ❸
        return calculateImportance(text) >=
            IMPORTANCE_THRESHOLD;
    }

    private Double calculateImportance(String text) {
        List<String> nouns = extractImportantNouns(text);
        List<String> verbs = extractImportantVerbs(text);
        List<String> adjectives =
            extractImportantAdjectives(text);
        ... a complicated equation ...
        return importanceScore;
    }

    private List<String> extractImportantNouns(String text) { ... }
    private List<String> extractImportantVerbs(String text) { ... }
```



```
private List<String> extractImportantAdjectives(String text) {  
    ... }  
}  
  
class ModelBasedScorer implements TextImportanceScorer { ❹  
    private final TextPredictionModel model;  
    ...  
  
    static ModelBasedScorer create() {  
        return new ModelBasedScorer(  
            TextPredictionModel.load(MODEL_FILE));  
    }  
  
    override Boolean isImportant(String text) {  
        return model.predict(text) >=  
            MODEL_THRESHOLD;  
    }  
}
```

❶ Индикатор важности текста теперь является интерфейсом, а не классом.

❷ Исходный класс оценки важности текста переименован и реализует новый интерфейс.

❸ Функция помечена как `override`, чтобы указать, что она переопределяет функцию из интерфейса.

❹ Новый оценщик на основе модели также реализует интерфейс.

Теперь легко настроить `TextSummarizer` для использования либо `WordBasedScorer`, либо `ModelBasedScorer`, используя одну из двух фабричных функций. В следующем листинге показано, как может выглядеть код для двух фабричных функций для создания экземпляра класса `TextSummarizer`.

Листинг 2.7. Фабричные функции

```
TextSummarizer createWordBasedSummarizer() {  
    return new TextSummarizer(  
        new ParagraphFinder(), new WordBasedScorer());  
}  
  
TextSummarizer createModelBasedSummarizer() {  
    return new TextSummarizer(  
        new ParagraphFinder(), ModelBasedScorer.create());  
}
```

Интерфейсы являются чрезвычайно полезным инструментом для создания кода, который обеспечивает чистые уровни абстракции. Всякий раз, когда нам нужно переключаться между двумя или более различными конкретными реализациями для данной подзадачи, обычно лучше всего определить интерфейс для представле-

ния уровня абстракции. Это сделает наш код более модульным и значительно упростит перенастройку.

Интерфейсы для всего?

Если у вас есть только одна реализация данного уровня абстракции и вы не планируете добавлять больше в будущем, то вы (и ваша команда) решаете, стоит ли скрывать слои за интерфейсом. Некоторые философии разработки программного обеспечения стремятся поощрять это. Если бы мы последовали этому и скрыли класс `TextSummarizer`, который мы только что видели, за интерфейсом, то он выглядел бы примерно так, как показано в листинге 2.8. В этом режиме `TextSummarizer` — это интерфейс, от которого будут зависеть уровни кода выше этого; они никогда не будут напрямую зависеть от класса реализации `TextSummarizerImpl`.

Листинг 2.8. Интерфейс и единичная реализация

```
interface TextSummarizer {
    String summarizeText(String text); ❶
}

class TextSummarizerImpl implements TextSummarizer { ❷
    . . .

    override String summarizeText(String text) {
        return paragraphFinder.find(text)
            .filter(paragraph ->
                importanceScorer.isImportant(paragraph))
            .join("\n\n");
    }
}
```

❶ Пользователям этого уровня абстракции будут видны только функции, определенные в интерфейсе.

❷ `TextSummarizerImpl` — это единственный класс, реализующий интерфейс сумматора текста.

Несмотря на то что существует только одна реализация `TextSummarizer`, и даже если мы не можем представить, что добавим другую ее реализацию в будущем, у этого подхода есть некоторые преимущества.

- ◆ Это делает общедоступный API очень понятным — нет никакой путаницы в том, какие функции инженеры, использующие этот уровень, должны/не должны использовать. Если инженер добавляет новую общедоступную функцию в класс `TextSummarizerImpl`, то она не будет доступна для слоев кода выше этого, поскольку они зависят только от интерфейса `TextSummarizer`.
- ◆ Возможно, мы ошиблись в том, что нам нужна только одна реализация — при первоначальном написании кода мы, возможно, были абсолютно уверены, что

нам не понадобится вторая реализация, но через месяц или два это предположение может оказаться неверным. Возможно, мы понимаем, что обобщение текста, просто опуская несколько абзацев, не очень эффективно, и решаем поэкспериментировать с другим алгоритмом, который обобщает текст совершенно по-другому.

- ◆ *Это может упростить тестирование* — например, если наш класс реализации сделал что-то особенно сложное или зависел от сетевого ввода-вывода, тогда мы могли бы заменить его макетной или поддельной реализацией во время тестирования. В зависимости от используемого языка программирования для этого может потребоваться определить интерфейс.
- ◆ *Один и тот же класс может решить две подзадачи* — иногда один класс может предоставить реализацию двум или более различным уровням абстракции. Примером этого является то, как класс реализации `LinkedList` может реализовывать как интерфейс списка — `List`, так и интерфейс очереди — `Queue`. Это означает, что его можно использовать в качестве очереди в одном сценарии, не позволяя коду в этом сценарии также использовать его в качестве списка. Это может значительно повысить обобщаемость кода.

С другой стороны, есть и недостатки определения интерфейса.

- ◆ *На это тратится немного больше усилий* — нам нужно написать еще несколько строк кода (и, возможно, новый файл), чтобы определить интерфейс.
- ◆ *Это может усложнить код* — когда другой инженер хочет понять код, ему может быть сложнее ориентироваться в логике. Если он хочет понять, как решается определенная подзадача, то, вместо того чтобы просто напрямую переходить к классу, реализующему слой ниже, он должен перейти к интерфейсу, а затем найти конкретный класс, реализующий этот интерфейс.

По моему личному опыту, занятие крайней позиции и скрытие каждого отдельного класса за интерфейсом часто выходит из-под контроля и может создавать код, который становится излишне сложным для понимания и изменения. Мой совет — использовать интерфейс там, где он дает ощутимую выгоду, но не делать этого просто ради того, чтобы делать. Несмотря на это, по-прежнему важно сосредоточиться на создании чистых и четких слоев абстракции. Даже если мы не определяем интерфейс, мы все равно должны очень тщательно подумать о том, какие общедоступные функции предоставляют наши классы, и убедиться, что детали реализации не проникли в класс. В общем, всякий раз, когда мы пишем или изменяем класс, мы должны убедиться, что будет простой задачей скрыть его за интерфейсом позднее, если это понадобится.

2.3.5. Когда слои становятся слишком тонкими

Несмотря на преимущества, существуют некоторые сопутствующие потери, связанные с разделением кода на отдельные уровни.

- ◆ Больше строк кода из-за всех стандартных шаблонов, необходимых для определения класса или импорта зависимостей в новый файл.

- ◆ Усилие, необходимое для переключения между файлами или классами при следовании логической цепочке.
- ◆ Если мы в итоге скроем слой за интерфейсом, то потребуются больше усилий, чтобы выяснить, какая реализация используется в каких сценариях. Это может затруднить понимание логики или отладку.

Эти потери, как правило, довольно низки по сравнению со всеми преимуществами кода, разделенного на отдельные слои, но стоит помнить, что нет смысла разделять код только ради этого. Может наступить момент, когда затраты перевесят выгоды, поэтому полезно руководствоваться здравым смыслом.

В листинге 2.9 показано, как мог бы выглядеть код, если бы класс `ParagraphFinder`, который мы видели ранее, был разделен на большее количество слоев путем разделения искателей смещения начала и конца на их собственные классы за общим интерфейсом. Слои кода, вероятно, сейчас слишком тонкие, потому что трудно представить, что классы `ParagraphStartOffsetDetector` и `ParagraphEndOffsetDetector` будут использоваться чем-либо еще, кроме класса `ParagraphFinder`.

Листинг 2.9. Слишком тонкие слои кода

```
class ParagraphFinder {
    private final OffsetDetector startDetector;
    private final OffsetDetector endDetector;
    ...

    List<String> find(String text) {
        List<String> paragraphs = [];
        Int? start = startDetector.detectOffset(text, 0);
        while (start != null) {
            Int? end = endDetector.detectOffset(text, start);
            if (end == null) {
                break;
            }
            paragraphs.add(text.substring(start, end));
            start = startDetector.detectOffset(text, end);
        }
        return paragraphs;
    }
}

interface OffsetDetector {
    Int? detectOffset(String text, Int fromOffset);
}

class ParagraphStartOffsetDetector implements OffsetDetector {
    override Int? detectOffset(String text, Int fromOffset) { ...
    }
}
```

```
class ParagraphEndOffsetDetector implements OffsetDetector {  
    override Int? detectOffset(String text, Int fromOffset) { ...  
}  
}
```

Даже если класс `ParagraphFinder` может быть полезен в другом месте, конечно, трудно представить, что кто-то будет использовать `ParagraphStartOffsetDetector`, а не эквивалентный `ParagraphEndOffsetDetector`, потому что их реализации должны иметь взаимно согласованное представление о том, как определить начало и конец абзаца.

Важно определить правильную толщину слоев кода; базы кода становятся совершенно неуправляемыми без значимых слоев абстракции. Если мы сделаем слои слишком толстыми, то несколько абстракций будут объединены, что приведет к тому, что код не будет модульным, многоразовым или читаемым. Если мы сделаем слои слишком тонкими, то в итоге можем разделить то, что должно быть одним слоем абстракции, на два, что приведет к ненужной сложности, а также может означать, что соседние слои не так хорошо разделены, как должны быть. В общем, наличие слишком толстых слоев часто вызывает более серьезные проблемы, чем наличие слишком тонких слоев, поэтому, если мы не уверены, часто лучше сделать слои тонкими.

Как мы видели ранее с классами, трудно придумать единственное правило или совет, который бы окончательно сказал нам, является ли слой слишком толстым, поскольку это часто будет зависеть от характера решаемой нами проблемы. Лучший совет — использовать свое суждение и тщательно подумать о том, обеспечат ли созданные слои читаемость, многоразовость, обобщаемость, модульность и тестируемость кода. И помните: даже инженерам с многолетним опытом работы часто может потребоваться несколько итераций проектирования или переработки кода, прежде чем отправлять его в базу кода, чтобы получить правильные слои абстракции.

2.4. Как насчет микросервисов?

В архитектуре микросервисов решения отдельных проблем развертываются как автономные службы, а не просто как библиотеки, скомпилированные в единую программу. Это означает, что система разбита на несколько небольших программ, каждая из которых посвящена определенной задаче. Эти небольшие программы развертываются как выделенные службы, которые можно удаленно вызывать через API. Микросервисы обладают рядом преимуществ и в последние годы становятся все более популярными. В настоящее время они являются основой архитектуры для многих организаций и команд.

Один из аргументов, который вы услышите, заключается в том, что при использовании микросервисов структура кода и создание слоев абстракции в коде не важны. Причина в том, что сами микросервисы обеспечивают чистый слой абстракции,

и поэтому на самом деле не имеет значения, как структурирован или разбит код внутри. Хотя это правда, что микросервисы обычно предоставляют довольно чистые слои абстракции, они обычно имеют размер и область применения, что означает, что все еще полезно правильно продумать слои абстракции внутри них.

Для демонстрации этого представьте, что мы работаем в онлайн-магазине в команде, которая разрабатывает и поддерживает микросервис для проверки и изменения уровня запасов. Наш микросервис вызывается всякий раз, когда происходит одно из следующих событий:

- ◆ новая поставка товара поступает на склад;
- ◆ интерфейсу магазина необходимо знать, есть ли товар на складе, чтобы он мог отображать его пользователю;
- ◆ клиент что-то покупает.

Номинально этот микросервис делает только одно: управляет уровнем запасов. Но, вероятно, сразу очевидно, что существует множество подзадач, которые необходимо решить, чтобы сделать это «одно».

- ◆ Работа с концепцией предметов (то, что она на самом деле отслеживает).
- ◆ Обработка того факта, что существуют различные склады и местоположения, в которых могут находиться запасы.
- ◆ Понятие о том, что товар может быть на складе для клиентов в одной стране, но не в другой, в зависимости от того, на каких складах они находятся в пределах диапазона доставки.
- ◆ Взаимодействие с базой данных, в которой хранятся фактические уровни запасов.
- ◆ Интерпретация данных, возвращаемых базой данных.

Все, что ранее говорилось о решении проблемы путем разбиения ее на подзадачи, по-прежнему применимо, даже в рамках этого микросервиса. Например, определение наличия товара на складе для клиента включает в себя следующее:

- ◆ определение того, какие склады находятся в пределах досягаемости этого клиента;
- ◆ запрос хранилища данных для поиска запасов товара на любом из этих складов;
- ◆ интерпретация формата данных, возвращаемых базой данных;
- ◆ возврат ответа обратно вызывающему абоненту службы.

Более того, очень вероятно, что другие инженеры захотят использовать часть этой логики повторно. Вероятно, в компании есть и другие команды, которые отслеживают аналитику и тенденции, чтобы выяснить, какие товары компании следует прекратить, закупать больше или запускать специальные предложения. По соображениям эффективности и задержки они, скорее всего, будут использовать конвейер для сканирования непосредственно через базу данных запасов, а не вызывать нашу службу, но им все равно может потребоваться некоторая логика, чтобы помочь им

интерпретировать данные, возвращаемые базой данных, поэтому было бы хорошо, если бы они могли просто повторно использовать наш код, который делает это.

Микросервисы могут быть чрезвычайно хорошим способом разбить систему и сделать ее более модульной, но обычно это не меняет того факта, что нам все равно потребуется решить несколько подзадач для реализации сервиса. Создание правильных абстракций и слоев кода по-прежнему важно.

Итоги

- ◆ Разделение кода на чистые и четкие слои абстракции делает его более читаемым, модульным, многократно используемым, обобщаемым и тестируемым.
- ◆ Мы можем использовать функции, классы и интерфейсы (а также другие языковые функции), чтобы разбить наш код на слои абстракции.
- ◆ Решение о том, как разбить код на уровни абстракции, требует использования нашего суждения и знаний о проблеме, которую мы решаем.
- ◆ Проблемы, возникающие из-за наличия слишком толстых слоев, обычно хуже, чем проблемы, возникающие из-за наличия слишком тонких слоев. Если мы не уверены, часто бывает лучше ошибиться, сделав слои слишком тонкими.

3 Другие программисты и соглашения по написанию кода

ЭТА ГЛАВА ОХВАТЫВАЕТ

- ◇ Как другие инженеры будут взаимодействовать с нашим кодом
- ◇ Соглашения по написанию кода и мелкий шрифт в них
- ◇ Как минимизация условий, прописанных мелким шрифтом, может помочь предотвратить неправильное использование и сюрпризы
- ◇ Если условий, прописанных мелким шрифтом, избежать нельзя, то как проверки и утверждения могут быть использованы, чтобы эти условия воплотить в жизнь

Написание и поддержка программного обеспечения обычно является командной работой. Компании, создающие программное обеспечение, обычно нанимают нескольких программистов: это может быть команда из двух человек, работающих над одним продуктом, или тысячи инженеров, работающих над сотнями различных продуктов. Точное число на самом деле не имеет значения; самое главное, что другим программистам в конечном итоге придется взаимодействовать с кодом, который мы пишем, и, в свою очередь, нам придется взаимодействовать с кодом, который пишут они.

Двумя столпами качества кода, представленными в *главе 1*, были «избегать сюрпризов» и «сделать код трудным для неправильного использования». Они касаются того, что может произойти (и что может пойти не так), когда другие программисты взаимодействуют с нашим кодом. В этой главе будут рассмотрены различные методы передачи важных деталей кода другим программистам (при этом некоторые методы являются более надежными, чем другие). Затем это будет оформлено с помощью концепции соглашений по написанию кодов и условий, прописанных мелким шрифтом. В последних двух разделах этой главы будет рассмотрен проработанный пример кода, который слишком открыт для неправильного использования и неправильного толкования, и показано, как улучшить код. В *главах 6 и 7* будет приведено больше конкретных примеров, основанных на настоящей главе.

3.1. Ваш код и код других программистов

Если вы пишете код в составе команды, то этот код, скорее всего, будет построен поверх слоев кода, написанных другими, а другие, скорее всего, будут создавать

слои поверх вашего кода. Если вы решили различные подзадачи на этом пути и разбили их на чистые слои абстракции, то другие инженеры-программисты, скорее всего, также захотят использовать некоторые из них для совершенно других задач, которые вы, возможно, даже не рассматривали.

Чтобы продемонстрировать это, представьте, что вы работаете в компании, которая выпускает онлайн-журнал, в котором пользователи могут находить и читать статьи. Вам поручено написать функцию резюмирования текста, чтобы создавать краткие изложения статей для пользователей, ищущих что-то почитать. В итоге вы пишете код, который мы видели в предыдущей главе, с помощью `TextSummarizer` и связанных классов. (Если вы не можете вспомнить точный код или пропустили эту главу, тогда не беспокойтесь.) На рис. 3.1 показано, как написанный вами код для обобщения текста может в конечном итоге использоваться в программном обеспечении. Вы сможете увидеть, что ваш код зависит от более низких слоев кода, написанного другими программистами, и, в свою очередь, другие рассчитывают на ваш код для решения задач более высокого уровня. Также вы увидите, что ваш код повторно используется для нескольких функциональных возможностей. Возможно, вы изначально предполагали использовать его только для обобщения статей, но другие программисты продолжили использовать его (или его части) для обобщения комментариев и оценки времени чтения статей.

Еще одна важная вещь, которую следует иметь в виду, заключается в том, что требования постоянно меняются и развиваются: меняются приоритеты, добавляются новые функции, а системы иногда нуждаются в переносе на новые технологии. Это означает, что код и программное обеспечение также постоянно меняются. Рис. 3.1 в значительной степени является моментальным снимком, запечатлевающим конкретный момент времени; маловероятно, что часть программного обеспечения будет выглядеть точно так же через год или даже через несколько месяцев.

Группа программистов, постоянно вносящих изменения в кодовую базу, делает ее оживленным местом. И, как и в любом оживленном месте, если там есть хрупкие вещи, они будут сломаны. Есть причина, по которой вы не достаете свою красивую стеклянную посуду, когда устраиваете массовую вечеринку, и почему барьеры на стадионах, как правило, сделаны из металла и прикреплены к земле болтами: хрупкие вещи и оживленные места плохо сочетаются.

Итак, чтобы другие программисты не «наследили», код должен быть надежным и простым в использовании. Одним из главных соображений при написании высококачественного кода является понимание, предупреждение и устранение тех нежелательных вещей, которые могут произойти, когда другие внесут изменения в ваш код или им потребуется взаимодействовать с ним. Если вы в буквальном смысле не являетесь единственным программистом в компании и никогда ничего не забываете, вы не сможете написать высококачественный код, не учитывая взаимодействия с ним других людей.

При написании кода полезно учитывать три вещи (в следующих трех подразделах они будут рассмотрены более подробно):

- ♦ вещи, которые очевидны для вас, не очевидны для других;

- ◆ другие программисты непреднамеренно попытаются взломать ваш код;
- ◆ со временем вы забудете о своем собственном коде.

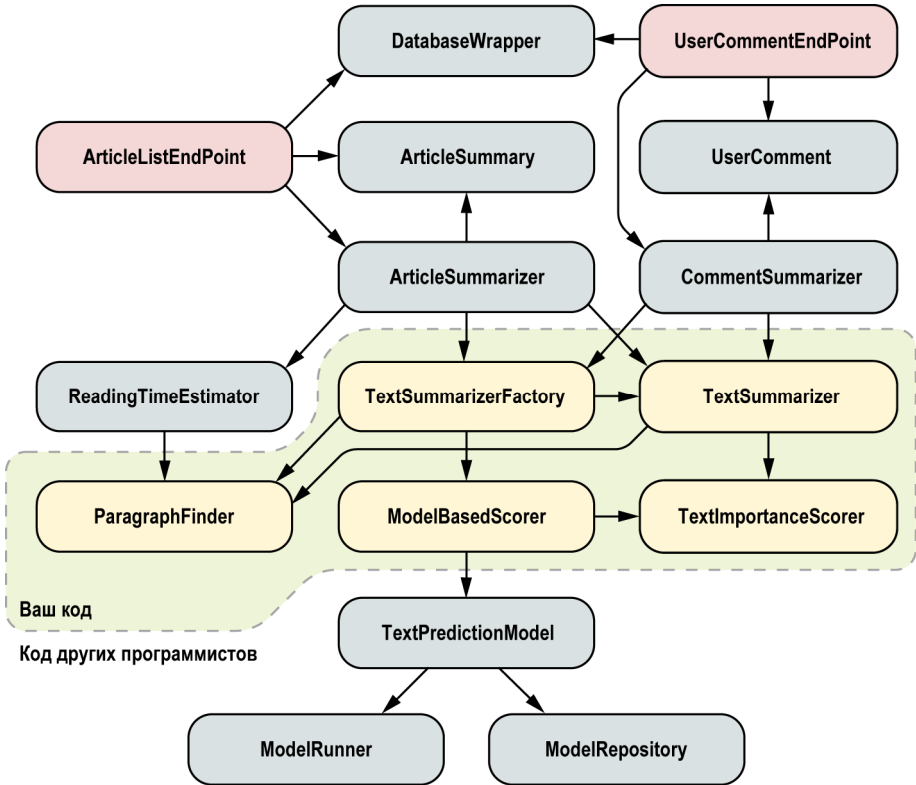


Рис. 3.1. Код, который вы пишете, редко живет в изоляции. Он зависит от кода, написанного другими программистами, а те, в свою очередь, пишут код, который зависит от вашего кода

3.1.1. Вещи, которые очевидны для вас, не очевидны для других

Перед тем как приступить к написанию какого-либо кода, вы, вероятно, уже потратите часы или дни на размышления о задаче, которую нужно решить. Возможно, вы прошли через несколько этапов проектирования, тестирования пользовательского опыта, получения обратной связи о продукте или сообщений об ошибках. Возможно, вы настолько знакомы со своей логикой, что все кажется очевидным, и вам едва ли нужно думать о том, почему что-то работает так, как оно работает, или почему вы решаете задачу так, а не иначе.

Но помните, что в какой-то момент другому программисту, вероятно, потребуется взаимодействовать с вашим кодом, вносить в него изменения или вносить изменения в что-то, от чего ваш код зависит. У этого человека не будет столько же време-

ни на то, чтобы понять и решить задачу, сколько было у вас. Вещи, которые казались вам совершенно очевидными, когда вы писали код, скорее всего, не очевидны для нового человека.

Полезно всегда учитывать это и убедиться, что ваш код сам объясняет то, как его следует использовать, что он делает и почему он это делает. Как вы узнаете в этой главе и в последующих, это не означает, что нужно писать кучу комментариев. Часто существуют лучшие способы сделать код ясным и понятным.

3.1.2. Другие программисты непреднамеренно попытаются взломать ваш код

Предположение, что другие программисты непреднамеренно попытаются взломать ваш код, может показаться чрезмерно циничным, но, как мы только что видели, ваш код не живет в вакууме; он зависит от множества других частей кода, а они, в свою очередь, зависят от еще большего количества частей кода. И, вероятно, существует даже больше фрагментов кода, которые зависят от вашего. Некоторые из этих фрагментов кода будут постоянно изменяться, по мере того как другие инженеры будут добавлять функции, рефакторинговать и вносить изменения. Таким образом, ваш код далек от того, чтобы жить в вакууме, более того, он живет на постоянно меняющемся фундаменте, а поверх него построены другие постоянно движущиеся части.

Ваш код может значить для вас целый мир, но большинство других программистов в вашей компании вряд ли знают о нем многое, и, когда они наткнутся на ваш код, у них не обязательно будет предварительное знание о том, зачем он нужен или что он делает. Весьма вероятно, что ваш коллега возьмет и добавит или изменит какой-то код таким образом, что это приведет к непреднамеренному повреждению или неправильному использованию кода.

Как мы видели в *главе 1*, программисты обычно вносят изменения в кодовую базу в локальной копии, а затем уже отправляют ее в основную кодовую базу. Если код не скомпилируется или тесты завершатся неудачно, они не смогут отправить свои изменения. Если кто-то другой внес изменения, которые повредили или неправильно использовали ваш код, то вы захотите убедиться, что их изменения хотя бы не будут внесены в основную кодовую базу, пока проблема не будет устранена. Единственные два надежных способа сделать это — убедиться, что, когда что-то сломается, код либо перестанет компилироваться, либо не пройдет тесты. Многие соображения, связанные с написанием высококачественного кода, в конечном счете сводятся к тому, чтобы гарантировать, что при повреждении кода произойдет одна из этих двух вещей.

3.1.3. Со временем вы забудете о своем собственном коде

Детали вашего кода могут казаться вам сейчас настолько свежими и важными, что вы не можете себе представить, что когда-нибудь их забудете. Но со временем они

перестанут быть свежими в вашем сознании и начнут забываться. Когда появится новая деталь или через год код выдаст ошибку, то возможно, что вам придется его изменить. Тогда вы вряд ли уже будете помнить все детали, как свои пять пальцев.

Все, что только что было сказано о неочевидности понятных вам вещей для других людей или о том, что другие могут повредить ваш код, в какой-то момент с большой долей вероятности станет и вашей реальностью. Когда вы смотрите на код, написанный вами год или два назад, это мало чем будет отличаться от знакомства с кодом, написанным другим человеком. Убедитесь, что ваш код понятен даже тому, у кого мало или вообще нет контекста, и сделайте так, чтобы его было трудно взломать. Вы не только окажете услугу своим коллегам, но и самому себе в будущем.

3.2. Как другие поймут, как надо использовать ваш код?

Когда вашему коллеге потребуется использовать ваш код или что-то поменять в коде, зависящем от вашего, придется выяснять, как использовать ваш код и что он делает. Этому человеку нужно будет понять следующее:

- ◆ в каких сценариях нужно вызывать различные функции, представленные в вашем коде;
- ◆ что представляют собой созданные вами классы и когда их следует использовать;
- ◆ какие значения чему присваивать;
- ◆ какие действия будет выполнять ваш код;
- ◆ какие значения может возвращать ваш код.

Через год вы, скорее всего, забудете подробности, связанные с вашим кодом. Вот почему смело можете рассматривать будущую версию себя как другого программиста. Чтобы понять, как использовать ваш код, другой программист может сделать следующее.

- ◆ Посмотреть на названия объектов (функций, классов, перечислений и т.д.).
- ◆ Посмотреть на типы данных объектов (типы параметров функций и конструкторов и типы возвращаемых значений).
- ◆ Прочитать документацию или комментарии на уровне функций/классов.
- ◆ Прийти и спросить вас лично или написать в чат/на почту.
- ◆ Изучить ваш код (основную логику реализации функций и классов, которые вы написали).

Как мы увидим в следующих подразделах, только первые три из них действительно практичны, и в рамках этих трех пунктов наименования объектов и типов данных обычно более надежны, чем документация.

3.2.1. Изучение названий объектов

На практике изучение названий объектов является одним из основных способов, с помощью которых программные инженеры выясняют, как использовать новый фрагмент кода. Названия пакетов, классов и функций немного похожи на оглавление книги: это удобный и быстрый способ найти код, который решит подзадачу. При использовании кода очень сложно игнорировать названия объектов: если функция называется `removeEntry()`, то ее трудно спутать с функцией, называемой `addEntry()`.

Поэтому правильные названия — один из лучших способов донести до коллег то, как следует использовать ваш код.

3.2.2. Анализ типов данных

При правильном выполнении анализ типов данных может быть очень надежным способом обеспечения правильного использования вашего кода. В любом скомпилированном, статически типизированном языке программисты должны обращать внимание и правильно использовать типы данных, иначе код просто не будет компилироваться. Вот почему необходимо установить, как можно использовать ваш код, при помощи системы типов. Это один из лучших способов убедиться, что другие программисты не смогут неправильно использовать или неправильно настраивать ваш код.

3.2.3. Чтение документации

Документация о том, как использовать ваш код, может существовать в нескольких формах, включающих следующее:

- ◆ неформальные комментарии на уровне функций и классов;
- ◆ более формальная документация в коде (например, `JavaDoc`);
- ◆ внешняя документация (например, `README.md`, веб-страница или документ с инструкциями).

Все это может быть чрезвычайно полезным, но лишь в некоторой степени надежным способом гарантировать, что другие будут знать, как правильно использовать ваш код.

- ◆ Нет никакой гарантии, что другие программисты прочтут документацию. На самом деле они нечасто это делают, а если и делают, то читают не полностью.
- ◆ Даже если они прочитают документацию, она может быть неправильно истолкована. Возможно, вы пользовались терминами, которые им незнакомы, или ошибочно предположили, что ваши коллеги знакомы с задачей, которую решает ваш код.

- ♦ Ваша документация может быть устаревшей. Программисты регулярно забывают обновлять документацию при внесении изменений в код, поэтому неизбежно то, что некоторая часть документации по коду будет устаревшей и неправильной.

3.2.4. Личный визит

Если вы работаете в команде, то вы, вероятно, обнаружите, что другие программисты часто задают вам вопросы о том, как использовать ваш код, и если код свеж в вашей памяти, то этот подход может быть весьма эффективным. Однако на это нельзя полагаться как на способ объяснить, как использовать фрагмент кода, по следующим причинам.

- ♦ Чем больше кода вы напишете, тем больше времени вы потратите на ответы на вопросы об этом коде. Рано или поздно у вас не останется времени, чтобы на все вопросы ответить.
- ♦ Через год вы сами, вероятно, забудете о своем коде, поэтому на самом деле существует только ограниченный период времени, в течение которого этот подход будет работать.
- ♦ Вы можете уйти в отпуск; тогда люди вообще не смогут вас ни о чем спросить.
- ♦ Вы можете уволиться, и тогда знания о том, как использовать код, будут для компании навсегда утеряны.

3.2.5. Изучение вашего кода

Если бы другой программист изучил основную логику реализации вашего кода, то он, вероятно, получил бы наиболее точный ответ о том, как его использовать, но этот подход не масштабируется и быстро становится непрактичным. Когда кто-то другой решает использовать ваш код в качестве зависимости, это, вероятно, всего лишь один из многих фрагментов кода, от которых зависит его работа. Если ему всегда приходится просматривать детали реализации каждой зависимости, чтобы понять, как их использовать, то в конечном итоге придется прочитать тысячи строк существующего кода каждый раз, когда происходит реализация функции.

И лучше не становится. Эти зависимости будут иметь свои собственные зависимости. Если бы каждый программист, работающий над кодовой базой, занимал позицию «Вам придется прочитать мой код, чтобы понять, как его использовать», тогда, вероятно, было бы необходимо прочитать реализации нескольких или даже всех подзависимостей, а их могут быть сотни. И тогда каждому программисту придется читать тысячи строк кода, только чтобы реализовать функцию среднего размера.

Смысл создания слоев абстракции заключается в том, чтобы программистам приходилось иметь дело только с несколькими концепциями одновременно и чтобы они могли использовать решение подзадачи без необходимости точно знать, как эта задача была решена. Требование читать детали реализации, чтобы знать, как использовать фрагмент кода, очевидно, сводит на нет многие преимущества уровней абстракции.

3.3. Соглашения по написанию кода

Вы, возможно, ранее сталкивались с термином *программирование по соглашению* (или *проектирование по соглашению*^❶). Это принцип, формализующий некоторые концепции, обсуждавшиеся в предыдущем разделе. Касаются они того, как другие смогут узнать, как использовать ваш код и что они могут ожидать от вашего кода. В соответствии с этой философией программисты думают о взаимодействии между различными частями кода, как если бы они были соглашением: вызывающие пользователи обязаны выполнять определенные обязательства, а взамен вызываемый код вернет желаемое значение или изменит некоторое состояние. Ничто не должно быть неясным или неожиданным, потому что все должно быть определено в этом соглашении.

❶ Термин «проектирование по соглашению» был впервые введен Бертраном Мейером в 1980-х годах и является центральной особенностью языка программирования и методологии Eiffel.

Программисты иногда считают полезным формально разделить условия соглашения с их кодом на разные категории.

- ◆ *Предварительные условия.* То, что должно быть истинным перед вызовом кода, например, в каком состоянии должна находиться система и какие входные данные должны быть предоставлены коду.
- ◆ *Постусловия.* То, что должно быть истинным после вызова части кода, например, система переводится в новое состояние или возвращаются определенные значения.
- ◆ *Инварианты.* Вещи, которые должны оставаться неизменными при сравнении состояния системы до и после вызова кода.

Даже если вы намеренно не программируете по соглашению или даже никогда не слышали об этом термине раньше, код, который вы пишете, почти наверняка будет иметь какое-то соглашение. Если вы написали функцию, которая имеет входные параметры, возвращает значение или изменяет состояние, то вы создали соглашение, потому что вы возлагаете на вызывающих пользователей обязательство что-то настроить или предоставить ввод (предварительное условие) и дать прогноз того, что произойдет или будет возвращено (постусловие).

Проблемы с кодовыми соглашениями возникают, когда инженеры не знают о некоторых или всех их условиях. Когда вы пишете код, важно подумать о таком соглашении и о том, как вы гарантируете, что любой использующий ваш код будет знать о нем и следовать ему.

3.3.1. Мелкий шрифт в соглашениях

В реальной жизни соглашения, как правило, состоят из вещей очевидных и вещей, которые напечатаны мелким шрифтом (и, следовательно, менее очевидны). Все знают, что необходимо читать мелкий шрифт каждого контракта, который подпи-

сываешь, но большинство людей этого не делают. Внимательно ли вы читаете каждое предложение каждого списка правил и условий, который вам встречается?

Чтобы продемонстрировать это различие между очевидными вещами и тем, что написано мелким шрифтом, давайте рассмотрим реальный пример соглашения в приложении для аренды электрического скутера. После регистрации и ввода номера вашей кредитной карты приложение позволяет вам найти скутер рядом с вами, забронировать его, прокатиться на нем, а затем завершить аренду, когда вы закончите. Экран, на котором вы бронируете аренду, выглядит как на рис. 3.2. Когда вы нажимаете кнопку ЗАРЕЗЕРВИРОВАТЬ, вы подписываете соглашение.

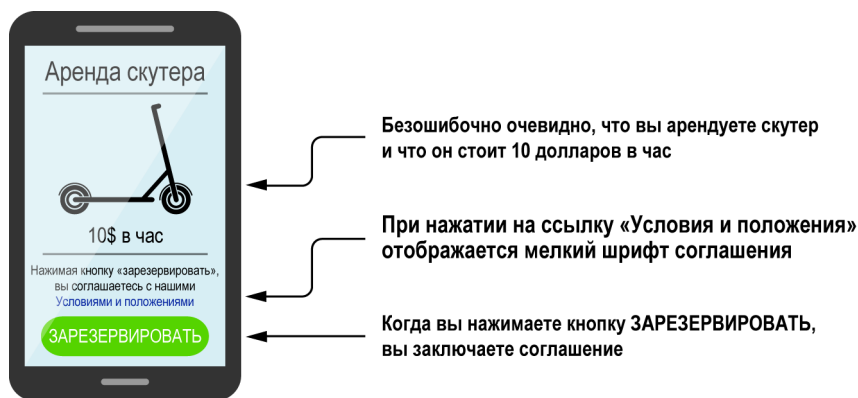


Рис. 3.2. Аренда электрического скутера с помощью приложения является реальным примером заключения соглашения. То, что вы арендуете, и стоимость являются безошибочными частями контракта

◆ **Безошибочные части контракта:**

- вы арендуете электрический скутер;
- аренда будет стоить 10 долларов в час.

◆ **Мелкий шрифт соглашения.** Если вы перейдете по ссылке «Условия и положения» и прочтете мелкий шрифт (рис. 3.3), вы обнаружите, что там написано следующее:

- если вы разобьете скутер, вам придется заплатить за это;
- вас оштрафуют на 100 долларов, если вы вывезете скутер за пределы города;
- если вы едете на скутере быстрее 30 миль в час, вы будете оштрафованы на 300 долларов, так как это повредит двигатель скутера. На скутере нет ограничителя скорости, и он может легко превысить эту скорость, поэтому пользователи несут полную ответственность за контроль своей скорости и отсутствие превышений.

Первые два термина, напечатанные мелким шрифтом, на самом деле не так уж удивительны; мы, вероятно, могли бы догадаться, что там будет что-то подобное.

С другой стороны, третий пункт о том, чтобы двигаться не быстрее 30 миль в час, потенциально является уловкой, и, если вы не прочтаете мелкий шрифт и не узнаете об этом, это может привести к неожиданному и изрядному штрафу.



Рис. 3.3. Соглашения обычно содержат мелкий шрифт, например, дополнительные пункты в условиях и положениях

◆ **Безошибочные части соглашения:**

- *названия функций и классов* — вызывающий пользователь не сможет использовать код, не зная их;
- *типы параметров* — код вообще не будет компилироваться, если вызывающий пользователь определит их неправильно;
- *типы возвращаемых данных* — вызывающий пользователь должен знать тип возвращаемой функции, чтобы иметь возможность ее использовать, и код, скорее всего, не будет компилироваться, если пользователь ошибется;
- *любые проверенные исключения (если язык их поддерживает)* — код не будет компилироваться, если вызывающий пользователь не знает о них или не разбирается в них.

◆ **Мелкий шрифт:**

- *комментарии и документация* — их необходимо прочитать, и к тому же полностью, так же как и мелкий шрифт в реальных соглашениях, но на самом деле люди редко это делают. Программисты должны относиться к этому факту прагматично;
- *любые непроверенные исключения* — если они перечислены в комментариях, значит, это мелкий шрифт. Иногда их не будет и там, если, например, функция, расположенная несколькими слоями ниже, выдаст что-то подобное, а автор функции, расположенной выше, забудет упомянуть об этом в своей документации.

Сделать условия кодового соглашения безошибочными намного лучше, чем полагаться на мелкий шрифт. Люди очень часто не читают мелкий шрифт, а даже если

и читают, то могут только бегло прочитать его и получить неверное представление. И, как обсуждалось в предыдущем разделе, документация имеет привычку устаревать, поэтому мелкий шрифт не всегда бывает корректен.

3.3.2. Не слишком полагайтесь на мелкий шрифт

Мелкий шрифт в виде комментариев и документации часто упускается из виду. Вот почему существует большая вероятность того, что другие программисты, использующие фрагмент кода, не будут полностью осведомлены обо всех вещах, которые указаны мелким шрифтом. Использование мелкого шрифта не является надежным способом донести соглашение по написанию кода. Чрезмерное использование мелкого шрифта, скорее всего, приведет к созданию хрупкого кода, который слишком легко неправильно использовать, а это приводит к сюрпризам, что, как мы установили в *главе 1*, являются врагами высококачественного кода.

Будут случаи, когда полагаться на мелкий шрифт придется; некоторые проблемы неизбежно будут содержать оговорки, которые требуют объяснения, или у нас может не быть другого выбора, кроме как зависеть от чьего-то плохого кода, что заставляет наш код делать что-то странное. В этих сценариях мы должны абсолютно четко написать документацию, чтобы объяснить это другим программистам, и сделать все возможное, чтобы побудить их прочитать ее. Но, несмотря на важность документации, все еще, к сожалению, существует высокая вероятность того, что наши коллеги не прочтут ее, или что со временем она устареет; так что документация не идеальна. В *главе 5* будут более подробно рассмотрены комментарии и документация. В общем, документировать вещи, которые в противном случае могли бы быть неясными, — это хорошая идея, но обычно лучше не слишком полагаться на то, что другие люди это прочитают. Если это возможно, то используйте безошибочно очевидные части кодового соглашения, чтобы прояснить ситуацию.

Чтобы это продемонстрировать, в следующем примере (листинг 3.1) показан код класса для загрузки и доступа к некоторым пользовательским настройкам. Он определяет соглашение о том, как должен использоваться класс, но он в значительной степени зависит от того, что вызывающие пользователи класса прочитали весь мелкий шрифт: после создания класса вызывающие пользователи должны вызвать функцию для загрузки некоторых настроек, а затем функцию инициализатора. Если они не смогут выполнить все эти действия в правильном порядке, то класс будет находиться в нерабочем состоянии.

Листинг 3.1. Код с большим количеством мелкого шрифта

```
class UserSettings {
    UserSettings() { ... }
    // Не вызывайте никаких других функций, пока настройки не будут
    // успешно загружены с помощью этой функции.
    // Возвращает значение true, если настройки были успешно загружены. Boolean
    loadSettings(File location) { ... }
```

```
// init() необходимо вызывать перед вызовом любых других функций, но
// только после загрузки настроек с помощью loadSettings().
void init() { ... }
// Возвращает выбранный пользователем цвет пользовательского интерфейса или значение
null, если не
// цвет выбран, или если настройки не были загружены или
// инициализированы.
Color? getUiColor() { ... } ❶
}
```

❶ Возврат значения `null` здесь может означать одно из двух: пользователь не выбрал цвет или класс не был полностью инициализирован.

Давайте подробно изложим соглашение.

◆ Безошибочные части соглашения:

- класс называется `UserSettings`, поэтому он, очевидно, содержит пользовательские настройки;
- `getUiColor()` почти наверняка возвращает цвет пользовательского интерфейса, выбранный пользователем. Она может возвращать либо цвет, либо значение `null`. Если не прочесть комментарии, будет некоторая двусмысленность в отношении того, что же означает нулевое значение `null`, но наиболее вероятным предположением может быть указание на то, что пользователь не выбрал цвет;
- `loadSettings()` принимает файл и возвращает логический тип `Boolean`. Даже не читая комментарий, можно предположить, что это значение `true` указывает на успех, а значение `false` указывает на неудачу.

◆ Мелкий шрифт:

- класс должен быть настроен с очень специфической серией вызовов функций: сначала необходимо вызвать функцию `loadSettings()`. Если она возвращает значение `true`, то необходимо вызвать функцию `init()`, и только после этого можно использовать класс;
- если `loadSettings()` возвращает `false`, то никакие другие функции в классе не должны вызываться;
- `getUiColor()`, возвращающий значение `null`, на самом деле может указывать на одну из двух вещей: что пользователь не выбрал цвет или что класс еще не настроен.

Это ужасное соглашение. Если программист, использующий этот класс, не очень внимательно прочитает весь мелкий шрифт, то есть большая вероятность, что он неправильно настроит этот класс. Если он неправильно настроит класс, то это может быть даже неочевидно, потому что функция `getUiColor()` перегружает значение `null` (и программист не узнает об этом, если не прочитает мелкий шрифт).

Чтобы продемонстрировать, почему это может быть проблемой, рассмотрим код в листинге 3.2. Если настройки пользователя не были правильно настроены перед

вызовом функции `setUiColor()`, то программа не выйдет из строя и сделает что-то почти разумное, но появится ошибка: мы проигнорируем выбранный пользователем цвет пользовательского интерфейса.

Листинг 3.2. Код с потенциальной ошибкой

```
void setUiColor(UserSettings userSettings) {
    Color? chosenColor = userSettings.getUiColor();
    if (chosenColor == null) {
        ui.setColor(DEFAULT_UI_COLOR); ❶
        return;
    }
    ui.setColor(chosenColor);
}
```

❶ Использует цвет по умолчанию, если `getColor()` возвращает значение `null`. Это может произойти, если пользователь не выбрал цвет или если класс `UserSettings` находится в недопустимом состоянии.

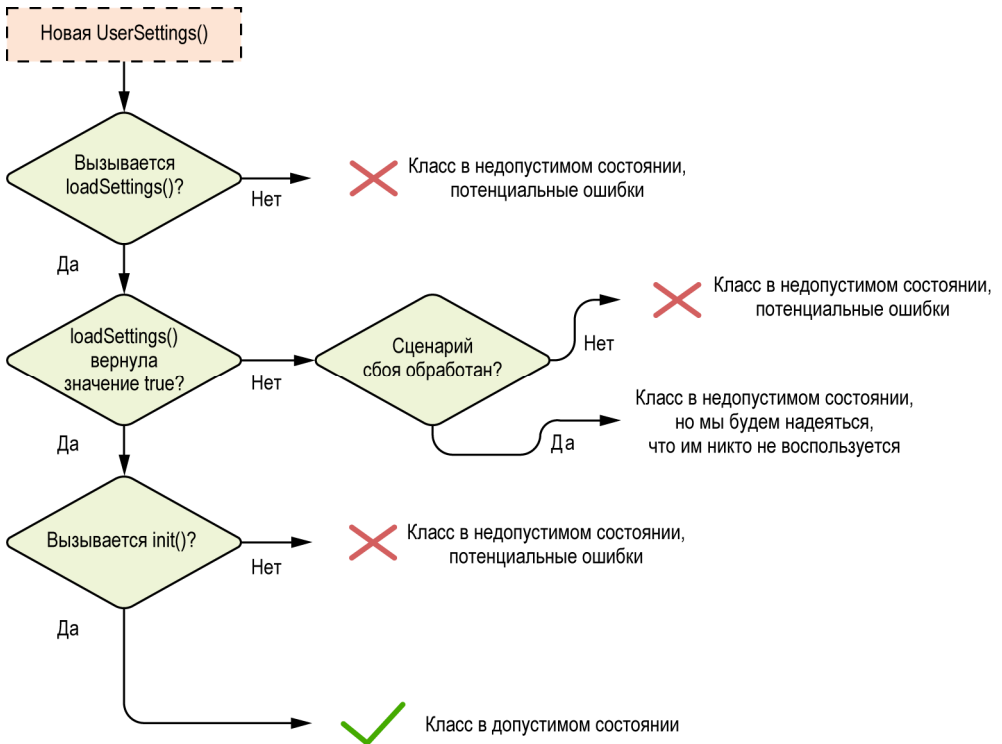


Рис. 3.4. Чем больше существует потенциальных способов неправильного использования фрагмента кода, тем больше вероятность его неправильного использования и появления ошибок в программном обеспечении

На рис. 3.4 перечислены все способы, которыми этот код может быть неправильно сконфигурирован и потенциально выдать ошибки. В настоящее время единственным средством предотвращения такого неправильного использования является мелкий шрифт, а он, как мы установили, не является надежным способом передачи кодового соглашения. Тогда создается высокая вероятность того, что ошибки будут проникать в программное обеспечение.

Как устранить мелкий шрифт

Мы видели ранее, что полагаться на мелкий шрифт ненадежно, потому что его слишком легко пропустить. В реальном примере соглашения на аренду скутера, приведенном выше, было бы лучше, если бы скорость более 30 миль в час была просто невозможна. Компания по производству скутеров могла бы установить ограничитель скорости на скутер, чтобы двигатель прекращал подачу питания всякий раз, когда скорость приближается к 30 милям в час, и снова начинал подавать питание только после снижения скорости. Если бы они так сделали, то не было бы необходимости в пункте мелким шрифтом, и это должно было бы полностью устранить проблему повреждения двигателей скутеров при превышении скорости.

Мы можем применить тот же принцип к коду: лучше просто сделать невозможным неправильное действие, чем полагаться на мелкий шрифт, чтобы попытаться убедиться, что другие разработчики правильно используют фрагмент кода. Часто можно изменить части кодового соглашения с мелкого шрифта на невозможный (или, по крайней мере, безошибочный), тщательно продумав, в какие состояния может попасть код или какие типы данных он принимает в качестве входных или возвращаемых. Цель состоит в том, чтобы гарантировать, что в случае неправильного использования или неправильной настройки кода он вообще не будет компилироваться.

Класс `UserSettings` может быть изменен для использования статической заводской функции, которая гарантирует, что можно получить только полностью инициализированный экземпляр класса. Это означает, что любой фрагмент кода в любом другом месте, использующий экземпляр `UserSettings`, гарантированно будет иметь полностью инициализированную версию. В примере ниже (листинг 3.3) класс `UserSettings` был изменен следующими способами.

- ◆ Добавлена статическая заводская функция `create()`. Это обрабатывает загрузку настроек и инициализацию и возвращает только экземпляр класса, который находится в допустимом состоянии.
- ◆ Конструктор был сделан закрытым, чтобы заставить внешний по отношению к классу код использовать функцию `create()`.
- ◆ Функции `loadSettings()` и `init()` были сделаны закрытыми, чтобы предотвратить их вызов внешним по отношению к классу кодом, который в противном случае мог бы перевести экземпляр класса в недопустимое состояние.
- ◆ Поскольку экземпляр класса теперь гарантированно находится в допустимом состоянии, функции `getUiColor()` больше не нужно перегружать нулевое значе-

ние `null`. Возвращаемое значение `null` теперь означает только то, что пользователь не указал цвет.

Листинг 3.3. Код почти без мелкого шрифта

```
class UserSettings {
    private UserSettings() { ... } ❶
    static UserSettings? create(File location) { ❷
        UserSettings settings = new UserSettings();
        if (!settings.loadSettings(location)) {
            return null; ❸
        }
        settings.init();
        return settings;
    }
    private Boolean loadSettings(File location) { ... } ❹
    private void init() { ... }
    // Возвращает выбранный пользователем цвет пользовательского интерфейса или значение
    null,
    // если он еще не выбран.
    Color? getUiColor() { ... } ❺
}
```

❶ Конструктор является закрытым. Это вынуждает программистов использовать вместо этого функцию `create()`.

❷ Вызов этой функции — единственный способ создать экземпляр `UserSettings`.

❸ Если загрузка настроек не удалась, возвращается значение `null`. Это предотвращает получение экземпляра этого класса, находящегося в недопустимом состоянии.

❹ Любые функции, которые изменяют состояние класса, являются закрытыми.

❺ Возвращаемое значение `null` теперь может означать только одно: пользователь не выбрал цвет.

Эти изменения успешно устранили почти весь мелкий шрифт из соглашения класса `UserSettings` и сделали невозможным создание экземпляра класса в недопустимом состоянии. Единственное, что осталось напечатать мелким шрифтом, — это объяснить, что означает возвращаемое из `getUiColor()` значение `null`, но даже это необязательно делать, так как большинство пользователей класса, скорее всего, и так догадываются, что означает `null`, и он больше не перегружен указанием на то, что класс находится в недопустимом состоянии.

На рис. 3.5 показано, как теперь можно использовать класс, в частности, как теперь невозможно получить экземпляр класса в недопустимом состоянии. Если вы уже немного знакомы с этим, то, возможно, заметили, что применяемая здесь техника заключается в том, чтобы исключить любое *состояние* или *изменчивость*, которые могут быть обнаружены за пределами класса.

СОСТОЯНИЕ И ИЗМЕНЧИВОСТЬ

Если вы не знакомы с терминами «состояние» и «изменчивость», то непременно будете к концу этой книги. Многие способы улучшения качества кода вращаются вокруг их минимизации. Состояние объекта относится к любым значениям или данным, хранящимся в нем. Объект является изменяемым, если возможно изменить любое из этих значений после создания объекта. И наоборот, если невозможно изменить какое-либо из этих значений после создания, то объект остается неизменным. Мы подробно обсудим это в *главе 7*.

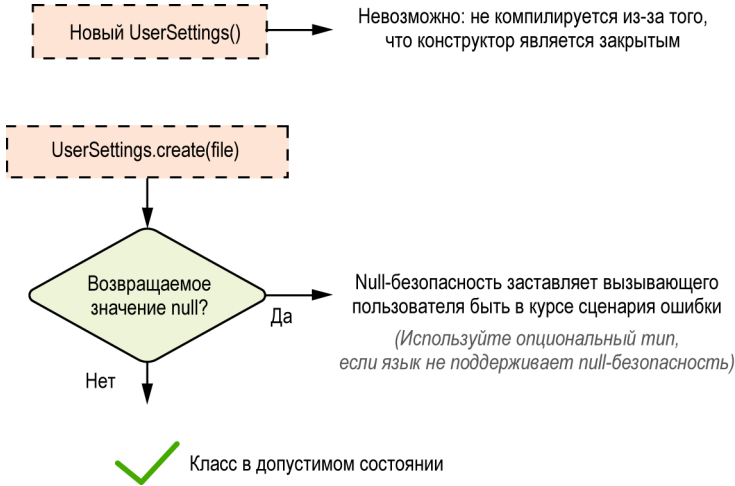


Рис. 3.5. Если код невозможно использовать неправильно, то гораздо ниже вероятность того, что ошибки проникнут в программное обеспечение, когда код понадобится другим программистам

Стоит отметить, что класс `UserSettings` все еще не идеален. Например, указание на то, что загрузка параметров завершилась неудачно, возвращая значение `null`, не делает код хорошо отлаживаемым; на самом деле было бы полезно получить информацию о причине сбоя. В следующей главе будут рассмотрены способы обработки ошибок, и мы увидим различные альтернативы, которые можно было бы использовать вместо этого.

Код в этом разделе является лишь одним примером того, как слишком много мелкого шрифта может привести к созданию некачественного кода, которым легко злоупотреблять и который приводит к сюрпризам. Существует множество других способов, с помощью которых слишком большой объем мелкого шрифта может сделать код хрупким и подверженным ошибкам, и мы рассмотрим некоторые из них в последующих главах.

3.4. Проверки и утверждения

Альтернативой использованию компилятора для принудительного выполнения кодового соглашения является использование принудительного выполнения во время

исполнения. Как правило, это не так надежно, как принудительное выполнение во время компиляции, поскольку обнаружение нарушения кодового соглашения будет зависеть от теста (или пользователя), столкнувшегося с проблемой во время исполнения кода. Это контрастирует с принудительным исполнением во время компиляции, что делает нарушение соглашения логически совершенно невозможным.

Тем не менее иногда бывают ситуации, когда нет практического способа принудительно выполнить соглашение с помощью компилятора. И, когда это происходит, выполнение соглашения с помощью проверки во время исполнения лучше, чем невыполнение соглашения.

3.4.1. Проверки

Распространенным способом обеспечения выполнения кодового соглашения является использование *проверок*. Проверки — это дополнительная логика, которая проверяет, соблюдено ли кодовое соглашение (например, ограничения на входные параметры или настройки, которые должны были быть выполнены). Если соглашение не соблюдено, то проверка выдает ошибку (или что-то подобное), которая приведет к очевидному и неизбежному сбою. (Проверки тесно связаны с *быстрой ошибкой*, которая обсуждается в следующей главе.)

Чтобы продолжить аналогию с электрическим скутером, добавление проверки было бы немного похоже на добавление отказоустойчивой прошивки скутера, что означает, что скутер полностью отключится, если гонщик разгонится до 30 миль в час. Затем гонщику придется остановиться, найти кнопку перезагрузки и подождать, пока скутер перезагрузится, прежде чем можно будет продолжать. Это предотвращает повреждение двигателя, но приводит к резкому выключению скутера, что в лучшем случае раздражает, а в худшем — опасно, если, например, гонщик находился в колонне движения на оживленной дороге. Это, пожалуй, лучше, чем повредить двигатель и быть оштрафованным на 300 долларов, но решение с использованием ограничителя скорости было бы еще лучше, потому что тогда возникновение опасной ситуации в принципе невозможно.

Проверки можно разделить на подкатегории в зависимости от того, какое условие соглашения они выполняют.

- ◆ *Проверки предварительных условий.* Например, проверка правильности входного аргумента, проверка выполнения какой-либо инициализации или, в более общем плане, проверка того, что система находится в допустимом состоянии перед запуском кода.
- ◆ *Проверки постусловий.* Например, проверка правильности возвращаемого значения или того, что система находится в допустимом состоянии после выполнения кода.

Мы видели, что класс `UserSettings` можно сделать менее подверженным ошибкам, сделав невозможной неправильную настройку. Альтернативой может быть использование проверок предварительных условий. Если бы автор класса сделал это, то код выглядел бы примерно так, как показано в листинге 3.4.

Листинг 3.4. Использование проверок для принудительного исполнения кодового соглашения

```
class UserSettings {
    UserSettings() { ... }
    // Не вызывайте никаких других функций, пока

    // настройки не будут успешно загружены с помощью этой функции.
    bool loadSettings(File location) { ... }
    // init() необходимо вызывать перед вызовом любых других функций, но
    // только после загрузки настроек с помощью loadSettings().
    void init() {
        if (!haveSettingsBeenLoaded()) {
            throw new StateException("Settings not loaded"); ❶
        }
        ...
    }
    // Возвращает выбранный пользователем цвет пользовательского интерфейса или значение
    null, если

    // он еще не выбран.
    Color? getUiColor() {
        if (!hasBeenInitialized()) {
            throw new StateException("Settings not initialized");
        }
        ...
    }
}
```

❶ Исключения возникают, если класс используется недопустимым образом.

По сравнению с исходным кодом с большим количеством мелкого шрифта это выглядит как улучшение, потому что меньше вероятности, что ошибки останутся незамеченными; если класс используется до настройки, произойдет громкая ошибка. Но увиденное нами решение, сделавшее неправильное использование невозможным, будет идеальным.

ПРОВЕРКИ НА РАЗНЫХ ЯЗЫКАХ

В листинге 3.4 реализована проверка предварительного условия пользовательским способом путем создания исключения `StateException`. Некоторые языки имеют встроенную поддержку и, следовательно, более приятный синтаксис для проверок, в то время как другие требуют более индивидуального подхода или использования сторонней библиотеки. Если вы решите использовать проверки, то обязательно найдите наилучший способ их реализации на любом языке, который вы используете.

Проверки дают нам надежду на то, что если кто-то неправильно использует код, это будет выявлено во время разработки или во время тестирования, и это будет замечено и исправлено до того, как код будет отправлен клиентам или использован в «дикой природе». Это намного лучше, чем программа, тихо переходящая в пло-

хое состояние, которое может проявляться только в виде странных ошибок, которые не сразу очевидны. Но эффективность проверок не гарантируется.

- ◆ Если проверяемое условие нарушается только в каком-то неясном сценарии, который никто не думал тестировать (или сценарии, который фаззинг не имитирует), то ошибка все равно может быть не обнаружена, до тех пор пока код не будет выпущен и не попадет в руки реальных пользователей.
- ◆ Несмотря на то что проверка привела к громкой ошибке, все еще существует риск, что этого никто не заметит. Исключения могут быть отслежены на более высоком уровне в программе и просто зарегистрированы, чтобы предотвратить полные сбои. Если программист, работающий над кодом, не потрудился проверить эти журналы, то он тоже может не заметить сбоя. Это признак довольно серьезной проблемы с методами разработки в команде (или обработкой исключений), но подобные вещи, к сожалению, случаются чаще, чем следовало бы.

Иногда наличие мелкого шрифта в кодовом соглашении неизбежно, и в этих случаях может быть хорошей идеей добавить проверки, чтобы убедиться, что соглашение соблюдается. Но лучше просто избегать мелкого шрифта, если это возможно. Если мы обнаружим, что добавляем много проверок к фрагменту кода, то это может быть признаком того, что вместо этого нам следует подумать о том, как вообще устранить мелкий шрифт.

Фаззинг — это тип тестирования, который пытается генерировать входные данные, которые могут выявить ошибки или неправильную конфигурацию в части кода или программного обеспечения. Например, если у нас есть программное обеспечение, которое принимает вводимые пользователем данные в виде строки, то фаззинг может генерировать множество различных случайных строк и предоставлять их в качестве входных данных одну за другой, чтобы увидеть, произойдет ли сбой или возникнет ли исключение. Если, например, строка, содержащая определенный символ, приводит к сбою программы, то мы надеемся, что фаззинг сможет это показать.

Если мы используем фаззинг, то включение проверок (или утверждений, см. следующий раздел) в наш код может помочь увеличить вероятность того, что этот тип тестирования выявит какие-либо неправильные настройки или ошибки, потому что фаззинг обычно основывается на ошибке или исключении и не будет обнаруживать менее очевидные ошибки, которые просто приводят к странному поведению.

3.4.2. Утверждения

Многие языки имеют встроенную поддержку *утверждений*. Утверждения концептуально очень похожи на проверки в том смысле, что они являются способом обеспечить соблюдение кодового соглашения. Когда код компилируется в режиме разработки или при выполнении тестов, утверждения ведут себя во многом так же, как и проверки: при нарушении условия будет выдана громкая ошибка или исключение. Ключевое различие между утверждениями и проверками заключается в том, что утверждения обычно компилируются, после того как код собран для выпуска, что означает, что при использовании кода в «дикой природе» не произойдет громкой ошибки.

Причина их компиляции при выпуске кода может быть двоякой.

- ◆ *Для повышения производительности.* Для вычисления того, было ли нарушено условие, очевидно, требуется несколько циклов процессора. Если у нас есть утверждение в фрагменте кода, который часто выполняется, то это может заметно снизить общую производительность части программного обеспечения.
- ◆ *Чтобы уменьшить вероятность сбоя кода.* То, действительно ли это является обоснованной мотивацией, будет зависеть от нашего конкретного приложения. Это увеличивает вероятность того, что ошибки останутся незамеченными, но если мы работаем над системой, в которой доступность важнее, чем предотвращение некоторых потенциально ошибочных действий, то это может быть верным компромиссом.

Обычно есть способ оставить утверждения включенными даже в готовых к выпуску сборках кода, и многие команды разработчиков так и делают. В этом случае утверждения на самом деле ничем не отличаются от проверок, за исключением некоторых подробностей о том, какие ошибки или исключения они могут вызывать. Если бы автор класса `UserSettings` использовал утверждения вместо проверок, то функция `getUiColor()` выглядела бы примерно так, как показано в следующем листинге.

Листинг 3.5. Использование утверждения для принудительного исполнения соглашения

```
class UserSettings {  
    ...  
    // Возвращает выбранный пользователем цвет интерфейса или значение null, если  
    // он еще не выбран.  
    Color? getUiColor() {  
        assert(hasBeenInitialized(), "Settings not initialized"); ❶  
        ...  
    }  
}
```

❶ Утверждение приведет к возникновению ошибки или исключения, если класс используется недопустимым образом.

То, что было сказано о проверках, также относится и к утверждениям: когда в соглашении по написанию нашего кода есть мелкий шрифт, его хорошо бы применять. Но еще лучше просто избегать мелкого шрифта.

Итоги

- ◆ Кодовые базы постоянно меняются, и обычно изменения вносят несколько программистов.
- ◆ Полезно подумать о том, как другие программисты могут сломать или неправильно использовать код, и спроектировать его таким образом, чтобы свести такую вероятность к минимуму или сделать это совершенно невозможным.

- ◆ Когда мы пишем код, мы неизменно создаем соглашение по написанию кода. Оно может содержать вещи, которые безошибочно очевидны, и вещи, которые больше похожи на мелкий шрифт.
- ◆ Мелкий шрифт в кодовых соглашениях не является надежным способом обеспечить соблюдение соглашения другими программистами. Обычно лучший подход — сделать вещи безошибочно очевидными.
- ◆ Принудительное выполнение соглашения с использованием компилятора, как правило, является наиболее надежным подходом. Когда это невозможно, альтернативой является принудительное выполнение контракта во время исполнения с помощью проверок или утверждений.

4 Ошибки

ЭТА ГЛАВА ОХВАТЫВАЕТ

- ◇ Различие между ошибками, от которых система может восстановиться, и теми, от которых она восстановиться не может.
- ◇ Быстрая ошибка и громкая ошибка.
- ◇ Различные методы сигнализации об ошибках и рекомендации, какие из них использовать.

Среда, в которой выполняется наш код, как правило, несовершенна: пользователи будут вводить неверные данные, внешние системы будут выходить из строя, а наш код и другой код вокруг него часто будут содержать некоторое количество ошибок. Учитывая это, ошибки неизбежны; все может пойти и пойдет не так, и в результате мы не сможем написать хороший и надежный код, тщательно не продумав возможные ошибки. Когда вы думаете об ошибках, часто бывает полезно различать ошибки, от которых часть программного обеспечения может захотеть избавиться, и те, от которых нет разумного способа восстановления. Эта глава начинается с изучения этого различия, прежде чем мы рассмотрим методы, которые можем использовать, чтобы ошибки не оставались незамеченными и обрабатывались надлежащим образом.

Разговор об ошибках и, в частности, о том, как их сигнализировать и обрабатывать, — это что-то вроде банки с червями, которую мы собираемся открыть. Многие инженеры-программисты и даже разработчики языков программирования придерживаются разных (иногда твердых) мнений о том, как код должен сигнализировать об ошибках и обрабатывать их. Во второй половине этой главы предпринята попытка дать достаточно широкий обзор основных техник, с которыми вы, вероятно, столкнетесь, и аргументов в пользу их использования. Хочу предупредить вас, что это большая и противоречивая тема, поэтому глава относительно длинная.

4.1. Возможность восстановления

Когда вы продумываете программное обеспечение, часто надо подумать о том, существует ли реальный способ восстановления после конкретного сценария ошибки. В этом разделе будет описано, что подразумевается под ошибкой, которая может или не может быть исправлена. Затем будет продолжено объяснение того, как это различие часто зависит от контекста. Это означает, что инженеры должны тща-

тельно продумать, как их код может быть использован при принятии решения о том, что делать в случае возникновения ошибки.

4.1.1. Ошибки, которые могут быть исправлены

Многие ошибки не являются фатальными для части программного обеспечения и существуют разумные способы их корректной обработки и восстановления. Очевидным примером этого является случай, когда пользователь предоставляет системе неверные входные данные (например, неверный номер телефона); было бы не очень удобно, если бы при вводе неверного номера телефона все приложение отказало (потенциально потеряв несохраненную работу). Вместо этого лучше просто предоставить пользователю приятное сообщение об ошибке, в котором говорится, что номер телефона недействителен, и попросить его ввести правильный.

В дополнение к таким вещам, как неверные пользовательские данные, другими примерами ошибок, от которых мы, вероятно, хотим, чтобы часть программного обеспечения была восстановлена, являются следующие.

- ◆ *Сетевые ошибки.* Если служба, от которой мы зависим, недоступна, то, возможно, лучше всего подождать несколько секунд и повторить попытку или попросить пользователя проверить свое сетевое подключение, если наш код выполняется на устройстве пользователя.
- ◆ *Некритическая ошибка задачи.* Например, если ошибка возникает в части программного обеспечения, которая просто регистрирует некоторую статистику использования, то, вероятно, можно продолжить выполнение.

Как правило, большинство ошибок, вызванных чем-то внешним по отношению к системе, — это вещи, которые система в целом, вероятно, должна попытаться исправить. Это связано с тем, что часто мы должны активно ожидать, что это произойдет: внешние системы и сети выйдут из строя, файлы будут повреждены, а пользователи (или хакеры) будут предоставлять неверные входные данные.

Обратите внимание, что это относится к системе в целом. Как мы увидим немного позже, низкоуровневый код часто не подходит для попыток восстановления после ошибок, и бывает необходимо сообщить об ошибке коду более высокого уровня, который знает, как следует обрабатывать ошибку.

4.1.2. Ошибки, которые не могут быть исправлены

Иногда возникают ошибки, и у системы нет разумного способа их исправить. Очень часто это происходит из-за ошибки программирования, когда инженер где-то что-то «напортачил». Примеры таких случаев включают следующие.

- ◆ Ресурс, который должен быть в комплекте с кодом, отсутствует.
- ◆ Некоторый код неправильно использует другой фрагмент кода, например:
 - вызов его с недопустимым входным аргументом;
 - не предварительная инициализация некоторого требуемого состояния.

Если нет мыслимого способа исправить ошибку, то единственное разумное, что может сделать фрагмент кода, — это попытаться ограничить ущерб и максимизировать вероятность того, что инженер заметит и устранит проблему. В *разд. 4.2* обсуждаются концепции *быстрой ошибки* и *громкая ошибки*, которые являются центральными для этого.

4.1.3. Часто только вызывающий код знает, можно ли исправить ошибку

Большинство типов ошибок проявляются, когда один фрагмент кода вызывает другой фрагмент кода. Поэтому при работе со сценарием ошибки важно тщательно продумать, какой другой код может вызывать наш код, и, в частности, следующее:

- ◆ Может ли вызывающий код потенциально захотеть восстановиться после ошибки?
- ◆ Если да, то как он узнает, что ему нужно обработать ошибку?

Код часто используется повторно и вызывается из нескольких мест, и если мы стремимся создать чистые слои абстракции, то, как правило, лучше всего делать как можно меньше предположений о потенциальных вызывающих пользователях нашего кода. Это означает, что, когда мы пишем или изменяем функцию, мы не всегда в состоянии знать, является ли состояние ошибки тем, которое можно или нужно восстановить.

Чтобы продемонстрировать это, рассмотрим листинг 4.1. Он содержит функцию, которая анализирует номер телефона из строки. Если строка является недопустимым номером телефона, то это представляет собой ошибку, но может ли фрагмент кода, вызывающий эту функцию (и программу в целом), реально восстановиться после этой ошибки?

Листинг 4.1. Анализ телефонного номера

```
class PhoneNumber {  
    ...  
    static PhoneNumber parse(String number) {  
        if (!isValidPhoneNumber(number)) {  
            ... some code to handle the error ... ❶  
        }  
        ...  
    }  
    ...  
}
```

- ❶ Может ли программа восстановиться после этого?

Ответ на вопрос о том, может ли программа восстановиться после этой ошибки, заключается в том, что мы не знаем, как используется эта функция и откуда она вызывается.

Если функция вызывается с жестко заданным значением, которое не является допустимым номером телефона, то это ошибка программирования. Это, скорее всего, не то, от чего программа может восстановиться. Представьте, что это используется в каком-то программном обеспечении для переадресации вызовов для компании, которая перенаправляет каждый звонок в головной офис; программа абсолютно никак не может восстановиться после этого:

```
PhoneNumber getHeadOfficeNumber() {  
    return PhoneNumber.parse("01234typo56789");  
}
```

И наоборот, если функция вызывается с предоставленным пользователем значением (как в следующем фрагменте) и этот ввод является недопустимым номером телефона, то, вероятно, это то, что программа может и должна восстановить. Было бы лучше всего показать в пользовательском интерфейсе красиво отформатированное сообщение об ошибке, информирующее пользователя о том, что номер телефона недействителен.

```
PhoneNumber getUserPhoneNumber(UserInput input) {  
    return PhoneNumber.parse(input.getPhoneNumber());  
}
```

Только вызывающая программа функции `PhoneNumber.parse()` может знать, в состоянии ли программа восстановить неверный номер телефона. В подобных сценариях автор такой функции, как `PhoneNumber.parse()`, должен предположить, что недопустимый номер телефона — это то, от чего абоненты могут захотеть избавиться.

В более общем плане если любое из нижеперечисленного верно, то ошибку, вызванную чем-либо, предоставленным функции, вероятно, следует рассматривать как нечто, от чего вызывающий код может захотеть избавиться:

- ◆ у нас нет точных (и полных) знаний о том, откуда может быть вызвана наша функция и откуда берутся значения, указанные в этих вызовах;
- ◆ существует даже самая малая вероятность того, что наш код может быть повторно использован в будущем, что означает, что наши предположения о том, откуда он вызывается, и происхождение любых значений могут стать недействительными.

Единственным реальным исключением из этого является то, что соглашение по написанию кода ясно указывает, что определенный ввод недопустим, и у вызывающего есть простой и очевидный способ проверить ввод перед вызовом функции. Примером этого может быть инженер, вызывающий средство получения списка с отрицательным индексом (на языке, который это не поддерживает); должно быть очевидно, что отрицательный индекс будет недействительным, и у вызывающего есть простой и очевидный способ проверить это перед вызовом функции, если есть риск, что индекс может быть отрицательным.

Для подобных сценариев мы, вероятно, можем с уверенностью предположить, что это ошибка программирования, и рассматривать ее как нечто, от чего невозможно избавиться. Но все же полезно понимать: то, что может показаться нам очевидным в отношении того, как следует использовать наш код, может быть неочевидным для

других. Если тот факт, что определенный ввод недействителен, глубоко скрыт в мелком шрифте кодового соглашения, то другие инженеры, скорее всего, пропустят его.

Определение того, что вызывающие могут захотеть восстановиться после ошибки, — это замечательно и хорошо, но если они даже не знают, что ошибка может произойти, то они вряд ли справятся с ней должным образом. В следующем разделе это объясняется более подробно.

4.1.4. Информируйте вызывающих об ошибках, которые они могут захотеть исправить

Когда другой код вызывает наш код, у него часто не будет практического способа заранее узнать, что его вызов приведет к ошибке. Например, то, что является или не является действительным номером телефона, может быть довольно сложной задачей для определения. «01234tyro56789» может быть недействительным номером телефона, но «1-800-I-LOVE-CODE» может быть совершенно действительным, что означает, что правила, определяющие это, сложны.

В предыдущем примере номера телефона (повторенном в листинге 4.2) класс `PhoneNumber` обеспечивает слой абстракции для работы с входами и выходами телефонных номеров; абоненты защищены от деталей реализации и, следовательно, сложности правил, которые определяют, что является/не является допустимым номером телефона. Поэтому было бы неразумно ожидать, что вызывающие абоненты будут вызывать `PhoneNumber.parse()` только с допустимыми входными данными, потому что весь смысл класса `PhoneNumber` состоит в том, чтобы не позволять вызывающим абонентам беспокоиться о правилах, которые это определяют.

Листинг 4.2. Анализ телефонного номера

```
class PhoneNumber { ❶
    ...
    static PhoneNumber parse(String number) {
        if (!isValidPhoneNumber(number)) {
            ... some code to handle the error ...
        }
        ...
    }
    ...
}
```

❶ Слой абстракции для телефонных номеров.

В дополнение к этому, поскольку вызывающие `PhoneNumber.parse()` не являются экспертами по телефонным номерам, они могут даже не осознавать, что концепция недействительности телефонного номера существует, или даже если они это сделают, они могут не ожидать, что проверка произойдет в этот момент. Они могут ожидать, что это произойдет, например, только при наборе номера.

Поэтому автор функции `PhoneNumber.parse()` должен убедиться, что вызывающие осведомлены о возможности возникновения ошибки. Невыполнение этого требования может привести к неожиданностям, когда ошибка действительно произойдет и никто не написал никакого кода для ее обработки. Это может привести к заметным пользователям ошибкам или сбоям в критически важной для бизнеса логике. Разделы 4.3 и 4.5 подробно описывают, как мы можем гарантировать, что вызывающий знает, что может произойти ошибка.

4.2. Надежность в сравнении с отказом

Когда возникает ошибка, часто приходится выбирать между сбоем, который может повлечь за собой либо обработку ошибки на более высоком уровне кода, либо сбой всей программы, и попыткой разобраться с ошибкой и продолжить работу.

Продолжение иногда может сделать код более надежным, но это также может означать, что ошибки остаются незамеченными и начинают происходить странные вещи. В этом разделе объясняется, почему отказ часто может быть лучшим вариантом и как надежность может быть встроена на соответствующих уровнях логики.

4.2.1. Быстрая ошибка

Представьте, что мы занимаемся поиском редких диких трюфелей и продаем их в элитные рестораны. Мы хотим купить собаку, которая поможет нам находить трюфели, вынюхивая их. У нас есть два варианта.

1. Собака, которая обучена останавливаться и лаять, как только обнаружит трюфель. Всякий раз, когда она это делает, мы смотрим, куда направлен ее нос, копаем, и — вуаля — мы нашли трюфель.
2. Собака, которая, обнаружив трюфель, замолкает, проходит 10 и более метров в случайном направлении и только потом начинает лаять.

Какую из этих собак мы должны выбрать? Охота на ошибки в коде немного похожа на охоту за трюфелями с собакой; в какой-то момент код будет «лаять» на нас, проявляя плохое поведение или выдавая ошибку. Мы узнаем, где код начал «лаять»: либо там, где мы увидели плохое поведение, либо номер строки в трассировке стека. Но если «лай» не начинается где-то рядом с фактическим источником ошибки, то это не очень полезно.

Быстрая ошибка заключается в том, чтобы сигнализировать об ошибке как можно ближе к реальному местоположению проблемы. При ошибке, от которой можно избавиться, это дает вызывающему пользователю максимальную вероятность того, что он сможет легко и безопасно восстановиться после нее; при ошибке, от которой невозможно избавиться, это дает инженерам максимальную возможность быстро идентифицировать и устранить проблему. В обоих случаях это также предотвращает попадание части программного обеспечения в непреднамеренное и потенциально опасное состояние.

Распространенным примером этого является случай, когда функция вызывается с недопустимым аргументом. Быстрая ошибка означала бы выдачу ошибки, как только эта функция будет вызвана с неверным вводом, в отличие от продолжения работы только для того, чтобы обнаружить, что неверный ввод вызывает проблему где-то еще в коде некоторое время спустя.

На рис. 4.1 показано, что может произойти, если код не завершится быстро: ошибка может проявиться только вдали от фактического местоположения ошибки, и это может потребовать значительных инженерных усилий для работы в обратном направлении через код, чтобы найти и исправить ошибку.

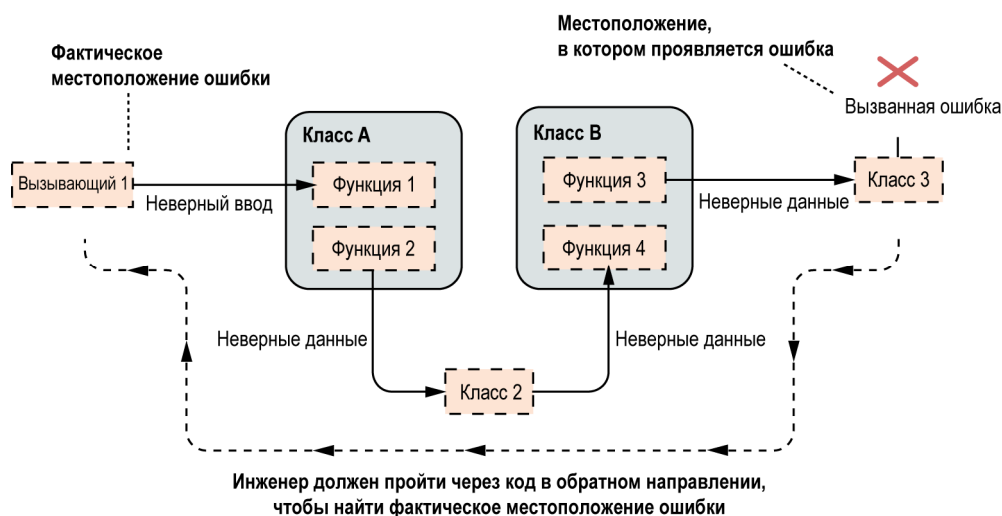


Рис. 4.1. Если код не завершается быстро при возникновении ошибки, то ошибка может проявиться намного позже в каком-либо коде, удаленном от фактического местоположения ошибки. Это может потребовать значительных инженерных усилий для отслеживания и устранения проблемы

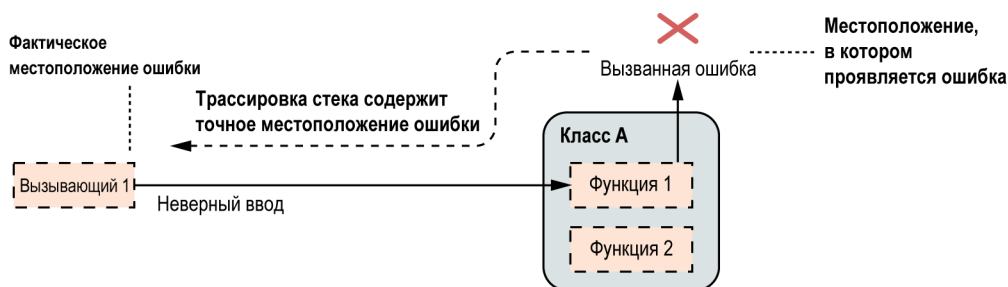


Рис. 4.2. Если при возникновении ошибки код быстро выходит из строя, то точное местоположение ошибки обычно становится очевидным сразу

Напротив, на рис. 4.2 показано, как быстрая ошибка может значительно улучшить ситуацию. При быстром сбое ошибка проявится рядом с ее фактическим местопо-

ложением, и трассировка стека часто будет указывать точный номер строки в коде, где ее можно найти.

В дополнение к усложнению отладки отсутствие быстрого сбоя может привести к тому, что код будет хромать и потенциально может привести к повреждению. Примером этого может быть сохранение некоторых поврежденных данных в базе данных: ошибка, которая может быть замечена только несколько месяцев спустя, к тому времени многие важные данные могут быть уничтожены навсегда.

Как и в случае с собакой и трюфелем, гораздо полезнее, если код «лает» как можно ближе к реальному источнику проблемы. Если ошибка не может быть устранена, то также важно убедиться, что код действительно «лает» (и «лает» громко), когда есть ошибка, как обсуждается в следующем разделе; это известно как *громкая ошибка*.

4.2.2. Громкая ошибка

Если возникает ошибка, от которой программа не может оправиться, то, скорее всего, это ошибка, вызванная ошибкой программирования или какой-либо ошибкой, допущенной инженером. Очевидно, что мы не хотим подобной ошибки в программном обеспечении и, скорее всего, хотим ее исправить, но мы не сможем ее исправить, если сначала не узнаем об этом.

Громкая ошибка — это просто гарантия того, что ошибки не останутся незамеченными. Самый очевидный (и жестокий) способ сделать это — завершить работу программы, создав исключение (или аналогичное). Альтернативой является регистрация сообщения об ошибке, хотя иногда они могут игнорироваться в зависимости от того, насколько старательно инженеры проверяют их и сколько других шумов содержится в журналах. Если код выполняется на устройстве пользователя, то мы можем захотеть отправить сообщение об ошибке обратно на сервер, чтобы зарегистрировать произошедшее (конечно, если у нас есть разрешение пользователя на это).

Если код быстро и громко выходит из строя, то есть большая вероятность, что ошибка будет обнаружена во время разработки или во время тестирования (еще до того как код будет выпущен).

Даже если это не так, мы, скорее всего, начнем видеть отчеты об ошибках довольно скоро после и сможем точно узнать, где произошла ошибка в коде, просмотрев отчет.

4.2.3. Объем восстанавливаемости

Область, в пределах которой что-то может быть восстановлено или не может быть восстановлено, может варьироваться. Например, если мы пишем код, который выполняется на сервере, обрабатывающем запросы от клиентов, отдельный запрос может вызвать путь кода с неполадкой в нем, которая вызывает ошибку. Возможно, не существует разумного способа восстановления в рамках обработки этого запроса, но это может не гарантировать сбой всего сервера. В этом случае ошибка

не может быть восстановлена в рамках этого запроса, но может быть восстановлена сервером в целом.

Как правило, полезно попытаться сделать программное обеспечение надежным; сбой всего сервера из-за одного неправильного запроса, вероятно, не будет хорошей идеей. Но также важно убедиться, что ошибки не останутся незамеченными, поэтому код должен завершаться громкой ошибкой. Между этими двумя целями часто существует дихотомия. Самый громкий способ потерпеть неудачу — это сбой программы, но это, очевидно, делает программное обеспечение менее надежным.

Решение этой дихотомии состоит в том, чтобы гарантировать, что в случае обнаружения ошибок программирования они регистрируются и отслеживаются таким образом, чтобы инженеры могли их заметить.

Обычно это включает в себя регистрацию подробной информации об ошибках, чтобы инженер мог отладить то, что произошло, и убедиться, что частота ошибок отслеживается, а инженеры предупреждаются, если частота ошибок становится слишком высокой (рис. 4.3 иллюстрирует это).

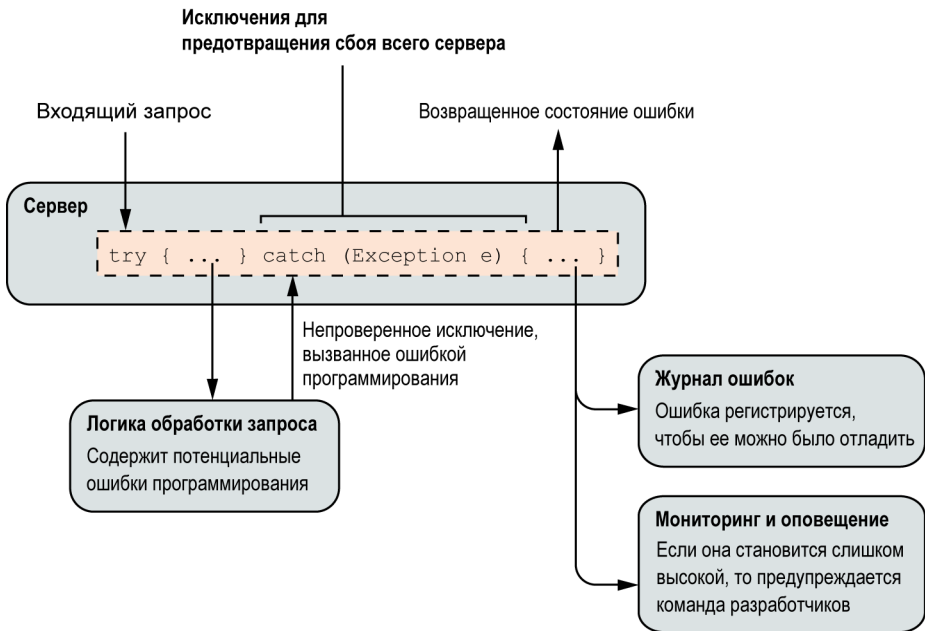


Рис. 4.3. На сервере при обработке одного запроса может возникнуть ошибка программирования. Поскольку запросы являются независимыми событиями, возможно, лучше не приводить к сбою весь сервер, когда это произойдет. Ошибка не может быть восстановлена в рамках одного запроса, но может быть восстановлена сервером в целом

СЕРВЕРНЫЕ ФРЕЙМВОРКИ

Большинство серверных фреймворков содержат встроенные функции для изоляции ошибок для отдельных запросов и сопоставления определенных типов ошибок

с различными ответами на ошибки и обработкой. Поэтому маловероятно, что нам придется писать собственный оператор `try-catch`, но что-то концептуально похожее на это внутри серверного фреймворка происходить будет.

Должен предупредить, этот метод обнаружения всех типов ошибок и последующего их протоколирования вместо передачи сигналов о них на более высокий уровень в программе следует применять с особой осторожностью. Часто в программе есть только несколько мест (если таковые имеются), где это уместно, например, точки входа очень высокого уровня в код или ветви логики, которые действительно независимы или некритичны к правильному функционированию остальной части программы. Как мы увидим в следующем разделе, обнаружение и регистрация ошибок (вместо того чтобы сигнализировать о них) могут привести к их скрытию, что может вызвать проблемы.

4.2.4. Не скрывайте ошибки

Как мы только что видели, надежность может быть обеспечена путем изоляции независимых или некритичных частей кода, чтобы гарантировать, что они не приведут к сбою всего программного обеспечения. Обычно это нужно делать осторожно, умеренно и в достаточно высокоуровневом коде. Обнаружение ошибок в несамостоятельных, критических или низкоуровневых частях кода, а затем продолжение работы независимо от этого часто может привести к тому, что программное обеспечение не будет должным образом выполнять то, для чего оно предназначено. И если ошибки должным образом не регистрируются или не сообщаются, то проблемы могут остаться незамеченными командой инженеров.

Иногда может показаться заманчивым просто скрыть ошибку и притвориться, что ее никогда не было. Это может сделать код намного проще и позволит избежать нагрузки на громоздкую обработку ошибок, но это почти никогда не бывает хорошей идеей. Скрытие ошибок проблематично как для ошибок, которые можно исправить, так и для ошибок, которые исправить невозможно:

- ◆ Скрытие ошибки, от которой вызывающий код может захотеть избавиться, лишает его возможности изящно восстановиться после нее. Вместо того чтобы отображать точное и значимое сообщение об ошибке или возвращаться к какому-либо другому поведению, он вместо этого совершенно не осознает, что что-то пошло не так, что означает, что программное обеспечение, скорее всего, не будет делать то, для чего оно предназначено
- ◆ Скрытие ошибки, которую невозможно исправить, скорее всего, скрывает ошибку программирования. Как было установлено в предыдущих подразделах о быстром и громком сбое, это ошибки, о которых действительно нужно знать команде разработчиков, чтобы их можно было исправить. Скрытие их означает, что команда разработчиков может никогда о них не узнать, и программное обеспечение будет содержать ошибки, которые могут оставаться незамеченными в течение довольно долгого времени.
- ◆ В обоих сценариях, если возникает ошибка, это обычно означает, что код не способен выполнить то, что ожидал от него вызывающий. Если код попытается

скрыть ошибку, то вызывающий будет считать, что все работало нормально, хотя на самом деле это не так. Код, скорее всего, будет хромать, но затем выведет неверную информацию, повредит некоторые данные или в конечном итоге выйдет из строя.

В следующих нескольких подразделах рассматриваются некоторые способы, с помощью которых код может скрыть тот факт, что произошла ошибка. Некоторые из этих методов полезны в других сценариях, но, когда дело доходит до обработки ошибок, все они, как правило, являются плохой идеей.

Возврат значения по умолчанию

Когда возникает ошибка и функция не может вернуть желаемое значение, иногда может показаться, что проще вернуть значение по умолчанию. Альтернатива добавления кода для правильной сигнализации и обработки ошибок может показаться по сравнению с этим большим усилием. Проблема со значениями по умолчанию заключается в том, что они скрывают тот факт, что произошла ошибка, а это означает, что вызывающие код, скорее всего, будут продолжать работать так, как будто все в порядке.

В листинге 4.3 содержится некоторый код для поиска баланса на счете клиента. Если при доступе к хранилищу учетных записей возникает ошибка, функция возвращает нулевое значение по умолчанию. Возврат нулевого значения по умолчанию скрывает тот факт, что произошла ошибка, и делает ее неотличимой от сценария, когда у клиента действительно нулевой баланс. Если бы клиент с кредитным балансом в 10 000 долларов вошел в систему в один прекрасный день и обнаружил, что его баланс отображается как ноль, он, вероятно, взбесился бы. Было бы лучше сообщить об ошибке вызывающему коду, чтобы вместо этого он мог отобразить сообщение об ошибке пользователю со словами: «Извините, мы не можем получить доступ к этой информации прямо сейчас».

Листинг 4.3. Возврат значения по умолчанию

```
class AccountManager {
    private final AccountStore accountStore;
    ...

    Double getAccountBalanceUsd(Int customerId) {
        AccountResult result = accountStore.lookup(customerId);
        if (!result.success()) {
            return 0.0; ❶
        }
        return result.getAccount().getBalanceUsd();
    }
}
```

❶ При возникновении ошибки возвращается нулевое значение по умолчанию.

Могут быть сценарии, в которых значения по умолчанию в коде могут быть полезны, но они почти никогда не подходят, когда дело доходит до устранения ошибок. Они нарушают принципы быстрого и громкого сбоя, потому что они заставляют систему хромать с неверными данными и означают, что ошибка проявится каким-то странным образом позже.

Шаблон нулевого объекта

Нулевой (`null`) объект концептуально аналогичен значению по умолчанию, но расширяет идею, чтобы охватить более сложные объекты (например, классы). Нулевой объект будет выглядеть как подлинное возвращаемое значение, но все его функции-члены либо ничего не будут делать, либо вернут значение по умолчанию, которое должно быть безобидным.

Примеры шаблона нулевого объекта могут варьироваться от простого возврата пустого списка до такого сложного, как реализация целого класса. Здесь мы просто сосредоточимся на примере пустого списка.

Листинг 4.4 содержит функцию поиска всех неоплаченных накладных для клиента. Если запрос в хранилище накладных `InvoiceStore` завершается неудачно, функция возвращает пустой список. Это может легко привести к ошибкам в программном обеспечении. Клиент может задолжать тысячи долларов по неоплаченным счетам, но если `InvoiceStore` не работает в день проведения аудита, то ошибка заставит вызывающего поверить, что у клиента нет неоплаченных накладных.

Листинг 4.4. Возврат пустого списка

```
class InvoiceManager {
    private final InvoiceStore invoiceStore;
    ...

    List<Invoice> getUnpaidInvoices(Int customerId) {
        InvoiceResult result = invoiceStore.query(customerId);
        if (!result.success()) {
            return []; ❶
        }
        return result
            .getInvoices()
            .filter(invoice -> !invoice.isPaid());
    }
}
```

❶ При возникновении ошибки возвращается пустой список.

Шаблон нулевого объекта более подробно рассматривается в *главе 6*. Что касается шаблонов проектирования, то это своего рода палка о двух концах; есть несколько сценариев, в которых он может быть весьма полезен, но, как показывает предыдущий пример, когда дело доходит до обработки ошибок, его часто не рекомендуется использовать.

Ничегонеделание

Если рассматриваемый код что-то делает (а не возвращает что-то), то один из вариантов — просто не сигнализировать о том, что произошла ошибка. Как правило, это плохо, так как вызывающие абоненты будут считать, что задача, для выполнения которой предназначался код, выполнена. Это, скорее всего, приведет к несоответствию между ментальной моделью инженера того, что делает код, и тем, что он делает в реальности. Это может вызвать неожиданности и привести к ошибкам в программном обеспечении.

В листинге 4.5 содержится некоторый код для добавления товара в `MutableInvoice`. Если добавляемый товар имеет цену в валюте, отличной от валюты `MutableInvoice`, то это ошибка, и товар не будет добавлен в накладную. Код не делает ничего, чтобы сигнализировать о том, что произошла эта ошибка и что элемент не был добавлен. Это, скорее всего, приведет к ошибкам в программном обеспечении, так как любой, вызывающий функцию `addItem()`, будет ожидать, что товар был добавлен в накладную.

Листинг 4.5. Ничегонеделание при возникновении ошибки

```
class MutableInvoice {  
    ...  
    void addItem(InvoiceItem item) {  
        if (item.getPrice().getCurrency() !=  
            this.getCurrency()) {  
            return; ❶  
        }  
        this.items.add(item);  
    }  
    ...  
}
```

❶ Если есть несоответствие в валютах, функция возвращает значение.

Предыдущий сценарий является примером того, как не сигнализировать об ошибке. Другой сценарий, с которым мы можем столкнуться, — это код, который активно подавляет ошибку, о которой сигнализирует другой фрагмент кода. В листинге 4.6 показано, как это может выглядеть. Вызов службы `emailService.sendPlainText()` может привести к `EmailException`, если при отправке сообщения электронной почты возникает ошибка. Если это исключение возникает, то код подавляет его и ничего не сигнализирует вызывающему, чтобы указать, что действие не удалось. Это, скорее всего, приведет к ошибкам в программном обеспечении, так как вызывающий эту функцию будет считать, что электронное письмо было отправлено, хотя на самом деле этого могло и не быть.

Листинг 4.6. Подавление исключения

```
class InvoiceSender {  
    private final EmailService emailService;  
    ...  
}
```

```

void emailInvoice(String emailAddress, Invoice invoice) {
    try {
        emailService.sendPlainText(
            emailAddress,
            InvoiceFormat.plainText(invoice));
    } catch (EmailException e) { } ❶
}

```

❶ `EmailException` перехватывается, а затем полностью игнорируется.

Небольшое улучшение этого было бы, если бы ошибка регистрировалась при возникновении сбоя (листинг 4.7), но это все равно почти так же плохо, как исходный код в листинге 4.6. Это небольшое улучшение, потому что, по крайней мере, инженер смог бы заметить эти ошибки, если бы он посмотрел в журналах. Но код по-прежнему скрывает ошибку от вызывающего абонента, что означает, что вызывающий будет считать, что электронное письмо было отправлено, хотя на самом деле это не так.

Листинг 4.7. Обнаружение исключения и регистрация ошибки

```

class InvoiceSender {
    private final EmailService emailService;
    ...

    void emailInvoice(String emailAddress, Invoice invoice) {
        try {
            emailService.sendPlainText(
                emailAddress,
                InvoiceFormat.plainText(invoice));
        } catch (EmailException e) {
            logger.logError(e); ❶
        }
    }
}

```

❶ `EmailException` регистрируется.

Будьте осторожны с тем, что регистрируется

Еще одна вещь, которая должна заставить нас нервничать по поводу кода в листинге 4.7, заключается в том, что `EmailException` может содержать адрес электронной почты, который может содержать личную информацию пользователя и попадать под действие определенной политики обработки данных. Запись адреса электронной почты в журнал ошибок может нарушить политику обработки данных.

Как показывают примеры, скрывать ошибки почти никогда не бывает хорошей идеей. Если бы у компании был код из нескольких предыдущих списков в их кодовой базе, у них, вероятно, было бы много неоплаченных накладных и нездоровый

баланс. Скрытие ошибок может иметь реальные (а иногда и серьезные) последствия. Лучше сигнализировать, когда произошла ошибка, и в следующем разделе рассказывается, как это сделать.

4.3. Способы сигнализации об ошибках

Когда возникает ошибка, обычно необходимо сообщить об этом на какой-то более высокий уровень в программе. Если невозможно восстановиться после ошибки, то это обычно означает, что какой-то гораздо более высокий уровень в программе прерывает и регистрирует ошибку или, возможно, даже прекращает выполнение всей программы. Если потенциально возможно исправить ошибку, это обычно означает, что необходимо сообщить об этом непосредственному вызывающему (или, возможно, вызывающему на один или два уровня выше по цепочке вызовов), чтобы они могли корректно ее обработать.

Существует несколько способов сделать это, и варианты, которые у нас есть для этого, будут зависеть от того, какие функции обработки ошибок поддерживает используемый нами язык. Вообще говоря, способы сигнализации об ошибке делятся на две категории.

- ◆ *Явный.* Непосредственно вызывающий нашего кода вынужден осознавать, что может произойти ошибка. Будет ли он затем обрабатывать ошибку, передавать следующему вызывающему или просто игнорировать, зависит от него. Но что бы он ни делал, это будет активный выбор, и он почти никак не может не обращать на это внимания: возможность возникновения ошибки содержится в безошибочной части нашего соглашения по написанию кода.
- ◆ *Неявный.* Ошибка сигнализируется, но вызывающие наш код могут не обращать на нее внимания. Чтобы вызывающий абонент знал, что ошибка может произойти, часто требуются активные усилия, такие, как чтение документации или кода. Если ошибка упоминается в документации, то это часть мелкого шрифта соглашения по написанию кода. Иногда ошибка здесь даже не упоминается, и в этом случае она вообще не является частью письменного соглашения.

Обратите особое внимание: в этой классификации мы имеем в виду, является ли возможность возникновения ошибки явной или неявной с точки зрения инженера, использующего фрагмент кода. Речь идет не о том, приведет ли ошибка к тому, что материал в конечном итоге выйдет из строя громко или тихо, речь идет о том, чтобы абоненты знали о сценариях, о которых им необходимо знать (с помощью явных методов), и о том, что они не обременены необходимостью обрабатывать сценарии, для которых они не могут сделать ничего разумного (с помощью неявных методов).

В табл. 4.1 перечислены некоторые примеры явных и неявных методов сигнализации об ошибках.

Таблица 4.1. Явные и неявные методы сигнализации об ошибках

Местоположение в соглашении по написанию кода	Явные методы сигнализации об ошибках	Неявные методы сигнализации об ошибках
	Безошибочная часть	Мелким шрифтом, если задокументировано; в противном случае даже не мелким шрифтом
Вызывающий абонент знает, что может произойти ошибка?	Да	Вероятно
Примеры методов	Проверенное исключение	Непроверенное исключение
	Nullable тип возвращаемого значения, допускающий значение null (если null безопасно)	Возврат магического значения (следует избегать)
	Optional тип возвращаемого значения	Обещание или будущее
	Result тип возвращаемого значения	Утверждение
	Outcome тип возвращаемого значения (в случае принудительной проверки возвращаемого значения)	Проверка (в зависимости от реализации)
	Быстрые ошибки	Паника

В следующих подразделах будут рассмотрены некоторые из методов, перечисленных в табл. 4.1, с примерами их использования и объяснениями того, почему они являются явными или неявными методами.

4.3.1. Резюме: исключения

Во многих языках программирования существует понятие *исключений*. Они предназначены для того, чтобы часть кода сигнализировала об ошибке или исключительных обстоятельствах. Когда возникает исключение, стек вызовов разматывается до тех пор, пока не встретится вызывающий объект, обрабатывающий исключение, либо до тех пор, пока стек вызовов не будет полностью размотан, после чего программа завершит работу и выведет сообщение об ошибке.

Реализация исключения, как правило, является полноценным классом. В языках обычно есть несколько готовых, которые мы можем использовать, но мы также можем определить свои собственные и инкапсулировать в них информацию об ошибке.

В Java существует концепция как *проверенных исключений*, так и *непроверенных исключений*. Большинство основных языков, поддерживающих исключения, имеют только понятие непроверенных исключений, поэтому, когда речь идет практически о любом языке, кроме Java, слово *исключение* обычно подразумевает *непроверенное исключение*.

4.3.2. Явный: проверенные исключения

С проверенным исключением компилятор заставляет вызывающую сторону признать, что это может произойти, либо написав код для его обработки, либо объявив, что исключение может быть выдано в их собственной сигнатуре функции. Таким образом, использование проверенных исключений является явным способом сигнализации об ошибке.

Сигнализация с использованием проверенного исключения

Чтобы продемонстрировать и сравнить различные методы сигнализации об ошибках, мы будем использовать пример функции, которая вычисляет квадратный корень из числа. Всякий раз, когда функция получает отрицательное число в качестве входных данных, это представляет собой ошибку, о которой необходимо каким-то образом сигнализировать. Очевидно, что в большинстве языков уже есть встроенный способ вычисления квадратного корня, поэтому мы, вероятно, не стали бы писать собственную функцию для этого в реальной жизни, но это хороший, простой пример для наших целей здесь.

В листинге 4.8 показано, как может выглядеть эта функция, если она создает проверенное исключение, называемое `NegativeNumberException`, когда она ей предоставляется отрицательное число. В Java расширение класса `Exception` делает исключение проверяемым (эта парадигма Java показана в листинге 4.8). Помимо сигнализации об ошибке, исключение `NegativeNumberException` также инкапсулирует ошибочное значение, которое вызвало ошибку, чтобы помочь в отладке. Сигнатура функции `getSquareroot()` содержит исключение `NegativeNumberException`, указывающее, что оно может вызвать это проверенное исключение; код не компилировался бы, если бы это было опущено.

Листинг 4.8. Создание проверенного исключения

```
class NegativeNumberException extends Exception { ❶
    private final Double erroneousNumber; ❷

    NegativeNumberException(Double erroneousNumber) {
        this.erroneousNumber = erroneousNumber;
    }
}
```

```

    Double getErroneousNumber() {
        return erroneousNumber;
    }
}

Double getSquareRoot(Double value)
    throws NegativeNumberException { ❸
    if (value < 0.0) {
        throw new NegativeNumberException(value); ❹
    }
    return Math.sqrt(value);
}

```

- ❶ Класс для представления определенного типа проверяемого исключения.
- ❷ Инкапсулирует дополнительную информацию: число, вызвавшее ошибку.
- ❸ Функции должны объявлять, какие проверенные исключения они могут создавать.
- ❹ Проверенное исключение выдается, если есть ошибка.

Обработка проверенного исключения

Любая другая функция, вызывающая функцию `getSquareRoot()`, должна либо перехватывать исключение `NegativeNumberException`, либо отмечать, что оно может быть вызвано в собственной сигнатуре функции.

В листинге 4.9 показана функция, которая вызывает `getSquareRoot()` со значением и отображает результат в пользовательском интерфейсе. Функция улавливает исключение `NegativeNumberException`, если оно возникает, и отображает сообщение об ошибке, в котором объясняется, какое число вызвало ошибку.

Листинг 4.9. Обнаружение проверенного исключения

```

void displaySquareRoot() {
    Double value = ui.getInputNumber();
    try {
        ui.setOutput("Square root is: " + getSquareRoot(value));
    } catch (NegativeNumberException e) {❶
        ui.setError("Can't get square root of negative number: "
            +
                e.getErroneousNumber());❷
    }
}

```

- ❶ Исключение `NegativeNumberException` перехватывается, если оно вызвано `getSquareRoot()`.
- ❷ Отображается информация об ошибке из исключения.

Если функция `displaySquareRoot()` не улавливает `NegativeNumberException`, то она должна объявить, что исключение может быть выдано в собственной сигнатуре функции (это показано в следующем листинге). Затем она переносит решение о том, как обработать ошибку, в любой код, называемый функцией `displaySquareRoot()`.

Листинг 4.10. Необнаружение проверенного исключения

```
void displaySquareRoot() throws NegativeNumberException { ❶
    Double value = ui.getInputNumber();
    ui.setOutput("Square root is: " + getSquareRoot(value));
}
```

❶ Исключение `NegativeNumberException` объявляется в сигнатуре функции `displaySquareRoot()`.

Если функция `displaySquareRoot()` не обнаружила исключение `NegativeNumberException` и не объявила его в своей собственной сигнатуре функции, то код не будет компилироваться. Это то, что делает проверенное исключение явным способом сигнализации об ошибке, потому что вызывающий абонент вынужден признать это в той или иной форме.

4.3.3. Неявный: непроверенные исключения

С непроверенными исключениями другие инженеры могут совершенно не обращать внимания на тот факт, что фрагмент кода может выдать его. Часто рекомендуется документировать, какие непроверенные исключения может создавать функция, но инженеры очень часто забывают это сделать. Даже если они это делают, исключение будет только частью мелкого шрифта в соглашении по написанию кода. Как мы видели ранее, мелкий шрифт часто не является надежным способом передачи части кодового соглашения. Таким образом, непроверенные исключения являются неявным способом сигнализации об ошибке, поскольку нет гарантии, что вызывающий будет знать, что ошибка может произойти.

Сигнализация с использованием непроверенного исключения

В листинге 4.11 показаны функция `getSquareRoot()` и исключение `NegativeNumberException` из предыдущего подраздела, но с изменениями, в результате которых исключение `NegativeNumberException` теперь является непроверенным исключением. Как упоминалось ранее, в большинстве языков все исключения являются непроверенными исключениями, но в Java расширение класса `RuntimeException` делает исключение непроверенным (эта парадигма Java показана в листинге 4.11). Функции `getSquareRoot()` теперь не нужно объявлять, что она может выдавать исключение. Исключение `NegativeNumberException` упоминается в документации по функциям, поскольку это желательно, но не обязательно.

Листинг 4.11. Создание непроверенного исключения

```

class NegativeNumberException extends RuntimeException { ❶
    private final Double erroneousNumber;

    NegativeNumberException(Double erroneousNumber) {
        this.erroneousNumber = erroneousNumber;
    }

    Double getErroneousNumber() {
        return erroneousNumber;
    }
}

/**
 * @throws NegativeNumberException если значение отрицательное ❷
 */
Double getSquareRoot(Double value) {
    if (value < 0.0) {
        throw new NegativeNumberException(value); ❸
    }
    return Math.sqrt(value);
}

```

❶ Класс для представления определенного типа непроверенного исключения.

❷ Желательно (но не обязательно), чтобы функции документировали, какие непроверенные исключения они могут создавать.

❸ При возникновении ошибки возникает непроверенное исключение.

Обработка непроверенного исключения

Другая функция, вызывающая функцию `getSquareRoot()`, может перехватывать `NegativeNumberException` точно так же, как в предыдущем примере, когда это было проверенное исключение (повторяется в следующем листинге).

Листинг 4.12. Обнаружение непроверенного исключения

```

void displaySquareRoot() {
    Double value = ui.getInputNumber();
    try {
        ui.setOutput("Square root is: " +
            getSquareRoot(value));
    } catch (NegativeNumberException e) { ❶
        ui.setError("Can't get square root of negative number:

```



```
        " +  
        e.getErroneousNumber());  
    }  
}
```

❶ `NegativeNumberException` перехватывается, если оно вызвано `getSquareRoot()`.

Важно отметить, что функция, вызывающая `getSquareRoot()`, не требуется для подтверждения исключения. Если она не обнаружила исключение, то нет необходимости объявлять его в своей собственной сигнатуре функции или даже в своей собственной документации. В листинге 4.13 показана версия функции `displaySquareRoot()`, которая не обрабатывает и не объявляет `NegativeNumberException`. Поскольку `NegativeNumberException` является непроверенным исключением, это было бы абсолютно нормально. Если бы исключение `NegativeNumberException` было вызвано `getSquareRoot()`, оно либо перешло бы к вызывающему, который его обнаружил, либо программа завершилась бы.

Листинг 4.13. Необнаруженное непроверенное исключение

```
void displaySquareRoot() {  
    Double value = ui.getInputNumber();  
    ui.setOutput("Square root is: " + getSquareRoot(value));  
}
```

Как мы можем видеть, вызывающий функцию, которая создает непроверенное исключение, может совершенно не обращать внимания на тот факт, что исключение может быть вызвано. Это делает непроверенные исключения неявным способом сигнализации об ошибке.

4.3.4. Явный: тип возвращаемого значения, допускающий значение Null

Возврат значения `null` из функции может быть эффективным и простым способом указать, что определенное значение не может быть вычислено (или получено). Если язык, который мы используем, поддерживает нулевую безопасность, то вызывающий будет вынужден знать, что значение может быть нулевым, и обрабатывать его соответствующим образом. Таким образом, использование возвращаемого типа с нулевым значением (когда у нас есть нулевая безопасность) является явным способом сигнализации об ошибке.

Если мы используем язык, который не поддерживает нулевую безопасность, то использование дополнительного возвращаемого типа может быть хорошей альтернативой. Это обсуждалось в главе 2, и более подробная информация о вариантах также доступна в приложении в конце книги.

Сигнализация с использованием `null`

В листинге 4.14 показана функция `getSquareRoot()`, на этот раз измененная таким образом, что она возвращает значение `null`, если входное значение отрицательное. Одна из проблем с возвратом `null` заключается в том, что он не дает никакой информации о причинах возникновения ошибки, поэтому часто может потребоваться добавить комментарий или документацию, чтобы объяснить, что означает `null`.

Листинг 4.14. Возвращение значения `null`

```
// Возвращает значение null, если предоставленное значение отрицательное
Double? getSquareRoot(Double value) { ❶
    if (value < 0.0) {
        return null; ❷
    }
    return Math.sqrt(value);
}
```

❶ Вопрос в `Double?` указывает, что возвращаемое значение может быть нулевым.

❷ При возникновении ошибки возвращается значение `null`.

Обработка значения `null`

Поскольку язык поддерживает нулевую безопасность, вызывающий объект вынужден проверять, является ли значение, возвращаемое `getSquareRoot()`, нулевым, прежде чем они смогут его использовать. В следующем листинге показана функция `displaySquareRoot()`, но на этот раз обрабатывающая возвращаемый тип с возможностью обнуления.

Листинг 4.15. Обработка нулевого значения

```
void displaySquareRoot() {
    Double? squareRoot = getSquareRoot(ui.getInputNumber());
    if (squareRoot == null) { ❶
        ui.setError("Can't get square root of a negative number");
    } else {
        ui.setOutput("Square root is: " + squareRoot);
    }
} 4.13 Not catching an
```

❶ Возвращаемое значение `getSquareRoot()` необходимо проверить на значение `null`.

Не совсем верно говорить, что вызывающий вынужден проверять, является ли значение нулевым. Он всегда мог бы просто присвоить значению значение, отличное от нуля, но это все еще активное решение, и при этом ему пришлось бы признать, что значение может быть нулевым.

4.3.5. Явный: возвращаемый тип Result

Одна из проблем с возвращением нулевого или опционального типа возвращаемого значения заключается в том, что мы не можем передать информацию об ошибке. В дополнение к информированию абонента о том, что значение не может быть получено, также может быть полезно сообщить ему, почему оно не может быть получено. Если это так, то использование типа результата может быть уместным.

Такие языки, как Swift, Rust и F#, имеют встроенную поддержку для этого и предоставляют хороший синтаксис, упрощающий их использование. Мы можем создать свой собственный тип результата на любом языке, но без встроенного синтаксиса их использование может быть немного более неуклюжим.

В листинге 4.16 приведен базовый пример того, как мы можем определить наш собственный тип результата на языке, который не имеет встроенной поддержки для этого.

Листинг 4.16. Простой тип `result`

```
class Result<V, E> { ❶
    private final Optional<V> value;
    private final Optional<E> error;

    private Result(Optional<V> value, Optional<E> error) { ❷
        this.value = value;
        this.error = error;
    }

    static Result<V, E> ofValue(V value) { ❸
        return new Result(Optional.of(value),
            Optional.empty());
    }

    static Result<V, E> ofError(E error) { ❹
        return new Result(Optional.empty(),
            Optional.of(error));
    }

    Boolean hasError() {
        return error.isPresent();
    }

    V getValue() {
        return value.get();
    }
}
```

```

Е getError() {
    return error.get();
}
}

```

- ❶ Общие/шаблонные типы, используемые для того, чтобы класс можно было использовать с любыми типами значений и ошибок.
- ❷ Частный конструктор, чтобы заставить вызывающих использовать одну из статических фабричных функций.
- ❸ Статические заводские функции, означающие, что экземпляр класса может быть создан только со значением или ошибкой, но не с обоими.
- ❹ Статические заводские функции, означающие, что экземпляр класса может быть создан только со значением или ошибкой, но не с обоими [2].

РЕАЛИЗАЦИИ ТИПА `Result`

Реальные реализации типов `result` часто будут более сложными, чем пример в листинге 4.16. Они часто лучше используют языковые конструкции, такие, как перечисления, и предоставляют вспомогательные функции для преобразования результатов.

Реализации результата `Rust` и `Swift` могут стать хорошим источником вдохновения:

<https://doc.rust-lang.org/beta/core/result/enum.Result.html>

<https://developer.apple.com/documentation/swift/result>

Если мы определяем наш собственный тип результата (потому что в языке он не встроен), то он зависит от того, знают ли другие инженеры о том, как его использовать. Если другой инженер не знает, как проверить функцию `hasError()` перед вызовом `getValue()`, то это лишает все смысла, хотя прилежный инженер, скорее всего, сможет довольно быстро разобраться в этом, даже если он никогда раньше не сталкивался с типом результата.

Предполагая, что язык поддерживает тип результата или что другие инженеры знакомы с ним (если мы определяем свой собственный), то использование его в качестве возвращаемого типа дает понять, что может произойти ошибка. Таким образом, использование типа возвращаемого результата является явным способом сигнализации об ошибке.

Сигнализация с использованием `result`

В листинге 4.17 показана функция `getSquareRoot()`, но на этот раз она изменена, чтобы возвращать тип результата. `NegativeNumberError` является пользовательской ошибкой, и тип возвращаемого значения `getsquareroot()` означает, что эта ошибка потенциально может возникнуть. `NegativeNumberError` инкапсулирует дополнительную информацию об ошибке: ошибочное число, которое ее вызвало.

Листинг 4.17. Возвращает тип `result`

```

class NegativeNumberError extends Error { ❶
    private final Double erroneousNumber;

```

```

    NegativeNumberError(Double erroneousNumber) { ❷
        this.erroneousNumber = erroneousNumber;
    }

    Double getErroneousNumber() {
        return erroneousNumber;
    }
}

Result<Double, NegativeNumberError> getSquareRoot(Double value) { ❸
    if (value < 0.0) {
        return Result.ofError(new NegativeNumberError(value)); ❹
    }
    return Result.ofValue(Math.sqrt(value)); ❺
}

```

- ❶ Класс для представления определенного типа ошибки.
- ❷ Инкапсулирует дополнительную информацию: число, вызвавшее ошибку.
- ❸ Тип возвращаемого значения означает, что может произойти ошибка `NegativeNumberError`.
- ❹ При возникновении ошибки возвращается результат ошибки.
- ❺ Ответ заключен в результате.

Обработка result

Инженеру, вызывающему `getSquareRoot()`, будет очевидно, что возвращаемый тип — `Result`. Предполагая, что он знаком с использованием `Result`, он будет знать, что сначала необходимо вызвать функцию `hasError()`, чтобы проверить, произошла ли ошибка, а если она не произошла, то можно вызвать функцию `GetValue()` для доступа к значению. Если произошла ошибка, то подробную информацию можно получить, вызвав `getError()` для получения результата. Это показано в следующем листинге.

Листинг 4.18. Возвращает тип результата

```

void displaySquareRoot() {
    Result<Double, NegativeNumberError> squareRoot =
        getSquareRoot(ui.getInputNumber());
    if (squareRoot.hasError()) { ❶
        ui.setError("Can't get square root of a negative
            number: " +
                squareRoot.getError().getErroneousNumber()); ❷
    }
}

```

```

    } else {
        ui.setOutput("Square root is: " +
            squareRoot.getValue());
    }
}

```

- ❶ Результат `squareRoot` должен быть проверен на наличие ошибки.
- ❷ Подробная информация об ошибке, отображаемая пользователю.

БОЛЕЕ ПРИЯТНЫЙ СИНТАКСИС

Языки со встроенной поддержкой типов результатов иногда будут иметь более сжатые синтаксисы для их обработки, чем показано в листинге 4.18. Существует также множество вспомогательных функций, которые мы могли бы добавить в пользовательскую реализацию типа результата, чтобы создать более приятный поток управления, например, такие вещи, как функция `and_then()` в реализации Rust: <http://mng.bz/Jv5P>.

4.3.6. Явный: возвращаемый тип `Outcome`

Некоторые функции просто что-то делают, а не получают значение и возвращают его. Если при выполнении этой операции может возникнуть ошибка и было бы полезно сообщить об этом вызывающему, то один из подходов заключается в изменении функции для возврата значения, указывающего результат операции. Как мы вскоре увидим, тип возвращаемого результата `outcome` является явным способом сигнализации об ошибке, если мы можем заставить вызывающих абонентов проверять возвращаемое значение.

Сигнализация с использованием `outcome`

В листинге 4.19 показан некоторый код для отправки сообщения по каналу. Сообщение может быть отправлено только в том случае, если канал открыт. Если канал не открыт, то это ошибка. Функция `SendMessage()` сигнализирует о возникновении ошибки, возвращая логическое значение. Если сообщение отправлено, то функция возвращает значение `true`. Если возникает ошибка, она возвращает значение `false`.

Листинг 4.19. Возврат `outcome`

```

Boolean sendMessage(Channel channel, String message) { ❶
    if (channel.isOpen()) {
        channel.send(message);
        return true; ❷
    }
    return false; ❸
}

```

- ❶ Функция возвращает логическое значение `Boolean`.
- ❷ Значение `true` возвращается, если сообщение было отправлено.
- ❸ Значение `false` возвращается, если произошла ошибка.

Если у нас есть более сложный сценарий, то мы могли бы счесть целесообразным использовать более сложный тип результата, чем `Boolean`. Перечисление полезно, если существует более двух возможных состояний результата или если из контекста не очевидно, что означает истина и ложь. Если нам потребуется более подробная информация, то определение целого класса для инкапсуляции этого может быть еще одним хорошим вариантом.

Обработка `outcome`

В примере, используемом `Boolean` в качестве возвращаемого типа, обработка результата довольно проста. Вызов функции может быть просто помещен в оператор `if-else`, а соответствующая логика обработки размещена в каждой ветви. В следующем листинге показан код для отправки сообщения «привет» по каналу и отображения сообщения в пользовательском интерфейсе, чтобы указать, было ли отправлено сообщение.

Листинг 4.20. Обработка `outcome`

```
void sayHello(Channel channel) {  
    if (sendMessage(channel, "hello")) {  
        ui.setOutput("Hello sent"); ❶  
    } else {  
        ui.setError("Unable to send hello"); ❷  
    }  
}
```

❶ Успешный сценарий обработан.

❷ Неуспешный сценарий обработан.

Обеспечение того, чтобы `outcome` не игнорировался

Одна из проблем с типом `outcome` заключается в том, что вызывающему довольно легко игнорировать возвращаемое значение или даже не знать, что функция возвращает значение. Это может показать, насколько явным является тип возвращаемого результата `outcome` способом сигнализации об ошибке. Чтобы продемонстрировать это, вызывающий может написать код в следующем листинге. Код полностью игнорирует возвращаемое значение результата из `SendMessage()` и при этом сообщает пользователю, что сообщение было отправлено, когда на самом деле его могло и не быть.

Листинг 4.21. Игнорирование `outcome`

```
void sayHello(Channel channel) {  
    sendMessage(channel, "hello"); ❶  
    ui.setOutput("Hello sent");  
}
```

❶ Возвращаемое значение `outcome` игнорируется.

В некоторых языках существует способ пометить функцию таким образом, чтобы создавалось предупреждение компилятора, если вызывающий объект игнорирует возвращаемое значение функции. Их название и использование варьируются от языка к языку, но вот некоторые примеры:

- ◆ Аннотация `CheckReturnValue` в Java (из пакета `javax.annotation`).
- ◆ Аннотация `MustUseReturnValue`, которую можно использовать в C# (с <https://www.jetbrains.com/help/resharper/>).
- ◆ `[[nodiscard]]` атрибут в C++.

Если бы функция `SendMessage()` была помечена одним из них, то код в листинге 4.21 выдал бы предупреждение компилятора, которое, вероятно, было бы замечено инженером, пишущим код. В следующем листинге показано, как выглядела бы функция `SendMessage()`, если бы она была помечена аннотацией `@checkreturnvalue`.

Листинг 4.22. Использование аннотации `CheckReturnValue`

`@CheckReturnValue` ❶

```
Boolean sendMessage(Channel channel, String message) {
    ...
}
```

❶ Указывает, что вызывающие не должны игнорировать возвращаемое значение функции.

Автор кода в листинге 4.21, скорее всего, заметит предупреждение компилятора и изменит свой код до версии, которую мы видели ранее, которая обрабатывала его (повторяется в следующем листинге).

Листинг 4.23. Принудительно проверяет возвращаемое значение

```
void sayHello(Channel channel) {
    if (sendMessage(channel, "hello")) {
        ui.setOutput("Hello sent"); ❶
    } else {
        ui.setError("Unable to send hello"); ❷
    }
}
```

❶ Успешный сценарий обработан.

❷ Неуспешный сценарий обработан.

4.3.7. Неявный: обещание или будущее

При написании кода, который выполняется асинхронно, обычно создается функция, которая возвращает *обещание* или *будущее* (или эквивалентную концепцию). На многих языках (но не на всех) обещание или будущее также могут передавать состояние ошибки.

Потребитель обещания или будущего, как правило, не вынужден обрабатывать ошибку, которая могла произойти, и он не знал бы, что ему нужно добавить обработку ошибок, если бы он не был знаком с мелким шрифтом в кодовом соглашении для рассматриваемой функции. Таким образом, сигнализация ошибок с использованием обещания или будущего является неявной техникой.

Асинхронный?

Если процесс синхронный, то это означает, что задачи выполняются по одной за раз: задача не может начаться, пока предыдущая задача не будет полностью завершена. Если мы делаем торт, то мы не можем испечь торт в духовке, пока сначала не смешаем тесто для торта.

Это пример синхронного процесса: процесс выпечки торта блокируется необходимостью сначала перемешать торт.

Если процесс асинхронен, то это означает, что мы можем выполнять различные задачи, ожидая завершения других задач. Пока наш торт выпекается в духовке, мы могли бы использовать потраченное впустую время, чтобы приготовить глазурь для торта. Это пример асинхронного процесса: мы можем сделать глазурь, не дожидаясь, пока торт будет испечен.

Когда коду приходится ждать, пока что-то произойдет (например, сервер вернет ответ), обычно его пишут асинхронным способом. Это означает, что код может выполнять другие действия во время ожидания сервера.

Большинство языков программирования предоставляют способы асинхронного выполнения кода.

Как именно это сделать, может сильно отличаться в разных языках, поэтому стоит поискать его на любом языке, который вы используете. Использование асинхронной функции и обещание в следующих примерах кода наиболее близко напоминает парадигму JavaScript.

Если вы не знакомы с этим и хотели бы узнать больше, на следующей справочной странице представлен хороший обзор с примерами: <http://mng.bz/w0wW>.

Сигнализация с использованием обещания

В листинге 4.24 показана функция `getSquareRoot()`, но на этот раз измененная, чтобы быть асинхронной функцией, которая возвращает обещание и ожидает одну секунду перед запуском. (Вам придется использовать свое воображение, чтобы понять, почему кто-то на самом деле написал этот конкретный код.) Если внутри функции возникает ошибка, то обещание будет *отклонено*; в противном случае обещание будет *выполнено* с возвращенным значением.

Листинг 4.24. Асинхронная функция

```
class NegativeNumberError extends Error { ❶
  ...
}

Promise<Double> getSquareRoot(Double value) async { ❷
  await Timer.wait(Duration.ofSeconds(1)); ❸
```

```

    if (value < 0.0) {
        throw new NegativeNumberError(value); ❹
    }
    return Math.sqrt(value); ❺
}

```

- ❶ Класс для представления определенного типа ошибок.
- ❷ `async` помечает функцию как асинхронную.
- ❸ Секундное ожидание до реального запуска.
- ❹ Ошибка, возникшая внутри функции, приведет к отклонению обещания.
- ❺ Возвращаемое значение приводит к выполнению обещания.

Выполнение обещания

В листинге 4.25 показана функция `displaySquareRoot()`, измененная для вызова асинхронной версии `getSquareRoot()`. Обещание, возвращаемое `getSquareRoot()`, содержит две функции-члена, которые можно использовать для установки обратных вызовов. Функцию `then()` можно использовать для задания функции обратного вызова, которая вызывается, если и когда обещание выполнено, а функцию `catch()` можно использовать для задания обратного вызова, который вызывается, если и когда обещание отклонено.

Листинг 4.25. Использование обещания

```

void displaySquareRoot() {
    getSquareRoot(ui.getInputNumber())
        .then(squareRoot ->
            ui.setOutput("Square root is: " + squareRoot)) ❶
        .catch(error ->
            ui.setError("An error occurred: " +
                error.toString())); ❷
}

```

- ❶ Функция обратного вызова `then()` вызывается, если и когда обещание будет выполнено.
- ❷ Функция обратного вызова `catch()` вызывается, если и когда обещание отклонено.

Почему обещания являются неявной сигнальной техникой

Чтобы знать, что может произойти ошибка и что обещание может быть отклонено, нам нужно знать либо мелкий шрифт, либо детали реализации любой функции, сгенерировавшей обещание. Не зная этого, потребитель обещания может легко не знать о потенциальном состоянии ошибки и может обеспечить обратный вызов только с помощью функции `then()`.

Если обратный вызов не был предоставлен с помощью функции `catch()`, ошибка может быть поймана каким-либо обработчиком ошибок более высокого уровня или она может просто остаться совершенно незамеченной (в зависимости от языка и настроек).

Обещания и будущие могут быть отличным способом возврата значений из асинхронных функций. Но, поскольку вызывающий может быть совершенно не осведомлен о потенциальном сценарии ошибки, использование обещания или будущего является неявным способом сигнализации об ошибке.

Выполнение явного обещания

Если мы возвращаем обещание или будущее и хотим использовать явный метод сигнализации об ошибках, то одним из вариантов является возврат обещания типа результата. Если мы сделаем это, то функция `getSquareRoot()` будет выглядеть следующим образом. Это может быть полезным приемом, но код становится довольно неуклюжим, поэтому он не всем нравится.

```
Promise<Result<Double, NegativeNumberError>> getSquareRoot( ❶  
    Double value) async {  
    await Timer.wait(Duration.ofSeconds(1));  
    if (value < 0.0) {  
        return Result.ofError(new NegativeNumberError(value));  
    }  
    return Result.ofValue(Math.sqrt(value));  
}
```

❶ Тип возвращаемого значения довольно неуклюжий.

4.3.8. Неявный: возврат магического значения

Магическое значение (или код ошибки) — это значение, которое соответствует обычному типу возвращаемого значения функции, но имеет особое значение. Инженер должен либо прочитать документацию, либо сам код, чтобы знать, что может быть возвращено магическое значение. Это делает их неявным методом сигнализации об ошибках.

Распространенным способом сигнализации об ошибке с использованием магического значения является возврат минус единицы. В следующем листинге показано, как это выглядит для функции `getSquareRoot()`.

Листинг 4.27. Возвращает магическое значение

```
// Возвращает -1, если указано отрицательное значение  
Double getSquareRoot(Double value) {  
    if (value < 0.0) {  
        return -1.0; ❶  
    }  
}
```

```
return Math.sqrt(value);  
}
```

❶ Минус один, возвращаемый при возникновении ошибки.

Магические значения могут легко вызывать сюрпризы и приводить к ошибкам, потому что их необходимо обрабатывать, но ничто в безошибочно очевидной части кодового контракта не сообщает об этом вызывающим абонентам. Проблемы, которые могут вызвать магические значения, подробно обсуждаются в *главе 6*, поэтому мы не будем подробно обсуждать их здесь. Сейчас важно помнить, что магические значения часто не являются хорошим способом сигнализировать об ошибке.

4.4. Сигнализация ошибок, после которых нельзя восстановиться

Когда возникает ошибка, после которой нет реальной перспективы восстановления программы, тогда лучше всего объявить о ней быстро и громко. Некоторые общие способы достижения этого заключаются в следующем.

- ◆ Создание непроверенного исключения.
- ◆ Вызывание *паники* в программе (при использовании языка, поддерживающего панику).
- ◆ Использование проверки или утверждения (как описано в *главе 3*).

Это приведет к выходу программы (или области безвозвратности), что должно означать, что инженеры заметили неисправность, полученное сообщение об ошибке обычно содержит трассировку стека или номер строки, чтобы четко указать, где произошла ошибка.

Использование неявной техники (подобной только что упомянутой) предотвращает необходимость для каждого вызывающего выше по цепочке вызовов писать код для подтверждения или обработки сценария ошибки. Когда нет никакого мыслимого способа исправить ошибку, это имеет смысл, потому что нет ничего разумного, что вызывающий мог бы сделать, кроме как просто передать ошибку следующему вызывающему.

4.5. Сообщения об ошибках, от которых вызывающий абонент может захотеть восстановиться

Именно здесь все становится интересным, потому что инженеры-программисты (и разработчики языков программирования) не все согласны с тем, какие рекомендации следует применять в отношении сигнализации ошибок, от которых вызывающий абонент может захотеть восстановиться. Здесь обычно обсуждается вопрос об использовании непроверенных исключений по сравнению с явными методами сигнализации об ошибках (такими, как проверенные исключения, безопасные зна-

чения `null`, необязательные типы или типы результатов). У обеих сторон есть веские аргументы и контраргументы, и я постараюсь обобщить их в этом разделе.

Прежде чем я это сделаю, стоит вспомнить, что, вероятно, важнее любого из следующих аргументов то, что вы и ваша команда согласны с вашей философией. Наихудшая возможная ситуация заключается в том, что половина вашей команды пишет код, который следует одной практике в отношении сигнализации и обработки ошибок, а другая половина придерживается совершенно другой практики. Будет кошмар, когда части кода будут взаимодействовать друг с другом.

Приведенные ниже аргументы, скорее всего, будут звучать как «все или ничего», но имейте в виду, что если вы поговорите с другим инженером, то, вероятно, обнаружите, что его мнения о сигнализации и обработке ошибок более детализированы.

Утечка сведений о реализации

Еще одна вещь, которую следует учитывать при обнаружении ошибок, от которых вызывающий может захотеть восстановиться, заключается в том, что в идеале вызывающим абонентам не нужно знать детали реализации вызываемого кода, чтобы обрабатывать ошибки, которые он может сигнализировать. Это обсуждается в контексте модульности в *главе 8* (разд. 8.6 и 8.7).

4.5.1. Аргументы в пользу использования непроверенных исключений

Ниже приведены некоторые общие аргументы в пользу того, почему лучше использовать непроверенные исключения даже для ошибок, которые потенциально могут быть исправлены.

Улучшение структуры кода

Некоторые инженеры утверждают, что создание непроверенного исключения (вместо использования явной техники) может улучшить структуру кода, поскольку большая часть обработки ошибок может выполняться в отдельном месте на более высоком уровне кода. Ошибки всплывают на этот уровень, и код между ними не должен быть загроможден множеством логики обработки ошибок. На рис. 4.4 показано, как это может выглядеть.

Слои в середине могут обрабатывать некоторые исключения, если захотят (например, повторять определенные задачи), но в противном случае ошибки будут просто всплывать до самого верхнего уровня обработки ошибок. Если бы это было пользовательское приложение, уровень обработки ошибок мог бы отображать наложение на пользовательский интерфейс для отображения сообщения об ошибке; или если бы это был серверный или серверный процесс, то сообщение об ошибке могло бы быть где-то зарегистрировано. Ключевое преимущество этого подхода заключается в том, что логика обработки ошибок может содержаться в нескольких отдельных слоях, а не распространяться по всему коду.

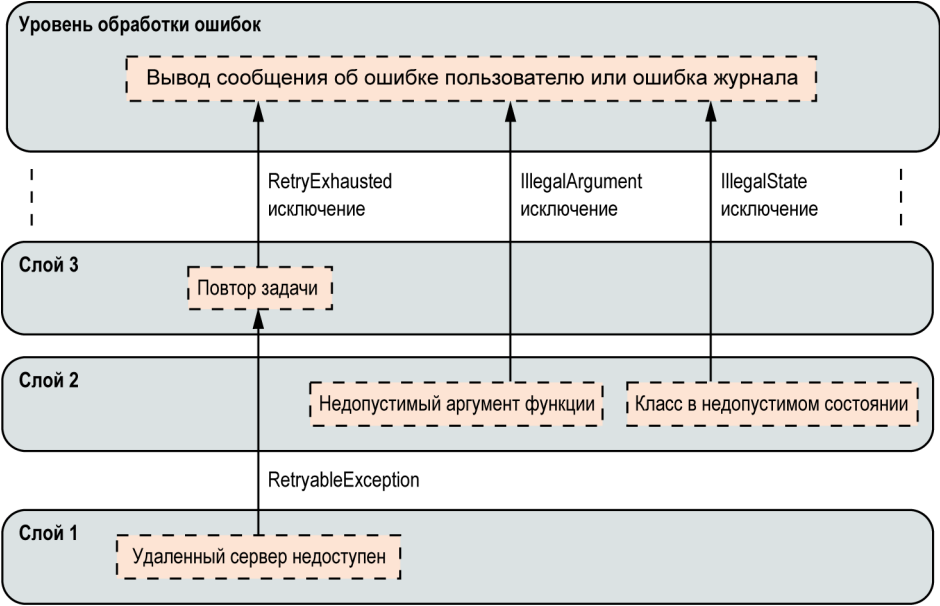


Рис. 4.4. Некоторые инженеры утверждают, что использование непроверенных исключений может улучшить структуру кода, поскольку большая часть обработки ошибок может выполняться на нескольких отдельных уровнях

Будьте прагматичны в отношении того, что будут делать инженеры

Некоторые утверждают, что с помощью более явных методов сигнализации об ошибках (возвращаемых типов и проверяемых исключений) инженеры в конце концов устанут и делают неправильные вещи, например перехватывают исключение и игнорируют его или приводят тип, допускающий значение `null`, к ненулевому без проверки.

Чтобы продемонстрировать это, представьте, что код в листинге 4.28 существовал где-то в кодовой базе. Список содержит код для регистрации температуры в регистраторе данных, который, в свою очередь, использует хранилище `InMemoryDataStore` для хранения зарегистрированных данных. Ничто в этой начальной версии кода не может вызвать ошибку, поэтому не требуется никаких методов сигнализации или обработки ошибок.

Листинг 4.28. Исходный код без сценариев ошибок

```
class TemperatureLogger {
    private final Thermometer thermometer;
    private final DataLogger dataLogger;
    ...
}
```

```
void logCurrentTemperature() {
    dataLogger.logDataPoint(
        Instant.now(),
        thermometer.getTemperature());
}

class DataLogger {
    private final InMemoryDataStore dataStore;
    ...

    void logDataPoint(Instant time, Double value) {
        dataStore.store(new DataPoint(time.toMillis(), value));
    }
}
```

Теперь представьте, что инженера попросили изменить класс `DataLogger` так, чтобы вместо простого сохранения значений в памяти он теперь сохранял их на диске для того, чтобы убедиться, что они сохраняются. Вместо этого инженер меняет класс `InMemoryDataStore` на класс `DiskDataStore`. Запись на диск может завершиться неудачно, поэтому теперь может возникнуть ошибка. Если используется явный метод сигнализации об ошибке, то ошибку необходимо либо обработать, либо явно передать следующему вызывающему по цепочке.

Продemonстрируем это с помощью `DiskDataStore.store()`, генерирующего проверенное исключение (`IOException`): принцип тот же, что и для любого другого явного метода сигнализации об ошибках. Поскольку `IOException` — это проверенное исключение, его нужно либо обработать, либо добавить в подпись функции `DataLogger.logDataPoint()`. Нет разумного способа обработать эту ошибку в функции `DataLogger.logDataPoint()`, но добавление ее к сигнатуре функции потребует изменения каждой точки вызова и, возможно, нескольких точек вызова на уровнях выше.

Обескураженный таким объемом работы, инженер решает просто скрыть ошибку и записывает код в следующий листинг.

Листинг 4.29. Скрытие проверенного исключения

```
class DataLogger {
    private final DiskDataStore dataStore;
    ...

    void logDataPoint(Instant time, Double value) {
        try {
            dataStore.store(new DataPoint(time.toMillis(),
                value));
        }
```

```
} catch (IOException e) {} ❶  
}  
  
}
```

❶ Ошибка `IOException` скрыта от вызывающих.

Как обсуждалось ранее в этой главе, скрывание ошибок почти никогда не бывает хорошей идеей. Функция `ILogger.LogDataPoint()` теперь не всегда выполняет то, что она утверждает; иногда точки данных не сохраняются, но вызывающий абонент не будет знать об этом. Использование явных методов сигнализации об ошибках может иногда вызывать каскад работ, которые необходимо выполнить, чтобы многократно сигнализировать об ошибке по уровням кода, и это может побудить инженеров срезать углы и делать неправильные вещи. Необходимость быть прагматичным в этом вопросе — один из наиболее часто высказываемых аргументов в пользу использования непроверенных исключений.

4.5.2. Аргументы в пользу использования явных методов

Ниже приведены некоторые общие аргументы в пользу того, почему лучше использовать явные методы сигнализации об ошибках для ошибок, которые потенциально могут быть устранены.

Изящная обработка ошибок

При использовании непроверенных исключений трудно создать единый слой, который мог бы корректно обрабатывать все ошибки. Например, если пользовательский ввод неверен, вероятно, имеет смысл иметь хорошее сообщение об ошибке рядом с полем ввода. Если инженер, пишущий код, обрабатывающий входные данные, не знал о сценарии ошибки и оставил его на более высоком уровне, это может привести к менее удобному для пользователя сообщению об ошибке, такому, как общее сообщение, наложенное на пользовательский интерфейс.

Принуждение вызывающих абонентов быть осведомленными о потенциальных ошибках (с помощью возвращаемого типа или проверенного исключения) означает, что существует большая вероятность того, что эти ошибки будут обработаны корректно. С помощью неявной техники вызывающий может не знать, что может произойти сценарий ошибки, так как же он будет знать, как с этим справиться?

Ошибки нельзя случайно игнорировать

Могут быть некоторые ошибки, которые действительно должны быть обработаны определенными вызывающими. Если используется непроверенное исключение, то неправильное действие (необработка ошибки) происходит по умолчанию, а не является активным решением. Это происходит потому, что инженеру (и рецензенту кода) легко попасть в полное неведение о том, что может произойти определенная ошибка.

Если использовать более явный метод сигнализации об ошибках, такой, как тип возвращаемого значения или проверенное исключение, то инженеры все равно могут поступать неправильно (например, перехватывать исключение и игнорировать его). Обычно это требует активных усилий, приводит к довольно вопиющему нарушению в коде и повышает вероятность того, что подобные проблемы могут быть устранены в обзорах кода, потому что это будет очевидно для рецензента. При использовании более явных методов сигнализации об ошибках неправильные действия не происходят по умолчанию или случайно.

**Изменение кода
при использовании непроверенного исключения**

```
class TemperatureLogger {
    private final Thermometer thermometer;
    private final DataLogger dataLogger;
    ...

    void startLogging() {
        Timer.schedule(
            logCurrentTemperature,
            Duration.ofMinutes(1));
    }

    void logCurrentTemperature() {
+       dataLogger.logDataPoint(
+           Instant.now(),
+           thermometer.getTemperature());
    }
}

class DataLogger {
    ...

    /**
     * @throws UncheckedIOException if
     * saving data point fails.
     */
    void logDataPoint(
        Instant time,
        Double value) { ... }
}
```

Исключение `UncheckedIOException` не было обработано, но ни одна из новых строк кода не выглядит явно неправильной

**Изменение кода
при использовании проверенного исключения**

```
class TemperatureLogger {
    private final Thermometer thermometer;
    private final DataLogger dataLogger;
    ...

    void startLogging() {
        Timer.schedule(
            logCurrentTemperature,
            Duration.ofMinutes(1));
    }

    void logCurrentTemperature() {
+       try {
+           dataLogger.logDataPoint(
+               Instant.now(),
+               thermometer.getTemperature());
+       } catch (IOException e) {
+           logError(e);
+       }
    }
}

class DataLogger {
    ...

    void logDataPoint(
        Instant time,
        Double value) throws IOException
    { ... }
}
```

Исключение `IOException` не было обработано должным образом, но это явно заметно в новых строках кода и, вероятно, будет замечено рецензентом

`Timer.schedule()` запускает новый поток, поэтому любые исключения, создаваемые кодом, который он планирует, не всплывают на уровне выше этого

Рис. 4.5. При использовании явных методов сигнализации об ошибках неправильное обращение с ошибкой часто приводит к преднамеренному и вопиющему нарушению кода.

Напротив, при использовании непроверенного исключения из кода может быть не очевидно, что ошибка не была обработана должным образом

На рис. 4.5 показано, как изменение кода может выглядеть для рецензента при использовании непроверенного исключения и проверенного исключения. Тот факт, что в коде происходит что-то плохое, совсем не очевиден при использовании непроверяемого исключения, тогда как он чрезвычайно очевиден в случае, когда используется проверенное исключение. Другие явные методы сигнализации об ошибках (такие, как тип возвращаемого результата, принудительно применяемый с помощью аннотации `@CheckReturnValue`) заставили бы проступок инженера быть столь же очевидным при изменении кода.

Будьте прагматичны в отношении того, что будут делать инженеры

Аргументы в пользу того, что инженеры поступают неправильно, потому что им надоела обработка ошибок, также применимы против использования непроверенных исключений. Нет никакой гарантии, что непроверенные исключения будут должным образом задокументированы во всей кодовой базе, а, судя по моему личному опыту, они часто этого не делают. Вот почему нет уверенности в том, какие именно непроверенные исключения может выдать код, и это может превратить их перехват в разочаровывающую игру в «ударь крота».

Листинг 4.30 содержит функцию для проверки правильности файла данных. Он делает это, проверяя, не возникают ли какие-либо исключения, указывающие на то, что файл недействителен. `DataFile.parse()` выдает ряд различных непроверенных исключений, которые не все были задокументированы. Автор функции `isDataFileValid()` добавил код для перехвата трех из этих непроверенных исключений.

Листинг 4.30. Обнаружение нескольких типов непроверенных исключений

```
Boolean isDataFileValid(byte[] fileContents) {
    try {
        DataFile.parse(fileContents); ❶
        return true;
    } catch (InvalidEncodingException | ❷
            ParseException |
            UnrecognizedDataKeyException e) {
        return false;
    }
}
```

❶ Может создавать ряд недокументированных, непроверенных исключений.

❷ Улавливает три разных типа непроверенных исключений.

После выпуска своего кода, автор функции `isDataFileValid()` замечает, что он видит множество отчетов о сбоях. Он исследует и обнаруживают, что сбои вызваны еще одним недокументированным, непроверенным исключением: `InvalidDataRangeException`. На этом этапе автору кода, может быть, так надоело валять дурака со всеми этими непроверенными исключениями, что он решает вместо этого перехватить все типы исключений и покончить с этим. Код в следующем листинге.

Листинг 4.31. Отслеживание всех типов исключений

```
Boolean isDataFileValid(byte[] fileContents) {  
    try {  
        DataFile.parse(fileContents);  
        return true;  
    } catch (Exception e) { ❶  
        return false;  
    }  
}
```

❶ Улавливает все типы исключений.

Ловля каждого типа исключений в подобном коде часто плохая идея. Это скроет почти все типы ошибок, в том числе многие, от которых программа не может восстановиться. Возможно, произошла серьезная ошибка программирования, которая теперь скрыта. Это может быть ошибка функции `DataFile.parse()` или серьезная неправильная конфигурация программного обеспечения, которая приводит к чему-то вроде `ClassNotFoundException`. В любом случае эти программные ошибки теперь останутся совершенно незамеченными, и программа выйдет из строя тихим и странным образом.

Нарушение, подобное приведенному в листинге 4.31, является весьма вопиющим, поэтому мы надеемся, что оно будет устранено во время проверки кода. Но если мы обеспокоены тем, что наш процесс проверки кода недостаточно надежен, чтобы выявлять подобные нарушения, то стоит понимать, что у нас, вероятно, возникнет проблема, решим ли мы использовать непроверенные исключения или явные методы сигнализации об ошибках. Настоящая проблема в том, что у нас есть инженеры, которые делают работу небрежно, и нет надежного способа их отсеять.

ПРИДЕРЖИВАЙТЕСЬ СТАНДАРТНЫХ ТИПОВ ИСКЛЮЧЕНИЙ

Чтобы избежать этого исключения, иногда инженеры, использующие непроверенные исключения, предпочитают использовать стандартные типы исключений (или создавать подклассы) (например, `ArgumentException` или `StateException`). Более вероятно, что другие инженеры предскажут, что они могут быть созданы, и обработают их соответствующим образом, а это ограничивает количество типов исключений, о которых инженерам обычно нужно беспокоиться.

Обратной стороной этого является то, что это может ограничить возможность различать различные сценарии ошибок: одной причиной `StateException` может быть то, от чего вызывающий хочет восстановиться, в то время как другой причины может и не быть. Как вы, вероятно, уже поняли, тема сигнализации об ошибках и обработки ошибок — несовершенная наука, и у любой техники есть свои плюсы и минусы.

4.5.3. Мое мнение: используйте явную технику

Мое мнение таково, что лучше избегать использования непроверенного исключения для ошибки, от которой вызывающий может захотеть восстановиться. По моему опыту, использование непроверенных исключений очень редко полностью до-

кументируется во всей кодовой базе, поэтому для инженера, использующего функцию, практически невозможно точно знать, какие сценарии ошибок могут произойти и с которыми необходимо справиться.

Я видел слишком много ошибок и сбоев, вызванных использованием недокументированных, непроверенных исключений для ошибок, которые вызывающий абонент хотел бы исправить, если бы только инженер, пишущий этот код, знал об исключении, поэтому я лично предпочитаю использовать явные методы сигнализации об ошибках, когда есть вероятность, что вызывающий захочет восстановиться.

Как уже обсуждалось в этом разделе, этот подход не лишен некоторых недостатков, но, по моему опыту, недостатки использования непроверенных исключений для такого рода ошибок еще хуже. Однако, как я уже говорил ранее, потенциально еще хуже, если вы работаете в команде, где одни инженеры придерживаются одного подхода, а другие — другого, поэтому лучше всего, чтобы вы и ваша команда согласовали философию сигнализации об ошибках и придерживались ее.

4.6. Не игнорируйте предупреждения компилятора

В главе 3 рассмотрены некоторые методы обеспечения того, чтобы ошибка компилятора возникала в случае нарушения или неправильного использования кода. В дополнение к ошибкам компилятора большинство компиляторов также выдают предупреждения. Предупреждение компилятора часто помечает код, который в некотором роде подозрителен, и это может быть ранним предупреждением о возможной ошибке. Внимание к этим предупреждениям может быть отличным способом выявить и устранить ошибки программирования, задолго до того как код приблизится к кодовой базе.

Чтобы продемонстрировать это, рассмотрим код в листинге 4.32 для класса, хранящего некоторую информацию о пользователе. Он содержит ошибку, поскольку функция `getDisplayName()` неправильно возвращает настоящее имя пользователя, а не его отображаемое имя.

Листинг 4.32. Код, вызывающий предупреждение компилятора

```
class UserInfo {
    private final String realName;
    private final String displayName;

    UserInfo(String realName, String displayName) {
        this.realName = realName;
        this.displayName = displayName;
    }

    String getRealName() {
        return realName;
    }
}
```

```
String getDisplayName() {  
    return realName; ❶  
}  
}
```

❶ Настоящее имя пользователя возвращено ошибочно.

Этот код будет компилироваться, но компилятор, скорее всего, выдаст предупреждение примерно следующего содержания: «ПРЕДУПРЕЖДЕНИЕ: закрытый член «`userInfo.DisplayName`» может быть удален, так как присвоенное ему значение никогда не будет прочитано». Если мы проигнорируем это предупреждение, мы можем не понять, что эта ошибка существует. Мы могли бы надеяться, что тестирование поймает его, но, если этого не произойдет, это может быть довольно серьезной ошибкой, которая сильно нарушает конфиденциальность пользователей.

Большинство компиляторов можно настроить так, чтобы любые предупреждения становились ошибками и препятствовали компиляции кода. Это может показаться немного чрезмерным и драконовским методом, но на самом деле это невероятно полезно, поскольку заставляет инженеров замечать предупреждения и действовать соответственно.

Если предупреждение о том, о чем на самом деле можно не беспокоиться, то обычно в языке есть механизм для подавления определенного предупреждения (без необходимости отключать все предупреждения). Например, если бы существовала веская причина для наличия неиспользуемой переменной в классе `userInfo`, то предупреждение могло бы быть подавлено. В следующем листинге показано, как это может выглядеть.

Листинг 4.33. Подавление предупреждения компилятора

```
class UserInfo {  
    private final String realName;  
  
    // displayName на данный момент не используется, так как мы  
    // переносим поставляемое  
    // используя настоящие имена. Это заполнитель, который будет  
    // использоваться  
    // скоро. Подробную информацию о миграции см. в записи # 7462.  
    @SuppressWarnings("unused") ❶  
    private final String displayName;  
  
    UserInfo(String realName, String displayName) {  
        this.realName = realName;  
        this.displayName = displayName;  
    }  
  
    String getRealName() {  
        return realName;  
    }  
}
```

```
String getDisplayName() {  
    return realName;  
}  
}
```

❶ Предупреждение подавлено.

Может возникнуть соблазн отклонить предупреждения компилятора как не столь важные; в конце концов, код все еще компилируется, поэтому легко предположить, что ничего катастрофически неправильного нет. Хотя предупреждения — это просто предупреждения, они часто могут быть признаком того, что в коде что-то не так, и в некоторых сценариях это может быть довольно серьезной ошибкой. Как показывают предыдущие примеры, полезно убедиться, что предупреждения компилятора замечены и обработаны. в идеале при создании нашего кода не должно быть предупреждений, потому что каждая проблема либо исправлена, либо явно устранена с обоснованным объяснением причин.

Итоги

- ◆ В целом существует два типа ошибок:
 - те, от которых система может восстановиться;
 - те, от которых система не может восстановиться.
- ◆ Часто только вызывающий фрагмент кода знает, можно ли исправить ошибку, сгенерированную этим кодом.
- ◆ Когда ошибка действительно возникает, хорошо бы получить быструю ошибку, а если ошибка не может быть устранена — громкую ошибку.
- ◆ Скрывать ошибку часто не очень хорошая идея, и лучше сигнализировать о том, что ошибка произошла.
- ◆ Способы сигнализации об ошибках можно разделить на две категории:
 - явный — в безошибочной части кодового соглашения. Вызывающие знают, что может произойти ошибка;
 - неявный — в мелком шрифте кодового соглашения или, возможно, вообще не в письменном соглашении. Вызывающие не обязательно знают, что может произойти ошибка.
- ◆ Ошибки, которые невозможно исправить, должны использовать неявные методы сигнализации об ошибках.
- ◆ Для ошибок, от которых потенциально можно восстановиться:
 - инженеры не все согласны с тем, следует ли использовать явные или неявные методы;
 - мое мнение таково, что следует использовать явные методы.
- ◆ Предупреждения компилятора часто могут указывать на то, что в коде что-то не так. Обращайте на них внимание.

Часть 2

В практике

Часть 1 установила величественно названные «шесть столпов качества кода». Они предоставляют высокоуровневые стратегии, которые могут помочь обеспечить высокое качество нашего кода. В *части 2* мы углубимся в первые пять из них более практичным способом.

Каждая глава *части 2* посвящена одному из «столпов» качества кода, причем каждый раздел в главах демонстрирует особое соображение или технику. Общая схема состоит в том, чтобы сначала показать общий способ, которым код может быть проблематичным, а затем показать, как конкретный метод может быть использован для улучшения ситуации. Каждый раздел в главах разработан таким образом, чтобы быть относительно автономным, и я надеюсь, что разделы могут послужить полезной справкой для любого, кто хочет объяснить конкретную концепцию или соображения другому инженеру (например, во время анализа кода).

Пожалуйста, обратите внимание, что список тем в каждой главе не является исчерпывающим. Например, в *главе 7* обсуждаются шесть конкретных тем, как сделать код таким, чтобы его неправильное использование было достаточно сложным. Это не означает, что это единственные шесть вещей, которые нам когда-либо нужно учитывать, чтобы сделать наш код сложным для неправильного использования. Но цель состоит в том, что, поняв причины, лежащие в основе этих шести пунктов, в сочетании с более теоретическими положениями, которые мы узнали в *части 1*, мы сможем развить более широкое чувство суждения, которое может направлять нас в любом сценарии, в котором мы можем оказаться.

5 Сделайте код читаемым

ЭТА ГЛАВА ОХВАТЫВАЕТ

- ◇ Методы создания кода, не требующего пояснений
- ◇ Обеспечение того, чтобы детали в коде были понятны другим
- ◇ Использование языковых функций по правильным причинам

Читаемость по своей сути является субъективной вещью, и поэтому трудно дать четкое определение тому, что именно это означает. Суть удобочитаемости заключается в том, чтобы инженеры могли быстро и точно понять, что делает код. На самом деле для достижения этого часто требуется быть чутким и пытаться представить, как все может быть запутанным или открытым для неправильного толкования, если смотреть на это с другой точки зрения.

Пятая глава должна дать прочную основу для некоторых наиболее распространенных и эффективных методов, позволяющих сделать код более читабельным. Однако стоит помнить, что каждый сценарий реальной жизни отличается и имеет свой собственный набор соображений, поэтому важно использовать здравый смысл и принимать правильные решения.

5.1. Используйте описательные имена

Имена необходимы для уникальной идентификации вещей, но они часто также дают краткое представление о том, что это за вещь. Слово *тостер* однозначно идентифицирует прибор на вашей кухне, но оно также дает мощный намек на то, что он делает: он поджаривает вещи. Если бы мы вместо этого настаивали на том, чтобы называть тостер «объектом А», тогда было бы довольно легко забыть, что именно такое «объект А» и что он делает.

Тот же принцип применяется при именовании объектов в коде. Имена необходимы для уникальной идентификации классов, функций и переменных. Но то, как мы их называем, также является прекрасной возможностью сделать код более читабельным, гарантируя, что на объекты ссылаются понятным образом.

5.1.1. Неописательные имена затрудняют чтение кода

Листинг 5.1 представляет собой несколько экстремальный пример того, как мог бы выглядеть код, если бы не было приложено никаких усилий для использования

описательных имен. Потратьте 20–30 секунд, глядя на него, и посмотрите, как трудно понять, что он делает.

Листинг 5.1. Неописательные имена

```
class T {
    Set<String> pns = new Set();
    Int s = 0;
    ...
    Boolean f(String n) {
        return pns.contains(n);
    }

    Int getS() {
        return s;
    }
}

Int? s(List<T> ts, String n) {
    for (T t in ts) {
        if (t.f(n)) {
            return t.getS();
        }
    }
    return null;
}
```

Если бы вас попросили описать, что делает этот код, что бы вы сказали? Если вы еще не заглянули вперед, вы, вероятно, понятия не имеете, что делает этот код или какие понятия представляют строки, целые числа и класс.

5.1.2. Комментарии являются плохой заменой описательных имен

Одним из способов улучшить это может быть добавление некоторых комментариев и документации. Если бы автор сделал это, то код мог бы выглядеть примерно так, как в листинге 5.2. Это немного улучшает ситуацию, но все еще остается куча проблем.

- ◆ Код теперь намного более загроможден, и автору и другим инженерам теперь приходится поддерживать все эти комментарии и документацию, а также сам код.
- ◆ Инженерам необходимо постоянно прокручивать файл вверх и вниз, чтобы разобраться в происходящем. Если инженер полностью находится в нижней части файла, просматривая функцию `getS()`, и забывает, для чего предназначена переменная `s`, то ему придется прокрутить весь путь до верхней части файла, чтобы

найти комментарий, объясняющий, что такое `s`. Если класс `T` имеет длину в несколько сотен строк, то это довольно быстро начинает раздражать.

- ◆ Если инженер смотрит на тело функции `s()`, все еще остается полной загадкой, что делает или возвращает вызов, подобный `t.f(n)`, если только он не пойдет и не посмотрит код для класса `T`.

Листинг 5.2. Комментарии вместо описательных имен

```
/** Представляет команду. */
class T {
    Set<String> pns = new Set(); // Имена игроков в команде.
    Int s = 0; // Счет команды.

    ...
    /**
     * @param n имя игроков
     * @return true если игрок в команде
     */
    Boolean f(String n) {
        return pns.contains(n);
    }

    /**
     * @return счет команды
     */
    Int getS() {
        return s;
    }
}

/**
 * @param ts это список всех команд
 * @param n имя игрока
 * @return счет команды, в которой находится игрок
 */
Int? s(List<T> ts, String n) {
    for (T t in ts) {
        if (t.f(n)) {
            return t.getS();
        }
    }
    return null;
}
```

Документация в листинге 5.2 вполне может оказаться полезной: описание того, какие параметры и типы возвращаемых значений представлены, может помочь дру-

гим инженерам понять, как использовать код. Но комментарии не должны использоваться вместо того, чтобы давать объектам описательные названия. В *разд. 5.2* более подробно обсуждается использование комментариев и документации.

5.1.3. Решение: сделайте имена описательными

Использование описательных имен может превратить непонятный код, который мы только что видели, во что-то, что внезапно становится очень простым для понимания. В следующем листинге показано, как выглядит код с описательными именами.

Листинг 5.3. Описательные имена

```
class Team {
    Set<String> playerNames = new Set();
    Int score = 0;
    ...
    Boolean containsPlayer(String playerName) {
        return playerNames.contains(playerName);
    }

    Int getScore() {
        return score;
    }
}

Int? getTeamScoreForPlayer(List<Team> teams, String playerName) {
    for (Team team in teams) {
        if (team.containsPlayer(playerName)) {
            return team.getScore();
        }
    }
    return null;
}
```

Код теперь намного проще для понимания:

- ◆ Переменные, функции и классы теперь понятны сами по себе.
- ◆ Фрагменты кода теперь имеют больше смысла, даже если рассматривать их изолированно; совершенно очевидно, что делает и возвращает вызов типа `team.containsPlayer(playerName)`, даже не глядя на код класса `Team`. Ранее этот вызов функции выглядел как `t.f(n)`, так что это явно значительное улучшение читаемости.

Код также менее загроможден, чем если бы использовались комментарии, и инженеры могут сосредоточиться на поддержании кода без необходимости поддерживать рядом с ним коллекцию комментариев.

5.2. Используйте комментарии надлежащим образом

Комментарии или документация в коде могут служить различным целям, таким, как следующие:

- ♦ объяснение *того*, что делает некоторый код;
- ♦ объяснение, *почему* некоторый код делает то, что он делает;
- ♦ предоставление другой информации, такой, как инструкции по использованию.

В этом разделе мы сосредоточимся на первых двух из них: использование комментариев для объяснения того, что и *почему*. Другая информация, такая, как инструкции по использованию, обычно является частью кодового соглашения и обсуждалась в *главе 3*.

Комментарии высокого уровня, обобщающие то, что делает большой кусок кода (например, класс), часто могут быть полезны. Однако, когда дело доходит до более низкого уровня, построчные сведения о коде, комментарии, объясняющие, что делает код, часто являются не самым эффективным способом сделать код читабельным.

Хорошо написанный код с описательными именами должен быть понятен с точки зрения того, что он делает на уровне строки за строкой. Если нам нужно добавить много низкоуровневых комментариев к нашему коду, чтобы объяснить, что он делает, то это, скорее всего, признак того, что наш код не так удобочитаем, как в идеале должно быть. Комментарии, которые объясняют, почему существует код, или которые предоставляют больше контекста, с другой стороны, часто могут быть весьма полезными, поскольку не всегда возможно прояснить это только с помощью одного кода.

ИСПОЛЬЗУЙТЕ ЗДРАВЫЙ СМЫСЛ

В этом разделе приведены некоторые общие рекомендации о том, как и когда использовать комментарии, но это не жесткие и быстрые правила. Мы должны использовать наш здравый смысл в отношении того, что делает код наиболее простым для понимания и поддержки. Если у нас нет выбора, кроме как включить некоторую грубую побитовую логику, или вынужденно использовать некоторые хитрые трюки для оптимизации нашего кода, то комментарии, объясняющие, что делает какой-то низкоуровневый код, вполне могут оказаться полезными.

5.2.1. Избыточные комментарии могут быть вредными

Код в листинге 5.4 создает идентификатор, соединяя имя с фамилией с помощью точки. Код использует комментарий, чтобы объяснить, что он делает, но код уже самоочевиден, поэтому комментарий бесполезен.

Листинг 5.4. Комментарий, объясняющий, что делает код

```
String generateId(String firstName, String lastName) {  
    // Выдает ID в форме "{first name}.{last name}".  
    return firstName + "." + lastName;  
}
```

Включение избыточного комментария, подобного этому, на самом деле может быть хуже чем бесполезным, потому что:

- ◆ теперь инженерам необходимо сохранить этот комментарий; если кто-то изменит код, им также необходимо не забыть обновить комментарий;
- ◆ это загромождает код: представьте, если бы с каждой строкой кода был связан такой комментарий. Чтение 100 строк кода теперь превращается в чтение 100 строк кода плюс 100 комментариев. Учитывая, что этот комментарий не добавляет никакой дополнительной информации, он просто тратит время инженеров впустую.

Вероятно, было бы лучше удалить этот комментарий и позволить объяснить смысл самому коду.

5.2.2. Комментарии являются плохой заменой читаемого кода

Код в листинге 5.5 снова генерирует ID, соединяя имя с фамилией с помощью точки. В этом примере код не является самоочевидным, поскольку имя и фамилия содержатся в первом и втором элементах массива соответственно.

Код включает комментарий, который объясняет это. В этом случае комментарий кажется полезным, потому что код сам по себе неясен, но реальная проблема здесь в том, что код не был сделан таким читабельным, каким он мог бы быть.

Листинг 5.5. Нечитаемый код, объясненный комментарием

```
String generateId(String[] data) {  
    // data[0] содержит имя пользователя, а data[1] содержит имя  
    пользователя  
    // last name. Выдает ID в форме "{first name}.{last name}".  
    return data[0] + "." + data[1];  
}
```

Поскольку комментарий необходим только потому, что сам код не очень удобочитаем, лучшим подходом, вероятно, было бы вместо этого сделать код более читабельным. В данном сценарии этого можно легко достичь с помощью некоторых хорошо названных вспомогательных функций, как показано в следующем листинге.

Листинг 5.6. Более читаемый код

```
String generateId(String[] data) {  
    return firstName(data) + "." + lastName(data);  
}  
  
String firstName(String[] data) {  
    return data[0];  
}  
  
String lastName(String[] data) {  
    return data[1];  
}
```

Создание кода, не требующего пояснений, часто предпочтительнее использования комментариев, поскольку это снижает затраты на обслуживание и устраняет вероятность того, что комментарии устареют.

5.2.3. Комментарии могут быть полезны для объяснения того, почему существует код

Код иногда менее хорош в самоописании, в случае если требуется объяснить, *почему* что-то делается. Причина, по которой существует определенный фрагмент кода, или причина, по которой он делает то, что он делает, иногда может быть связана с некоторым контекстом или знаниями, которые не обязательно будут известны другому инженеру, изучающему код. Комментарии могут быть очень полезны, когда подобный контекст важен для понимания кода или возможности его безопасно изменения. Примеры вещей, которые было бы полезно прокомментировать, чтобы объяснить, почему существует некоторый код, включают следующее.

- ◆ Продукт или бизнес-решение.
- ◆ Исправление странной, неочевидной ошибки.
- ◆ Работа с нелогичной причудой в зависимости.

В листинге 5.7 содержится функция для получения ID пользователя. Существует два различных способа создания ID в зависимости от того, когда пользователь зарегистрировался. Причина этого не была бы очевидна из одного только кода, поэтому для ее объяснения используются комментарии. Это предотвращает путаницу кода для других инженеров и гарантирует, что они будут знать, какие соображения применимы, если им потребуется изменить этот код.

Листинг 5.7. Комментарий, объясняющий, почему существует код

```
class User {  
    private final Int username;  
    private final String firstName;
```

```

private final String lastName;
private final Version signupVersion;
...

String getUserId() {
    if (signupVersion.isOlderThan("2.0")) {
        // Прежним пользователям (которые
        // зарегистрировались до версии 2.0) были назначены ❶
        // IDs основанные на их имени. Смотрите запись
        // #4218 для получения дополнительной информации ❶
        // Подробная информация
        return firstName.toLowerCase() + "." +
            lastName.toLowerCase();
    }
    // Новым пользователям (которые зарегистрировались
    // начиная с версии 2.0 и далее) назначаются
    // IDs основанные на их имени. ❶
    return username;
}
...
}

```

❶ Комментарии, объясняющие, почему существует некоторый код.

Это немного загромождает код, но выгода перевешивает эти затраты. Один только код без комментариев мог бы вызвать путаницу.

5.2.4. Комментарии могут содержать полезные аннотации высокого уровня

Комментарии и документация, объясняющие, что делает код, в чем-то похожи на краткий обзор при чтении книги.

- ◆ Если вы возьмете книгу и каждому абзацу на каждой странице будет предшествовать краткое изложение из одного предложения, получится довольно раздражающая и трудная книга для чтения. Это похоже на комментарии низкого уровня, объясняющие, что делает код, и вредит читабельности.
- ◆ С другой стороны, краткое изложение на задней обложке книги (или даже в начале каждой главы), в котором кратко излагается содержание, может быть невероятно полезным. Оно позволяет вам быстро оценить, является ли книга (или глава) полезной или интересной для вас. Это похоже на комментарии высокого уровня, обобщающие то, что делает класс, и позволяет инженеру очень легко оценить, полезен ли ему класс, или на что он может повлиять.

Ниже приведены некоторые примеры, в которых может быть полезна документация высокого уровня о том, что делает код.

- ◆ Документация, объясняющая на высоком уровне, что делает класс и любые важные детали, о которых должны быть осведомлены другие инженеры.

- ◆ Документация, объясняющая, что такое входной параметр функции или что он делает.
- ◆ Документация, объясняющая, что представляет собой возвращаемое значение функции.

Полезно помнить, что было сказано в *главе 3*: документация важна, но мы должны реалистично относиться к тому факту, что инженеры часто ее не читают. Лучше не полагаться на нее слишком сильно, чтобы избежать неожиданностей или предотвратить неправильное использование нашего кода (*главы 6 и 7* соответственно будут посвящены более надежным методам для этого).

В листинге 5.8 показано, как документация может быть использована для обобщения того, что делает класс пользователя в целом. Он предоставляет некоторые полезные сведения высокого уровня, например то, что он относится к пользователю «службы потоковой передачи» и потенциально может быть не синхронизирован с базой данных.

Листинг 5.8. Документация класса высокого уровня

```
/**
 * Инкапсулирует подробную информацию о пользователе потоковой службы.
 *
 * Это не имеет прямого доступа к базе данных, а вместо этого
 * создается со значениями, хранящимися в памяти.
 * Следовательно, он может не синхронизироваться с любыми
 * изменениями, внесенными в базу данных после создания класса.
 */
class User {
    ...
}
```

Комментарии и документация полезны для заполнения деталей, которые не может передать один код, или для обобщения того, что делает больший фрагмент кода. Недостатком является то, что их необходимо поддерживать, они могут легко устареть и загромождать код. Эффективное их использование — балансирование между этими плюсами и минусами.

5.3. Не зацикливайтесь на количестве строк кода

В общем, чем меньше строк кода в кодовой базе, тем лучше. Код обычно требует некоторого объема постоянного обслуживания, и большое количество строк кода иногда может свидетельствовать о том, что код слишком сложен или не использует существующие решения повторно. Большее количество строк кода также может увеличить когнитивную нагрузку на инженеров, потому что их явно больше для чтения.

Инженеры иногда доводят это до крайности и утверждают, что минимизация количества строк кода важнее любых других факторов, касающихся качества кода. Иногда высказывается жалоба на то, что так называемое улучшение качества кода превратило три строки кода в десять строк кода и, следовательно, сделало код хуже.

Однако важно помнить, что количество строк кода является косвенным показателем того, что нас действительно волнует, и, как и в случае большинства косвенных измерений, это полезный руководящий принцип, но не жесткое правило. То, о чем мы действительно заботимся, — это обеспечение того, чтобы код было:

- ◆ легко понять;
- ◆ трудно понять неправильно;
- ◆ трудно случайно сломать.

Не все строки кода одинаковы: одна чрезвычайно трудная для понимания строка кода может легко снизить качество кода по сравнению с наличием 10 (или даже 20) простых для понимания строк кода на ее месте. Следующие два подраздела демонстрируют это на примере.

5.3.1. Избегайте краткого, но нечитаемого кода

Чтобы продемонстрировать, как меньшее количество строк кода может быть менее читаемым, рассмотрим листинг 5.9. Он показывает функцию для проверки правильности 16-разрядного ID. Как только вы взглянете на код, спросите себя: сразу ли очевидно, каков критерий допустимости ID? Для большинства инженеров ответ на этот вопрос — нет.

Листинг 5.9. Краткий, но нечитаемый код

```
Boolean isValid(UInt16 id) {  
    return countSetBits(id & 0x7FFF) % 2 == ((id & 0x8000) >> 15);  
}
```

Код проверяет бит четности, который иногда используется для обнаружения ошибок при передаче данных. 16-разрядный идентификатор содержит 15-разрядное значение, хранящееся в наименее значимых 15 битах, и бит четности, хранящийся в наиболее значимом бите. Бит четности указывает, установлено ли четное или нечетное число битов в 15-битном значении.

Строки кода в листинге 5.9 не очень удобочитаемы или не требуют пояснений, и, несмотря на краткость, они содержат множество предположений и сложностей:

- ◆ Наименее значимые 15 бит ID содержат значение.
- ◆ Самый значительный бит ID содержит бит четности.
- ◆ Бит четности равен нулю, если в 15-битном значении задано четное число битов.
- ◆ Бит четности равен единице, если в 15-битном значении задано нечетное число битов.

- ◆ 0x7FFF — это битовая маска для наименее значимых 15 бит.
- ◆ 0x8000 — это битовая маска для наиболее значимого бита.

Проблемы с уплотнением всех этих деталей и предположений в одну строку очень краткого кода заключаются в следующем:

- ◆ другим инженерам приходится прилагать много усилий и ломать голову, чтобы выяснить и извлечь эти детали и предположения из одной строки кода. Это тратит их время впустую, а также увеличивает вероятность того, что они что-то не правильно поймут и в итоге нарушат код;
- ◆ эти предположения должны соответствовать некоторым предположениям, сделанным в других местах. Где-то еще есть код, который кодирует эти ID. Если он будет изменен, чтобы поместить бит четности в младший значащий бит (например), то код в листинге 5.9 перестанет функционировать правильно. Было бы лучше, если бы подзадачи, такие, как расположение бита четности, были разбиты на единый источник истины, который можно использовать повторно.

Код в листинге 5.9 может быть кратким, но он также почти нечитаем. Несколько инженеров потенциально потратят много времени, пытаясь понять, что делает код. Количество неочевидных и недокументированных предположений в коде приводит к хрупкости, его легко сломать.

5.3.2. Решение: сделайте код читабельным, даже если для этого требуется больше строк

Было бы намного лучше, если бы предположения и детали кодировки ID и бита четности были очевидны для любого, кто читает код, даже если для этого требуется использовать больше строк. В листинге 5.10 показано, как код можно сделать намного более читабельным. Он определяет некоторые хорошо известные вспомогательные функции и константы. Это делает код намного проще для понимания и гарантирует, что решения подзадач можно использовать повторно, но также приводит к появлению гораздо большего количества строк кода.

Листинг 5.10. Подробный, но читаемый код

```
Boolean isValid(UInt16 id) {
    return extractEncodedParity(id) ==
        calculateParity(getIdValue(id));
}

private const UInt16 PARITY_BIT_INDEX = 15;
private const UInt16 PARITY_BIT_MASK = (1 << PARITY_BIT_INDEX);
private const UInt16 VALUE_BIT_MASK = ~PARITY_BIT_MASK;

private UInt16 getIdValue(UInt16 id) {
    return id & VALUE_BIT_MASK;
}
```

```
private UInt16 extractEncodedParity(UInt16 id) {  
    return (id & PARITY_BIT_MASK) >> PARITY_BIT_INDEX;  
}  
  
// Четность равна 0, если задано четное число битов, и 1, если  
// устанавливается нечетное количество битов.  
private UInt16 calculateParity(UInt16 value) {  
    return countSetBits(value) % 2;  
}
```

Обычно полезно следить за количеством добавляемых строк кода, так как это может быть предупреждающим признаком того, что код не использует существующие решения повторно или что-то усложняет. Но часто важнее убедиться, что код понятен, надежен и вряд ли приведет к ошибкам. Если для эффективного выполнения этого требуется больше строк кода, то все в порядке.

5.4. Придерживайтесь последовательного стиля кодирования

Если мы пишем предложение, существуют определенные правила, которым мы должны следовать, если хотим создать что-то грамматически правильное. Кроме того, существуют и другие стилистические рекомендации, которым мы должны следовать, чтобы наше предложение было читабельным.

В качестве примера представьте, что мы пишем что-то о *программном обеспечении как услуге*. Обычно, если такие слова, как *a* и *as*, включены в аббревиатуру (или инициализм), они сокращаются с использованием символа нижнего регистра. Поэтому наиболее знакомая аббревиатура *программного обеспечения как услуги* пишется *SaaS*. Если бы мы написали аббревиатуру как *SAAS*, то любой, кто читает наш документ, мог бы задаться вопросом, не имеем ли мы в виду что-то другое, потому что они не ожидают, что *программное обеспечение как услуга* будет сокращено.

То же самое относится и к коду. Синтаксис языка и компилятор диктуют, что разрешено (немного похоже на правила грамматики), но, поскольку инженеры пишут код, у нас есть большая свобода в отношении того, какие стилистические соглашения мы принимаем.

5.4.1. Непоследовательный стиль кодирования может привести к путанице

В листинге 5.11 содержится некоторый код из класса для управления чатом между группой пользователей. Этот класс используется на сервере, который управляет несколькими групповыми чатами одновременно. Класс содержит функцию `end()`, которая при вызове должна завершить чат, прервав соединения со всеми пользователями в этом чате.

Общее соглашение о стиле при написании кода заключается в том, что имена классов записываются в *PascalCase* (первая буква с заглавной буквы), тогда как имена переменных записываются в *camelCase* (первая буква в нижнем регистре). Поэтому, не видя всего определения класса, очевидным предположением было бы, что `connectionManager` является переменной экземпляра в классе `GroupChat`. Вызов `ConnectionManager.terminateAll()`, следовательно, должен прервать соединения для данного чата, но оставить другие чаты, администрируемые сервером, без изменений.

Листинг 5.11. Непоследовательный стиль именования

```
class GroupChat {  
    ...  
  
    end() {  
        connectionManager.terminateAll(); ❶  
    }  
}
```

❶ Мы предположили, что `ConnectionManager` является переменной экземпляра.

К сожалению, наше предположение неверно, и этот код очень сильно поврежден. `ConnectionManager` не является переменной экземпляра; на самом деле это класс, а `terminateAll()` — статическая функция для него. Вызов `connectionManager.terminateAll()` завершает все соединения для каждого чата, которым управляет сервер, а не только для того, которое связано с конкретным экземпляром класса `GroupChat`. В следующем листинге показан код класса `connectionManager`.

Листинг 5.12. Класс `ConnectionManager`

```
class connectionManager {  
    ...  
    static terminateAll() { ❶  
        ...  
    }  
}
```

❶ Завершает все соединения, которыми в настоящее время управляет сервер.

Эта ошибка, вероятно, была бы обнаружена (и предотвращена), если бы класс `connectionManager` следовал стандартному соглашению об именах и вместо этого назывался `ConnectionManager`. Если не придерживаться этого соглашения, код, использующий класс `connectionManager`, легко неправильно понять, и это может привести к серьезной ошибке, которая может остаться незамеченной.

5.4.2. Решение: примите руководство по стилю и следуйте ему

Как только что отмечалось, общепринятое соглашение о стиле кодирования состоит в том, что имена классов должны быть написаны в *PascalCase*, а имена переменных должны быть написаны в *camelCase*. Если бы это соглашение соблюдалось, то-

гда класс `connectionManager` вместо этого назывался бы `ConnectionManager`. Код с ошибками из последнего раздела будет выглядеть как листинг 5.13. Теперь совершенно очевидно, что `ConnectionManager` — это класс, а не переменная экземпляра в классе `GroupChat`. Также теперь очевидно, что такой вызов, как `ConnectionManager.terminateAll()`, вероятно, изменяет какое-то глобальное состояние и, вероятно, влияет на другие части сервера.

Листинг 5.13. Согласованный стиль именования

```
class GroupChat {  
    ...  
  
    end() {  
        ConnectionManager.terminateAll(); ❶  
    }  
}
```

❶ Очевидно, что `ConnectionManager` — это класс, а не переменная экземпляра.

Это всего лишь один пример того, как согласованный стиль кодирования может сделать код более читабельным и помочь предотвратить ошибки. Стили кодирования часто охватывают гораздо больше аспектов, чем просто то, как называть вещи, например, следующие:

- ◆ использование определенных языковых функций;
- ◆ отступ в коде;
- ◆ структурирование пакетов и каталогов;
- ◆ документирование кода.

В большинстве организаций и команд есть руководство по стилю кодирования, которому, как они ожидают, будут следовать инженеры, поэтому вряд ли вам нужно принимать какие-либо решения или слишком усердно думать о том, какой стиль выбрать. Скорее всего, это просто случай чтения и усвоения руководства по стилю, которое предписывает ваша команда, и следования ему.

Если у вашей команды нет руководства по стилю и вы хотели бы иметь его, то есть много готовых, которые вы можете принять. Google, например, опубликовал руководства по стилю для ряда языков: <https://google.github.io/styleguide/>.

Когда вся команда или организация придерживаются одного и того же стиля кодирования, это сродни тому, что все они свободно говорят на одном языке. Риск недопонимания друг друга значительно снижается, и это приводит к меньшему количеству ошибок и меньшему количеству времени, потраченного на попытки понять запутанный код.

Линтеры

Также доступны инструменты, которые могут информировать нас о любых нарушениях руководства по стилю, которые могут содержаться в нашем коде. Они на-

зываются линтерами и обычно специфичны для любого языка, который мы используем. Некоторые линтеры делают гораздо больше, чем просто проверяют нарушения руководства по стилю: они могут предупреждать нас о коде, который подвержен ошибкам или демонстрирует известные плохие методы.

Линтеры, как правило, улавливают только довольно простые проблемы, поэтому они, конечно, не заменяют хороший код. Но запуск линтера может быть быстрым и простым путем определить способы улучшения кода.

5.5. Избегайте глубоко вложенного кода

Типичный фрагмент кода состоит из блоков, которые вложены друг в друга, например:

- ◆ функция определяет блок кода, который выполняется при вызове функции;
- ◆ оператор `if` определяет блок кода, который выполняется, когда условие истинно;
- ◆ цикл `for` определяет блок кода, который выполняется на каждой итерации цикла.

На рис. 5.1 показано, как логика потока управления (например, операторы `if` и циклы `for`) может приводить к блокам кода, вложенным друг в друга. Обычно существует несколько способов структурирования заданной части логики в коде. Некоторые формы могут привести к большому количеству вложенных блоков кода, в то время как другие могут привести практически к отсутствию вложенности. Важно учитывать, как структура кода влияет на читаемость.

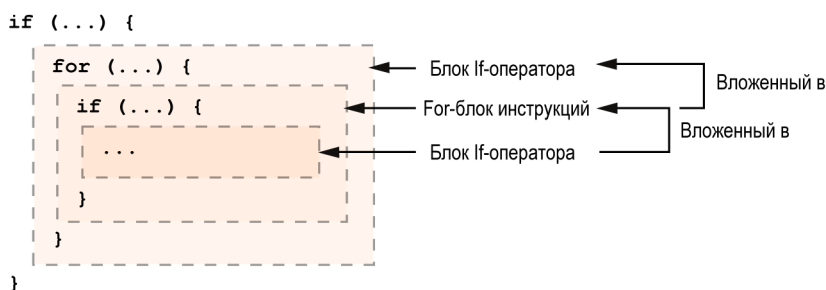


Рис. 5.1. Логика потока управления (например, операторы `if` и циклы `for`) часто может приводить к блокам кода, вложенным друг в друга

5.5.1. Глубоко вложенный код может быть трудночитаемым

В листинге 5.14 содержится код для поиска адреса владельца транспортного средства. Код содержит несколько операторов `if`, вложенных друг в друга. Это приводит к тому, что код довольно трудно читать — и потому, что за ним трудно усле-

дить, и из-за всей плотной логики «если — то», по которой необходимо перемещаться, чтобы определить, когда возвращаются определенные значения.

Листинг 5.14. Глубоко вложенные операторы `if`

```
Address? getOwnersAddress(Vehicle vehicle) {
    if (vehicle.hasBeenScraped()) {
        return SCRAPYARD_ADDRESS;
    } else {
        Purchase? mostRecentPurchase =
            vehicle.getMostRecentPurchase();
        if (mostRecentPurchase == null) { ❶
            return SHOWROOM_ADDRESS;
        } else {
            Buyer? buyer = mostRecentPurchase.getBuyer();
            if (buyer != null) {
                return buyer.getAddress();
            }
        }
    }
    return null; ❷
}
```

❶ Операторы `If`, вложенные в другие операторы `if`.

❷ Трудно понять сценарии, в которых эта линия достижима.

Человеческие глаза не очень хорошо отслеживают, какой именно уровень вложенности имеет каждая строка кода. Это может затруднить любому, кто читает код, точное понимание того, когда выполняются различные части логики. Глубоко вложенный код снижает читабельность, и часто лучше структурировать код таким образом, чтобы свести к минимуму объем вложенности.

5.5.2. Решение: реструктуризация для минимизации вложенности

Когда у нас есть такая функция, как в предыдущем примере, часто бывает довольно легко перестроить логику, чтобы избежать вложенности операторов `if` друг в друга. В листинге 5.15 показано, как эту функцию можно переписать без вложенности операторов `if`. Код более удобочитаем, так как за ним легче следить, а логика менее жесткая.

Листинг 5.15. Код с минимальной вложенностью

```
Address? getOwnersAddress(Vehicle vehicle) {
    if (vehicle.hasBeenScraped()) {
        return SCRAPYARD_ADDRESS;
    }
}
```

```

Purchase? mostRecentPurchase =
    vehicle.getMostRecentPurchase();
if (mostRecentPurchase == null) {
    return SHOWROOM_ADDRESS;
}
Buyer? buyer = mostRecentPurchase.getBuyer();
if (buyer != null) {
    return buyer.getAddress();
}
return null;
}

```

Когда каждая ветвь вложенной логики приводит к оператору `return`, обычно довольно легко изменить логику, чтобы избежать вложенности. Однако, когда ветви вложенности не приводят к операторам `return`, обычно это признак того, что функция делает слишком много. В следующем подразделе рассмотрим это.

5.5.3. Вложенность часто является результатом слишком больших усилий

В листинге 5.16 показана функция, которая делает слишком много, поскольку содержит как логику поиска адреса владельца транспортного средства, так и логику, которая использует адрес для запуска отправки письма. Из-за этого непросто применить исправление, описанное в предыдущем подразделе, так как раннее возвращение из функции, очевидно, будет означать, что письмо не отправлено.

Листинг 5.16. Функция, которая делает слишком много

```

SentConfirmation? sendOwnerALetter(
    Vehicle vehicle, Letter letter) {
    Address? ownersAddress = null;
    if (vehicle.hasBeenScraped()) {
        ownersAddress = SCRAPYARD_ADDRESS;
    } else {
        Purchase? mostRecentPurchase =
            vehicle.getMostRecentPurchase();
        if (mostRecentPurchase == null) { ❶❷
            ownersAddress = SHOWROOM_ADDRESS;
        } else {
            Buyer? buyer = mostRecentPurchase.getBuyer();
            if (buyer != null) {
                ownersAddress = buyer.getAddress();
            }
        }
    }
}

```



```

    if (ownersAddress == null) {
        return null;
    }
    return sendLetter(ownersAddress, letter); ❸
}

```

❶ Изменяемая переменная для хранения результата поиска адреса.

❷ Операторы If, вложенные в другие операторы if.

❸ Логика, использующая адрес.

Реальная проблема здесь в том, что эта функция делает слишком много. Он содержит основную логику для поиска адреса, а также логику для запуска отправки письма. Мы можем решить эту проблему, разбив код на более мелкие функции, которые будут показаны в следующем подразделе.

5.5.4. Решение: разбить код на более мелкие функции

Код из предыдущего подраздела можно улучшить, разбив логику поиска адреса владельца на другую функцию. Затем становится легко применить исправление, которое мы видели ранее в этом разделе, чтобы устранить вложенность операторов if. В следующем листинге показано, как это будет выглядеть.

Листинг 5.17. Меньшие функции

```

SentConfirmation? sendOwnerALetter(
    Vehicle vehicle, Letter letter) {
    Address? ownersAddress = getOwnersAddress(vehicle); ❶
    if (ownersAddress != null) {
        return sendLetter(ownersAddress, letter);
    }
    return null;
}

Address? getOwnersAddress(Vehicle vehicle) {
    if (vehicle.hasBeenScraped()) {
        return SCRAPYARD_ADDRESS;
    }
    Purchase? mostRecentPurchase =
        vehicle.getMostRecentPurchase();
    if (mostRecentPurchase == null) { ❷
        return SHOWROOM_ADDRESS;
    }
    Buyer? buyer = mostRecentPurchase.getBuyer();
    if (buyer == null) {

```

```
        return null;
    }
    return buyer.getAddress();
}
```

- ❶ Логика поиска адреса владельца в отдельной функции.
- ❷ Вложенность операторов `if` устранена.

В *главе 2* обсуждалось, как слишком много работы внутри одной функции может привести к плохим слоям абстракции, поэтому часто все равно рекомендуется разбивать большие функции даже без большого количества вложений. Когда в коде много вложенности, разбиение большой функции становится вдвойне важным, потому что часто это необходимый первый шаг для устранения вложенности.

5.6. Сделайте вызовы функций читаемыми

Если функция хорошо названа, то должно быть очевидно, что она делает, но даже для хорошо названной функции легко получить нечитаемые вызовы функций, если неясно, для чего нужны аргументы или что они делают.

БОЛЬШОЕ КОЛИЧЕСТВО ПАРАМЕТРОВ

Вызовы функций, как правило, становятся менее читаемыми по мере увеличения числа аргументов. Если функция или конструктор имеют большое количество параметров, это часто может быть признаком более фундаментальной проблемы с кодом, такой, как отсутствие определения соответствующих слоев абстракции или недостаточная модульность. В *главе 2* уже обсуждались слои абстракции, а в *главе 8* модульность будет рассмотрена более подробно.

5.6.1. Аргументы могут быть трудны для расшифровки

Рассмотрим следующий фрагмент кода, содержащий вызов функции, которая отправляет сообщение. Неясно, что представляют аргументы в вызове функции. Мы можем предположить, что `"hello"`, вероятно, и есть сообщение, но мы понятия не имеем, что означают `1` или `true`.

```
sendMessage("hello", 1, true);
```

Чтобы выяснить, что означают аргументы `1` и `true` в вызове `SendMessage()`, нам нужно было бы пойти и посмотреть на определение функции. Если бы мы это сделали, мы бы увидели, что `1` представляет приоритет сообщения, а значение `true` указывает, что отправка сообщения может быть повторена:

```
void sendMessage(String message, Int priority, Boolean allowRetry) {
    ...
}
```

Значения в вызове функции помогает понять определение функции, поиск которой может быть довольно трудоемкой задачей: определение функции вполне может на-

ходиться в совершенно другом файле или в сотнях строк кода. Если нам нужно обратиться к другому файлу или чему-то, состоящему из множества строк, чтобы выяснить, что делает данный фрагмент кода, то этот код не очень удобочитаем. Существует несколько потенциальных способов улучшить это, и в следующих подразделах рассматриваются некоторые из таких решений.

5.6.2. Решение: используйте именованные аргументы

Именованные аргументы поддерживаются все большим числом языков, особенно более современными. При использовании именованных аргументов в вызове функции аргументы сопоставляются на основе их имени, а не их положения в списке аргументов. Если бы мы использовали именованные аргументы, то вызов функции `SendMessage()` был бы очень удобочитаемым, даже если бы мы никогда не видели определения функции:

```
sendMessage(message: "hello", priority: 1, allowRetry: true);
```

К сожалению, не все языки поддерживают именованные аргументы, поэтому этот вариант возможен только в том случае, если мы используем язык, который их поддерживает. Несмотря на это, иногда существуют способы подделки именованных аргументов. Это довольно распространено в TypeScript (и других формах JavaScript) с использованием *деструктурирования объектов*. В листинге 5.18 показано, как функция `SendMessage()` могла бы использовать деструкцию объектов, если бы она была написана на машинописном языке. Функция принимает один объект (с типом `SendMessageParams`) в качестве параметра, но этот объект немедленно раскладывается на составляющие его свойства. Код внутри функции может затем непосредственно считывать эти свойства.

Листинг 5.18. Разрушение объектов в машинописном тексте

```
interface SendMessageParams { ❶
  message: string,
  priority: number,
  allowRetry: boolean,
}

async function sendMessage(
  {message, priority, allowRetry} : SendMessageParams) { ❷
  const outcome = await XhrWrapper.send(
    END_POINT, message, priority); ❸
  if (outcome.failed() && allowRetry) {
    ...
  }
}
```

❶ Интерфейс для определения типа параметра функции.

❷ Параметр функции немедленно преобразуется в его свойства.

❸ Свойства структурированного объекта можно использовать напрямую.

Следующий фрагмент показывает, как будет выглядеть вызов функции `SendMessage()`. Функция вызывается с объектом, то есть каждое значение связано с именем свойства. Это обеспечивает более или менее то же самое, что и именованные аргументы.

```
sendMessage({
    message: 'hello', ❶
    priority: 1,
    allowRetry: true,
});
```

❶ Имена аргументов связаны с каждым значением.

Использование деструктурированного объекта для достижения тех же преимуществ, что и именованные аргументы, относительно распространено в TypeScript (а также в других формах JavaScript), поэтому, несмотря на то что это немного обходной путь, обычно это то, с чем другие инженеры будут знакомы. Иногда существуют способы подделки именованных аргументов на других языках, но они могут вызвать проблемы, а не решить их, если они используют языковую функцию, с которой другие инженеры могут быть незнакомы.

5.6.3. Решение: используйте описательные типы

Независимо от того, поддерживает ли используемый нами язык именованные аргументы или нет, часто бывает полезно использовать более описательные типы при определении функции. В сценарии, который мы видели в начале этого раздела (повторяется снова в следующем фрагменте), автор функции `SendMessage()` использовал целое число для представления приоритета и логическое значение для представления того, разрешены ли повторные попытки.

```
void sendMessage(String message, Int priority, Boolean allowRetry) {
    ...
}
```

Целые числа и логические значения сами по себе не очень описательны, потому что они могут означать все что угодно в зависимости от сценария. Альтернативой может быть использование типов, описывающих, что они представляют при написании функции `SendMessage()`. В листинге 5.19 показаны два различных метода достижения этой цели:

Класс — приоритет сообщения был заключен в класс.

Перечисление — политика повторных попыток теперь использует перечисление с двумя параметрами вместо `Boolean`.

Листинг 5.19. Описательные типы в вызове функции

```
class MessagePriority {
    ...
    MessagePriority(Int priority) { ... }
    ...
}
```

```
enum RetryPolicy {
    ALLOW_RETRY,
    DISALLOW_RETRY
}

void sendMessage(
    String message,
    MessagePriority priority,
    RetryPolicy retryPolicy) {
    ...
}
```

Вызов этой функции тогда был бы очень удобочитаемым, даже без знания определения функции:

```
sendMessage("hello", new MessagePriority(1), RetryPolicy.ALLOW_RETRY);
```

5.6.4. Иногда нет хорошего решения

Иногда нет особенно хорошего способа убедиться, что вызовы функций читаемы. Чтобы продемонстрировать это, представьте, что нам нужен класс для представления двумерной ограничительной рамки. Мы могли бы написать код для класса `BoundingBox`, аналогичный тому, что приведен в листинге 5.20. Конструктор принимает четыре целых числа для представления положения краев рамки.

Листинг 5.20. Класс `BoundingBox`

```
class BoundingBox {
    ...
    BoundingBox(Int top, Int right, Int bottom, Int left) {
        ...
    }
}
```

Если язык, который мы используем, не поддерживает именованные аргументы, то вызов этого конструктора не очень удобочитаем, потому что он содержит серию чисел без намеков на то, что представляет каждое число. Поскольку все аргументы являются целыми числами, инженеру также очень легко запутаться в порядке, создать полный беспорядок и при этом иметь код, который отлично компилируется. В следующем фрагменте показан пример вызова конструктора `BoundingBox`:

```
BoundingBox box = new BoundingBox(10, 50, 20, 5);
```

В этом сценарии нет особенно удовлетворительного решения, и лучшее, что мы можем здесь сделать, — это использовать некоторые встроенные комментарии при вызове конструктора, чтобы объяснить, что такое каждый аргумент.

Если бы мы это сделали, то вызов конструктора выглядел бы примерно так:

```
BoundingBox box = new BoundingBox(  
    /* top= */ 10,  
    /* right= */ 50,  
    /* bottom= */ 20,  
    /* left= */ 5);
```

Встроенные комментарии определенно делают вызов конструктора более читабельным. Но они полагаются на то, что мы не допустили ошибки при их написании, и на то, что другие инженеры поддерживают их в актуальном состоянии, поэтому качество такого решения не является достаточно удовлетворительным. Также есть аргумент в пользу того, чтобы не использовать эти встроенные комментарии из-за риска того, что они устареют; устаревший (и, следовательно, неправильный) комментарий, вероятно, хуже, чем отсутствие комментариев.

Добавление функций настройки или использование чего-то вроде шаблона компоновщика (рассмотренного в *главе 7*) являются альтернативными вариантами, но у обоих есть недостаток, заключающийся в том, что они упрощают неправильное использование кода, позволяя создавать экземпляры класса с отсутствующими значениями. Это необходимо предотвратить с помощью проверки во время выполнения (а не проверки во время компиляции), чтобы убедиться в правильности кода.

5.6.5. Как насчет IDE?

Некоторые интегрированные среды разработки (*integrated development environments*, IDE) просматривают определения функций в фоновом режиме. Затем они расширяют представление кода, чтобы имена аргументов функций отображались на сайтах вызовов. На рис. 5.2 показано, как это может выглядеть.

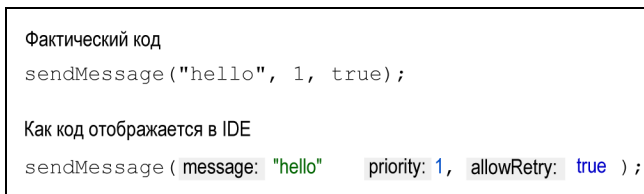


Рис. 5.2. Некоторые IDE расширяют представление кода, чтобы сделать вызовы функций более удобочитаемыми

Хотя это может быть невероятно полезно при редактировании кода, обычно лучше не полагаться на это, чтобы сделать код читабельным. Не гарантируется, что каждый инженер использует IDE, которая делает это, и, вероятно, существуют другие инструменты, в которых просматривается код, которые не имеют этой функции, такие, как инструменты проводника базы кода, инструменты слияния и средства проверки кода.

5.7. Избегайте использования необъяснимых значений

Существует множество сценариев, в которых требуется жестко заданное значение. Приведем распространенные примеры.

- ◆ Коэффициент для преобразования одной величины в другую.
- ◆ Число, которое является настраиваемым параметром, например, максимальное количество попыток повторить определенную задачу в случае ее сбоя.
- ◆ Строка, представляющая шаблон, в котором могут быть заполнены некоторые значения.

Для всех этих жестко закодированных значений есть две важные части информации:

- ◆ *каково значение* — компьютер должен знать это, когда он выполняет код;
- ◆ *что означает значение* — инженер должен знать это, чтобы понять код. Без этой информации код не читается.

Очевидно, что значение существует, иначе код, по-видимому, не будет компилироваться или функционировать, но достаточно легко забыть разъяснить другим инженерам, что на самом деле означает значение.

5.7.1. Необъяснимые значения могут сбивать с толку

В листинге 5.21 показаны некоторые функции класса для представления транспортного средства (*vehicle*). Функция `getKineticEnergyJ()` вычисляет текущую кинетическую энергию транспортного средства в джоулях (Дж) на основе массы и скорости транспортного средства. Масса транспортного средства хранится в тоннах США, а скорость — в милях в час (миль/ч). Уравнение для расчета кинетической энергии в джоулях ($\frac{1}{2} \cdot m \cdot v^2$) требует, чтобы масса была в килограммах, а скорость — в метрах в секунду, поэтому `getKineticEnergyJ()` содержит два коэффициента преобразования. Что означают эти коэффициенты, не очевидно из кода; любой, кто еще не знаком с уравнением кинетической энергии, скорее всего, не будет знать, что представляют собой эти константы.

Листинг 5.21. Класс `Vehicle`

```
class Vehicle {  
    ...  
  
    Double getMassUsTon() { ... }  
  
    Double getSpeedMph() { ... }  
  
    // Возвращает текущую кинетическую энергию транспортного  
    // средства в джоулях.  
    Double getKineticEnergyJ() {
```

```

        return 0.5 *
            getMassUsTon() * 907.1847 * ❶
            Math.pow(getSpeedMph() * 0.44704, 2); ❷
    }
}

```

❶ Необъяснимая величина, которая преобразует наши тонны в килограммы.

❷ Необъяснимое значение, которое преобразует мили в час в метры в секунду.

Наличие необъяснимых значений, подобных этому, делает код менее читаемым, потому что многие инженеры не поймут, почему эти значения существуют или что они делают. Когда инженеру приходится изменять какой-то код, который он не понимает, вероятность того, что код будет нарушен, возрастает.

Представьте, что инженер изменяет класс транспортного средства, чтобы избавиться от функции `getMassUsTon()` и заменить ее функцией `getMassKg()`, которая возвращает массу в килограммах. Они должны изменить вызов `getmassuston()` в `getKineticEnergyJ()`, чтобы вызвать эту новую функцию. Но поскольку он не понимает, что значение 907,1847 переводит тонны США в килограммы, он может не понимать, что теперь ему также нужно удалить и это. После проведенной им модификации функция `getKineticEnergyJ()` будет нарушена:

```

...
// Возвращает текущую кинетическую энергию транспортного
// средства в джоулях.
Double getKineticEnergyJ() {
    return 0.5 *
        getMassKg() * 907.1847 * ❶
        Math.pow(getSpeedMph() * 0.44704, 2);
}
...

```

❶ 907.1847 не было удалено; функция возвращает неверное значение.

Наличие необъяснимого значения в коде может привести к путанице и ошибкам. Важно убедиться, что означает значение для других инженеров. В следующих двух подразделах показаны различные способы, с помощью которых этого можно достичь.

5.7.2. Решение: используйте хорошо названную константу

Один из простых способов объяснить значения — дать им имя, поместив их внутри константы. Вместо того чтобы использовать значение непосредственно в коде, используется константа, т. е. ее имя объясняет код. В следующем листинге показано, как будет выглядеть функция `getkineticenergy()` и окружающий класс `Vehicle`, если значения будут помещены внутри констант.

Листинг 5.22. Хорошо названные константы

```
class Vehicle {
    private const Double KILOGRAMS_PER_US_TON = 907.1847; ❶
    private const Double METERS_PER_SECOND_PER_MPH = 0.44704;
    ...

    // Возвращает текущую кинетическую энергию транспортного
    // средства в джоулях.
    Double getKineticEnergyJ() {
        return 0.5 *
            getMassUsTon() * KILOGRAMS_PER_US_TON * ❷
            Math.pow(getSpeedMph() *
                METERS_PER_SECOND_PER_MPH, 2);
    }
}
```

❶ Определения констант.

❷ Использование констант в коде.

Код теперь намного более удобочитаем, и, если инженер изменит класс `Vehicle`, чтобы использовать килограммы вместо американских тонн, для него, вероятно, будет очевидно, что `KILOGRAMS_PER_US_TON` больше не является правильным.

5.7.3. Решение: используйте хорошо названную функцию

Альтернативой использованию хорошо названной константы является использование хорошо названной функции. Существует два альтернативных способа использования функции для обеспечения читаемости кода:

- ◆ функция поставщика, которая возвращает константу;
- ◆ вспомогательная функция, которая выполняет преобразование.

Поставщик функций

Концептуально это почти то же самое, что и использование константы, только достигается немного другим способом. В следующем листинге показана функция `getKineticEnergyJ()` и две дополнительные функции для обеспечения коэффициентов преобразования: `kilogramsPerUsTon()` и `metersPerSecondPerMph()`.

Листинг 5.23. Хорошо названные функции для предоставления значений

```
class Vehicle {
    ...
    // Возвращает текущую кинетическую энергию транспортного
    // средства в джоулях.
    Double getKineticEnergyJ() {
        return 0.5 *
            getMassUsTon() * kilogramsPerUsTon() * ❶
    }
}
```

```

        Math.pow(getSpeedMph() * metersPerSecondPerMph(),
            2);
    }

    private static Double kilogramsPerUsTon() { ❷
        return 907.1847;
    }

    private static Double metersPerSecondPerMph() {
        return 0.44704;
    }
}

```

❶ Вызванные функции поставщика.

❷ Функции поставщика.

Вспомогательная функция

Альтернативой было бы рассматривать преобразование величин как подзадачи, которые должны решаться с помощью специальных функций. Тот факт, что значение участвует в конкретном преобразовании, является деталью реализации, о которой вызывающим не нужно знать. В следующем листинге показана функция `getKineticEnergy()` и две дополнительные функции, которые решают подзадачи преобразования: `usTonsToKilograms()` и `mphToMetersPerSecond()`.

Листинг 5.24. Вспомогательные функции для выполнения преобразований

```

class Vehicle {
    ...
    // Возвращает текущую кинетическую энергию транспортного
    // средства в джоулях.
    Double getKineticEnergyJ() {
        return 0.5 *
            usTonsToKilograms(getMassUsTon()) * ❶
            Math.pow(mphToMetersPerSecond(getSpeedMph()), 2);
    }

    private static Double usTonsToKilograms(Double usTons) { ❷
        return usTons * 907.1847;
    }

    private static Double mphToMetersPerSecond(Double mph) {
        return mph * 0.44704;
    }
}

```

❶ Вызванные вспомогательные функции.

❷ Вспомогательные функции.

Как показывают предыдущие примеры, есть три хороших способа избежать необъяснимых значений в нашем коде. Как правило, для размещения значения внутри константы или функции требуется очень мало дополнительной работы, и это может значительно улучшить читаемость.

В качестве заключительного момента также стоит подумать, не захотят ли другие инженеры повторно использовать значение или вспомогательную функцию, которую мы определяем. Если есть шанс, что они могут это сделать, то лучше разместить его где-нибудь в открытом полезном классе, а не просто хранить внутри класса, в котором мы его используем.

5.8. Используйте анонимные функции надлежащим образом

Анонимные функции — это функции, у которых нет имени, и они обычно определяются встроенным кодом в тот момент, когда они необходимы. Синтаксис определения анонимной функции различается в разных языках. В листинге 5.25 показана функция для получения всех отзывов, содержащих непустой комментарий. Он вызывает функцию `List.filter()`, используя анонимную функцию. Для полноты картины в листинге также показано, как может выглядеть функция фильтра в классе `List`. `List.filter()` принимает функцию в качестве параметра, и вызывающие абоненты могут предоставить здесь анонимную функцию, если захотят.

Листинг 5.25. Анонимная функция в качестве аргумента

```
class List<T> {  
    ...  
    List<T> filter(Function<T, Boolean> retainIf) { ❶  
        ...  
    }  
}  
  
List<Feedback> getUsefulFeedback(List<Feedback> allFeedback) {  
    return allFeedback  
        .filter(feedback -> !feedback.getComment().isEmpty()); ❷  
}
```

❶ Принимает функцию в качестве параметра.

❷ `List.filter()` вызывается с помощью встроенной анонимной функции.

Большинство основных языков в той или иной форме поддерживают анонимные функции. Использование их для небольших, не требующих пояснений целей может повысить читабельность кода, но использование их для чего-либо большого, не требующего пояснений, или того, что может быть использовано повторно, может вызвать проблемы. В следующих подразделах объясняется почему.

Функциональное программирование

Анонимные функции и использование функций в качестве параметров — это методы, наиболее часто связанные с функциональным программированием и, в частности, с использованием лямбда-выражений. Функциональное программирование — это парадигма, в которой логика выражается в виде вызовов или ссылок на функции, а не в виде императивных утверждений, изменяющих состояние. Существует ряд языков, которые являются «чистыми» функциональными языками программирования. Языки, к которым эта книга наиболее применима, не будут считаться чисто функциональными языками. Тем не менее большинство из них включают функции, которые позволяют писать код в функциональном стиле во многих сценариях.

Если вы хотите узнать больше о функциональном программировании, следующая статья содержит более подробное описание: <http://mng.bz/qewE>

5.8.1. Анонимные функции могут отлично подходить для небольших задач

Код, который мы только что видели (повторяется в следующем фрагменте), использует анонимную функцию для получения фрагментов обратной связи, содержащих непустой комментарий. Для этого требуется только один оператор кода, и поскольку решаемая проблема тривиальна, этот единственный оператор очень удобочитаем и компактен.

```
List<Feedback> getUsefulFeedback(List<Feedback> allFeedback) {  
    return allFeedback  
        .filter(feedback -> !feedback.getComment().isEmpty()); ❶  
}
```

❶ Анонимная функция для проверки комментария непустая.

В этом случае использование анонимной функции подходит, потому что логика в ней небольшая, простая и понятная. Альтернативой было бы определить именованную функцию для определения того, содержит ли часть обратной связи непустой комментарий. В листинге 5.26 показано, как будет выглядеть код, если вместо него будет использоваться именованная функция. Для определения именованной функции требуется больше стандартных шаблонов, которые некоторые инженеры могут счесть менее удобочитаемыми.

Листинг 5.26. Именованная функция в качестве аргумента

```
List<Feedback> getUsefulFeedback(List<Feedback> allFeedback) {  
    return allFeedback.filter(hasNonEmptyComment); ❶  
}  
  
private Boolean hasNonEmptyComment(Feedback feedback) { ❷  
    return !feedback.getComment().isEmpty();  
}
```

❶ Именованная функция, используемая в качестве аргумента.

❷ Именованная функция.

ПРИМЕЧАНИЕ

Даже с такой простой логикой, как в листинге 5.26, определение выделенной именованной функции все равно может быть полезно с точки зрения возможности повторного использования кода. Если есть хоть какой-то шанс, что кому-то может понадобиться повторно использовать логику, чтобы проверить, есть ли в отзыве непустой комментарий, то, возможно, лучше поместить его в именованную функцию, а не в анонимную.

5.8.2. Анонимные функции могут быть трудночитаемыми

Как было рассмотрено ранее в этой главе (а также ранее в книге), имена функций могут быть очень полезны для повышения читаемости кода, поскольку они предоставляют краткое описание того, что делает код внутри функции. Поскольку анонимные функции по определению безымянны, они не предоставляют никому, кто читает код, эту сводку. Независимо от того, насколько она мала, если содержимое анонимной функции не является самоочевидным, то код, скорее всего, не читается.

В листинге 5.27 показан код функции, которая принимает список 16-разрядных идентификаторов и возвращает только допустимые. Формат идентификатора представляет собой 15-битное значение в сочетании с битом четности, и идентификатор считается действительным, если он ненулевой и бит четности правильный. Логика проверки бита четности находится внутри анонимной функции, но то, что этот код проверяет бит четности, не является самоочевидным. Это означает, что код менее читаем, чем он должен быть в идеале.

Листинг 5.27. Не требующая пояснений анонимная функция

```
List<UInt16> getValidIds(List<UInt16> ids) {  
    return ids  
        .filter(id -> id != 0)  
        .filter(id -> countSetBits(id & 0x7FFF) % 2 == ❶  
            ((id & 0x8000) >> 15));  
}
```

❶ Анонимная функция для проверки бита четности.

Это тот же самый краткий, но нечитаемый код, который мы видели ранее в этой главе. Подобная логика нуждается в объяснении, потому что большинство инженеров понятия не имеют, что она делает. И поскольку анонимные функции не предлагают никаких объяснений, кроме кода внутри них, это, вероятно, не очень хорошее применение.

5.8.3. Решение: вместо этого используйте именованные функции

Любой, кто читает функцию `getValidIds()` из предыдущего примера, вероятно, заинтересован только в знании подробностей высокого уровня о том, как получить

действительные идентификаторы. Для этого им нужно только знать о двух концептуальных причинах того, что идентификатор действителен.

- ◆ Это ненулевое значение.
- ◆ Бит четности указан правильно.

Их не следует заставлять бороться с концепциями более низкого уровня, такими, как побитовые операции, чтобы понять причину высокого уровня, по которой идентификатор является или не является допустимым. Было бы лучше использовать именованную функцию, чтобы абстрагироваться от деталей реализации проверки бита четности.

В листинге 5.28 показано, как это может выглядеть. Функция `getValidIds()` теперь невероятно проста для чтения; любой, кто ее читает, сразу понимает, что она выполняет две вещи: отфильтровывает ненулевые идентификаторы и отфильтровывает идентификаторы с неправильным битом четности. Если они хотят понять детали битов четности, они могут взглянуть на вспомогательную функцию, но они не вынуждены бороться с этими деталями, только для того чтобы понять функцию `getValidIds()`. Еще одно преимущество использования именованной функции заключается в том, что логику проверки бита четности теперь можно легко использовать повторно.

Листинг 5.28. Использование именованной функции

```
List<UInt16> getValidIds(List<UInt16> ids) {  
    return ids  
        .filter(id -> id != 0)  
        .filter(isParityBitCorrect); ❶  
}  
  
private Boolean isParityBitCorrect(UInt16 id) { ❷  
    ...  
}
```

❶ Именованная функция, используемая в качестве аргумента.

❷ Именованная функция для проверки бита четности.

Как мы видели в *главе 3*, изучение названий вещей — один из основных способов, с помощью которых инженеры понимают код. Недостатком именования вещей является то, что оно часто создает дополнительную многословность и шаблонность. Анонимные функции отлично подходят для уменьшения многословия и шаблонности, но с тем недостатком, что у функции больше нет имени. Для небольших, не требующих пояснений задач обычно это нормально, но для чего-либо большего или более сложного преимущества присвоения функции имени обычно перевешивают недостатки дополнительной многословности.

5.8.4. Большие анонимные функции могут быть проблематичными

Исходя из личного опыта, я нахожу, что инженеры иногда объединяют *функциональный* стиль программирования с использованием встроенных анонимных функций. Принятие функционального стиля программирования имеет много преимуществ, которые часто могут сделать код более читаемым и более надежным. Как показывают предыдущие примеры в этом разделе, мы можем довольно легко писать код в функциональном стиле с использованием именованных функций; принятие функционального стиля не означает, что мы должны использовать встроенные анонимные функции.

В главе 2 обсуждалась важность сохранения функций небольшими и лаконичными, чтобы инженерам было легко их читать, понимать и повторно использовать. При написании кода в функциональном стиле некоторые инженеры забывают об этом и создают огромные анонимные функции, которые содержат слишком много логики, а иногда даже другие анонимные функции, вложенные друг в друга. Если анонимная функция начинает приближаться к чему-либо длиной более двух или трех строк, то вполне вероятно, что код был бы более читабельным, если бы анонимная функция была разбита на части и помещена в одну или несколько именованных функций.

Чтобы продемонстрировать это, в листинге 5.29 показан код для отображения списка элементов обратной связи в пользовательском интерфейсе. Функция `buildFeedbackListItems()` содержит в себе очень большую встроенную анонимную функцию. Эта анонимная функция, в свою очередь, содержит в себе другую анонимную функцию. Количество логики, плотно упакованной вместе, а также количество вложений и отступов, затрудняют чтение этого кода. В частности, довольно сложно понять, какая информация на самом деле отображается в пользовательском интерфейсе, поскольку эта информация распространяется повсюду. Как только мы прочитаем весь код, мы увидим, что пользовательский интерфейс отображает заголовок, комментарий обратной связи и некоторые категории, но разобраться в этом непросто.

Листинг 5.29. Большая анонимная функция

```
void displayFeedback(List<Feedback> allFeedback) {
    ui.getFeedbackWidget().setItems(
        buildFeedbackListItems(allFeedback));
}

private List<ListItem> buildFeedbackListItems(
    List<Feedback> allFeedback) {

    return allFeedback.map(feedback ->
        new ListItem(
            title: new TextBox(
```

```

        text: feedback.getTitle(), ❶
        options: new TextOptions(weight: TextWeight.BOLD),
    ),
    body: new Column(
        children: [
            new TextBox(
                text: feedback.getComment(), ❷
                border: new Border(style: BorderStyle.DASHED),
            ),
            new Row(
                children: feedback.getCategories().map(category -> ❸
                    new TextBox(
                        text: category.getLabel(), ❹
                        options: new TextOptions(style: TextStyle.ITALIC),
                    ),
                ),
            ),
        ],
    ),
);
}

```

❶ Отображается заголовок.

❷ Отображается комментарий.

❸ Вторая анонимная функция, вложенная в основную.

❹ Отображаются некоторые категории.

Справедливости ради многие проблемы с кодом в листинге 5.29 не полностью связаны с тем фактом, что используются анонимные функции. Даже если бы весь этот код был перенесен в одну массивную именованную функцию, это все равно было бы беспорядком. Реальная проблема заключается в том, что функция делает слишком много, что усугубляется использованием анонимных функций, но не полностью вызвано этим. Код был бы намного более читабельным, если бы он был разбит на более мелкие именованные функции.

5.8.5. Решение: разбейте большие анонимные функции на именованные функции

В листинге 5.30 показано, как выглядит функция `buildProductListItems()`, которую мы только что видели, если логика разбита на ряд хорошо названных вспомогательных функций. Код более подробный, но значительно более читаемый. Важно отметить, что теперь инженер может просмотреть функцию `buildFeedbackItem()` и сразу увидеть, какая информация отображается в пользовательском интерфейсе

для каждой обратной связи: заголовок, комментарий и категории, к которым относится отзыв.

Листинг 5.30. Меньшие именованные функции

```
private List<ListItem> buildFeedbackListItems(
    List<Feedback> allFeedback) {
    return allFeedback.map(buildFeedbackItem); ❶
}

private ListItem buildFeedbackItem(Feedback feedback) {
    return new ListItem(
        title: buildTitle(feedback.getTitle()), ❷
        body: new Column(
            children: [
                buildCommentText(feedback.getComment()), ❸
                buildCategories(feedback.getCategories()), ❹
            ],
        ),
    );
}

private TextBox buildTitle(String title) {
    return new TextBox(
        text: title,
        options: new TextOptions(weight: TextWeight.BOLD),
    );
}

private Row buildCategories(List<Category> categories) {
    return new Row(
        children: categories.map(buildCategory), ❺
    );
}

private TextBox buildCategory(Category category) {
    return new TextBox(
        text: category.getLabel(),
        options: new TextOptions(style: TextStyle.ITALIC),
    );
}
```

❶ List.map() вызывается с помощью именованной функции.

❷ Отобразится заголовок.

❸ Отобразится комментарий.

❹ Отображаются некоторые категории.

❺ List.map() вызывается с помощью именованной функции.

Разделение больших функций, которые делают слишком много, — отличный способ улучшить читаемость кода (а также его повторное использование и модульность). Важно не забывать об этом при написании кода в функциональном стиле: если анонимная функция начинает становиться большой и громоздкой, то, вероятно, пришло время перенести логику на некоторые именованные функции.

5.9. Используйте блестящие новые языковые функции надлежащим образом

Все любят блестящие новые вещи, и инженеры ничем не отличаются. Многие языки программирования все еще активно разрабатываются, и время от времени разработчики языков добавляют приятную, новую, блестящую функцию. Когда это происходит, инженеры часто стремятся использовать эту новую функцию.

Разработчики языков программирования очень тщательно продумывают, прежде чем добавлять новую функцию, поэтому, вероятно, существует множество сценариев, в которых новая функция делает код значительно более читаемым или надежным. Хорошо, что инженеры приходят в восторг от этих вещей, потому что это увеличивает вероятность того, что новая функция будет использована для улучшения кода. Но, если вы обнаружите, что вам не терпится использовать новую или блестящую языковую функцию, убедитесь, что вы честны с собой в том, действительно ли это лучший инструмент для работы.

5.9.1. Новые функции могут улучшить код

Когда Java 8 представила потоки, многие инженеры пришли в восторг, так как это позволило написать гораздо более лаконичный код в функциональном стиле. В качестве примера того, как потоки могут улучшить код, в листинге 5.31 показан некоторый традиционный код Java, который берет список строк и отфильтровывает все пустые. Код довольно подробный (для концептуально довольно простой задачи) и требует использования цикла `for` и создания экземпляра нового списка.

Листинг 5.31. Традиционный код Java для фильтрации списка

```
List<String> getNonEmptyStrings(List<String> strings) {
    List<String> nonEmptyStrings = new ArrayList<>();
    for (String str : strings) {
        if (!str.isEmpty()) {
            nonEmptyStrings.add(str);
        }
    }
    return nonEmptyStrings;
}
```

Использование потока может сделать этот код намного более кратким и читабельным. В следующем листинге показано, как одна и та же функциональность может быть реализована с использованием потока и фильтра.

Листинг 5.32. Фильтрация списка с помощью потока

```
List<String> getNonEmptyStrings(List<String> strings) {  
    return strings  
        .stream()  
        .filter(str -> !str.isEmpty())  
        .collect(toList());  
}
```

Это кажется хорошим использованием языковой функции, так как код стал более читабельным и лаконичным. Использование языковой функции (вместо ручной прокрутки чего-либо) также увеличивает вероятность того, что код будет оптимально эффективным и без ошибок. Использование языковой функции, когда она улучшает код, как правило, является хорошей идеей (смотрите, однако, следующие два подраздела).

5.9.2. Неясные функции могут сбивать с толку

Даже если языковая функция дает явное преимущество, все равно стоит подумать о том, насколько хорошо эта функция известна другим инженерам. Как правило, это требует обдумывания нашего конкретного сценария и того, кому в итоге придется поддерживать код.

Если мы работаем в команде, которая поддерживает только небольшое количество кода Java, и никто из других инженеров не знаком с потоками Java, то, возможно, лучше избегать их использования. В этом случае улучшение кода, которое мы получаем от их использования, может быть относительно незначительным по сравнению с путаницей, которую они могут вызвать.

В целом использование языковой функции, когда она улучшает код, является хорошей идеей. Но если улучшение невелико или есть вероятность, что другие могут быть не знакомы с этой функцией, все равно лучше избегать ее использования.

5.9.3. Используйте лучший инструмент для работы

Потоки Java невероятно универсальны, и с их помощью можно решить множество проблем. Однако это не означает, что они всегда являются лучшим способом решения проблемы. Если бы у нас была карта и нам нужно было найти в ней значение, то наиболее разумным кодом для этого, вероятно, был бы

```
String value = map.get(key);
```

Но мы также могли бы решить эту проблему, получив поток записей карты и отфильтровав их на основе ключа. В листинге 5.33 показано, как это может вы-

глядеть. Очевидно, что это не особенно хороший способ получить значение с карты. Он не только намного менее удобочитаем, чем вызов `map.get()`, но и намного менее эффективен (потому что потенциально повторяет каждую запись на карте).

Листинг 5.33. Получение значения карты с помощью потока

```
Optional<String> value = map
    .entrySet()
    .stream()
    .filter(entry -> entry.getKey().equals(key))
    .map(Entry::getValue)
    .findFirst();
```

Листинг 5.33 может показаться каким-то чрезмерным примером, призванным доказать свою точку зрения, но я уже видел код, который более или менее идентичен этому в реальной кодовой базе.

Новые языковые функции часто добавляются по определенной причине и могут принести большие преимущества, но, как и в случае с любым написанным вами кодом, убедитесь, что вы используете функцию, потому что это подходящий инструмент для работы, а не только потому, что он блестящий или новый.

Итоги

- ◆ Если код не читается и его нелегко понять, это может привести к следующим проблемам.
 - Другие инженеры тратят время, пытаясь расшифровать его.
 - Недопонимания приводят к появлению ошибок.
 - Код нарушается, когда другим инженерам нужно его изменить.
- ◆ Повышение читабельности кода иногда может привести к тому, что он станет более подробным и займет больше строк. Это часто является стоящим компромиссом.
- ◆ Чтобы сделать код читабельным, часто требуется проявить чуткость и представить, как другие могут в чем-то запутаться.
- ◆ Сценарии реальной жизни разнообразны и обычно представляют свой собственный набор проблем. Написание читаемого кода почти всегда требует применения здравого смысла и использования вашего суждения.

6 Избегайте сюрпризов

ЭТА ГЛАВА ОХВАТЫВАЕТ

- ◇ Как код может приводить к сюрпризам
- ◇ Как сюрпризы могут приводить к ошибкам в программном обеспечении
- ◇ Способы обеспечения того, чтобы код не вызывал сюрпризов

Мы видели в *главе 2* и *3*, как код часто строится слоями, причем код на более высоких уровнях зависит от кода на более низких. Когда мы пишем код, это обычно всего лишь одна часть гораздо большей кодовой базы. Мы строим поверх других частей кода, полагаясь на них, а другие программисты строят поверх нашего кода, полагаясь на него. Чтобы это работало, программисты должны понимать, что делает код и как его следует использовать.

В *главе 3* говорилось о соглашениях по написанию кода как о способе поразмышлять о том, как другие программисты используют фрагмент кода. В кодовом соглашении такие вещи, как имена, типы параметров и типы возвращаемых данных безошибочно очевидны, в то время как комментарии и документация больше похожи на мелкий шрифт и часто упускаются из виду.

В конечном счете программист так или иначе построит мысленную модель того, как использовать фрагмент кода. Это будет основано на том, что он заметил в кодовом соглашении, любых имеющихся у него фоновых знаниях и общих парадигмах, которые, по его мнению, могут быть применимы. Если эта мысленная модель не соответствует реальности того, что на самом деле делает код, то, скорее всего, произойдет неприятный сюрприз. В лучшем случае это может просто привести к потере времени, а в худшем — к катастрофической ошибке.

Избегать неожиданностей часто означает быть предельно ясным. Если функция иногда ничего не возвращает или существует особый сценарий, требующий обработки, мы должны убедиться, что другие программисты знают об этом. Если мы этого не сделаем, то есть риск, что их мысленная модель того, что, по их мнению, делает код, не будет соответствовать действительности. В этой главе рассматриваются некоторые распространенные способы, с помощью которых код может приводить к сюрпризам, и некоторые методы, позволяющие избежать этого.

6.1. Избегайте возврата магических чисел

Магическое число — это значение, которое соответствует обычному типу возвращаемого значения функции, но которое имеет особый смысл. Очень распространенным примером магического числа является возврат минус единицы из функции, чтобы указать, что значение отсутствует (или произошла ошибка).

Поскольку магическое число вписывается в обычный тип возвращаемого значения функции, вызывающие пользователи, которые не знают об этом и проявляют активную бдительность, могут легко принять его за обычное возвращаемое значение. В этом разделе объясняется, как это может приводить к сюрпризам и как можно избежать магических чисел.

6.1.1. Магические числа могут привести к ошибкам

Возврат минус единицы из функции, чтобы указать, что значение отсутствует, — это практика, с которой вы, несомненно, будете время от времени сталкиваться. Так иногда делают фрагменты устаревшего кода и даже некоторые встроенные языковые функции (например, вызов `indexOf()` для массива в JavaScript).

В прошлом были некоторые более или менее разумные причины для возврата магического числа (например, минус один), потому что более явные методы сигнализации об ошибках или возврата значения `null` или необязательного значения не всегда были доступны или практичны в использовании. Если мы работаем над устаревшим кодом или у нас есть некий код, который нам необходимо тщательно оптимизировать, то некоторые из этих причин все еще могут быть актуальны. Но в целом возвращение магических чисел сопряжено с риском вызвать сюрприз, поэтому лучше просто избегать их использования.

Чтобы продемонстрировать, как магическое число может вызвать ошибку, рассмотрим код в листинге 6.1. Он содержит класс для хранения информации о пользователе. Одной из составляющих информации, хранящейся в классе, является возраст пользователя, и к ней можно получить доступ, вызвав функцию `getAge()`. Функция `getAge()` возвращает целое число, не допускающее значения `null`, поэтому программист, рассматривающий сигнатуру этой функции, скорее всего, предположит, что возраст всегда будет доступен.

Листинг 6.1. Класс пользователя и функция `getAge()`

```
class User {  
    ...  
    Int getAge() { ... } ❶  
}
```

❶ Возвращает возраст пользователя. Никогда не возвращает значение `null`.

Теперь представьте, что программисту нужно получить статистику обо всех пользователях сервиса. Один из статистических показателей, который необходимо рассчитать, — это средний возраст пользователей. Для этого пишется код в листинге 6.2. Код работает, суммируя все возрасты пользователей, а затем деля их на количество пользователей. Код предполагает, что `user.getAge()` всегда возвращает фактический возраст пользователя. Ничто в коде не выглядит явно неправильным, и автор, а с ним и рецензент этого кода, скорее всего, будут удовлетворены его работой.

Листинг 6.2. Расчет среднего возраста пользователей

```
Double? getMeanAge(List<User> users) {  
    if (users.isEmpty()) {  
        return null;  
    }  
    Double sumOfAges = 0.0;  
    for (User user in users) {  
        sumOfAges += user.getAge().toDouble(); ❶  
    }  
    return sumOfAges / users.size().toDouble();  
}
```

❶ Суммирует все возвращаемые значения `user.getAge()`.

На самом деле этот код не работает и часто возвращает неверное значение среднего возраста, потому что не все пользователи указали свой возраст. В этом случае функция `User.getAge()` возвращает минус один. Это упомянуто в мелком шрифте кода, но, к сожалению, автор функции `getMeanAge()` не осознал этого (как это часто бывает с мелким шрифтом, скрытым в комментарии). Если мы более подробно рассмотрим класс `User`, то увидим код в листинге 6.3. Тот факт, что `User.getAge()` возвращает минус один, не сообщая об этом вызывающим пользователям предельно ясно, преподнес неприятный сюрприз автору функции `getMeanAge()`. Функция `getMeanAge()` вернет правдоподобное, но неверное значение, поскольку при расчете среднего значения, скорее всего, будет включено некоторое количество значений минус один.

Листинг 6.3. Более подробный обзор класса `User`

```
class User {  
    private final Int? age; ❶  
    ...  
    // Возвращает значение -1, если возраст не указан. ❷  
    Int getAge() {  
        if (age == null) {  
            return -1; ❸  
        }  
        return age;  
    }  
}
```

- ❶ Возраст, возможно, не был указан.
- ❷ Мелким шрифтом (в комментарии) указано, что `getAge()` может возвращать минус один.
- ❸ Возвращает минус один, если возраст не указан.

Это может показаться чем-то вроде досадной, но не очень серьезной ошибки, но, не зная точно, где и как вызывается этот код, мы не можем быть полностью уверены. Представьте, что команда, собирающая статистику для годового отчета компании перед акционерами, повторно использовала функцию `getMeanAge()`. Указанное значение среднего возраста пользователей может оказать существенное влияние на цену акций компании. Если сообщенное значение неверно, то это может привести к серьезным юридическим последствиям.

Еще одна вещь, которую следует отметить, это то, что модульное тестирование может не выявить эту проблему. Автор функции `getMeanAge()` твердо (но неверно) убежден, что возраст пользователя всегда доступен. Существует большая вероятность, что ему не придет в голову написать модульный тест, в котором отсутствует возраст пользователя, потому что он даже не знает, что это то, что поддерживает класс пользователя. Тестирование — это здорово, но, если мы пишем код, который может привести к сюрпризу, мы полагаемся на чье-то усердие, чтобы не попасть в расставленную нами ловушку. В какой-то момент этого не произойдет, и кто-то в эту ловушку попадет.

6.1.2. Решение: вернуть значение `null`, необязательное значение или ошибку

В главе 3 говорилось о соглашениях по написанию кода и о том, как они содержат вещи, которые безошибочно очевидны, в дополнение к вещам, которые лучше всего описать как мелкий шрифт. Проблема с возвращением магического числа из функции заключается в том, что для этого вызывающим пользователям требуется знать то, что написано мелким шрифтом в соглашении функции. Некоторые программисты не будут читать этот мелкий шрифт, или же прочитают его, но потом забудут. Когда это происходит, это может привести к неприятному сюрпризу.

Если значение может отсутствовать, гораздо лучше убедиться, что это часть безошибочно очевидной части кодового соглашения. Один из самых простых способов сделать это — просто вернуть тип, допускающий значение `null`, если у нас есть `null`-безопасность, или вернуть необязательное значение, если у нас ее нет. Это гарантирует, что пользователи будут знать, что значение может отсутствовать, и они смогут справиться с этим соответствующим образом.

В листинге 6.4 показан класс `User` с функцией `getAge()`, измененной для возврата значения `null`, если значение не было указано. `Null`-безопасность гарантирует, что вызывающие пользователи знают о том, что `getAge()` может возвращать значение `null`. Если язык, который мы используем, не имеет `null`-безопасности, то возврат `Optional<Int>` приведет к тому же результату.

Листинг 6.4. Функция `getAge()` изменена, чтобы возвращать значение `null`

```
class User {  
    private final Int? age; ❶  
    ...  
    Int? getAge() { ❷  
        return age; ❸  
    }  
}
```

❶ Возраст, возможно, не был указан (? означает, что для возраста допускается значение `null`).

❷ Тип возвращаемого значения допускает значение `null`.

❸ Возвращает значение `null`, если возраст не указан.

Ошибочная функция `getMeanAge()` (повторенная в листинге 6.5) теперь вызывает ошибку компилятора, которая заставляет программиста, написавшего ее, понять, что в его коде есть потенциальная ошибка. Чтобы заставить код с `getMeanAge()` компилироваться, программисту пришлось бы обработать случай, когда `User.getAge()` возвращает значение `null`.

Листинг 6.5. Ошибочный код не компилируется

```
Double? getMeanAge(List<User> users) {  
    if (users.isEmpty()) {  
        return null;  
    }  
    Double sumOfAges = 0.0;  
    for (User user in users) {  
        sumOfAges += user.getAge().toDouble(); ❶  
    }  
    return sumOfAges / users.size().toDouble();  
}
```

❶ Вызывает ошибку компилятора, потому что функция `getAge()` может возвращать значение `null`

С организационной точки зрения (рассматривая код как продукт) возврат допускающего значение `null` заставляет программиста осознать, что вычисление среднего возраста пользователей не так просто, как он изначально думал. Он может сообщить об этом своему менеджеру или команде разработчиков, так как требования, возможно, придется уточнить на основании этой информации.

Недостатком возврата значения `null` (или пустого необязательного значения) является то, что оно не передает никакой явной информации о том, почему значение отсутствует: является ли возраст пользователя нулевым, потому что он не предоставил значение или потому, что в нашей системе произошла какая-то ошибка? Ес-

ли различать эти сценарии желательно, то нам следует рассмотреть возможность использования одного из методов сигнализации об ошибках, описанных в *главе 4*.

РАЗВЕ ВОЗВРАЩАЕМЫЙ ТИП, ДОПУСКАЮЩИЙ ЗНАЧЕНИЕ `NULL`, НЕ НАКЛАДЫВАЕТ БРЕМЯ НА ВЫЗЫВАЮЩЕГО ПОЛЬЗОВАТЕЛЯ?

Короткий ответ на это часто звучит так: «Да, это так». Если функция может возвращать значение `null`, то вызывающим пользователям часто приходится писать небольшое количество дополнительного кода для обработки сценария, в котором значение равно `null`. Некоторые программисты ссылаются на это как на причину против возврата значений `null` или необязательных типов, но стоит подумать, какой может быть альтернатива. Если значение может отсутствовать, а функция не делает это достаточно очевидным для вызывающих пользователей, то существует очень реальный риск того, что мы получим ошибочный код (как мы только что видели при расчете среднего возраста пользователей). В среднесрочной и долгосрочной перспективе усилия и затраты, связанные с устранением и исправлением подобной ошибки, могут быть на порядки выше, чем стоимость нескольких дополнительных строк кода для правильной обработки возвращаемого значения `null`. В *разд. 6.2* обсуждается шаблон объекта `null`, который иногда используется в качестве альтернативы возвращению значения `null`. Но, как мы увидим, это может быть проблематично при неправильном использовании. Еще одной причиной, иногда приводимой против возврата `null`, является риск возникновения исключений `NullPointerException`, `NullReferenceExceptions` или аналогичных. Но, как обсуждалось в *главе 2 (разд. 2.1)* и в *приложении В*, использование `null`-безопасности или необязательных типов, как правило, устраняет этот риск.

6.1.3. Иногда магические числа могут произойти случайно

Возврат магического числа не всегда происходит, потому что программист намеренно хочет этого. Это может произойти, когда программист не до конца продумал все входные данные, которые может получить его код, и какой эффект они могут оказать. Чтобы продемонстрировать это, в листинге 6.6 содержится код для поиска минимального значения из списка целых чисел. Способ реализации функции означает, что она возвращает магическое число (`Int.MAX_VALUE`), если список входных данных пуст.

Листинг 6.6. Поиск минимального значения

```
Int minValue(List<Int> values) {  
    Int minValue = Int.MAX_VALUE;  
    for (Int value in values) {  
        minValue = Math.min(value, minValue);  
    }  
    return minValue; ❶  
}
```

❶ Если список значений пуст, то возвращается значение `Int.MAX_VALUE`.

Очевидно, нам придется спросить программиста, написавшего код, но возврат `Int.MAX_VALUE` может быть неслучайным. Несколько аргументов, которые автор этого

кода мог бы привести в обоснование того, почему это разумно, заключаются в следующем.

- ◆ Для вызывающих пользователей должно быть очевидно, что получение минимального значения пустого списка не имеет смысла, поэтому значение, возвращаемое в этом сценарии, на самом деле не играет роли.
- ◆ `Int.MAX_VALUE` — это разумное возвращаемое значение, так как ни одно целое число не может быть больше его. Это означает, что, если его сравнить с каким-либо пороговым значением, код, скорее всего, будет по умолчанию вести себя разумно.

Проблема с этими аргументами заключается в том, что они делают предположения о том, как будет вызываться функция и как будет использоваться результат. Эти предположения легко могут оказаться ошибочными, и если это так, то это преподнесет сюрприз.

Возврат значения `Int.MAX_VALUE` может не привести к какому-либо разумному поведению по умолчанию. Примером этого может быть использование его как части алгоритма `maximin` (максимального минимума). Представьте, что мы являемся частью команды программистов, разрабатывающих игру, и мы хотим определить, какой уровень в игре самый простой. Мы решаем, что для каждого уровня мы найдем минимальное количество очков, которое набрал любой игрок. Какой бы уровень ни набрал наибольшее минимальное количество очков, он будет считаться самым легким.

В листинге 6.7 показан код, который мы могли бы написать. В нем используется функция `MinValue()`, которую мы только что видели. Мы запускаем код, и он выводит, что уровень 14 — самый простой уровень. На самом деле 14-й уровень настолько сложен, что его еще никто не проходил. Это означает, что за него никогда не было записано ни одного очка. При обработке уровня 14 функция `MinValue()` вызывается с пустым списком и, таким образом, возвращает значение `Int.MAX_VALUE`. Это делает 14-й уровень абсолютным победителем с точки зрения получения максимального минимального количества очков. Но, как мы только что отметили, реальность такова, что очки не записываются, потому что уровень настолько сложный, что никто никогда не проходил его до конца.

Листинг 6.7. Алгоритм `maximin`

```
class GameLevel {
    ...
    List<Int> getAllScores() { ... } ❶
}

GameLevel? getEasiestLevel(List<GameLevel> levels) {
    GameLevel? easiestLevel = null;
    Int? highestMinScore = null;
    for (GameLevel level in levels) {
        Int minScore = minValue(level.getAllScores()); ❷
        if (highestMinScore == null || minScore > highestMinScore) {
```

```

    easiestLevel = level;
    highestMinScore = minScore;
}
}
return easiestLevel; ❸
}

```

- ❶ Возвращает пустой список, если не набрано ни одного очка.
- ❷ Сменяется на значение `Int.MAX_VALUE`, если не набрано ни одного очка.
- ❸ Если на уровне не набрано ни одного очка, то возвращается это значение.

Существует несколько других ситуаций, в которых значение `Int.MAX_VALUE` может быть проблематичным.

- ◆ Значение `Int.MAX_VALUE` часто зависит от используемого языка программирования. Если функция `MinValue()` находится внутри сервера Java, а ответ отправляется в клиентское приложение, написанное на JavaScript, то важность этого значения не будет столь очевидна: `Integer.MAX_VALUE` (в Java) — это совсем другое число, чем `Number.MAX_SAFE_INTEGER` (в JavaScript).
- ◆ Если выходные данные функции были сохранены в базе данных, это может вызвать много путаницы и проблем у любого, кто выполняет запросы, или у других систем, которые читают базу данных.

Было бы лучше просто вернуть значение `null`, пустое необязательное значение или сообщить о какой-либо ошибке функции `MinValue()`, чтобы вызывающие абоненты знали, что значение может быть невычисляемым для некоторых входных данных. Если мы вернем значение `null`, это создаст дополнительную нагрузку на вызывающего пользователя, потому что он должен написать логику для обработки этого значения. Но это снимает еще одну ответственность: необходимость перед вызовом `MinValue()` проверять, пуст ли список, а также устраняет вероятность ошибки кода, если эту проверку не выполнить. В следующем листинге показано, как может выглядеть функция `MinValue()`, если она будет возвращать значение `null` для пустого списка.

Листинг 6.8. Возврат значение `null` для пустого списка

```

Int? minValve(List<Int> values) {
    if (values.isEmpty()) {
        return null; ❶
    }
    Int minValve = Int.MAX_VALUE;
    for (Int value in values) {
        minValve = Math.min(value, minValve);
    }
    return minValve;
}

```

- ❶ Если список значений пуст, то возвращается значение `null`.

Возврат магического числа иногда является сознательным решением, принятым программистом, но также это может произойти случайно. Независимо от причины магические числа могут запросто привести к сюрпризам, поэтому полезно быть бдительными в отношении сценариев, в которых это может произойти. Возврат значения `null`, необязательного значения или использование метода сигнализации об ошибках являются простыми и эффективными альтернативами.

ИСПОЛЬЗУЙТЕ СООТВЕТСТВУЮЩУЮ ВОЗМОЖНОСТЬ ЯЗЫКА

Пример `MinValue()` используется для демонстрации общей идеи о возврате магических значений. Если нам нужно найти минимальное значение в списке целых чисел, то, скорее всего, в языке уже существует подходящая возможность или утилита, которая это сделает. Если есть, то, вероятно, лучше использовать эту возможность, а не вручную добавлять нашу собственную функцию.

6.2. Используйте шаблон нулевого объекта соответствующим образом

Шаблон нулевого (`null`) объекта является альтернативой возвращению значения `null` (или пустого необязательного значения), когда какое-то значение не может быть получено. Идея заключается в том, что вместо возврата значения `null` возвращается допустимое значение, которое приведет к тому, что любая последующая логика будет вести себя безобидным образом. Простейшими формами этого являются возврат пустой строки или пустого списка, в то время как более сложные формы предполагают реализацию целого класса, в котором каждая функция-член либо ничего не делает, либо возвращает значение по умолчанию.

Шаблон нулевого объекта был кратко упомянут в *главе 4*, когда речь шла об ошибках. В *главе 4* показано, почему часто бывает плохой идеей использовать шаблон нулевого объекта, чтобы скрыть тот факт, что произошла ошибка. За пределами обработки ошибок шаблон нулевого объекта может быть весьма полезен, но при неправильном использовании он может приводить к неприятным сюрпризам и неочевидным ошибкам, которые трудно найти.

Примеры в этом разделе сравнивают шаблон нулевого объекта с возвращением значения `null`. Если вы используете язык, который не обеспечивает `null`-безопасность, то обычно вместо безопасного `null` можно использовать необязательный тип возвращаемого значения.

6.2.1. Возврат пустой коллекции может улучшить код

Когда функция возвращает коллекцию (например, список, набор или массив), иногда бывает так, что значения в коллекции не могут быть получены. Это может быть связано с тем, что они не были установлены или могут быть неприменимы в данном сценарии. Один из подходов — вернуть значение `null`, когда такое происходит.

В листинге 6.9 показан код для проверки того, выделен ли элемент HTML. Код делает это, вызывая функцию `getClassNames()` и проверяя, находится ли «выделенный» класс в наборе классов элемента. Функция `getClassNames()` возвращает значение `null`, если у элемента нет атрибута «класс». Это означает, что функция `isElementHighlighted()` должна проверить, является ли набор имен классов нулевым, прежде чем использовать его.

Листинг 6.9. Возврат значения `null`

```
Set<String>? getClassNames(HtmlElement element) {
    String? attribute = element.getAttribute("class");
    if (attribute == null) {
        return null; ❶
    }
    return new Set(attribute.split(" "));
}

...

Boolean isElementHighlighted(HtmlElement element) {
    Set<String>? classNames = getClassNames(element);
    if (classNames == null) { ❷
        return false;
    }
    return classNames.contains("highlighted");
}
```

❶ Возвращается значение `null`, если в элементе нет атрибута класса.

❷ Необходимо проверить, является ли `classNames` нулевым, прежде чем использовать его.

Кто-то может возразить, что у функции `getClassNames()`, возвращающей тип, допускающий значение `null`, есть преимущество: она проводит различие между сценарием, в котором атрибут «класс» не был установлен (возвращено значение `null`) и был явно установлен как отсутствующий (возвращен пустой набор). Но это тонкое различие, которое в большинстве случаев, вероятно, более запутывает, чем помогает. Слой абстракции, предоставляемый функцией `getClassNames()`, также должен быть направлен на то, чтобы скрыть эти сведения о реализации атрибутов элементов.

Возврат типа, допускающего значение `null`, также заставляет каждого вызывающего `getClassNames()` проверять, является ли возвращаемое значение нулевым, прежде чем использовать его. Это добавляет больше беспорядка в код без особой пользы, потому что очень маловероятно, что кто-либо вызывающий будет заботиться о распознавании отличий между тем, что атрибут класса не был установлен, и тем, что он был установлен в пустую строку.

Это сценарий, в котором шаблон нулевого объекта может улучшить код. Функция `getClassNames()` может вместо этого возвращать пустой набор, если в элементе нет атрибута класса. Это означает, что вызывающим абонентам никогда не придется обрабатывать значение `null`. В листинге 6.10 показан код, измененный для исполь-

зования шаблона нулевого объекта путем возврата пустого набора. Функция `isElementHighlighted()` теперь значительно проще и лаконичнее.

Листинг 6.10. Возврат пустого набора

```
Set<String> getClassNames(HtmlElement element) {
    String? attribute = element.getAttribute("class");
    if (attribute == null) {
        return new Set(); ❶
    }
    return new Set(attribute.split(" "));
}

...

Boolean isElementHighlighted(HtmlElement element) {
    return getClassNames(element).contains("highlighted"); ❷
}
```

❶ Возвращает пустой набор, если в элементе нет атрибута класса.

❷ Нет необходимости проверять наличие значения `null`.

Это пример шаблона нулевого объекта, улучшающего качество кода. Это упрощает логику вызывающих и вряд ли приведет к неприятному сюрпризу. Но в более сложных сценариях риск неприятного сюрприза при использовании шаблона нулевого объекта может начать перевешивать преимущества. В следующих подразделах объясняется почему.

ИСКЛЮЧЕНИЯ НУЛЕВОГО УКАЗАТЕЛЯ

Более старомодный аргумент для использования шаблона нулевого объекта состоит в том, чтобы свести к минимуму вероятность возникновения исключений `NullPointerException`, `NullReferenceExceptions` и подобных. Если используется язык с небезопасными значениями `null`, то возврат значения `null` всегда сопряжен с риском, поскольку вызывающие пользователи могут не потрудиться проверить значение `null` перед использованием. Если же мы используем `null`-безопасность или необязательные значения (если у нас нет `null`-безопасности), этот аргумент будет неактуален. Но это будет иметь значение, если мы рассматриваем устаревший код, в котором используются небезопасные значения `null`.

6.2.2. Возврат пустой строки иногда может быть проблематичным

Предыдущий подраздел показал, как возврат пустой коллекции вместо значения `null` может улучшить качество кода. Некоторые программисты выступают за то, чтобы это также применялось к строкам, тогда вместо значения `null` должна быть возвращена пустая строка. Уместность этого зависит от того, как используется строка. В некоторых случаях строка представляет собой не что иное, как набор символов, и в этом случае возврат пустой строки вместо `null` вполне может быть

разумным. Когда строка имеет какое-то значение помимо вышесказанного, это уже не будет такой хорошей идеей. Чтобы продемонстрировать это, рассмотрим следующие сценарии.

Строка как набор символов

Когда строка представляет собой просто набор символов и не имеет внутреннего значения для кода, то использование шаблона нулевого объекта при отсутствии строки обычно работает. В случае со строкой это означает возврат пустой строки вместо возврата `null`, когда значение недоступно. Когда строка не имеет внутреннего значения, различие между тем, является ли она нулевой или пустой, вряд ли будет иметь значение для вызывающих.

Листинг 6.11 демонстрирует это с помощью функции доступа к комментариям в свободной форме, введенным пользователем при предоставлении обратной связи. Очень маловероятно, что существует какое-либо полезное различие между пользователем, который не вводил никаких комментариев, и тем, кто явно ввел пустую строку. Поэтому функция возвращает пустую строку, если комментарии не были предоставлены.

Листинг 6.11. Возвращает пустую строку

```
class UserFeedback {
    private String? additionalComments;
    ...
    String getAdditionalComments() {
        if (additionalComments == null) {
            return ""; ❶
        }
        return additionalComments;
    }
}
```

❶ Возвращает пустую строку, если комментарии не введены.

Строка в качестве идентификатора

Строка не всегда является просто набором символов. Часто она имеет определенное значение, которое обладает некоторой важностью для кода. Распространенным примером этого является использование строки в качестве идентификатора. В этом случае часто бывает важно знать, отсутствует ли строка, так как это может повлиять на то, какая логика должна выполняться. Поэтому важно убедиться, что все вызывающие абоненты функции явно осведомлены о том, что строка может отсутствовать.

Чтобы продемонстрировать это, в листинге 6.12 показан класс для представления платежа. Он содержит поле, допускающее значение `null`, называемое `cardTransactionId`. Если платеж был связан с транзакцией по карте, то в этом поле будет содержаться

идентификатор для этой транзакции. Если платеж не был связан с транзакцией по карте, то это поле будет пустым. Ясно, что строка `cardTransactionId` — это не просто набор символов: она имеет определенную важность, и ее значение `null` говорит о чем-то.

В этом примере кода функция `getCardTransactionId()` использует шаблон нулевого объекта, возвращая пустую строку, если значение `cardTransactionId` равно `null`. Это приводит к проблемам, потому что программисты могут увидеть, что поле не может быть обнулено, и предположить, что всегда есть ссылка на транзакцию по карте. Бизнес, использующий этот код, может в конечном итоге получить неточные учетные данные, если инженеры не смогут должным образом обработать сценарии, в которых платежи не связаны с транзакциями по картам.

Листинг 6.12. Возврат пустой строки для идентификатора

```
class Payment {  
    private final String? cardTransactionId; ❶  
    ...  
    String getCardTransactionId() { ❷  
        if (cardTransactionId == null) {  
            return ""; ❸  
        }  
        return cardTransactionId;  
    }  
}
```

❶ Идентификатор `cardTransactionId` может быть равен `null`.

❷ Сигнатура функции не указывает на то, что идентификатор может отсутствовать.

❸ Пустая строка возвращается, если значение `cardTransactionId` равно `null`.

Было бы намного лучше, если бы функция `getCardTransactionId()` вместо этого возвращала значение `null`, когда значение `cardTransactionId` равно `null`. Это дает вызывающим пользователям понять, что оплата может не включать транзакцию по карте, и позволяет избежать сюрпризов. Если бы это было сделано, то код выглядел бы следующим образом.

Листинг 6.13. Возврат значения `null` для идентификатора

```
class Payment {  
    private final String? cardTransactionId;  
    ...  
    String? getCardTransactionId() { ❶  
        return cardTransactionId;  
    }  
}
```

❶ Сигнатура функции дает понять, что идентификатор может отсутствовать.

6.2.3. Более сложные нулевые объекты могут привести к сюрпризам

Представьте, что вы хотите купить новый смартфон. Вы идете в магазин электроники и говорите продавцу, какую модель вы хотели бы купить. Там вам продают запечатанную коробку, которая выглядит точно так, как будто в ней должен быть новый блестящий телефон. Вы идете домой, разворачиваете целлофан, открываете коробку и обнаруживаете, что внутри ничего нет. Получился неприятный и раздражающий сюрприз, и, в зависимости от того зачем вам нужен новый телефон, это может иметь последствия и похуже: теперь вы можете пропустить важный рабочий звонок или сообщение от друга.

В этом случае в магазине закончилась модель телефона, которую вы хотели. Вместо того чтобы сказать вам это и позволить вам либо пойти в другой магазин, либо выбрать другую модель, продавец промолчал и продал вам пустую коробку. Если мы не будем осторожны с тем, как мы используем шаблон нулевого объекта, он может легко привести к аналогичному сценарию. По сути, мы продаем вызывающим пользователей нашей функции пустую коробку. Если есть хоть какой-то шанс, что они будут удивлены или раздражены, получив такую пустую коробку, то, вероятно, лучше избегать шаблона нулевого объекта.

Более сложная форма шаблона нулевого объекта может включать создание целого класса с некоторыми предположительно безобидными значениями. Листинг 6.14 содержит два класса: один для представления кофейной кружки и один для представления инвентарного списка кофейных кружек. Класс `CoffeeMugInventory` содержит функцию для получения случайной кофейной кружки из инвентаря. Если в инвентаре нет кофейных кружек, то, очевидно, невозможно получить случайную. Когда это происходит, функция `getRandomMug()` создает и возвращает кофейную кружку нулевого размера вместо возврата `null`. Это еще один пример шаблона нулевого объекта, но в этом сценарии он может очень легко создать сюрпризы для вызывающих пользователей. Любой, кто вызовет `getRandomMug()` и получит то, что выглядит как кружка, предположит, что у них есть реальная кружка из инвентаря, хотя на самом деле это может быть не так.

Листинг 6.14. Нулевой объект с сюрпризом

```
class CoffeeMug { ❶
    ...
    CoffeeMug(Double diameter, Double height) { ... }
    Double getDiameter() { ... }
    Double getHeight() { ... }
}

class CoffeeMugInventory {
    private final List<CoffeeMug> mugs;
    ...
    CoffeeMug getRandomMug() {
        if (mugs.isEmpty()) {
```

```

        return new CoffeeMug(diameter: 0.0, height: 0.0); ❷
    }

    return mugs[Math.randomInt(0, mugs.size())];
}
}

```

❶ Класс `CoffeeMug`.

❷ Создает и возвращает кофейную кружку нулевого размера, когда кружки недоступны.

Для некоторых пользователей получение кофейной кружки нулевого размера может удовлетворить их потребности, и это избавит их от необходимости проверять наличие значения `null`, но для других пользователей это может привести к серьезной ошибке, которая произойдет незаметно. Представьте, что консалтинговой фирме заплатили большую сумму денег за подготовку отчета о распределении размеров кофейных кружек, и они использовали этот код. Тогда могут быть предоставлены крайне неверные выводы из-за всех этих кофейных кружек нулевого размера, которые появились в наборе данных, никем не замеченные.

Автор кода в листинге 6.14, несомненно, имеет благие намерения; он пытается облегчить жизнь вызывающим функцию `getRandomMug()`, не заставляя их обрабатывать значение `null`. Но, к сожалению, это создает ситуацию, которая потенциально приводит к сюрпризам, потому что у вызывающего функцию создается ложное впечатление, что он всегда получит реальную `CoffeeMug`.

Вероятно, было бы лучше просто вернуть `null` из `getRandomMug()`, когда нет кружек, из которых можно выбрать случайную. Тогда будет безошибочно очевидным пунктом соглашения по написанию кода тот факт, что функция может не вернуть реальную кружку и не оставит места для сюрприза, когда это действительно произойдет. В следующем листинге показано, как выглядит функция `getRandomMug()`, если она возвращает значение `null`.

Листинг 6.15. Возврат значения `null`

```

CoffeeMug? getRandomMug(List<CoffeeMug> mugs) {
    if (mugs.isEmpty()) {
        return null; ❶
    }
    return mugs[Math.randomInt(0, mugs.size())];
}

```

❶ Возвращает значение `null`, если случайная кружка не может быть получена.

6.2.4. Реализация нулевого объекта может привести к сюрпризам

Некоторые программисты идут дальше в работе с шаблоном нулевого объекта и определяют выделенные реализации нулевых объектов интерфейсов или классов.

Одной из причин этого может быть то, что интерфейс или класс имеют функции, которые *что-то делают*, а не только те, которые *что-то возвращают*.

Листинг 6.16 содержит интерфейс для представления кофейной кружки вместе с двумя реализациями: `CoffeeMugImpl` и `NullCoffeeMug`. `NullCoffeeMug` — это реализация нулевого объекта для `CoffeeMug`. Он реализует все функции интерфейса `CoffeeMug`, но возвращает ноль при вызове `getDiameter()` или `getHeight()`. В этом примере `CoffeeMug` теперь также имеет функцию, которая *что-то делает*: `reportMugBroken()`. Это можно использовать для обновления записи о том, какие кружки разбиты. Реализация `NullCoffeeMug` просто ничего не делает, если вызывается эта функция.

Листинг 6.16. Реализация нулевого объекта

```
interface CoffeeMug { ❶
    Double getDiameter();
    Double getHeight();
    void reportMugBroken();
}

class CoffeeMugImpl implements CoffeeMug { ❷
    ...
    override Double getDiameter() { return diameter; }
    override Double getHeight() { return height; }
    override void reportMugBroken() { ... }
}

class NullCoffeeMug implements CoffeeMug { ❸
    override Double getDiameter() { return 0.0; } ❹
    override Double getHeight() { return 0.0; } ❺
    override void reportMugBroken() {
        // Ничего не делать
    }
}
```

❶ Интерфейс `CoffeeMug`.

❷ Нормальная реализация `CoffeeMug`.

❸ Реализация нулевого объекта для `CoffeeMug`.

❹ Функции, которые должны что-то возвращать, возвращают ноль.

❺ Функции, которые должны что-то делать, вместо этого ничего не делают.

В листинге 6.17 показано, как выглядит функция `getRandomMug()` (которую мы видели ранее), если она возвращает `NullCoffeeMug`, когда нет кружек. Это обеспечивает более или менее то же самое, что и в предыдущем примере, в котором была создана и возвращена кофейная кружка нулевого размера. И данный код страдает от того же набора проблем и все еще может легко привести к сюрпризам.

Листинг 6.17. Возврат `NullCoffeeMug`

```
CoffeeMug getRandomMug(List<CoffeeMug> mugs) {  
    if (mugs.isEmpty()) {  
        return new NullCoffeeMug(); ❶  
    }  
    return mugs[Math.randomInt(0, mugs.size())];  
}
```

❶ Возвращает `NullCoffeeMug`, если случайная кофейная кружка не может быть получена.

Одно небольшое улучшение при возврате `NullCoffeeMug` заключается в том, что вызывающие теперь могут посмотреть, имеют ли они дело с нулевым объектом, проверив, является ли возвращаемое значение примером `NullCoffeeMug`. Однако это не является большим улучшением, так как пользователям непонятно, могут ли они вообще захотеть это проверить. Даже если вызывающий пользователь осведомлен, требовать от него проверить, является ли значение примером `NullCoffeeMug`, является решением довольно неуклюжим и, вероятно, худшим, чем просто проверка на `null` (что является гораздо более распространенной и меньше приводящей к сюрпризам парадигмой).

Шаблон нулевого объекта может проявляться во многих формах. Стоит осознать, когда мы его используем или сталкиваемся с ним, и думать о том, действительно ли это уместно или может вызвать сюрпризы. Рост популярности `null`-безопасности и необязательных значений позволил намного проще и безопаснее сигнализировать об отсутствии значения. Учитывая это, многие из первоначальных аргументов в пользу использования шаблона нулевого объекта в наши дни менее убедительны.

6.3. Избегайте возникновения неожиданных побочных эффектов

Побочным эффектом является любое состояние, которое функция изменяет вне себя при вызове. Если функция имеет какие-либо эффекты, отличные от просто возвращаемого значения, то у нее есть побочные эффекты. Перечислим распространенные типы побочных эффектов.

- ◆ Отображение выводных данных пользователю.
- ◆ Сохранение чего-либо в файл или базу данных.
- ◆ Получение сетевого трафика путем вызова другой системы.
- ◆ Обновление или аннулирование кеша.

Побочные эффекты являются неизбежной частью написания программного обеспечения. Часть программного обеспечения без побочных эффектов, скорее всего, будет бессмысленной: в какой-то момент ей нужно что-то вывести пользователю, базе данных или другой системе. Это означает, что, по крайней мере, некоторые части

кода должны иметь побочные эффекты. Когда побочный эффект ожидаем и совпадает с запросом вызывающего пользователя, тогда все нормально, но когда побочный эффект неожиданный, это может привести к сюрпризам и ошибкам.

Один из лучших способов избежать неожиданного побочного эффекта — это в первую очередь не вызывать побочный эффект. В примерах, приведенных в этом разделе и *разд. 6.4*, это будет обсуждаться. Но создание неизменяемых классов также является отличным способом минимизировать потенциальные побочные эффекты, и это будет рассмотрено в *главе 7*. В тех случаях когда побочный эффект является частью желаемой функциональности или его невозможно избежать, важно убедиться, что пользователи знают об этом.

6.3.1. Побочные эффекты, которые очевидны и преднамеренны, — это нормально

Как только что упоминалось, побочные эффекты в коде порой необходимы. В листинге 6.18 показан класс для управления пользовательским дисплеем. Функция `displayErrorMessage()` вызывает побочный эффект: она обновляет то, что отображается пользователю. Но, учитывая, что класс называется `UserDisplay`, а функция называется `displayErrorMessage()`, совершенно очевидно, что это вызовет этот побочный эффект. Здесь нет места для сюрприза.

Листинг 6.18. Ожидаемый побочный эффект

```
class UserDisplay {  
    private final Canvas canvas;  
    ...  
  
    void displayErrorMessage(String message) {  
        canvas.drawText(message, Color.RED); ❶  
    }  
}
```

❶ Побочный эффект: обновляется видимый пользователю интерфейс.

Функция `displayErrorMessage()` является примером очевидного и преднамеренного побочного эффекта. Обновление видимого интерфейса с сообщением об ошибке — это именно то, чего хочет и ожидает вызывающий пользователь. С другой стороны, функции с побочными эффектами, которые пользователь не обязательно ожидает или желает, могут быть проблематичными. Это обсуждается в следующих подразделах.

6.3.2. Неожиданные побочные эффекты могут быть проблемой

Когда целью функции является получение или считывание значения, другие программисты обычно предполагают, что это не вызовет побочного эффекта. В лис-

тинге 6.19 показана функция, позволяющая получить цвет в определенном пикселе на пользовательском дисплее. Похоже, что это должно быть относительно просто и побочные эффекты здесь маловероятны. К сожалению, это не так; перед считыванием цвета пикселя функция `GetPixel()` вызывает событие перерисовки на видимом интерфейсе. Это побочный эффект, и для тех, кто не знаком с реализацией функции `GetPixel()`, это будет неожиданностью.

Листинг 6.19. Неожиданный побочный эффект

```
class UserDisplay {
    private final Canvas canvas;
    ...
    Color getPixel(Int x, Int y) {
        canvas.redraw(); ❶
        PixelData data = canvas.getPixel(x, y);
        return new Color(
            data.getRed(),
            data.getGreen(),
            data.getBlue());
    }
}
```

❶ Запуск события перерисовки является побочным эффектом.

Есть несколько вариантов, при которых неожиданный побочный эффект, подобный этому, может быть проблемой. В следующих нескольких подразделах будут рассмотрены некоторые из них.

Побочные эффекты могут быть затратными

Вызов `canvas.redraw()` потенциально является довольно затратной операцией и может также вызвать мерцание дисплея для пользователя. Программист, вызывающий `GetPixel()`, скорее всего, не ожидает, что это будет затратно или вызовет видимую пользователю проблему: ничто в названии функции `GetPixel()` не предполагает этого. Но если это затратно и вызывает мерцание, то могут произойти довольно неприятные вещи, которые большинство пользователей истолкуют как ужасную ошибку.

Представьте, что в приложение была добавлена функция, позволяющая пользователю сделать снимок экрана дисплея. В листинге 6.20 показано, как это может быть реализовано. Функция `captureScreenshot()` считывает пиксели один за другим, вызывая функцию `GetPixel()`. Это приводит к тому, что `canvas.redraw()` вызывается для каждого отдельного пикселя на снимке экрана. Предположим, что одно событие перерисовки занимает 10 миллисекунд, а пользовательский дисплей составляет 400×700 пикселей (или 280 000 пикселей в общей сложности). Создание снимка экрана приведет к тому, что приложение зависнет и будет мерцать в течение 47 минут. Почти каждый пользователь истолковал бы это как сбой приложения и, скорее всего, перезапустил бы его, потенциально потеряв несохраненную работу.

Листинг 6.20. Создание снимка экрана

```
class UserDisplay {
    private final Canvas canvas;
    ...
    Color getPixel(Int x, Int y) { ... } ❶
    ...
    Image captureScreenshot() {
        Image image = new Image(
            canvas.getWidth(), canvas.getHeight());
        for (Int x = 0; x < image.getWidth(); ++x) {
            for (Int y = 0; y < image.getHeight(); ++y) {
                image.setPixel(x, y, getPixel(x, y)); ❷
            }
        }
        return image;
    }
}
```

❶ Из-за побочного эффекта требуется примерно 10 мс для запуска.

❷ `GetPixel()` вызывается много раз.

Нарушение предположения, сделанного вызывающим

Даже если перерисовка пользовательского интерфейса не затратна, функция с названием `CaptureScreenshot()` не похожа на такую, которая вызовет побочный эффект, поэтому большинство программистов, вызывающих эту функцию, предположат, что все будет в порядке. Тот факт, что это предположение неверно, потенциально может привести к ошибке. В листинге 6.21 показана функция для получения отредактированного снимка экрана. При этом удаляются все области пользовательского интерфейса, содержащие личную информацию пользователя, а затем вызывается функция `CaptureScreenshot()`. Эта функция используется для получения анонимизированного снимка экрана всякий раз, когда пользователь дает обратную связь или отправляет отчет об ошибке. Удаление областей содержимого экрана очистит эти пиксели до следующего вызова `canvas.redraw()`.

Автор функции `captureRedactedScreenshot()` предположил, что функция `canvas.redraw()` не будет вызываться функцией `CaptureScreenshot()`. К сожалению, это предположение неверно, потому что `CaptureScreenshot()` вызывает `GetPixel()`, которая, в свою очередь, вызывает `canvas.redraw()`. Это означает, что функциональность редактирования полностью нарушена, и личная информация будет отправлена вместе с обратной связью. Это серьезное нарушение конфиденциальности пользователей и серьезная ошибка.

Листинг 6.21. Получение отредактированного снимка экрана

```
class UserDisplay {
    private final Canvas canvas;
    ...
```



```
Color getPixel(Int x, Int y) { ... } ❶

Image captureScreenshot() { ... } ❷

List<Box> getPrivacySensitiveAreas() { ... } ❸

Image captureRedactedScreenshot() {
    for (Box area in getPrivacySensitiveAreas()) {
        canvas.delete( ❹
            area.getX(), area.getY(),
            area.getWidth(), area.getHeight());
    }
    Image screenshot = captureScreenshot(); ❺
    canvas.redraw(); ❻
    return screenshot;
}
```

❶ Приводит к побочному эффекту, вызывая `canvas.redraw()`.

❷ Косвенно приводит к побочному эффекту, вызывая `GetPixel()`.

❸ Возвращает любые области пользовательского интерфейса, содержащие личную информацию пользователя.

❹ Удаляет все пиксели, содержащие личную информацию пользователя.

❺ Снимок экрана сделан.

❻ Преднамеренная очистка: единственное место, которое, по мнению автора, вызывает `canvas.redraw()`.

Ошибки в многопоточном коде

Если программе необходимо выполнить несколько задач относительно независимо друг от друга, обычным способом достижения этого является выполнение каждой задачи в своем собственном потоке. Затем компьютер может быстро переключаться между задачами, многократно прерывая и возобновляя потоки по очереди. Это называется многопоточностью. Поскольку разные потоки часто могут иметь доступ к одним и тем же данным, побочный эффект, вызванный одним потоком, иногда может вызвать проблемы для другого потока.

Представьте, что еще одна часть функциональности в приложении позволяет пользователю в реальном времени делиться своим экраном с другом. Это может быть реализовано с помощью другого потока, который периодически делает снимок экрана и отправляет его другу. Если несколько потоков одновременно вызывают функцию `CaptureScreenshot()`, снимки экрана могут быть повреждены, потому что один поток может перерисовывать видимый пользователю интерфейс, в то время как другой поток пытается его прочитать. Рис. 6.1 иллюстрирует это, по-

казывая, как могут взаимодействовать два случая вызова `GetPixel()` из двух отдельных потоков.

Вероятность возникновения проблемы с многопоточностью при отдельном вызове функции обычно довольно мала. Но когда функция вызывается тысячи (или даже миллионы) раз, совокупная вероятность того, что это произойдет, становится довольно высокой. Ошибки, связанные с проблемами многопоточности, также, как известно, трудно отлаживать и тестировать.

Программист, который видит функцию с именем `CaptureScreenshot()` или `GetPixel()`, не будет ожидать, что у любой из них будут побочные эффекты, которые могут привести к нарушению функциональности кода, выполняемого в другом потоке. Написание кода, который плохо ведет себя в многопоточной среде, может привести к особенно неприятным сюрпризам. Отладка и устранение ошибок могут отнять у программистов много времени. Гораздо лучше либо избегать побочных эффектов, либо делать их очевидными.

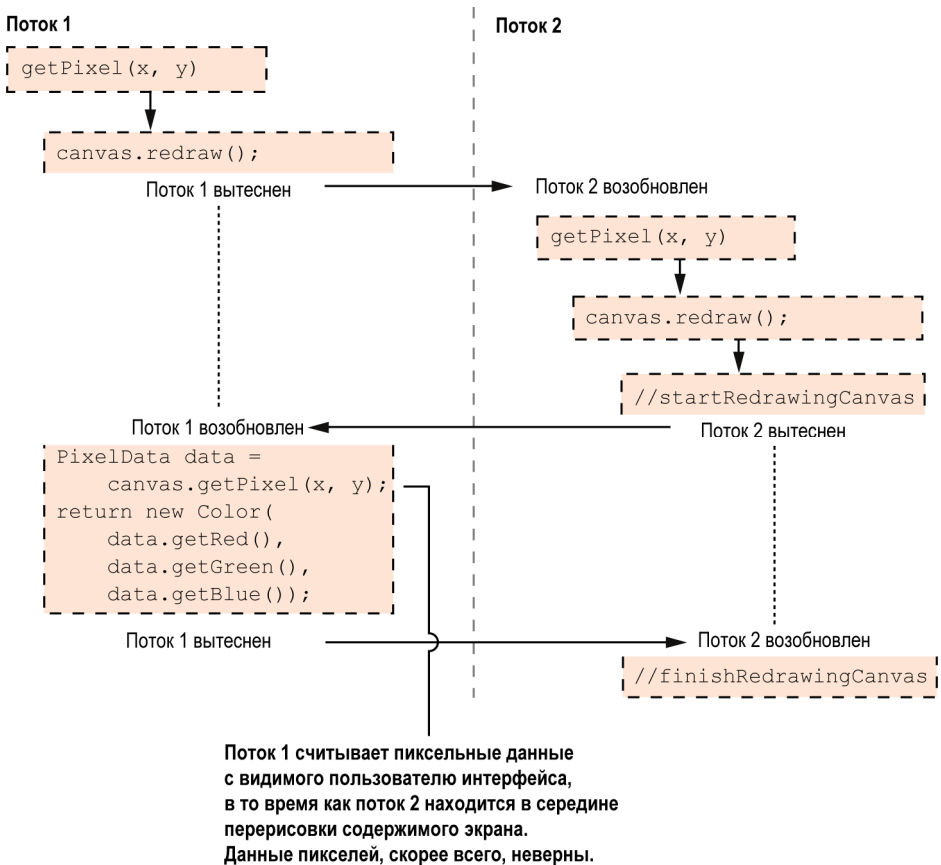


Рис. 6.1. Код с побочными эффектами часто может быть проблемным, если он когда-либо выполнялся в многопоточной среде и автор не предпринял активных шагов, чтобы сделать его потокобезопасным (например, с помощью блокировки)

6.3.3. Решение: избегайте побочного эффекта или сделайте его очевидным

Первый вопрос, который мы должны задать, заключается в том, необходимо ли вызывать `canvas.redraw()` перед чтением пикселя. Это может быть какой-то чрезмерно осторожный код, который не был должным образом продуман. И отсутствие побочного эффекта в первую очередь — лучший способ избежать сюрпризов. Если вызов `canvas.redraw()` не нужен, то мы должны просто удалить `canvas.redraw()`, это означает, что проблемы исчезнут.

Если необходимо вызвать `canvas.redraw()` перед чтением пикселя, то функцию `GetPixel()` следует переименовать, чтобы сделать этот побочный эффект очевидным. Лучшим названием было бы что-то вроде `redrawAndGetPixel()`, так что безошибочно очевидно, что эта функция имеет побочный эффект, вызывающий событие перерисовки. Это показано в следующем листинге.

Листинг 6.22. Более информативное название

```
class UserDisplay {
    private final Canvas canvas;
    ...
    Color redrawAndGetPixel(Int x, Int y) { ❶
        canvas.redraw();
        PixelData data = canvas.getPixel(x, y);
        return new Color(
            data.getRed(),
            data.getGreen(),
            data.getBlue());
    }
}
```

❶ Название функции делает побочный эффект очевидным.

Это очень простое изменение, которое нужно внести, и оно подчеркивает пользу правильных названий. Программист, вызывающий функцию `redrawAndGetPixel()`, теперь вынужден заметить, что она имеет побочный эффект и вызовет событие перерисовки. Это имеет большое значение для решения трех проблем, которые мы видели в предыдущем подразделе.

- ◆ Перерисовка звучит как нечто затратное, поэтому автор функции `CaptureScreenshot()`, вероятно, дважды подумает, прежде чем вызывать `redrawAndGetPixel()` тысячи раз внутри цикла `for`. Это предупреждает автора о том, что он, вероятно, захочет реализовать свою функцию по-другому, например, выполнить одну перерисовку, а затем прочитать все пиксели за один раз.
- ◆ Если автор функции `captureScreenshot()` тоже захочет назвать ее так, чтобы сделать побочный эффект очевидным, то он может дать ей название вроде `redrawAndCaptureScreenshot()`. Теперь программисту довольно сложно сделать

ошибочное предположение, что функция не вызывает события перерисовки, поскольку ее название прямо противоречит этому.

- ◆ Если функция называется `redrawAndCaptureScreenshot()`, то программист, реализующий функцию совместного использования экрана, сразу же осознает опасность ее вызова из многопоточной среды. Очевидно, ему придется проделать некоторую работу, чтобы сделать реализацию безопасной (например, с помощью блокировки), но это намного лучше, чем не обратить внимания на этот факт и столкнуться с неприятным сюрпризом.

Большинство функций, которые получают некоторую информацию, не вызывают побочных эффектов, поэтому для программиста естественно будет предположить, что в этой ситуации никаких проблем не будет. Таким образом, автору функции, которая вызывает побочный эффект, необходимо сделать этот факт безошибочно очевидным для всех вызывающих пользователей. Не вызывать побочного эффекта вообще — лучший способ избежать сюрприза, но это не всегда практично. Когда побочный эффект неизбежен, правильное название может быть очень эффективным способом сделать это очевидным.

6.4. Остерегайтесь изменения входных параметров

В предыдущем разделе обсуждалось, как неожиданные побочные эффекты могут вызвать проблемы. В этом разделе обсуждается конкретный тип побочного эффекта: функция, изменяющая входной параметр. Это может быть особенно распространённым источником сюрпризов и ошибок и, таким образом, заслуживает отдельного раздела.

6.4.1. Изменение входного параметра может привести к ошибкам

Если вы одолжите книгу другу, а он вернет ее с вырванными страницами и заметками, нацарапанными на полях, вы, вероятно, будете очень раздражены. Возможно, вы намереваетесь прочитать книгу самостоятельно или одолжить ее другому человеку, и вас ждет неприятный сюрприз, когда вы в конце концов поймете, что книга подверглась вандализму. Друг, который вырывает страницы и строчит на полях книги, которую вы ему одолжили, вероятно, плохой друг.

Передача объекта другой функции в качестве входных данных немного похожа на передачу книги другу. В этом объекте есть некоторая информация, необходимая другой функции, но также вполне вероятно, что объект все еще может понадобиться для других целей после вызова этой функции. Если функция изменяет входной параметр, существует реальный риск того, что она делает с кодом нечто, эквивалентное вырыванию страниц и написанию на полях. Вызывающие обычно передают функции объект с пониманием того, что объект заимствуется. Если функция в процессе подвергает объект вандализму, это немного похоже на плохого друга.

Трансформация (или изменение) входного параметра является еще одним примером побочного эффекта, поскольку функция влияет на что-то вне себя. Обычно функции принимают (или заимствуют) входные данные посредством параметров и предоставляют результаты посредством возвращаемых значений. Поэтому для большинства программистов изменение входного параметра является неожиданным побочным эффектом и, скорее всего, вызовет сюрприз.

В листинге 6.23 показано, как изменение входного параметра может привести к сюрпризам и ошибкам. В листинге показан код для обработки заказов для компании, продающей онлайн-услуги. Компания предлагает бесплатную пробную версию для новых пользователей. Функция `processOrders()` выполняет два действия: отправляет счета-фактуры для оплаты, а затем подключает заказанные услуги для каждого пользователя.

Функция `getBillableInvoices()` определяет, какие счета подлежат оплате. Счет-фактура подлежит оплате, если у пользователя нет бесплатной пробной версии. К сожалению, при расчете этого `getBillableInvoices()` изменяет один из своих входных параметров (карту `userInvoices`), удаляя все записи для пользователей с бесплатной пробной версией. Это вызывает ошибку в коде, потому что `processOrders()` позже повторно использует карту `userInvoices`, чтобы подключить заказанные пользователями услуги. Это означает, что для пользователей с бесплатной пробной версией никакие сервисы не заработают.

Листинг 6.23. Изменение входного параметра

```
List<Invoice> getBillableInvoices(  
    Map<User, Invoice> userInvoices,  
    Set<User> usersWithFreeTrial) {  
    userInvoices.removeAll(usersWithFreeTrial); ❶  
    return userInvoices.values();  
}  
  
void processOrders(OrderBatch orderBatch) {  
    Map<User, Invoice> userInvoices =  
        orderBatch.getUserInvoices();  
    Set<User> usersWithFreeTrial =  
        orderBatch.getFreeTrialUsers();  
  
    sendInvoices(  
        getBillableInvoices(userInvoices, usersWithFreeTrial)); ❷  
    enableOrderedServices(userInvoices); ❸  
}  
  
void enableOrderedServices(Map<User, Invoice> userInvoices) {  
    ...  
}
```

❶ Изменяет пользовательские запросы, удаляя все записи для пользователей с бесплатной пробной версией.

❷ `getBillableInvoices()` неожиданно изменяет `userInvoices`.

❸ Услуги не будут доступны для пользователей с бесплатной пробной версией.

Эта ошибка связана с тем, что функция `getBillableInvoices()` изменяет карту счетов-фактур пользователей (немного похоже на плохого друга, вырывающего страницы из заимствованной книги). Было бы намного лучше сделать так, чтобы эта функция не изменяла входной параметр.

6.4.2. Решение: скопируйте, прежде чем изменять

Если набор значений, содержащихся во входном параметре, действительно нуждается в изменении, то лучше всего скопировать их в новую структуру данных перед выполнением каких-либо изменений. Это предотвращает изменение исходного объекта. В следующем листинге показано, как выглядела бы функция `getBillableInvoices()`, если бы она так делала.

Листинг 6.24. Функция не изменяет входной параметр

```
List<Invoice> getBillableInvoices(  
    Map<User, Invoice> userInvoices,  
    Set<User> usersWithFreeTrial) {  
    return userInvoices  
        .entries() ❶  
        .filter(entry -> ❷  
            !usersWithFreeTrial.contains(entry.getKey()))  
        .map(entry -> entry.getValue());  
}
```

❶ Получает список всех пар ключевых значений карты `userInvoice`.

❷ `filter()` копирует все значения, соответствующие условию, в новый список.

Копирование значений может очевидно повлиять на производительность кода (с точки зрения использования памяти, загрузки процессора или того и другого). Часто это меньшее из двух зол по сравнению с сюрпризами и ошибками, которые могут возникнуть в результате изменения входного параметра. Но если существует вероятность, что фрагмент кода будет обрабатывать очень большие объемы данных, или существует вероятность использования посредственного аппаратного обеспечения, то изменение входного параметра может стать необходимым злом. Распространенным примером этого является сортировка списка или массива. Количество значений потенциально может быть довольно большим, и может быть намного эффективнее сортировать их на месте, а не создавать копию. Если нам действительно нужно изменить входной параметр по таким соображениям производительности, как это, то желательно убедиться, что в названии нашей функции (и в любой документации) ясно указано, что это произойдет.

ИЗМЕНЕНИЕ ПАРАМЕТРОВ ДОВОЛЬНО РАСПРОСТРАНЕНО

В некоторых языках и кодовых базах изменение параметров функции может быть довольно распространенным явлением. В C++ во многих кодах используется концепция выходных параметров, так как раньше было сложно возвращать классоподобные объ-

екты из функций эффективным и безопасным способом. В C++ теперь есть функции, которые делают выходные параметры менее распространенными в новом коде (например, семантика перемещения). Просто имейте в виду, что в некоторых языках изменение параметра более ожидаемо, чем в других.

В ЦЕЛЯХ ЗАЩИТЫ

В этом разделе мы обсуждали необходимость убедиться в том, что код, который мы пишем, ведет себя хорошо и не «разрушает» объекты, принадлежащие другому коду. Обратной стороной этого является защита объектов, принадлежащих нашему коду, от актов вандализма со стороны другого кода. В *главе 7* будет обсуждаться, как сделать объекты неизменяемыми, что может быть эффективным способом достижения этой цели.

6.5. Избегайте написания вводящих в заблуждение функций

Когда программист сталкивается с неким кодом, вызывающим функцию, он формирует представление о происходящем, основываясь на том, что он видит. Безошибочно очевидные части соглашения по написанию кода (например, названия) часто будут главными вещами, которые замечает программист, просматривая какой-либо код.

Как мы уже видели в этой главе, если в безошибочно очевидных частях кодового соглашения чего-то не хватает, это может вызвать сюрпризы. Однако еще хуже может то, что безошибочно очевидные части соглашения активно вводят в заблуждение. Если мы видим функцию с названием `displayLegalDisclaimer()`, мы предположим, что ее вызов отобразит юридическую оговорку. Если это не всегда так, это может легко привести к неожиданному поведению и ошибкам.

6.5.1. Бездействие при отсутствии критических входных данных может вызвать сюрпризы

Функция может вводить в заблуждение о том, что она делает, если она позволяет вызывать себя с отсутствующим параметром, а затем ничего не делает, когда этот параметр отсутствует. Вызывающие пользователи могут не осознавать важность вызова функции без указания значения для этого параметра, и любой, кто читает код, может быть введен в заблуждение, думая, что вызов функции всегда что-то делает.

В листинге 6.25 показан некий код для отображения юридического заявления об отказе от ответственности на дисплее пользователя. Функция `displayLegalDisclaimer()` берет некий юридический текст в качестве параметра и отображает его в виде наложения. Параметр `legalText` может быть равен значению `null`, и, когда это так, функция `displayLegalDisclaimer()` возвращает значение, ничего не отображая пользователю.

Листинг 6.25. Допускающий значение null, но критический параметр

```
class UserDisplay {
    private final LocalizedMessages messages;
    ...
    void displayLegalDisclaimer(String? legalText) { ❶
        if (legalText == null) { ❷
            return;
        }
        displayOverlay(
            title: messages.getLegalDisclaimerTitle(),
            message: legalText,
            textColor: Color.RED);
    }
}

class LocalizedMessages { ❸
    ...
    String getLegalDisclaimerTitle();
    ...
}
```

❶ `legalText` может быть равен значению `null`.

❷ Когда значение `legalText` равно `null`, функция возвращает значение, ничего не отображая.

❸ Содержит сообщения, переведенные на местный язык пользователя.

ЗАЧЕМ ПРИНИМАТЬ `NULL`, А ЗАТЕМ НИЧЕГО НЕ ДЕЛАТЬ?

Вы можете задаться вопросом, зачем кому-то понадобилось писать функцию, подобную приведенной в листинге 6.25. Ответ заключается в том, что программисты иногда делают это, чтобы вызывающим пользователям не приходилось проверять наличие значения `null` перед вызовом функции (как показано в следующем фрагменте). Их намерения благие: они пытаются разгрузить вызывающих, но, к сожалению, это может привести к вводящему в заблуждение и неожиданному коду.

```
...
String? message = getMessage();
if (message != null) { ❶
    userDisplay.displayLegalDisclaimer(message);
}
...
```

❶ Если функция `displayLegalDisclaimer()` не принимает значение `null`, то вызывающие должны проверять наличие значения `null`.

Чтобы понять, почему подобный код может вызывать сюрпризы, необходимо подумать о том, как будет выглядеть код при вызове функции `displayLegalDisclaimer()`. Представьте, что компания внедряет поток регистрации пользователей для получе-

ния услуги. Есть очень важные требования, которым должен соответствовать необходимый для этого код.

- ◆ Прежде чем пользователь сможет зарегистрироваться, компания по закону обязана предоставить им отказ от ответственности на их местном языке.
- ◆ Если отказ от ответственности не может быть отображен на местном языке пользователя, то регистрацию следует прервать. Продолжение могло бы потенциально нарушить закон.

Мы скоро рассмотрим полную реализацию, но сначала давайте сосредоточимся на функции, которая предназначена для обеспечения выполнения этих требований, — `ensureLegalCompliance()` и будет показана в следующем фрагменте. Программист, читающий этот код, скорее всего, придет к выводу, что отказ от ответственности отображается всегда. Так произойдет потому, что `userDisplay.displayLegalDisclaimer()` вызывается всегда, и ничто в безошибочно очевидной части его соглашения не говорит о том, что эта функция иногда ничего не делает.

```
void ensureLegalCompliance() {
    userDisplay.displayLegalDisclaimer(
        messages.getSignupDisclaimer());
}
```

В отличие от большинства программистов, читающих этот код, мы знакомы с деталями реализации `userDisplay.displayLegalDisclaimer()`, потому что мы видели их ранее (в листинге 6.25), и поэтому мы знаем, что если эта функция вызывается со значением `Null`, то она ничего не будет делать.

В листинге 6.26 показана полная реализация логики потока регистрации. Теперь мы можем видеть, что `messages.getSignupDisclaimer()` иногда может возвращать значение `null`. Это означает, что на самом деле функция `ensureLegalCompliance()` не всегда гарантирует соблюдение всех юридических требований. Компания, использующая этот код, вполне может нарушать закон.

Листинг 6.26. Вводящий в заблуждение код

```
class SignupFlow {
    private final UserDisplay userDisplay;
    private final LocalizedMessages messages;
    ...

    void ensureLegalCompliance() {
        userDisplay.displayLegalDisclaimer( ❶
            messages.getSignupDisclaimer());
    }
}

class LocalizedMessages {
    ...
```

```
// Возвращает значение null, если перевод на языке пользователя
// недоступен, поскольку использование языка по умолчанию для конкретного
// юридического текста может быть несовместимым.
String? getSignupDisclaimer() { ... } ❷
...
}
```

❶ Кажется, что код всегда отображает отказ от ответственности. На самом деле это не так.

❷ Возвращает значение `null`, если перевод на языке пользователя недоступен.

Немалая часть проблемы здесь заключается в том, что функция `userDisplay.displayLegalDisclaimer()` принимает значение, допускающее `null`, а затем ничего не делает, когда оно-таки равно `null`. Любой, кто читает фрагмент кода, который вызывает `displayLegalDisclaimer()`, подумает: «Как здорово, отказ от ответственности точно отобразится». На самом деле этому человеку пришлось бы знать, что, чтобы это утверждение было верным, функция не должна вызываться со значением `null`. В следующем подразделе объясняется, как мы можем избежать подобного потенциального сюрприза.

6.5.2. Решение: сделать необходимые критические входные данные обязательными

Если сделать критический параметр допускающим значение `null`, это будет означать, что вызывающим абонентам не нужно будет проверять наличие значения `null` перед вызовом. Это может сделать код вызывающего абонента более кратким, но, к сожалению, это также может сделать этот код вводящим в заблуждение. Как правило, это не очень хороший компромисс: код вызывающего абонента становится немного короче, но в процессе значительно возрастает вероятность путаницы и ошибок.

Какой-либо параметр является критическим для функции, если данная функция не может делать то что должна, без этого параметра. Если у нас есть такой параметр, часто бывает безопаснее сделать его обязательным, чтобы невозможно было вызвать функцию, если значение недоступно.

В листинге 6.27 показана функция `displayLegalDisclaimer()`, измененная таким образом, что она принимает только параметр, не равный значению `null`. Вызов функции `displayLegalDisclaimer()` теперь гарантированно отображает отказ от ответственности. Все пользователи `displayLegalDisclaimer()` вынуждены сталкиваться с тем фактом, что если у них нет юридического текста, они не смогут отобразить отказ от ответственности.

Листинг 6.27. Обязательный критический параметр

```
class UserDisplay {
    private final LocalizedMessages messages;
    ...
}
```

```
void displayLegalDisclaimer(String legalText) { ❶
    displayOverlay( ❷
        title: messages.getLegalDisclaimerTitle(),
        message: legalText,
        textColor: Color.RED);
}
```

❶ `legalText` не может быть равен `null`.

❷ Отказ от ответственности всегда будет отображаться.

Код в функции `ensureLegalCompliance()` теперь вынужден быть намного менее вводящим в заблуждение. Автор кода поймет, что ему придется работать со сценарием, в котором нет перевода. В листинге 6.28 показано, как теперь может выглядеть код для функции `ensureLegalCompliance()`. Теперь такой код должен проверить, доступен ли локализованный юридический текст, и если его нет, это сигнализирует о том, что соответствие не может быть обеспечено путем возврата значения `false`. Функция также снабжена аннотацией `CheckReturnValue`, чтобы убедиться, что возвращаемое значение не игнорируется (как описано в *главе 4*).

Листинг 6.28. Однозначный код

```
class SignupFlow {
    private final UserDisplay userDisplay;
    private final LocalizedMessages messages;
    ...

    // Возвращает значение false, если соответствие не
    // может быть обеспечено, что означает, что от
    // регистрации следует отказаться. Возвращает значение true, если
    // соответствие было обеспечено.
    @CheckReturnValue ❶
    Boolean ensureLegalCompliance() { ❷
        String? signupDisclaimer = messages.getSignupDisclaimer();
        if (signupDisclaimer == null) {
            return false; ❸
        }
        userDisplay.displayLegalDisclaimer(signupDisclaimer); ❹
        return true;
    }
}
```

❶ Гарантирует, что возвращаемое значение не будет проигнорировано.

❷ Возвращает логический тип `Boolean`, указывающий на то, было ли обеспечено соответствие.

❸ Возвращает значение `false`, если соответствие не было обеспечено.

❹ Вызов `displayLegalDisclaimer()` всегда будет отображать отказ от ответственности.

В главе 5 говорилось о том, почему важно не заикливаться на общем количестве строк кода в ущерб другим аспектам качества кода. Перемещение оператора `if-null` к вызывающему может увеличить количество строк кода (особенно если вызывающих много), но это также снижает вероятность того, что код будет неверно истолкован или сделает что-то неожиданное. Потребуется, скорее всего, потратить гораздо больше времени и усилий на исправление даже одной ошибки, вызванной каким-то неожиданным кодом, чем на чтение нескольких дополнительных операторов `if-null`. Преимущества ясности и однозначности кода часто намного перевешивают затраты на несколько дополнительных строк кода.

6.6. Обработка перечислений, ориентированная на будущее

Примеры, приведенные до сих пор в этой главе, были сосредоточены на том, чтобы пользователи нашего кода не были удивлены тем, что он делает или возвращает, другими словами, чтобы код, зависящий от нашего кода, был правильным и без ошибок. Однако сюрпризы также могут случиться, если мы сделаем хрупкие предположения о коде, от которого мы зависим. В этом разделе демонстрируется пример такой ситуации.

Перечисления вызывают разногласия среди разработчиков программного обеспечения. Некоторые утверждают, что это отличный и простой способ обеспечить безопасность типов и избежать недопустимых входных данных для функций или систем. Другие утверждают, что перечисления предотвращают чистые слои абстракции, потому что логика обработки определенного значения перечисления в конечном итоге распространяется повсюду. Программисты из этой последней группы часто утверждают, что полиморфизм является лучшим подходом: инкапсулируйте информацию и поведение для каждого значения внутри класса, посвященного этому значению, а затем все эти классы реализуют общий интерфейс.

Независимо от вашего личного мнения о перечислениях, велика вероятность, что вы столкнетесь с ними и вам придется иметь с ними дело. Это может быть связано с тем, что

- ◆ вам приходится использовать выходные данные чужого кода, и автор по какой-то причине очень любит перечисления; или
- ◆ вы используете выходные данные, предоставляемые другой системой. Перечисления часто могут быть единственным практичным вариантом в формате данных по проводам.

Когда вам действительно приходится обрабатывать перечисление, часто важно помнить, что в будущем в перечисление может быть добавлено больше значений. Если вы пишете код, игнорирующий этот факт, то можете преподнести неприятные сюрпризы себе или другим программистам.

6.6.1. Неявная обработка будущих перечисляемых значений может быть источником проблем

Иногда программисты смотрят на текущий набор значений в перечислении и думают: «Как здорово, я могу справиться с этим с помощью оператора `if`». Это может сработать для набора значений, которые в настоящее время имеет перечисление, но часто оно неустойчиво к большому количеству значений, добавляемых в будущем.

Чтобы продемонстрировать это, представьте, что компания разработала модель для прогнозирования того, что произойдет, если она будет следовать определенной бизнес-стратегии. Листинг 6.29 содержит определение перечисления, которое указывает на прогноз, полученный из модели. В листинге также содержится функция, которая использует прогноз модели, а затем указывает, является ли это безопасным результатом. Если функция `isOutcomeSafe()` возвращает значение `true`, то последующая автоматизированная система инициирует бизнес-стратегию. Если она вернет значение `false`, бизнес-стратегия не будет запущена.

В настоящее время перечисление `PredictedOutcome` содержит только два значения: `COMPANY_WILL_GO_BUST` и `COMPANY_WILL_MAKE_A_PROFIT`. Программист, пишущий функцию `isOutcomeSafe()`, заметил, что один из этих результатов безопасен, а другой — нет, и поэтому решил обработать перечисление с помощью простого оператора `if`. Функция `isOutcomeSafe()` явно обрабатывает случай `COMPANY_WILL_GO_BUST` как небезопасный и неявно обрабатывает все остальные перечисляемые значения как безопасные.

Листинг 6.29. Неявная обработка перечисляемых значений

```
enum PredictedOutcome {  
    COMPANY_WILL_GO_BUST, ❶  
    COMPANY_WILL_MAKE_A_PROFIT,  
}  
...  
Boolean isOutcomeSafe(PredictedOutcome prediction) {  
    if (prediction == PredictedOutcome.COMPANY_WILL_GO_BUST) { ❷  
        return false;  
    }  
    return true; ❸  
}
```

❶ Два перечисляемых значения.

❷ Результат `COMPANY_WILL_GO_BUST` явно обрабатывается как небезопасный.

❸ Все остальные перечисляемые значения неявно обрабатываются как безопасные.

Код в предыдущем листинге работает, пока есть только два перечисляемых значения. Но все может пойти прахом, если кто-то введет новое значение. Представьте,

что модель и перечисление теперь обновлены при помощи нового потенциального результата: `WORLD_WILL_END`. Как следует из названия («произойдет конец света»), это перечисляемое значение указывает на предсказание модели о том, что всему миру придет конец, если компания инициирует данную бизнес-стратегию. Определение перечисления теперь выглядит следующим образом.

Листинг 6.30. Новое перечисляемое значение

```
enum PredictedOutcome {  
    COMPANY_WILL_GO_BUST,  
    COMPANY_WILL_MAKE_A_PROFIT,  
    WORLD_WILL_END,    ❶  
}
```

❶ Значение, указывающее на предсказание о конце света.

Определение функции `isOutcomeSafe()` может находиться на расстоянии многих сотен строк кода от определения перечисления или находиться в совершенно другом файле или пакете. Они также могут обслуживаться совершенно разными командами. Поэтому небезопасно предполагать, что любой программист, добавляющий перечисляемое значение в тип `PredictedOutcome`, будет знать о необходимости обновить также функцию `isOutcomeSafe()`.

Если функция `isOutcomeSafe()` не обновлена (что повторяется в следующем фрагменте), она вернет значение `true` для предсказания `WORLD_WILL_END`, что укажет на безопасность результата. Очевидно, что `WORLD_WILL_END` не является безопасным результатом, и, если система нижестоящего звена инициирует какую-либо бизнес-стратегию с таким прогнозируемым результатом, последствия будут катастрофическими.

```
Boolean isOutcomeSafe(PredictedOutcome prediction) {  
    if (prediction == PredictedOutcome.COMPANY_WILL_GO_BUST) {  
        return false;  
    }  
    return true;    ❶  
}
```

❶ Возвращает значение `true`, если предсказание равно `WORLD_WILL_END`.

Автор функции `isOutcomeSafe()` проигнорировал тот факт, что в будущем может быть добавлено больше перечисляемых значений. В результате код содержит хрупкое и ненадежное предположение, которое может привести к катастрофическому исходу. Маловероятно, что реальный сценарий приведет к концу света, но последствия для организации все равно могут быть серьезными, если данные клиентов были неправильно обработаны или были приняты неправильные автоматизированные решения.

6.6.2. Решение: используйте исчерпывающий оператор switch

Проблема кода из предыдущего подраздела заключается в том, что функция `isOutcomeSafe()` неявно обрабатывает некоторые перечисляемые значения, вместо того чтобы делать это явно. Лучшим подходом было бы явно обработать все известные перечисляемые значения, а затем убедиться, что код либо перестанет компилироваться, либо тест завершится неудачно, если будет добавлено новое необработанное перечисляемое значение.

Распространенным способом достижения этой цели является использование исчерпывающего оператора `switch`. В листинге 6.31 показано, как выглядит функция `isOutcomeSafe()`, если она использует этот подход. Если оператор `switch` когда-либо завершается без совпадения одного из случаев, это указывает на то, что было обнаружено необработанное перечисляемое значение. Если так происходит, это означает, что произошла ошибка программирования: программисту не удалось обновить код в функции `isOutcomeSafe()` для обработки нового перечисляемого значения. Об этом сигнализирует создание непроверенного исключения, гарантирующего быструю и громкую ошибку кода (как обсуждалось в главе 4).

Листинг 6.31. Исчерпывающий оператор `switch`

```
enum PredictedOutcome {
    COMPANY_WILL_GO_BUST,
    COMPANY_WILL_MAKE_A_PROFIT,
}
...
```

Listing 6.31 An exhaustive switch statement

```
Boolean isOutcomeSafe(PredictedOutcome prediction) {
    switch (prediction) {
        case COMPANY_WILL_GO_BUST: ❶
            return false;
        case COMPANY_WILL_MAKE_A_PROFIT:
            return true;
    }
    throw new UncheckedException(
        "Unhandled prediction: " + prediction); ❷
}
```

❶ Каждое перечисляемое значение обрабатывается явно.

❷ Необработанное перечисляемое значение является ошибкой программирования, поэтому возникает непроверенное исключение.

Этот подход может быть объединен с модульным тестом, который выполняет вызов функции с каждым потенциальным перечисляемым значением. Если для любо-

го значения возникает исключение, то тест завершится неудачей, и программист, добавляющий новое значение в тип `PredictedOutcome`, будет проинформирован о необходимости обновления функции `isOutcomeSafe()`. В следующем листинге показано, как может выглядеть такой модульный тест.

Листинг 6.32. Модульное тестирование всех перечисляемых значений

```
testIsOutcomeSafe_allPredictedOutcomeValues() {  
    for (PredictedOutcome prediction in  
        PredictedOutcome.values()) {❶  
        isOutcomeSafe(prediction);    ❷  
    }  
}
```

❶ Перебирает каждое значение в перечислении

❷ Если из-за необработанного значения возникает исключение, тест завершится неудачей.

Предполагая, что определение перечисления `PredictedOutcome` и функция `isOutcomeSafe()` являются частью одной и той же кодовой базы и что существуют достаточные проверки перед отправкой, программисту запрещается отправлять свой код до тех пор, пока он не обновит функцию `isOutcomeSafe()`. Это заставляет программиста обратить внимание на проблему, и тогда он обновит функцию, чтобы явно обработать значение `WORLD_WILL_END`. В следующем листинге показан обновленный код.

Листинг 6.33. Обработка нового перечисления

```
Boolean isOutcomeSafe(PredictedOutcome prediction) {  
    switch (prediction) {  
        case COMPANY_WILL_GO_BUST:  
        case WORLD_WILL_END:    ❶  
            return false;  
        case COMPANY_WILL_MAKE_A_PROFIT:  
            return true;  
    }  
    throw new UncheckedException(  
        "Unhandled prediction: " + prediction);  
}
```

❶ Перечисляемое значение `WORLD_WILL_END` обработано явно.

С обновленным кодом тест `testIsOutcomeSafe_allPredictedOutcomeValues()` снова проходит удачно. Если программист выполняет свою работу должным образом, то он также добавит дополнительный тестовый результат для обеспечения того, что функция `isOutcomeSafe()` будет возвращать значение `false` для прогноза `WORLD_WILL_END`.

Используя исчерпывающий оператор `switch` в сочетании с модульным тестированием, удастся избежать неприятного сюрприза и потенциально катастрофической ошибки в коде.

БЕЗОПАСНОСТЬ ВО ВРЕМЯ КОМПИЛЯЦИИ

В некоторых языках (например, C++) компилятор может выдавать предупреждение для оператора `switch`, когда он не полностью обрабатывает каждое перечисляемое значение. Если настройки сборки у вашей команды сконфигурированы таким образом, чтобы предупреждения рассматривались как ошибки, то это может быть очень эффективным способом немедленного выявления подобной ошибки. Также часто по-прежнему рекомендуется создавать исключение (или каким-либо образом быстро завершать работу) для необработанного значения, если оно может быть получено из другой системы. Это связано с тем, что в другой системе может быть запущена более обновленная версия кода, включающая новое перечисляемое значение, в то время как текущая версия вашего кода может быть старой и не содержать обновленной логики оператора `switch`.

6.6.3. Остерегайтесь результата по умолчанию

Операторы `switch` обычно поддерживают результат по умолчанию, который является универсальным для любых необработанных значений. Добавление одного из них в оператор `switch`, который обрабатывает перечисление, может привести к неявной обработке будущих перечисляемых значений и потенциально приводить к сюрпризам и ошибкам.

Если в функцию `isOutcomeSafe()` будет добавлен результат по умолчанию, тогда все будет выглядеть, как в листинге 6.34. Теперь функция по умолчанию возвращает значение `false` для любых новых перечисляемых значений. Это означает, что любая бизнес-стратегия с прогнозом, который явно не обрабатывается, считается небезопасной и не иницируется. Это может показаться разумным результатом по умолчанию, однако это не обязательно так на самом деле. Новый результат прогнозирования может быть `COMPANY_WILL_AVOID_LAWSUIT`, и в этом случае значение `false` по умолчанию определенно нецелесообразно. Использование результата по умолчанию приводит к неявной обработке новых перечисляемых значений, и, как мы установили ранее в этом разделе, может привести к сюрпризам и ошибкам.

Листинг 6.34. Результат по умолчанию

```
Boolean isOutcomeSafe(PredictedOutcome prediction) {  
    switch (prediction) {  
        case COMPANY_WILL_GO_BUST:  
            return false;  
        case COMPANY_WILL_MAKE_A_PROFIT:  
            return true;  
    }
```

```

default: ❶
    return false;
}
}

```

❶ По умолчанию возвращается значение `false` для любых новых перечисляемых значений.

Возникновение ошибки из результата по умолчанию

Другая ситуация, в которой иногда используется результат по умолчанию, это создание исключения, указывающего на то, что перечисляемое значение не обработано. Это показано в листинге 6.35. Код лишь незначительно отличается от версии, которую мы видели ранее в листинге 6.33: оператор `throw new UncheckedException()` теперь используется по умолчанию, а не находится за пределами оператора `switch`. Это может показаться несущественным стилистическим выбором, но на некоторых языках это может сделать код немного более подверженным ошибкам, закрадывающимся едва заметными способами.

Некоторые языки (например, C++) могут отображать предупреждение компилятора, когда оператор `switch` не полностью обрабатывает все значения. Это очень полезное предупреждение. Даже если у нас есть модульный тест, который должен обнаружить необработанное перечисляемое значение, действительно не мешает иметь дополнительный уровень защиты, который предлагает предупреждение компилятора. Предупреждение компилятора может быть замечено до сбоя теста, что экономит программистам время. А также всегда существует некоторый риск того, что тест может быть случайно удален или отключен. Добавив результат по умолчанию в оператор `switch` (как в листинге 6.35), компилятор теперь определит, что оператор `switch` обрабатывает все значения, даже если в будущем в перечисление будут добавлены новые. Это означает, что компилятор не выдаст предупреждение, и дополнительный уровень защиты будет потерян.

Листинг 6.35. Исключение в результате по умолчанию

```

Boolean isOutcomeSafe(PredictedOutcome prediction) {
    switch (prediction) {
        case COMPANY_WILL_GO_BUST:
            return false;
        case COMPANY_WILL_MAKE_A_PROFIT:
            return true;
        default: ❶
            throw new UncheckedException(
                "Unhandled prediction: " + prediction); ❷
    }
}

```

❶ Результат по умолчанию означает, что компилятор всегда будет думать, что все значения обработаны.

❷ Исключение, созданное из результата по умолчанию.

Чтобы убедиться, что компилятор по-прежнему выводит предупреждение для не-обработанных перечисляемых значений, лучше поместить оператор `throw new UncheckedException()` после оператора `switch`. Код, который мы видели ранее в этом разделе (в листинге 6.31), демонстрирующий это, повторяется в следующем листинге.

Листинг 6.35. Исключение после оператора `switch`

```
Boolean isOutcomeSafe(PredictedOutcome prediction) {  
    switch (prediction) {  
        case COMPANY_WILL_GO_BUST:  
            return false;  
        case COMPANY_WILL_MAKE_A_PROFIT:  
            return true;  
    }  
    throw new UncheckedException(  
        "Unhandled prediction: " + prediction); ❶  
}
```

❶ Исключение, возникающее после оператора `switch`.

6.6.4. Предостережение: полагаться ли на перечисление из другого проекта

Иногда наш код может основываться на перечислении, принадлежащем другому проекту или организации. То, как мы должны обращаться с этим перечислением, будет зависеть от характера наших отношений с этим другим проектом, а также от нашего собственного цикла разработки и выпуска. Если другой проект с большой долей вероятности без предупреждения добавит новые перечисляемые значения, и это немедленно повредит наш код, у нас может не быть другого выбора, кроме как быть более снисходительными в том, как мы обрабатываем новые значения. Как и во многих других вещах, нам нужно использовать рассудительность.

6.7. Нельзя ли все эти проблемы решить тестированием?

Аргумент, который иногда выдвигается против усилий по обеспечению качества кода, направленных на предотвращение сюрпризов, заключается в том, что это пустая трата времени, потому что все эти проблемы должны выявляться тестами. По моему опыту это несколько идеалистический аргумент, который не работает в реальности.

В тот момент, когда вы пишете код, у вас, вероятно, есть контроль над тем, как вы тестируете этот код. Вы можете быть чрезвычайно прилежны и хорошо осведомле-

ны о тестировании и написать почти идеальный набор тестов, которые фиксируют все правильные модели поведения и допущения для вашего кода. Но избегание сюрпризов — это не только техническая корректность вашего собственного кода. Речь также идет о том, чтобы убедиться, что код, написанный другими программистами и вызывающий ваш код, функционирует правильно. Одного тестирования может быть недостаточно для обеспечения этого по следующим причинам.

- ◆ Другие программисты могут быть не столь усердны в тестировании, что означает, что они не тестируют достаточно сценариев или критических случаев, чтобы выявить, что предположение, которое они сделали о вашем коде, неверно. Это может быть особенно актуально, если проблема проявляется только в определенных сценариях или при очень больших входных данных.
- ◆ Тесты не всегда точно имитируют реальный мир. Программист, тестирующий свой код, может быть вынужден имитировать одну из зависимостей. Если это произойдет, он запрограммирует имитацию таким образом, чтобы она вела себя так, как, *по его мнению*, ведет себя имитируемый код. Если реальный код ведет себя неожиданным образом, а программист этого не осознает, то он, скорее всего, неправильно запрограммирует имитацию. Если это произойдет, ошибка, вызванная неожиданным поведением, может никогда не всплыть во время тестирования.
- ◆ Некоторые вещи очень трудно проверить. В разделе о побочных эффектах показано, как они могут вызывать проблемы для многопоточного кода. Ошибки, связанные с проблемами многопоточности, как известно, трудно проверить, поскольку они часто возникают с низкой вероятностью и проявляются только при масштабировании кода.

Эти же пункты применимы в отношении того, чтобы сделать код трудным для неправильного использования (описано в *главе 7*). Повторяю, тестирование чрезвычайно важно. Никакое структурирование кода или беспокойство о соглашениях по написанию кода никогда не заменит необходимость высококачественного и тщательного тестирования.

Но, по моему опыту, верно и обратное: тестирование само по себе не компенсирует неинтуитивный или полный сюрпризов код.

Итоги

Код, который мы пишем, часто зависит от кода, написанного другими программистами.

- ◆ Если другие программисты неправильно истолкуют то, что делает наш код, или не смогут заметить особые сценарии, которые им необходимо обработать, то, скорее всего, код, построенный поверх нашего, будет содержать ошибки.

- ◆ Один из лучших способов избежать сюрпризов для вызывающих пользователей кода — это убедиться, что важные детали содержатся в безошибочно очевидной части соглашения по написанию кода.

Еще один источник сюрпризов может возникнуть, если мы сделаем неверные предположения о коде, от которого мы зависим.

- ◆ Примером этого является неспособность предвидеть добавление новых значений в перечисление.
- ◆ Важно убедиться, что либо наш код перестанет компилироваться, либо тест завершится неудачей, если код, от которого мы зависим, нарушит одно из наших предположений.

Тестирование само по себе не компенсирует код, который преподносит сюрпризы: если другой программист неверно истолкует наш код, он также может неверно истолковать, какие сценарии ему нужно протестировать.

7 Постарайтесь затруднить неверное использование кода

ЭТА ГЛАВА ОХВАТЫВАЕТ

- ◇ Как неправильное использование кода может привести к ошибкам
- ◇ Распространенные способы, с помощью которых код может быть неправильно использован
- ◇ Методы, затрудняющие неправильное использование кода

В *главе 3* обсуждалось, как код, который мы пишем, нередко является лишь частью мозаики гораздо более объемного программного обеспечения. Чтобы часть программного обеспечения функционировала правильно, различные фрагменты кода должны соответствовать друг другу и работать. Если фрагмент кода легко использовать не по назначению, то есть вероятность, что рано или поздно он будет использован не по назначению и программное обеспечение будет работать неправильно.

Кодом часто легко злоупотреблять, когда в нем заложены неинтуитивные или двусмысленные предположения, и другим инженерам не мешают делать неправильные вещи. Некоторые распространенные способы неправильного использования кода заключаются в следующем.

Вызывающие абоненты предоставляют неверные входные данные:

- ◆ побочные эффекты от других частей кода (например, изменение входных параметров);
- ◆ вызывающие абоненты не вызывают функции в нужное время или в правильном порядке (как показано в *главе 3*);
- ◆ связанный фрагмент кода, изменяемый таким образом, нарушает определение.

Написание документации и предоставление инструкций по использованию кода могут помочь избежать этих проблем. Но, как мы видели в *главе 3*, они похожи на мелкий шрифт в кодовом соглашении и часто могут быть упущены из виду и устареть. Поэтому важно разрабатывать и писать код таким образом, чтобы его было трудно использовать не по назначению. В этой главе показаны некоторые общие способы, с помощью которых код можно легко измерить, и продемонстрированы методы, позволяющие вместо этого затруднить его неправильное использование.

ТРУДНО ИСПОЛЬЗОВАТЬ НЕПРАВИЛЬНО

Идея избегания проблем путем делания вещей трудными (или невозможными) для неправильного использования является устоявшимся принципом в проектировании и производстве. Примером этого является концепция бережливого производства рока уоке^a, разработанная Сигео Синго в 1960-х годах в контексте сокращения дефектов при производстве автомобилей. В более общем плане это общая черта принципов оборонительного проектирования. Приведем реальные примеры того, как сделать вещи трудными для неправильного использования.

Многие конструкции кухонных комбайнов будут работать только при правильном креплении крышки. Это предотвращает случайное вращение лезвий, когда рядом с ними могут находиться чьи-то пальцы.

Разные розетки и вилки имеют разную форму; например, вилку питания нельзя подключить к разъему HDMI (этот пример использовался в *главе 1*).

Рукоятка для управления катапультируемым креслом в реактивном истребителе расположена достаточно далеко от других органов управления самолетом, что сводит к минимуму вероятность его случайного срабатывания. В более старых конструкциях катапультируемых кресел (с верхней выдвижной ручкой) положение ручки также означало, что при ее нажатии пассажир выпрямлял спину^b (снижая риск получения травмы во время катапультирования), поэтому положение ручки одновременно выполняло две функции с точки зрения затруднения неправильного использования.

В мире разработки программного обеспечения этот принцип иногда выражается в утверждении, что API и интерфейсы должны быть «простыми в использовании и трудными для неправильного использования», что иногда сокращенно называется EUNM (англ. *easy to use and hard to misuse*).

^a <https://tulip.co/ebooks/poka-yoke/>

^b <http://mng.bz/XYM1>

7.1. Подумайте о том, чтобы сделать вещи неизменными

Что-то является *неизменным*, если его состояние не может быть изменено после его создания. Чтобы понять, почему желательна неизменность, важно рассмотреть, как противоположное, *изменяемость*, может вызвать проблемы. Некоторые проблемы, связанные с изменяемостью вещей, уже возникали в этой книге.

В *главе 3* мы видели, как наличие изменяемого класса с функциями настройки облегчает неправильную настройку, в результате чего он находится в недопустимом состоянии. В *главе 6* мы видели, как функция, изменяющая входной параметр, может вызвать неприятный сюрприз.

Кроме того, существует еще больше причин, по которым изменяемость может вызвать проблемы, в том числе следующие.

- ♦ *Изменяемый код сложнее понять.* Чтобы проиллюстрировать это, давайте рассмотрим сценарий реального мира, который в чем-то аналогичен. Если вы купите пакет сока в магазине, он, скорее всего, будет защищен от взлома. Это позволяет вам знать, что содержимое коробки не претерпело изменений между выходом с завода и вашей покупкой. Вам очень легко с высокой степенью уверен-

ности узнать, что находится внутри упаковки (сок) и кто его туда положил (производитель). Теперь представьте, что в магазине есть коробка сока без печати: кто знает, что могло случиться с этой коробкой? Возможно, он содержит какую-то грязь или кто-то гнусный мог что-то добавить к нему. Очень трудно рассуждать о том, что именно находится в коробке и кто мог это туда положить. При написании кода, если объект неизменяем, то это немного похоже на наличие защищенной от несанкционированного доступа печати, которую никто никогда не сможет взломать. Вы можете передавать объект повсюду и с уверенностью знать, что никто его не изменял и ничего к нему не добавлял.

- ◆ *Изменяемый код может вызвать проблемы с многопоточностью.* В главе 6 мы видели, как побочные эффекты могут вызывать проблемы в многопоточном коде. Если объект является изменяемым, то многопоточный код, использующий этот объект, может быть особенно подвержен проблемам. Если один поток находится в процессе чтения из объекта, в то время как другой изменяет его, может возникнуть ошибка. Примером этого может быть ситуация, когда один поток как раз собирается прочитать последний элемент в списке, в то время как другой поток удаляет этот элемент из списка.

Не всегда возможно или уместно делать вещи неизменными. Неизбежно существуют некоторые части нашего кода, которые должны отслеживать изменение состояния, и для этого, очевидно, потребуются некоторые виды изменяемых структур данных. Но, как мы только что объяснили, наличие изменяемых объектов может увеличить сложность кода и привести к проблемам, поэтому часто бывает хорошей идеей занять позицию по умолчанию, согласно которой вещи должны быть как можно более неизменными, и делать вещи изменяемыми только там, где это необходимо.

7.1.1. Изменяемые классы могут быть легко использованы не по назначению

Одним из наиболее распространенных способов сделать класс изменяемым является предоставление функций настройки. В листинге 7.1 показан пример этого. Класс `TextOptions` содержит информацию о стилизации для отображения некоторого текста. Как шрифт, так и размер шрифта можно задать, вызвав функции `setFont()` и `setFontSize()` соответственно.

В этом примере нет ограничений на то, кто может вызывать функции `setFont()` и `setFontSize()`, поэтому любой код, имеющий доступ к экземпляру `TextOptions`, может изменять шрифт или размер шрифта. Это может очень упростить неправильное использование экземпляра класса `TextOptions`.

Листинг 7.1. Изменяемый класс

```
class TextOptions {  
    private Font font;  
    private Double fontSize;
```



```

TextOptions(Font font, Double fontSize) {
    this.font = font;
    this.fontSize = fontSize;
}

void setFont(Font font) { ❶
    this.font = font;
}

void setFontSize(Double fontSize) { ❷
    this.fontSize = fontSize;
}

Font getFont() {
    return font;
}

Double getFontSize() {
    return fontSize;
}
}

```

❶ Шрифт можно изменить в любое время, вызвав `setFont()`.

❷ Размер шрифта можно изменить в любое время, вызвав `SetFontSize()`.

В листинге 7.2 показано, как можно неправильно использовать экземпляр `TextOptions`. Функция `sayHello()` создает экземпляр `TextOptions` с некоторой информацией о стилях по умолчанию. Он передает этот экземпляр в `messageBox.renderTitle()`, а затем в `messageBox.renderMessage()`. К сожалению, `messageBox.renderTitle()` изменяет `TextOptions`, устанавливая размер шрифта на 18. Это означает, что `messageBox.renderMessage()` вызывается с `TextOptions`, которые определяют размер шрифта 18 (вместо предполагаемого значения 12).

В главе 6 мы видели, что изменение входного параметра часто является плохой практикой, поэтому функция `messageBox.renderTitle()`, вероятно, не лучший код. Но, несмотря на разочарование, подобный код все еще может существовать в базе кода, и в настоящее время класс `TextOptions` ничего не делает для защиты от такого рода злоупотреблений.

Листинг 7.2. Ошибка из-за изменяемости

```

class UserDisplay {
    private final MessageBox messageBox;
    ...

    void sayHello() {
        TextOptions defaultStyle = new TextOptions(Font.ARIAL,
            12.0); ❶
    }
}

```

```
        messageBox.renderTitle("Important message",
                                defaultStyle);
        messageBox.renderMessage("Hello", defaultStyle); ❷
    }
}
...

class MessageBox {
    private final TextField titleField;
    private final TextField messageField;
    ...

    void renderTitle(String title, TextOptions baseStyle) {
        baseStyle.setFontSize(18.0); ❸
        titleField.display(title, baseStyle);
    }

    void renderMessage(String message, TextOptions style) {
        messageField.display(message, style);
    }
}
```

❶ Создает экземпляр `TextOptions`.

❷ Передает экземпляр в `messageBox.renderTitle()`, а затем в `MessageBox.renderMessage()`.

❸ Экземпляр `TextOptions` изменяется путем изменения размера шрифта.

Поскольку класс `TextOptions` является изменяемым, любой код, который передает его экземпляр в какой-либо другой код, подвергается риску неправильного использования из-за изменений. Было бы намного лучше, если бы код мог свободно передавать экземпляр `TextOptions` и знать, что он не будет видоизменен. Как и упаковка из-под сока, мы хотим, чтобы класс `TextOptions` имел защищенную от несанкционированного доступа печать. В следующих двух подразделах демонстрируются некоторые способы достижения этой цели.

7.1.2. Решение: устанавливайте значения только во время проектирования

Мы можем сделать класс неизменяемым (и предотвратить его неправильное использование), убедившись, что все значения предоставлены во время создания и что они не могут быть изменены после этого. В листинге 7.3 показан класс `TextOptions` с удаленными функциями настройки. Это предотвращает изменение переменных-членов `font` и `fontSize` в любом коде вне класса.

При определении переменной в классе часто можно предотвратить ее переназначение даже с помощью кода внутри класса. Как это сделать, может варьироваться в зависимости от языка, но общие ключевые слова — `const`, `final` или `readonly`. В соглашении о псевдокоде в этой книге используется ключевое слово `final` для этой концепции. Переменные `font` и `fontSize` помечены как окончательные. Это предотвращает случайное добавление кода в класс для их переназначения и ясно дает понять, что они не будут (и не должны) когда-либо изменяться.

Листинг 7.3. Неизменяемый класс `TextOptions`

```
class TextOptions {
    private final Font font;
    private final Double fontSize; ❶

    TextOptions(Font font, Double fontSize) {
        this.font = font;
        this.fontSize = fontSize; ❷
    }

    Font getFont() {
        return font;
    }

    Double getFontSize() {
        return fontSize;
    }
}
```

❶ Переменные-члены, помеченные как окончательные (`final`).

❷ Переменные-члены, заданные только во время проектирования.

Теперь это делает невозможным неправильное использование объекта `TextOptions` другим кодом путем его изменения. Но это еще не все, потому что функция `MessageBox.renderTitle()`, которую мы видели ранее, нуждается в способе переопределения только размера шрифта некоторых `TextOptions`. Для этого мы можем использовать шаблон копирования при записи. Это будет рассмотрено в следующем подразделе, но в итоге функция `MessageBox.renderTitle()` будет иметь вид, как в следующем листинге.

Листинг 7.4. `TextOptions` не изменен

```
class MessageBox {
    private final TextField titleField;
    ...

    void renderTitle(String title, TextOptions baseStyle) {
        titleField.display(
```

```
        title,  
        baseStyle.withFontSize(18.0)); ❶  
    }  
    ...  
}
```

❶ Возвращает копию `baseStyle`, но с измененным размером шрифта. Исходный объект `baseStyle` остается неизменным.

В примере `TextOptions`, который мы только что видели, требуются все значения параметров текста. Но если бы некоторые из них были необязательными, было бы лучше использовать либо шаблон проектирования, либо шаблон копирования при записи (оба они рассматриваются в следующем подразделе). Использование именованных аргументов вместе с необязательными параметрами также может быть хорошим подходом, но, как отмечалось в главе 5, не все языки поддерживают именованные аргументы.

ПОСТОЯННЫЕ ПЕРЕМЕННЫЕ-ЧЛЕНЫ В C++ В C++,

эквивалент пометки переменной-члена как окончательной — использование ключевого слова `const`. В коде C++ пометка переменной-члена как константы может быть плохой идеей, потому что это может вызвать проблемы с семантикой перемещения. Для более полного объяснения см.: <http://mng.bz/y9Xo>.

7.1.3. Решение: используйте шаблон проектирования для неизменности

Удаление функций-установщиков из класса и пометка переменных-членов как `final` может избежать ошибок, предотвратив мутацию класса. Но, как только что было отмечено, это также может сделать использование класса непрактичным. Если некоторые значения являются необязательными или если необходимо создать измененные версии класса, часто может возникнуть необходимость реализовать класс более универсальным способом. Для этого могут быть полезны следующие два шаблона проектирования:

- шаблон проектирования;
- шаблон копирования при записи.

Шаблон проектирования

Когда некоторые значения, с помощью которых может быть создан класс, являются необязательными, указание всех в конструкторе может стать довольно громоздким. Вместо того, чтобы делать класс изменяемым путем добавления функций настройки, часто бывает лучше использовать шаблон компоновщика ❶

❶ Форма шаблона проектирования была популяризирована Эрихом Гаммой, Ричардом Хелмом, Ральфом Джонсоном и Джоном Влиссидесом в книге *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994).

Шаблон проектирования эффективно разделяет класс на два:

- ◆ класс проектирования, который позволяет задавать значения по одному;
- ◆ неизменяемая версия класса, доступная только для чтения, созданная из шаблона.

При создании класса часто бывает так, что некоторые значения требуются, а некоторые — необязательны. Чтобы продемонстрировать, как шаблон проектирования может справиться с этим, мы предположим, что для класса `TextOptions` шрифт является обязательным значением, а размер шрифта — необязательным значением. В листинге 7.5 показан класс `TextOptions` вместе с классом проектирования для него.

Важно отметить, что класс `TextOptionsBuilder` принимает необходимое значение шрифта в качестве параметра для своего конструктора (не через функцию установки). Тогда невозможно написать код, создающий недопустимый объект. Если бы шрифт был указан с помощью функции установки, нам потребовалась бы проверка во время выполнения, чтобы убедиться, что объект действителен, что обычно уступает проверке во время компиляции (как описано в *главе 3*).

Листинг 7.5. Шаблон проектирования

```
class TextOptions { ❶
    private final Font font;
    private final Double? fontSize;

    TextOptions(Font font, Double? fontSize) {
        this.font = font;
        this.fontSize = fontSize;
    }

    Font getFont() {
        return font;
    }

    Double? getFontSize() {
        return fontSize;
    }
}

class TextOptionsBuilder { ❷
    private final Font font;
    private Double? fontSize;

    TextOptionsBuilder(Font font) {
        this.font = font;
    }
}
```

```

TextOptionsBuilder setFontSize(Double fontSize) { ❸
    this.fontSize = fontSize;
    return this; ❹
}

TextOptions build() { ❺
    return new TextOptions(font, fontSize);
}
}

```

- ❶ Класс `TextOptions` содержит только функции получения данных только для чтения.
- ❷ Конструктор принимает все необходимые значения в своем конструкторе.
- ❸ Конструктор принимает любые необязательные значения с помощью функций настройки.
- ❹ Установщик возвращает `this`, чтобы разрешить цепочку вызовов функций.
- ❺ Как только все значения указаны, вызывающий вызывает `build`, чтобы получить объект `TextOptions`.

Рис. 7.1 иллюстрирует взаимосвязь между классом `TextOptions` и классом `TextOptionsBuilder`.

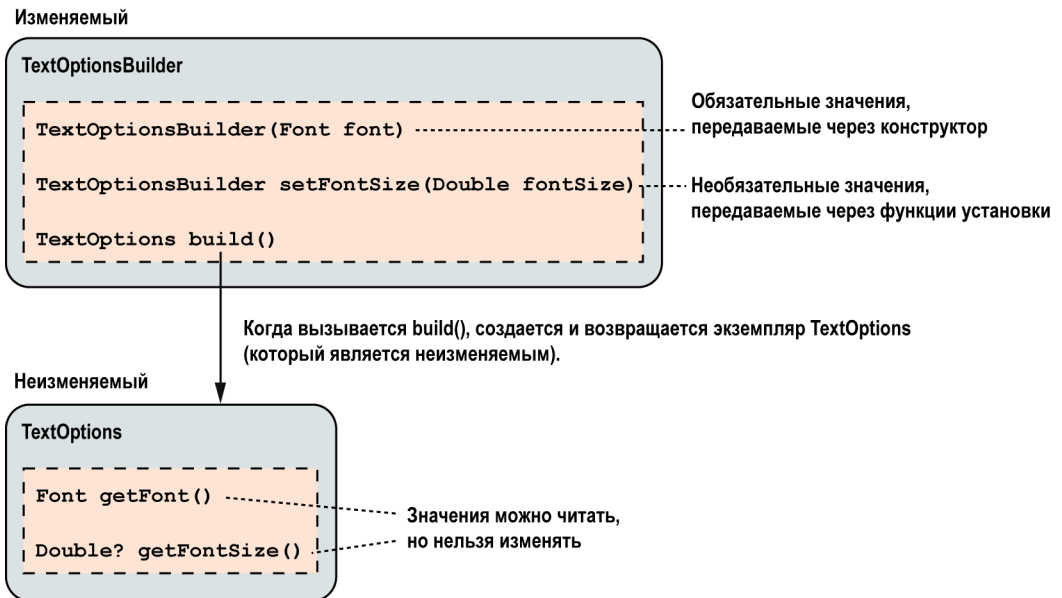


Рис. 7.1. Шаблон проектирования эффективно разделяет класс на два. Класс проектирования может быть изменен для установки значений. Затем вызов функции `build()` возвращает экземпляр неизменяемого класса, содержащего настроенные значения

В следующем фрагменте кода показан пример создания экземпляра `TextOptions`, в котором указаны как требуемое значение шрифта, так и необязательное значение размера шрифта:

```
TextOptions getDefaultTextOptions() {  
    return new TextOptionsBuilder(Font.ARIAL)  
        .setFontSize(12.0)  
        .build();  
}
```

В следующем фрагменте кода показан пример создания экземпляра `TextOptions`, в котором указано только необходимое значение шрифта:

```
TextOptions getDefaultTextOptions() {  
    return new TextOptionsBuilder(Font.ARIAL)  
        .build();  
}
```

Шаблон проектирования может быть очень полезным способом создания неизменяемого класса, когда некоторые (или все) значения являются необязательными. Если нам нужно получить слегка измененную копию экземпляра класса после проектирования, то есть способы заставить это работать с шаблоном проектирования (предоставляя функцию для создания предварительно заполненного проектировщика из класса), но это может стать немного громоздким. В следующем подразделе обсуждается альтернативный шаблон, который может сделать это намного легче.

РЕАЛИЗАЦИИ ШАБЛОНА ПРОЕКТИРОВАНИЯ

При реализации шаблона проектирования инженеры часто используют определенные методы и функции языка программирования, чтобы упростить использование и поддержку кода. Вот некоторые примеры этого:

Использование внутренних классов, чтобы сделать интервал между именами немного лучше.

Создание циклической зависимости между классом и его проектировщиком, чтобы предварительно заполненный можно было создать из класса (с помощью функции `toBuilder()`).

Создание конструктора класса закрытым, чтобы вызывающие абоненты использовали проектировщик.

Использование экземпляра проектировщика в качестве параметра конструктора для уменьшения количества шаблонов.

Приложение C (в конце книги) содержит более полный пример (на Java) реализации шаблона проектировщика с использованием всех этих методов.

Существуют также инструменты, которые могут автоматически создавать определения классов и проектировщиков. Примером этого является инструмент `AutoValue` для Java: <http://mng.bz/MgPD>.

Шаблон копирования при записи

Иногда необходимо получить модифицированную версию экземпляра класса. Примером этого является функция `renderTitle()`, которую мы видели ранее (повторяется в следующем фрагменте). Она должна сохранить все стили из `baseStyle`, но изме-

нить только размер шрифта. К сожалению, если позволить этому произойти, сделав `TextOptions` изменяемым, могут возникнуть проблемы, как мы видели ранее:

```
void renderTitle(String title, TextOptions baseStyle) {  
    baseStyle.setFont(18.0);  
    titleField.display(title, baseStyle);  
}
```

Способ поддержки этого варианта использования, а также обеспечение неизменности `TextOptions` — это шаблон копирования при записи. В листинге 7.6 показано, как выглядит класс `TextOptions` с двумя добавленными функциями копирования при записи. Функции `withFont()` и `withFontSize()` возвращают новый объект `TextOptions` с измененным только шрифтом или размером шрифта (соответственно).

В дополнение к общедоступному конструктору, который принимает требуемое значение шрифта, класс `TextOptions` также имеет частный конструктор, который принимает все значения (обязательные и необязательные). Это позволяет функциям копирования при записи создавать копию `TextOptions` только с одним измененным значением.

Листинг 7.6. Шаблон копирования при записи

```
class TextOptions {  
    private final Font font;  
    private final Double? fontSize;  
  
    TextOptions(Font font) { ❶  
        this(font, null); ❷  
    }  
  
    private TextOptions(Font font, Double? fontSize) { ❸  
        this.font = font;  
        this.fontSize = fontSize;  
    }  
  
    Font getFont() {  
        return font;  
    }  
  
    Double? getFontSize() {  
        return fontSize;  
    }  
  
    TextOptions withFont(Font newFont) {  
        return new TextOptions(newFont, fontSize); ❹  
    }  
  
    TextOptions withFontSize(Double newFontSize) {  
        return new TextOptions(font, newFontSize); ❺  
    }  
}
```


- ❶ Открытый конструктор, который принимает любые требуемые значения.
- ❷ Вызывает частный конструктор.
- ❸ Закрытый конструктор, принимающий все значения (обязательные и необязательные).
- ❹ Возвращает новый объект параметров текста с измененным шрифтом.
- ❺ Возвращает новый объект параметров текста с измененным размером шрифта.

На рис. 7.2 показано, как работает реализация класса `TextOptions` с копированием при записи.

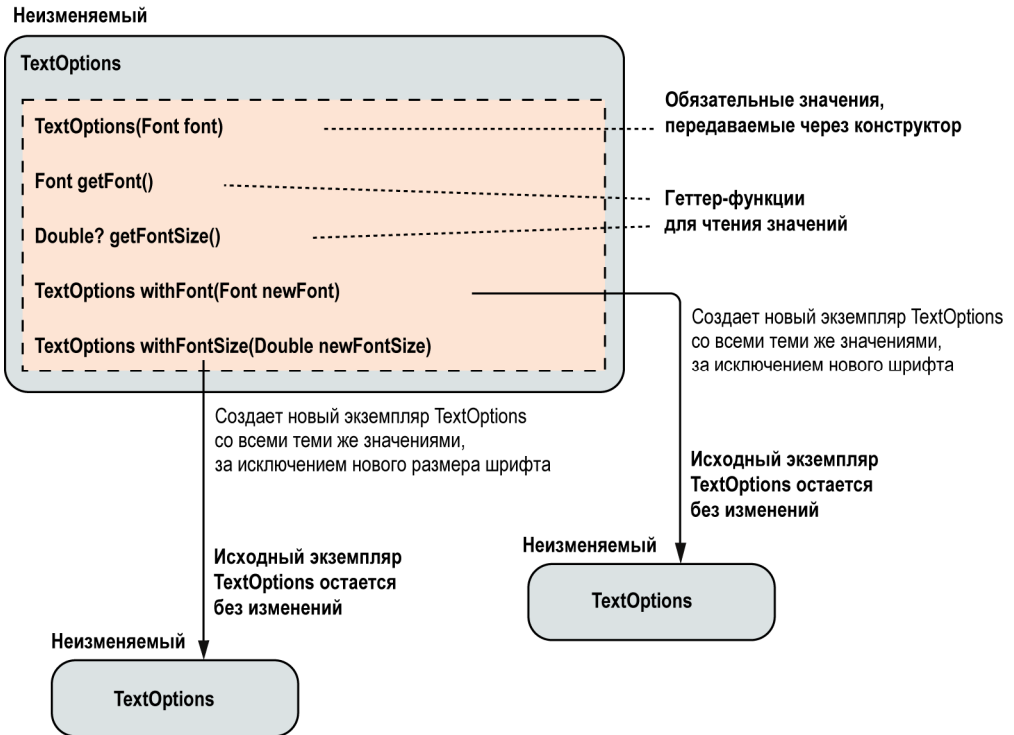


Рис. 7.2. С шаблоном копирования при записи любое изменение значения приводит к созданию нового экземпляра создаваемого класса, который содержит желаемое изменение. Существующий экземпляр класса никогда не изменяется

Экземпляр `TextOptions` можно создать с помощью конструктора и вызовов функций копирования при записи:

```
TextOptions getDefaultTextOptions() {
    return new TextOptions(Font.ARIAL)
        .withFontSize(12.0);
}
```

Когда некоторому коду, например функции `renderTitle()`, требуется измененная версия объекта `TextOptions`, он может легко получить измененную копию, не затрагивая исходный объект:

```
void renderTitle(String title, TextOptions baseStyle) {  
    titleField.display(  
        title,  
        baseStyle.withFontSize(18.0)); ❶  
}
```

❶ Новая измененная версия `baseStyle` создается путем вызова `withFontSize()`.

Создание неизменяемых классов может быть отличным способом свести к минимуму вероятность их неправильного использования. Иногда это может быть так же просто, как удалить методы настройки и предоставить значения только во время построения. В других сценариях может потребоваться использовать соответствующий шаблон проектирования. Даже при таких подходах изменяемость все еще может проникать в код менее очевидными способами; обсудим это в следующем разделе.

7.2. Подумайте о том, чтобы сделать вещи глубоко неизменяемыми

Инженеры очень часто осознают преимущества неизменяемости и следуют советам, приведенным в *разд. 7.1*. Но более тонкие способы, которыми класс может непреднамеренно стать изменяемым, могут быть упущены из виду. Распространенный способ, которым класс может случайно стать изменяемым, связан с глубокой неизменяемостью. Это может произойти, когда переменная-член имеет тип, который сам по себе является изменяемым, и другой код каким-то образом имеет к нему доступ.

7.2.1. Глубокая изменяемость может привести к неправильному использованию

Если класс `TextOptions` (из *разд. 7.1*) хранит семейство шрифтов вместо одного шрифта, он может использовать список шрифтов в качестве переменной-члена. В следующем листинге показано, как выглядит класс `TextOptions` с этим изменением.

Листинг 7.7. Сильно изменяемый класс

```
class TextOptions {  
    private final List<Font> fontFamily; ❶  
    private final Double fontSize;  
  
    TextOptions(List<Font> fontFamily, Double fontSize) {  
        this.fontFamily = fontFamily;  
    }  
}
```

```

        this.fontSize = fontSize;
    }

    List<Font> getFontFamily() {
        return fontFamily;
    }

    Double getFontSize() {
        return fontSize;
    }
}

```

❶ fontFamily — это список шрифтов.

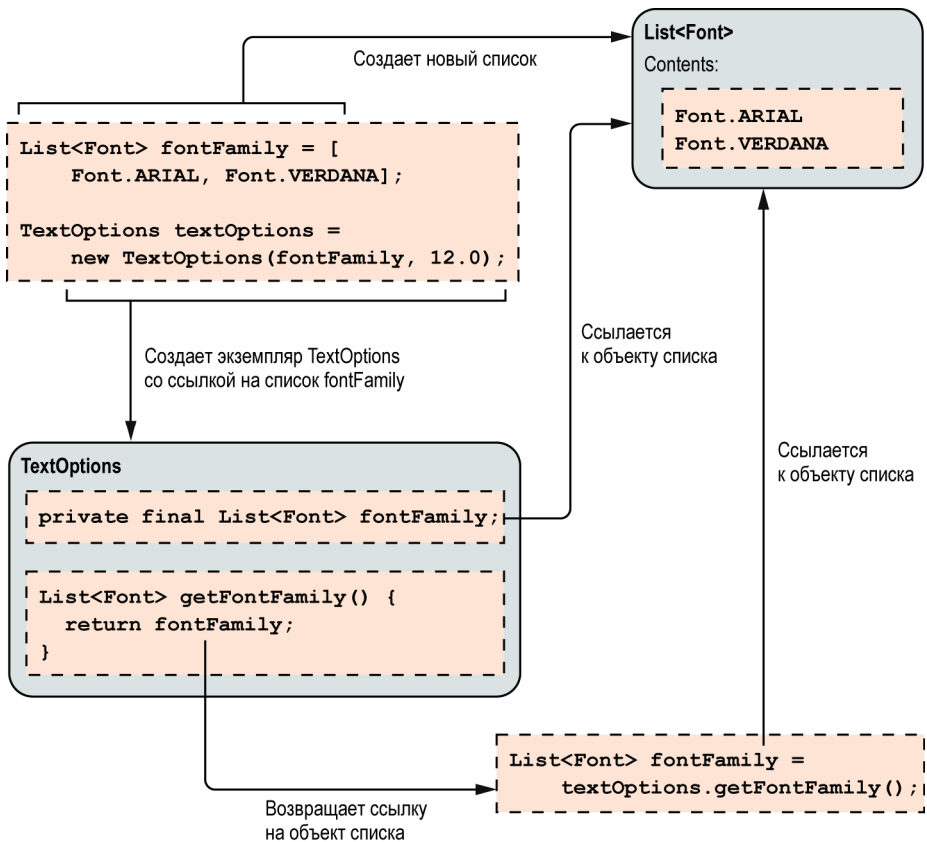


Рис. 7.3. Объекты часто хранятся по ссылке, что означает, что несколько фрагментов кода могут ссылаться на один и тот же объект. Это может быть причиной глубокой изменчивости

Это может непреднамеренно сделать класс изменяемым, поскольку класс не имеет полного контроля над списком шрифтов. Чтобы понять почему, важно помнить,

что класс `TextOptions` не содержит списка шрифтов; вместо этого он содержит ссылку на список шрифтов (как показано на рис. 7.3). Если другой фрагмент кода также имеет ссылку на этот же список шрифтов, то любые изменения, которые он вносит в список, также повлияют на класс `TextOptions`, потому что они оба ссылаются на один и тот же точный список.

Как показано на рис. 7.3, существует два сценария, в которых другой код может иметь ссылку на тот же список шрифтов, который содержит класс `TextOptions`:

- ◆ *Сценарий А.* Код, конструирующий класс `TextOptions`, может содержать ссылку на список шрифтов и вносить изменения позже.
- ◆ *Сценарий В.* Коду, который вызывает `TextOptions.getFontFamily()`, дается ссылка на список шрифтов. Он может использовать эту ссылку для изменения содержимого списка.

Сценарий А пример кода

В листинге 7.8 показан сценарий А. Код создает список шрифтов, содержащий `Font.ARIAL` и `Font.VERDANA`. Затем он создает экземпляр `TextOptions` с этим списком. После этого список очищается и добавляется `Font.COMIC_SANS`. Поскольку код в листинге и экземпляр `TextOptions` ссылаются на один и тот же список, `fontFamily` в экземпляре `TextOptions` теперь также устанавливается на `Font.COMIC_SANS`.

Листинг 7.8. Список, измененный после построения

```
...
List<Font> fontFamily = [Font.ARIAL, Font.VERDANA];

TextOptions textOptions =
    new TextOptions(fontFamily, 12.0); ❶

fontFamily.clear(); ❷
fontFamily.add(Font.COMIC_SANS);
...
```

❶ Ссылка на список `fontFamily` передается конструктору `TextOptions`.

❷ Список `fontFamily` изменен. Это тот же список, на который ссылается `textOptions`.

Сценарий В пример кода

В листинге 7.9 показан сценарий В. Экземпляр `TextOptions` создается со списком шрифтов, содержащим `Font.ARIAL` и `Font.VERDANA`. Затем некоторый код получает ссылку на этот список, вызывая `textOptions.getFontFamily()`. Затем код изменяет указанный список, очищая его и добавляя `Font.COMIC_SANS`. Это означает, что семейство шрифтов в экземпляре `TextOptions` теперь также установлено на `Font.COMIC_SANS`.

Листинг 7.9. Список, измененный вызывающей стороной

```
...
TextOptions textOptions =
    new TextOptions([Font.ARIAL, Font.VERDANA], 12.0);

List<Font> fontFamily = textOptions.getFontFamily(); ❶
fontFamily.clear(); ❷
fontFamily.add(Font.COMIC_SANS);
...
```

❶ Возвращает ссылку на тот же список, на который ссылается TextOptions.

❷ Изменяет точно такой же список, на который ссылается TextOptions.

Если сделать код изменяемым подобным образом, то это может очень легко привести к его неправильному использованию. Когда инженер вызывает такую функцию, как `textOptions.getFontFamily()`, список может передаваться несколько раз при вызове других функций или конструкторов. Так очень легко потерять из виду то, откуда он появился и можно ли его изменить. Рано или поздно какой-нибудь код может изменить список, и это вызовет странную ошибку, которую чрезвычайно трудно отследить. Гораздо лучше сделать класс полностью неизменяемым и вообще избежать подобных проблем. В следующих подразделах демонстрируются несколько способов достижения этого.

7.2.2. Решение: защитное копирование

Как мы только что видели, проблемы с глубокой изменяемостью могут возникать, когда класс содержит ссылку на объект, на который также может содержать ссылку другой фрагмент кода. Этого можно избежать, убедившись, что объект, на который ссылается класс, известен только ему и что никакой другой код никогда не сможет получить ссылку.

Это может быть достигнуто путем создания защитных копий объектов как при создании класса, так и при каждом возврате объекта из функции получения. Это не обязательно лучшее решение (объясним в этом подразделе и в следующем), но оно действительно работает и может быть простым способом сделать вещи глубоко неизменными.

В листинге 7.10 показано, как выглядел бы класс `TextOptions`, если бы он делал защитные копии списка `fontFamily`. Конструктор создает копию списка `fontFamily` и сохраняет ссылку на эту копию (решение сценария А). А функция `getFontFamily()` создает копию `fontFamily` и возвращает ссылку на эту копию (решение сценария В).

Листинг 7.10. Защитное копирование

```
class TextOptions {
    private final List<Font> fontFamily; ❶
    private final Double fontSize;
```

```
TextOptions(List<Font> fontFamily, Double fontSize) {  
    this.fontFamily = List.copyOf(fontFamily); ❷  
    this.fontSize = fontSize;  
}  
  
List<Font> getFontFamily() {  
    return List.copyOf(fontFamily); ❸  
}  
  
Double getFontSize() {  
    return fontSize;  
}  
}
```

- ❶ Копия списка `fontFamily`, на который ссылается только этот класс.
- ❷ Конструктор копирует список и сохраняет ссылку на эту копию.
- ❸ Возвращается копия списка.

Защитное копирование вещей может быть весьма эффективным, для того чтобы сделать класс глубоко неизменяемым, но у него есть некоторые очевидные недостатки:

- ◆ Копирование вещей может дорого обойтись. В случае класса `TextOptions` это, вероятно, нормально, потому что мы не ожидаем, что в семействе шрифтов будет слишком много шрифтов, а конструктор и функция `getFontFamily()`, вероятно, не будут вызываться так много раз. Но если бы в семействе шрифтов были сотни шрифтов и широко использовался класс `TextOptions`, то все это копирование могло бы стать большой проблемой для производительности.
- ◆ Часто это не защищает от изменений внутри класса. В большинстве языков программирования пометка переменной-члена как `final` (или `const`, или `readonly`) не предотвращает глубоких мутаций. Даже если список `fontFamily` помечен как окончательный, инженер может добавить код в класс, который вызывает `fontFamily.add(Font.COMIC_SANS)`. Если бы инженер случайно сделал это, код все равно скомпилировался бы и запустился, поэтому простое копирование обычно не является полной гарантией глубокой неизменяемости.

К счастью, во многих сценариях часто существует более эффективный и надежный способ сделать классы глубоко неизменяемыми. В следующем подразделе это обсуждается.

ПЕРЕДАЧА ПО ЗНАЧЕНИЮ

В таких языках, как C++, программист имеет гораздо больший контроль над тем, как объекты передаются в функцию или возвращаются. Существует различие между передачей по ссылке (или указателю) и передачей по значению. Передача по значению будет означать, что создается копия объекта, а не просто ссылка (или указатель)

на него. Это предотвращает изменение кода исходного объекта, но все равно влечет за собой недостатки копирования.

В C++ также есть идея корректности констант (упомянутая в следующем подразделе), которая часто может быть лучшим способом сохранить неизменяемость.

7.2.3. Решение: используйте неизменяемые структуры данных

Сделать вещи неизменяемыми — это широко распространенная передовая практика, и в результате было создано множество утилит, которые предоставляют неизменяемые версии общих типов или структур данных. Их преимущество в том, что после их создания никто не может изменять их содержимое. Это означает, что их можно передавать без необходимости делать защитные копии.

В зависимости от того, какой язык мы используем, некоторые подходящие варианты неизменяемой структуры данных для списка `fontFamily` могут быть следующими:

- ◆ *Java* — класс `ImmutableList` из библиотеки `Guava` (<http://mng.bz/aK09>).
- ◆ *C#* — класс `ImmutableList` из `System.Collections.Immutable` (<http://mng.bz/eMWG>).
- ◆ *Язык на основе JavaScript*. Вот несколько вариантов:
 - Класс `List` из модуля `Immutable.js` (<http://mng.bz/pJAR>).
 - Массив `JavaScript`, но с модулем `Immer`, используемым для его неизменности (<https://immerjs.github.io/immer/>).

Эти библиотеки содержат множество различных неизменяемых типов, таких, как наборы, карты и многие другие, поэтому мы часто можем найти неизменяемую версию любого стандартного типа данных, который нам может понадобиться.

В листинге 7.11 показано, как выглядит класс `TextOptions`, если его изменить для использования `ImmutableList`. Нет необходимости копировать что-либо в целях защиты, поскольку не имеет значения, есть ли в другом коде ссылка на тот же список (потому что он неизменяемый).

Листинг 7.11. Использование `ImmutableList`

```
class TextOptions {  
    private final ImmutableList<Font> fontFamily; ❶  
    private final Double fontSize;  
  
    TextOptions(ImmutableList<Font> fontFamily, Double fontSize) { ❷  
        this.fontFamily = fontFamily;  
        this.fontSize = fontSize;  
    }  
  
    ImmutableList<Font> getFontFamily() { ❸  
        return fontFamily;  
    }  
}
```

```
Double getFontSize() {  
    return fontSize;  
}  
}
```

- ❶ Даже код в этом классе не может изменять содержимое `ImmutableList`.
- ❷ Вызывающий конструктор не может изменить список позже.
- ❸ Возвращается `ImmutableList`, зная, что вызывающие абоненты не могут его изменить.

Использование неизменяемых структур данных — один из лучших способов гарантировать неизменность классов. Они избегают недостатков защитного копирования вещей и гарантируют, что даже код внутри класса не может случайно вызвать мутации.

КОРРЕКТНОСТЬ КОНСТАНТ В C++

C++ имеет довольно продвинутую поддержку неизменяемости на уровне компилятора. При определении класса инженеры могут указать, какие функции-члены не вызывают мутаций, пометив их как `const`. Если функция возвращает ссылку (или указатель) на объект, помеченный как `const`, компилятор гарантирует, что это можно использовать только для вызова неизменяемых функций-членов для этого объекта.

Это часто может свести на нет необходимость в отдельных классах для представления неизменной версии чего-либо. Более подробную информацию о корректности констант в C++ можно найти на <https://isocpp.org/wiki/faq/const-correctness>.

7.3. Избегайте слишком общих типов данных

Простые типы данных, такие, как целые числа, строки и списки, являются одними из самых фундаментальных строительных блоков кода. Они невероятно универсальны и разносторонни и могут представлять самые разные вещи. Обратной стороной очень общих и универсальных явлений является то, что они не очень описательны, а также довольно либеральны с точки зрения того, какие значения они могут содержать.

Тот факт, что что-то *может* быть представлено таким типом, как целое число или список, не обязательно означает, что это *хороший* способ представить эту вещь. Отсутствие информативности и вседозволенности может сделать код слишком простым для неправильного использования.

7.3.1. Чрезмерно общие типы могут использоваться неправильно

Для полного представления определенной части информации часто может потребоваться более одного значения. Примером этого является местоположение на 2D-карте: для его полного описания требуется значение как широты, так и долготы.

Если мы пишем какой-то код для обработки местоположений на карте, то нам, скорее всего, понадобится структура данных для представления местоположения. Структура данных должна содержать значения как для широты местоположения, так и для его долготы. Быстрым и простым способом сделать это может быть использование списка (или массива), где первое значение в списке представляет широту, а второе значение представляет долготу. Это означает, что одно местоположение будет иметь тип `List<Double>`, а список из нескольких местоположений будет иметь тип `List<Список<Двойной>>`. На рис. 7.4 показано, как это выглядит.

Одно местоположение на карте:

```
List<Double> location = [51.178889, -1.826111];
```

Коллекция локаций:

```
List<List<Double>> locations = [
    [51.178889, -1.826111],
    [53.068497, -4.076231],
    [57.291302, -4.463927]
];
```

Рис. 7.4. Очень общий тип данных, такой, как список, может использоваться для представления местоположения на карте (пара широта и долгота).

Но только потому что он может представлять это, не обязательно означает, что это хороший способ представить это

К сожалению, список — это невероятно общий тип данных, и его использование может упростить неправильное использование кода. Чтобы продемонстрировать это, листинг 7.12 содержит класс для отображения местоположений на карте. Функция `markLocationsOnMap()` принимает список местоположений и для каждого местоположения отмечает его на карте. Как видно на рис. 7.4, каждое местоположение представлено списком `List<Double>`, что означает, что набор всех местоположений, которые необходимо отметить на карте, имеет тип `List<List<Double>>`. Это становится довольно сложным, и требуется документация, чтобы объяснить, как следует использовать входной параметр.

Листинг 7.12. Чрезмерно общий тип данных

```
class LocationDisplay {
    private final DrawableMap map;
    ...

    /**
     * Отмечает расположение всех предоставленных координат
     * на карте.
```

```

*
* Принимает список списков, где внутренний список должен
* содержать ровно два значения. Первое значение должно быть
* широтой местоположения, а второе — долготой
* (оба в градусах).
*/
//Для объяснения входного параметра
// требуется полусложная документация.
void markLocationsOnMap(List<List<Double>> locations) {
    for (List<Double> location in locations) {
        map.markLocation(location[0], location[1]); ❶
    }
}
}

```

❶ Первый и второй элементы считываются из каждого внутреннего списка.

Это может показаться быстрым и простым, но есть ряд недостатков, которые облегчают неправильное использование кода, например следующие (они показаны на рис. 7.5):

- ◆ Тип `List <List <Double>>` абсолютно ничего не объясняет: если бы программист не знал документации для функции `markLocationsOnMap()`, то он не имел бы представления, что это за список и как его интерпретировать.
- ◆ Инженеру очень легко запутаться в том, где должна быть широта и долгота. Если он не прочитал документацию полностью или неправильно ее истолковал, то он мог бы поставить долготу перед широтой, что привело бы к ошибке.

Неправильная широта и долгота:

```
List<Double> местоположение = [-1.826111, 51.178889];
```

↑ Широта ↑ Долгота

Предоставлено слишком мало значений:

```
List<Double> местоположение = [51.178889];
```

Значения не указаны:

```
List<Double> местоположение = [];
```

Предоставлено слишком много значений:

```
List<Double> местоположение = [4.0, 8.0, 15.0, 16.0, 23.0, 42.0];
```

Все они неверны, но поскольку все они по-прежнему являются допустимым `List <Double>`, код компилируется нормально

Рис. 7.5. Представление чего-то особенного, например, пары широта-долгота с использованием списка двойников, может упростить неправильное использование кода

- ◆ Типовой безопасности очень мало: компилятор не может гарантировать, количество элементов, находящихся внутри списка. Вполне возможно, что некоторые из внутренних списков содержат неправильное количество значений (как показано на рис. 7.5). Если это произойдет, то код будет правильно скомпилирован и проблема будет замечена только во время выполнения (если вообще будет).

Таким образом, почти невозможно правильно вызвать функцию `markLocationsOnMap()` без детального знания (и правильного выполнения) мелкого шрифта в кодовом соглашении. Учитывая, что мелкий шрифт часто не очень надежный способ гарантировать, что другие инженеры что-то делают, это повышает вероятность того, что функция `markLocationsOnMap()` в какой-то момент будет неправильно использована, что, очевидно, может привести к ошибкам.

Парадигмы имеют обыкновение распространяться

Аналогия с полкой в *главе 1* объясняет, как выполнение одной вещи почти хакерским способом часто может заставить делать больше вещей хакерским способом. Это может легко произойти с представлением `List<Double>` местоположения на карте. Представьте, что другой инженер реализует класс для представления объекта на карте, и выходные данные этого класса необходимо передать в функцию `markLocationsOnMap()`. Они вынуждены использовать представление местоположения `List<Double>`, поэтому их код может легко взаимодействовать с функцией `markLocationsOnMap()`.

В листинге 7.13 показан код, который они могли бы написать. Функция `getLocation()` возвращает `List<Double>`, содержащий широту и долготу. Обратите внимание, что для объяснения типа возвращаемого значения функции требуется еще один кусок полусложной документации, что само по себе должно нас немного беспокоить: инструкции о том, как сохранять широту и долготу в списке, теперь закодированы и задокументированы в двух разных местах (класс `MapFeature` и класс `LocationDisplay`). Вот пример того, что существует два источника истины вместо одного источника. Это может привести к ошибкам и будет подробнее рассмотрено в *разд. 7.6*.

Листинг 7.13. Другой код принимает эту парадигму

```
class MapFeature {
    private final Double latitude;
    private final Double longitude;
    ...

    /*
     * Возвращает список с 2 элементами в нем
     * Первое значение представляет широту, а
     * второе значение представляет долготу (оба в градусах).
     */
}
```

```
List<Double> getLocation() {  
    return [latitude, longitude];  
}  
}
```

Автор исходной функции `LocationDisplay.markLocationsOnMap()`, вероятно, знает, что использование `List <Double>` — это своего рода хитрый способ представления местоположения на карте. Но они могли оправдаться тем, что это была всего лишь одна функция и, следовательно, вряд ли нанесет слишком большой ущерб кодовой базе в целом. Проблема в том, что такие слегка взломанные вещи имеют привычку распространяться, так как другим инженерам становится трудно взаимодействовать с ними, не прибегая к чему-то слегка взломанному. И это может распространяться довольно быстро и довольно далеко: если еще одному инженеру понадобится использовать класс `MapFeature` для чего-то еще, он может быть вынужден принять представление `List <Double>` и для этого другого. Прежде чем мы это узнаем, представление `List <Double>` станет повсеместным, и от него будет очень сложно избавиться.

7.3.2. Парные типы легко использовать неправильно

Многие языки программирования имеют парный тип данных. Иногда это часть стандартных библиотек, а если нет, часто где-то есть дополнительная библиотека, которая обеспечивает ее реализацию.

Суть пары в том, что она хранит два значения, которые могут быть одного или разных типов. Значения называются первым и вторым. Простая реализация парного типа данных будет выглядеть примерно так, как в следующем листинге.

Листинг 7.14. Тип данных пары

```
class Pair<A, B> { ❶  
    private final A first;  
    private final B second;  
  
    Pair(A first, B second) {  
        this.first = first;  
        this.second = second;  
    }  
  
    A getFirst() { ❷  
        return first;  
    }  
  
    B getSecond() {  
        return second;  
    }  
}
```

- ❶ Обобщения (или шаблоны) позволяют `Pair` хранить любые типы.
- ❷ Значения, называемые «первым» и «вторым».

Если бы для представления местоположения на карте использовалась `Pair <Double, Double>` (вместо `List <Double>`), то функция `markLocationsOnMap()` выглядела бы как листинг 7.15. Обратите внимание, что для объяснения входного параметра все еще требуется полусложная документация, а тип входного параметра (`List <Pair <Double, Double >>`) все еще не очень информативен.

Листинг 7.15. Использование `Pair` для определения местоположения

```
class LocationDisplay {
    private final DrawableMap map;
    ...

    /**
     * Отмечает расположение всех предоставленных координат
     * на карте.
     *
     * Принимает список пар, где каждая пара представляет
     * местоположение. Первым элементом пары должна быть
     * широта, а вторым элементом
     * пары — долгота (оба в градусах).
     */
    // Для объяснения входного параметра
    // требуется полусложная документация.
    void markLocationsOnMap(List<Pair<Double, Double>> locations) {
        for (Pair<Double, Double> location in locations) {
            map.markLocation(
                location.getFirst(),
                location.getSecond());
        }
    }
}
```

Использование `Pair<Double, Double>` вместо `List<Double>` решает некоторые проблемы, которые мы отметили в предыдущем подразделе: пара должна содержать ровно два значения, поэтому это предотвращает случайное предоставление вызывающими объектами слишком малого или слишком большого количества значений. Но другие проблемы это не решает.

- ◆ Тип `List <Pair <Double, Double >>` по-прежнему очень мало объясняет себя.
- ◆ Инженеру по-прежнему нелегко понять, в каком направлении должны быть широта и долгота.

Инженеру по-прежнему требуются подробные знания мелкого шрифта в кодовом соглашении для правильного вызова `markLocationsOnMap()`, поэтому использование `Pair <Double, Double>` по-прежнему не является отличным решением в этом сценарии.

7.3.3. Решение: используйте специальный тип

В главе 1 объясняется, как короткие пути часто действительно замедляют нас в среднесрочной и долгосрочной перспективе. Примером такого ярлыка часто может быть использование слишком общего типа данных (например, списка или пары) для очень конкретной вещи. Может показаться, что определение нового класса (или структуры) для представления чего-либо требует больших или чрезмерных усилий, но обычно это меньше, чем может показаться, и избавляет инженеров от множества головокружений и потенциальных ошибок в дальнейшем.

В случае представления двумерного местоположения карты простой способ сделать код менее легким для неправильного использования и неверной интерпретации — это определить специальный класс для представления широты и долготы. В листинге 7.16 показано, как может выглядеть этот новый класс. Это невероятно простой класс, и вряд ли на его код и тестирование уйдет больше нескольких минут.

Листинг 7.16. Класс `LatLng`

```
/**
 * Представляет широту и долготу в градусах.
 */
class LatLng {
    private final Double latitude;
    private final Double longitude;

    LatLng(Double latitude, Double longitude) {
        this.latitude = latitude;
        this.longitude = longitude;
    }

    Double getLatitude() {
        return latitude;
    }

    Double getLongitude() {
        return longitude;
    }
}
```

Используя этот новый класс `LatLng`, функция `markLocationsOnMap()` выглядит как в листинге 7.17. Теперь не требуется документации для объяснения тонкостей входного параметра, поскольку он полностью понятен, и существует хорошая безопасность типов, так, что сложно спутать широту с долготой.

Листинг 7.17. Использование `LatLng`

```
class LocationDisplay {
    private final DrawableMap map;
    ...
}
```

```
/**
 * Отмечает расположение всех предоставленных координат
 * на карте.
 */
void markLocationsOnMap(List<LatLng> locations) {
    for (LatLng location in locations) {
        map.markLocation(
            location.getLatitude(),
            location.getLongitude());
    }
}
```

Использование очень общих, готовых типов данных иногда может показаться быстрым и простым способом представления чего-либо. Но когда нам нужно представить конкретную вещь, часто бывает лучше приложить небольшое дополнительное усилие, чтобы определить для нее специальный тип. В среднесрочной и долгосрочной перспективе это обычно экономит время, потому что код становится намного более понятным и трудным для неправильного использования.

ОБЪЕКТЫ ДАННЫХ

Определение простых объектов, которые просто группируют данные, является довольно распространенной задачей, и поэтому ряд языков имеет функции (или дополнительные утилиты), которые делают это еще проще.

В Kotlin есть концепция классов данных, которая позволяет определить класс для хранения данных с помощью одной строки кода: <http://mng.bz/O15j>.

В более поздних версиях Java можно использовать записи: <https://openjdk.java.net/jeps/395>. Для более старых версий Java альтернативой является инструмент `AutoValue`: <http://mng.bz/YAaj>.

В различных языках (таких, как C++, C#, Swift и Rust) можно определять структуры, которые иногда бывает проще определить, чем класс.

В TypeScript можно определить интерфейс, а затем использовать его для обеспечения безопасности во время компиляции, для каких свойств объект должен содержать: <http://mng.bz/G6PA>.

Сторонники более традиционного подхода к объектно-ориентированному программированию иногда считают определение объектов, содержащих только данные, плохой практикой. Они утверждают, что данные и любые функции требуют, чтобы данные были инкапсулированы вместе в одном классе.

Если некоторые данные тесно связаны с определенной функциональностью, это имеет большой смысл. Но многие инженеры также понимают, что есть сценарии, в которых полезно сгруппировать некоторые данные вместе, не привязывая их к какой-либо конкретной функции. И в этом сценарии объекты, содержащие только данные, могут быть невероятно полезными.

7.4. Работа со временем

В предыдущем разделе обсуждалось, как использование чрезмерно общих типов данных для представления конкретных вещей может привести к неправильному использованию кода. Одним из конкретных примеров, который часто возникает, является представление концепций, основанных на времени.

Время может показаться простой вещью, но представление времени на самом деле довольно тонкое:

- ◆ Иногда мы ссылаемся на мгновение времени, которое может быть абсолютным, например, «02:56 UTC 21 июля 1969 года», или относительным, например, «через пять минут».
- ◆ Иногда мы имеем в виду определенное количество времени, например, «выпекать в духовке в течение 30 минут». Промежутки времени могут быть выражены в одной из любого количества различных единиц измерения, таких, как часы, секунды или миллисекунды.
- ◆ Чтобы еще больше усложнить ситуацию, у нас также есть такие понятия, как часовые пояса, переход на летнее время, високосные годы и даже високосные секунды.

Пространство для путаницы и неправильного использования кода при работе со временем огромно. В этом разделе обсуждается, как мы можем избежать путаницы и неправильного использования, используя соответствующие типы данных и языковые конструкции при работе с концепциями, основанными на времени.

7.4.1. Представление времени целыми числами может быть проблематичным

Распространенным способом представления времени является использование целого числа (или длинного целого числа), которое представляет количество секунд (или миллисекунд). Это часто используется для представления как моментов времени, так и промежутков времени:

- ◆ Момент времени часто представляется в виде количества секунд (без учета високосных секунд) с эпохи `unix` (00:00:00 UTC 1 января 1970 года).
- ◆ Промежуток времени часто представляется в виде количества секунд (или миллисекунд).

Целое число является очень общим типом и, таким образом, может упростить неправильное использование кода при использовании для представления времени подобным образом. Теперь мы рассмотрим три распространенных способа, которыми это может произойти.

Мгновение во времени или какое-то количество времени?

Рассмотрим код в листинге 7.18. Функция `sendMessage()` имеет целочисленный параметр с именем `deadline`. Документация для функции объясняет, что делает параметр `deadline` и что единицы измерения выражаются в секундах, но при этом не упоминается, что на самом деле представляет значение `deadline`. Непонятно, что следует указывать в качестве аргумента параметра `deadline` при вызове функции. Вот несколько возможных вариантов.

- ◆ Параметр представляет собой абсолютный момент времени, и мы должны указать количество секунд с эпохи `unix`.
- ◆ Параметр представляет количество времени. Когда функция вызывается, она запускает таймер, и крайний срок будет пройден, когда этот таймер достигнет указанного количества секунд.

Листинг 7.18. Мгновенное время или количество времени?

```
/**
 * @param message Сообщение для отправки
 * @param deadline Срок в секундах. Если сообщение
 * не было отправлено до истечения крайнего срока,
 * то отправка будет прервана
 * @return true, если сообщение было отправлено, и false в противном случае.
 */
// Объясняет, что делает параметр и единицы
// измерения, но не объясняет, что представляет
// собой значение.

Boolean sendMessage(String message, Int64 deadline) {
    ...
}
```

Документация, очевидно, невелика, если она оставляет столько двусмысленности. Улучшение документации было бы одним из способов улучшить это, но это привело бы к тому, что в мелкий шрифт кодового соглашения было бы добавлено больше материала. Мелкий шрифт не является особенно надежным способом предотвращения неправильного использования кода. И, учитывая, что для объяснения этого параметра уже требуется три строки документации, добавление еще большего количества для объяснения того, что представляет собой число, вероятно, не идеально.

Несовпадающие единицы измерения

Как упоминалось в начале этого раздела, существует множество различных единиц измерения времени. Наиболее распространенными единицами измерения, используемыми в коде, являются миллисекунды и секунды, но другие единицы (например, микросекунды) также используются в зависимости от контекста.

Целочисленный тип абсолютно ничего не делает, чтобы указать, в каких единицах находится значение. Мы можем указать единицы, используя имя функции, имя параметра или документацию, но это часто может сделать относительно легким неправильное использование некоторого кода.

В листинге 7.19 показаны две разные части кодовой базы. Функция `UiSettings.getMessageTimeout()` возвращает целое число, представляющее количество секунд. Функция `showMessage()` имеет параметр с именем `timeoutMs`, который представляет количество миллисекунд.

Листинг 7.19. Несоответствие в единицах времени

```
class UiSettings {  
    ...  
  
    /**  
     * @return Количество секунд, в течение которых должны  
     * отображаться сообщения пользовательского интерфейса.  
     */  
    // В этой части кода используются секунды.  
    Int64 getMessageTimeout() {  
        return 5;  
    }  
}  
  
...  
  
/**  
 * @param message Сообщение для отображения  
 * @param timeoutMs Время, в течение которого отображается  
 * сообщение в миллисекундах.  
 */  
// В этой части кода используются миллисекунды.  
void showMessage(String message, Int64 timeoutMs) {  
    ...  
}
```

Несмотря на документацию (и суффикс «Ms» в имени параметра `timeoutMs`), инженеру довольно легко сделать ошибку, когда он соединяет эти два фрагмента кода вместе. Вызов функции в следующем фрагменте не выглядит явно неправильным, но приведет к тому, что предупреждение будет отображаться в течение пяти миллисекунд, а не пяти секунд. Это означает, что сообщение исчезнет еще до того, как пользователь его заметит.

```
showMessage("Warning", uiSettings.getMessageTimeout());
```

Соответствие часовым поясам

Распространенный способ представления момента времени — это количество секунд (без учета високосных секунд) с эпохи `unix`. Это часто называют отметкой времени; она является очень точным способом определения того, когда произошло (или произойдет) какое-либо событие. Но мы, люди, часто считаем желательным говорить о событии во времени менее точным способом.

Примером этого может служить разговор о дне рождения. Если кто-то родился 2 декабря 1990 года, нас не особенно волнует точное время его рождения. Вместо этого мы просто заботимся о том, чтобы каждый год 2 декабря поздравить именинника с днем рождения, и едим немного торта в этот день.

Разница между датой и моментом времени может быть незначительной, но, если мы не будем относиться к ним по-разному, это может привести к проблемам. Рис. 7.6 иллюстрирует, как это может пойти не так. Если пользователь вводит дату (например, свой день рождения), это интерпретируется как дата и время в пределах местного часового пояса, что может привести к отображению другой даты, когда пользователь в другом часовом поясе обращается к информации.

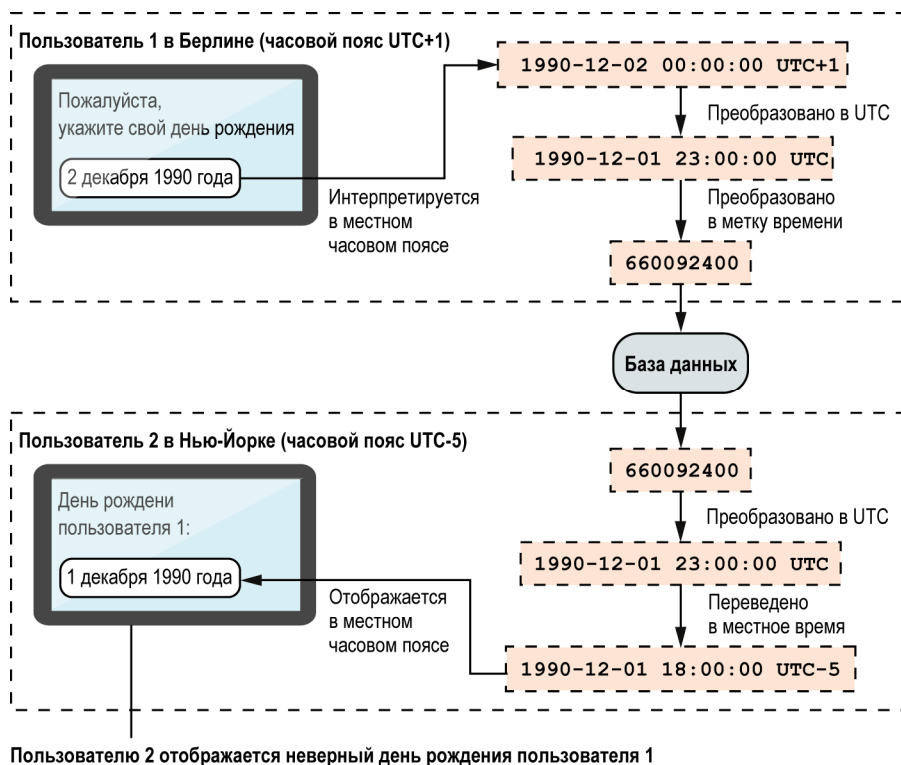


Рис. 7.6. Неправильная обработка часовых поясов может легко привести к ошибкам

Проблема, аналогичная описанной на рис. 7.6, также может возникнуть в чисто серверной логике, если серверы работают в разных местах и их системы настроены на разные часовые пояса. Например, сервер в Калифорнии может сохранять значение даты, которое в конечном итоге обрабатывает другой сервер в Европе.

Концепции, основанные на времени, такие, как моменты времени, промежутки времени и даты, могут быть сложными для работы в лучшие времена. Но мы усложняем нашу собственную жизнь и жизнь других инженеров еще больше, когда пытаемся представить их с помощью очень общего типа, такого, как целое число. Целые числа передают очень мало информации о том, что они означают или представляют, и это может сделать их очень простыми для неправильного использования. В следующем подразделе объясняется, как использование более подходящих типов может улучшить код, учитывающий время.

7.4.2. Решение: используйте соответствующие структуры данных для определения времени

Как мы видим, работа со временем является сложной и тонкой, что дает много места для путаницы. В большинстве языков программирования есть встроенные библиотеки для обработки времени, но, к сожалению, у некоторых из них есть недостатки или проблемы с разработкой, которые могут сделать их весьма подверженными ошибкам. К счастью, для большинства языков программирования с плохой встроенной поддержкой основанных на времени концепций люди создали сторонние библиотеки с открытым исходным кодом, чтобы предоставить более надежный набор утилит. Это означает, что обычно есть способ надежно обрабатывать концепции, основанные на времени, но часто бывает необходимо приложить некоторые усилия, чтобы найти лучшую библиотеку для языка, который мы используем. Вот некоторые примеры доступных опций:

- ◆ В Java можно использовать классы из пакета `java.time` (<http://mng.bz/0rPE>).
- ◆ В C# библиотека `Noda Time` предоставляет ряд утилит для надежной работы со временем (<https://nodatime.org>).
- ◆ В C++ можно использовать библиотеку `chrono` (<https://en.cppreference.com/w/cpp/header/chrono>).
- ◆ В JavaScript есть ряд сторонних библиотек на выбор. Одним из примеров является библиотека `js-joda` (<https://js-joda.github.io/js-joda/>).

Эти библиотеки значительно упрощают решение проблем, обсуждаемых в предыдущем подразделе. В следующих подразделах объясняются некоторые способы, которыми эти библиотеки могут улучшить код.

Различение мгновения времени и количества времени

Библиотеки `java.time`, `Noda Time` и `js-joda` предоставляют класс под названием `Instant` (для представления момента времени) и отдельный класс под названием `Duration`

(для представления количества времени). Точно так же библиотека `chrono` C++ предоставляет класс с именем `time_point` и отдельный класс с именем `duration`.

Использование одного из них означает, что тип параметра функции определяет, представляет ли он момент времени или количество времени. Например, функция `sendMessage()`, которую мы видели ранее, выглядела бы как листинг 7.20, если бы использовала тип `Duration`. Теперь безошибочно очевидно, что значение представляет собой количество времени, а не момент времени.

Листинг 7.20. Использование типа `Duration`

```
/**
 * @param message Сообщение для отправки
 * @param deadline Если сообщение не было отправлено
 * до истечения крайнего срока, то отправка будет прервана
 * @return true, если сообщение было отправлено, и false в противном случае
 */
Boolean sendMessage(String message, Duration deadline) { ❶
    ...
}
```

❶ Тип `Duration` дает понять, что представляет собой `deadline`.

Больше никакой путаницы в единицах измерения

Другая вещь, которой достигают такие типы, как `Instant` и `Duration`, заключается в том, что единицы измерения инкапсулированы внутри типа. Это означает, что в соглашении нет необходимости указывать мелким шрифтом, какие единицы измерения ожидаются, и становится невозможным случайно указать неверное значение. Следующий фрагмент демонстрирует, как различные заводские функции можно использовать для создания длительности с использованием разных единиц измерения. Независимо от того, какие единицы измерения используются для создания длительности, позже она может быть считана в виде количества миллисекунд. Это позволяет каждой части кода использовать те блоки, которые ей нравятся, без риска их несоответствия при взаимодействии разных частей кода.

```
Duration duration1 = Duration.ofSeconds(5);
print(duration1.toMillis()); // Output: 5000
```

```
Duration duration2 = Duration.ofMinutes(2);
print(duration2.toMillis()); // Output: 120000
```

В следующем списке показано, как можно устранить проблемы с функцией `ShowMessage()`, используя типы `Duration` вместо целых чисел для обработки количества времени, по истечении которого сообщение должно истекать.

Листинг 7.21. Единицы, инкапсулированные в типе `Duration`

```
class UiSettings {
    ...
}
```

```

/**
 * @return Продолжительность, в течение которой должны
 * отображаться сообщения пользовательского интерфейса.
 */
Duration getMessageTimeout() { ❶
    return Duration.ofSeconds(5);
}

...

/**
 * @param message Сообщение для отображения
 * @param timeout Время, в течение которого отображается сообщение.
 */
void showMessage(String message, Duration timeout) {
    ...
}

```

❶ Тип `Duration` полностью инкапсулирует единицы.

Лучшее управление часовыми поясами

В примере представления дня рождения нас не волнует часовой пояс. Но если мы хотим представить день рождения, связав его с точным моментом времени (используя временную метку), то мы вынуждены тщательно подумать о часовых поясах. К счастью, библиотеки для обработки времени часто предоставляют способ представления даты (и времени) без необходимости связывать их с точным моментом времени, подобным этому. Библиотеки `java.time`, `Noda Time` и `js-joda` предоставляют класс `LocalDateTime`, который обеспечивает именно это.

Как показано в этом разделе, справиться со временем может быть сложно, и, если мы не будем осторожны, мы можем получить код, который слишком легко использовать неправильно и внести ошибки. К счастью, мы не первые инженеры, которые столкнулись с этими проблемами, и в результате уже существует множество библиотек, которые делают работу со временем намного более надежной. Мы можем улучшить наш код, используя их.

7.5. Имейте единственные источники достоверных данных

Чаще всего код имеет дело с какими-либо данными, будь то числа, строки или потоки байтов. Данные часто могут поступать в двух формах:

- ◆ *Первичные данные* — вещи, которые необходимо внести в код. Невозможно, чтобы код мог обработать эти данные без того, чтобы ему об этом сообщили.

- ◆ *Производные данные* — вещи, которые код может вычислить на основе первичных данных.

Примером этого могут быть данные, необходимые для описания состояния банковского счета.

Есть две части первичных данных: сумма кредита и сумма дебета. Часть производных данных, которые мы, возможно, захотим узнать, — это баланс счета, представляющий собой сумму кредита минус сумму дебета.

Первичные данные обычно являются источником достоверности для программы. Значения для кредита и дебета полностью описывают состояние учетной записи и являются единственными данными, которые необходимо сохранить для ее отслеживания.

7.5.1. Вторые источники истины могут привести к неверным состояниям

В случае банковского счета величина остатка на счете полностью ограничена двумя частями первичных данных. Нет смысла говорить, что баланс составляет 10 долларов, если кредит составляет 5 долларов, а дебет — 2 доллара; это логически неверно. Это случай наличия двух источников «истины», которые не согласуются друг с другом: значения кредита и дебета указывают на одно (что баланс составляет 3 доллара США), в то время как предоставленное значение баланса говорит о чем-то другом (10 долларов США).

При написании кода, который имеет дело как с первичными, так и с производными данными, часто может возникнуть вероятность логически неправильных состояний, подобных этому. Если мы напишем код, который допускает возникновение этих логически неправильных состояний, то это может сделать неправильное использование кода слишком легким.

Листинг 7.22 демонстрирует это. Класс `UserAccount` создается со значениями для кредита, дебета и баланса счета. Как мы только что видели, баланс счета является избыточной информацией, поскольку он может быть получен из кредита и дебета, поэтому этот класс позволяет вызывающим пользователям создавать его в логически неправильном состоянии.

Листинг 7.22. Второй источник истины для баланса

```
class UserAccount {
    private final Double credit;
    private final Double debit;
    private final Double balance;

    UserAccount(Double credit, Double debit, Double balance) {
        this.credit = credit; ❶
        this.debit = debit;
```

```
        this.balance = balance;
    }

    Double getCredit() {
        return credit;
    }
    Double getDebit() {
        return debit;
    }
    Double getBalance() {
        return balance;
    }
}
```

❶ Кредит, дебет и балансодержатель — все это предоставляется конструктору.

В следующем фрагменте показан пример того, как класс `UserAccount` может быть создан в недопустимом состоянии. Инженер случайно рассчитал баланс как дебет минус кредит вместо кредит минус дебет.

```
UserAccount account =
    new UserAccount(credit, debit, debit - credit); ❶
```

❶ Баланс указан как дебет минус кредит, что неверно.

Мы надеемся, что тестирование выявит подобную ошибку, но, если этого не произойдет, это может привести к некоторым неприятным ошибкам. Банк может в итоге разослать выписки с неверными остатками средств. Или внутренние системы могут начать делать непредсказуемые вещи из-за логически неверных значений.

7.5.2. Решение: используйте первичные данные в качестве единственного источника истины

Поскольку баланс счета может быть полностью выведен из кредита и дебета, было бы гораздо лучше просто рассчитать его по мере необходимости. В листинге 7.23 показано, как выглядит класс `UserAccount` с этим изменением. Баланс больше не используется в качестве параметра конструктора и даже не сохраняется в переменной-члене. Функция `getBalance()` просто вычисляет его мгновенно при каждом вызове функции.

Листинг 7.23. Мгновенный расчет баланса

```
class UserAccount {
    private final Double credit;
    private final Double debit;

    UserAccount(Double credit, Double debit) {
        this.credit = credit;
        this.debit = debit;
    }
}
```



```
Double getCredit() {  
    return credit;  
}  
  
Double getDebit() {  
    return debit;  
}  
  
Double getBalance() { ❶  
    return credit - debit;  
}  
}
```

❶ Баланс рассчитывается по кредиту и дебету.

Пример баланса банковского счета довольно прост, и большинство инженеров, скорее всего, заметят тот факт, что предоставление баланса является избыточным, учитывая, что оно может быть получено из кредита и дебета. Но часто могут возникать более сложные ситуации, аналогичные этой, и их труднее обнаружить. Стоит потратить время на то, чтобы подумать о любых моделях данных, которые мы могли бы определять, и о том, допускают ли они существование каких-либо логически неправильных состояний.

Когда получение данных обходится дорого

Расчет остатка на счете из кредита и дебета тривиально прост и совсем не требует больших вычислительных затрат. Но иногда вычисление производного значения может быть намного дороже. Представьте, что вместо отдельных значений для кредита и дебета у нас есть список транзакций. Теперь список транзакций — это первичные данные, а общие данные по кредиту и дебету — это производные данные. Но вычисление этих производных данных сейчас довольно дорого, потому что требует обхода всего списка транзакций.

Если вычисление производного значения так дорого обходится, часто бывает полезно вычислить его лениво и кешировать результат. Ленивые вычисления означают, что мы откладываем выполнение работы до тех пор, пока она не станет абсолютно необходимой (как в реальной жизни). В листинге 7.24 показано, как выглядит класс `UserAccount` с этими изменениями. Переменные-члены `cachedCredit` и `cachedDebit` начинаются с нуля, но заполняются значениями, если и когда вызываются функции `getCredit()` и `getDebit()` соответственно.

Переменные-члены `cachedCredit` и `cachedDebit` хранят производную информацию, поэтому они фактически являются вторым источником истины. В этой ситуации это нормально, потому что этот второй источник истины полностью содержится в классе `UserAccount`, а класс и список транзакций неизменны. Это означает, что мы знаем, что переменные `cachedCredit` и `cachedDebit` будут соответствовать списку транзакций и что это никогда не изменится.

Листинг 7.24. Ленивые вычисления и кеширование

```
class UserAccount {
    private final ImmutableList<Transaction> transactions;

    private Double? cachedCredit; ❶
    private Double? cachedDebit;

    UserAccount(ImmutableList<Transaction> transactions) {
        this.transactions = transactions;
    }

    ...

    Double getCredit() { ❷
        if (cachedCredit == null) {
            cachedCredit = transactions
                .map(transaction -> transaction.getCredit())
                .sum();
        }
        return cachedCredit;
    }

    Double getDebit() { ❸
        if (cachedDebit == null) {
            cachedDebit = transactions
                .map(transaction -> transaction.getDebit())
                .sum();
        }
        return cachedDebit;
    }

    Double getBalance() { ❹
        return getCredit() - getDebit();
    }
}
```

❶ Переменные-члены для хранения кешированных значений для кредита и дебета.

❷ Кредит рассчитывается (и кешируется), если он еще не кеширован.

❸ Дебет рассчитывается (и кешируется), если он еще не кеширован.

❹ Вычисляет баланс с использованием потенциально кешированных значений.

Если класс не является неизменяемым, то все становится намного сложнее: мы должны убедиться, что переменные кеша сбрасываются в нуль всякий раз, когда класс видоизменяется. Это может быть довольно неудобно и подвержено ошибкам, так что это еще один веский аргумент в пользу того, чтобы сделать вещи неизменяемыми.

7.6. Имейте единственные источники истины для логики

Источники истины относятся не только к данным, предоставленным в код; они также применимы к логике кода. Есть много сценариев, когда что-то, что делает один фрагмент кода, должно соответствовать чему-то, что делает другой фрагмент кода. Если два фрагмента кода не соответствуют друг другу, программное обеспечение перестанет работать должным образом. Поэтому важно убедиться, что существует единый источник истины для подобной логики.

7.6.1. Множественные источники истины для логики могут привести к ошибкам

В листинге 7.25 показан класс, который можно использовать для регистрации некоторых целочисленных значений и последующего сохранения их в файл. В этом коде есть две важные детали о том, как значения хранятся в файле:

1. Каждое значение преобразуется в строковый формат (с использованием основания 10).
2. Строки для каждого значения затем соединяются вместе запятой, разделяющей их.

Листинг 7.25. Код для сериализации и сохранения значений

```
class DataLogger {  
    private final List<Int> loggedValues;  
    ...  
  
    saveValues(FileHandler file) {  
        String serializedValues = loggedValues  
            .map(value -> value.toString(Radix.BASE_10)) ❶  
            .join(","); ❷  
        file.write(serializedValues);  
    }  
}
```

❶ Значения преобразуются в строки с использованием основания 10.

❷ Значения объединяются вместе с разделением запятой.

Очень вероятно, что где-то еще есть какой-то другой код, который используется для чтения файлов и анализа из них целых чисел (процесс, обратный тому, что делает `DataLogger.saveValues()`). В листинге 7.26 показан код, который это делает. Этот код находится в совершенно другом файле (и, возможно, в другой части кодовой базы), чем класс `DataLogger`, но логика должна совпадать. В частности, чтобы

успешно проанализировать значения из содержимого файла, необходимо выполнить следующие шаги:

1. Строку необходимо разбить на список строк по символу запятой.
2. Каждую строку в списке необходимо преобразовать в целое число (с использованием системы счисления с основанием 10).

Листинг 7.26. Код для чтения и десериализации значений

```
class DataLoader {  
    ...  
  
    List<Int> loadValues(FileHandler file) {  
        return file.readAsString()  
            .split(",") ❶  
            .map(str -> Int.parse(str, Radix.BASE_10)); ❷  
    }  
}
```

❶ Содержимое файла разбито на список строк.

❷ Каждая строка преобразуется в целое число с использованием основания 10.

ОБРАБОТКА ОШИБОК

Очевидно, что существуют соображения, касающиеся обработки ошибок, когда дело доходит до записи данных в файл или чтения и анализа данных из файла. В листингах 7.25 и 7.26 они опущены для краткости, но в реальной жизни мы, вероятно, захотим рассмотреть возможность использования одного из методов, рассмотренных в главе 4, для подачи сигнала при сбое записи или чтения из файла, или когда строки не могут быть разобраны на целые числа.

В этом сценарии формат, в котором значения хранятся в файле, является важной частью логики, но есть два источника истинности того, что это за формат. Оба класса `DataLogger` и `DataLoader` независимо содержат логику, определяющую формат. Когда оба класса содержат одну и ту же логику, все работает нормально, но если один будет изменен, а другой — нет, возникнут проблемы.

Вот некоторые потенциальные изменения, которые инженер может внести в логику. Если инженер внесет одно из этих изменений в класс `DataLogger`, но не в класс `DataLoader`, все пойдет не так.

- ◆ Инженер решает, что было бы лучше хранить значения в шестнадцатеричной системе счисления вместо десятичной, чтобы сэкономить место (это означает, что файл будет содержать такие строки, как «7D» вместо «125»).
- ◆ Инженер решает, что было бы лучше разделять значения, используя новую строку вместо запятой, чтобы файлы были более удобочитаемыми.

Наличие двух источников истины для логики может легко вызвать проблемы, когда инженер изменяет один из них, не осознавая, что ему также необходимо изменить другой.

7.6.2. Решение: имейте единый источник истины

В главе 2 обсуждалось, как данный фрагмент кода обычно решает проблему высокого уровня, разбивая ее на ряд подзадач. Классы `DataLogger` и `DataLoader` решают проблему высокого уровня: регистрируют данные и загружают данные соответственно. Но при этом им обоим необходимо решить подзадачу о том, какой формат следует использовать для хранения списка сериализованных целых чисел в файле.

На рис. 7.7 показано, как классы `DataLogger` и `DataLoader` решают одну и ту же подзадачу (формат для хранения сериализованных целых чисел). Но, вместо того чтобы решить эту проблему один раз и использовать это единственное решение, каждый класс содержит свою собственную логику для ее решения.

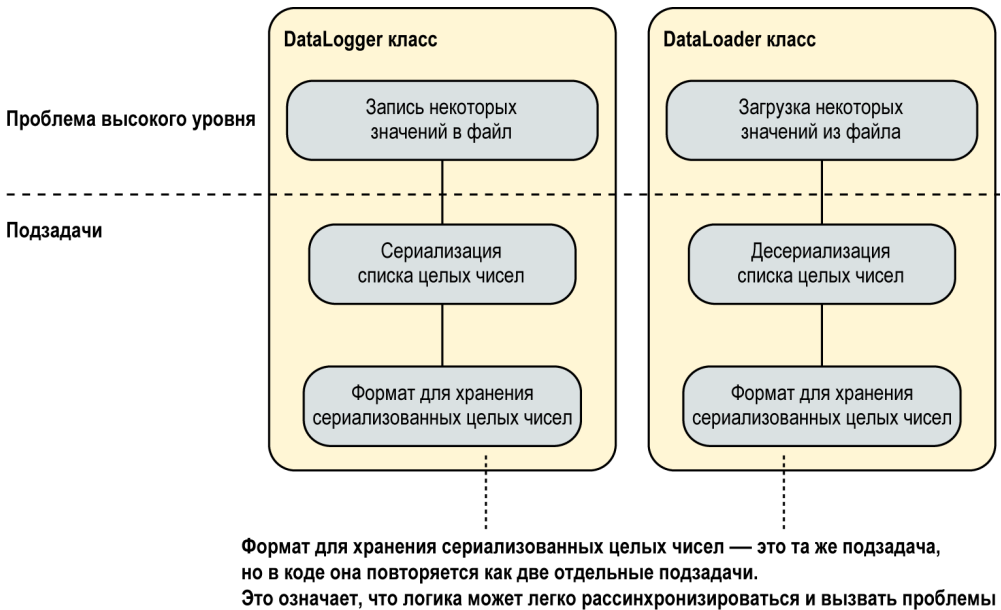


Рис. 7.7. Формат хранения сериализованных целых чисел является подзадачей, которая является общей как для классов регистратора данных, так и для загрузчика данных.

Но, вместо того чтобы использовать одно и то же решение, каждый из них содержит свою собственную логику для его решения

Мы могли бы сделать код более надежным и с меньшей вероятностью взломать, имея единый источник истинности для формата хранения сериализованных целых чисел. Мы можем добиться этого, превратив сериализацию и десериализацию списка целых чисел в единый многократно используемый уровень кода.

В листинге 7.27 показан один из способов сделать это путем определения класса `IntListFormat`, который содержит две функции: `serialize()` и `deserialize()`. Вся логика, связанная с форматом хранения сериализованных целых чисел, теперь содер-

жится в одном классе, который обеспечивает единый источник истины. Еще одна деталь, на которую следует обратить внимание, заключается в том, что разделитель запятой и основание системы счисления указываются один раз в константе, так что для них существует единственный источник истины даже внутри класса.

Листинг 7.27. Класс `IntListFormat`

```
class IntListFormat {
    private const String DELIMITER = ","; ❶
    private const Radix RADIX = Radix.BASE_10;

    String serialize(List<Int> values) {
        return values
            .map(value -> value.toString(RADIX))
            .join(DELIMITER);
    }

    List<Int> deserialize(String serialized) {
        return serialized
            .split(DELIMITER)
            .map(str -> Int.parse(str, RADIX));
    }
}
```

❶ Разделитель и основание системы счисления указываются в константах.

В листинге 7.28 показано, как теперь выглядят классы `DataLogger` и `DataLoader`, если они оба используют класс `IntListFormat` для сериализации и десериализации. Все подробности того, как сериализовать и десериализовать список целых чисел в строку и из строки, теперь обрабатываются с помощью класс `IntListFormat`.

Листинг 7.28. `DataLogger` и `DataLoader`

```
class DataLogger {
    private final List<Int> loggedValues;
    private final IntListFormat intListFormat;
    ...
    saveValues(FileHandler file) {
        file.write(intListFormat.serialize(loggedValues)); ❶
    }
}

...

class DataLoader {
    private final IntListFormat intListFormat;
    ...
}
```

```

List<Int> loadValues(FileHandler file) {
    return intListFormat.deserialize(file.readAsString());
}
}

```

❶ Класс `IntListFormat`, используемый для решения подзадачи.

На рис. 7.8 показано, как проблемы и подзадачи высокого уровня теперь распределяются между уровнями кода. Мы видим, что класс `IntListFormat` теперь предоставляет единственный источник истинности для формата хранения сериализованных целых чисел. Это почти полностью исключает риск того, что инженер может изменить формат, используемый классом `DataLogger`, и случайно забыть изменить формат, используемый классом `DataLoader`.

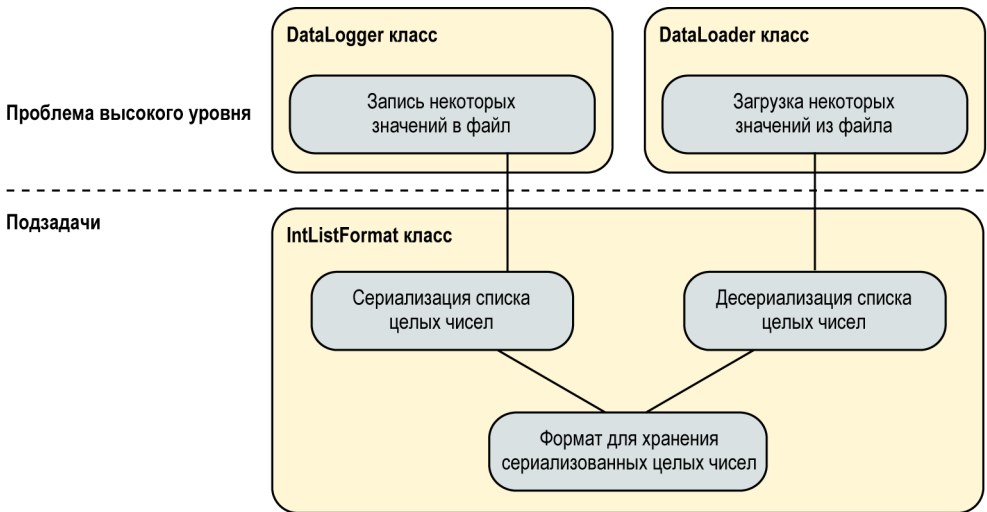


Рис. 7.8. Класс `IntListFormat` предоставляет единый источник достоверных данных о формате для хранения сериализованных целых чисел

Когда логика, выполняемая двумя разными частями кода, должна совпадать, мы не должны оставлять это на волю случая. Инженеры, работающие над одной частью кодовой базы, могут не знать о предположении, сделанном некоторым кодом в другой части кодовой базы. Мы можем сделать код намного более надежным, убедившись, что важные элементы логики имеют единый источник истины. Это почти полностью устраняет риск ошибок, вызванных тем, что разные фрагменты кода не синхронизированы друг с другом.

Итоги

Если кодом легко злоупотреблять, то велика вероятность того, что в какой-то момент им будут злоупотреблять. Это может привести к ошибкам.

Ниже приведены некоторые распространенные способы неправильного использования кода.

- ◆ Вызывающие абоненты, предоставляющие неверные входные данные.
- ◆ Побочные эффекты от других частей кода.
- ◆ Вызывающие абоненты, не вызывающие функции в нужное время или в правильном порядке.
- ◆ Связанный фрагмент кода, изменяемый таким образом, который нарушает предположение.

Часто бывает возможно спроектировать и структурировать код таким образом, чтобы его было трудно или невозможно использовать неправильно. Это может значительно снизить вероятность ошибок и сэкономить инженерам много времени в среднесрочной и долгосрочной перспективе.

8 Сделайте код модульным

ЭТА ГЛАВА ОХВАТЫВАЕТ

- ◇ Преимущества модульного кода
- ◇ Распространенные причины того, почему код менее модульный, чем в идеале
- ◇ Как сделать код более модульным

В *главе 1* обсуждалось, что требования могут меняться в течение срока службы программного обеспечения. Во многих случаях они меняются еще до выпуска программного обеспечения, поэтому нередко пишется какой-то код, а затем его приходится адаптировать всего несколько недель или месяцев спустя. Попытка точно предсказать, как будут развиваться требования, обычно является пустой тратой времени, потому что сделать это с какой-либо точностью практически невозможно. Но обычно мы можем быть уверены в том, что так или иначе требования меняться будут.

Одной из главных целей модульности является создание кода, который можно легко адаптировать и реконфигурировать без необходимости точно знать как. Ключевой целью в достижении этого является то, что различные функциональные возможности (или требования) должны соответствовать различным частям кодовой базы. Если мы достигнем этого, а позже одно из требований к программному обеспечению изменится, нам потребуется внести нетривиальные изменения только в одно место в кодовой базе, относящееся к этому требованию или элементу.

Эта глава в значительной степени основывается на идее чистых слоев абстракции (которая обсуждалась в *главе 2*). Создание модульного кода часто сводится к обеспечению того, чтобы основные детали решений подзадач были самодостаточными и не были тесно связаны друг с другом. В дополнение к тому, что код становится более адаптируемым, это также облегчает анализ программных систем. И, как мы увидим в *главах 9, 10 и 11*, это также делает код более пригодным для повторного использования и более тестируемым, поэтому модульность кода имеет ряд преимуществ.

8.1. Рассмотрите возможность использования внедрения зависимости

Классы обычно зависят от других классов. В *главе 2* показано, как код часто решает задачу высокого уровня, разбивая ее на подзадачи. В хорошо структурированном

коде каждая из этих подзадач часто решается выделенным классом. Однако не всегда существует единое решение подзадачи, поэтому может быть полезно структурировать код таким образом, чтобы можно было реконфигурировать решения подзадач. Внедрение зависимости может помочь нам достичь этого.

8.1.1. Жестко запрограммированные зависимости могут повлечь за собой проблемы

В листинге 8.1 показан некий код в классе для реализации планировщика маршрутов автомобильных поездок. Класс `RoutePlanner` зависит от экземпляра `RoadMap`. `RoadMap` — это интерфейс с потенциальным множеством различных реализаций (по одной для каждого географического региона). Но в этом примере класс `RoutePlanner` создает карту Северной Америки в своем конструкторе, что означает, что он имеет жестко запрограммированную зависимость от конкретной реализации `RoadMap`. Получается, что класс `RoutePlanner` можно использовать только для планирования поездок в Северной Америке. Это совершенно бесполезно для планирования поездок в любую другую часть света.

Листинг 8.1. Жестко запрограммированная зависимость

```
class RoutePlanner {  
    private final RoadMap roadMap; ❶  
  
    RoutePlanner() {  
        this.roadMap = new NorthAmericaRoadMap(); ❷  
    }  
    Route planRoute(LatLong startPoint, LatLong endPoint) {  
        ...  
    }  
}  
  
interface RoadMap { ❸  
    List<Road> getRoads();  
    List<Junction> getJunctions();  
}  
  
class NorthAmericaRoadMap implements RoadMap { ❹  
    ...  
    override List<Road> getRoads() { ... }  
    override List<Junction> getJunctions() { ... }  
}
```

❶ `RoutePlanner` зависит от `RoadMap`.

❷ Класс `RoutePlanner` создает `NorthAmericaRoadMap`.

❸ `RoadMap` — это интерфейс.

❹ `NorthAmericaRoadMap` — одна из потенциально многих реализаций `RoadMap`.

Добавление зависимости от конкретной реализации `RoadMap` делает невозможным перенастройку кода с помощью другой реализации. Но это не единственная проблема, связанная с жестким программированием подобной зависимости. Представьте себе, если бы класс `NorthAmericaRoadMap` был изменен и теперь требовал некоторых параметров конструктора. В листинге 8.2 показано, как теперь будет выглядеть класс `NorthAmericaRoadMap`. Он принимает два параметра в свой конструктор:

- ♦ параметр `useOnlineVersion` определяет, будет ли класс пытаться подключиться к серверу, чтобы получить самую последнюю версию карты;
- ♦ параметр `includeSeasonalRoads` определяет, включены ли на карте дороги, открытые только в определенное время года.

Листинг 8.2. Настраиваемая зависимость

```
class NorthAmericaRoadMap implements RoadMap {  
    ...  
  
    NorthAmericaRoadMap(  
        Boolean useOnlineVersion,  
        Boolean includeSeasonalRoads) { ... }  
  
    override List<Road> getRoads() { ... }  
    override List<Junction> getJunctions() { ... }  
}
```

Побочный эффект заключается в том, что класс `RoutePlanner` не может создать экземпляр `NorthAmericaRoadMap` без предоставления этих значений. Класс `RoutePlanner` обрабатывает концепции, которые характерны для класса `NorthAmericaRoadMap`: нужно ли подключаться к серверу для получения актуальной карты и включать ли сезонные дороги. Это начинает усложнять слои абстракции и может еще больше ограничить адаптивность кода. В листинге 8.3 показано, как теперь может выглядеть класс `RoutePlanner`. Теперь жестко запрограммировано, что карта будет использовать онлайн-версию и что она не будет включать сезонные дороги. Такие своего рода произвольные решения создают сценарии, в которых класс `RoutePlanner` может использоваться еще более ограниченно. Теперь он бесполезен, когда нет подключения к Интернету или когда требуются сезонные дороги.

Листинг 8.3. Настройка жестко запрограммированной зависимости

```
class RoutePlanner {  
    private const Boolean USE_ONLINE_MAP = true; ❶  
    private const Boolean INCLUDE_SEASONAL_ROADS = false;  
    private final RoadMap roadMap;  
  
    RoutePlanner() {  
        this.roadMap = new NorthAmericaRoadMap(  

```

```

        USE_ONLINE_MAP, INCLUDE_SEASONAL_ROADS);
    }
    Route planRoute(LatLong startPoint, LatLong endPoint) {
        ...
    }
}

```

❶ Аргументы конструктора для `NorthAmericaRoadMap` теперь добавлены внутрь кода.

У класса `RoutePlanner` есть одно преимущество: его очень легко создать. Его конструктор не принимает параметров, и поэтому вызывающим пользователям не нужно беспокоиться о предоставлении какой-либо конфигурации.

Недостатком, однако, является то, что класс `RoutePlanner` не очень модульный и не очень универсальный. В нем жестко запрограммировано использование карты Северной Америки, он всегда будет пытаться подключиться к онлайн-версии карты и всегда будет исключать сезонные дороги. Вероятно, это не идеально, так как у нас, скорее всего, есть пользователи за пределами Северной Америки, и мы также можем захотеть, чтобы наше приложение работало, даже когда пользователь находится в автономном режиме.

8.1.2. Решение: используйте внедрение зависимостей

Мы можем сделать класс `RoutePlanner` намного более модульным и универсальным, если позволим создавать его с использованием различных дорожных карт. Мы можем добиться этого, *внедрив* некоторую `RoadMap`, предоставив ее через параметр в конструкторе. Это устраняет необходимость в том, чтобы класс `RoutePlanner` имел жестко запрограммированную зависимость от конкретной дорожной карты и означает, что мы можем настроить его с помощью любой дорожной карты, которая нам нравится. В следующем листинге показано, как выглядит класс `RoutePlanner` с таким изменением.

Листинг 8.4. Внедрение зависимости

```

class RoutePlanner {
    private final RoadMap roadMap;

    RoutePlanner(RoadMap roadMap) { ❶
        this.roadMap = roadMap;
    }
    Route planRoute(LatLong startPoint, LatLong endPoint) {
        ...
    }
}

```

❶ `RoadMap` вводится через конструктор.

Теперь программисты могут создать экземпляр `RoutePlanner`, используя любую дорожную карту, которая им нравится. Ниже приведены некоторые примеры того, как теперь можно использовать класс `RoutePlanner`:

```
RoutePlanner europeRoutePlanner =
    new RoutePlanner(new EuropeRoadMap());
RoutePlanner northAmericaRoutePlanner =
    new RoutePlanner(new NorthAmericaRoadMap(true, false));
```

Недостатком такого внедрения `RoadMap` является то, что класс `RoutePlanner` теперь сложнее сконструировать. Теперь программист должен создать экземпляр `RoadMap`, прежде чем он сможет создать `RoutePlanner`. Мы можем упростить этот процесс, если предоставим некоторые заводские функции, которые могут использоваться другими программистами. В листинге 8.5 показано, как они могут выглядеть. Функция `createDefaultNorthAmericaRoutePlanner()` создает `RoutePlanner` с `NorthAmericaRoadMap`, используя некоторые «разумные» значения по умолчанию. Это может облегчить и ускорить создание `RoutePlanner`, который с большой долей вероятности делает то, что от него нужно программистам, но не мешает использовать `RoutePlanner` с другой дорожной картой. Поэтому для случая по умолчанию `RoutePlanner` почти так же прост в использовании, как и в предыдущем подразделе, но теперь он может быть адаптирован и к другим вариантам.

Листинг 8.5. Заводские функции

```
class RoutePlannerFactory {
    ...

    static RoutePlanner createEuropeRoutePlanner() {
        return new RoutePlanner(new EuropeRoadMap());
    }

    static RoutePlanner createDefaultNorthAmericaRoutePlanner() {
        return new RoutePlanner(
            new NorthAmericaRoadMap(true, false)); ❶
    }
}
```

❶ Создает `NorthAmericaRoadMap` с некоторыми «разумными» значениями по умолчанию.

Альтернативой написанию заводских функций вручную является использование платформы внедрения зависимостей.

Фреймворки внедрения зависимостей

Мы видели, что внедрение зависимостей может сделать классы более настраиваемыми, но у этого также может быть недостаток, заключающийся в усложнении построения классов. Мы можем использовать вручную закодированные заводские функции, чтобы облегчить это, но если таких функций получится много, процесс

будет трудоемким, и это приведет к большому количеству шаблонного кода (англ. *boilerplate code*).

Мы можем облегчить себе жизнь, используя фреймворк внедрения зависимостей, который автоматизирует большую часть работы. Существует множество различных фреймворков для внедрения зависимостей, и, какой бы язык вы ни использовали, у вас, вероятно, будет большой выбор. Поскольку их так много и они настолько специфичны для языка, мы не будем вдаваться в подробности. Главное, что следует отметить: фреймворки внедрения зависимостей могут позволить нам создавать очень модульный и универсальный код, не погружаясь в множество шаблонов заводских функций. Стоит посмотреть, какие опции доступны для языка, который вы используете, и решить, может ли это быть полезным.

В качестве предостережения следует отметить, что даже программисты, которые любят внедрение зависимостей, не всегда являются поклонниками фреймворков внедрения зависимостей. Если их использовать неосторожно, они могут привести к коду, о котором трудно рассуждать. Это может быть связано с тем, что становится трудно определить, какие части конфигурации фреймворка применяются к каким частям кода. Если вы решите использовать фреймворк внедрения зависимостей, то стоит ознакомиться с рекомендациями, чтобы избежать любых потенциальных ошибок.

8.1.3. Разрабатывайте код с учетом внедрения зависимостей

При написании кода часто бывает полезно сознательно учитывать, что мы можем захотеть использовать внедрение зависимостей. Существуют способы написания кода, которые делают практически невозможным использование внедрения зависимостей, поэтому если мы знаем, что, возможно, захотим внедрить зависимость, лучше избегать таких способов.

Чтобы продемонстрировать это, давайте рассмотрим другой способ, при помощи которого программист может реализовать `RoutePlanner` и пример карты дорог. Это показано в листинге 8.6. Класс `NorthAmericaRoadMap` теперь содержит статические функции (а не те, которые вызываются через экземпляр класса). Это означает, что класс `RoutePlanner` не зависит от экземпляра класса `NorthAmericaRoadMap`; вместо этого он напрямую зависит от статических функций `NorthAmericaRoadMap.getRoads()` и `NorthAmericaRoadMap.getJunctions()`. Это демонстрирует ту же проблему, которую мы видели в начале этого раздела: невозможно использовать класс `RoutePlanner` с чем-либо, кроме карты дорог Северной Америки. Но теперь проблема стала еще серьезнее, потому что мы не можем решить ее, изменив класс `RoutePlanner`, чтобы использовать внедрение зависимостей, даже если мы этого захотим.

Ранее, когда класс `RoutePlanner` создавал экземпляр `NorthAmericaRoadMap` в своем конструкторе, мы смогли улучшить код, используя внедрение зависимостей, чтобы вместо этого внедрить реализацию `RoadMap`. Но сейчас мы не можем этого сделать, потому что класс `RoutePlanner` не зависит от экземпляра `RoadMap`; вместо этого он напрямую зависит от статических функций в классе `NorthAmericaRoadMap`.

Листинг 8.6. Зависимость от статических функций

```
class RoutePlanner {
    Route planRoute(LatLong startPoint, LatLong endPoint) {
        ...
        List<Road> roads = NorthAmericaRoadMap.getRoads(); ❶
        List<Junction> junctions =
            NorthAmericaRoadMap.getJunctions();
        ...
    }
}

class NorthAmericaRoadMap {
    ...
    static List<Road> getRoads() { ... } ❷

    static List<Junction> getJunctions() { ... }
}
```

❶ Вызывает статические функции в классе `NorthAmericaRoadMap`.

❷ Статические функции.

Когда мы пишем код для решения подзадачи, легко предположить, что это единственное решение, которое кому-либо когда-либо понадобится. Если мы придерживаемся такого образа мыслей, часто может показаться, что самое очевидное, что нужно сделать, — это просто создать статическую функцию. Для действительно фундаментальных подзадач только с одним решением такой подход, как правило, работает нормально. Но для подзадач, которые могут потребоваться для перенастройки более высоких уровней кода, это может создавать проблемы.

СТАТИЧЕСКОЕ ЦЕПЛЯНИЕ

Чрезмерную зависимость от статических функций (или переменных) часто называют *статическим цеплянием*. Потенциальные проблемы, связанные с этим, хорошо известны и хорошо задокументированы. Это может быть особенно проблематично при модульном тестировании кода, поскольку может сделать невозможным использование имитированных реализаций (англ. *test doubles*) (см. главу 10).

В главе 2 обсуждалось, что часто бывает полезно определить интерфейс, если существует более одного потенциального решения подзадачи. В этом случае карта дорог решает подзадачу, и нетрудно представить, что код (или тесты) могут иногда требовать разных вариантов решения этой подзадачи для различных географических зон (или в различных тестовых сценариях). Из-за того что мы можем предвидеть как вероятный исход, пожалуй, будет лучше определить интерфейс для карты дорог и сделать `NorthAmericaRoadMap` классом, который его реализует (что также означает, что функции нестатические). Если мы сделаем это, то в итоге получим код, который мы видели ранее (повторяется в листинге 8.7). Это означает, что любой, кто использует `RoadMap`, может использовать внедрение зависимостей и при желании сделать свой код адаптируемым.

Листинг 8.7. Класс, экземпляр которого можно создать

```
interface RoadMap { ❶
    List<Road> getRoads();
    List<Junction> getJunctions();
}

class NorthAmericaRoadMap implements RoadMap { ❷
    ...
    override List<Road> getRoads() { ... }
    override List<Junction> getJunctions() { ... }
}
```

❶ RoadMap — это интерфейс.

❷ NorthAmericaRoadMap является одной из потенциально многих реализаций RoadMap.

Внедрение зависимостей — отличный способ сделать код модульным и обеспечить его адаптацию к различным вариантам использования. Всякий раз, когда мы имеем дело с подзадачами, которые имеют альтернативные решения, это особенно важно. Даже если это не так, внедрение зависимостей все равно может быть полезным. В *главе 9* будет показано, как это поможет нам избежать глобального состояния. А в *главе 11* будет рассмотрено, как это сделает код более тестируемым.

8.2. Предпочитайте зависимость от интерфейсов

В предыдущем разделе были продемонстрированы преимущества использования внедрения зависимостей: это обеспечивает более легкую перенастройку класса RoutePlanner. Но это возможно только потому, что все различные классы карты дорог реализуют один и тот же интерфейс RoadMap; класс RoutePlanner может от этого зависеть. Это позволяет использовать любую реализацию RoadMap, делая код значительно более модульным и адаптируемым.

Более общий метод того, как можно сделать код более модульным и простым в адаптации, таков: если мы зависим от класса, который реализует интерфейс, и этот интерфейс отражает необходимую нам функциональность, тогда обычно лучше зависеть от этого интерфейса, нежели напрямую от класса. В предыдущем разделе мы упомянули эту ситуацию, а теперь рассмотрим ее более подробно.

8.2.1. Зависимость от конкретных реализаций ограничивает адаптивность

В листинге 8.8 (из предыдущего раздела) показано, как выглядел бы класс RoutePlanner, если бы он использовал внедрение зависимостей, но напрямую зависел от класса NorthAmericaRoadMap, а не интерфейса RoadMap. Мы все еще получаем некоторые преимущества внедрения зависимостей: классу RoutePlanner не нужно ничего знать о том, как построить NorthAmericaRoadMap. Но мы упускаем одно из главных

преимуществ использования внедрения зависимостей: мы не можем использовать класс `RoutePlanner` с другими реализациями `RoadMap`.

Листинг 8.8. Зависимость от определенного класса

```
interface RoadMap { ❶
    List<Road> getRoads();
    List<Junction> getJunctions();
}

class NorthAmericaRoadMap implements RoadMap { ❷
    ...
}

class RoutePlanner {
    private final NorthAmericaRoadMap roadMap; ❸

    RoutePlanner(NorthAmericaRoadMap roadMap) {
        this.roadMap = roadMap;
    }

    Route planRoute(LatLong startPoint, LatLong endPoint) {
        ...
    }
}
```

❶ Интерфейс `RoadMap`.

❷ `NorthAmericaRoadMap` реализует интерфейс `RoadMap`.

❸ Напрямую зависит от класса `NorthAmericaRoadMap`.

В предыдущем разделе мы уже установили, насколько велика вероятность того, что у нас могут быть другие пользователи за пределами Северной Америки, поэтому использование класса `RoutePlanner`, который не работает ни в одном другом географическом местоположении, далеко от идеала. Было бы лучше, если бы код мог работать с любой картой дорог.

8.2.2. Решение: по возможности полагаться на интерфейсы

Зависимость от определенных классов реализации часто ограничивает адаптивность по сравнению с зависимостью от интерфейса. Мы можем рассматривать интерфейс как обеспечивающий слой абстракции для решения подзадачи. Определенная реализация этого интерфейса обеспечивает менее абстрактное и более ориентированное на реализацию решение подзадачи. При зависимости от более абстрактного интерфейса обычно достигаются более чистые слои абстракции и лучшая модульность.

В случае класса `RoutePlanner` это означает, что мы зависим от интерфейса `RoadMap`, а не напрямую от класса `NorthAmericaRoadMap`. Если мы это сделаем, то вернемся к тому же коду, что и в *разд. 8.1.2* (повторяется в листинге 8.9). Программисты теперь могут создать экземпляр `RoutePlanner` с любой картой дорог, которая им нравится.

Листинг 8.9. Зависимость от интерфейса

```
Class RoutePlanner {  
    private final RoadMap roadMap;  
  
    RoutePlanner(RoadMap roadMap) { ❶  
        this.roadMap = roadMap;  
    }  
  
    Route planRoute(LatLong startPoint, LatLong endPoint) {  
        ...  
    }  
}
```

❶ Зависит от интерфейса `RoadMap`.

В главе 2 рассказывалось об использовании интерфейсов и, в частности, о том, что бывает полезно определить интерфейс там, где может существовать более одного способа решения данной подзадачи. Этот совет существует именно для сценариев, подобных описанному в этом разделе. Если класс реализует интерфейс и этот интерфейс отражает поведение, которое нам нужно, то это сильный намек на то, что другие программисты могут захотеть использовать наш код с различными реализациями этого интерфейса. Зависимость от интерфейса, а не от определенного класса редко требует больших усилий, но делает код значительно более модульным и адаптируемым.

ПРИНЦИП ИНВЕРСИИ ЗАВИСИМОСТЕЙ

Идея о том, что лучше зависеть от абстракций, а не от более определенных реализаций, занимает центральное место в *принципе инверсии зависимостей*. Принцип инверсии зависимостей часто ассоциируется с Робертом К. Мартином. Это один из пяти принципов проектирования SOLID, продвигаемых Мартином (SOLID — это аббревиатура, придуманная Майклом Персом, а D означает принцип инверсии зависимостей). Смотрите по ссылке <http://mng.bz/K4Pg>. Более подробное описание этого принципа проектирования можно найти по адресу <https://stackify.com/dependency-inversion-principle/>.

8.3. Остерегайтесь наследования классов

Одной из определяющих особенностей большинства объектно-ориентированных языков программирования является то, что они позволяют одному классу наследовать от другого. Каноническим, можно сказать, примером является моделирование иерархии транспортных средств с использованием классов (рис. 8.1). Как легковые, так и грузовые автомобили являются одним типом транспортных средств, поэтому мы можем определить класс `Vehicle` для обеспечения функциональности, общей для всех транспортных средств, а затем определить классы `Car` и `Truck`, которые наследуют от класса `Vehicle`. В свою очередь, любой класс, представляющий определенный тип автомобиля, может наследовать от класса `Car`. Это формирует *иерархию классов*.

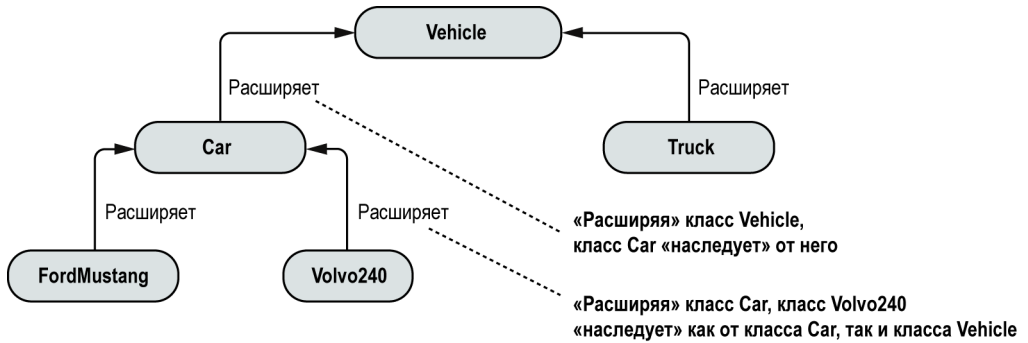


Рис. 8.1. Классы могут наследовать друг от друга, образуя иерархию классов

Наследование классов, безусловно, применяется и иногда является подходящим инструментом для достижения цели. Когда две вещи имеют подлинную взаимосвязь (например, автомобиль — это действительно транспортное средство), это может быть признаком того, что наследование целесообразно (однако см. предупреждения в *разд. 8.3.3*). Наследование — это мощный инструмент, но оно также может иметь недостатки и быть довольно неумолимым с точки зрения проблем, которые оно вызывает, поэтому обычно стоит тщательно подумать, прежде чем писать код, в котором один класс наследуется от другого.

Во многих сценариях альтернативой использованию наследования является использование композиции. Это означает, что мы komponуем один класс из другого при помощи включения его экземпляра, а не расширяя его. Это часто позволяет избежать недостатков наследования и приводит к более модульному и надежному коду. В этом разделе демонстрируются некоторые проблемы, которые может вызвать наследование, и то, как композиция может быть лучшей альтернативой.

8.3.1. Наследование классов может приводить к проблемам

Пример с транспортными средствами и автомобилями показывает, что мы подразумеваем под наследованием классов, но он слишком абстрактен, чтобы продемонстрировать различные подводные камни, с которыми обычно сталкиваются программисты, поэтому мы рассмотрим более реалистичный сценарий, в котором у программиста может возникнуть соблазн использовать наследование классов. Давайте предположим, что нас попросили написать класс, который будет считывать целые числа одно за другим из файла, содержащего значения, разделенные запятыми. Мы обдумали это задание и определили следующие подзадачи.

- ◆ Мы должны прочитать данные из файла.
- ◆ Мы должны поделить содержимое файла, разделенное запятыми, на отдельные строки.
- ◆ Мы должны пропарсить каждую из этих строк в целочисленный тип данных (англ. *integer*).

ОШИБКИ

В примере мы будем игнорировать сценарии ошибок (например, файл недоступен или содержит неверные данные). В реальной жизни мы, вероятно, захотим иметь ошибки в виду и использовать один из методов *главы 4*.

Мы заметили, что первые две из этих подзадач уже решены существующим классом, называемым `CsvFileHandler` (показан в листинге 8.10). Этот класс открывает файл и позволяет нам считывать из него строки, разделенные запятыми, одну за другой. Класс `CsvFileHandler` реализует два интерфейса: `FileValueReader` и `FileValueWriter`.

Нам нужна только функциональность, предоставляемая интерфейсом `FileValueReader`, но, как мы сейчас увидим, наследование классов не позволяет нам зависеть от такого интерфейса.

Листинг 8.10. Класс, считывающий CSV-файл

```
interface FileValueReader {
    String? getNextValue();
    void close();
}

interface FileValueWriter {
    void writeValue(String value);
    void close();
}

/**
 * Утилита для чтения и записи из/в файл, содержащая
 * значения, разделенные запятой.
 */
class CsvFileHandler
    implements FileValueReader, FileValueWriter {
    ...

    CsvFileReader(File file) { ... }
    override String? getNextValue() { ... } ❶

    override void writeValue(String value) { ... }

    override void close() { ... }
}
```

❶ Одну за другой считывает из файла строки, разделенные запятыми.

Чтобы использовать класс `CsvFileHandler` для решения нашей задачи высокого уровня, мы должны каким-то образом включить его в наш код. В листинге 8.11 показано, как мог бы выглядеть такой код, если бы мы использовали для этого наследование. Некоторые моменты, на которые следует обратить внимание в коде, заключаются в следующем.

- ◆ Класс `IntFileReader` *расширяет* класс `CsvFileHandler`, что означает, что `IntFileReader` является *подклассом* `CsvFileHandler`, или, другими словами, `CsvFileHandler` — это *суперкласс* `IntFileReader`.

- ◆ Конструктор `IntFileReader` должен создать экземпляр суперкласса `CsvFileHandler`, вызвав его конструктор. Он делает это, вызывая `super()`.
- ◆ Код внутри класса `IntFileReader` имеет доступ к функциям из суперкласса `CsvFileHandler`, как если бы они были частью класса `IntFileReader`, поэтому вызов `getNextValue()` изнутри класса `IntFileReader` вызывает эту функцию на суперклассе.

Листинг 8.11. Наследование классов

```
/**
 * Утилита для чтения целых чисел из файла по одному.
 * Файл должен содержать значения, разделенные запятыми.
 */
class IntFileReader extends CsvFileHandler { ❶
    ...
    IntFileReader(File file) {
        super(file); ❷
    }

    Int? getNextInt() {
        String? nextValue = getNextValue(); ❸
        if (nextValue == null) {
            return null;
        }
        return Int.parse(nextValue, Radix.BASE_10);
    }
}
```

❶ `IntFileReader` (подкласс) расширяет `CsvFileHandler` (суперкласс).

❷ Конструктор `IntFileReader` вызывает конструктор суперкласса.

❸ Вызывает функцию `getNextValue()` из суперкласса.

Одной из ключевых особенностей наследования является то, что подкласс наследует все функциональные возможности, предоставляемые суперклассом, поэтому любой код, содержащий экземпляр `IntFileReader`, может вызывать любую из функций, предоставляемых `CsvFileHandler`, например, функцию `close()`. Пример использования класса `IntFileReader` может выглядеть следующим образом:

```
IntFileReader reader = new IntFileReader(myFile);
Int? firstValue = reader.getNextInt();
reader.close();
```

Помимо доступа к функции `close()`, любой код с экземпляром `IntFileReader` также будет иметь доступ ко всем другим функциям из `CsvFileHandler`, как `getNextValue()` и `writeValue()`, которые, как мы сейчас увидим, могут приводить к проблемам.

Наследование может препятствовать созданию чистых слоев абстракции

Когда один класс расширяет другой класс, он наследует все функциональные возможности суперкласса. Иногда это полезно (как в случае функции `close()`), но также может привести к раскрытию большего количества функций, чем нам хотелось бы в идеале. Это может привести к запутанным слоям абстракции и утечке деталей реализации.

Чтобы продемонстрировать это, давайте рассмотрим, как будет выглядеть API (программный интерфейс приложения) для класса `IntFileReader`, если мы явно покажем как функции, которые он предоставляет, так и те, которые он наследует от суперкласса `CsvFileHandler`. В листинге 8.12 показано, как выглядит API для `IntFileReader`. Мы видим, что любые пользователи класса `IntFileReader` могут при желании вызвать функции `getNextValue()` и `writeValue()`. Для класса, который утверждает, что просто считывает целые числа из файла, это очень странные функции в общедоступном API.

Листинг 8.12. Общедоступный API `IntFileReader`

```
class IntFileReader extends CsvFileHandler {  
    ...  
    Int? getNextInt() { ... }  
  
    String? getNextValue() { ... } ❶  
    void writeValue(String value) { ... }  
    void close() { ... }  
}
```

❶ Функции, унаследованные от суперкласса.

Если API какого-либо класса предоставляет некоторую функциональность, то мы должны ожидать, что, по крайней мере, некоторые программисты будут использовать эту функциональность. Через несколько месяцев или лет мы можем обнаружить, что функции `getNextValue()` и `writeValue()` вызываются в нескольких местах через кодовую базу. Это очень затруднит изменение реализации классов `IntFileReader` в будущем. Использование `CsvFileHandler` на самом деле должно быть деталью реализации, но, используя наследование, мы случайно сделали его частью общедоступного API.

Наследование может затруднить адаптацию кода

Когда мы реализовали класс `IntFileReader`, проблема, которую нам было предложено решить, заключалась в чтении целых чисел из файла, содержащего значения, разделенные запятыми. Представьте, что теперь существует требование, согласно которому в дополнение к этому нам также необходимо предоставить способ считывания целых чисел из файлов, содержащих значения, разделенные точкой с запятой.

Мы снова замечаем, что уже существует решение для чтения строк из файла, содержащего значения, разделенные точкой с запятой. Программист уже внедрил класс под названием `SemicolonFileHandler` (показан в листинге 8.13). Этот класс реализует точно такие же интерфейсы, как и `CsvFileHandler`, `FileValueReader` и `FileValueWriter`.

Листинг 8.13. Класс, считывающий файл, разделенный точкой с запятой

```
/**
 * Утилита для чтения и записи из/в файл, содержащий
 * значения, разделенные точкой с запятой.
 */
class SemicolonFileHandler
    implements FileValueReader, FileValueWriter { ❶
    ...

    SemicolonFileHandler(File file) { ... }

    override String? getNextValue() { ... }

    override void writeValue(String value) { ... }

    override void close() { ... }
}
```

❶ Реализует те же интерфейсы, что и класс `CsvFileHandler`.

Проблема, которую нам нужно решить, почти идентична проблеме, которую мы уже решили, но с одним небольшим отличием: иногда нам нужно использовать `SemicolonFileHandler` вместо `CsvFileHandler`. Мы хотели бы надеяться, что такое небольшое изменение требований приведет лишь к небольшому изменению кода, но, к сожалению, если мы используем наследование, это может быть не так.

Требование состоит в том, что в *дополнение* к обработке содержимого файла, разделенного запятыми, нам *также* необходимо обрабатывать содержимое, разделенное точкой с запятой, поэтому мы не можем просто заставить `IntFileReader` наследовать от `SemicolonFileHandler` вместо `CsvFileHandler`, потому что это нарушило бы существующую функциональность. Наш единственный вариант — написать новую, отдельную версию класса `IntFileReader`, который наследует от `SemicolonFileHandler`. В листинге 8.14 показано, как это будет выглядеть.

Новый класс называется `SemicolonIntFileReader`, и он почти идентичен исходному классу `IntFileReader`. В подобном дублировании кода обычно нет ничего хорошего, поскольку это увеличивает расходы на обслуживание, а также вероятность ошибок (как обсуждалось в *главе 1*).

Листинг 8.14. Класс чтения файлов с точкой с запятой

```
/**
 * Утилита для чтения целых чисел из файла по одному.
```

```
* Файл должен содержать значения, разделенные точкой с запятой.
*/
class SemicolonIntFileReader extends SemicolonFileHandler {
    ...
    SemicolonIntFileReader(File file) {
        super(file);
    }

    Int? getNextInt() {
        String? nextValue = getNextValue();
        if (nextValue == null) {
            return null;
        }
        return Int.parse(nextValue, Radix.BASE_10);
    }
}
```

Тот факт, что нам приходится дублировать так много кода, особенно расстраивает, если учесть, что оба класса, `CsvFileHandler` и `SemicolonFileHandler`, реализуют интерфейс `FileValueReader`. Этот интерфейс обеспечивает слой абстракции для считывания значений без необходимости знать формат файла. Но, поскольку мы использовали наследование, мы не можем использовать этот слой абстракции. Через мгновение мы увидим, как композиция может решить эту проблему.

8.3.2. Решение: использовать композицию

Нашей первоначальной мотивацией для использования наследования было то, что мы хотели повторно использовать некоторые функции класса `CsvFileHandler`, чтобы помочь реализовать класс `IntFileReader`. Наследование — один из способов достижения этой цели, но, как мы только что видели, у него может быть несколько недостатков. Альтернативным способом повторного использования логики из `CsvFileHandler` было бы использование композиции. Это означает, что мы *компонуем* один класс из другого при помощи включения его экземпляра, а не расширяя его.

В листинге 8.15 показано, как может выглядеть наш код, если мы используем композицию. Некоторые моменты, на которые следует обратить внимание в коде, заключаются в следующем.

- ♦ Как упоминалось ранее, интерфейс `FileValueReader` отражает интересующую нас функциональность, поэтому вместо прямого использования класса `CsvFileHandler` мы используем интерфейс `FileValueReader`. Это обеспечивает более чистые слои абстракции и облегчает перенастройку кода.
- ♦ Вместо расширения класса `CsvFileHandler` класс `IntFileReader` содержит экземпляр `FileValueReader`. Так класс `IntFileReader` *компонуется* из экземпляра `FileValueReader` (поэтому мы называем этот прием *композицией*).

- ◆ Экземпляр `FileValueReader` является зависимостью, внедренной через конструктор класса `IntFileReader` (об этом говорилось в разд. 8.1).
- ◆ Поскольку класс `IntFileReader` больше не расширяет класс `CsvFileHandler`, класс `IntFileReader` больше не наследует от него метод `close()`. Чтобы позволить пользователям класса `IntFileReader` закрыть файл, мы вручную добавляем функцию `close()` в класс, которая просто вызывает функцию `close()` в экземпляре `FileValueReader`. Это называется *пересылкой*, потому что функция `IntFileReader.close()` пересылает инструкцию по закрытию файла функции `FileValueReader.close()`.

Листинг 8.15. Класс, который использует композицию

```
/**
 * Утилита для считывания целых чисел из файла по одному.
 */
class IntFileReader {
    private final FileValueReader valueReader; ❶

    IntFileReader(FileValueReader valueReader) { ❷
        this.valueReader = valueReader;
    }

    Int? getNextInt() {
        String? nextValue = valueReader.getNextValue();
        if (nextValue == null) {
            return null;
        }
        return Int.parse(nextValue, Radix.BASE_10);
    }

    void close() { ❸
        valueReader.close();
    }
}
```

- ❶ `IntFileReader` содержит экземпляр `FileValueReader`.
- ❷ В экземпляр `FileValueReader` внедряется зависимость.
- ❸ Функция `close()` пересылает в `valueReader.close()`.

ДЕЛЕГИРОВАНИЕ

В листинге 8.15 показано, как функция `IntFileReader.close()` пересылает данные в функцию `FileValueReader.close()`. Когда нам нужно переслать только одну функцию, это не вызывает особых проблем. Но могут быть сценарии, в которых необходимо переслать множество функций в komponующий класс, и может стать чрезвычайно утомительным писать все это вручную.

Это признанная проблема, и поэтому некоторые языки имеют встроенную или дополнительную поддержку делегирования, что может значительно упростить эту задачу. Как правило, это позволяет одному классу предоставлять некоторые функции из компонуемого класса контролируемым образом. Ниже приведены несколько примеров из конкретных языков:

В Kotlin есть встроенная поддержка делегирования:

<https://kotlinlang.org/docs/reference/delegation.html>.

Java Project Lombok предоставляет дополнительную аннотацию делегата, которую можно использовать для делегирования методов составному классу:

<https://projectlombok.org/features/Delegate.html>.

Использование композиции дает нам преимущества повторного использования кода, но позволяет избежать проблем с наследованием, которые мы видели ранее в этом разделе. В следующих подразделах объясняется почему.

Более чистые слои абстракции

При использовании наследования подкласс наследует и предоставляет любые функциональные возможности суперкласса. Это означает, что наш класс `IntFileReader` в конечном итоге предоставил функции из класса `CsvFileHandler`. Такое действие привело к появлению очень странного публичного API, который позволил вызывающим пользователям читать строки и даже записывать значения. Если вместо этого мы используем композицию, то ни одна из функциональностей класса `CsvFileHandler` не будет предоставлена (если только класс `IntFileReader` не предоставляет их явно с помощью пересылки или делегирования).

Чтобы продемонстрировать, насколько чище будет слой абстракции, в листинге 8.16 показано, как API класса `IntFileReader` выглядит теперь, когда мы используем композицию. Доступны только функции `getNextInt()` и `close()`, и вызывающие пользователи больше не могут читать строки или записывать значения.

Листинг 8.16. Общедоступный API `IntFileReader`

```
class IntFileReader {  
    ...  
    Int? getNextInt() { ... }  
    void close() { ... }  
}
```

Более адаптируемый код

Давайте рассмотрим изменение требований, которое мы видели ранее: нам также необходимо поддерживать файлы, в которых используются значения, разделенные точкой с запятой. В связи с тем что класс `IntFileReader` теперь зависит от интерфейса `FileValueReader`, и поскольку в него внедрена зависимость, это требование очень легко поддерживать. Класс `IntFileReader` может быть создан с помощью любой реализации `FileValueReader`, поэтому его безумно легко настроить с помощью

`CsvFileHandler` или `SemicolonFileHandler` без дублирования какого-либо кода. Мы могли бы сделать это особенно простым, предоставив две заводские функции для создания соответствующим образом настроенных экземпляров класса `IntFileReader`. В следующем листинге показано, как они могут выглядеть.

Листинг 8.17. Заводские функции

```
class IntFileReaderFactory {
    IntFileReader createCsvIntReader(File file) {
        return new IntFileReader(new CsvFileHandler(file));
    }

    IntFileReader createSemicolonIntReader(File file) {
        return new IntFileReader(new SemicolonFileHandler(file));
    }
}
```

Класс `IntFileReader` относительно прост, поэтому использование композиции может показаться не очень удачным с точки зрения адаптации кода и предотвращения дублирования, но это намеренно простой пример. В реальной жизни классы часто могут содержать больше кода и функциональности, чем здесь, поэтому затраты на наличие кода, который нельзя адаптировать даже к небольшим изменениям требований, могут стать довольно высокими.

8.3.3. Как насчет подлинных отношений «is-a»?

В начале этого раздела упоминалось, что наследование может иметь смысл, когда два класса имеют подлинные отношения `is-a`: `Ford Mustang` — это автомобиль, поэтому мы могли бы сделать так, чтобы класс `FordMustang` расширял класс `Car`. Пример того, как классы `IntFileReader` и `CsvFileHandler` определенно не соответствуют этой взаимосвязи: `IntFileReader`, по сути, не является `CsvFileHandler`, так что это довольно четкий сценарий, в котором композиция почти наверняка лучше, чем наследование. Но когда есть подлинные отношения `is-a`, может быть менее ясно, является ли наследование хорошим подходом. К сожалению, на этот вопрос нет однозначного ответа, и это будет зависеть от данного сценария и кода, над которым мы работаем. Но стоит знать, что даже при наличии подлинных отношений `is-a` наследование все равно может быть проблематичным.

Некоторые вещи, на которые следует обратить внимание, заключаются в следующем.

- ◆ *Проблема хрупкого базового класса.* Если подкласс наследует от суперкласса (иногда называемого базовым классом), и этот суперкласс позже модифицируется, иногда может произойти поломка подкласса. Это может затруднить рассуждения о том, безопасно ли определенное изменение кода.

- ♦ **Проблема с ромбовидным наследованием** (англ. *the diamond problem*). Некоторые языки поддерживают **множественное наследование** (класс, расширяющий более одного суперкласса). Это может привести к проблемам, если несколько суперклассов предоставляют версии одной и той же функции, поскольку может быть неоднозначно, от какого суперкласса должна наследоваться функция.
- ♦ **Проблемные иерархии**. Многие языки не поддерживают множественное наследование, что означает, что класс может напрямую расширять максимум один другой класс. Это называется **одиночным наследованием** и может вызвать проблемы другого типа. Представьте, что у нас есть класс под названием `Car`, на который должны расширять все классы, представляющие тот или иной тип автомобиля. В дополнение к этому представьте, что у нас также есть класс под названием `Aircraft`, который должны расширять все классы, представляющие тип воздушного судна. На рис. 8.2 показано, как выглядит наша иерархия классов. Теперь представьте, что кто-то изобретает летающий автомобиль; что же нам теперь делать? Нет разумного способа вписать это в нашу иерархию классов, потому что класс `FlyingCar` может расширять либо класс `Car`, либо класс `Aircraft`, но не оба.

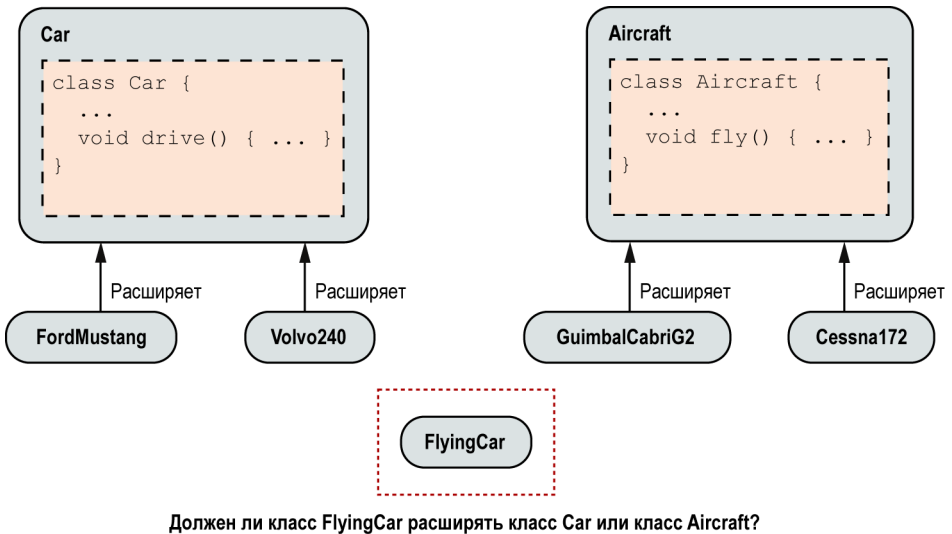


Рис. 8.2. Многие языки поддерживают только одно наследование. Это может привести к проблемам, когда класс логически принадлежит более чем к одной иерархии

Иногда возникает неизбежная необходимость в иерархии объектов. Для достижения этой цели, избегая многих недостатков наследования классов, программисты часто делают следующее:

- ♦ используют интерфейсы для определения иерархии объектов;
- ♦ используют композицию для обеспечения повторного использования кода.

На рис. 8.3 показано, как выглядят иерархии автомобилей и самолетов, если `Car` и `Aircraft` — это интерфейсы. Чтобы обеспечить повторное использование кода, общего для всех автомобилей, каждый класс автомобилей состоит из экземпляра `DrivingAction`. Аналогичным образом классы для воздушных судов состоят из экземпляра `FlyingAction`.

С наследованием классов связано достаточно подводных камней, поэтому стоит быть начеку. Многие программисты стараются полностью избегать наследования при любой возможности. К счастью, использование композиции и интерфейсов часто позволяет получить многие преимущества наследования, не страдая от недостатков.

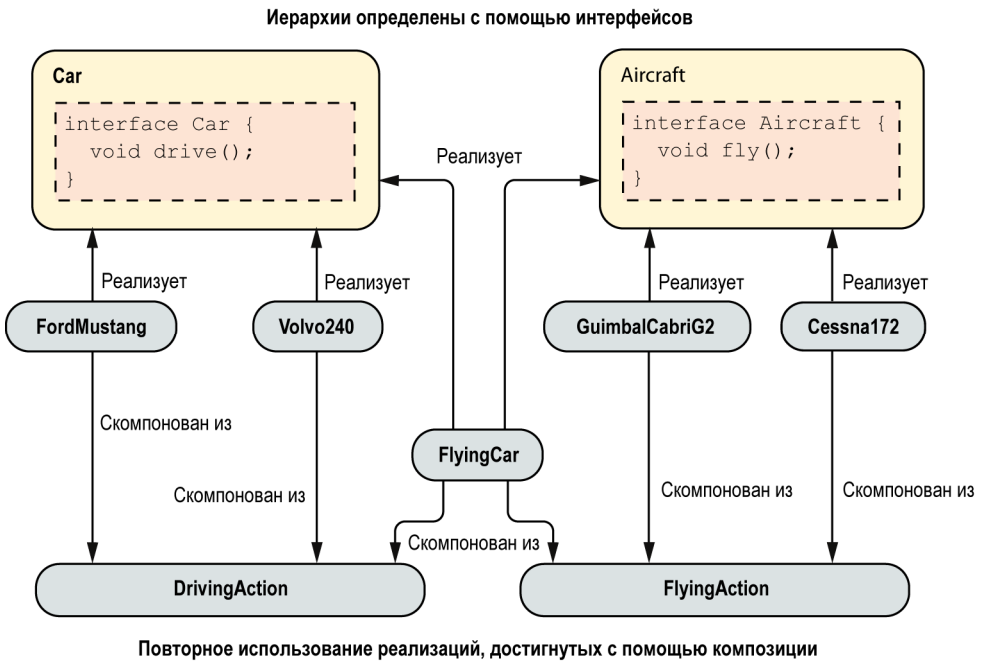


Рис. 8.3. Иерархии могут быть определены с помощью интерфейсов, в то время как повторное использование кода может быть достигнуто с помощью композиции

ПРИМЕСИ (АНГЛ. MIXIN) И ТИПАЖИ (АНГЛ. TRAITS)

Примеси и *типажи* — это элементы, поддерживаемые некоторыми языками. Они позволяют добавлять части функциональности в классы (и совместно использовать их разными классами) без необходимости использования традиционного наследования классов. Точные определения примеси и типажа и различия между ними варьируются от языка к языку. Способы их реализации также могут сильно различаться в разных языках.

Примеси и типажи помогают преодолеть некоторые проблемы, связанные с множественным наследованием и проблемными иерархиями классов. Но, как и в случае наследования классов, они все равно могут привести к коду, в котором нет чистых слоев

абстракции и который не очень легко адаптируется, поэтому обычно все равно рекомендуется проявлять осторожность и внимание при рассмотрении вопроса об использовании примесей и типажей. Ниже приведены некоторые примеры примесей и типажей для конкретных языков.

Язык программирования Dart поддерживает примеси и предоставляет практический пример того, как их можно использовать: <http://mng.bz/9NPq>. Использование примесей также относительно распространено в TypeScript: <http://mng.bz/jBI8>.

Язык программирования Rust поддерживает типаж: <http://mng.bz/Wryl>. А включение методов с использованием интерфейса по умолчанию в более поздние версии Java и C# обеспечивает возможность реализации типажей и на этих языках.

8.4. Классы должны заботиться о себе сами

Как было указано в начале этой главы, одна из ключевых целей модульности заключается в том, что изменение требований должно нуждаться в изменениях только в коде, непосредственно связанном с этим требованием. Если одна концепция полностью содержится в одном классе, то эта цель часто достигается. Любое изменение требований, связанных с данной концепцией, потребует изменения только этого одного класса.

Противоположность этому — когда одна концепция распространяется на несколько классов. Любое изменение требований, связанных с этой концепцией, потребует изменения нескольких классов. И если программист забудет изменить один из этих классов, то это может привести к ошибке. Распространенный способ того, как это может произойти, это когда класс слишком сильно заботится о деталях другого класса.

8.4.1. Чрезмерная забота о других классах может быть проблематичной

Листинг 8.18 содержит часть кода для двух отдельных классов. Первый класс представляет книгу, а второй представляет главу в книге. Класс `Book` предоставляет функцию `WordCount()` для подсчета количества слов в книге. Это включает в себя подсчет слов в каждой главе, а затем их суммирование. Класс `Book` также содержит функцию `getChapterWordCount()`, которая подсчитывает количество слов в главе. Несмотря на то что эта функция находится в пределах класса `Book`, она заботится о том, что связано только с классом `Chapter`. Это означает, что множество деталей класса `Chapter` теперь жестко закодированы в класс `Book`. Например, класс `Book` предполагает, что глава будет содержать только предисловие и список разделов.

Листинг 8.18. Классы `Book` и `Chapter`

```
class Book {  
    private final List<Chapter> chapters;  
    ...  
}
```

```

Int wordCount() {
    return chapters
        .map(getChapterWordCount)
        .sum();
}

private static Int getChapterWordCount(Chapter chapter) { ❶
    return chapter.getPrelude().wordCount() +
        chapter.getSections()
            .map(section -> section.wordCount())
            .sum();
}
}

class Chapter {
    ...
    TextBlock getPrelude() { ... }
    List<TextBlock> getSections() { ... }
}

```

❶ Эта функция заботится только о классе `Chapter`.

Размещение функции `getChapterWordCount()` в классе `Book` делает код менее модульным. Если требование изменится и теперь будет предполагаться, что в конце главы должны быть итоги, то необходимо будет обновить функцию `getChapterWordCount()`, чтобы также подсчитывать и в итогах. Это означает, что изменения в требованиях, которые касаются только глав, затронут не только класс `Chapter`. И если программист добавит поддержку итогов в класс `Chapter`, но затем забудет обновить функцию `Book.getChapterWordCount()`, логика подсчета слов в книге будет нарушена.

8.4.2. Решение: заставить классы заботиться о себе самих

Чтобы сохранить модульность кода и гарантировать, что изменения одной вещи влияют только на одну часть кода, мы должны обеспечить, чтобы классы `Book` и `Chapter` заботились только о себе самих, насколько это возможно. Класс `Book`, очевидно, нуждается в некоторых знаниях о классе `Chapter` (потому что книга содержит главы). Но, чтобы максимально открепить эти классы друг от друга, достаточно перенести в класс `Chapter` ту логику, которая заключена в функции `getChapterWordCount()`.

В листинге 8.19 показано, как теперь выглядит код. Класс `Chapter` теперь имеет внутри себя функцию под названием `wordCount()`, а класс `Book` использует эту функцию. Класс `Book` теперь заботится только о себе, а не о деталях класса `Chapter`. Если требование изменится и главы должны будут содержать итоги в конце, то необходимо будет изменить только класс `Chapter`.

Листинг 8.19. Улучшенные классы `Book` и `Chapter`

```

class Book {
    private final List<Chapter> chapters;
    ...

```

```
Int wordCount() {
    return chapters
        .map(chapter -> chapter.wordCount())
        .sum();
}

class Chapter {
    ...

    TextBlock getPrelude() { ... }

    List<TextBlock> getSections() { ... }

    Int wordCount() { ❶
        return getPrelude().wordCount() +
            getSections()
                .map(section -> section.wordCount())
                .sum();
    }
}
```

❶ Логика подсчета слов в главе полностью содержится в классе `Chapter`.

ЗАКОН ДЕМЕТРЫ

Закон Деметры (англ. *Law of Demeter*, или *LoD*, предложен Яном Холландом в 1980-х гг.) — это принцип разработки программного обеспечения, который гласит, что один объект должен делать как можно меньше предположений о содержании или структуре других объектов. В частности, принцип гласит, что объект должен взаимодействовать только с другими объектами, с которыми он непосредственно связан.

В контексте примера в этом разделе Закон Деметры будет выступать за то, чтобы класс `Book` взаимодействовал только с экземплярами класса `Chapter`, а не с какими-либо объектами внутри класса `Chapter` (такими, как `TextBlocks`, представляющими предисловие и разделы). Исходный код в листинге 8.18 явно нарушает это с помощью строки вроде `chapter.getPrelude().WordCount()`, поэтому Закон Деметры мог бы быть использован для выявления проблемы с исходным кодом в этом сценарии.

При использовании любого принципа разработки программного обеспечения важно учитывать его обоснование и преимущества, а также недостатки, которые он может обеспечить в различных сценариях. Закон Деметры в этом ничем не отличается от других принципов, поэтому если вы хотите узнать об этом больше, я бы посоветовал вам ознакомиться с различными мнениями об этом законе и сформировать собственное. Вам могут быть полезны: статья, где более подробно объясняется принцип и представлены некоторые преимущества — <http://mng.bz/8WP5>; статья, в которой представлены некоторые недостатки — <http://mng.bz/EVPX>.

Одна из ключевых целей создания модульного кода состоит в том, чтобы изменение требований приводило к изменениям только в тех частях кода, которые непосредственно связаны с этим требованием. Классам часто требуется некоторое количество знаний друг о друге, но стоит свести эту необходимость к минимуму, насколько это возможно. Это может помочь сохранить модульность кода и значительно повысить адаптивность и поддерживаемость.

8.5. Инкапсулируйте связанные данные вместе

Классы позволяют нам группировать вещи вместе. В *главе 2* говорилось о проблемах, которые могут возникнуть, когда мы пытаемся объединить слишком много вещей в один класс. Здесь нужно быть осторожным, но при этом не упускать из виду преимущества объединения вещей вместе, когда это имеет смысл.

Иногда разные фрагменты данных неизбежно связаны друг с другом, и часть кода должна передавать их вместе. В этом сценарии часто имеет смысл сгруппировать их в класс (или аналогичную структуру). Это позволяет коду работать с концепцией более высокого уровня, которую представляет группа элементов, вместо того чтобы всегда иметь дело с основными деталями. Это может сделать код более модульным и сделать изменения в требованиях более изолированными.

8.5.1. Неинкапсулированные данные могут быть трудны в обработке

Рассмотрим код в листинге 8.20. Класс `TextBox` представляет элемент в пользовательском интерфейсе, а функция `renderText()` отображает некоторый текст внутри этого элемента. Функция `renderText()` имеет четыре параметра, которые связаны с тем, какой стиль должен иметь текст.

Листинг 8.20. Класс и функция для рендеринга текста

```
class TextBox {  
    ...  
  
    void renderText(  
        String text,  
        Font font,  
        Double fontSize,  
        Double lineHeight,  
        Color textColor) {  
        ...  
    }  
}
```

Класс `TextBox`, скорее всего, является относительно низкоуровневым фрагментом кода, поэтому функция `renderText()`, вероятно, вызывается функцией, которая, в свою очередь, вызывается другой функцией, и т. д. Это означает, что значения, связанные с оформлением текста, потенциально должны передаваться от одной функции к другой несколько раз. В листинге 8.21 показана упрощенная версия этого. В этом сценарии класс `UiSettings` является источником значений стиля текста. Функция `UserInterface.DisplayMessage()` считывает эти значения из пользовательских настроек и передает их функции `renderText()`.

Функция `DisplayMessage()` на самом деле не заботится ни о каких особенностях оформления текста. Все, что ее волнует, — это тот факт, что `UiSettings` предоставляет некоторый стиль, и `renderText()` нуждается в этом. Но поскольку параметры стилизации текста не инкапсулированы вместе, функция `DisplayMessage()` вынуждена иметь основные сведения о деталях стилизации текста.

Листинг 8.21. Классы `UiSettings` и `UserInterface`

```
class UiSettings {
    ...

    Font getFont() { ... }
    Double getFontSize() { ... }
    Double getLineHeight() { ... }
    Color getTextColor() { ... }
}

class UserInterface {
    private final TextBox messageBox;
    private final UiSettings uiSettings;

    void displayMessage(String message) {
        messageBox.renderText(
            message,
            uiSettings.getFont(), ❶
            uiSettings.getFontSize(),
            uiSettings.getLineHeight(),
            uiSettings.getTextColor());
    }
}
```

❶ Функция `DisplayMessage()` содержит основные сведения о деталях стилизации текста.

В этом сценарии функция `DisplayMessage()` немного похожа на курьера, доставляющего некоторую информацию из класса `UiSettings` в функцию `renderText()`. В реальной жизни курьеру обычно бывает все равно, что именно находится внутри посылки. Если вы отправляете коробку конфет другу, курьеру не нужно знать, отправляете ли вы карамельные трюфели или пралине. Но в этом сценарии класс `DisplayMessage()` должен точно знать, что он передает.

Если требование изменится и теперь необходимо определить стиль шрифта (например, курсив) для функции `renderText()`, нам придется изменить функцию `DisplayMessage()` для передачи этой новой информации. Как мы видели ранее, одной из целей модульности является обеспечение того, чтобы изменение требования затрагивало только части кода, непосредственно связанные с этим требованием. В этом случае только классы `UiSettings` и `TextBox` на самом деле имеют дело со стилизацией текста, поэтому то, что функция `DisplayMessage()` также нуждается в изменении, для нас неидеально.

8.5.2. Решение: сгруппировать связанные данные в объекты или классы

В этом сценарии шрифт, размер шрифта, высота строки и цвет текста неразрывно связаны друг с другом: чтобы знать, как стилизовать текст, нам нужно знать всё это. Учитывая, что эти настройки связаны таким образом, имеет смысл инкапсулировать их вместе в один объект, который затем можно передавать по кругу. В следующем листинге показан класс, называемый `TextOptions`, который делает именно это.

Листинг 8.22. Инкапсулирующий класс `TextOptions`

```
class TextOptions {
    private final Font font;
    private final Double fontSize;
    private final Double lineHeight;
    private final Color textColor;

    TextOptions(Font font, Double fontSize,
                Double lineHeight, Color textColor) {
        this.font = font;
        this.fontSize = fontSize;
        this.lineHeight = lineHeight;
        this.textColor = textColor;
    }

    Font getFont() { return font; }
    Double getFontSize() { return fontSize; }
    Double getLineHeight() { return lineHeight; }
    Color getTextColor() { return textColor; }
}
```

АЛЬТЕРНАТИВА ОБЪЕКТАМ ДАННЫХ

Инкапсуляция данных вместе (как это делает класс `TextOptions`) может быть еще одним вариантом использования объектов данных, которые обсуждались в предыдущей главе в *разд. 7.3.3*.

Как было сказано в предыдущей главе, сторонники более традиционного подхода к объектно-ориентированному программированию иногда считают объекты только для данных плохой практикой, поэтому стоит отметить, что другим подходом в этом сценарии может быть объединение информации о стиле и логики, реализующей стиль текста, в один класс. Если бы мы сделали это, то, скорее всего, получили бы класс `TextStyler`, который мы передаем. Однако общий принцип инкапсуляции связанных данных все еще применим.

Используя класс `TextOptions`, теперь мы можем инкапсулировать информацию о стилизации текста вместе и вместо этого просто передавать экземпляр `TextOptions`. В листинге 8.23 показано, как теперь выглядит код из предыдущего подраздела. Функция `DisplayMessage()` теперь не знает особенностей оформления текста. Если бы

нам действительно нужно было добавить стиль шрифта, то нам не нужно было бы вносить какие-либо изменения в функцию `DisplayMessage()`. Она стала больше похожа на хорошего курьера: он старательно доставляет посылку, не слишком заботясь о том, что внутри.

Листинг 8.23. Передача инкапсулированного объекта

```
class UiSettings {
    ...

    TextOptions getTextStyle() { ... }
}

class UserInterface {
    private final TextBox messageBox;
    private final UiSettings uiSettings;

    void displayMessage(String message) {
        messageBox.renderText(
            message, uiSettings.getTextStyle()); ❶
    }
}

class TextBox {
    ...

    void renderText(String text, TextOptions textStyle) {
        ...
    }
}
```

❶ Функция `DisplayMessage()` не имеет конкретных знаний о стилизации текста.

Решение о том, когда инкапсулировать вещи вместе, может потребовать некоторого размышления. В *главе 2* показано, как могут возникать проблемы, когда слишком много концепций объединено в один класс, поэтому стоит быть осторожным. Но когда разные фрагменты данных неизбежно связаны друг с другом и нет практического сценария, в котором кому-то понадобились бы некоторые, но не все, фрагменты данных, тогда обычно имеет смысл инкапсулировать их вместе.

8.6. Остерегайтесь утечки деталей реализации в типах возвращаемых значений

В *главе 2* была подчеркнута важность создания слоев уровней абстракции. Чтобы получить чистые слои абстракции, необходимо убедиться, что не происходит утечки деталей реализации. Если такая утечка произойдет, это может раскрыть информацию о нижних уровнях кода и сильно затруднить изменение или перенастройку в будущем. Одним из наиболее распространенных способов утечки деталей реализации в коде является возврат типа, который тесно связан с этой деталью.

8.6.1. Утечка деталей реализации в типе возвращаемого значения может быть проблематичной

В листинге 8.24 показан некоторый код, который можно использовать для поиска фотографии профиля для какого-либо пользователя. `ProfilePictureService` реализован с использованием `HttpFetcher` для извлечения данных о фотографии профиля с сервера. Тот факт, что используется `HttpFetcher`, является деталью реализации, и поэтому любой программист, использующий класс `ProfilePictureService`, в идеале не должен беспокоиться об этой информации.

Несмотря на то что класс `ProfilePictureService` напрямую не допускает утечки того факта, что используется `HttpFetcher`, он, к сожалению, допускает эту утечку косвенно через тип возвращаемого значения. Функция `getProfilePicture()` возвращает экземпляр `ProfilePictureResult`. Если мы заглянем внутрь класса `ProfilePictureResult`, то увидим, что он использует `HttpResponse.Status`, указывающий, был ли запрос успешным, а также `HttpResponse.Payload` для вложения данных изображения для фотографии профиля. Оба они допускают утечку того факта, что `ProfilePictureService` использует HTTP-соединение для извлечения фотографии профиля.

Листинг 8.24. Детали реализации в типе возвращаемого значения

```
class ProfilePictureService {
    private final HttpFetcher httpFetcher; ❶
    ...
    ProfilePictureResult getProfilePicture(Int64 userId) { ... } ❷
}

class ProfilePictureResult {
    ...

    /**
     * Indicates if the request for the profile picture was
     * successful or not.
     */
    HttpResponse.Status getStatus() { ... } ❸
    /**
     * The image data for the profile picture if it was successfully
     * found.
     */
    HttpResponse.Payload? getImageData() { ... }
}
```

❶ `ProfilePictureService` реализован с использованием `HttpFetcher`.

❷ Возвращает экземпляр `ProfilePictureResult`.

❸ Типы данных, характерные для HTTP-ответа.

В главе 2 подчеркивалась важность недопущения утечки деталей реализации, поэтому с этой точки зрения мы, вероятно, сразу увидим, что этот код далек от идеала. Но, чтобы действительно понять, насколько вредным может быть этот код, давайте рассмотрим следующие последствия.

- ◆ Любой программист, использующий класс `ProfilePictureService`, должен иметь дело с рядом концепций, характерных для `HttpResponse`. Чтобы понять, был ли запрос фотографии профиля успешным и почему он мог завершиться неудачей, программист должен интерпретировать перечисляемое значение `HttpResponse.Status`. Это требует знания о кодах состояния HTTP, а также о том, какие конкретные коды состояния HTTP на самом деле может использовать сервер. Программист может догадаться, что ему нужно проверить наличие `STATUS_200` (что указывает на успех) и `STATUS_404` (что указывает на то, что ресурс не найден). Но как насчет других 50 с лишним кодов состояния HTTP, которые иногда используются?
- ◆ Очень сложно изменить реализацию `ProfilePictureService`. Любой код, вызывающий `ProfilePictureService.getProfilePicture()`, должен иметь дело с типами `HttpResponse.Status` и `HttpResponse.Payload` для понимания ответа, поэтому уровни кода, построенные поверх `ProfilePictureService`, полагаются на тот факт, что он возвращает типы, характерные для `HttpResponse`. Представьте, что требования изменились, а это означает, что наше приложение должно иметь возможность извлекать фотографии профиля с помощью, например, протокола `WebSocket`. Поскольку так много кода полагается на использование типов, характерных для `HttpResponse`, для поддержки любых изменений в требованиях, подобных этому, потребуется много изменений кода в разных местах.

Было бы лучше, если бы `ProfilePictureService` не допускал утечки таких деталей реализации, как эти. Было бы лучше, если бы он возвращал тип, соответствующий слою абстракции, который он призван обеспечить.

8.6.2. Решение: вернуть тип, соответствующий слою абстракции

Задача, которую решает класс `ProfilePictureService`, заключается в получении фотографии профиля пользователя. Это определяет слой абстракции, который в идеале должен быть обеспечен этим классом, и любые возвращаемые типы должны отражать это. Мы должны постараться свести к минимуму количество концепций, которые мы раскрываем программистам, использующим этот класс. В этом сценарии минимальный набор концепций, которые нам необходимо раскрыть, заключается в следующем.

- ◆ Запрос может быть выполнен успешно или может завершиться неудачей по одной из следующих причин:
 - пользователь не существует;
 - произошла какая-то кратковременная ошибка (например, сервер недоступен).
- ◆ Байты данных, которые представляют фотографию профиля.

В листинге 8.25 показано, как мы могли бы реализовать сервис `ProfilePictureService` и классы `ProfilePictureResult`, если мы попытаемся ограничить количество концепций, которые мы раскрываем, этим минимальным набором. Важные изменения, которые мы внесли, заключаются в следующем.

- ◆ Вместо использования перечисления `HttpResponse.Status` мы определили пользовательское перечисление, которое содержит только набор статусов, о которых действительно должен заботиться программист, использующий этот класс.
- ◆ Вместо возврата `HttpResponse.Payload` мы возвращаем список байтов.

Листинг 8.25. Тип возвращаемого значения соответствует слою абстракции

```
class ProfilePictureService {
    private final HttpFetcher httpFetcher;
    ...

    ProfilePictureResult getProfilePicture(Int64 userId) { ... }
}

class ProfilePictureResult {
    ...

    enum Status { ❶
        SUCCESS,
        USER_DOES_NOT_EXIST,
        OTHER_ERROR,
    }

    /**
     * Указывает, был ли запрос изображения фотографии профиля
     * успешным.
     */
    Status getStatus() { ... } ❷

    /**
     * Данные изображения для фотографии профиля, если оно было
     * успешно найдено.
     */
    List<Byte>? getImageData() { ... } ❸
}
```

❶ Пользовательское перечисление для определения только тех статусов, которые нам требуются.

❷ Возвращает пользовательское перечисление.

❸ Возвращает список байтов.

ПЕРЕЧИСЛЕНИЯ

Как упоминалось в *главе 6*, перечисления вызывают некоторые разногласия среди программистов. Некоторым нравится их использовать, а другие считают полиморфизм лучшим подходом (создание разных классов, реализующих общий интерфейс).

Независимо от того нравятся вам перечисления или нет, ключевым моментом здесь является использование типа, соответствующего слою абстракции (будь то перечисление или класс).

В целом повторное использование кода — это хорошо, поэтому на первый взгляд может показаться хорошей идеей повторно использовать типы `HttpResponse.Status` и `HttpResponse.Payload` в классе `ProfilePictureResult`. Но если хорошо подумать, станет понятно, что эти типы не соответствуют слою абстракции, который мы предоставляем, поэтому определение наших собственных типов, которые отражают минимальный набор концепций, и их использование приводит к более чистым слоям абстракции и более модульному коду.

8.7. Остерегайтесь утечки деталей реализации в исключениях

В предыдущем разделе было показано, как утечка деталей реализации в типах возвращаемых значений может вызвать проблемы. Тип возвращаемого значения находится в безошибочно очевидной части соглашения по написанию кода, поэтому (хотя это и проблематично) обычно довольно легко определить, когда происходит утечка, что может облегчить ее предотвращение. Еще один распространенный способ утечки деталей реализации — это типы исключений, которые мы создаем.

В частности, в *главе 4* обсуждалось, что непроверенные исключения указаны мелким шрифтом в кодовом соглашении, а иногда не прописаны вообще, поэтому если мы используем непроверенные исключения для ошибок, которые вызывающие пользователи могут захотеть исправить, то утечка деталей реализации в них может быть особенно проблематичной.

8.7.1. Утечка деталей реализации в исключениях может быть проблематичной

Одной из определяющих особенностей непроверенных исключений является то, что компилятор ничего не предписывает о том, где и когда они могут появиться, или о том, где код их улавливает (и происходит ли это вообще). Знания о непроверенных исключениях либо передаются мелким шрифтом в кодовом соглашении, либо вообще не передаются в записанном соглашении, если программисты забывают их документировать.

Листинг 8.26 содержит код для двух смежных слоев абстракции. Нижний слой — это интерфейс `TextImportanceScorer`, а верхний слой — это класс `TextSummarizer`. В этом сценарии `ModelBasedScorer` представляет собой определенную реализацию,

которая реализует интерфейс `TextImportanceScorer`, но `ModelBasedScorer.isImportant()` может вызвать непроверенное исключение `PredictionModelException`.

Листинг 8.26. Исключение, вызывающее утечку деталей реализации

```
class TextSummarizer {
    private final TextImportanceScorer importanceScorer; ❶
    ...

    String summarizeText(String text) {
        return paragraphFinder.find(text)
            .filter(paragraph =>
                importanceScorer.isImportant(paragraph))
            .join("\n\n");
    }
}

interface TextImportanceScorer {
    Boolean isImportant(String text);
}

class ModelBasedScorer implements TextImportanceScorer { ❷
    ...
    /**
     * @throws PredictionModelException if there is an error ❸
     *     running the prediction model.
     */
    override Boolean isImportant(String text) {
        return model.predict(text) >= MODEL_THRESHOLD;
    }
}
```

- ❶ Зависит от интерфейса `TextImportanceScorer`.
- ❷ Реализация интерфейса `TextImportanceScorer`.
- ❸ Непроверенное исключение, которое может быть вызвано.

Программист, использующий класс `TextSummarizer`, скорее всего, рано или поздно заметит, что его код иногда выходит из строя из-за `PredictionModelException`, и тогда он может захотеть обработать этот сценарий ошибки поизящнее и оправиться от этой ошибки. Для этого программисту пришлось бы написать что-то вроде кода в листинге 8.27. Такой код улавливает исключение `PredictionModelException` и выводит пользователю сообщение об ошибке. Для того чтобы заставить код работать, программист должен осознать тот факт, что класс `TextSummarizer` может использовать прогнозирование на основе модели (это деталь реализации).

Это не только нарушает концепцию слоев абстракции, но также может быть ненадежным и подверженным ошибкам. `Textsummarizer` зависит от интерфейса `TextImportanceScorer` и, таким образом, может быть сконфигурирован с любой реали-

зацией этого интерфейса. `ModelBasedScorer` — лишь одна из таких реализаций, но, скорее всего, не единственная. `TextSummarizer` может быть сконфигурирован с другой реализацией `TextImportanceScorer`, которая выдает исключение совершенно другого типа. Если это произойдет, то оператор `catch` не уловит исключение, и программа либо выйдет из строя, либо пользователь увидит потенциально менее полезное сообщение об ошибке с более высокого уровня в коде.

Листинг 8.27. Улавливание исключений, зависящих от конкретной реализации

```
void updateTextSummary(UserInterface ui) {  
    String userText = ui.getUserText();  
    try {  
        String summary = textSummarizer.summarizeText(userText);  
        ui.getSummaryField().setValue(summary);  
    } catch (PredictionModelException e) { ❶  
        ui.getSummaryField().setError("Unable to summarize text");  
    }  
}
```

❶ `PredictionModelException` уловлено и обработано.

Риск утечки деталей реализации характерен не только для непроверенных исключений, в данном случае их использование усугубляет проблему. Программистам слишком легко не документировать, какие непроверенные исключения могут быть вызваны, и классы, реализующие интерфейс, не принуждаются выдавать только те ошибки, которые диктует интерфейс.

8.7.2. Решение: сделайте исключения соответствующими слою абстракции

Чтобы предотвратить утечку деталей реализации, каждый уровень кода в идеале должен выявлять только те типы ошибок, которые отражают данный слой абстракции. Мы можем достичь этого, обернув любые ошибки с нижних слоев в типы ошибок, соответствующие текущему слою. Это означает, что вызывающим пользователям предоставляется соответствующий слой абстракции, а также гарантируется, что исходная информация об ошибке не будет потеряна (поскольку она все еще присутствует в обернутой ошибке).

Листинг 8.28 демонстрирует это. Для нового типа исключения под названием `TextSummarizerException` было определено, что оно должно сигнализировать о любых ошибках, связанных с резюмированием текста. Аналогично для исключения `TextImportanceScorerException` было определено, что оно сигнализирует о любых ошибках, связанных с количественной оценкой текста (независимо от того, какая реализация интерфейса используется). Наконец, код был изменен для использования явного метода сигнализации об ошибках. В примере это достигается с помощью проверенных исключений.

Очевидным недостатком является то, что теперь появилось больше строк кода, так как нам пришлось определить некоторые пользовательские классы исключений и улавливать, оборачивать и повторно создавать различные исключения. На первый взгляд код может показаться «более сложным», но на самом деле это не так, если рассматривать программное обеспечение в целом. Программист, использующий класс `TextSummarizer`, теперь получает только один тип ошибок, с которыми ему придется иметь дело, и он точно знает, какой это тип. Недостатки дополнительного шаблона обработки ошибок в коде, скорее всего, будут перевешены преимуществами улучшенной модульности и более предсказуемого поведения класса `TextSummarizer`.

РЕЗЮМЕ: АЛЬТЕРНАТИВЫ ПРОВЕРЕННЫМ ИСКЛЮЧЕНИЯМ

Проверенные исключения являются лишь одним из видов явного метода сигнализации об ошибках и (среди основных языков программирования) более или менее уникальны для Java. В *главе 4* это подробно описано; продемонстрировано несколько альтернативных явных методов, которые можно использовать на любом языке (например, типы `Result` и `Outcome`). Проверенные исключения используются в листинге 8.28, чтобы код был более сопоставим с кодом в листинге 8.26.

Еще одна вещь, обсуждавшаяся в *главе 4*, заключается в том, что сигнализация и обработка ошибок являются спорной темой, и, в частности, программисты расходятся во мнениях относительно использования непроверенных исключений по сравнению с более явными методами сигнализации ошибок для ошибок, которые вызывающий пользователь может захотеть исправить. Но даже если мы работаем над кодовой базой, где для этого поощряется использование непроверенных исключений, все равно важно убедиться, что не происходит утечки деталей реализации (как показано в предыдущем подразделе).

Иногда подход программистов, использующих непроверенные исключения, состоит в том, чтобы предпочесть стандартные типы исключений (например, `ArgumentException` или `StateException`), потому что более вероятно, что другие программисты будут предсказывать, что именно такие исключения могут быть созданы, и обрабатывать их соответствующим образом. Недостатком этого является то, что это может ограничить способность дифференцировать различные сценарии ошибок (это также обсуждалось в *главе 4*, *разд. 4.5.2*).

Листинг 8.28. Исключения, соответствующие слоям

```
class TextSummarizerException extends Exception { ❶
    ...
    TextSummarizerException(Throwable cause) { ... } ❷
    ...
}

class TextSummarizer {
    private final TextImportanceScorer importanceScorer;
    ...
    String summarizeText(String text)
        throws TextSummarizerException {
        try {
```

```

        return paragraphFinder.find(text)
            .filter(paragraph =>
                importanceScorer.isImportant(paragraph))
            .join("\n\n");
    } catch (TextImportanceScorerException e) {
        throw new TextSummarizerException(e); ❸
    }
}

class TextImportanceScorerException extends Exception { ❹
    ...
    TextImportanceScorerException(Throwable cause) { ... }
    ...
}

interface TextImportanceScorer {
    Boolean isImportant(String text)
        throws TextImportanceScorerException;
}

class ModelBasedScorer implements TextImportanceScorer {
    ...
    Boolean isImportant(String text)
        throws TextImportanceScorerException {
        try {
            return model.predict(text) >= MODEL_THRESHOLD;
        } catch (PredictionModelException e) {
            throw new TextImportanceScorerException(e); ❺
        }
    }
}

```

❶ Исключение для сигнализации об ошибке, связанной с резюмированием текста.

❷ Конструктор принимает другое исключение для оборачивания (Throwable — это суперкласс Exception).

❸ TextImportanceScorerException завернуто в TextSummarizerException и повторно создано.

❹ Исключение для сигнализации об ошибке, связанной с количественной оценкой текста.

❺ Интерфейс определяет типы ошибок, предоставляемые слоем абстракции.

Программисту, использующему класс TextSummarizer, теперь потребуется обрабатывать только TextSummarizerException. Это означает, что ему не нужно знать о каких-либо деталях реализации, а также то, что обрабатывание им ошибок будет продолжать работать, независимо от того как класс TextSummarizer был сконфигурирован или будет изменен в будущем. Это показано ниже.

Листинг 8.29. Улавливание исключения, соответствующего слоя абстракции

```
void updateTextSummary(UserInterface ui) {  
    String userText = ui.getUserText();  
    try {  
        String summary = textSummarizer.summarizeText(userText);  
        ui.getSummaryField().setValue(summary);  
    } catch (TextSummarizerException e) {  
        ui.getSummaryField().setError("Unable to summarize text");  
    }  
}
```

Если мы точно знаем, что ошибка не такая, от которой любой вызывающий пользователь никогда не захочет оправиться, то утечка деталей реализации не является серьезной проблемой, поскольку более высокие слои, вероятно, в любом случае не будут пытаться обработать эту конкретную ошибку. Но всякий раз, когда у нас возникает ошибка, которую вызывающий пользователь может захотеть исправить, важно убедиться, что тип ошибки соответствует слою абстракции. Явные методы сигнализации об ошибках (например, проверенные исключения, типы `Result` и `Outcome`) могут упростить применение этого правила.

Итоги

- ◆ Модульный код обычно легче адаптировать к меняющимся требованиям.
- ◆ Одна из ключевых целей модульности заключается в том, что изменение требования должно затрагивать только те части кода, которые непосредственно связаны с этим требованием.
- ◆ Создание модульного кода тесно связано с созданием чистых слоев абстракции.
- ◆ Для создания модульного кода можно использовать следующие методы:
 - внедрение зависимостей;
 - зависимость от интерфейсов вместо конкретных классов;
 - интерфейсы и композицию вместо наследования классов;
 - делать классы способными заботиться сами о себе;
 - инкапсулировать связанные данные вместе;
 - обеспечивать, чтобы возвращаемые типы и исключения не допускали утечки деталей реализации.

9 Сделайте код многоразовым и обобщаемым

ЭТА ГЛАВА ОХВАТЫВАЕТ

- ◇ Как написать код, который можно безопасно использовать повторно
- ◇ Как написать код, который можно обобщить для решения различных задач

В *главе 2* обсуждалось, как мы, инженеры, часто решаем проблему высокого уровня, разбивая ее на ряд подзадач. Когда мы делаем это снова и снова, в одном проекте за другим, мы часто обнаруживаем, что одни и те же подзадачи возникают снова и снова. Если мы или другие инженеры уже решили данную подзадачу, то имеет смысл повторно использовать это решение. Это экономит нам время и снижает вероятность ошибок (потому что код уже опробован и протестирован).

К сожалению, только потому что решение подзадачи уже существует, это не всегда означает, что мы можем использовать его повторно. Это может произойти, если решение делает предположения, которые не соответствуют нашему варианту использования, или если оно связано с какой-либо другой логикой, которая нам не нужна. Поэтому стоит активно обдумать это и сознательно написать, структурируя код таким образом, чтобы его можно было использовать повторно в будущем. Это может потребовать немного больше первоначальных усилий (хотя часто ненамного больше), но обычно экономит нам и нашим товарищам по команде время и усилия в долгосрочной перспективе.

Эта глава тесно связана с созданием чистых слоев абстракции (*глава 2*) и созданием модульного кода (*глава 8*). Создание чистых слоев абстракции и модульность кода, как правило, приводят к тому, что решения подзадач разбиваются на отдельные фрагменты кода, которые слабо связаны. Обычно это делает код намного проще и безопаснее для повторного использования и обобщения. Но то, что мы обсуждали в *главах 2* и *8*, — не единственные соображения, которые помогают сделать код многоразовым и обобщаемым. В этой главе рассматриваются некоторые дополнительные вещи, о которых стоит подумать.

9.1. Остерегайтесь предположений

Допущения иногда могут привести к тому, что код станет проще, эффективнее или и то, и другое вместе. Но предположения также, как правило, приводят к тому, что

код становится более хрупким и менее универсальным, что может сделать его менее безопасным для повторного использования. Невероятно сложно отследить, какие именно предположения в каких частях кода были сделаны, поэтому они могут легко превратиться в неприятные ловушки, куда непреднамеренно попадут другие инженеры. То что поначалу могло показаться простым способом улучшения кода, на самом деле может иметь противоположный эффект, когда ошибки и странное поведение проявляются сразу после повторного использования кода.

Учитывая это, стоит продумать затраты и выгоды, прежде чем вносить предположение в код. Если очевидный выигрыш в упрощении или эффективности кода незначителен, то возможно, лучше избегать таких предположений, поскольку затраты, связанные с повышенной хрупкостью, могут перевесить их. В следующих подразделах это рассматривается.

9.1.1. Предположения могут привести к ошибкам при повторном использовании кода

Рассмотрим код в листинге 9.1. Класс `Article` представляет статью на новостном веб-сайте, которую могут читать пользователи. Функция `getAllImages()` возвращает все изображения, содержащиеся в статье. Для этого он перебирает разделы в статье, пока не найдет тот, который содержит изображения, а затем возвращает изображения из этого раздела. В коде предполагается, что будет только один раздел, содержащий изображения. Это предположение прокомментировано в коде, но вряд ли это заметит любой вызывающий код.

При таком допущении производительность кода незначительно повышается, поскольку он может выйти из цикла `for`, как только будет найден раздел, содержащий изображение, но этот выигрыш, вероятно, настолько мал, что не имеет реальных последствий. Однако может иметь значение тот факт, что функция `getAllImages()` не вернет все изображения, если она когда-либо использовалась со статьей, содержащей изображения более чем в одном разделе. Это вполне может быть несчастным случаем, ожидающим своего часа, и, когда это произойдет, это, скорее всего, вызовет ошибку.

Листинг 9.1. Код, содержащий предположение

```
class Article {
    private List<Section> sections;
    ...

    List<Image> getAllImages() {
        for (Section section in sections) {
            if (section.containsImages()) {
                // В статье, содержащей изображения, ❶
                // должно быть не более одного раздела.
```

```
        return section.getImages(); ❷  
    }  
}  
}  
return [];  
}  
}
```

- ❶ Предположение, прокомментированное в коде.
- ❷ Возвращает изображения только из первого раздела, содержащего изображение.

Предположение о том, что будет только один раздел изображения, несомненно, было правильным для исходного варианта использования, который имел в виду автор. Но он может легко стать неправильным, если класс `Article` когда-либо будет повторно использоваться для чего-то другого (или если размещение изображений в статьях когда-либо изменится). И поскольку это предположение глубоко запрятано в коде, очень маловероятно, что вызывающие абоненты узнают об этом. Они увидят функцию `getAllImages()` и предположат, что она возвращает «все» изображения. К сожалению, это верно только в том случае, если скрытое предположение верно.

9.1.2. Решение: избегайте ненужных предположений

Компромисс между затратами и выгодами, заключающийся в предположении, что существует только один раздел с изображениями, предполагает, что это, вероятно, нецелесообразное предположение. С одной стороны, наблюдается незначительный прирост производительности (который, скорее всего, не будет замечен), но, с другой стороны, существует реальная вероятность появления ошибок, если кто-то повторно использует код или если требования изменятся. Учитывая это, вероятно, лучше просто избавиться от этого предположения: его присутствие сопряжено с риском без заметного выигрыша.

ПРЕЖДЕВРЕМЕННАЯ ОПТИМИЗАЦИЯ

Стремление избежать преждевременной оптимизации является устоявшейся концепцией в области разработки программного обеспечения и информатики. Оптимизация кода, как правило, связана с затратами: для реализации оптимизированного решения часто требуется больше времени и усилий, а результирующий код часто менее удобочитаем, сложнее в обслуживании и потенциально менее надежен (если вводятся допущения). В дополнение к этому оптимизация обычно приносит заметную пользу только в тех случаях, когда выполняется в фрагментах кода, которые выполняются много тысяч или миллионов раз в рамках программы.

Поэтому в большинстве сценариев лучше сосредоточиться на том, чтобы сделать код читабельным, поддерживаемым и надежным, а не гнаться за незначительным повышением производительности. Если часть кода в итоге выполняется много раз и было бы полезно ее оптимизировать, это можно сделать позже, когда это станет очевидным.

В листинге 9.2 показано, как будет выглядеть код, если убрать предположение, изменив функцию `getAllImages()` так, чтобы она возвращала изображения из всех раз-

делов (а не только из первого, содержащего изображения). Это делает функцию намного более универсальной и надежной для различных вариантов использования. Недостатком является то, что это приводит к тому, что цикл `for` может выполняться еще несколько итераций, но, как только что было отмечено, маловероятно, что это окажет заметное влияние на производительность.

Листинг 9.2. Код с удаленным допущением

```
class Article {
    private List<Section> sections;
    ...

    List<Image> getAllImages() {
        List<Image> images = []; ❶
        for (Section section in sections) {
            images.addAll(section.getImages());
        }
        return images;
    }
}
```

❶ Собирает и возвращает изображения из всех разделов.

При написании кода мы часто очень внимательно относимся к таким вещам, как затраты на производительность при выполнении строки кода больше раз, чем необходимо. Но важно помнить, что допущения также сопряжены с соответствующими издержками с точки зрения хрупкости. Если принятие определенного предположения приводит к значительному увеличению производительности или чрезвычайно упрощению кода, то, возможно, его стоит сделать. Но если выгоды будут незначительными, то затраты, связанные с внедрением допущения в код, вполне могут перевесить выгоды.

9.1.3. Решение: если предположение необходимо, приведите его в исполнение

Иногда необходимо сделать предположение или упростить код до такой степени, что выгоды перевешивают затраты. Когда мы решаем сделать предположение в коде, мы все равно должны помнить о том факте, что другие инженеры могут не знать об этом, поэтому, чтобы они случайно не попались на наше предположение, мы должны обеспечить его соблюдение. Как правило, для достижения этой цели мы можем использовать два подхода.

1. *Сделайте допущение «невозможно сломать».* Если мы сможем написать код таким образом, что он не будет компилироваться в случае нарушения допущения, то это гарантирует, что допущение всегда выполняется. Об этом говорилось в главах 3 и 7.

2. *Используйте метод сигнализации об ошибке.* Если невозможно сделать предположение невозможным для нарушения, то мы можем написать код, обнаруживающий его нарушение, и использовать технику сигнализации о быстрой ошибке. Это было рассмотрено в *главе 4* (и в конце *главы 3*).

Потенциально проблематичное, необоснованное предположение

Чтобы продемонстрировать, как необоснованное предположение может быть проблематичным, давайте рассмотрим другую функцию, которую может содержать класс `Article`, возвращающую раздел с изображениями. В листинге 9.3 показано, как может выглядеть эта функция, если она не обеспечивает выполнение предположения. Она находит разделы, содержащие изображения, а затем возвращает раздел или нулевое значение (`null`), если ни один раздел не содержит изображений. В этом коде снова делается предположение, что статья всегда будет содержать максимум один раздел с изображением. Если это предположение нарушено и статья содержит несколько разделов с изображениями, то код не выйдет из строя и не выдаст никаких предупреждений. Вместо этого он просто вернет первый такой раздел и продолжит, как будто все в порядке (противоположность быстрой ошибке).

Листинг 9.3. Код, содержащий предположение

```
class Article {  
    private List<Section> sections;  
    ...  
  
    Section? getImageSection() {  
        // В статье, содержащей изображения,  
        // должно быть не более одного раздела.  
        return sections ❶  
            .filter(section -> section.containsImages())  
            .first();  
    }  
}
```

❶ Возвращает первое изображение, содержащее раздел, или `null`, если таковых нет.

Функция `getImageSection()` вызывается одним из фрагментов кода, который преобразовывает статьи для отображения их пользователям. Это показано в листинге 9.4. Шаблон, в котором этот код отображает статьи, имеет место только для одного раздела изображения. Следовательно, для этого конкретного случая использования необходимо предположение, что статья имеет максимум один раздел с изображениями.

Листинг 9.4. Вызывающий абонент, который полагается на предположение

```
class ArticleRenderer {  
    ...
```

```

void render(Article article) {
    ...
    Section? imageSection = article.getImageSection(); ❶
    if (imageSection != null) {
        templateData.setImageSection(imageSection);
    }
    ...
}
}

```

❶ Шаблон статьи может обрабатывать максимум один раздел, содержащий изображения.

Если кто-то создаст статью с несколькими разделами изображений, а затем попытается отобразить ее с помощью этого кода, все будет вести себя странно и неожиданно. Будет казаться, что все работает (в том смысле, что никаких ошибок или предупреждений не возникает), но на самом деле в статье будет отсутствовать множество изображений. В зависимости от характера статьи это может привести к тому, что она будет вводящей в заблуждение или совершенно бессмысленной.

Подкрепление предположения

Как отмечалось в *главе 4*, обычно лучше следить за тем, чтобы сбои и ошибки не оставались незамеченными. В этом сценарии попытка отобразить статью с несколькими разделами изображений не поддерживается и, следовательно, является сценарием ошибки. Вероятно, было бы лучше, если бы в этом сценарии код быстро выходил из строя, а не пытался хромать дальше. Мы можем изменить код, чтобы сделать это, применив предположение с использованием метода сигнализации об ошибках.

В листинге 9.5 показано, как могла бы выглядеть функция `Article.getImageSection()`, если бы она использовала утверждение для обеспечения предположения о том, что существует не более одного раздела изображения. Функция также была переименована в `getOnlyImageSection()`, чтобы лучше сообщить всем вызывающим эту функцию, что они предполагают, что существует только один раздел изображения. Это делает маловероятным, что какие-либо вызывающие абоненты, которые не хотят делать это предположение, будут вызывать его.

Листинг 9.5. Подтверждение предположения

```

class Article {
    private List<Section> sections;
    ...

    Section? getOnlyImageSection() { ❶
        List<Section> imageSections = sections
            .filter(section -> section.containsImages());
    }
}

```

```
    assert(imageSections.size() <= 1, ❷  
        "Article contains multiple image sections");  
  
    return imageSections.first(); ❸  
}  
}
```

- ❶ Имя функции передает предположение, которое делают вызывающие.
- ❷ Утверждение подкрепляет предположение.
- ❸ Возвращает первый элемент в списке `imageSections` или `null`, если он пуст.

Методы сигнализации об ошибках

В главе 4 подробно обсуждались различные методы сигнализации об ошибках, в частности, то, как выбор метода часто зависит от того, захочет ли вызывающий абонент исправить ошибку.

В листинге 9.5 используется утверждение, которое уместно, если мы уверены, что ни один вызывающий абонент не захочет восстанавливаться после ошибки. Если статья сгенерирована внутри нашей программы, то нарушение предположения подразумевает ошибку программирования, а это означает, что утверждение, вероятно, уместно. Но если статья предоставлена внешней системой или пользователем, то вполне вероятно, что некоторые абоненты захотят обнаружить ошибку и обработать ее более изящным способом. В этом сценарии более подходящим может быть явный метод сигнализации об ошибках.

Итак, допущения, как правило, сопряжены с определенными издержками в плане повышенной хрупкости. Когда затраты на данное предположение перевешивают выгоды, вероятно, лучше его не делать. Если предположение необходимо, то мы должны сделать все возможное, чтобы другие инженеры не попались на его удочку; мы можем добиться этого, применяя предположение.

9.2. Остерегайтесь глобального состояния

Часть *глобального состояния* (или *глобальная переменная*) — это часть, которая является общей для всех контекстов в данном экземпляре программы. Вот некоторые распространенные способы определения глобальной переменной.

- ◆ Пометка переменной как статической (`static`) в таких языках, как Java или C# (это парадигма, используемая в псевдокоде в этой книге).
- ◆ Определение переменной уровня файла (вне класса или функции) в таких языках, как C++.
- ◆ Определение свойств глобального объекта окна в языках на основе JavaScript.

Чтобы продемонстрировать, что означает глобальная переменная, рассмотрим код в листинге 9.6. Вот некоторые вещи, на которые стоит обратить внимание в коде.

- ◆ а — переменная экземпляра. Каждый экземпляр `MyClass` будет иметь свою собственную выделенную переменную. Один экземпляр класса, модифицирующий эту переменную, не повлияет ни на один другой экземпляр класса.

- ◆ `b` — статическая переменная (то есть глобальная переменная). Поэтому она является общей для всех экземпляров `MyClass` (и к ней можно получить доступ даже без экземпляра `MyClass`; это объясняется в следующем пункте).
- ◆ Функция `getBStatically()` помечена как `static`, что означает, что ее можно вызывать без необходимости в экземпляре класса, используя такой вызов: `MyClass.getBStatically()`. Статическая функция, подобная этой, может получить доступ к любым статическим переменным, определенным в классе, но никогда не может получить доступ ни к каким переменным экземпляра.

Листинг 9.6. Класс с глобальной переменной

```
class MyClass {
    private Int a = 3; ❶
    private static Int b = 4; ❷

    void setA(Int value) { a = value; }
    Int getA() { return a; }

    void setB(Int value) { b = value; }
    Int getB() { return b; }

    static Int getBStatically() { return b; } ❸
}
```

❶ Переменная экземпляра.

❷ Глобальная переменная (потому что она помечена как статическая).

❸ Статическая функция.

Следующий фрагмент демонстрирует, как переменная экземпляра `a` применяется к отдельным экземплярам класса, в то время как глобальная переменная `b` является общей для всех экземпляров класса (а также статических контекстов):

```
MyClass instance1 = new MyClass();
MyClass instance2 = new MyClass();

instance1.setA(5); ❶
instance2.setA(7);
print(instance1.getA()) // Вывод: 5
print(instance2.getA()) // Вывод: 7

instance1.setB(6); ❷
instance2.setB(8);
print(instance1.getB()) // Вывод: 8
print(instance2.getB()) // Вывод: 8
print(MyClass.getBStatically()) // Вывод: 8 ❸
```

- ❶ Каждый экземпляр `MyClass` имеет свою собственную отдельную переменную `a`.
- ❷ Глобальная переменная `b` является общей для всех экземпляров `MyClass`.
- ❸ Доступ к `b` также можно получить статически, не требуя экземпляра `MyClass`.

НЕ ПУТАЙТЕ ГЛОБАЛЬНОСТЬ С ВИДИМОСТЬЮ

Независимо от того является ли переменная глобальной или нет, не следует путать это с видимостью переменной. Видимость переменной зависит от того, является ли она общедоступной или закрытой, что определяет, какие другие части кода могут видеть ее и получать к ней доступ. Переменная может быть общедоступной или закрытой, независимо от того, является ли она глобальной. Дело в том, что глобальная переменная является общей для всех контекстов в программе, вместо того чтобы каждый экземпляр класса или функции имел свою собственную версию.

Поскольку глобальные переменные влияют на каждый контекст внутри программы, их использование часто делает неявное предположение, что никто никогда не захочет повторно использовать код для несколько иной цели. Как мы видели в предыдущем разделе, допущения сопряжены с соответствующими затратами. Глобальное состояние, как правило, делает код чрезвычайно хрупким и совершенно небезопасным для повторного использования, поэтому затраты обычно перевешивают выгоды. В следующих подразделах объясняется почему и предлагается альтернатива.

9.2.1. Глобальное состояние может сделать повторное использование небезопасным

Когда есть какое-то состояние, к которому необходимо получить доступ различным частям программы, может показаться заманчивым поместить его где-нибудь в глобальную переменную. Это позволяет любому фрагменту кода очень легко получить доступ к состоянию. Но, как только что упоминалось, это также часто приводит к тому, что код становится невозможно безопасно повторно использовать. Чтобы продемонстрировать почему, давайте представим, что мы создаем приложение для онлайн-покупок. В нашем приложении пользователи могут просматривать товары, добавлять их в свою корзину, а затем проверять в конце.

В этом сценарии содержимое корзины покупок пользователя составляет состояние, к которому необходимо получить доступ многим различным частям приложения, таким, как любой код, добавляющий в нее товары, экран, на котором пользователи просматривают содержимое своей корзины, и код, который обрабатывает оформление заказа. Поскольку так много частей приложения нуждаются в доступе к этому общему состоянию, у нас может возникнуть соблазн сохранить содержимое корзины пользователя в глобальной переменной. В листинге 9.7 показано, как может выглядеть код корзины покупок, если мы используем глобальное состояние. Некоторые моменты, на которые следует обратить внимание в коде, заключаются в следующем.

- ◆ Переменная `items` отмечена словом `static`. Это означает, что эта переменная не связана с конкретным экземпляром класса `ShoppingBasket`, что делает ее глобальной переменной.

- ◆ Функции `addItem()` и `getItems()` также помечены как статические. Это означает, что их можно вызывать из любого места в коде (без использования экземпляра `ShoppingBasket()`) следующим образом: `ShoppingBasket.addItem(...)` и `ShoppingBasket.getItems()`. При вызове они получают доступ к глобальной переменной `items`.

Листинг 9.7. Класс `ShoppingBasket`

```
class ShoppingBasket {
    private static List<Item> items = []; ❶

    static void addItem(Item item) { ❷
        items.add(item);
    }

    static void List<Item> getItems() {
        return List.copyOf(items);
    }
}
```

❶ Помечен как статический, что делает его глобальной переменной.

❷ Функции, помеченные как статические.

Это можно легко сделать в любом месте кода, которому требуется доступ к корзине покупок пользователя. Некоторые примеры показаны в листинге 9.8. `ViewItemWidget` позволяет пользователям добавлять просмотренный товар в свою корзину. Это достигается вызовом функции `ShoppingBasket.addItem()`. `ViewBasketWidget` позволяет пользователям просматривать содержимое своей корзины. Доступ к содержимому корзины можно получить, вызвав `ShoppingBasket.getItems()`.

Листинг 9.8. Классы с использованием `ShoppingBasket`

```
class ViewItemWidget {
    private final Item item;

    ViewItemWidget(Item item) {
        this.item = item;
    }
    ...

    void addItemToBasket() {
        ShoppingBasket.addItem(item); ❶
    }
}

class ViewBasketWidget {
    ...
}
```

```
void displayItems() {  
    List<Item> items = ShoppingBasket.getItems(); ❷  
    ...  
}  
}
```

❶ Изменяет глобальное состояние.

❷ Читает глобальное состояние.

Изменять и считывать содержимое корзины тривиально просто, и именно поэтому может показаться заманчивым использовать глобальное состояние таким образом. Но при использовании глобального состояния, подобного этому, мы создали код, который будет ломаться и делать потенциально странные вещи, если кто-нибудь когда-нибудь попытается его повторно использовать. В следующем подразделе объясняется почему.

Что происходит, когда кто-то пытается повторно использовать этот код?

Осознавали мы это или нет, но при написании этого кода мы подразумевали, что для каждого запущенного экземпляра нашего программного обеспечения потребуется только одна корзина. Если наше приложение для покупок работает только на устройстве пользователя, то для базовой функциональности это предположение будет верным, и все будет работать правильно. Но есть много причин, по которым это предположение может быть нарушено, а это значит, что оно довольно хрупкое. Некоторые потенциальные сценарии, которые могут привести к нарушению этого предположения, заключаются в следующем.

- ◆ Мы решили сделать резервную копию содержимого корзины пользователей на нашем сервере, поэтому мы начинаем использовать класс `ShoppingBasket` в нашем коде на стороне сервера. Один экземпляр нашего сервера будет обрабатывать множество запросов от многих разных пользователей, поэтому теперь у нас есть много корзин для каждого запущенного экземпляра нашего программного обеспечения (в данном случае сервера).
- ◆ Мы добавляем функцию, которая позволяет пользователю сохранять содержимое своей корзины на потом. Это означает, что клиентское приложение теперь должно обрабатывать несколько различных корзин: все сохраненные на потом в дополнение к активной.
- ◆ Мы начинаем продавать свежие продукты в дополнение к нашему обычному складу. Для этого используется совершенно другой набор поставщиков и механизм доставки, поэтому его нужно обрабатывать как отдельную корзину для покупок.

Мы, вероятно, могли бы сидеть здесь весь день, придумывая различные сценарии, которые нарушают наши первоначальные предположения. Произойдет ли какое-либо из них на самом деле, остается только гадать. Но дело в том, что существует достаточно правдоподобно звучащих сценариев, которые нарушают наше первоначальное предположение, и мы, вероятно, должны понимать, что оно хрупкое и, вероятно, так или иначе будет нарушено в какой-то момент.

Когда наше первоначальное предположение нарушается, с программным обеспечением что-то пойдет не так.

Если два разных фрагмента кода используют класс `ShoppingBasket`, они будут мешать друг другу (рис. 9.1). Если один из них добавит элемент, этот элемент окажется в корзине для всех остальных фрагментов кода, использующих его. В любом из только что перечисленных сценариев это, вероятно, приведет к ошибкам в поведении, поэтому класс `ShoppingBasket` практически невозможно повторно использовать безопасным способом.



Рис. 9.1. Использование глобального состояния может сделать повторное использование кода

В лучшем случае инженер поймет, что повторное использование класса `shoppingBasket` небезопасно, и напишет новый, совершенно отдельный код для своего нового варианта использования. В худшем случае они могут не осознавать, что его повторное использование небезопасно и программное обеспечение будет содержать ошибки. Эти ошибки могут быть весьма катастрофическими, если клиенты в итоге заказывают товары, которые им не нужны, или если мы нарушаем их конфиденциальность, раскрывая товары в их корзинах другим лицам. Подводя итог, в лучшем случае мы получаем кучу почти дублированного кода, который инженерам теперь необходимо поддерживать, а в худшем случае мы получаем несколько неприятных ошибок. Ни то, ни другое не является особенно желательным, поэтому, вероятно, было бы гораздо лучше избегать использования глобального состояния. В следующем подразделе обсуждается альтернатива.

9.2.2. Решение: общее состояние с внедрением зависимостей

В предыдущей главе обсуждалась техника внедрения зависимостей. Это означает, что мы создаем класс, «внедряя» его зависимости, а не заставляя его жестко запро-

граммировать зависимость от них. Внедрение зависимостей также является отличным способом обмена состоянием между различными классами более контролируемым способом, чем использование глобального состояния.

Класс `ShoppingBasket`, который мы видели в предыдущем подразделе, использует статическую переменную и статические функции, что означает, что состояние было глобальным, поэтому первый шаг — изменить это, создав класс `ShoppingBasket`, экземпляр которого необходимо создавать, и убедиться, что каждый экземпляр класса имеет свое собственное особое состояние. В листинге 9.9 показано, как это должно выглядеть. Вот некоторые вещи, на которые стоит обратить внимание в коде.

- ◆ Переменная `items` больше не является статической. Теперь это переменная экземпляра, то есть она связана с конкретным экземпляром класса `ShoppingBasket`, поэтому если мы создадим два экземпляра класса `ShoppingBasket`, они оба будут иметь разные и отдельные списки элементов, содержащихся в них.
- ◆ Функции `addItem()` и `getItems()` больше не являются статическими. Это означает, что к ним можно получить доступ только через экземпляр класса `ShoppingBasket`, поэтому такие вызовы, как `ShoppingBasket.addItem(...)` или `ShoppingBasket.getItems()`, больше не работают.

Листинг 9.9. Модифицированный класс `ShoppingBasket`

```
class ShoppingBasket {  
    private final List<Item> items = []; ❶  
  
    void addItem(Item item) { ❷  
        items.add(item);  
    }  
  
    void List<Item> getItems() {  
        return List.copyOf(items);  
    }  
}
```

❶ Переменная экземпляра (нестатическая).

❷ Нестатические функции-члены.

Второй шаг — внедрение зависимостей экземпляра `ShoppingBasket` во все классы, которым требуется доступ к нему. Делая это, мы можем контролировать, какие фрагменты кода используют одну и ту же корзину, а какие фрагменты кода используют другую корзину. В листинге 9.10 показано, как выглядят `ViewItemWidget` и `ViewBasketWidget`, если в `ShoppingBasket` внедрена зависимость через их конструкторы. Вызовы `addItem()` и `getItems()` теперь относятся к конкретному внедренному экземпляру `ShoppingBasket`.

Листинг 9.10. Внедрение зависимости ShoppingBasket

```
class ViewItemWidget {
    private final Item item;
    private final ShoppingBasket basket;

    ViewItemWidget(Item item, ShoppingBasket basket) { ❶
        this.item = item;
        this.basket = basket;
    }
    ...

    void addItemToBasket() { ❷
        basket.addItem(item);
    }
}

class ViewBasketWidget {
    private final ShoppingBasket basket;

    ViewBasketWidget(ShoppingBasket basket) {
        this.basket = basket;
    }

    void displayItems() {
        List<Item> items = basket.getItems();
        ...
    }
}
```

❶ Внедренная зависимость ShoppingBasket.

❷ Вызывается для конкретного экземпляра ShoppingBasket, который был введен.

Чтобы продемонстрировать, как теперь мы можем безопасно повторно использовать код ShoppingBasket, в листинге 9.11 создаются две ShoppingBasket: одна для обычных продуктов, а другая для свежих продуктов. Также создается два ViewBasketWidget: по одному для каждой корзины. Две корзины полностью независимы друг от друга и никогда не будут мешать друг другу. И каждый ViewBasketWidget будет отображать только элементы из корзины, с которой он был создан.

Листинг 9.11. Отдельные экземпляры ShoppingBasket

```
ShoppingBasket normalBasket = new ShoppingBasket();
ViewBasketWidget normalBasketWidget =
    new ViewBasketWidget(normalBasket);
```

```
ShoppingBasket freshBasket = new ShoppingBasket();  
ViewBasketWidget freshBasketWidget =  
    new ViewBasketWidget(freshBasket);
```

На рис. 9.2 показано, как теперь выглядит внутренняя структура кода. Вместо того чтобы все разделяло одно и то же глобальное состояние, для которого элементы находятся в корзине, каждый экземпляр `ShoppingBasket` теперь автономен.



Рис. 9.2. Сохраняя инкапсуляцию состояния внутри экземпляров классов, повторное использование кода становится безопасным

Глобальное состояние — одна из наиболее известных и хорошо документированных ошибок кодирования. Может быть очень заманчиво использовать его, потому что это может показаться быстрым и простым способом обмена информацией между различными частями программы. Но использование глобального состояния может сделать повторное использование кода совершенно небезопасным. Тот факт, что используется глобальное состояние, может быть неочевиден для другого программиста, поэтому, если он попытается повторно использовать код, это может привести к странному поведению и ошибкам. Если нам нужно разделить состояние между различными частями программы, обычно безопаснее делать это более контролируемым способом, используя внедрение зависимостей.

9.3. Используйте возвращаемые значения по умолчанию надлежащим образом

Использование разумных значений по умолчанию может быть отличным способом сделать программное обеспечение более удобным для пользователя. Представьте,

что при открытии приложения для обработки текстов мы всегда вынуждены выбирать, какой именно шрифт, размер текста, цвет текста, цвет фона, межстрочный интервал и высоту строки мы хотим, прежде чем мы сможем ввести одно слово. Программное обеспечение было бы невыносимо в использовании, и мы, скорее всего, просто переключились бы на альтернативу.

На самом деле большинство приложений для обработки текстов предоставляют набор разумных значений по умолчанию. При открытии приложения оно настраивается с настройками по умолчанию для таких параметров, как шрифт, размер текста и цвет фона. Это означает, что мы можем сразу начать печатать и редактировать эти настройки только тогда, когда захотим.

Даже в части программного обеспечения, которая не ориентирована на пользователя, значения по умолчанию все равно могут быть полезны. Если данный класс может быть настроен с использованием десяти различных параметров, то это облегчает жизнь вызывающим, если им не нужно указывать все эти значения. Поэтому класс может предоставлять некоторые значения по умолчанию для всего, что они не предоставляют.

Предоставление значения по умолчанию часто требует принятия двух допущений:

- какое значение по умолчанию является разумным;
- что более высоким уровням кода все равно, получают ли они значение по умолчанию или значение, которое было задано явно.

Как мы видели ранее, при принятии предположения стоит учитывать как затраты, так и выгоды. Создание подобных предположений в коде высокого уровня, как правило, обходится дешевле, чем их создание в коде более низкого уровня. Код более высокого уровня, как правило, более тесно связан с конкретным вариантом использования, что означает, что проще выбрать значение по умолчанию, подходящее для всех видов использования кода. С другой стороны, код более низкого уровня, как правило, решает более фундаментальные подзадачи и, следовательно, может использоваться более широко для нескольких вариантов использования. Это значительно усложняет выбор значения по умолчанию, которое будет соответствовать всем видам использования кода.

9.3.1. Возвращаемые значения по умолчанию в низкоуровневом коде могут повредить возможности повторного использования

Давайте представим, что мы создаем приложение для обработки текстов. Мы только что установили, что вероятным требованием является то, что у нас есть несколько вариантов стилей текста по умолчанию, чтобы пользователь мог сразу приступить к работе. Если пользователь хочет переопределить их, он может это сделать. В листинге 9.12 показан один из способов реализации этого для выбора шрифта. Класс `UserDocumentSettings` хранит предпочтения пользователя для конкретного документа, одним из которых является шрифт, который он хотел бы использовать.

Если шрифт не указан, функция `getPreferredFont()` возвращает значение по умолчанию `Font.ARIAL`.

Это соответствует требованию, которое мы только что заявили, но, если кто-то когда-либо захочет повторно использовать класс `UserDocumentSettings` для сценария, в котором он не хочет использовать `Arial` в качестве шрифта по умолчанию, ему придется нелегко. Невозможно провести различие между случаем, когда пользователь специально выбрал `Arial`, и случаем, когда он не указал предпочтение (то есть было возвращено значение по умолчанию).

Листинг 9.12. Возврат значения по умолчанию

```
class UserDocumentSettings {
    private final Font? font;
    ...

    Font getPreferredFont() {
        if (font != null) {
            return font;
        }
        return Font.ARIAL; ❶
    }
}
```

❶ Значение по умолчанию для `Font.ARIAL` возвращается, если нет предпочтений пользователя.

Этот подход также вредит адаптивности: если требования к значениям по умолчанию когда-либо изменятся, это будет проблематично. Одним из примеров является то, что мы начинаем продавать наше приложение для обработки текстов крупным организациям, которые хотят иметь возможность указывать шрифт по умолчанию для всей организации. Это сложно реализовать, потому что класс `UserDocumentSettings` не позволяет нам определить, когда нет предпочтения, предоставленного пользователем (и, следовательно, когда может применяться значение по умолчанию для всей организации).

Объединив возвращаемое значение по умолчанию в класс `UserDocumentSettings`, мы сделали предположение о каждом потенциальном уровне кода выше: что `Arial` является разумным шрифтом по умолчанию. Сначала это, вероятно, будет хорошо, но если другие инженеры захотят повторно использовать наш код или изменятся требования, то это предположение может легко стать проблематичным. На рис. 9.3 показано, как такое допущение влияет на слои кода выше. Чем ниже уровень в коде, для которого мы определяем значение по умолчанию, тем выше слоем мы делаем предположение.

В главе 2 подчеркивались преимущества чистых уровней абстракции. Один из ключевых способов добиться этого — убедиться, что мы разделяем отдельные подзадачи на отдельные фрагменты кода. Класс `UserDocumentSettings` противоречит этому: получение некоторых пользовательских настроек и определение разумных значений по умолчанию для нашего приложения — это две отдельные подзадачи. Но

класс `UserDocumentSettings` связывает их вместе таким образом, что делает их совершенно неразделимыми. Это заставляет любого, кто использует класс `UserDocumentSettings`, также использовать нашу реализацию значений по умолчанию. Было бы лучше, если бы мы сделали эти отдельные подзадачи, чтобы более высокие уровни кода могли обрабатывать значения по умолчанию любым подходящим для них способом.

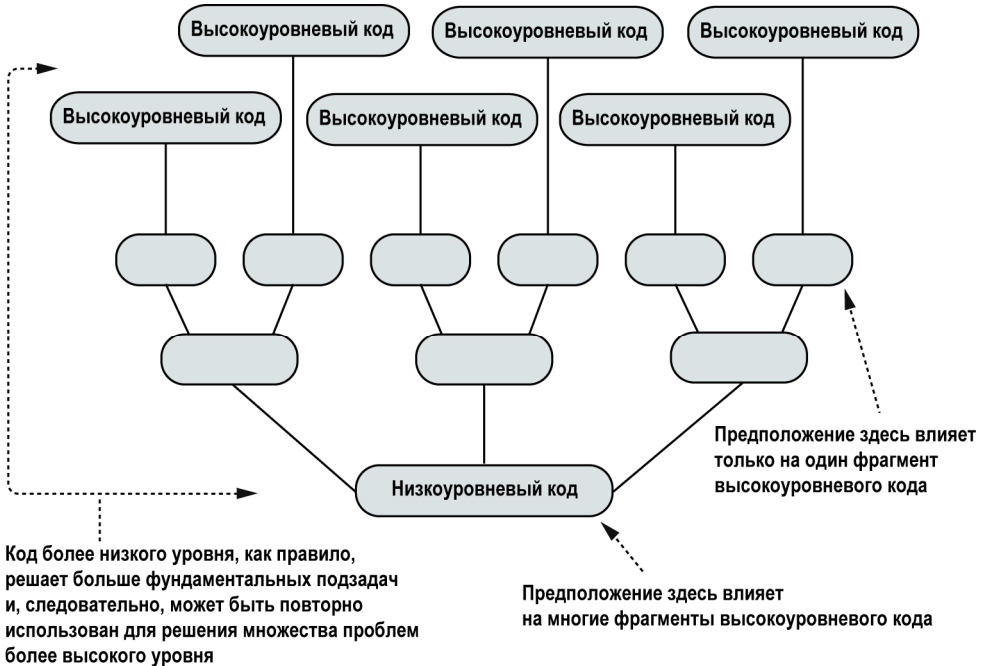


Рис. 9.3. Предположения влияют на уровни кода выше.

Возврат значения по умолчанию в низкоуровневом коде делает предположение, которое, как правило, влияет на многие фрагменты высокоуровневого кода

9.3.2. Решение: укажите значения по умолчанию в коде более высокого уровня

Чтобы удалить решение о значении по умолчанию из класса `UserDocumentSettings`, самое простое, что нужно сделать, это просто вернуть значение `null`, когда нет предоставленного пользователем значения. В следующем листинге показано, как выглядит класс с этим изменением.

Листинг 9.13. Возврат `null`

```
class UserDocumentSettings {
    private final Font? font;
    ...
}
```

```
Font? getPreferredFont() { ❶  
    return font;  
}  
}
```

❶ Null возвращается, если нет предпочтений пользователя.

Это делает предоставление значений по умолчанию отдельной подзадачей, отличной от обработки пользовательских настроек. Это означает, что разные вызывающие абоненты могут решить эту подзадачу любым удобным для них способом, сделав код более многоразовым. В нашем коде более высокого уровня мы могли бы теперь определить выделенный класс для решения подзадачи предоставления значений по умолчанию (как показано в следующем листинге).

Листинг 9.14. Класс для инкапсуляции значений по умолчанию

```
class DefaultDocumentSettings {  
    ...  
  
    Font getDefaultFont() {  
        return Font.ARIAL;  
    }  
}
```

Затем мы могли бы определить класс `DocumentSettings`, который обрабатывает логику выбора между значением по умолчанию и значением, предоставленным пользователем. Листинг 9.15 показывает это. Класс `DocumentSettings` обеспечивает чистый уровень абстракции для кода более высокого уровня, который просто хочет знать, какие параметры использовать. Он скрывает все детали реализации о значениях по умолчанию и предоставленных пользователем значениях, но в то же время также гарантирует, что эти детали реализации могут быть переконфигурированы (с помощью внедрения зависимостей). Это гарантирует возможность повторного использования и адаптации кода.

Листинг 9.15. Уровень абстракции для настроек

```
class DocumentSettings {  
    private final UserDocumentSettings userSettings;  
    private final DefaultDocumentSettings defaultSettings;  
  
    DocumentSettings(  
        UserDocumentSettings userSettings, ❶  
        DefaultDocumentSettings defaultSettings) {  
        this.userSettings = userSettings;  
        this.defaultSettings = defaultSettings;  
    }  
    ...  
}
```



```
Font getFont() {
    Font? userFont = userSettings.getPreferredFont();
    if (userFont != null) {
        return userFont;
    }
    return defaultSettings.getFont();
}
```

❶ Пользовательские настройки и значения по умолчанию вводятся зависимостями.

В листинге 9.15 обработка нулевого значения с помощью оператора `if`, по общему признанию, немного неуклюжа. Во многих языках (таких, как C#, JavaScript и Swift) мы можем использовать оператор объединения `null`, чтобы сделать код менее громоздким. На большинстве языков это записывается как `nullableValue ?? defaultValue`, которое будет оценивать значение `nullableValue`, если оно не равно `null`, и значение `defaultValue` в противном случае. Например, в C# мы могли бы написать функцию `getFont()` следующим образом:

```
Font getFont() {
    return userSettings.getPreferredFont() ?? ❶
        defaultSettings.getFont();
}
```

❶ `??` является оператором объединения `null`.

ПАРАМЕТРЫ ВОЗВРАЩАЕМОГО ЗНАЧЕНИЯ ПО УМОЛЧАНИЮ

Передача решения о том, какое значение по умолчанию использовать вызывающей стороне, делает код более пригодным для повторного использования. Но, возвращая `null` в языках, которые не поддерживают оператор объединения `null`, мы также вынуждаем вызывающую сторону писать шаблонный код для обработки этого.

Один из подходов, который используют некоторые фрагменты кода, заключается в использовании параметра возвращаемого значения по умолчанию. Примером этого является функция `Map.getOrDefault()` в Java. Если карта содержит значение для ключа, то оно будет возвращено. Если карта не содержит значения для ключа, то будет возвращено указанное значение по умолчанию. Вызов этой функции выглядит следующим образом:

Строковое значение = `map.getOrDefault(ключ, «значение по умолчанию»);`

Это позволяет вызывающей стороне решить, какое значение по умолчанию является подходящим, но не требует от вызывающей стороны обработки нулевого значения.

Значения по умолчанию могут сделать код (и программное обеспечение) намного проще в использовании, поэтому их стоит включить. Но стоит быть осторожным с тем, где они включены в код. Возврат значения по умолчанию делает предположение обо всех слоях кода выше, которые используют это значение, и, таким образом, может ограничить повторное использование кода и его адаптируемость. Возврат значений по умолчанию из низкоуровневого кода может быть особенно проблематичным. Часто бывает лучше просто вернуть `null` и реализовать значения по умолчанию на более высоком уровне, где предположения с большей вероятностью будут выполняться.

9.4. Сосредоточьтесь на функциональных параметрах

В главе 8 мы видели пример совместной инкапсуляции различных параметров стиля текста. Для этого мы определили класс `TextOptions` (повторяется в следующем листинге).

Листинг 9.16. Класс `TextOptions`

```
class TextOptions {
    private final Font font; ❶
    private final Double fontSize;
    private final Double lineHeight;
    private final Color textColor;

    TextOptions(
        Font font,
        Double fontSize,
        Double lineHeight,
        Color textColor) {
        this.font = font;
        this.fontSize = fontSize;
        this.lineHeight = lineHeight;
        this.textColor = textColor;
    }

    Font getFont() { return font; }
    Double getFontSize() { return fontSize; }
    Double getLineHeight() { return lineHeight; }
    Color getTextColor() { return textColor; }
}
```

❶ Объединяет несколько вариантов стилей вместе.

В сценариях, когда функции требуются все фрагменты информации, содержащиеся в объекте данных или классе, для этой функции имеет смысл использовать экземпляр объекта или класса в качестве параметра. Это сокращает количество параметров функций и избавляет промежуточный код от необходимости разбираться с деталями инкапсулированных данных. Но в сценариях, когда функции требуется только одна или две части информации, использование экземпляра объекта или класса в качестве параметра может повредить возможности повторного использования кода. В следующих подразделах объясняется почему и демонстрируется простая альтернатива.

9.4.1. Функцию, которая требует больше, чем ей нужно, может быть трудно повторно использовать

В листинге 9.17 показана часть кода виджета текстового поля, который можно использовать в пользовательском интерфейсе.

Вот некоторые вещи, на которые стоит обратить внимание в коде.

- ◆ Класс `TextBox` предоставляет две открытые функции: `setTextStyle()` и `setTextColor()`. Обе эти функции принимают в качестве параметра экземпляр `TextOptions`.
- ◆ Функция `setTextStyle()` использует всю информацию из `TextOptions`, поэтому для этой функции имеет смысл использовать это в качестве параметра.
- ◆ Функция `setTextColor()` использует только информацию о цвете текста из `TextOptions`. В этом смысле функция `setTextColor()` берет больше, чем ей нужно, потому что ей не нужны никакие другие значения в `TextOptions`.

В настоящее время функция `setTextColor()` вызывается только из функции `setTextStyle()`, так что это не вызывает особых проблем. Но если кто-то когда-нибудь захочет повторно использовать функцию `setTextColor()`, у него могут возникнуть трудности, как мы вскоре увидим.

Листинг 9.17. Функция, которая требует больше, чем ей нужно

```
class TextBox {
    private final Element textContainer;
    ...

    void setTextStyle(TextOptions options) {
        setFont(...);
        setFontSize(...);
        setLineHeight(...);
        setTextColor(options); ❶
    }

    void setTextColor(TextOptions options) {❷
        textContainer.setStyleProperty(
            "color", options.getTextColor().asHexRgb());❸
    }
}
```

❶ Вызывает функцию `SetTextColor()`.

❷ Принимает экземпляр `TextOptions` в качестве параметра.

❸ Использует только цвет текста.

Теперь представьте, что инженеру нужно реализовать функцию, которая стилизует `TextBox` как предупреждение. Требование для этой функции состоит в том, что она устанавливает красный цвет текста, но оставляет всю другую информацию о стиле без изменений. Инженер, скорее всего, захочет повторно использовать для этого функцию `TextBox.setTextColor()`, но, поскольку эта функция принимает в качестве параметра экземпляр `TextOptions`, это не так просто.

В листинге 9.18 показан код, который в итоге пишет инженер. Все что им нужно сделать — это установить красный цвет текста, но для этого им пришлось создать целый экземпляр `TextOptions` с различными нерелевантными выдуманными значениями. Этот код очень сбивает с толку: если мы взглянем на него, у нас может сложиться впечатление, что в дополнение к установке красного цвета он также устанавливает шрифт `Arial`, размер 12 и высоту строки 14. Это не так, но мы должны знать подробности о функции `TextBox.setTextColor()`, чтобы это было очевидно.

Листинг 9.18. Вызов функции, который занимает слишком много времени

```
void styleAsWarning(TextBox textBox) {  
    TextOptions style = new TextOptions(  
        Font.ARIAL, ❶  
        12.0,  
        14.0  
        Color.RED);  
    textBox.setTextColor(style);  
}
```

❶ Нерелевантные, выдуманные значения.

Весь смысл функции `TextBox.setTextColor()` в том, что она устанавливает только цвет текста. Поэтому нет необходимости принимать весь экземпляр `TextOptions` в качестве параметра. И, помимо того что он не нужен, он становится активно вредным, когда кто-то хочет повторно использовать функцию для немного другого сценария. Было бы лучше, если бы функция брала только то, что ей нужно.

9.4.2. Решение: заставьте функции принимать только то, что им нужно

Единственное, что функция `TextBox.setTextColor()` считывает из `TextOptions`, — это цвет текста. Таким образом, вместо того чтобы функция принимала весь экземпляр `TextOptions`, она может просто принимать экземпляр `Color` в качестве параметра. В следующем листинге показано, как выглядит код класса `TextBox` с этим изменением.

Листинг 9.19. Функция, которая принимает только то, что ей нужно

```
class TextBox {  
    private final Element textElement;  
    ...  
  
    void setTextStyle(TextOptions options) {  
        setFont(...);  
        setFontSize(...);  
        setLineHeight(...);  
    }  
}
```

```

        setTextColor(options.getTextColor()); ❶
    }
    void setTextColor(Color color) { ❷
        textElement.setStyleProperty("color", color.asHexRgb());
    }
}

```

❶ Вызывает `setTextColor()` только с цветом текста.

❷ Принимает экземпляр `Color` в качестве параметра.

Теперь функция `styleAsWarning()` стала намного проще и менее запутанной. Нет необходимости создавать экземпляр `TextOptions` с нерелевантными выдуманнами значениями:

```

void styleAsWarning(TextBox textBox) {
    textBox.setTextColor(Color.RED);
}

```

В общем, создание функций, которые берут только то, что им нужно, приводит к созданию кода, который более удобен для повторного использования и более понятен. Тем не менее все равно полезно применить свое суждение. Если у нас есть класс, который инкапсулирует десять вещей вместе, и функция, которой требуется восемь из них, то все равно может иметь смысл передать весь инкапсулирующий объект в функцию.

Альтернатива передачи восьми неинкапсулированных значений может нанести ущерб модульности (как мы видели в предыдущей главе). Как и во многих других вещах, нет единого ответа, применимого к любой ситуации, но полезно осознавать компромиссы, которые мы делаем, и последствия, которые они могут иметь.

9.5. Рассмотрите возможность использования универсальных типов

Классы часто содержат (или ссылаются на) экземпляры других типов или классов. Очевидным примером этого является класс `list`. Если у нас есть список строк, то класс `list` содержит экземпляры класса `string`. Хранение вещей в списке — это очень общая подзадача: в некоторых сценариях нам может потребоваться список строк, но в других сценариях нам может потребоваться список целых чисел. Было бы довольно досадно, если бы нам понадобились совершенно отдельные классы списков для хранения строк и целых чисел.

К счастью, многие языки поддерживают универсальные типы (иногда называемые шаблонами). Это позволяет нам писать класс без необходимости конкретно указывать все типы, на которые он ссылается. В случае списка это позволяет нам легко использовать один и тот же класс для хранения любого нужного нам типа.

Ниже приведены некоторые примеры использования списка для хранения различных типов.

```
List<String> stringList = ["hello", "world"];
```

```
List<Int> intList = [1, 2, 3];
```

Если мы пишем какой-то код, который ссылается на другой класс, но нас не особенно волнует, что это за другой класс, то это часто признак того, что нам следует рассмотреть возможность использования универсальных типов. Выполнение этого часто требует очень мало дополнительной работы, но делает наш код значительно более обобщаемым. В следующих подразделах представлен рабочий пример.

9.5.1. В зависимости от конкретного типа ограничивайте обобщаемость

Представьте, что мы создаем игру на угадывание слов. Каждая группа игроков представляет слова, а затем по очереди разыгрывает одно слово за раз, чтобы другие игроки могли его угадать. Одна из подзадач, которую нам нужно решить, — это хранение набора слов. Кроме того, нам нужно иметь возможность выбирать слова одно за другим случайным образом, а также иметь возможность возвращать слово в коллекцию, если его не удалось угадать за время каждого хода.

Мы решили, что можем решить эту подзадачу, внедрив рандомизированную очередь. В листинге 9.20 показан код класса `RandomizedQueue`, который мы реализуем. Он хранит коллекцию строк. Мы можем добавить новые строки, вызвав `add()`, и мы можем получить и удалить случайную строку из коллекции, вызвав `getNext()`. Класс `RandomizedQueue` имеет жесткую зависимость от `String` и поэтому никогда не может использоваться для хранения какого-либо другого типа.

Листинг 9.20. Жестко запрограммированное использование типа `String`

```
class RandomizedQueue {
    private final List<String> values = []; ❶

    void add(String value) {
        values.add(value);
    }

    /**
     * Удаляет случайный элемент из очереди и возвращает его.
     */
    String? getNext() {
        if (values.isEmpty()) {
```

```

        return null;
    }
    Int randomIndex = Math.randomInt(0, values.size());
    values.swap(randomIndex, values.size() - 1);
    return values.removeLast();
}
}

```

❶ Жестко закодированная зависимость от String.

Эта реализация `RandomizedQueue` решает наш очень специфический вариант хранения слов (которые могут быть представлены в виде строк), но она не обобщает для решения той же подзадачи для других типов. Представьте, что другая команда в нашей компании разрабатывает почти идентичную игру, где, вместо того чтобы игроки отправляли слова, они отправляли картинки. Многие из подзадач между двумя играми почти идентичны, но, поскольку мы жестко запрограммировали наши решения для использования строк, ни одно из них не обобщается для решения подзадач, с которыми сталкивается другая команда. Было бы намного лучше, если бы код обобщался для решения почти идентичных подзадач.

9.5.2. Решение: используйте универсальные типы

В случае с классом `RandomizedQueue` очень просто сделать код обобщаемым с помощью универсальных типов. Вместо того чтобы жестко запрограммировать зависимость от `String`, мы можем указать заполнитель (или шаблон) для типа, который затем может быть продиктован позже при использовании класса. В листинге 9.21 показано, как класс `RandomizedQueue` выглядит с использованием шаблонов. Определение класса начинается с класса `RandomizedQueue<T>`.

`<T>` сообщает компилятору, что мы будем использовать `T` в качестве заполнителя для типа. Затем мы можем использовать `T` во всем определении класса, как если бы это был реальный тип.

Листинг 9.21. Использование универсального типа

```

class RandomizedQueue<T> { ❶
    private final List<T> values = []; ❷

    void add(T value) {
        values.add(value);
    }

    /**
     * Removes a random item from the queue and returns it.
     */
    T? getNext() {
        if (values.isEmpty()) {

```

```

        return null;
    }
    int randomIndex = Math.randomInt(0, values.size());
    values.swap(randomIndex, values.size() - 1);
    return values.removeLast();
}
}

```

❶ Т указывается как заполнитель для универсального типа.

❷ Заполнитель типа можно использовать во всем классе.

Класс `RandomizedQueue` теперь можно использовать для хранения чего угодно, поэтому в нашей версии игры, в которой используются слова, мы можем определить класс для хранения строк следующим образом:

```
RandomizedQueue<String> words = new RandomizedQueue<String>();
```

Другая команда, которая хочет использовать его для хранения изображений, может легко определить его для хранения изображений следующим образом:

```
RandomizedQueue<Picture> pictures =
    new RandomizedQueue<Picture>();
```

УНИВЕРСАЛЬНЫЕ ТИПЫ И ТИПЫ, ДОПУСКАЮЩИЕ ЗНАЧЕНИЕ NULL

В листинге 9.21 функция `getNext()` возвращает `null`, если очередь пуста. Это нормально, если никто не хочет хранить нулевые значения в очереди, что, вероятно, является разумным предположением. (Хотя мы можем захотеть применить это предположение с помощью проверки или утверждения, как обсуждалось в *главе 3*.)

Если кто-то захочет сохранить нулевые значения в очереди, создав что-то вроде `RandomizedQueue<String?>`, это может быть проблематично. Это связано с тем, что было бы невозможно различить `getNext()`, возвращающий нулевое значение из очереди, и сигнал о том, что очередь пуста. Если бы мы действительно хотели поддерживать этот вариант использования, мы могли бы предоставить функцию `hasNext()`, которую можно вызывать для проверки, не пуста ли очередь перед вызовом `getNext()`.

Когда мы разбиваем проблему высокого уровня на подзадачи, мы часто сталкиваемся с такими из них, которые являются довольно фундаментальными и могут применяться ко всем видам различных вариантов использования. Когда решение подзадачи может быть легко применимо к любому типу данных, часто очень мало усилий надо для использования дженериков вместо зависимости от конкретного типа. Это может быть легкой победой с точки зрения того, чтобы сделать код более обобщаемым и пригодным для повторного использования.

Итоги

- ◆ Одни и те же подзадачи часто возникают снова и снова, поэтому многократное использование кода может сэкономить вам и вашим товарищам по команде значительное время и усилия в будущем.

- ◆ Постарайтесь определить основные подзадачи и структурировать код таким образом, чтобы другие могли повторно использовать решения конкретных подзадач, даже если они решают другую проблему высокого уровня.
- ◆ Создание чистых уровней абстракции и модульность кода часто приводят к тому, что код намного проще и безопаснее использовать повторно и обобщать.
- ◆ Принятие предположения часто сопряжено с издержками с точки зрения того, что код становится более хрупким и менее пригодным для повторного использования.
 - Убедитесь, что преимущества предположения перевешивают затраты.
 - Если необходимо сделать предположение, убедитесь, что оно находится в соответствующем слое кода, и, если возможно, примените его.
- ◆ Использование глобального состояния часто приводит к особенно дорогостоящему допущению, которое приводит к коду, который совершенно небезопасен для повторного использования. В большинстве сценариев глобального состояния лучше избегать.

Часть 3

Модульное тестирование

Тестирование является неотъемлемой частью создания кода и программного обеспечения, которые работают правильно (и продолжают работать правильно). Как обсуждалось в *главе 1*, существуют различные уровни тестирования, но модульное тестирование обычно является тем, с чем инженеры чаще всего взаимодействуют в повседневной жизни. Эта часть, посвященная модульному тестированию, размещена в конце книги, но, пожалуйста, не делайте из этого вывод, что модульное тестирование — это дополнительное соображение, которое следует учитывать только после написания кода. Как мы видели в предыдущих главах, тестирование и тестируемость — это вещи, которые нам часто приходится учитывать при написании кода.

И, как мы увидим в *главе 10*, некоторые школы доходят до того, что выступают за то, чтобы тесты писались до написания кода. Эта часть книги разделена на две главы. В *главе 10* рассматриваются некоторые основополагающие принципы модульного тестирования: чего мы пытаемся достичь и некоторые фундаментальные концепции, такие, как имитированные реализации. В *главе 11* этот вопрос расширяется с помощью ряда более практических соображений и методов, которые могут помочь нам достичь целей, определенных в *главе 10*.

10 Принципы модульного тестирования

ЭТА ГЛАВА ОХВАТЫВАЕТ

- ◇ Основы модульного тестирования
- ◇ Что делает хороший модульный тест хорошим
- ◇ Имитированные реализации, в том числе когда и как их использовать
- ◇ Философии тестирования

Каждый раз, когда инженер изменяет строку кода, существует риск того, что он может непреднамеренно что-то сломать или совершить ошибку. Даже очень маленькие, невинные на вид изменения могут иметь плохие последствия: «Это всего лишь изменение одной строки» — знаменитые последние слова перед сбоем системы. Поскольку каждое изменение сопряжено с риском, нам нужен способ убедиться в том, что код работает, как изначально, так и всякий раз, когда он изменяется. Тесты часто являются главным, что дает нам эту уверенность.

Как инженеры, мы обычно концентрируемся на написании автоматизированных тестов. Это означает, что мы пишем тестовый код, который использует «реальный» код, чтобы проверить, правильно ли он работает. В *главе 1* описано, что существуют различные уровни тестирования и, в частности, что модульное тестирование — это уровень тестирования, с которым инженеры чаще всего сталкиваются в своем повседневном программировании. Вот почему в этих двух заключительных главах мы сосредоточимся на модульном тестировании.

На этом этапе, вероятно, было бы полезно дать точное определение того, что именно мы подразумеваем под *модульным тестированием*. Но, к сожалению, это не совсем определенный термин. Модульное тестирование связано с тестированием отдельных блоков кода относительно изолированным образом. Что именно мы подразумеваем под *единицей кода*, может варьироваться, но часто это относится к определенному классу, функции или файлу кода. То, что мы подразумеваем под относительно изолированным образом, также может варьироваться и быть открытым для интерпретации. Большинство фрагментов кода не живут изолированно; они зависят от ряда других фрагментов кода. Как мы увидим в *разд. 10.4.6*, некоторые инженеры стараются изолировать код от его зависимостей в модульных тестах, в то время как другие предпочитают включать их.

Модульное тестирование может не быть точно определенным термином, но обычно это не слишком большая проблема. Лучше не слишком заикливаться на том, что именно представляет собой модульный тест и точно ли тесты, которые мы пишем, соответствуют его надуманному определению. В конечном счете, важно то, что мы гарантируем, что наш код хорошо протестирован и что мы делаем это таким образом, чтобы его можно было поддерживать. В этой главе рассматриваются некоторые ключевые принципы модульного тестирования, которые могут помочь нам достичь этой цели. Глава 11 будет основана на этом, чтобы обсудить ряд практических методов.

10.1. Учебник по модульному тестированию

Если вы никогда не писали программное обеспечение в профессиональной среде, то, возможно, никогда раньше не сталкивались с модульным тестированием. Если это так, то в этом разделе вы найдете важные детали, которые понадобятся для того, чтобы эта глава и следующая имели смысл.

Когда дело доходит до модульного тестирования, необходимо запомнить некоторые важные понятия и терминологию.

- ◆ *Тестируемый код*, иногда называемый «реальным кодом». Относится к фрагменту кода, который мы пытаемся протестировать.
- ◆ *Тестовый код* — относится к коду, который формирует наши модульные тесты. Тестовый код обычно находится в файле, отдельном от «реального кода», но часто существует однозначное сопоставление между файлами реального кода и файлами тестового кода, поэтому если у нас есть некоторый реальный код в файле под названием `guestlist.lang`, мы можем поместить код модульного теста в файл под названием `GuestListTest.lang`. Иногда реальный код и тестовый код хранятся рядом друг с другом в одном каталоге, а иногда тестовый код хранится в совершенно другой части кодовой базы. Это варьируется от языка к языку и от команды к команде.
- ◆ *Контрольный пример*. Каждый файл тестового кода обычно делится на несколько контрольных примеров, где каждый контрольный пример проверяет определенное поведение или сценарий. На практическом уровне контрольный пример обычно представляет собой просто функцию, и для всего, кроме простейших контрольных примеров, принято разделять код внутри каждого из них на три отдельных раздела следующим образом:
 - *упорядочивание* — часто необходимо выполнить некоторую настройку, прежде чем мы сможем вызвать конкретное поведение, которое мы хотим протестировать. Это может включать определение некоторых тестовых значений, настройку некоторых зависимостей или создание правильно настроенного экземпляра тестируемого кода (если это класс). Это часто помещается в отдельный блок кода в начале контрольного примера;

- *действие* — это относится к строкам кода, которые фактически вызывают тестируемое поведение. Обычно это включает вызов одной или нескольких функций, предоставляемых тестируемым кодом;
- *утверждение* — как только тестируемое поведение было вызвано, тест должен проверить, действительно ли произошли правильные вещи. Обычно это включает проверку того, что возвращаемое значение равно ожидаемому значению или что некоторое результирующее состояние соответствует ожиданиям.



Рис. 10.1. Как различные концепции модульного тестирования сочетаются друг с другом

- ♦ *Исполнитель тестов.* Как следует из названия, исполнитель тестов — это инструмент, который фактически выполняет тесты. Получив файл тестового кода (или несколько файлов тестового кода), он запустит каждый контрольный пример и выведет подробную информацию о том, какой тест прошел, а какой не прошел.

На рис. 10.1 показано, как некоторые из этих концепций сочетаются друг с другом.

Дано, когда, тогда

Некоторые инженеры предпочитают термины «*дано*», «*когда*» и «*тогда*», чем «*упорядочивание*», «*действие*» и «*утверждение*». Существуют некоторые нюансы, вокруг которых философии тестирования отстаивают эти разные наборы терминов, но в контексте кода в контрольном примере они эквивалентны.

О важности тестирования говорят так часто, что это может начать звучать как клише. Клише или нет, тестирование важно. В настоящее время в большинстве профессиональных сред разработки программного обеспечения почти каждый фрагмент «реального кода» должен сопровождаться модульным тестом. И ожидается, что каждое поведение, демонстрируемое «настоящим кодом», будет сопровождаться контрольным примером. Это идеал, и это то, к чему мы должны стремиться.

Вы быстро обнаружите, что не каждая часть существующего кода соответствует этому идеалу, а в некоторых кодовых базах тестирование может быть особенно плохим. Но это не оправдание для снижения наших собственных стандартов от идеала. Плохое или неадекватное тестирование часто является несчастным случаем. Большинство инженеров, занимающихся созданием программного обеспечения в течение некоторого времени, вероятно, могут рассказать несколько ужасных историй, связанных с плохим тестированием.

Недостаток тестов — самый очевидный, но не единственный способ плохого тестирования. Чтобы иметь хорошее тестирование, нам нужны не просто тесты, нам нужны хорошие тесты. Следующий раздел определяет, что мы подразумеваем под этим.

10.2. Что делает модульный тест хорошим?

На первый взгляд, модульное тестирование может показаться довольно простым: нам просто нужно написать тестовый код, чтобы убедиться, что реальный код работает. К сожалению, это обманчиво, и с годами многие инженеры поняли, как легко ошибиться в модульном тестировании. Когда с модульным тестированием что-то идет не так, это может привести к тому, что код будет очень сложно поддерживать, а ошибки останутся незамеченными. Поэтому важно подумать о том, что делает модульный тест хорошим. Для этого мы определим пять ключевых особенностей, которыми должен обладать хороший модульный тест.

- ♦ *Точно обнаруживает поломки* — если код неисправен, тест должен завершиться неудачно. И тест должен завершиться неудачно только в том случае, если код действительно сломан (нам не нужны ложные тревоги).

- ◆ *Независим от деталей реализации.* Изменения в деталях реализации в идеале не должны приводить к изменениям в тестах.
- ◆ *Хорошо объясненные сбои.* Если код неисправен, сбой теста должен дать четкое объяснение проблемы.
- ◆ *Понятный тестовый код.* Другие инженеры должны понимать, что именно тестирует тест и как он это делает.
- ◆ *Легко и быстро запускаем* — инженерам обычно приходится довольно часто запускать модульные тесты в своей повседневной работе. Медленный или сложный модульный тест отнимет у разработчиков много времени.

В следующих подразделах эти цели рассматриваются более подробно.

10.2.1. Точное обнаружение поломок

Основная и наиболее очевидная цель модульных тестов — убедиться, что код не нарушен: он делает то, для чего предназначен, и не содержит ошибок. Если тестируемый код каким-либо образом нарушен, он либо не должен компилироваться, либо тест должен завершиться неудачей. Это играет две очень важные роли:

- ◆ *дает нам первоначальную уверенность в коде.* Независимо от того насколько тщательно мы пишем код, почти невозможно избежать некоторого количества ошибок. Написав тщательный набор тестов вместе с любым новым кодом или изменениями кода, которые мы создаем, вполне вероятно, что мы обнаружим и исправим многие из этих ошибок еще до того, как код будет отправлен в кодовую базу;
- ◆ *защищает от поломки в будущем.* В главе 3 обсуждалось, что кодовая база часто бывает занята, когда несколько инженеров постоянно вносят изменения. Весьма вероятно, что в какой-то момент другой инженер внесет изменение, которое непреднамеренно нарушит наш код. Наша единственная эффективная защита от этого состоит в том, чтобы гарантировать, что либо код перестанет компилироваться, либо тест начнет терпеть неудачу, когда это произойдет. Невозможно спроектировать все так, чтобы код переставал компилироваться, когда что-то ломается, поэтому абсолютно необходимо убедиться, что все правильное поведение зафиксировано с помощью тестов. Часть функциональности, нарушенная изменением кода (или каким-либо другим событием), называется *регрессией*. Выполнение тестов с целью обнаружения любых таких регрессий называется *регрессионным тестированием*.

Также важно учитывать еще один аспект точности: тест должен провалиться только в том случае, если тестируемый код действительно неисправен. Может показаться, что это естественно следует из того, что мы только что обсудили, но на практике это часто не так. Любой, кто имеет опыт логических ошибок, поймет, что «тест обязательно провалится, если код неисправен» не обязательно означает, что «тест провалится, только если код неисправен».

Тест, который иногда проходит, а иногда терпит неудачу, несмотря на то что тестируемый код в порядке, называется *ненадежным*. Обычно это результат неопределенного поведения в тесте, такого, как случайность, условия гонки на основе времени или зависимость от внешней системы. Наиболее очевидным недостатком ненадежных тестов является то, что они тратят время инженеров, потому что в конечном итоге им приходится расследовать сбои, которые на самом деле ничего не значат. Но ненадежные тесты на самом деле намного опаснее, чем может показаться на первый взгляд. Любой, кто знаком с басней о мальчике, который кричал «Волк!», поймет почему: если тест продолжает выдавать ложную тревогу о том, что код взломан, инженеры научатся его игнорировать. Они могут даже отключить тест, если он действительно раздражает. Если на провалы тестов уже никто не обращает внимания, то это ничем не отличается от ситуации, когда тестов нет. Тогда существует очень слабая защита от будущих поломок, и вероятность появления ошибок становится высокой. Невероятно важно добиться того, чтобы тесты терпели неудачу, когда что-то сломано, и только тогда, когда что-то сломано.

10.2.2. Не зависит от деталей реализации

Грубо говоря, существует два вида изменений, которые инженер может внести в кодовую базу.

- ◆ *Функциональное изменение* — изменяет внешне видимое поведение фрагмента кода. Примерами этого являются добавление новой функции, исправление ошибки или другая обработка сценария ошибки.
- ◆ *Рефакторинг* — это структурное изменение кода, такое, как разделение большой функции на более мелкие или перемещение некоторого служебного кода из одного файла в другой, чтобы его было легче использовать повторно. Теоретически, если рефакторинг выполнен правильно, он не должен изменять какое-либо внешне видимое поведение (или функциональные атрибуты) кода.

Первое из них (функциональное изменение) в значительной степени затрагивает всех, кто использует наш код, и поэтому необходимо внимательно рассмотреть всех вызывающих код, прежде чем вносить такого рода изменения. Поскольку функциональное изменение изменяет поведение нашего кода, мы надеемся и ожидаем, что оно также потребует модификации тестов. Если это не так, вероятно, наши первоначальные тесты были недостаточными.

Второй из них (рефакторинг) — это то, что не должно затрагивать никого, кто использует наш код. Мы меняем детали реализации, но не поведение, о котором должны заботиться все остальные. Однако изменение кода всегда сопряжено с риском, и рефакторинг не является исключением. Наше намерение состоит в том, чтобы изменить только структуру кода, но как мы можем быть уверены, что не изменяем непреднамеренно поведение кода в процессе?

Чтобы ответить на этот вопрос, давайте рассмотрим два подхода, которые мы могли использовать, когда писали исходные модульные тесты для нашего кода (задолго до того, как решили провести этот рефакторинг):

- ◆ *Подход А.* Помимо фиксации всех вариантов поведения нашего кода, тесты также фиксируют различные детали реализации. Мы тестируем несколько закрытых функций, делая их видимыми для тестов, моделируем состояние, напрямую манипулируя закрытыми переменными-членами и зависимостями, а также проверяем состояние различных переменных-членов после запуска тестируемого кода.
- ◆ *Подход Б.* Наши тесты фиксируют все варианты поведения, но не детали реализации. Мы стремимся использовать общедоступный API кода для настройки состояния и проверки поведения везде, где это возможно. И мы никогда ничем не манипулируем и ничего не проверяем, используя приватные переменные или функции.

Теперь давайте рассмотрим, что произойдет, если мы придем через несколько месяцев и проведем рефакторинг кода. Если мы выполним рефакторинг правильно, то должны быть изменены только детали реализации, и никакие внешне видимые модели поведения не должны быть затронуты. Если затронут внешне видимое поведение, значит, мы совершили ошибку. Давайте рассмотрим, что произойдет, если мы будем использовать разные подходы к тестированию:

- ◆ *Подход А.* Независимо от того правильно ли мы выполнили рефакторинг, тесты начнут давать сбои, и нам нужно будет внести в них множество изменений, чтобы они снова прошли. Теперь нам нужно протестировать различные частные функции, установить состояние в различных частных переменных-членах и зависимостях и проверить другой набор переменных-членов после запуска тестируемого кода.
- ◆ *Подход Б.* Если мы правильно провели рефакторинг, тесты все равно должны пройти (без необходимости их модификации). Если тест не проходит, то мы явно допустили ошибку, потому что это означает, что мы непреднамеренно изменили внешне видимое поведение.

При подходе А очень сложно быть уверенным в том, что мы допустили ошибку при рефакторинге кода. Тесты терпят неудачу и нуждаются в модификации в любом случае, и выяснить, какие из этих модификаций тестов следует ожидать, а какие нет, вероятно, непростая задача. С подходом В невероятно легко быть уверенным в нашем рефакторинге: если тесты все еще проходят, значит, все в порядке; если тест не проходит, значит, мы допустили ошибку.

НЕ СМЕШИВАЙТЕ ФУНКЦИОНАЛЬНЫЕ ИЗМЕНЕНИЯ И РЕФАКТОРИНГ

При внесении изменений в кодовую базу обычно лучше либо внести функциональные изменения, либо провести рефакторинг, но не делать и то, и другое одновременно.

Рефакторинг не должен изменять какое-либо поведение, тогда как функциональные изменения должны. Если мы вносим и функциональное изменение, и рефакторинг одновременно, трудно определить, какие изменения в поведении являются ожидаемыми от функционального изменения, а какие могут быть связаны с ошибкой, которую мы

допустили при рефакторинге. Обычно лучше провести рефакторинг, а затем внести функциональные изменения отдельно. Это значительно упрощает выявление причины любых потенциальных проблем.

Код очень часто подвергается рефакторингу. В зрелых кодовых базах объем рефакторинга часто может превышать объем написанного нового кода, поэтому обеспечение того чтобы код не ломался при рефакторинге, имеет первостепенное значение. Делая тесты независимыми от деталей реализации, мы можем гарантировать, что существует надежный и чистый сигнал, который любой рефакторинг кода может использовать, чтобы увидеть, допустил ли он ошибку.

10.2.3. Хорошо объяснимые сбои

Как мы видели выше, одной из основных целей тестов является защита от поломок в будущем. Распространенным сценарием является то, что другой инженер вносит изменение, которое непреднамеренно нарушает чужой код. Затем тест начинает давать сбой, что предупреждает инженера о том, что он что-то сломал. Затем этот инженер пойдет и посмотрит на провал теста, чтобы выяснить, что не так. Инженер может быть совершенно не знаком с кодом, который он непреднамеренно сломал, поэтому, если сбой теста не указывает, что сломано, он, вероятно, потратит много времени, пытаясь понять это.

Плохо объясненный провал теста

Имя контрольного примера не указывает, какое поведение тестируется

```
Сбой контрольного примера testGetEvents:  
Ожидается: [Event @ea4a92b, Event @3c5a99da]  
А на самом деле было: [Event@3c5a99da, Event@ea4a92b]
```

Сообщение об ошибке трудно расшифровать

Хорошо объяснимый провал теста

Название контрольного примера дает понять, какое поведение тестируется

```
Сбой контрольного примера testGetEvents_inChronologicalOrder:  
Содержимое совпадает, но порядок отличается  
Ожидается:  
[<Космический полет, 12 апреля 1961 г.>, <Высадка на Луну, 20 июля 1969 г.>]  
Но на самом деле было:  
[<Посадка на Луну, 20 июля 1969 г.>, <Космический полет, 12 апреля 1961 г.>]
```

Сообщение об ошибке понятно

Рис. 10.2. Неудачный тест, который четко объясняет, что не так, гораздо полезнее, чем тот, который просто указывает на то, что что-то не так

Чтобы тесты четко и точно объясняли, что сломано, необходимо подумать о том, какое сообщение о сбое выдаст тест, когда что-то пойдет не так, и будет ли это полезно другому инженеру. На рис. 10.2 показаны два возможных сообщения об ошибке, которые мы можем увидеть, когда тест не пройден. Первый из этих флагов сообщает, что что-то не так с получением событий, но не дает нам абсолютно никакой информации о том, что именно не так. Второе сообщение, с другой стороны, дает довольно четкое описание того, что не так. Мы видим, что проблема в том, что события не возвращаются в хронологическом порядке.

Один из лучших способов убедиться в том, что сбой в тестировании хорошо объяснен, — это тестировать по одной вещи за раз и использовать описательные имена для каждого контрольного примера. Это часто приводит к множеству небольших контрольных примеров, каждый из которых фиксирует одно конкретное поведение, а не один большой контрольный пример, который пытается протестировать все за один раз. Когда тест начинает терпеть неудачу, довольно легко точно определить, какое поведение было нарушено, проверив имена контрольных примеров, которые завершились неудачей.

10.2.4. Понятный тестовый код

До этого момента мы предполагали, что сбой теста указывает на то, что код сломан. Но это не совсем так; если быть более точным, сбой теста указывает на то, что код теперь ведет себя по-другому. Действительно ли тот факт, что он ведет себя по-другому, означает, что он сломан (или нет), зависит от обстоятельств. Другой инженер может намеренно изменять функциональность кода, например, в соответствии с новыми требованиями. В этом случае изменение поведения является преднамеренным.

Инженер, вносящий это изменение, очевидно, должен быть осторожен, но как только он выполнит свою работу и убедится, что изменение безопасно, ему необходимо будет обновить тесты, чтобы отразить новую функциональность. Как мы видели ранее, изменение кода сопряжено с риском, и это также относится к самому тестовому коду. Допустим, фрагмент кода имеет три варианта поведения, которые фиксируются тестами. Если инженер вносит преднамеренные изменения только в одно из этих поведений, то в идеале ему необходимо вносить изменения только в контрольные примеры, которые проверяют это поведение. Контрольные примеры, которые проверяют два других поведения, в идеале должны быть оставлены нетронутыми.

Чтобы инженер был уверен в том, что его изменение влияет только на желаемое поведение, он должен знать, на какие части теста оно влияет, и ожидается ли, что они нуждаются в обновлении. Вот почему инженеры должны понимать тесты — и то, *что* тестируют различные контрольные примеры, и то, *как* они их тестируют.

Как мы увидим в следующей главе, два наиболее распространенных способа, которыми это может пойти не так, — это тестирование слишком большого количества вещей одновременно и использование слишком большого количества общих настроек тестирования. И то, и другое может привести к тестам, которые очень труд-

но понять и обдумать. Это делает будущие модификации тестируемого кода гораздо менее безопасными, потому что инженерам будет сложно понять, безопасны ли конкретные изменения, которые они вносят.

Еще одна причина стремиться сделать тестовый код понятным заключается в том, что некоторым инженерам нравится использовать тесты в качестве своего рода руководства по эксплуатации кода. Если им интересно, как использовать конкретный фрагмент кода или какие функции он предоставляет, то чтение модульных тестов может быть хорошим способом выяснить это. Если тесты трудны для понимания, из них не получится очень полезного руководства по эксплуатации.

10.2.5. Легкий и быстрый запуск

Большинство модульных тестов выполняются довольно часто. Одной из наиболее важных функций модульного теста является предотвращение отправки поврежденного кода в кодовую базу. Поэтому во многих кодовых базах будут использоваться предварительные проверки, которые гарантируют, что все соответствующие тесты будут пройдены до того, как изменение может быть отправлено. Если выполнение модульных тестов займет час, это замедлит работу каждого инженера, потому что отправка изменения кода займет минимум час, независимо от того, насколько оно маленькое или тривиальное. В дополнение к запуску перед внесением изменений в кодовую базу инженеры часто многократно выполняют модульные тесты во время разработки кода, так что это еще один способ замедления модульных тестов, замедляющих работу инженеров.

Еще одна причина, по которой тесты должны выполняться быстро и легко, заключается в максимизации вероятности того, что инженеры действительно проверят материал. Когда тесты идут медленно, тестирование становится болезненным, а если тестирование болезненное, инженеры, как правило, делают его меньше. Вероятно, это не то, что многие уважающие себя инженеры с готовностью признали бы, но, судя по опыту, это реальность. Максимально простое и быстрое выполнение тестов не только повышает эффективность работы инженеров, но и приводит к более обширному и тщательному тестированию.

10.3. Сосредоточьтесь на общедоступном API, но не игнорируйте важные параметры поведения

Мы только что обсудили, почему важно, чтобы модульные тесты не зависели от деталей реализации. В *главе 2* говорилось, что различные аспекты фрагмента кода можно разделить на две отдельные части: общедоступный API и детали реализации. Если одна из наших целей состоит в том, чтобы избежать тестирования деталей реализации, это означает, что мы должны попытаться протестировать фрагмент кода, используя только его общедоступный API. «Тестируйте, используя только общедоступный API» — это, по сути, очень распространенный совет в отношении модульного тестирования. Если у вас уже имеются некоторые знания

об этом, то вы, скорее всего, слышали его раньше. Сосредоточение внимания на общедоступном API заставляет нас сконцентрироваться на поведении, которое, в конечном счете, волнует пользователей части кода, а не на деталях, которые являются просто средством достижения цели. Это помогает гарантировать, что мы тестируем то, что действительно имеет значение, и в процессе также стараемся, чтобы тесты не зависели от деталей реализации.

Чтобы продемонстрировать преимущества использования общедоступного API при тестировании, рассмотрим функцию в следующем фрагменте кода для расчета кинетической энергии (в джоулях). Любого, кто вызывает эту функцию, заботит то, что она возвращает правильное значение для заданной массы (в килограммах) и скорости (в метрах в секунду). Тот факт, что функция вызывает `Math.pow()`, является деталью реализации. Мы могли бы заменить `Math.pow(speedMs, 2.0)` на `speedMs * speedMs`, и функция будет вести себя точно так же, если кто-то ее вызывает:

```
Double calculateKineticEnergyJ(Double massKg, Double speedMs) {  
    return 0.5 * massKg * Math.pow(speedMs, 2.0);  
}
```

Концентрируясь на общедоступном API, мы вынуждены писать тесты, которые фиксируют поведение, которое на самом деле волнует вызывающих. Поэтому мы могли бы написать серию тестов, которые проверяют, что ожидаемое значение возвращается для заданных входных данных. В следующем фрагменте показан один из таких тестов. *(Обратите внимание: поскольку возвращаемое значение является двойным, мы проверяем, находится ли оно в определенном диапазоне, а не проверяем точное равенство.)*

```
void testCalculateKineticEnergy_correctValueReturned() {  
    assertThat(calculateKineticEnergyJ(3.0, 7.0)) ❶  
        .isWithin(1.0e-10)  
        .of(73.5);  
}
```

❶ Утверждает, что значение находится в пределах 0,0000000001 от 73,5.

Если бы у нас возникло искушение написать тест, который проверял бы, что функция `calculateKineticEnergyJ()` вызывает `Math.pow()`, то принцип «тестирование с использованием только общедоступного API» увел бы нас от этого. Это не позволяет нам связывать наши тесты с деталями реализации и гарантирует, что мы сосредоточимся на тестировании вещей, которые действительно важны для вызывающих. На таком простом примере все кажется довольно ясным. Но все может стать более запутанным, когда у нас есть более сложные фрагменты кода для тестирования.

Важные действия могут находиться за пределами общедоступного API

Функция `calculateKineticEnergyJ()`, которую мы только что видели, вполне автономна. Единственные входные данные, которые она принимает, — это параметры, и единственный эффект, который она имеет, — это возврат ответа. На самом деле

код редко бывает настолько автономным. Он часто зависит от множества других фрагментов кода, и тестирование может стать более тонким, если некоторые из этих зависимостей предоставляют внешние входные данные для кода или если код вызывает в них побочные эффекты.

В таких сценариях то, что именно подразумевается под «общедоступным API», может быть субъективным, и я сталкивался с ситуациями, когда инженеры цитировали «тестировать, используя только общедоступный API» в качестве оправдания того, что не тестировали важные функции. Их аргумент заключается в том, что если поведение нельзя запустить или проверить с помощью того, что они считают общедоступным API, то его не следует тестировать. Именно здесь становится важным использовать здравый смысл и мыслить прагматично.



Рис. 10.3. Автомат по продаже кофе имеет общедоступный API, но мы не можем полностью протестировать автомат, используя только общедоступный API

Определение *деталей реализации*, приведенное в *главе 2*, слишком упрощено, когда дело доходит до модульного тестирования. Является ли что-то деталью реализации или нет, на самом деле в некоторой степени зависит от контекста. В *главе 2* это обсуждалось с точки зрения уровней абстракции, где фрагменты кода зависят друг от друга. В этом сценарии все, что нужно знать одному фрагменту кода о другом, — это то, что содержится в общедоступном API, поэтому все остальное — это детали реализации. Но когда дело доходит до тестирования, могут быть и другие вещи, о которых необходимо знать тестовому коду, которые не считаются частью общедоступного API. Чтобы лучше объяснить это, давайте рассмотрим аналогию.

Представьте, что мы работаем в компании, которая управляет сетью автоматов по продаже кофе. На рис. 10.3 показана одна из моделей машин, которые произво-

дит и внедряет наша компания. Наша задача — протестировать его, чтобы убедиться, что он работает правильно. То что представляет собой общедоступный API этой машины, открыто для некоторой интерпретации, но инженер может определить его как «способ, которым клиент, покупающий чашку кофе, должен взаимодействовать с машиной». Если мы примем это определение, то общедоступный API довольно прост: клиент прикладывает свою кредитную карту к считывателю, выбирает, какой напиток он хотел бы, и автомат возвращает выбранный напиток в чашке. Существуют также некоторые сценарии ошибок, о которых общедоступный API может сообщить клиенту, например, о том, что его кредитная карта отклонена или машина не работает.

На первый взгляд кажется, что мы можем протестировать основные функции торгового автомата, используя то, что мы определили как общедоступный API: оплата и выбор напитка, а также проверка того, что автомат возвращает правильный выбор. Но это не совсем так, и с нашей точки зрения, как тестировщиков машины, нам нужно учитывать нечто большее, чем просто общедоступный API. Для начала у торгового автомата есть некоторые зависимости, которые нам нужно настроить. Мы не можем протестировать машину, пока не подключим ее к розетке, не наполним резервуар для воды и не положим немного кофейных зерен в емкость для зерен. Для заказчика все эти вещи являются деталями реализации, но для нас, тестировщиков, нет никакого реального способа протестировать машину без предварительной настройки этих вещей.

Также могут быть модели поведения, которые нам необходимо протестировать, которые не являются частью общедоступного API и которые клиент рассмотрит в деталях реализации. Этот торговый автомат является «умным» торговым автоматом. Он подключен к Интернету и автоматически уведомляет техника, когда вода или кофейные зерна заканчиваются. (Это пример преднамеренного побочного эффекта, который может вызвать торговый автомат.) Клиенты, вероятно, не знают об этой функции, и даже если бы знали, то сочли бы ее деталью реализации. Но, тем не менее, это важное поведение, которое демонстрирует торговый автомат, и поэтому нам необходимо его протестировать.

С другой стороны, есть много вещей, которые определенно являются деталями реализации как для клиентов, так и для нас, тестировщиков. Примером этого является то, как машина нагревает воду для приготовления кофе: использует ли она термоблок или бойлер? Это не то, что мы должны проверять, потому что это внутренняя деталь машины, которая не имеет прямого значения. Знаток кофе может возразить, что это действительно имеет значение, потому что бойлер производит кофе лучшего вкуса. Но если мы распакуем их аргумент, это все равно говорит о том, что способ нагрева воды является деталью реализации. То что в конечном счете волнует знатока, — это вкус кофе, а метод нагрева воды — всего лишь средство для достижения этой цели, поэтому, если мы беспокоимся о том, что знатоки жалуются, мы должны убедиться, что вкус кофе — это то, что мы тестируем (а не метод нагрева воды). На рис. 10.4 показаны различные зависимости, которые есть у торгового автомата, и то, как тесты могут потребоваться для взаимодействия с ними.

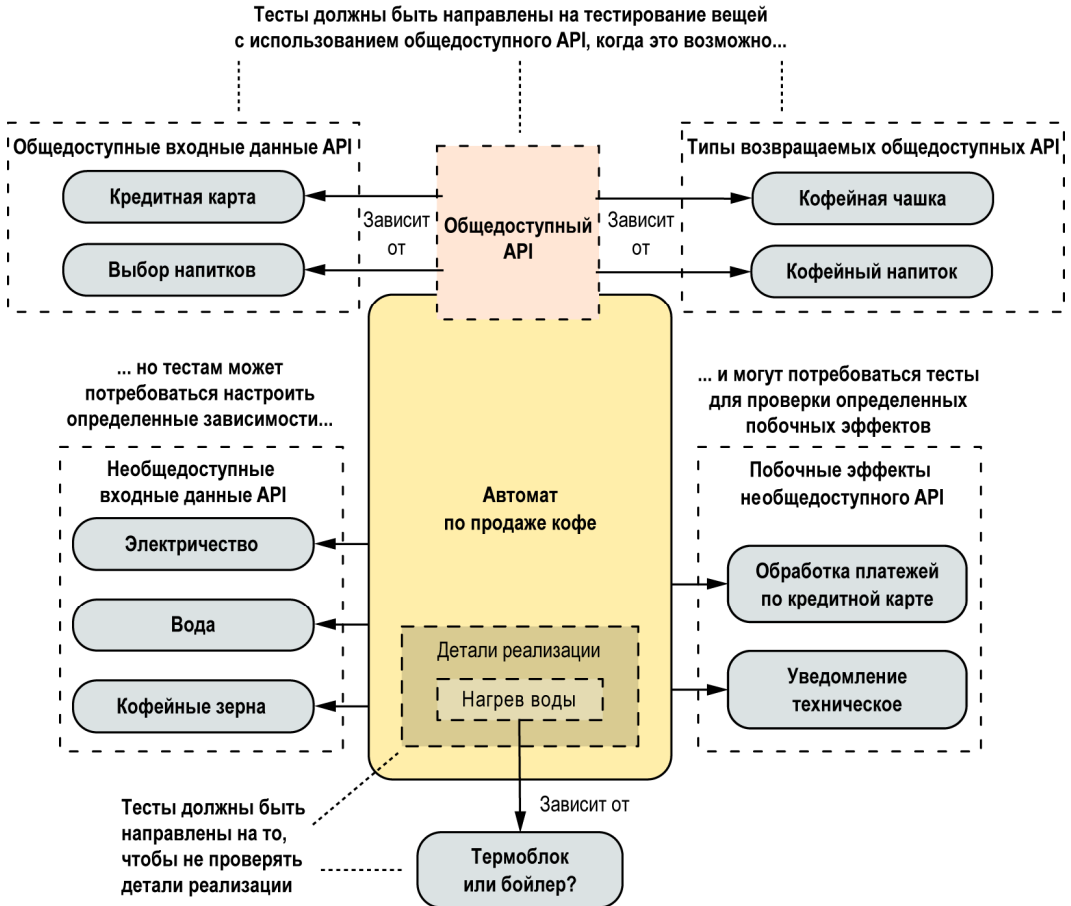


Рис. 10.4. Тесты должны быть направлены на тестирование с использованием общедоступного API, когда это возможно. Но часто бывает необходимо, чтобы тесты взаимодействовали с зависимостями, которые не являются частью общедоступного API, для выполнения настройки и проверки желаемых побочных эффектов

Тестирование торгового автомата аналогично модульному тестированию фрагмента кода. Чтобы продемонстрировать это на примере, рассмотрим листинг 10.1. Класс `AddressBook` позволяет вызывающей стороне искать адрес электронной почты пользователя. Это достигается путем получения адреса электронной почты с сервера. Он также кеширует любые ранее извлеченные адреса электронной почты, чтобы предотвратить перегрузку сервера повторными запросами. Что касается любого, кто использует этот класс, то они вызывают `lookupEmailAddress()` с идентификатором пользователя и возвращают адрес электронной почты (или `null`, если адреса электронной почты нет), поэтому было бы разумно сказать, что функция `lookupEmailAddress()` — общедоступный API этого класса. Это означает, что тот факт, что он зависит от класса `ServerEndPoint` и кеширует адреса электронной почты, является деталью реализации для пользователей класса.

Листинг 10.1. Класс AddressBook

```
class AddressBook {  
    private final ServerEndPoint server; ❶  
    private final Map<Int, String> emailAddressCache;  
    ...  
  
    String? lookupEmailAddress(Int userId) { ❷  
        String? cachedEmail = emailAddressCache.get(userId);  
        if (cachedEmail != null) {  
            return cachedEmail;  
        }  
        return fetchAndCacheEmailAddress(userId);  
    }  
    private String? fetchAndCacheEmailAddress(Int userId) { ❸  
        String? fetchedEmail = server.fetchEmailAddress(userId);  
        if (fetchedEmail != null) {  
            emailAddressCache.put(userId, fetchedEmail);  
        }  
        return fetchedEmail;  
    }  
}
```

❶ Детали реализации в отношении пользователей класса.

❷ Общедоступный API.

❸ Подробности деталей реализации.

Общедоступный API отражает наиболее важное поведение класса: поиск адреса электронной почты с заданным идентификатором пользователя. Но мы не можем это проверить, пока не настроим (или не смоделируем) `ServerEndPoint`. В дополнение к этому еще одним важным поведением является то, что повторные вызовы `lookupEmailAddress()` с одним и тем же идентификатором пользователя не приводят к повторным обращениям к серверу. Это не часть общедоступного API (как мы его определили), но все же это важное поведение, потому что мы не хотим, чтобы наш сервер был перегружен, и поэтому мы должны протестировать его. Обратите внимание, что на самом деле нас волнует (и нужно проверить) то, что повторные запросы не отправляются на сервер. Тот факт, что класс достигает этого с помощью кеша, является лишь средством для достижения этой цели и, следовательно, является деталью реализации даже для тестов. На рис. 10.5 показаны зависимости класса `AddressBook` и способы взаимодействия с ними тестов.

По возможности мы должны тестировать поведение кода с помощью общедоступного API. Это, вероятно, применимо к любому варианту поведения, которое происходит исключительно через параметры общедоступной функции, возвращаемые значения или сигнализацию об ошибке. Но в зависимости от того, как мы определяем общедоступный API нашего кода, могут быть сценарии, в которых невозможно протестировать все варианты поведения, используя только общедоступный API.

Это может произойти, если необходимо настроить различные зависимости или проверить наличие или отсутствие определенных побочных эффектов. Некоторые примеры этого могут быть следующими.

- ◆ *Код, взаимодействующий с сервером.* Чтобы протестировать код, может потребоваться настроить или смоделировать сервер, чтобы он мог предоставлять необходимые входные данные. Мы также можем захотеть проверить, какие побочные эффекты код оказывает на сервер, например, как часто он вызывает его, и что запросы имеют допустимый формат.
- ◆ *Код, который сохраняет значения или считывает значения из базы данных.* Нам может потребоваться протестировать код с несколькими различными значениями в базе данных, чтобы проверить все варианты поведения. И мы, вероятно, захотим проверить, какие значения код сохраняет в базе данных (побочный эффект).

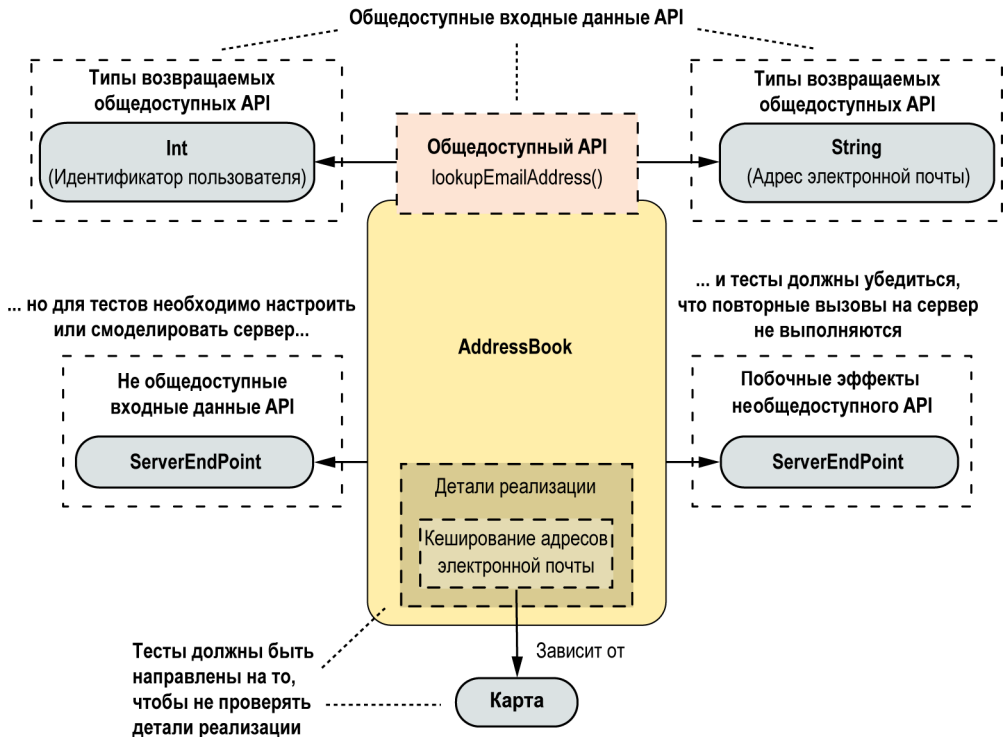


Рис. 10.5. Мы не можем полностью протестировать все важные функции класса адресной книги, используя то, что мы определили как общедоступный API

«Тестировать, используя только общедоступный API» и «не тестировать детали реализации» — оба совета превосходны, но мы должны понимать, что они являются руководящими принципами и что определения «общедоступного API» и «деталей реализации» могут быть разными, субъективными и зависеть от контекста.

Что, в конечном счете, важно, так это то, что мы должным образом тестируем все важные функции кода, и могут быть случаи, когда мы не можем сделать это, используя только то, что мы считаем общедоступным API. Но мы все равно должны помнить о желании сделать тесты максимально независимыми от деталей реализации, поэтому мы должны отклоняться от общедоступного API только тогда, когда действительно нет альтернативы.

10.4. Имитированные реализации

В начале этой главы было сказано, что модульные тесты нацелены на тестирование модуля кода «относительно изолированным образом». Но, как мы только что видели, код имеет тенденцию зависеть от других вещей, и для полного тестирования всех вариантов поведения кода нам часто требуется настроить входные данные и проверить побочные эффекты. Но, как мы вскоре увидим, не всегда возможно или желательно использовать реальные зависимости в тестах.

Альтернативой использованию реальной зависимости является использование *имитированной реализации* (англ. *test double*). Имитированная реализация — это объект, который имитирует зависимость, но таким образом, что делает его более подходящим для использования в тестах. Мы начнем с изучения некоторых причин использования имитированной реализации. Затем мы рассмотрим три конкретных типа имитированных реализаций: имитации (англ. *mocks*), заглушки (англ. *stubs*) и подделки (англ. *fakes*). Попутно мы увидим, насколько проблематичными могут быть имитации и заглушки и почему использование подделки часто предпочтительнее, если она доступна.

10.4.1. Причины использования имитированной реализации

Вот три распространенные причины, по которым мы можем захотеть использовать имитированную реализацию.

- ♦ *Упрощение теста.* Некоторые зависимости сложно и болезненно использовать в тестах. Зависимость может потребовать большого количества настроек или может потребовать, чтобы мы также настроили множество ее подзависимостей. Если это так, наши тесты могут стать сложными и тесно связанными с деталями реализации. Использование имитированной реализации вместо реальной зависимости может упростить ситуацию.
- ♦ *Защита внешнего мира от тестирования.* Некоторые зависимости имеют реальные побочные эффекты. Если одна из зависимостей кода отправляет запросы на реальный сервер или записывает значения в реальную базу данных, это может иметь плохие последствия для пользователей или критически важных бизнес-процессов. В таких сценариях мы могли бы использовать имитированную реализацию для защиты наших систем во внешнем мире от действий теста.

- ♦ **Защита теста от внешнего мира.** Внешний мир может быть недетерминированным. Если одна из зависимостей кода считывает значение из реальной базы данных, в которую записываются другие системы, возвращаемое значение со временем может измениться. Это может привести к тому, что наши тесты будут ненадежными. С другой стороны, имитированную реализацию можно настроить так, чтобы она всегда вела себя одним и тем же детерминированным образом.

В следующих подразделах эти причины рассматриваются более подробно; показано, как можно использовать имитированную реализацию в этих сценариях.

Упрощение теста

Для настройки некоторых зависимостей может потребоваться много усилий. Сама зависимость может потребовать, чтобы мы указали множество параметров, или у нее может быть много подзависимостей, которые требуют настройки. Помимо настройки, нашим тестам также может понадобиться проверить желаемые побочные эффекты в зависимостях. В такой ситуации все может выйти из-под контроля. Мы можем получить массу установочного кода в наших тестах, и они также могут оказаться тесно связанными с большим количеством деталей реализации (рис. 10.6).

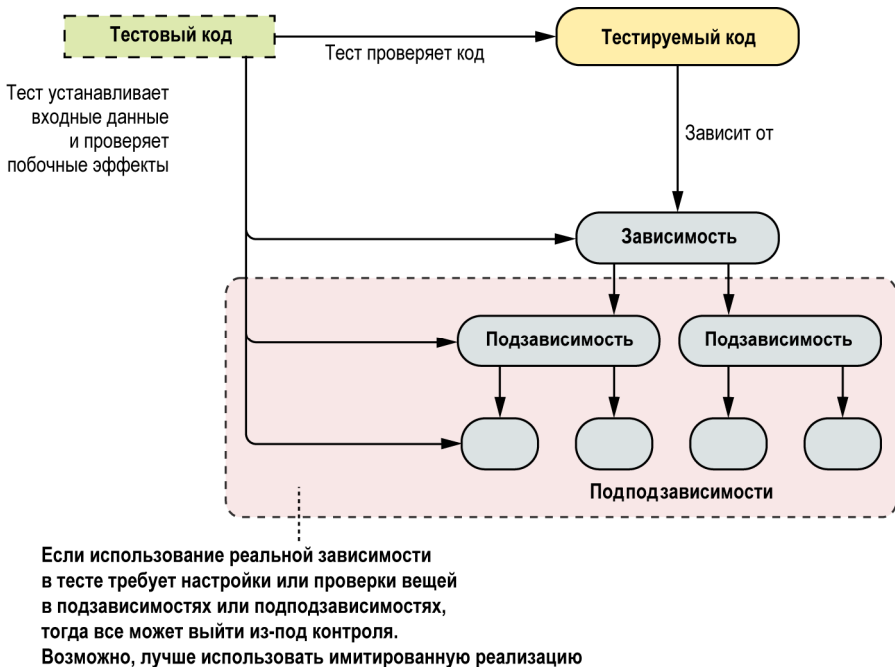


Рис. 10.6. Иногда использование реальных зависимостей в тестах может оказаться непрактичным. Это может быть в том случае, если у зависимости есть много подзависимостей, с которыми также необходимо взаимодействовать

Напротив, если мы используем имитированную реализацию, то мы обходим необходимость настройки реальной зависимости или проверки вещей в ее зависимостях. Тестовый код должен взаимодействовать только с имитированной реализацией для настройки и проверки побочных эффектов (которые должны быть относительно простыми). Рис. 10.7 иллюстрирует, насколько проще становится тест.

Другой мотивацией для упрощения может быть ускорение выполнения тестов; это может быть применимо, если одна из зависимостей вызывает дорогостоящий алгоритм или требует большой медленной настройки.

Как мы рассмотрим далее, существуют сценарии, в которых использование имитированной реализации может фактически сделать тест более связанным с деталями реализации. И настройка имитированной реализации также иногда может быть сложнее, чем использование реальной зависимости, поэтому аргументы за и против использования имитированной реализации для упрощения теста необходимо рассматривать в каждом конкретном случае.

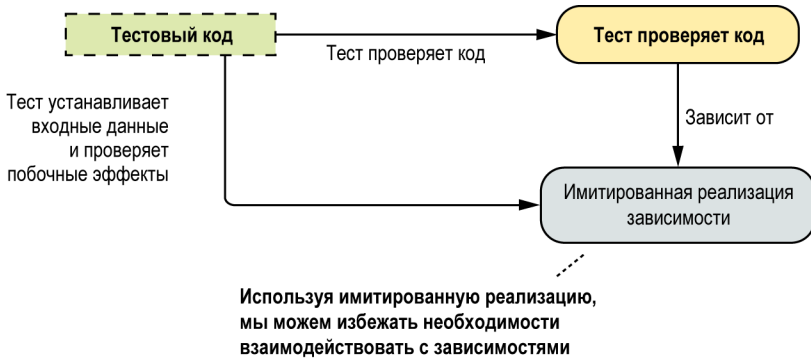


Рис. 10.7. Имитированная реализация может упростить тест, избавив от необходимости беспокоиться о зависимостях

Защита внешнего мира от тестирования

В дополнение к желанию тестировать код относительно изолированным образом могут также существовать неизбежные причины, которые означают, что мы должны тестировать его изолированно. Представьте, что мы работаем над системой, которая обрабатывает платежи, и проводим модульное тестирование фрагмента кода, который списывает деньги с банковского счета клиента. Когда код запускается в реальном мире, одним из побочных эффектов будет снятие реальных денег со счета реального клиента. Код достигает этого, полагаясь на класс под названием `BankAccount`, который, в свою очередь, взаимодействует с реальной банковской системой. Если мы используем экземпляр класса `BankAccount` в нашем тесте, то при каждом запуске теста с реального счета будут сниматься реальные деньги (рис. 10.8). Это почти наверняка не очень хорошая идея, потому что могут быть плохие по-

следствия, такие, как влияние на деньги реальных людей или искажение аудита и бухгалтерского учета компании.

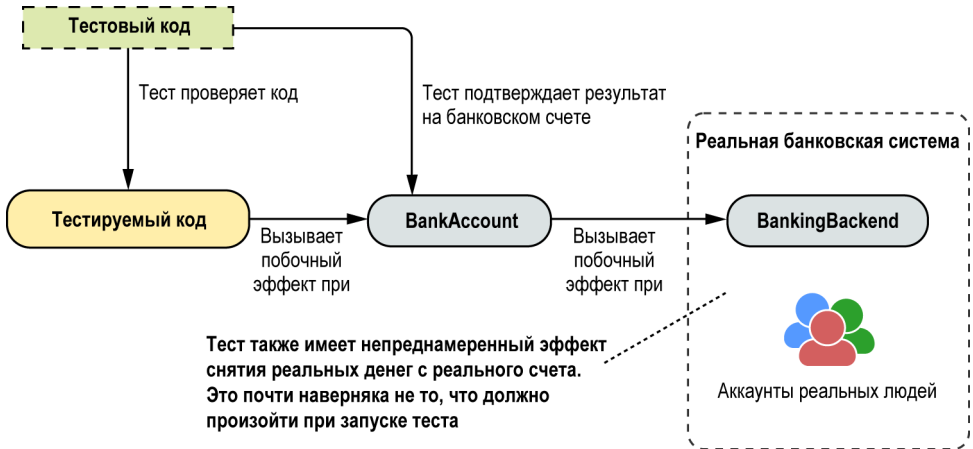


Рис. 10.8. Если зависимость вызывает реальные побочные эффекты, мы, скорее всего, захотим использовать двойной тест вместо использования реальной зависимости

Это пример того, как нам нужно защитить внешний мир от последствий теста. Мы можем достичь этого, используя имитированную реализацию вместо реального экземпляра `BankAccount`. Это изолирует тест от реальной банковской системы и означает, что при выполнении теста не затрагиваются реальные банковские счета или денежные средства (рис. 10.9).

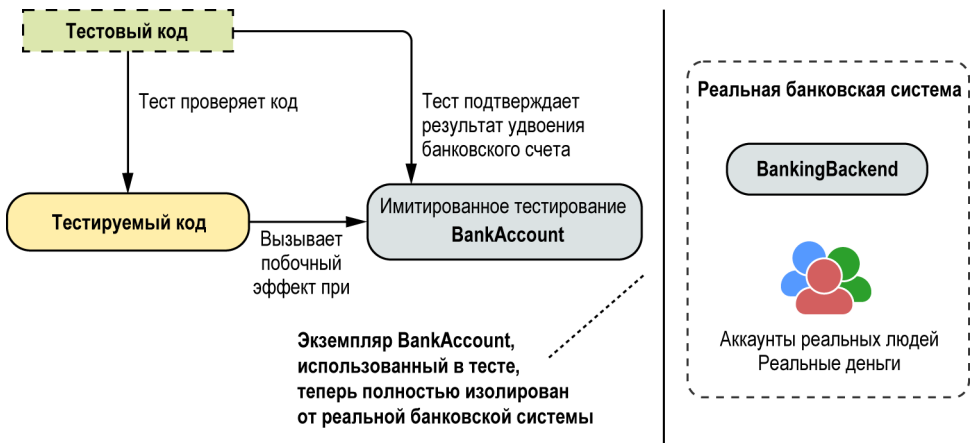


Рис. 10.9. Имитированная реализация может защитить реальные системы во внешнем мире от побочных эффектов

Тест, побочным эффектом которого является снятие реальных денег с реального банковского счета, возможно, является крайним примером, но то, что он демонстрирует, широко применимо. Более вероятным сценарием может быть тест, побочным эффектом которого является отправка запросов на реальный сервер или запись значений в реальную базу данных. Хотя они не могут быть катастрофическими, они могут привести к следующим проблемам.

- ◆ *Пользователи видят странные и запутанные значения.* Представьте, что мы занимаемся электронной коммерцией, и один из наших тестов вносит записи в нашу реальную базу данных. Эти «тестовые» записи могут быть видны пользователям. Пользователь, посещающий домашнюю страницу, может обнаружить, что половина отображаемых продуктов называется «поддельный тестовый элемент», и это вызовет ошибку, если он попытается добавить какой-либо из них в свою корзину. Большинство пользователей, вероятно, не сочтут это хорошим опытом.
- ◆ *Это может повлиять на наш мониторинг и ведение журнала* — тест может намеренно отправить недопустимый запрос на сервер, чтобы проверить правильность обработки результирующего ответа об ошибке. Если этот запрос отправляется на реальный сервер, то частота ошибок для этого сервера будет увеличена. Это может заставить инженеров думать, что есть проблема, когда ее нет. Или, если люди научатся ожидать этого исходного количества ошибок от тестов, они могут не заметить увеличения количества ошибок, когда в системе возникают реальные ошибки.

Важно, чтобы тесты не вызывали побочных эффектов в системах, ориентированных на клиента или критически важных для бизнеса. Эти системы должны быть защищены от тестов, и имитированная реализация может быть эффективным способом добиться этого, сохраняя тест изолированным.

Защита теста от внешнего мира

В дополнение к защите внешнего мира от теста еще одной причиной использования имитированной реализации может быть обратное: защита теста от внешнего мира. Реальные зависимости могут иметь недетерминированное поведение. Примерами этого может быть реальная зависимость, считывающая регулярно меняющееся значение из базы данных или генерирующая что-то вроде идентификатора с помощью генератора случайных чисел. Использование подобной зависимости в тесте может привести к тому, что тест будет ошибочным, и, как мы видели ранее, это, вероятно, то, чего мы хотим избежать.

Чтобы продемонстрировать это и то, как может помочь имитированная реализация, давайте рассмотрим еще одну вещь, которую часть кода может захотеть сделать с банковским счетом: считать баланс. Баланс реального банковского счета потенциально меняется довольно часто, так как владелец счета вносит деньги и снимает их. Даже если мы создадим специальную учетную запись, которую используем только для тестирования, баланс все равно может измениться по мере выплаты процентов или ежемесячного списания комиссии со счета, поэтому, если в тесте

используется реальный банковский счет и тестируемый код считывает баланс, тест может оказаться ошибочным (рис. 10.10).

Решение состоит в том, чтобы изолировать тест от реальной банковской системы, и опять же это то, что мы можем сделать с помощью имитированной реализации. Если мы используем имитированную реализацию для банковского счета, то тестовый код может настроить его с заранее определенным значением для баланса счета (рис. 10.11). Это означает, что баланс счета всегда будет иметь одно и то же детерминированное значение при каждом запуске теста.

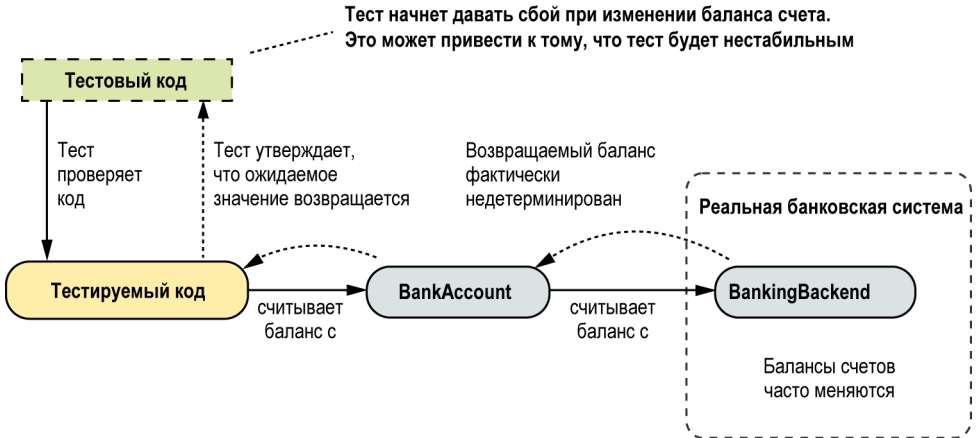


Рис. 10.10. Если зависимость ведет себя недетерминированным образом, это может привести к нестабильности тестов



Рис. 10.11. Имитированное тестирование может защитить тест от любого неопределенного поведения, которое может проявлять реальная зависимость

Как мы видели, есть несколько причин, по которым мы можем решить, что нежелательно или нецелесообразно использовать реальную зависимость. Как только мы

определим, что предпочли бы использовать имитированную реализацию, нам нужно решить, какой тип имитированной реализации использовать. В следующих подразделах будут рассмотрены три наиболее распространенных варианта: имитации, заглушки и подделки.

10.4.2. Имитации

Имитация (mock) имитирует класс или интерфейс, не предоставляя никаких функций, кроме записи вызовов функций-членов. При этом она также записывает, какие значения предоставляются для аргументов при вызове функции. Имитацию можно использовать в тесте, чтобы убедиться, что тестируемый код выполняет определенные вызовы функций, предоставляемых зависимостью. Таким образом, имитация наиболее полезна для имитации зависимости, в которой тестируемый код вызывает побочный эффект. Чтобы продемонстрировать, как можно использовать имитацию, давайте рассмотрим пример с банковским счетом, который мы видели ранее, но на этот раз с некоторым сопутствующим кодом.

В листинге 10.2 показан код класса `PaymentManager`. Этот класс содержит функцию `setInvoice()`, которая, как следует из названия, позволяет вызывающим сторонам оплатить счет путем списания остатка с банковского счета клиента. Если мы пишем модульные тесты для этого класса, то одно из поведений, которое нам, очевидно, нужно будет протестировать, заключается в том, что со счета клиента действительно списывается правильная сумма. Параметр `customerBankAccount` является экземпляром `BankAccount`, поэтому наш тест должен будет взаимодействовать с этой зависимостью, чтобы убедиться, что желаемый побочный эффект вызывается.

Листинг 10.2. Код, зависящий от `BankAccount`

```
class PaymentManager {  
    ...  
  
    PaymentResult settleInvoice(  
        BankAccount customerBankAccount, ❶  
        Invoice invoice) {  
        customerBankAccount.debit(invoice.getBalance()); ❷  
        return PaymentResult.paid(invoice.getId());  
    }  
}
```

❶ Принимает экземпляр `BankAccount` в качестве параметра.

❷ Списание с баланса счета — это одно из поведений, которые нам нужно протестировать.

`BankAccount` — это интерфейс, а реализующий его класс называется `BankAccountImpl`. В листинге 10.3 показан интерфейс `BankAccount` вместе с классом `BankAccountImpl`. Мы видим, что класс `BankAccountImpl` зависит от `BankingBackend`, который подключается к реальной банковской системе. Как мы видели ранее, это означает, что мы не мо-

жем использовать экземпляр `BankAccountImpl` в нашем тесте, потому что это приведет к перемещению реальных денег на реальных счетах (нам нужно защитить внешний мир от теста).

Листинг 10.3. Интерфейс и реализация `BankAccount`

```
interface BankAccount {  
    void debit(MonetaryAmount amount);  
    void credit(MonetaryAmount amount);  
    MonetaryAmount getBalance();  
}  
  
class BankAccountImpl implements BankAccount {  
    private final BankingBackend backend; ❶  
    ...  
  
    override void debit(MonetaryAmount amount) { ... }  
    override void credit(MonetaryAmount amount) { ... }  
    override MonetaryAmount getBalance() { ... }  
}
```

❶ Зависит от `BankingBackend`, который влияет на реальные деньги на реальных банковских счетах.

Альтернативой использованию `BankAccountImpl` является использование имитации интерфейса `BankAccount`, а затем проверка того, что функция `debit()` вызывается с правильными аргументами.

В листинге 10.4 показан код контрольного примера, который проверяет, что со счета списана правильная сумма. Вот некоторые вещи, на которые стоит обратить внимание в коде:

- имитация интерфейса банковского счета создается вызовом `createMock(BankAccount)`;
- `mockAccount` передается функции `setInvoice()` (тестируемый код);
- тест проверяет, что `mockAccount.debit()` был вызван один раз с ожидаемой суммой (в данном случае балансом счета).

Листинг 10.4. Контрольный пример, в котором используется имитация

```
void testSettleInvoice_accountDebited() {  
    BankAccount mockAccount = createMock(BankAccount); ❶  
    MonetaryAmount invoiceBalance =  
        new MonetaryAmount(5.0, Currency.USD);  
    Invoice invoice = new Invoice(invoiceBalance, "test-id");  
    PaymentManager paymentManager = new PaymentManager();  
  
    paymentManager.settleInvoice(mockAccount, invoice); ❷
```

```

verifyThat(mockAccount.debit() ❸
    .wasCalledOnce()
    .withArguments(invoiceBalance);
}

```

- ❶ Создана имитация `BankAccount`.
- ❷ Тестируемый код вызывается с помощью `mockAccount`.
- ❸ Тест утверждает, что `mockAccount.debit()` вызывается с ожидаемыми аргументами.

Использование имитации позволило нам протестировать функцию `PaymentManager.settleInvoice()` без использования класса `BankAccountImpl`. Это успешно защитило внешний мир от теста, но, как мы увидим в разд. 10.4.4, существует реальный риск того, что теперь тест может оказаться нереалистичным и не выявить важные ошибки.

10.4.3. Заглушки

Зажушка имитирует функцию, возвращая предопределенные значения при каждом вызове функции. Это позволяет тестам имитировать зависимости, заглушая определенные функции-члены, которые будет вызывать тестируемый код, и использовать возвращаемые значения. Таким образом, зажушки полезны для имитации зависимостей, из которых код получает входные данные.

Хотя существует четкая разница между имитациями и зажушками, в непринужденной беседе многие инженеры просто используют слово *mock* для обозначения обоих. И во многих инструментах тестирования, которые обеспечивают функциональность зажушки, необходимо создать то, что инструмент называет *mock*, даже если мы хотим использовать его только для зажушки определенных функций-членов. Примеры кода в этом подразделе демонстрируют это.

Давайте представим, что теперь нам нужно изменить функцию `PaymentManager.settleInvoice()`, чтобы проверить, есть ли достаточный баланс на банковском счете, прежде чем он попытается списать средства с банковского счета. Это поможет свести к минимуму количество отклоненных транзакций, которые в противном случае могут повлиять на кредитную историю клиента в его банке. В следующем листинге показано, как выглядит код после внесения этого изменения.

Листинг 10.5. Код, вызывающий `getBalance()`

```

class PaymentManager {
    ...

    PaymentResult settleInvoice(
        BankAccount customerBankAccount,
        Invoice invoice) {
        if (customerBankAccount.getBalance() ❶
            .isLessThan(invoice.getBalance())) {
            return

```

```

        PaymentResult.insufficientFunds(invoice.getId());
    }
    customerBankAccount.debit(invoice.getBalance());
    return PaymentResult.paid(invoice.getId());
}
}

```

- ❶ Код основан на значении, возвращаемом методом `customerBankAccount.getBalance()`.

Новая функциональность, которую мы добавили в функцию `PaymentManager.settleInvoice()`, означает, что теперь есть больше вариантов поведения, для которых нам нужно добавить контрольные примеры:

- «недостаточно средств» — `PaymentResult` возвращается, если средств недостаточно;
- не предпринимается попытка дебетовать счет, если средств недостаточно;
- дебетование счета происходит при достаточном количестве средств.

Понятно, что нам нужно написать несколько юнит-тестов, которые будут зависеть от баланса банковского счета. Если мы используем `BankAccountImpl` в тесте, то тестируемый код будет считывать баланс реального банковского счета, и, как мы установили ранее, он может время от времени меняться, поэтому использование `BankAccountImpl` внесет в наши тесты недетерминированность и потенциально сделает их хрупкими.

Это сценарий, в котором нам нужно защитить тесты от внешнего мира. Мы можем сделать это, используя заглушку для функции `BankAccount.getBalance()`. Мы можем настроить заглушку так, чтобы она возвращала предопределенное значение всякий раз, когда она вызывается. Это позволяет нам проверить, правильно ли ведет себя код, а также убедиться, что тесты детерминированы и надежны.

В листинге 10.6 показан контрольный пример для первого из только что упомянутых вариантов поведения (когда возвращается `PaymentResult` «недостаточно средств», если средств недостаточно).

Вот некоторые моменты, на которые стоит обратить внимание в коде.

- ◆ Как упоминалось ранее, со многими инструментами тестирования необходимо создать то, что инструмент называет макетом, даже если мы хотим использовать его только для создания заглушек, поэтому мы создаем `mockAccount`, но затем заглушаем функцию `getBalance()`, вместо того чтобы фактически создавать использование любой фиктивной функциональности.
- ◆ Заглушка `mockAccount.getBalance()` настроена на возврат заданного значения 9,99\$.

Листинг 10.6. Контрольный пример, использующий заглушку

```

void testSettleInvoice_insufficientFundsCorrectResultReturned() {
    MonetaryAmount invoiceBalance =
        new MonetaryAmount(10.0, Currency.USD);
}

```

```
Invoice invoice = new Invoice(invoiceBalance, "test-id");
BankAccount mockAccount = createMock(BankAccount); ❶
when(mockAccount.getBalance()) ❷
    .thenReturn(new MonetaryAmount(9.99, Currency.USD));
PaymentManager paymentManager = new PaymentManager();

PaymentResult result =
    paymentManager.settleInvoice(mockAccount, invoice);
assertThat(result.getStatus()).isEqualTo(INSUFFICIENT_FUNDS); ❸
}
```

❶ Интерфейс `BankAccount` «имитируется», хотя мы хотим создать только заглушку.

❷ Функция `mockAccount.getBalance()` заглушена и настроена так, чтобы всегда возвращать 9,99\$.

❸ Тест утверждает, что возвращается результат «недостаточно средств».

Использование заглушки позволило защитить тест от внешнего мира и предотвратить нестабильность. В этом (и предыдущем подразделе) показано, как макеты и заглушки могут помочь нам изолировать наши тесты, имитируя зависимости, которые в противном случае могли быть проблематичными.

Иногда это необходимо, но у использования имитаций и заглушек есть и недостатки. В следующем подразделе объясняются два основных недостатка.

10.4.4. Имитации и заглушки могут быть проблематичными

Существуют разные взгляды на использование имитаций и заглушек, которые мы рассмотрим в *разд. 10.4.6*. Прежде чем мы обсудим эти различные школы мысли (и прежде чем мы рассмотрим подделки), важно обсудить некоторые проблемы, которые могут вызвать имитации и заглушки. Два основных недостатка их использования заключаются в том, что они могут привести:

- к нереалистичным тестам, если имитация или заглушка настроены на поведение, отличное от реальной зависимости;
- к тому, что тесты станут тесно связаны с деталями реализации, что, как мы видели ранее, может затруднить рефакторинг.

В следующих двух подразделах рассмотрим их подробнее.

Имитации и заглушки могут привести к нереальным испытаниям

Всякий раз, когда мы имитируем или заглушаем класс или функцию, мы (как инженеры, пишущие тест) должны решить, как будет вести себя эта имитация или заглушка. Существует реальный риск того, что мы заставим их вести себя не так, как класс или функция ведут себя в реальной жизни. Если мы сделаем это, наш тест может пройти, и мы будем думать, что все работает, но когда наш код запустится в реальной жизни, он может вести себя некорректно или выдавать ошибки.

Ранее, когда мы использовали имитацию для тестирования функции `PaymentManager.settleInvoice()`, мы протестировали сценарий, в котором накладная имеет положительное сальдо в размере 5\$, что означает, что клиент должен компании 5\$. Но накладные также могут иметь отрицательное сальдо, например, если клиент получает возмещение или компенсацию за что-то, так что это тоже сценарий, который мы должны протестировать. На первый взгляд это может показаться довольно простым. Мы просто копируем код из предыдущего контрольного примера, который мы видели, и используем значение минус 5\$ для баланса счета. В листинге 10.7 показан код, который мы получили для нашего контрольного примера. Тест пройден, поэтому мы пришли к выводу, что функция `PaymentManager.settleInvoice()` может нормально обрабатывать счета с отрицательным балансом. К сожалению, как мы вскоре увидим, это не так.

Листинг 10.7. Тестирование отрицательного баланса накладной

```
void testSettleInvoice_negativeInvoiceBalance() {  
    BankAccount mockAccount = createMock(BankAccount);  
    MonetaryAmount invoiceBalance =  
        new MonetaryAmount(-5.0, Currency.USD); ❶  
    Invoice invoice = new Invoice(invoiceBalance, "test-id");  
    PaymentManager paymentManager = new PaymentManager();  
  
    paymentManager.settleInvoice(mockAccount, invoice);  
  
    verifyThat(mockAccount.debit() ❷  
        .wasCalledOnce()  
        .withArguments(invoiceBalance);  
}
```

❶ Отрицательный баланс накладной.

❷ Тест утверждает, что `mockAccount.debit()` вызывается с ожидаемой отрицательной суммой.

Наш контрольный пример утверждает, что наш код вызывает `mockAccount.debit()` с правильным балансом счета (в данном случае отрицательным). Но это не означает, что вызов `BankAccountImpl.debit()` с отрицательным значением на самом деле будет делать то, что мы ожидаем в реальной жизни. При написании класса `PaymentManager` мы сделали неявное предположение, что списание отрицательной суммы с банковского счета приведет к добавлению денег на счет. Затем, используя имитацию, мы повторили это предположение в нашем тесте.

Это означает, что правильность этого предположения никогда не проверяется, и, по сути, это тавтология, что тест будет пройден независимо от того, работает ли код на самом деле в реальной жизни.

В действительности наше предположение, к сожалению, неверно. Если мы более внимательно посмотрим на интерфейс `BankAccount`, мы увидим следующую доку-

ментацию, указывающую, что `ArgumentException` будет сгенерировано, если либо `debit()`, либо `Credit()` вызываются с отрицательным значением:

```
interface BankAccount {
    /**
     * @ выбрасывает ArgumentException, если вызывается
     * с отрицательной суммой
     */
    void debit(MonetaryAmount amount);

    /**
     * @ выбрасывает ArgumentException, если вызывается
     * с отрицательной суммой
     */
    void credit(MonetaryAmount amount);

    ...
}
```

Очевидно, что в функции `PaymentManager.settleInvoice()` есть ошибка, но поскольку в нашем тесте мы использовали имитацию, она не выявила эту ошибку. Это один из основных недостатков использования имитации. Инженер, пишущий тест, должен решить, как будет вести себя имитация, и если он допустил ошибку в понимании того, как работает настоящая зависимость, то, скорее всего, он допустит ту же ошибку при настройке имитации.

Эта же проблема может относиться и к использованию заглушек. Использование заглушки проверит, что наш код ведет себя так, как мы хотим, когда зависимость возвращает определенное значение. Но он не проверяет, действительно ли это реалистичное значение для возврата этой зависимости.

В предыдущем подразделе мы использовали заглушку для имитации функции `BankAccount.getBalance()`, но, возможно, мы не рассмотрели должным образом кодовое соглашение этой функции. Представьте, что мы внимательно изучаем интерфейс `BankAccount` и обнаруживаем следующую документацию. Это то, что мы упустили при настройке нашей заглушки:

```
interface BankAccount {
    ...

    /**
     * @ вернуть баланс банковского счета, округленный в меньшую
     * сторону до ближайшего кратного 10 если реальный баланс
     * составляет 19$, то эта функция вернет 10$. Это
     * сделано из соображений безопасности, поскольку точные
     * остатки на счетах иногда используются банком в качестве
     * секретного вопроса.
     */
    MonetaryAmount getBalance();
}
```

ОКРУГЛЕНИЕ БАЛАНСА В МЕНЬШУЮ СТОРОНУ

Пример функции `getBalance()`, возвращающей округленное значение, иллюстрирует, как легко можно упустить определенные детали при заглушке функции. На самом деле округление баланса счета в меньшую сторону, вероятно, не является особенно надежной функцией безопасности. Злоумышленник все еще может определить точный баланс, например, неоднократно зачисляя на счет 0,01\$, пока значение, возвращаемое функцией `getBalance()`, не изменится.

Имитации и заглушки могут создать тесную связь между тестами и деталями реализации

В предыдущем подразделе мы увидели, как вызов `customerBankAccount.debit()` не работает, если счет имеет отрицательный баланс, и как использование имитации означает, что эта ошибка осталась незамеченной во время тестирования. Если инженер в конце концов заметит эту ошибку, он может решить ее, введя оператор `if` в функцию `setInvoice()`, например, в следующем фрагменте кода. Это вызывает функцию `customerBankAccount.debit()`, если баланс положительный, и функцию `customerBankAccount.credit()`, если он отрицательный:

```
PaymentResult settleInvoice(...) {
    ...
    MonetaryAmount balance = invoice.getBalance();
    if (balance.isPositive()) {
        customerBankAccount.debit(balance);
    } else {
        customerBankAccount.credit(balance.absoluteAmount());
    }
    ...
}
```

Если инженер использует имитации для тестирования этого кода, то в конечном итоге он получит различные контрольные примеры, в которых он проверяет, что вызывается `customerBankAccount.debit()`, и другие, где он проверяет, что вызывается `customerBankAccount.credit()`:

```
void testSettleInvoice_positiveInvoiceBalance() {
    ...
    verifyThat(mockAccount.debit()
        .wasCalledOnce()
        .withArguments(invoiceBalance);
}

...

void testSettleInvoice_negativeInvoiceBalance() {
    ...
```



```
verifyThat(mockAccount.credit)
    .wasCalledOnce()
    .withArguments(invoiceBalance.absoluteAmount());
}
```

Это проверяет, вызывает ли код ожидаемые функции, но напрямую не проверяет поведение, которое на самом деле волнует того, кто использует класс. Поведение, о котором проверяющие заботятся, заключается в том, что функция `setInvoice()` переводит правильную сумму денег на счет или со счета. Точная механика этого — всего лишь средство для достижения цели, поэтому вызывается ли функция `credit()` или `debit()` — это деталь реализации.

Чтобы подчеркнуть это, давайте рассмотрим рефакторинг, который может решить выполнить инженер. Он заметил, что несколько фрагментов кода в разных частях кодовой базы содержат этот неуклюжий оператор `if-else` для переключения между вызовами `debit()` и `credit()`. Чтобы улучшить код, они решили перенести эту функциональность в класс `BankAccountImpl`, где ее можно будет использовать повторно. Это означает, что в интерфейс `BankAccount` добавлена новая функция, которая называется `transfer()`:

```
interface BankAccount {
    ...

    /**
     * Переводит указанную сумму на счет. Если сумма
     * отрицательная, то это приводит к переводу денег со счета
     */
    void transfer(MonetaryAmount amount);
}
```

Затем функция `setInvoice()` подвергается рефакторингу для вызова новой функции `transfer()` следующим образом:

```
PaymentResult settleInvoice(...) {
    ...
    MonetaryAmount balance = invoice.getBalance();
    customerBankAccount.transfer(balance.negate());
    ...
}
```

Этот рефакторинг не изменил поведения; изменена только деталь реализации. Но многие из тестов теперь терпят неудачу, потому что они используют имитации, которые ожидают вызова либо `debit()`, либо `credit()`, что теперь больше не происходит. Это противоположно цели, которую мы заявили в *разд. 10.2.2*: тесты не должны зависеть от деталей реализации. Инженеру, выполнившему рефакторинг, придется модифицировать множество контрольных примеров, чтобы они снова прошли успешно, поэтому ему трудно быть уверенным, что его рефакторинг не изменил непреднамеренно какое-либо поведение.

Как упоминалось ранее, существуют разные взгляды на использование имитаций и заглушек, но, на мой взгляд, лучше свести их использование к минимуму. Если реальной альтернативы нет, то лучше использовать имитацию или заглушку в тесте, чем не проводить тест. Но если можно использовать настоящую зависимость или подделку (о чем мы поговорим в следующем подразделе), то, на мой взгляд, это обычно предпочтительнее.

10.4.5. Подделки

Подделка — это альтернативная реализация класса (или интерфейса), которую можно безопасно использовать в тестах. Подделка должна точно имитировать общедоступный API реальной зависимости, но реализация обычно упрощена. Это часто может быть достигнуто путем сохранения состояния в переменной-члене внутри подделки вместо связи с внешней системой.

Весь смысл подделки в том, что ее кодовое соглашение идентично реальной зависимости, поэтому если реальный класс (или интерфейс) не принимает определенный ввод, то подделка тоже не должна. Обычно это означает, что подделка должна поддерживаться той же командой, которая поддерживает код реальной зависимости, потому что если кодовое соглашение реальной зависимости когда-либо изменится, кодовое соглашение подделки также необходимо будет обновить. Давайте рассмотрим интерфейс `BankAccount` и класс `BankAccountImpl`, которые мы видели ранее. Если команда, которая их поддерживает, реализует фальшивый банковский счет, это может выглядеть примерно так, как в листинге 10.8. Вот некоторые вещи, на которые стоит обратить внимание в коде.

- ◆ `FakeBankAccount` реализует интерфейс `BankAccount`, поэтому во время тестирования его можно использовать в любом коде, требующем реализации `BankAccount`.
- ◆ Вместо того чтобы связываться с серверной банковской системой, подделка просто отслеживает баланс счета, используя переменную-член.
- ◆ Подделка выдает `ArgumentException`, если любой из `debit()` или `credit()` вызывает-ся с отрицательной суммой. Это обеспечивает выполнение кодового соглашения и означает, что подделка ведет себя точно так же, как настоящие реализации `BankAccount`. Такие детали делают подделки весьма полезными. Если инженер пишет код, который ошибочно вызывает любую из этих функций с отрицательным значением, то тест с использованием имитации или заглушки может этого не поймать, тогда как тест с использованием подделки обнаружит такую ошибку.
- ◆ Функция `getBalance()` возвращает баланс, округленный в меньшую сторону до ближайших 10, потому что это то, что указано в кодовом соглашении, и так ведут себя реальные реализации `BankAccount`. Опять же, это максимизирует вероятность того, что любые ошибки, возникающие из-за этого слегка неожиданного поведения, будут обнаружены во время тестирования.

- ◆ Помимо реализации всех функций интерфейса `BankAccount`, подделка также предоставляет функцию `getActualBalance()`, которую тесты могут использовать для проверки фактического баланса поддельной учетной записи. Это важно, потому что функция `getBalance()` округляет баланс в меньшую сторону, а это означает, что тесты не могут использовать ее для точной проверки состояния счета.

Листинг 10.8. Поддельный `BankAccount`

```
class FakeBankAccount implements BankAccount { ❶
    private MonetaryAmount balance; ❷

    FakeBankAccount(MonetaryAmount startingBalance) {
        this.balance = startingBalance;
    }

    override void debit(MonetaryAmount amount) {
        if (amount.isNegative()) { ❸
            throw new ArgumentException("Amount can't be
                negative");
        }
        balance = balance.subtract(amount);
    }

    override void credit(MonetaryAmount amount) {
        if (amount.isNegative()) { ❹
            throw new ArgumentException("Amount can't be
                negative");
        }
        balance = balance.add(amount);
    }

    override void transfer(MonetaryAmount amount) {
        balance.add(amount);
    }

    override MonetaryAmount getBalance() { ❺
        return roundDownToNearest10(balance);
    }

    MonetaryAmount getActualBalance() { ❻
        return balance;
    }
}
```

❶ Реализует интерфейс `BankAccount`.

❷ Отслеживает состояние с помощью переменных-членов.

❸ `ArgumentException` выбрасывается, если сумма отрицательная.

- ❷ `ArgumentException` выбрасывается, если сумма отрицательная.
- ❸ Возвращает баланс, округленный в меньшую сторону до ближайших 10.
- ❹ Дополнительная функция, позволяющая тестам проверять фактический (неокругленный) баланс.

Использование подделки вместо имитации или заглушки поможет избежать проблем, которые мы определили в предыдущем подразделе, как мы сейчас увидим.

Подделки могут привести к более реалистичным тестам

В предыдущем подразделе мы видели контрольный пример, целью которого было убедиться, что функция `PaymentManager.settleInvoice()` правильно обработала счет с отрицательным балансом. В этом примере контрольный пример использовал имитацию для проверки того, что `BankAccount.debit()` был вызван с правильной отрицательной суммой. Это привело к прохождению теста, несмотря на то что код был сломан (поскольку на самом деле `debit()` не принимает отрицательные суммы). Если бы мы использовали подделку в этом тесте вместо имитации, то эта ошибка была бы выявлена.

Если мы перепишем контрольный пример с отрицательным балансом накладной, используя `FakeBankAccount`, он будет выглядеть, как в листинге 10.9. Когда вызывается `paymentManager.settleInvoice()`, последующий вызов `FakeBankAccount.debit()` с отрицательной суммой вызовет исключение и вызовет сбой теста. Это позволит нам немедленно узнать, что в коде есть ошибка, и предложит нам исправить ее, прежде чем отправлять что-либо в кодовую базу.

Листинг 10.9. Проверка отрицательного баланса накладной с использованием подделки

```
void testSettleInvoice_negativeInvoiceBalance() {  
    FakeBankAccount fakeAccount = new FakeBankAccount( ❶  
        new MonetaryAmount(100.0, Currency.USD));  
    MonetaryAmount invoiceBalance =  
        new MonetaryAmount(-5.0, Currency.USD); ❷  
    Invoice invoice = new Invoice(invoiceBalance, "test-id");  
    PaymentManager paymentManager = new PaymentManager();  
  
    paymentManager.settleInvoice(fakeAccount, invoice); ❸  
  
    assertThat(fakeAccount.getActualBalance()) ❹  
        .isEqualTo(new MonetaryAmount(105.0, Currency.USD));  
}
```

- ❶ Создан поддельный аккаунт с начальным балансом в 100\$.
- ❷ Остаток счета минус 5\$.
- ❸ Тестируемый код вызывается с помощью `fakeAccount`.
- ❹ Тест утверждает, что новый баланс составляет 105\$.

Подделки могут отделить тесты от деталей реализации

Еще одно преимущество использования подделки вместо имитации или заглушки заключается в том, что это приводит к тому, что тест менее тесно связан с деталями реализации. Ранее мы видели, как использование имитации приводило к сбою тестов, когда инженер выполнял рефакторинг. Это произошло потому, что тесты с использованием имитации подтвердили, что были сделаны определенные вызовы либо к `debit()`, либо к `credit()` (что является деталью реализации). Напротив, если в тесте используется подделка, то вместо проверки этих деталей реализации он будет утверждать, что окончательный баланс счета правильный:

```
...
    assertThat(fakeAccount.getActualBalance())
        .isEqualTo(new MonetaryAmount(105.0, Currency.USD));
...
```

Тестируемый код может переводить деньги на счет или с него, используя любые вызовы функций, которые ему нравятся, но до тех пор пока конечный результат один и тот же, тест будет пройден. Это делает тест более независимым от деталей реализации; рефакторинг, который не меняет поведение, не приведет к провалу тестов.

Не каждая зависимость будет иметь эквивалентную подделку. Это зависит от того, создала ли команда программистов, которая поддерживает настоящую зависимость, ее и готовы ли они ее поддерживать. Но мы можем быть активными; если наша команда владеет определенным классом или интерфейсом и мы знаем, что было бы нецелесообразно использовать настоящую вещь в тесте, возможно, нам стоит реализовать подделку для нее. Это, вероятно, улучшит наше собственное тестирование, а также может принести пользу многим другим инженерам, которые зависят от нашего кода.

Если невозможно использовать настоящую зависимость в тесте, может потребоваться использование имитированной реализации. Если это так и существует подделка, то, на мой взгляд, предпочтительнее использовать эту подделку, а не имитацию или заглушку. Я говорю «на мой взгляд», потому что существуют разные взгляды на имитации и заглушки, которые мы кратко обсудим в следующем подразделе.

10.4.6. Школы мыслей об имитациях

Говоря кратко, есть две точки зрения на использование имитаций (и заглушек) в модульных тестах:

- ◆ *Имитирование*, иногда его называют «Лондонской школой мысли». Сторонники утверждают, что инженеры должны избегать использования реальных зависимостей в тестах и вместо этого использовать имитации. Избегание использования реальных зависимостей и использование большого количества макетов часто подразумевает необходимость использования заглушек для любых частей зави-

симостей, которые предоставляют входные данные, поэтому использование фиктивного подхода часто включает заглушки и имитации.

- ◆ *Классицизм*, иногда его называют «Детройтской школой мысли». Сторонники утверждают, что использование имитаций и заглушек должно быть сведено к минимуму и что инженеры должны предпочитать использовать в тестах реальные зависимости. Когда невозможно использовать настоящую зависимость, следующим предпочтением является использование подделки. Имитации и заглушки следует использовать только в крайнем случае, когда невозможно использовать ни настоящую зависимость, ни подделку.

Одно из основных практических различий между тестами, написанными с использованием этих двух подходов, заключается в том, что имитированные тесты, как правило, проверяют взаимодействия, а классические тесты, как правило, проверяют результирующее состояние в коде и его зависимости. В этом смысле имитированный подход стремится зафиксировать, *как* тестируемый код что-то делает, в то время как классический подход стремится зафиксировать *конечный результат* выполнения кода (не обязательно заботясь о том, как это достигается).

Ниже приведены некоторые аргументы в пользу имитаций.

- ◆ *Это делает модульные тесты более изолированными*. Использование имитаций означает, что тест не заканчивается проверкой зависимостей. Это означает, что сбой в конкретном фрагменте кода вызовет сбой тестов только в модульных тестах для этого кода, а не в тестах для другого кода, который от него зависит.
- ◆ *Это может упростить написание тестов*. Использование реальных зависимостей требует выяснения, какие из них нужны для теста и как правильно в них что-то настроить и проверить. Имитацию или заглушку, с другой стороны, часто несложно настроить, потому что это можно сделать без необходимости фактического создания зависимости и беспокойства о настройке подзависимостей.

Некоторые аргументы в пользу использования классического подхода и против использования имитаций заключаются в следующем (оба они обсуждались в предыдущих подразделах).

- ◆ Имитация проверяет, что код выполняет конкретный вызов, но не проверяет, действителен ли этот вызов. Использование большого количества имитаций (или заглушек) может привести к тому, что тесты будут пройдены, даже если код полностью сломан.
- ◆ Классический подход может привести к тому, что тесты будут более независимы от деталей реализации. При классическом подходе акцент делается на проверке конечного результата: что возвращает код или результирующее состояние. А что касается теста, то неважно, как код этого добивается. Это означает, что тесты терпят неудачу только при изменении поведения, а не при изменении деталей реализации.

Честно говоря, в первые годы работы инженером-программистом я понятия не имел, что эти два подхода существуют как формализованные школы мысли. Не подозревая об этом в то время, я, казалось, естественно принял более фиктивный

подход и писал модульные тесты, в которых большинство зависимостей были имитированы или заглушены. По общему признанию, в то время я не особенно задумывался об этом, и моя главная причина использовать то, что оказалось фиктивным подходом, заключалось в том, что это, казалось, облегчало мою жизнь. Но я пожалел об этом, так как это привело к тому, что тесты не проверяли должным образом, что все действительно работает, и очень усложнили рефакторинг кода.

Попробовав оба подхода, я теперь твердо отдаю предпочтение классической школе мысли, и содержание главы отражает это. Но важно подчеркнуть, что это мнение и не каждый инженер с ним согласится. Если вам интересно прочитать более подробное описание ложных и классических направлений мысли, то во второй половине статьи Мартина Фаулера эта тема обсуждается очень подробно: <http://mng.bz/N8Pv>.

10.5. Выбор из основных принципов тестирования

Как вы, возможно, уже заметили, существует множество философий и методологий, связанных с тестированием, и иногда они представляются как принцип «все или ничего»: вы либо соглашаетесь со всеми частями философии, либо ни с одной из них. На самом деле жизнь не такая, и мы вольны выбирать из разных философий то, что считаем нужным.

Одним из примеров философии тестирования является *разработка через тестирование* (англ. *test-driven development*, TDD). Самая известная часть этой философии гласит, что инженеры должны писать тесты, прежде чем писать какой-либо код реализации. Хотя многие признают теоретические преимущества этого, я не часто встречаю инженеров, которые делают так на практике; это просто не соответствует их методам работы. Это не означает, что они полностью игнорируют всю философию TDD; это просто означает, что они не полностью подписываются на это. Многие из них по-прежнему стремятся достигнуть многих других целей, предписанных TDD, таких, как изоляция тестов, сосредоточение на них и отказ от тестирования деталей реализации. Ниже приведены некоторые примеры философии и методологии тестирования.

- ◆ *Разработка через тестирование* ❶ — TDD поддерживает процесс, при котором контрольный пример пишется до написания любого реального кода; затем пишется минимум реального кода, чтобы пройти контрольный пример, а затем код подвергается рефакторингу для улучшения структуры или устранения дублирования. Инженерам рекомендуется повторять эти шаги небольшими итерациями. Как только что упоминалось, сторонники TDD обычно также выступают за различные другие передовые методы, такие, как изоляция и фокусированность контрольных примеров и отказ от тестирования деталей реализации.
- ◆ *Разработка, основанная на поведении* (англ. *behavior-driven development*, BDD) ❷. BDD может означать немного разные вещи для разных людей, но суть заключается в том, чтобы сосредоточиться на определении поведения (или функциональности), которое должно демонстрировать программное обеспечение

(часто с точки зрения пользователя, клиента или бизнеса). Это желаемое поведение фиксируется и записывается в формате, для которого затем может быть разработано программное обеспечение. Тесты должны отражать это желаемое поведение, а не атрибуты самого программного обеспечения. То, как именно фиксируется и регистрируется это поведение, какие заинтересованные стороны участвуют в процессе и насколько оно формализовано, может сильно различаться в разных организациях.

- ◆ *Разработка, основанная на приемочных тестированиях* (англ. *acceptance test-driven development*, ATDD). ATDD может означать разное для разных людей, и степень, в которой она пересекается (или соответствует) с BDD, варьирует в зависимости от определения. ATDD включает в себя определение поведения (или функциональности), которое должно демонстрировать программное обеспечение (часто с точки зрения клиента), и создание автоматических приемочных тестов, которые проверяют, работает ли программное обеспечение в целом должным образом. Подобно TDD, эти тесты должны быть созданы до реализации реального кода. Теоретически, как только все приемочные тесты пройдены, программное обеспечение готово к приему заказчиком.

❶ Некоторые утверждают, что TDD может проследить свое происхождение до 1960-х годов, но более современная, формализованная философия, чаще всего ассоциируемая с этим термином, широко приписывается Кенту Беку в 1990-х годах (Бек, как известно, утверждает, что он «заново открыл» TDD, а не изобрел его).

❷ Идея развития, основанного на поведении, широко приписывается Дэниелу Терхорсту-Норту в 2000-х годах. Копию статьи, в которой Terhorst-North представил эту идею, можно найти здесь: <https://dannorth.net/introduction-bdd/>.

Философия и методология тестирования, как правило, документируют способы работы, которые некоторые инженеры считают эффективными. Но, в конце концов, цель, которую мы пытаемся достичь, имеет большее значение, чем метод работы, который мы выбрали для ее достижения. Важно убедиться, что мы пишем хорошие, тщательные тесты и выпускаем высококачественное программное обеспечение. Разные люди работают по-разному; если вы работаете наиболее эффективно, следуя заданной философии или методологии до буквы, тогда отлично, но если вы работаете более эффективно другим способом, то это тоже абсолютно нормально.

Итоги

- ◆ Практически каждый фрагмент «реального кода», отправленный в кодовую базу, должен сопровождаться модульным тестом.
- ◆ Каждое поведение, которое демонстрирует «реальный код», должно сопровождаться контрольным примером, который выполняет его и проверяет результат. Для всего, кроме простейших контрольных примеров, принято делить код внутри каждого из них на три отдельных раздела: упорядочивать, действовать и утверждать.

- ◆ Ключевые особенности хорошего модульного теста заключаются в следующем:
 - точно обнаруживает поломки;
 - не зависит от деталей реализации;
 - имеет хорошо объясненные неудачи;
 - имеет понятный тестовый код;
 - его легко и быстро запустить.
- ◆ Имитированные реализации можно использовать в модульном тесте, когда невозможно или нецелесообразно использовать настоящую зависимость. Это могут быть имитации, заглушки, подделки.
- ◆ Имитации и заглушки могут привести к нереалистичным тестам, тесно связанным с деталями реализации.
- ◆ Существуют разные взгляды на использование имитаций и заглушек. Я считаю, что реальные зависимости должны использоваться в тестах, где это возможно. В противном случае имитация является следующим лучшим вариантом, а имитации и заглушки следует использовать только в крайнем случае.

11 Методы модульного тестирования

ЭТА ГЛАВА ОХВАТЫВАЕТ

- ◇ Эффективное и надежное модульное тестирование всех моделей поведения кода
- ◇ Обеспечение понятности тестов и правильного объяснения сбоев
- ◇ Использование внедрения зависимостей для обеспечения тестируемости кода

В *главе 10* был определен ряд принципов, которые могут быть использованы нами при написании эффективных модульных тестов. Данная глава основана на этих принципах и охватывает ряд практических приемов, которые мы можем применять в повседневном программировании. В предыдущей главе описаны ключевые особенности, которые должны демонстрировать хорошие модульные тесты. Мотивация для многих методов непосредственно вытекает из этого, поэтому напоминаю, что ключевые функции эффективного теста заключаются в следующем.

- ◆ *Точное обнаружение повреждений.* Если код неисправен, тест должен завершиться неудачей. Причем тест должен завершиться неудачей только в том случае, если код действительно неисправен (нам не нужны ложные срабатывания).
- ◆ *Безразличие к деталям реализации.* Изменения в деталях реализации в идеале не должны приводить к изменениям в тестах.
- ◆ *Хорошо объясненные сбои.* Если код неисправен, сбой теста должен содержать четкое объяснение проблемы.
- ◆ *Понятный тестовый код.* Другие программисты должны быть в состоянии понять, что именно тест тестирует и как он это делает.
- ◆ *Простота и быстрота запуска.* Программистам довольно часто приходится запускать модульные тесты во время своей повседневной работы. Медленный или сложный в выполнении модульный тест приведет к потере большого количества рабочего времени.

Вовсе не обязательно, что любой тест, который мы напишем, будет обладать всеми этими элементами, и в результате слишком легко получить тесты, которые не имеют эффективности и сложны в поддержке. К счастью, существует ряд практических методов, которые мы можем применить, чтобы максимально увеличить вероятность того, что наши тесты действительно продемонстрируют эти особенности. В следующих разделах рассматриваются некоторые из основных этих элементов.

11.1. Тестируйте поведение, а не только функции

Тестирование фрагмента кода отчасти похоже на работу со списком дел. Есть ряд вещей, которые делает код, подвергающийся тестированию (или будет делать, если мы написали тесты перед тем, как написали код), и нам нужно написать контрольный пример для проверки каждого из пунктов. Но, как и в случае с любым списком дел, успешный результат зависит от того, какие вещи действительно есть в списке, и от того, насколько они для нас важны.

Ошибка, которую иногда совершают программисты, заключается в том, что они смотрят на фрагмент кода и добавляют только имена функций в свой список дел для тестирования, поэтому, если класс имеет две функции, программист может написать только два контрольных примера (по одному для каждой функции). В *главе 10* мы установили, что нам необходимо проверять все важные модели поведения, которые демонстрирует фрагмент кода. Проблема с концентрацией внимания на тестировании каждой функции заключается в том, что функция часто может демонстрировать более одного поведения, а поведение иногда может охватывать несколько функций. Если мы напишем только один контрольный пример для каждой функции, вполне вероятно, что мы можем пропустить некоторые важные модели поведения. Лучше заполнить наш список дел всеми моделями поведения, которые нас интересуют, а не только именами функций, которые мы видим.

11.1.1. Одного контрольного примера для каждой функции часто недостаточно

Представьте, что мы работаем в банке, поддерживающем систему, которая автоматически оценивает заявки на ипотеку. Код в листинге 11.1 показывает класс, который принимает решение о том, может ли клиент получить ипотеку, и если да, то какую сумму он может занять. В коде происходит довольно много интересных вещей, например следующее:

- ◆ Функция `assess()` вызывает приватную вспомогательную функцию, чтобы определить, имеет ли клиент право на получение ипотеки. Клиент имеет право на ипотеку, если он
 - имеет хорошую кредитную историю;
 - не имеет иной текущей ипотеки;
 - допущен компанией.
- ◆ Если клиент имеет право на получение кредита, то вызывается другая приватная вспомогательная функция для определения максимальной суммы займа для клиента. Это рассчитывается как годовой доход клиента минус его ежегодные расходы, умноженный на 10.

Листинг 11.1. Код для оценки ипотечного кредита

```

class MortgageAssessor {
    private const Double MORTGAGE_MULTIPLIER = 10.0;

    MortgageDecision assess(Customer customer) {
        if (!isEligibleForMortgage(customer)) { ❶
            return MortgageDecision.rejected();
        }
        return MortgageDecision.approve(getMaxLoanAmount(customer));
    }

    private static Boolean isEligibleForMortgage(Customer customer) { ❷
        return customer.hasGoodCreditRating() &&
            !customer.hasExistingMortgage() &&
            !customer.isBanned();
    }

    private static MonetaryAmount getMaxLoanAmount(Customer customer) { ❸
        return customer.getIncome()
            .minus(customer.getOutgoings())
            .multiplyBy(MORTGAGE_MULTIPLIER);
    }
}

```

❶ Заявка отклонена, если клиент непригоден.

❷ Приватная вспомогательная функция для определения того, имеет ли клиент право получить ипотеку.

❸ Приватная вспомогательная функция для определения максимальной суммы кредита.

Теперь представьте, что мы посмотрим на тесты для этого кода и увидим только один контрольный пример, который проверяет функцию `assess()`. В листинге 11.2 показан этот единственный контрольный пример. Так проверяются некоторые функции, выполняемые функцией `assess()`, например следующие:

- ипотека, одобренная для клиента, допущенного банком, с хорошей кредитной историей, без иной текущей ипотеки;
- максимальная сумма кредита, которая равна доходу клиента минус его расходы, получившаяся сумма умножается на 10.

Но ясно, что такой тест также оставляет многое непроверенным, например все причины, по которым ипотека может быть отклонена. Это недостаточный объем тестирования: мы могли бы изменить функцию `MortgageAssessor.assess()` для утверждения ипотечных кредитов даже для недопущенных клиентов, и тесты все равно завершались бы успешно!

Листинг 11.2. Тест для оценки ипотеки

```

testAssess() {
    Customer customer = new Customer(

```

```
income: new MonetaryAmount(50000, Currency.USD),
outgoings: new MonetaryAmount(20000, Currency.USD),
hasGoodCreditRating: true,
hasExistingMortgage: false,
isBanned: false);

MortgageAssessor mortgageAssessor = new MortgageAssessor();

MortgageDecision decision = mortgageAssessor.assess(customer);

assertThat(decision.isApproved()).isTrue();
assertThat(decision.getMaxLoanAmount()).isEqualTo(
    new MonetaryAmount(300000, Currency.USD));
}
```

Здесь проблема заключается в том, что программист, пишущий тесты, сосредоточился на тестировании функций, а не поведения. Функция `assess()` является единственной функцией в общедоступном API из класса `MortgageAssessor`, поэтому он написал только один контрольный пример. К сожалению, одного тестового примера далеко не достаточно, чтобы полностью гарантировать, что функция `mortgageAssessor.assess()` ведет себя правильно.

11.1.2. Решение: сосредоточьтесь на тестировании каждого поведения

Как показывает предыдущий пример, порой не существует однозначного соответствия между функциями и поведением. Если мы сосредоточимся на тестировании только функций, то очень легко получить набор контрольных примеров, которые не проверяют все важные модели поведения, которые нас действительно волнуют. В примере с классом `MortgageAssessor` существует несколько моделей поведения, которые нас интересуют, в том числе следующие.

- ◆ Заявка на ипотеку отклоняется для любых клиентов, к которым относится хотя бы один из следующих пунктов:
 - у клиента плохая кредитная история;
 - у клиента уже есть иная текущая ипотека;
 - клиент не допущен компанией.
- ◆ Если заявка на ипотеку принята, то максимальная сумма кредита равна доходу клиента минус его расходы, получившаяся сумма умножается на 10.

Каждая из этих моделей поведения должна быть протестирована, для чего требуется написать гораздо больше, чем один контрольный пример. Чтобы повысить наш уровень доверия к коду, также имеет смысл протестировать различные значения и граничные условия, поэтому мы, вероятно, захотим включить в тест следующие контрольные примеры:

- несколько разных значений доходов и расходов, чтобы убедиться, что арифметика в коде верна;

- некоторые крайние значения, такие, как нулевой доход или расходы, а также очень большие суммы доходов или расходов.

Вполне вероятно, что в итоге мы получим 10 или более различных контрольных примеров для полного тестирования класса `MortgageAssessor`. Это совершенно нормально и ожидаемо: нередко можно увидеть 300 строк тестового кода для 100-строчного фрагмента реального кода. На самом деле иногда это тревожный признак, когда объем тестового кода не превышает объем реального кода, так как это может свидетельствовать о том, что не каждая модель поведения проверяется должным образом.

Упражнение на придумывание поведения для тестирования также является отличным способом выявления потенциальных проблем с кодом. Например, когда мы думаем о поведении для тестирования, мы, вероятно, в конечном итоге зададимся вопросом о том, что произойдет, если расходы клиента превысят его доход. В настоящее время функция `MortgageAssessor.assess()` одобрит такую заявку с отрицательной максимальной суммой кредита. Это довольно странная функциональность, поэтому эта реализация, вероятно, побудит нас пересмотреть логику и справиться с этим сценарием немного более изящно.

Дважды проверьте, что каждый вариант поведения был протестирован

Хороший способ оценить, правильно ли протестирован фрагмент кода, это подумать о том, как кто-то теоретически может взломать код и при этом пройти тесты. Ниже представлены хорошие вопросы, которые неплохо задать, просматривая код. Если ответ на любой из них положительный, то это говорит о том, что тестируются не все модели поведения.

- ◆ Существуют ли какие-либо строки кода, которые могут быть удалены, при этом все равно произойдет компиляция кода и тесты будут пройдены?
- ◆ Может ли полнота любых операторов `if` (или эквивалентных) быть изменена, при этом тесты все равно будут пройдены (например, замена `if (something) { на if (!something) }`)?
- ◆ Могут ли какие-либо логические или арифметические операторы быть заменены альтернативами, при этом тесты все равно будут пройдены? Примерами этого может быть замена `&&` на `||` или замена `+` на `-`.
- ◆ Могут ли значения каких-либо констант или жестко запрограммированных значений быть изменены, при этом тесты все равно будут пройдены?

Дело в том, что каждая строка кода, оператор `if`, логическое выражение или значение в тестируемом коде должны существовать с определенной целью. Если фрагмент кода действительно лишний, то его следует удалить. Если фрагмент кода не лишний, то это означает, что должна быть какая-то важная модель поведения, которая каким-то образом зависит от него. Если в коде наблюдается важное поведение, то для проверки этого поведения должен быть контрольный пример, поэтому

любое изменение функциональности кода должно привести, по крайней мере, к одному сбою в контрольном примере. Если это не так, значит, не все модели поведения тестируются.

Единственным настоящим исключением из этого правила является код, который автоматически проверяет наличие ошибок программирования. Например, у нас в коде может быть проверка или утверждение, для того чтобы убедиться в правильности определенного предположения. Возможно, это неосуществимо в тесте, потому что единственный способ проверить защитную логику — это нарушить предположение, повредив код.

Проверка того, что изменения в функциональности приводят к сбою теста, иногда может быть в некоторой степени автоматизирована с помощью *мутационного тестирования*. Инструмент такого тестирования создаст версии кода с небольшими изменениями. Если тесты все еще проходятся после изменения кода, это признак того, что не каждая модель поведения проверяется должным образом.

Не забывайте о сценариях ошибок

Еще один важный набор моделей поведения, которые легко упустить из виду, — это то, как код ведет себя при возникновении сценариев ошибок. Это может показаться похожим на крайние случаи, потому что мы не обязательно ожидаем, что ошибки будут возникать так часто. Но то, как фрагмент кода обрабатывает и сигнализирует о различных сценариях ошибок, тем не менее, является важным поведением, о котором мы (и вызывающие пользователи нашего кода) заботимся. Поэтому такие сценарии должны быть проверены.

Чтобы продемонстрировать это, рассмотрим листинг 11.3. Функция `BankAccount.debit()` создает исключение `ArgumentException`, если она вызывается с отрицательной суммой. Функция, вызываемая с отрицательным значением, является сценарием ошибки, и тот факт, что она создает `ArgumentException`, когда такое происходит, является важным типом поведения. Поэтому его следует проверить.

Листинг 11.3. Код, обрабатывающий ошибку

```
class BankAccount {  
    ...  
    void debit(MonetaryAmount amount) {  
        if (amount.isNegative()) {  
            throw new ArgumentException("Amount can't be negative"); ❶  
        }  
        ...  
    }  
}
```

❶ Создает исключение `ArgumentException`, если сумма отрицательна.

В листинге 11.4 показано, как мы можем проверить поведение функции в сценарии этой ошибки. Контрольный пример указывает на то, что исключение `ArgumentException` возникает при вызове функции `debit()` с суммой `-0,01` доллара.

Он также утверждает, что возникшее исключение содержит ожидаемое сообщение об ошибке.

Листинг 11.4. Обработка ошибок при тестировании

```
void testDebit_negativeAmount_throwsArgumentException {
    MonetaryAmount negativeAmount =
        new MonetaryAmount(-0.01, Currency.USD);
    BankAccount bankAccount = new BankAccount();

    ArgumentException exception = assertThrows( ❶
        ArgumentException,
        () -> bankAccount.debit(negativeAmount));
    assertThat(exception.getMessage()) ❷
        .isEqualTo("Amount can't be negative");
}
```

❶ Утверждает, что `ArgumentException` возникает при вызове функции `debit()` с отрицательной суммой.

❷ Утверждает, что созданное исключение содержит ожидаемое сообщение об ошибке.

Фрагмент кода имеет тенденцию демонстрировать множество вариантов поведения, и довольно часто бывает так, что даже одна функция может демонстрировать множество различных типов поведения в зависимости от значений, с которыми она вызывается, или состояния, в котором находится система. Написание только одного контрольного примера для каждой функции редко приводит к достаточному объему тестирования. Вместо того чтобы концентрироваться на функциях, обычно эффективнее определить все модели поведения, которые, в конечном счете, имеют значение, и убедиться, что для каждой из них есть контрольный пример.

11.2. Избегайте делать вещи видимыми только для тестирования

Класс (или модуль кода) обычно имеет некоторое количество функций, которые видны внешнему коду; мы часто называем их *общедоступными* функциями. Этот набор общедоступных функций обычно формирует общедоступный API кода. В дополнение к общедоступным функциям довольно часто код также содержит некоторое количество *приватных* функций. Они видны только для кода внутри класса (или модуля кода). Следующий фрагмент демонстрирует это различие:

```
class MyClass {

    String publicFunction() { ... } ❶
    private String privateFunction1 { ... } ❷
    private String privateFunction2 { ... }
}
```


- ❶ Видимый для кода за пределами класса.
- ❷ Видимый только для кода внутри класса.

Приватные функции — это детали реализации, и они не являются чем-то, о чем код вне класса должен знать или когда-либо использовать напрямую. Иногда может показаться заманчивым сделать некоторые из этих приватных функций видимыми для тестового кода, чтобы их можно было напрямую протестировать. Однако это не очень хорошая идея, так как это может привести к тестам, которые тесно связаны с деталями реализации и которые не проверяют то, что нас, в конечном счете, интересует.

11.2.1. Тестирование приватных функций — это обычно плохая идея

В предыдущем разделе мы установили, что важно протестировать все модели поведения класса `MortgageAssessor` (повторяется в листинге 11.5). Общедоступным API этого класса является функция `assess()`. В дополнение к этой общедоступной функции класс также имеет две приватные вспомогательные функции: `isEligibleForMortgage()` и `getMaxLoanAmount()`. Они не видны никакому коду за пределами класса и, следовательно, являются деталями реализации.

Листинг 11.5. Класс с приватными вспомогательными функциями

```
class MortgageAssessor {  
    ...  
  
    MortgageDecision assess(Customer customer) { ... } ❶  
  
    private static Boolean isEligibleForMortgage(  
        Customer customer) { ... } ❷  
  
    private static MonetaryAmount getMaxLoanAmount(  
        Customer customer) { ... }  
}
```

- ❶ Общедоступный API.
- ❷ Приватные вспомогательные функции.

Давайте сосредоточимся на одной из моделей поведения класса `MortgageAssessor`, которую нам нужно проверить: заявка на ипотеку отклоняется, если у клиента плохая кредитная история. Один из распространенных способов, которым программисты могут в конечном итоге протестировать неправильную функцию, — это объединить желаемый конечный результат с промежуточной детализацией реализации. Если мы более внимательно посмотрим на класс `MortgageAssessor`, мы увидим, что частная вспомогательная функция `isEligibleForMortgage()` возвращает значение `false`, если у клиента плохая кредитная история. Это может вызвать соблазн сделать функцию `isEligibleForMortgage()` видимой для тестового кода, чтобы ее можно было

протестировать. В листинге 11.6 показано, как будет выглядеть класс, если программист сделает функцию `isEligibleForMortgage()` видимой таким образом. Делая ее общедоступной, программист делает ее видимой для всего остального кода (а не только для тестового кода). Программист добавил комментарий «Видно только для тестирования», чтобы другие программисты не вызывали эту функцию из чего-либо, кроме тестового кода. Но, как мы уже видели на протяжении всей этой книги, такой комментарий мелким шрифтом очень легко не заметить.

Листинг 11.6. Приватная функция стала видимой

```
class MortgageAssessor {
    private const Double MORTGAGE_MULTIPLIER = 10.0;

    MortgageDecision assess(Customer customer) { ❶
        if (!isEligibleForMortgage(customer)) { ❷
            return MortgageDecision.rejected();
        }
        return MortgageDecision.approve(getMaxLoanAmount(customer));
    }
    /** Видно только для тестирования */ ❸
    static Boolean isEligibleForMortgage(Customer customer) {
        return customer.hasGoodCreditRating() &&
            !customer.hasExistingMortgage() &&
            !customer.isBanned();
    }
    ...
}
```

❶ Общедоступный API.

❷ То, какие вспомогательные функции вызываются, зависит от деталей реализации.

❸ Сделано общедоступным только для того, чтобы этот фрагмент кода можно было непосредственно протестировать.

Сделав функцию `isEligibleForMortgage()` видимой, программист, скорее всего, напишет кучу контрольных примеров, которые ее вызовут, и проверит, возвращает ли она `true` или `false` в верных сценариях. В листинге 11.7 показан один из таких контрольных примеров. Он проверяет, что функция `isEligibleForMortgage()` возвращает значение `false`, если у клиента плохая кредитная история. Как мы сейчас увидим, существует ряд причин, по которым тестирование подобной приватной функции может оказаться плохой идеей.

Листинг 11.7. Тестирование приватной функции

```
testIsEligibleForMortgage_badCreditRating_ineligible() {
    Customer customer = new Customer(
        income: new MonetaryAmount(50000, Currency.USD),
        outgoing: new MonetaryAmount(25000, Currency.USD),
```

```
hasGoodCreditRating: false,  
hasExistingMortgage: false,  
isBanned: false);  
  
assertThat (MortgageAssessor.isEligibleForMortgage(customer)) ❶  
    .isFalse();  
}
```

❶ Непосредственно проверяет «приватную» функцию `isEligibleForMortgage()`.

Проблема с тем, чтобы сделать приватную функцию видимой и протестировать ее таким образом, заключается в трех аспектах.

1. Тест на самом деле не проверяет модель поведения, которая нас интересует. Выше мы сказали, что результат, интересный нам, заключается в том, что заявка на ипотеку отклоняется, если у клиента плохая кредитная история. Контрольный пример в листинге 11.7 на самом деле проверяет наличие в коде функции под названием `isEligibleForMortgage()`, которая возвращает значение `false` при вызове клиента с плохой кредитной историей. Это не гарантирует, что заявка на ипотеку в конечном итоге будет отклонена при таком сценарии. Программист может непреднамеренно изменить функцию `assess()`, чтобы вызвать `isEligibleForMortgage()` неправильно (или вообще не вызывать ее). Контрольный пример в листинге 11.7 все равно пройдет, несмотря на то что класс `MortgageAssessor` сильно поврежден.
2. Это делает тест небезразличным к деталям реализации. Тот факт, что существует приватная функция, называемая `isEligibleForMortgage()`, является деталью реализации. Программисты могут захотеть переработать код, например, переименовав эту функцию или переместив ее в отдельный вспомогательный класс. В идеале любой подобный рефакторинг не должен приводить к сбою ни одного из тестов. Но поскольку мы непосредственно тестируем функцию `isEligibleForMortgage()`, подобный рефакторинг приведет к сбою тестов.
3. Мы фактически изменили общедоступный API класса `MortgageAssessor`. Комментарий типа «Видно только для тестирования» очень легко пропустить (он напечатан мелким шрифтом в соглашении по написанию кода), поэтому мы можем обнаружить, что другие программисты начинают вызывать функцию `isEligibleForMortgage()` и полагаться на нее. Прежде чем мы это узнаем, мы никогда не сможем изменить или переработать эту функцию, потому что от нее зависит такое большое количество другого кода.

Хороший модульный тест должен проверять модели поведения, которые, в конечном счете, имеют значение. Это увеличивает вероятность того, что тест точно обнаружит повреждения в коде, и, как правило, делает тест независимым от деталей реализации. Это две ключевые особенности хорошего модульного теста, которые были определены в *главе 10*. Тестирование приватной функции часто противоречит обоим этим целям. Как мы увидим в следующих двух подразделах, мы можем избежать тестирования приватных функций, либо тестируя через общедоступный API, либо гарантируя, что наш код разбит на соответствующие слои абстракции.

11.2.2. Решение: выбирайте тестирование через общедоступный API

В предыдущей главе мы обсудили руководящий принцип: тестируйте с использованием только общедоступного API. Этот принцип направлен на то, чтобы направлять нас к тестированию моделей поведения, которые действительно имеют значение, а не к тестированию деталей реализации. Всякий раз, когда мы обнаруживаем, что делаем видимой приватную функцию, чтобы тесты могли ее вызывать, обычно это красный флаг, сигнализирующий о том, что мы нарушаем этот руководящий принцип.

В случае с классом `MortgageAssessor` поведение, которое действительно имеет значение, заключается в том, что заявка на ипотеку отклоняется для клиента с плохой кредитной историей. Мы можем протестировать это поведение, используя только общедоступный API, вызвав функцию `MortgageAssessor.assessment()`. В листинге 11.8 показано, как мог бы выглядеть контрольный пример, если бы мы так и сделали. Контрольный пример теперь проверяет поведение, которое на самом деле имеет значение, а не детали реализации, и нам больше не нужно делать какие-либо другие приватные функции в классе `MortgageAssessor` видимыми.

Листинг 11.8. Тестирование с помощью общедоступного API

```
testAssess_badCreditRating_mortgageRejected() {  
    Customer customer = new Customer(  
        income: new MonetaryAmount(50000, Currency.USD),  
        outgoings: new MonetaryAmount(25000, Currency.USD),  
        hasGoodCreditRating: false,  
        hasExistingMortgage: false,  
        isBanned: false);  
    MortgageAssessor mortgageAssessor = new MortgageAssessor();  
  
    MortgageDecision decision = mortgageAssessor.assess(customer); ❶  
  
    assertThat(decision.isApproved()).isFalse();  
}
```

❶ Поведение проверено с помощью общедоступного API.

БУДЬТЕ ПРАГМАТИЧНЫ

Сделать закрытую функцию видимой для тестирования — это почти всегда красный флаг, указывающий на то, что тестируются детали реализации, и обычно есть лучшая альтернатива. Но при применении принципа «тестируйте с использованием только общедоступного API» для других вещей (таких, как зависимости) важно помнить советы, приведенные в *главе 10 (разд. 10.3)*. Определение понятия «общедоступный API» может быть открыто для интерпретации, и некоторые важные модели поведения (например, побочные эффекты) могут выходить за рамки того, что программисты считают общедоступным API. Но если поведение важно и является чем-то, что нас, в конечном счете, интересует, то оно должно быть протестировано.

Для относительно простых классов (или модулей кода) часто очень легко протестировать все поведение, используя только общедоступный API. Выполнение этого приводит к улучшению тестов, которые будут более точно выявлять повреждения и не будут привязаны к деталям реализации. Но когда класс (или модуль кода) уже сложнее или содержит много логики, тестирование всего содержимого через общедоступный API может усложниться. Зачастую это является признаком того, что слой абстракции слишком толстый и что код может быть разделен на более мелкие модули.

11.2.3. Решение: разделите код на более мелкие модули

В предыдущих двух подразделах логика определения того, имеет ли клиент хорошую кредитную историю, была относительно простой: она всего лишь заключала в себе вызов `customer.hasGoodCreditRating()`, поэтому было не слишком сложно полностью протестировать класс `MortgageAssessor`, используя только общедоступный API. На самом деле соблазн сделать приватную функцию видимой для тестирования чаще возникает, когда приватная функция включает в себя более сложную логику.

Чтобы продемонстрировать это, представьте, что определение того, имеет ли клиент хорошую кредитную историю, заключает в себе вызов внешней службы и обработку результата. В листинге 11.9 показано, как в этом случае мог бы выглядеть класс `MortgageAssessor`. Логика проверки кредитной истории клиента теперь значительно усложнилась, о чем свидетельствует следующее:

- ◆ Класс `MortgageAssessor` теперь зависит от службы `CreditScoreService`.
- ◆ Служба `CreditScoreService` запрашивает идентификатор клиента для поиска его кредитной истории.
- ◆ Вызов функции `CreditScoreService.query()` может завершиться неудачей, поэтому код должен обработать этот сценарий ошибки.
- ◆ Если вызов проходит успешно, то возвращенная кредитная история сравнивается с пороговым значением, чтобы определить, является ли кредитная история клиента хорошей.

Тестирование всей этой сложной ситуации и всех этих критических случаев (таких, как сценарии ошибок) с помощью общедоступного API теперь является гораздо более сложным делом. Именно в этот момент программисты чаще всего прибегают к тому, чтобы сделать видимой приватную в иных ситуациях функцию и упростить тестирование. По этой причине в листинге 11.9 функция `isCreditRatingGood()` была сделана «видимой только для тестирования». Это по-прежнему влечет за собой все те же проблемы, которые мы видели ранее, но решение с тестированием через общедоступный API больше не кажется легко выполнимым, из-за того что логика очень сложна. Но, как мы сейчас увидим, здесь есть более фундаментальная проблема: класс `MortgageAssessor` делает слишком много вещей.

Листинг 11.9. Более сложная проверка кредитной истории

```

class MortgageAssessor {
    private const Double MORTGAGE_MULTIPLIER = 10.0;
    private const Double GOOD_CREDIT_SCORE_THRESHOLD = 880.0;

    private final CreditScoreService creditScoreService; ❶
    ...

    MortgageDecision assess(Customer customer) {
        ...
    }
    private Result<Boolean, Error> isEligibleForMortgage(
        Customer customer) {
        if (customer.hasExistingMortgage() || customer.isBanned()) {
            return Result.ofValue(false);
        }
        return isCreditRatingGood(customer.getId());
    }

    /** Видно только для тестирования */ ❷
    Result<Boolean, Error> isCreditRatingGood(Int customerId) {
        CreditScoreResponse response = creditScoreService ❸
            .query(customerId);
        if (response.errorOccurred()) {
            return Result.ofError(response.getError()); ❹
        }
        return Result.ofValue( ❺
            response.getCreditScore() >= GOOD_CREDIT_SCORE_THRESHOLD);
    }

    ...
}

```

❶ Класс `MortgageAssessor` зависит от службы `CreditScoreService`.

❷ Функция `isCreditRatingGood()` стала видимой для тестирования.

❸ Запрос в службу `CreditScoreService`.

❹ Сценарий ошибки при сбое вызова службы сигнализируется с помощью типа `Result`.

❺ Кредитная история сравнивается с пороговым значением.

Рис. 11.1 иллюстрирует взаимосвязь между тестовым кодом `MortgageAssessorTest` и кодом, подвергающимся тестированию `MortgageAssessor`.

В главе 2, обсуждая слои абстракции, мы видели, как порой лучше не помещать слишком много разных концепций в один класс. Класс `MortgageAssessor` содержит

множество различных концепций, поэтому на языке главы 2 слой абстракции, который он предоставляет, является «слишком толстым». Это настоящая причина, по которой кажется трудным полностью протестировать все с помощью общедоступного API.

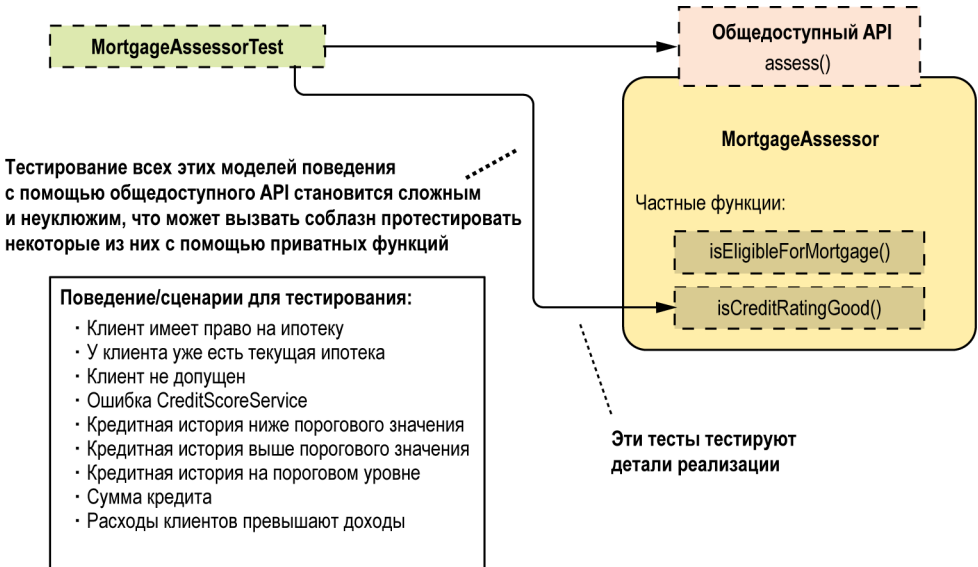


Рис. 11.1. Когда класс делает слишком много, может быть трудно протестировать все, используя только общедоступный API

Решение состоит в том, чтобы разделить код на более тонкие слои абстракции. Один из способов добиться этого — перенести логику определения того, имеет ли клиент хорошую кредитную историю, в отдельный класс. В листинге 11.10 показано, как может выглядеть этот класс. Класс `RatingChecker` решает подзадачу определения того, имеет ли клиент хорошую кредитную историю. Класс `MortgageAssessor` зависит от `CreditRatingChecker`, что означает, что он значительно упрощен, поскольку он больше не содержит всей основной логики для решения подзадач.

Листинг 11.10. Код разделен на два класса

```
class CreditRatingChecker { ❶
    private const Double GOOD_CREDIT_SCORE_THRESHOLD = 880.0;

    private final CreditScoreService creditScoreService;
    ...

    Result<Boolean, Error> isCreditRatingGood(Int customerId) {
        CreditScoreService response = creditScoreService
            .query(customerId);
```

```

    if (response.errorOccurred()) {
        return Result.ofError(response.getError());
    }
    return Result.ofValue(
        response.getCreditScore() >= GOOD_CREDIT_SCORE_THRESHOLD);
}
}

class MortgageAssessor {
    private const Double MORTGAGE_MULTIPLIER = 10.0;

    private final CreditRatingChecker creditRatingChecker; ❷
    ...

    MortgageDecision assess(Customer customer) {
        ...
    }

    private Result<Boolean, Error> isEligibleForMortgage(
        Customer customer) {
        if (customer.hasExistingMortgage() || customer.isBanned()) {
            return Result.ofValue(false);
        }
        return creditRatingChecker
            .isCreditRatingGood(customer.getId()); ❷
    }
    ...
}

```

❶ Отдельный класс, содержащий логику для проверки того, является ли кредитная история хорошей.

❷ Класс `MortgageAssessor` зависит от класса `CreditRatingChecker`.

Как класс `MortgageAssessor`, так и `CreditRatingChecker` имеют дело с гораздо более управляемым количеством концепций. Это означает, что оба они могут быть легко протестированы с использованием соответствующих общедоступных API, как показано на рис. 11.2.

Когда мы обнаруживаем, что делаем приватную функцию видимой, чтобы можно было протестировать код, обычно это предупреждающий знак, сообщающий о том, что мы не тестируем поведение, которое нас действительно интересует. Почти всегда лучше тестировать код, используя функции, уже являющиеся общедоступными. Если это невозможно, то это обычно признак того, что класс (или модуль кода) слишком велик и что нам следует подумать о разделении его на более мелкие классы (или модули), каждый из которых решает только одну подзадачу.

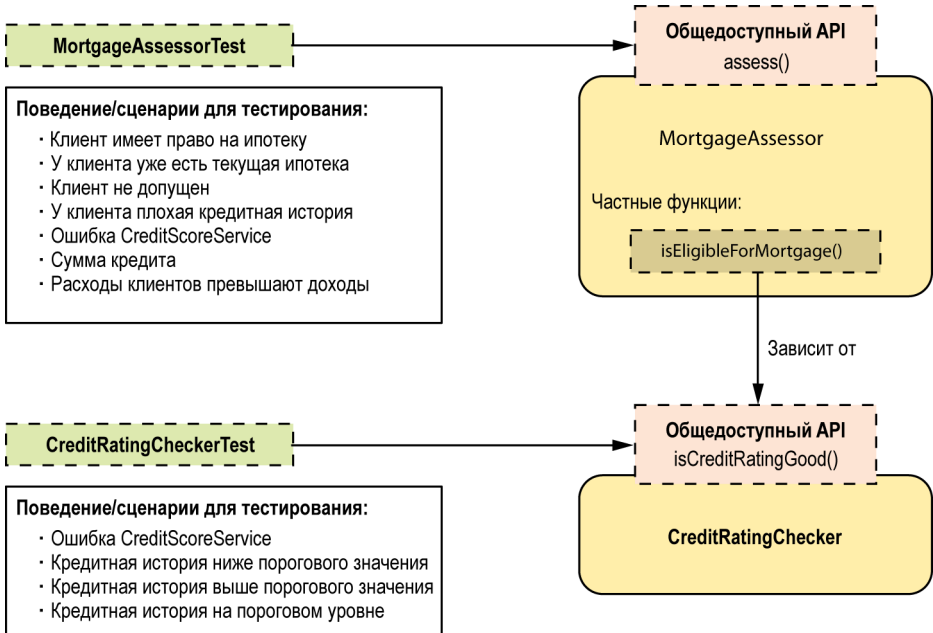


Рис. 11.2. Разделение большого класса на мелкие классы может сделать код более тестируемым

11.3. Тестируйте одну модель поведения за раз

Как мы уже видели, зачастую существует несколько вариантов поведения, которые необходимо протестировать для некоторого фрагмента кода. Во многих случаях для каждого из этих вариантов поведения требуется настроить отдельный сценарий, чтобы его протестировать, а это означает, что наиболее естественным будет протестировать каждый сценарий (и связанное с ним поведение) в своем собственном контрольном примере. Однако иногда может существовать способ придумать один сценарий, который проверяет несколько вариантов поведения за один раз. Но только потому, что мы можем это сделать, не означает, что это хорошая идея.

11.3.1. Одновременное тестирование нескольких моделей поведения может привести к некачественным тестам

В листинге 11.11 показан код функции для фильтрации только действительных купонов из списка. Функция принимает список купонов-кандидатов и возвращает другой список, содержащий только те, которые соответствуют набору критериев, для того чтобы являться действительными. Эта функция демонстрирует ряд важных моделей поведения.

- ◆ Возвращаются только действительные купоны.
- ◆ Купон считается недействительным, если он уже был погашен.

- ◆ Купон считается недействительным, если срок его действия истек.
- ◆ Купон считается недействительным, если он был выдан другому клиенту, а не тому, который был указан при вызове функции.
- ◆ Возвращенный список купонов отсортирован в порядке убывания ценности.

Листинг 11.11. Код для получения действительных купонов

```
List<Coupon> getValidCoupons(  
    List<Coupon> coupons, Customer customer) {  
    return coupons  
        .filter(coupon -> !coupon.alreadyRedeemed())  
        .filter(coupon -> !coupon.hasExpired())  
        .filter(coupon -> coupon.issuedTo() == customer)  
        .sortBy(coupon -> coupon.getValue(), SortOrder.DECENDING);  
}
```

Как мы уже обсуждали, важно тестировать каждый вариант поведения фрагмента кода, и функция `getValidCoupons()` не является исключением из этого правила. Один из подходов, воспользоваться которым для нас будет соблазнительно, состоит в том, чтобы написать один огромный контрольный пример, который проверяет все модели поведения функции за один раз. В листинге 11.12 показано, как это может выглядеть. Первое, на что следует обратить внимание: довольно трудно понять, что именно делает контрольный пример. Название `testGetValidCoupons_allBehaviors` не очень точно описывает то, что именно тестируется, а объем кода в контрольном примере делает его довольно сложным для понимания. В *главе 10* мы определили *понятный тестовый код* как одну из ключевых особенностей хорошего модульного теста. Мы сразу видим, что тестирование всех вариантов поведения за один раз, подобное этому, не соответствует данному критерию.

Листинг 11.12. Тестирование всего сразу

```
void testGetValidCoupons_allBehaviors() {  
    Customer customer1 = new Customer("test customer 1");  
    Customer customer2 = new Customer("test customer 2");  
    Coupon redeemed = new Coupon(  
        alreadyRedeemed: true, hasExpired: false,  
        issuedTo: customer1, value: 100);  
    Coupon expired = new Coupon(  
        alreadyRedeemed: false, hasExpired: true,  
        issuedTo: customer1, value: 100);  
    Coupon issuedToSomeoneElse = new Coupon(  
        alreadyRedeemed: false, hasExpired: false,  
        issuedTo: customer2, value: 100);  
    Coupon valid1 = new Coupon(  
        alreadyRedeemed: false, hasExpired: false,  
        issuedTo: customer1, value: 100);
```

```
Coupon valid2 = new Coupon(  
    alreadyRedeemed: false, hasExpired: false,  
    issuedTo: customer1, value: 150);  
  
List<Coupon> validCoupons = getValidCoupons(  
    [redeemed, expired, issuedToSomeoneElse, valid1, valid2],  
    customer1);  
  
assertThat(validCoupons)  
    .containsExactly(valid2, valid1)  
    .inOrder();  
}
```

Тестирование всех моделей поведения за один раз также не соответствует еще одному критерию, который мы определили в *главе 10*: необходимы хорошо объясненные сбои. Чтобы понять почему, давайте рассмотрим, что произойдет, если программист случайно сломает одну из моделей поведения функции `getValidCoupons()`, удалив логику проверки того, был ли купон погашен. Контрольный пример `testGetValidCoupons_allBehaviors()` завершится неудачей, что хорошо (потому что код поврежден), но сообщение об ошибке не будет особенно полезным для объяснения того, какая именно модель поведения была повреждена (рис. 11.3).

Поскольку контрольный пример проверяет все модели поведения, мы не можем определить, какая именно модель поведения повреждена, взглянув на название контрольного примера

```
Test case testGetValidCoupons_allBehaviors failed:  
Expected:  
[  
    Coupon(redeemed: false, expired: false,  
            issuedTo: test customer 1, value: 150),  
    Coupon(redeemed: false, expired: false,  
            issuedTo: test customer 1, value: 100)  
]  
But was actually:  
[  
    Coupon(redeemed: false, expired: false,  
            issuedTo: test customer 1, value: 150),  
    Coupon(redeemed: true, expired: false,  
            issuedTo: test customer 1, value: 100),  
    Coupon(redeemed: false, expired: false,  
            issuedTo: test customer 1, value: 100)  
]
```

Довольно сложно определить, какая модель поведения повреждена из сообщения об ошибке

Рис. 11.3. Тестирование нескольких вариантов поведения за один раз может привести к плохо объясненным сбоям в тестировании

Наличие трудного для понимания тестового кода и плохо объясненных сбоев не только отнимает время у других программистов, но и может увеличить вероятность ошибок. Как обсуждалось в *главе 10*, если какой-либо программист намеренно изменяет одно из поведений кода, тогда желательно удостовериться, что другие, казалось бы несвязанные, модели поведения также не будут случайно затронуты. Один контрольный пример, который проверяет все за один раз, как правило, говорит нам только о том, что что-то изменилось, а не о том, что именно изменилось, поэтому гораздо сложнее быть уверенным в том, на какой вариант поведения повлияло или не повлияло преднамеренное изменение.

11.3.2. Решение: протестируйте каждую модель поведения в своем собственном контрольном примере

Подход получше состоит в том, чтобы тестировать каждое поведение отдельно, используя специальный, хорошо названный контрольный пример. В листинге 11.13 показано, как может выглядеть тестовый код, если мы так сделаем. Мы видим, что код внутри каждого контрольного примера теперь намного проще и понятнее. По названию каждого контрольного примера мы можем точно определить, какая модель поведения тестируется, и относительно легко следовать коду, чтобы увидеть, как работает тест. Теперь тесты гораздо больше соответствуют критерию о том, что модульные тесты должны иметь понятный тестовый код.

Листинг 11.13. Тестирование одной вещи за раз

```
void testGetValidCoupons_validCoupon_included() { ❶
    Customer customer = new Customer("test customer");
    Coupon valid = new Coupon(
        alreadyRedeemed: false, hasExpired: false,
        issuedTo: customer, value: 100);

    assertThat(validCoupons).containsExactly(valid);
}

void testGetValidCoupons_alreadyRedeemed_excluded() { ❶
    Customer customer = new Customer("test customer");
    Coupon redeemed = new Coupon(
        alreadyRedeemed: true, hasExpired: false,
        issuedTo: customer, value: 100);

    List<Coupon> validCoupons =
        getValidCoupons([redeemed], customer);

    assertThat(validCoupons).isEmpty();
}
```

```
void testGetValidCoupons_expired_excluded() { ... } ❶

void testGetValidCoupons_issuedToDifferentCustomer_excluded() { ... }
void testGetValidCoupons_returnedInDescendingValueOrder() { ... }
```

❶ Каждая модель поведения проверяется в специальном контрольном примере.

Тестируя каждую модель поведения отдельно и используя соответствующее название для каждого контрольного случая, мы теперь добьемся и хорошо объясненных сбоев. Давайте еще раз рассмотрим сценарий, в котором программист случайно повреждает функцию `getValidCoupons()`, удаляя логику проверки того, был ли погашен купон. Это приведет к тому, что контрольный пример `testGetValidCoupons_alreadyRedeemed_excluded()` завершится неуспехом. Название этого контрольного примера дает понять, какое именно поведение было повреждено, и сообщение об ошибке (рис. 11.4) гораздо легче понять, чем то, которое мы видели ранее.

Несмотря на преимущества тестирования одной вещи за раз, написание отдельной функции контрольного примера для каждого варианта поведения иногда может привести к излишнему дублированию кода. Это может показаться особенно неуклюжим, когда значения и настройки, используемые в каждом контрольном примере, почти идентичны, за исключением некоторых незначительных различий. Одним из способов уменьшить такой объем дублирования кода является использование *параметризованных тестов*. Это рассматривается в следующем подразделе.

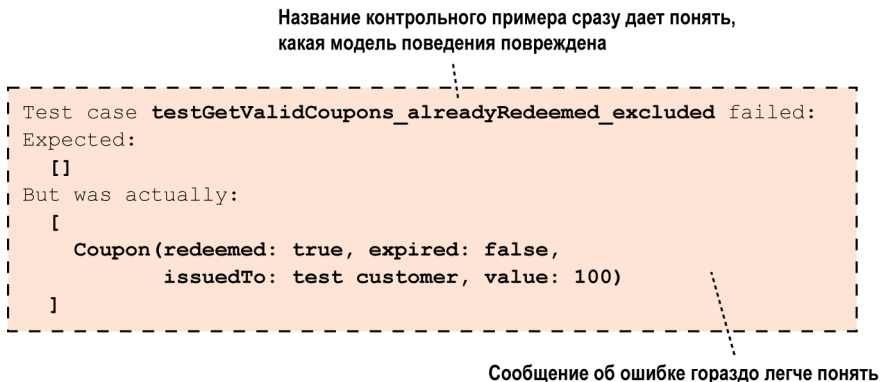


Рис. 11.4. Тестирование одной модели поведения за раз приводит к хорошо объясненным сбоям в тестировании

11.3.3. Параметризованные тесты

Некоторые фреймворки для тестирования предоставляют функциональные возможности для написания параметризованных тестов; это позволяет нам написать функцию контрольного примера один раз, но затем запустить ее несколько раз с разными

наборами значений для тестирования разных сценариев. В листинге 11.14 показано, как мы могли бы использовать параметризованный тест для проверки двух моделей поведения функции `getValidCoupons()`. Функция контрольного примера помечена несколькими атрибутами `TestCase`. Каждый из них определяет два значения `Boolean` и название теста. В функции `testGetValidCoupons_excludesInvalidCoupons()` имеется два параметра функции `Boolean`; они соответствуют двум значениям `Boolean`, определенным в атрибутах `TestCase`. При выполнении тестов контрольный пример будет выполняться один раз для каждого набора значений параметров, определенных в атрибутах контрольного примера.

Листинг 11.14. Параметризованный тест

```
[TestCase(true, false, TestName = "alreadyRedeemed")] ❶
[TestCase(false, true, TestName = "expired")]
void testGetValidCoupons_excludesInvalidCoupons(
    Boolean alreadyRedeemed, Boolean hasExpired) { ❷
    Customer customer = new Customer("test customer");
    Coupon coupon = new Coupon(
        alreadyRedeemed: alreadyRedeemed, ❸
        hasExpired: hasExpired,
        issuedTo: customer, value: 100);

    List<Coupon> validCoupons =
        getValidCoupons([coupon], customer);

    assertThat(validCoupons).isEmpty();
}
```

❶ Контрольный пример будет выполняться один раз для каждого набора значений параметров.

❷ Контрольный пример принимает различные значения с помощью параметров функции.

❸ Значения параметров используются во время тестовой настройки.

УБЕДИТЕСЬ, ЧТО СБОИ ХОРОШО ОБЪЯСНЕНЫ

В листинге 11.14 каждому набору параметров соответствует `TestName`. Это гарантирует, что любые сбои в тестировании будут хорошо объяснены, поскольку приводит к появлению таких сообщений, как `Test case testGetValidCoupons_excludesInvalidCoupons.alreadyRedeemed failed`. (Обратите внимание, что название контрольного примера имеет суффикс с названием набора параметров, которые привели к сбою `alreadyRedeemed`.)

Добавление названий для каждого набора параметров обычно необязательно при написании параметризованных тестов. Но их опущение может привести к плохо объясненным сбоям в тестировании, поэтому полезно подумать о том, как будут выглядеть сбои в тестировании, если вы решите их не добавлять.

Параметризованные тесты могут быть отличным инструментом для обеспечения того, чтобы все модели поведения тестировались по одной за раз, не повторяя большое количество кода. Синтаксис и способ настройки параметризованных тестов могут сильно различаться в разных фреймворках для тестирования. Настройка параметризованных тестов также может быть невероятно детальной и неуклюжей в некоторых фреймворках и сценариях, поэтому стоит изучить варианты для любого языка, который вы используете, и рассмотреть все плюсы и минусы. Вот некоторые варианты.

- ◆ Для C# фреймворк тестирования `JUnit` предоставляет атрибут `TestCase` (аналогично примеру в листинге 11.14): <http://mng.bz/qewE>.
- ◆ Для Java `JUnit` обеспечивает поддержку параметризованных тестов: <http://mng.bz/1Ayy>.
- ◆ Для JavaScript с помощью тестового фреймворка `Jasmine` относительно легко писать параметризованные тесты на заказ, как описано в статье <http://mng.bz/PaQg>.

11.4. Используйте совместную настройку тестирования надлежащим образом

Контрольные примеры часто требуют настройки: построения зависимостей, заполнения значений в хранилище тестовых данных или инициализации разных других состояний. Эта настройка иногда может быть довольно трудоемкой или дорогостоящей с точки зрения вычислений, и поэтому многие фреймворки для тестирования предоставляют функциональные возможности, облегчающие совместное использование настроечного кода (англ. *setup code*) с разными контрольными примерами.

Как правило, речь идет о двух различных ситуациях, в которых совместно использованный настроечный код может быть сконфигурирован и запущен; эти ситуации различаются следующим образом:

- `BeforeAll` — настроечный код внутри блока `BeforeAll` будет выполняться один раз перед запуском любого из контрольных примеров. Некоторые фреймворки для тестирования называют это `OneTimeSetUp`;
- `BeforeEach` — настроечный код внутри блока `BeforeEach` будет выполняться один раз перед запуском каждого контрольного примера. Некоторые фреймворки для тестирования называют это просто `SetUp`.

В дополнение к предоставлению способов запуска настроечного кода фреймворки также часто предоставляют способы запуска кода переналадки (англ. *teardown code*). Такой код может быть полезен для отмены любого состояния, которое было создано настроечным кодом или контрольными примерами. И опять же обычно существуют две разных ситуации, в которых код для переналадки может быть сконфигурирован и запущен.

Эти ситуации различаются следующим образом:

- **AfterAll** — код переналадки внутри блока **AfterAll** будет выполняться один раз после выполнения всех контрольных примеров. Некоторые фреймворки для тестирования называют это **OneTimeTearDown**;
- **AfterEach** — код переналадки внутри блока **AfterEach** будет выполняться один раз после запуска каждого контрольного примера. Некоторые фреймворки для тестирования называют это просто **TearDown**.

Рис. 11.5 иллюстрирует, как могут выглядеть эти различные коды для настройки и переналадки в некотором тестовом коде, а также последовательность, в которой они будут выполняться.

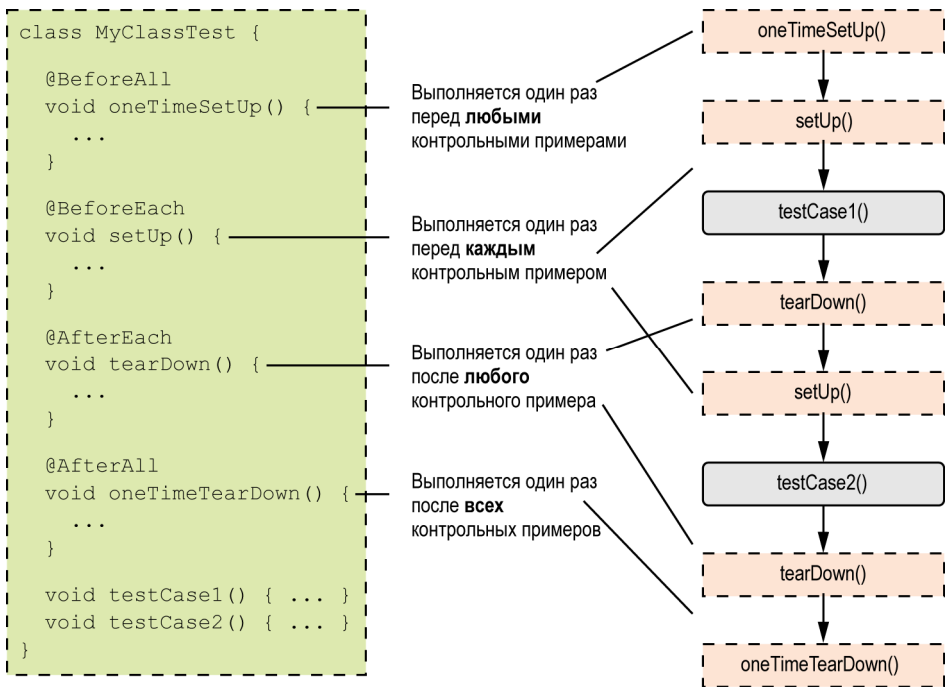


Рис. 11.5. Фреймворки для тестирования нередко предоставляют способ запуска кодов для настройки и переналадки в разных ситуациях, соотносящихся с контрольными примерами

Использование подобных блоков настроечного кода приводит к тому, что настройка распределяется между различными контрольными примерами. Это может произойти двумя важными, но отличными друг от друга способами.

1. **Состояние совместного использования.** Если настроечный код добавлен в блок **BeforeAll**, он будет выполняться один раз перед всеми контрольными примерами. Это означает, что любое состояние, которое он устанавливает, будет совместно использовано между всеми контрольными примерами. Этот тип настройки

может быть полезен, когда настройка выполняется медленно или является дорогостоящей (например, при запуске тестового сервера или создании контрольного экземпляра базы данных). Но если настраиваемое состояние изменчиво, то существует реальный риск того, что контрольные примеры могут оказать неблагоприятное влияние друг на друга (мы скоро рассмотрим это подробнее).

2. *Конфигурация совместного использования.* Если настроочный код добавлен в блок `BeforeEach`, он будет выполняться перед каждым контрольным примером, что означает, что все контрольные примеры имеют общую конфигурацию, которую устанавливает код. Если этот настроочный код содержит определенное значение или настраивает зависимость определенным образом, то каждый контрольный пример будет выполняться с этим заданным значением или зависимостью, настроенной таким образом. Поскольку настройка выполняется перед каждым контрольным случаем, контрольные случаи не обладают совместным состоянием. Но, как мы скоро увидим (в разд. 11.4.3), конфигурация совместного использования все еще может быть проблематичной.

Если настройка какого-либо конкретного состояния или зависимости является дорогостоящей, использование совместной настройки может быть необходимостью. Даже если это не так, совместная настройка может быть полезным способом упрощения тестов. Если для каждого контрольного примера требуется определенная зависимость, то может быть полезно настроить ее совместным способом, а не повторять множество шаблонов в каждом контрольном примере. Но совместная настройка тестирования может быть обоюдоострым мечом; неправильное ее использование может привести к хрупким и неэффективным тестам.

11.4.1. Совместное состояние может быть проблематичным

Как правило, тестовые контрольные должны быть изолированы друг от друга, поэтому любые действия, выполняемые одним контрольным примером, не должны влиять на результаты других контрольных примеров. Совместное использование изменяемого состояния разными контрольными примерами позволяет очень легко непреднамеренно нарушить это правило.

Чтобы продемонстрировать это, в листинге 11.15 показана часть класса и функции для обработки заказа. Сосредоточимся на двух вариантах поведения:

- ◆ если заказ содержит товар, которого нет на складе, идентификатор заказа будет отмечен в базе данных как отложенный;
- ◆ если оплата заказа еще не завершена, идентификатор заказа будет отмечен в базе данных как отложенный.

Листинг 11.15. Код, делающий запись в базу данных

```
class OrderManager {  
    private final Database database;  
    ...  
}
```

```

void processOrder(Order order) {
    if (order.containsOutOfStockItem() ||
        !order.isPaymentComplete()) {
        database.setOrderStatus(
            order.getId(), OrderStatus.DELAYED);
    }
    ...
}
}

```

Модульные тесты должны содержать контрольный пример для каждого из этих видов поведения (они показаны в листинге 11.16). Класс `OrderManager` зависит от класса `Database`, поэтому в наших тестах необходимо настроить один из них. К сожалению, создание экземпляра `Database` является дорогостоящей и медленной процедурой с точки зрения вычислений, поэтому мы создаем его в блоке `BeforeAll`. Это означает, что один и тот же экземпляр базы данных используется совместно всеми контрольными примерами (имеется в виду, что контрольные примеры совместно используют состояние). Это также делает тесты неэффективными. Чтобы понять почему, рассмотрим последовательность событий, происходящих при выполнении тестов.

- ◆ Блок `BeforeAll` настраивает базу данных.
- ◆ Запускается контрольный пример `testProcessOrder_outOfStockItem_orderDelayed()`. Это приводит к тому, что идентификатор заказа помечается в базе данных как отложенный.
- ◆ Затем запускается контрольный пример `testProcessOrder_paymentNotComplete_orderDelayed()`. Все, что предыдущие контрольные примеры помещали в базу данных, до сих пор присутствует (потому что состояние является совместным), поэтому, когда вызывается тестируемый код, может произойти одно из двух:
 - все работает правильно, и идентификатор заказа помечается как отложенный. Контрольный пример завершается успехом;
 - код поврежден, он ничего не сохраняет в базе данных, чтобы пометить идентификатор заказа как отложенный. Поскольку код поврежден, мы надеемся, что контрольный пример завершится неудачей. Но вместо этого он завершается успехом, потому что `database.getOrderStatus(orderId)` все еще возвращает значение `DELAYED`, поскольку в предыдущем тестовом примере это значение было сохранено в базе данных.

Листинг 11.16. Состояние, совместно используемое контрольными примерами

```

class OrderManagerTest {
    private Database database;

    @BeforeAll ❶
    void oneTimeSetUp() {

```

```
database = Database.createInstance();
database.waitForReady();
}

void testProcessOrder_outOfStockItem_orderDelayed() {
    Int orderId = 12345;
    Order order = new Order(
        orderId: orderId,
        containsOutOfStockItem: true,
        isPaymentComplete: true);
    OrderManager orderManager = new OrderManager(database); ❷

    orderManager.processOrder(order); ❸

    assertThat(database.getOrderStatus(orderId))
        .isEqualTo(OrderStatus.DELAYED);
}

void testProcessOrder_paymentNotComplete_orderDelayed()
{
    Int orderId = 12345;
    Order order = new Order(
        orderId: orderId,
        containsOutOfStockItem: false,
        isPaymentComplete: false);
    OrderManager orderManager = new OrderManager(database); ❹

    orderManager.processOrder(order);

    assertThat(database.getOrderStatus(orderId)) ❺
        .isEqualTo(OrderStatus.DELAYED);
}
...
}
```

❶ Один и тот же экземпляр базы данных используется совместно всеми контрольными примерами.

❷ OrderManager создан с использованием совместной базы данных.

❸ Приводит к тому, что идентификатор заказа помечается в базе данных как отложенный.

❹ OrderManager создан с использованием совместной базы данных.

❺ Может завершиться успехом, даже если код поврежден, потому что предыдущий контрольный пример сохранил это значение в базе данных.

Совместное использование изменяемого состояния различными контрольными примерами может очень легко привести к проблемам. Если это возможно, то лучше избегать совместного использования такого состояния. Но если это необходимо,

мы должны быть очень осторожны, чтобы убедиться, что изменения, вносимые одним контрольным примером в состояние, не влияют на другие контрольные примеры.

11.4.2. Решение: избегайте совместного использования состояния или сбросьте его

Наиболее очевидное решение проблемы совместного использования изменяемого состояния состоит в том, чтобы просто не использовать его совместно. В случае с `OrderManagerTest` было бы лучше, если бы мы не использовали один и тот же экземпляр `Database` в разных контрольных примерах, поэтому если настройка `Database` происходит медленнее, чем мы думали, тогда мы можем рассмотреть возможность создания нового экземпляра для каждого контрольного примера (либо в рамках контрольных примеров, либо с использованием блока `BeforeEach`).

Другим потенциальным способом избежать совместного использования изменяемого состояния является использование имитированной реализации (как обсуждалось в *главе 10*). Такое решение будет полезным, если команда, которая поддерживает класс `Database`, также напишет и класс `FakeDatabase` для использования в тестировании. Создать экземпляр `FakeDatabase`, скорее всего, удастся достаточно быстро, и мы сможем сделать новый экземпляр для каждого контрольного примера, что означает, что состояние не является совместным.

Если создание экземпляра `Database` займет слишком много времени и будет слишком затратным (и мы не можем использовать подделку), тогда совместное использование ее экземпляра контрольными примерами вполне может быть неизбежным. Если это так, мы должны быть очень осторожны, чтобы убедиться, что состояние сбрасывается между каждым контрольным примером. Этого можно достичь с помощью блока `AfterEach` в тестовом коде. Как упоминалось ранее, это будет выполняться после каждого контрольного примера, поэтому мы можем использовать это для обеспечения того, чтобы состояние всегда сбрасывалось перед запуском следующего контрольного примера. В следующем листинге показано, как может выглядеть тест `OrderManagerTest`, если мы используем блок `AfterEach` для сброса базы данных между контрольными примерами.

Листинг 11.17. Сброс состояния между контрольными примерами

```
class OrderManagerTest {
    private Database database;
    @BeforeAll
    void oneTimeSetUp() {
        database = Database.createInstance();
        database.waitForReady();
    }

    @AfterEach ❶
    void tearDown() {
```

```
database.reset();  
}  
  
void testProcessOrder_outOfStockItem_orderDelayed() { ... } ❷  
  
void testProcessOrder_paymentNotComplete_orderDelayed() { ... }  
...  
}
```

❶ База данных сбрасывается после каждого контрольного примера.

❷ На контрольные примеры никогда не будут влиять значения, сохраненные другими контрольными примерами.

ГЛОБАЛЬНОЕ СОСТОЯНИЕ

Стоит отметить, что тестовый код — не единственный способ совместного использования состояния контрольными примерами. Если тестируемый код поддерживает какое-либо глобальное состояние, нам нужно убедиться, что тестовый код сбрасывает его между контрольными случаями. Глобальное состояние обсуждалось в *главе 9*, и был сделан вывод, что лучше его избегать. Влияние, которое глобальное состояние может оказать на тестируемость кода, является еще одной веской причиной для отказа от его использования.

Совместное использование изменяемого состояния контрольными примерами — вещь далеко не идеальная. Предпочтительнее этого избегать, если такое возможно. Если избежать этого нельзя, необходимо убедиться, что состояние сбрасывается между каждым контрольным случаем. Это гарантирует, что контрольные примеры не окажут неблагоприятного влияния друг на друга.

11.4.3. Совместная конфигурация может быть проблематичной

Совместное использование конфигурации контрольными примерами поначалу не кажется таким опасным, как совместное использование состояния, но это все равно может привести к неэффективным тестам. Представьте, что еще одной частью нашей инфраструктуры обработки заказов является система, которая генерирует почтовые ярлыки для посылок. Листинг 11.18 содержит функцию, которая генерирует объект данных для представления почтового ярлыка для заказа. Есть несколько важных поведенческих характеристик, которые нам необходимо протестировать, но мы сосредоточимся на том, помечена ли упаковка как крупная. Логика этого довольно проста: если заказ содержит более двух позиций, посылка считается крупной.

Листинг 11.18. Код для почтового ярлыка

```
class OrderPostageManager {  
    ...  
    PostageLabel getPostageLabel(Order order) {  
        return new PostageLabel(  
            address: order.getCustomer().getAddress(),  

```

```

        isLargePackage: order.getItems().size() > 2, ❶
    );
}
}

```

❶ Если заказ содержит более двух позиций, посылка помечается как крупная.

Если мы сосредоточимся только на поведении `isLargePackage`, то нам понадобятся контрольные примеры, как минимум, для двух разных сценариев:

- *заказ, содержащий две позиции.* Это должно привести к тому, что посылка не будет помечена как крупная;
- *заказ, содержащий три позиции.* Это должно привести к тому, что посылка будет помечена как крупная.

Если кто-то непреднамеренно изменит логику в коде для определения того, какое количество элементов делает посылку крупной, то один из этих контрольных примеров должен завершиться неудачей. Давайте теперь представим, что создание действительного экземпляра класса `Order` является более трудоемким, чем в предыдущих подразделах: нам нужно предоставить экземпляры класса `Item` и экземпляры класса `Customer`, что также означает создание экземпляра класса `Address`. Чтобы избавить себя от повторения этого кода конфигурации в каждом контрольном примере, мы решили создать экземпляр класса `Order` в блоке `BeforeEach` (который выполняется один раз перед каждым контрольным примером). В листинге 11.19 показано, как это выглядит. В контрольном примере, который тестирует сценарий, когда в заказе есть три позиции, используется экземпляр класса `Order`, созданный при помощи совместной конфигурации. Таким образом, контрольный пример `testGetPostageLabel_threeItems_largePackage()` основывается на том факте, что совместная конфигурация создает заказ, содержащий ровно три позиции.

Листинг 11.19. Конфигурация совместного тестирования

```

class OrderPostageManagerTest {
    private Order testOrder;

    @BeforeEach ❶
    void setUp() {
        testOrder = new Order(
            customer: new Customer(
                address: new Address("Test address"),
            ),
            items: [
                new Item(name: "Test item 1"),
                new Item(name: "Test item 2"),
                new Item(name: "Test item 3"),
            ]
        );
    }
    ...
}

```

```

void testGetPostageLabel_threeItems_largePackage() {
    PostageManager postageManager = new PostageManager();

    PostageLabel label = ❷
        postageManager.getPostageLabel(testOrder);

    assertThat(label.isLargePackage()).isTrue();
}
...
}

```

❶ Совместная конфигурация.

❷ Контрольный пример полагается на то, что совместная конфигурация добавляет в заказ ровно три позиции.

Так тестируется один из тех вариантов поведения, которые нас интересуют, и это позволяет избежать необходимости повторять загрузку неуклюжего кода для создания `Order` в каждом контрольном примере. Но, к сожалению, все может пойти не так, если другим программистам когда-нибудь понадобится изменить тесты. Представьте, что другому программисту теперь нужно добавить новую функциональность в функцию `getPostageLabel()`: если какие-либо товары в заказе являются опасными, на почтовом ярлыке должно быть указано, что посылка является опасной. Программист изменяет функцию `getPostageLabel()`, чтобы она выглядела так, как показано в следующем листинге.

Листинг 11.20. Новая функциональность

```

class PostageManager {
    ...
    PostageLabel getPostageLabel(Order order) {
        return new PostageLabel(
            address: order.getCustomer().getAddress(),
            isLargePackage: order.getItems().size() > 2,
            isHazardous: containsHazardousItem(order.getItems()); ❶
        )
    }

    private static Boolean containsHazardousItem(List<Item> items) {
        return items.anyMatch(item -> item.isHazardous());
    }
}

```

❶ Новая функциональность для определения того, является ли посылка опасной.

Программист добавил новое поведение в код, поэтому ему необходимо добавить и новые контрольные примеры для проверки этого поведения. Программист видит, что в блоке `BeforeEach` был создан экземпляр `Order`, и думает: «Как здорово! Я могу просто добавить опасную позицию в этот заказ и использовать его в одном из моих контрольных примеров». В листинге 11.21 показан тестовый код, после того как

программист так поступит. У программиста получилось протестировать новое поведение, но он непреднамеренно повредил контрольный пример `testGetPostageLabel_threeItems_largePackage()`. Весь смысл этого контрольного примера в том, что он проверяет, что происходит, когда в заказе ровно три позиции, но теперь он проверяет, что происходит, когда есть четыре позиции, поэтому тест больше не полностью защищает код от повреждения.

Листинг 11.21. Неудачное изменение совместной конфигурации

```
class OrderPostageManagerTest {
    private Order testOrder;

    @BeforeEach
    void setUp() {
        testOrder = new Order(
            customer: new Customer(
                address: new Address("Test address"),
            ),
            items: [
                new Item(name: "Test item 1"),
                new Item(name: "Test item 2"),
                new Item(name: "Test item 3"),
                new Item(name: "Hazardous item", isHazardous: true), ❶
            ]
        );
    }
    ...

    void testGetPostageLabel_threeItems_largePackage() { ... } ❷

    void testGetPostageLabel_hazardousItem_isHazardous() { ❸
        PostageManager postageManager = new PostageManager();

        PostageLabel label =
            postageManager.getPostageLabel(testOrder);

        assertThat(label.isHazardous()).isTrue();
    }
    ...
}
```

❶ Четвертая товарная позиция, добавленная в заказ в совместной конфигурации.

❷ Теперь тестируется пример с четырьмя позициями, а не предполагаемый пример с тремя позициями.

❸ Новый контрольный пример для проверки того, что на ярлыке есть пометка о содержании опасной позиции.

Блоки `BeforeEach` или `BeforeAll` — не единственные способы создания совместной тестовой конфигурации. Использование совместной тестовой константы может привести к точно такому же результату, а также страдать от одного и того же набора потенциальных проблем, которые мы только что обсудили. Если `OrderPostageManagerTest` сконфигурировал порядок тестирования в совместной константе вместо блока `BeforeEach`, это может выглядеть следующим образом:

```
class OrderPostageManagerTest {  
    private const Order TEST_ORDER = new Order( ❶  
        customer: new Customer(  
            address: new Address("Test address"),  
        ),  
        items: [  
            new Item(name: "Test item 1"),  
            new Item(name: "Test item 2"),  
            new Item(name: "Test item 3"),  
            new Item(name: "Hazardous item", isHazardous: true),  
        ]  
    );  
    ...  
}
```

❶ Совместная тестовая константа.

Технически это также позволяет совместное использование состояния разными контрольными примерами. Однако рекомендуется создавать константы только с использованием неизменяемых типов данных, что означает, что изменяемые состояния никогда не являются совместными. В этом примере класс `Order` является неизменяемым. Если бы он не был неизменяемым, то совместное использование экземпляра `Order` в совместной константе было бы, скорее всего, еще менее удачной идеей (по причинам, обсуждавшимся в *разд. 11.4.1*).

Совместная конфигурация может быть полезна для предотвращения повторения кода, но обычно лучше не использовать ее для настройки каких-либо значений или состояний, которые особенно важны для контрольных примеров. Очень сложно точно отследить, какие контрольные примеры зависят от конкретных элементов в совместной конфигурации, и когда в будущем будут внесены изменения, это может привести к тому, что контрольные примеры больше не будут тестировать то, что должны.

11.4.4. Решение: определить, какая конфигурация важна в контрольных примерах

Может показаться трудоемким повторять конфигурацию в каждом контрольном примере, но когда контрольный пример зависит от определенных значений или настройки состояния, это бывает безопаснее. Как правило, это можно сделать менее трудоемким, используя вспомогательные функции, чтобы нам не приходилось повторять много шаблонного кода.

В случае тестирования функции `getPostageLabel()` создание экземпляра класса `Order` казалось довольно неуклюжим, но его создание в совместной конфигурации привело к проблемам, которые мы увидели в предыдущем подразделе. В основном мы избежим этих проблем, определив вспомогательную функцию для создания экземпляра `Order`. Отдельные контрольные примеры могут затем вызывать эту функцию с конкретными тестовыми значениями, которые их интересуют. Это позволяет избежать большого количества повторений кода без необходимости использовать совместную конфигурацию и страдать от проблем, которые могут при этом возникнуть. В следующем листинге показано, как выглядит тестовый код при таком подходе.

Листинг 11.22. Важная конфигурация в контрольных примерах

```
class OrderPostageManagerTest {
    ...
    void testGetPostageLabel_threeItems_largePackage() { ❶
        Order order = createOrderWithItems([
            new Item(name: "Test item 1"),
            new Item(name: "Test item 2"),
            new Item(name: "Test item 3"),
        ]);
        PostageManager postageManager = new PostageManager();
        PostageLabel label = postageManager.getPostageLabel(order);

        assertThat(label.isLargePackage()).isTrue();
    }

    void testGetPostageLabel_hazardousItem_isHazardous() {
        Order order = createOrderWithItems([
            new Item(name: "Hazardous item", isHazardous: true),
        ]);
        PostageManager postageManager = new PostageManager(); ❶

        PostageLabel label = postageManager.getPostageLabel(order);

        assertThat(label.isHazardous()).isTrue();
    }
    ...

    private static Order createOrderWithItems(List<Item> items) { ❷
        return new Order(
            customer: new Customer(
                address: new Address("Test address"),
            ),
```

```

        items: items);
    }
}

```

- ❶ Контрольные примеры выполняют свою собственную настройку для важных вещей.
- ❷ Вспомогательная функция для создания экземпляра `Order` с конкретными позициями.

Когда некоторая конфигурация напрямую влияет на результат контрольного примера, обычно лучше сохранить ее автономной в рамках этого контрольного примера. Это защищает от будущих изменений, непреднамеренно повреждающих тесты, а также проясняет причину и следствие в каждом контрольном примере (потому что все, что влияет на контрольный пример значимым образом, присутствует в нем самом). Однако не каждая конфигурация соответствует этому описанию, и в следующем подразделе обсуждается, когда совместная конфигурация может быть хорошей идеей.

11.4.5. Когда использование совместной конфигурации возможно

Предыдущие подразделы демонстрируют, почему полезно проявлять осторожность при использовании совместной тестовой конфигурации, но это не значит, что пользоваться ею не нужно вообще. Некоторые элементы конфигурации необходимы, но напрямую не влияют на результаты контрольных примеров. В подобных сценариях использование совместной конфигурации может быть отличным способом сделать тесты узконаправленными и понятными, избегая ненужного повторения кода и шаблонности.

Чтобы продемонстрировать это, представьте, что для создания экземпляра класса `Order` также требуется предоставить некоторые метаданные о заказе. Класс `PostageManager` игнорирует эти метаданные, поэтому он не имеет никакого отношения к результатам контрольных примеров в `OrderPostageManagerCase`. Но контрольным примерам все равно необходимо это конфигурировать, потому что без этого не может быть создан экземпляр класса `Order`. В такой ситуации имеет смысл один раз определить метаданные заказа как совместную конфигурацию. Это демонстрирует листинг 11.23. Экземпляр `OrderMetadata` помещается в совместную константу, называемую `ORDER_METADATA`. Затем контрольные примеры могут использовать эту константу, вместо того чтобы многократно создавать эти необходимые, но в остальном не относящиеся к делу данные.

Листинг 11.23. Подходящее использование совместной конфигурации

```

class OrderPostageManagerTest {
    private const OrderMetadata ORDER_METADATA = ❶
        new OrderMetadata(
            timestamp: Instant.ofEpochSecond(0),
            serverIp: new IpAddress(0, 0, 0, 0)); ❶
}

```

```

void testGetPostageLabel_threeItems_largePackage() { ... }
void testGetPostageLabel_hazardousItem_isHazardous() { ... }
...
void testGetPostageLabel_containsCustomerAddress() {
    Address address = new Address("Test customer address");
    Order order = new Order(
        metadata: ORDER_METADATA, ❷
        customer: new Customer(
            address: address,
        ), items: []);

    PostageLabel label = postageManager.getPostageLabel(order);

    assertThat(label.getAddress()).isEqualTo(address);
}
...

private static Order createOrderWithItems(List<Item> items) {
    return new Order(
        metadata: ORDER_METADATA, ❷
        customer: new Customer(
            address: new Address("Test address"),
        ),
        items: items);
}
}

```

❶ Экземпляр `OrderMetadata` создается в совместной константе.

❷ Совместный `OrderMetadata` используется в контрольных примерах.

В идеале функции должны принимать только то, что им нужно. В *главе 9* обсуждалось, что в идеале параметры функций должны быть узконаправленными, а это означает, что функции принимают только то, что им нужно. Если тесты для части кода требуют конфигурации большого количества значений, которые необходимы, но в остальном не имеют отношения к поведению кода, то это может быть признаком того, что параметры функции (или конструктора) недостаточно узконаправленны. Например, мы могли бы утверждать, что функция `PostageManager.getPostageLabel()` должна принимать только экземпляр адреса и список товарных позиций вместо законченного экземпляра класса `Order`. Если бы это было так, то тестам не нужно было бы создавать нерелевантные вещи, такие, как экземпляр `OrderMetadata`.

Совместная настройка тестирования может быть палкой о двух концах. Она может быть очень полезна для предотвращения повторения кода или многократного выполнения дорогостоящей настройки, но также может привести к неэффективным тестам, которые трудно обосновать. Стоит тщательно подумать, чтобы убедиться, что совместная настройка используется надлежащим образом.

11.5. Используйте подходящие обнаружители совпадений в утверждениях

Обнаружитель совпадений в утверждениях (англ. *assertion matcher*) — это такой элемент контрольного примера, который обычно в конечном итоге и определяет успех теста. Следующий фрагмент содержит два обнаружителя совпадений — `(isEqualTo())` и `(contains())`:

```
assertThat(некоторое значение).isEqualTo("expected value"); assertThat(некоторый список).contains("expected value");
```

Если контрольный пример терпит неудачу, то обнаружитель совпадений в утверждениях также выдает сообщение об ошибке, объясняющее причину. Разные средства обнаружения совпадений в утверждениях выдают разные сообщения об ошибках (в зависимости от того, что они утверждают). В *главе 10* мы определили *хорошо объясненные сбои* как одну из ключевых особенностей хорошего модульного теста, поэтому важно убедиться, что мы выбрали наиболее подходящее средство обнаружения совпадений в утверждениях.

11.5.1. Неподходящие средства обнаружения совпадений могут привести к плохо объясненным сбоям

Чтобы продемонстрировать, как использование неподходящего обнаружителя совпадений может привести к плохо объясненным сбоям в тестировании, мы сосредоточимся на тестировании кода в листинге 11.24. `TextWidget` — это компонент, используемый в пользовательском интерфейсе веб-приложения для отображения текста. Чтобы управлять стилем компонента, к нему можно добавлять различные имена классов. Некоторые из этих имен классов закодированы жестко, а другие, пользовательские, могут быть предоставлены при помощи конструктора. Функция `getClassNamees()` возвращает объединенный список всех имен классов. Важной деталью, которую следует отметить, является то, что документация для функции `getClassNamees()` указывает, что порядок возвращаемых имен классов не гарантируется.

Листинг 11.24. Код для `TextWidget`

```
class TextWidget {  
    private const ImmutableList<String> STANDARD_CLASS_NAMES = ❶  
        ["text-widget", "selectable"];  
    private final ImmutableList<String> customClassNames;  
  
    TextWidget(List<String> customClassNames) { ❷  
        this.customClassNames = ImmutableList.copyOf(customClassNames);  
    }  
  
    /**  
     * The class names for the component. The order of the class
```

```

* names within the returned list is not guaranteed.
*/
ImmutableList<String> getClassNames() { ❸
    return STANDARD_CLASS_NAMES.concat(customClassNames);
}
...
}

```

- ❶ Жестко закодированные имена классов.
- ❷ Пользовательские имена классов, предоставляемые с помощью конструктора.
- ❸ Получает список всех имен классов (жестко закодированных и пользовательских).

Как мы видели ранее, в идеале мы должны стремиться тестировать одно поведение за раз. Одна из моделей поведения, которую нам нужно проверить, заключается в том, что список, возвращаемый `getClassNames()`, содержит `customClassNames`. Один из подходов, который можно использовать для проверки этого, заключается в сравнении возвращаемого списка с ожидаемым списком значений. Это показано в листинге 11.25. Но у этого подхода есть несколько проблем, а именно:

- ◆ Контрольный пример тестирует больше, чем предполагалось. Название контрольного примера предполагает, что это всего лишь проверка того, содержит ли результат имена пользовательских классов. Но на самом деле это также проверка и того, содержит ли результат стандартные имена классов.
- ◆ Если порядок, в котором возвращаются имена классов, когда-либо изменится, то этот тест завершится неудачей. В документации для функции `getClassNames()` явно говорится, что такой порядок не гарантируется, поэтому мы не должны создавать тест, который завершается неудачей при изменении порядка. Это может привести к ложным тревогам или капризным тестам.

Листинг 11.25. Тестовое утверждение с чрезмерными ограничениями

```

void testGetClassNames_containsCustomClassNames() {
    TextWidget textWidget = new TextWidget(
        ["custom_class_1", "custom_class_2"]);

    assertThat(textWidget.getClassNames()).isEqualTo([
        "text-widget",
        "selectable",
        "custom_class_1",
        "custom_class_2",
    ]);
}

```

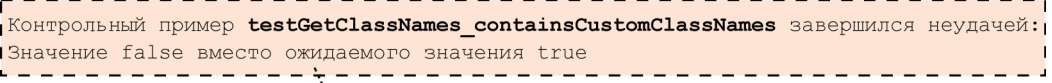
Давайте рассмотрим еще одну идею, которую можно попробовать. Вместо сравнения возвращаемого результата с ожидаемым списком мы могли бы индивидуально проверить, содержит ли возвращаемый список два значения, которые нас интересуют: `custom_class_1` и `custom_class_2`. В листинге 11.26 показан один из способов

достижения этой цели: утверждение, что `result.contains(...)` возвращает значение `true`. Это решило две проблемы, которые мы только что видели: тест теперь проверяет только то, для чего он предназначен, и изменение порядка не приведет к сбою теста. Но мы столкнулись с другой проблемой: сбои в тестировании не будут хорошо объяснены (рис. 11.6).

Листинг 11.26. Тестовое утверждение с плохой объяснимостью

```
void testGetClassNames_containsCustomClassNames() {  
    TextWidget textWidget = new TextWidget(  
        ["custom_class_1", "custom_class_2"]);  
  
    ImmutableList<String> result = textWidget.getClassNames();  
  
    assertThat(result.contains("custom_class_1")).isTrue();  
    assertThat(result.contains("custom_class_2")).isTrue();  
}
```

На рис. 11.6 показано, как будет выглядеть сообщение об ошибке, если контрольный пример завершится неудачей из-за отсутствия одного из пользовательских классов. Из этого сообщения об ошибке не очевидно, чем фактический результат отличается от ожидаемого.



Контрольный пример `testGetClassNames_containsCustomClassNames` завершился неудачей:
Значение `false` вместо ожидаемого значения `true`

Сообщение об ошибке плохо объясняет причину

Рис. 11.6. Неправильное сопоставление утверждений может привести к плохо объясненному сбою теста

Очень важно обеспечить, чтобы тест не завершился успехом, когда что-то сломано, но, как мы видели в *главе 10*, это не единственное, что нужно продумать. Нужно также убедиться, что тест завершается неудачей только в том случае, если что-то действительно сломано, и что сбои в тестировании хорошо объяснены. Для достижения всех этих целей нам необходимо выбрать подходящий обнаружитель совпадений в утверждениях.

11.5.2. Решение: используйте подходящий обнаружитель совпадений

Большинство современных инструментов тестовых утверждений содержат множество различных обнаружителей совпадений, которые можно использовать в тестах. Одним из предлагаемых обнаружителей совпадений может быть такой, который позволяет нам утверждать, что список содержит, по крайней мере, определенный

набор элементов в неопределенном порядке. Примерами таких обнаружителей совпадений являются следующие.

- ◆ В Java — обнаружитель совпадений `containsAtLeast()` из библиотеки `Truth` ([https:// truth.dev/](https://truth.dev/)).
- ◆ В JavaScript — обнаружитель совпадений `jasmine.arrayContaining()` из фреймворка `Jasmine` (<https://jasmine.github.io/>).

В листинге 11.27 показано, как будет выглядеть наш тестовый пример, если мы воспользуемся обнаружителем совпадений `containsAtLeast()`. Контрольный пример завершится неудачей, если функция `getClassNamees()` не вернет ни одного из пользовательских имен классов. Но это не приведет к сбою из-за изменений в других вариантах поведения, таких, как обновление жестко закодированных имен классов или изменение порядка.

Листинг 11.27. Подходящий обнаружитель совпадений в утверждениях

```
testGetClassNames_containsCustomClassNames() {
  TextWidget textWidget = new TextWidget(
    ["custom_class_1", "custom_class_2"]);

  assertThat(textWidget.getClassNamees())
    .containsAtLeast ("custom_class_1", "custom_class_2");
}
```

Если контрольный пример завершится неудачей, то сообщение об ошибке будет хорошо объяснено, как показано на рис. 11.7.

Контрольный пример `testGetClassNames_containsCustomClassNames` завершился неудачей:
 Неверно, что
 [text-widget, selectable, custom_class_2]
 содержит по крайней мере
 [custom_class_1, custom_class_2]

 отсутствующий элемент списка: custom_class_1

Сообщение об ошибке дает четкое объяснение того,
 как отличается фактическое поведение от ожидаемого

Рис. 11.7. Подходящий обнаружитель совпадений в утверждениях выдаст хорошо объясненный сбой теста

В дополнение к получению более объясненных сбоев использование подходящего обнаружителя совпадений может немного облегчить понимание тестового кода. В следующем фрагменте первая строка кода больше похожа на реальное предложение, чем вторая строка:

```
assertThat(someList).contains("expected value");
assertThat(someList.contains("expected value")).isTrue();
```


Помимо гарантии того что тест завершится неудачей при сбое в коде, важно подумать о том, как именно тест завершится неудачей. Использование подходящего средства обнаружения совпадений в утверждениях может помочь хорошо объяснить сбой в тесте и избавить других программистов от лишней головной боли.

11.6. Используйте внедрение зависимостей для улучшения тестируемости

В главе 2, 8 и 9 приведены примеры того, как использование внедрения зависимостей смогло улучшить код. Однако есть еще одна очень веская причина использовать внедрение зависимостей: это может сделать код значительно более тестируемым.

В предыдущей главе мы увидели, что тестам порой приходится взаимодействовать с некоторыми зависимостями тестируемого кода. Это происходит всякий раз, когда тесту необходимо установить некоторые начальные значения внутри зависимости или проверить, случился ли в одной из них побочный эффект. В разд. 10.4 объясняется, что иногда необходимо использовать имитированную реализацию в качестве замены реальной зависимости. Поэтому очевидно, что существуют сценарии, в которых тесту потребуется предоставить конкретный экземпляр зависимости для тестируемого кода. Если тестовый код не сможет это сделать, то у вас не получится проверить некоторые варианты поведения.

11.6.1. Жестко запрограммированные зависимости могут сделать невозможным тестирование кода

Чтобы продемонстрировать это, в листинге 11.28 показан класс для отправки клиентам напоминаний о счетах. Класс `InvoiceReminder` не использует внедрение зависимостей и вместо этого создает свои собственные зависимости в своем конструкторе. Зависимость `AddressBook` используется этим классом для поиска адресов электронной почты клиентов, а зависимость `EmailSender` используется для отправки электронных писем.

Листинг 11.28. Класс без внедрения зависимостей

```
class InvoiceReminder {  
    private final AddressBook addressBook;  
    private final EmailSender emailSender;  
  
    InvoiceReminder() {  
        this.addressBook = DataStore.getAddressBook(); ❶  
        this.emailSender = new EmailSenderImpl();  
    }  
}
```

```
@CheckReturnValue
Boolean sendReminder(Invoice invoice) {
    EmailAddress? address =
        addressBook.lookupEmailAddress(invoice.getCustomerId()); ❷
    if (address == null) {
        return false;
    }
    return emailSender.send( ❸
        address,
        InvoiceReminderTemplate.generate(invoice));
}
```

❶ Зависимости создаются в конструкторе.

❷ Поиск адреса электронной почты происходит при помощи `addressBook`.

❸ Электронное письмо отправляется при помощи `emailSender`.

Есть несколько вариантов поведения (например, те, что представлены ниже), и в идеале мы должны протестировать каждый из них. Итак, нужно протестировать то, что функция `sendReminder()`:

- отправляет электронное письмо клиенту, когда его адрес есть в адресной книге;
- возвращает значение `true` при отправке напоминания по электронной почте;
- не отправляет электронное письмо, если адрес электронной почты клиента не может быть найден;
- возвращает значение `false`, когда напоминание по электронной почте не отправляется.

К сожалению, довольно сложно (и порой даже невозможно) протестировать все эти модели поведения с классом в его текущей форме по двум причинам.

Во-первых, класс создает свою собственную `AddressBook`, вызывая `datastore.getAddressBook()`. Когда код выполняется в реальной жизни, создается `AddressBook`, который подсоединяется к базе данных клиентов для поиска контактной информации. Но такой вариант не подходит для использования в тестах, потому что использование реальных данных клиентов может привести к ошибкам, поскольку данные меняются с течением времени. Еще одна более фундаментальная проблема заключается в том, что среда, в которой выполняется тест, скорее всего, не имеет разрешения на доступ к реальной базе данных, поэтому во время тестирования возвращенный `AddressBook` может вообще не работать.

Во-вторых, класс создает свой собственный `EmailSenderImpl`. Это означает, что у теста будут реальные последствия в виде отправки реальных электронных писем. Это не является побочным эффектом, к которому должен приводить тест; это пример, демонстрирующий, что нам необходимо защитить внешний мир от теста (как обсуждалось в *главе 10*).

Как правило, простым решением обеих этих проблем будет использование имитированной реализации для `AddressBook` и `EmailSender`. Но в этом сценарии мы не можем так поступить, потому что у нас нет способа создать экземпляр класса `InvoiceReminder` с имитированными реализациями вместо реальных зависимостей. Класс `InvoiceReminder` обладает плохой тестируемостью, и вероятным следствием этого является то, что не все модели его поведения будут проверены должным образом, что увеличивает вероятность ошибок в коде.

11.6.2. Решение: использовать внедрение зависимостей

Мы можем сделать класс `InvoiceReminder` намного более тестируемым и решить эту проблему с помощью внедрения зависимостей. В листинге 11.29 показано, как будет выглядеть класс, если мы изменим его так, чтобы его зависимости можно было внедрять с помощью конструктора. Класс также включает статическую заводскую функцию, поэтому реальным пользователям класса по-прежнему легко создавать ее, не беспокоясь о зависимостях.

Листинг 11.29. Класс с внедрением зависимости

```
class InvoiceReminder {
    private final AddressBook addressBook;
    private final EmailSender emailSender;

    InvoiceReminder( ❶
        AddressBook addressBook,
        EmailSender emailSender) {
        this.addressBook = addressBook;
        this.emailSender = emailSender;
    }

    static InvoiceReminder create() { ❷
        return new InvoiceReminder(
            DataStore.getAddressBook(),
            new EmailSenderImpl());
    }

    @CheckReturnValue
    Boolean sendReminder(Invoice invoice) {
        EmailAddress? address =
            addressBook.lookupEmailAddress(invoice.getCustomerId());
        if (address == null) {
            return false;
        }
        return emailSender.send(
            address,
            InvoiceReminderTemplate.generate(invoice));
    }
}
```

- ❶ Зависимости, внедренные при помощи конструктора.
- ❷ Статическая заводская функция.

Теперь тестам очень легко создать класс `InvoiceReminder` с использованием имитированных реализаций (в данном случае это `FakeAddressBook` и `FakeEmailSender`):

```
...
FakeAddressBook addressBook = new FakeAddressBook();
fakeAddressBook.addEntry(
    customerId: 123456,
    emailAddress: "test@example.com");
FakeEmailSender emailSender = new FakeEmailSender();
InvoiceReminder invoiceReminder =
    new InvoiceReminder(addressBook, emailSender);
...
```

Как упоминалось в *главе 1*, тестируемость в значительной степени связана с модульностью. Когда различные фрагменты кода слабо связаны и могут быть реконфигурированы, их, как правило, намного проще тестировать. Внедрение зависимостей — это эффективный метод увеличения модульности, а также эффективный метод повышения тестируемости кода.

11.7. Несколько заключительных слов о тестировании

Тестирование программного обеспечения — это обширная тема, и темы, которые мы рассмотрели в последних двух главах, являются лишь верхушкой айсберга. В этих главах рассматривалось модульное тестирование, которое представляет собой уровень тестирования, с которым программисты обычно чаще всего сталкиваются в повседневной работе. Как обсуждалось в *главе 1*, два других уровня тестирования, с которыми вы, скорее всего, столкнетесь (и будете использовать), это следующие:

- ◆ *интеграционные тесты* — система обычно состоит из нескольких компонентов, модулей или подсистем. Процесс соединения этих компонентов и подсистем во едино известен как *интеграция*. Интеграционные тесты пытаются гарантировать, что эти интеграции работают и продолжают работать;
- ◆ *сквозные тесты* (англ. *end-to-end*, *E2E tests*) — эти тесты проверяют типичные маршруты (или потоки работ) через всю программную систему от начала до конца. Если бы рассматриваемое программное обеспечение было интернет-магазином, то примером сквозного теста мог бы быть тест, который автоматически управляет веб-браузером, чтобы гарантировать, что пользователь может пройти весь процесс завершения покупки.

В дополнение к различным уровням тестирования существует множество различных типов тестирования. Эти определения иногда могут пересекаться, и програм-

мисты не всегда сходятся во мнении о том, что именно они означают. Далеко не исчерпывающий список концепций, о которых полезно знать, выглядит следующим образом.

- ◆ *Регрессионное тестирование*. Тесты, которые регулярно выполняются для того, чтобы убедиться, что поведение или функциональность программного обеспечения не изменились нежелательным образом. Модульные тесты обычно являются важной частью регрессионного тестирования, но они также могут включать другие уровни тестирования, например, интеграционные тесты.
- ◆ «*Золотое тестирование*» (англ. *Golden testing*), иногда называемое *характеризационным тестированием*. Такие тесты обычно основаны на сохраненном моментальном снимке вывода кода для заданного набора входных данных. Если наблюдаемый вывод кода когда-либо изменится, то тесты завершатся неудачей. Это может быть полезно, для того чтобы убедиться, что ничего не изменилось, однако когда тесты действительно терпят неудачу, бывает трудно определить причину сбоя. Эти тесты также могут быть невероятно хрупкими и ненадежными в некоторых сценариях.
- ◆ *Фаззинг* (англ. *fuzz testing*). Этот тип тестирования обсуждался в *главе 3*. При тестировании методом фаззинга вызывается код с большим количеством случайных или «интересных» входных данных и проверяется, чтобы ни одна из этих ситуаций не завершалась сбоем кода.

Существует широкий и разнообразный набор методов, которые программисты могут использовать для тестирования программного обеспечения. Написание и поддержание программного обеспечения на высоком уровне зачастую требует использования сочетания этих методов. Хотя модульное тестирование, скорее всего, является тем типом тестирования, с которым вы будете сталкиваться чаще всего, маловероятно, что только лишь этот метод удовлетворит все ваши потребности в тестировании, поэтому стоит прочитать о различных типах и уровнях тестирования и быть в курсе всех новых инструментов и методов.

Итоги

- ◆ Концентрация на тестировании каждой функции может легко привести к недостаточному тестированию. Как правило, более эффективным способом будет определить все важные модели поведения и написать контрольный пример для каждой из них.
- ◆ Проверяйте модели поведения кода, которые действительно важны. Тестирование приватных функций почти всегда свидетельствует о том, что мы не тестируем то, что на самом деле имеет значение.
- ◆ Тестирование одной вещи за раз приводит к тестам, которые легче понять, а также к лучше объясненным сбоям теста.

- ◆ Совместная настройка тестирования может быть палкой о двух концах. Это позволяет избежать повторения кода или дорогостоящей с вычислительной точки зрения настройки, но также может привести к неэффективным или капризным тестам при неправильном использовании.
- ◆ Использование внедрения зависимостей может значительно повысить тестируемость кода.
- ◆ Модульное тестирование — это уровень тестирования, с которым программисты чаще всего имеют дело, но это вовсе не единственный уровень. Качественное написание и поддержание программного обеспечения требует использования нескольких методов тестирования.

Вы добрались до конца моей книги и даже прочитали главы о тестировании! Я надеюсь, что вам понравилось путешествие по этой книге и в пути вы узнали кое-что полезное. Теперь, когда мы покончили с одиннадцатью главами вступления, можно перейти к самой важной части книги: читабельной версии рецепта шоколадных пирожных в *приложении А*.

Часть 4

Приложения

Приложение А. Рецепт шоколадных пирожных

Приложение В. Null-безопасность и необязательные значения

Приложение С. Дополнительные примеры кода

Приложение А

Рецепт шоколадных пирожных

Вам понадобится:

- 100 г сливочного масла
- 185 г темного шоколада с содержанием какао 70%
- 2 яйца
- ½ чайной ложки ванильной эссенции
- 185 г сахарной пудры (или сахара высшего сорта)
- 50 г муки
- 35 г какао-порошка
- ½ чайной ложки соли
- 70 г шоколадной стружки

Способ приготовления

1. Разогрейте духовку до 160 °C.
2. Смажьте маслом и застелите пергаментом небольшую форму для выпечки (15×15см).
3. Сливочное масло и темный шоколад растопите в миске над кастрюлей с горячей водой. Когда смесь полностью растает, снимите ее с огня и дайте остыть.
4. Отдельно смешайте яйца, сахар и ванильную эссенцию.
5. Добавьте растопленное сливочное масло и темный шоколад к смеси яиц и сахара и перемешайте.
6. В отдельной миске смешайте муку, какао-порошок и соль, а затем просейте получившуюся смесь, добавляя ее к яйцам, сахару, маслу и шоколаду. Смешайте до получения однородной смеси.
7. Добавьте шоколадную стружку и смешайте до получения однородной смеси.
8. Выложите смесь в форму для выпечки и выпекайте в течение 20 минут.

Дайте остыть в течение нескольких часов.

Приложение В

Null-безопасность и необязательные значения

В.1. Использование null-безопасности

Если язык, который мы используем, поддерживает null-безопасность (и мы включили ее по требованию), будет существовать механизм для аннотирования типов, указывающий на то, что они могут иметь значение `null`. Часто это включает символ `?`, указывающий, что значение `null` допускается. Код может выглядеть примерно так:

```
Element? getFifthElement(List<Element> elements) { ❶
    if (elements.size() < 5) {
        return null;
    }
    return elements[4];
}
```

❶ Символ `?` в `Element?` указывает, что тип возвращаемого значения может иметь значение `null`.

Если программист, использующий этот код, забудет обработать сценарий, в котором `getFifthElement()` возвращает `null`, такой код не будет компилироваться, как показано в следующем листинге:

```
void displayElement(Element element) { ... } ❶

void displayFifthElement(List<Element> elements) {
    Element? fifthElement = getFifthElement(elements); ❷
    displayElement(fifthElement); ❸
}
```

❶ Параметр этой функции не допускает значения `null` (поскольку тип — `Element`, а не `Element?`).

❷ Переменная `fifthElement` допускает значение `null`, так как ее тип — `Element?`.

❸ В этой строке может произойти ошибка компилятора, поскольку функция ожидала аргумент, не допускающий `null`, но здесь она вызывается со значением `null`.

Чтобы выполнить компиляцию кода, программист должен проверить, что значение, возвращаемое `getFifthElement()`, не равно `null`, прежде чем использовать его для вызова функции, параметр которой не допускает `null`. Компилятор может определить,

какие пути кода доступны только в том случае, если значение не равно `null`, и, таким образом, определить, безопасно ли использование значения.

```
void displayFifthElement(List<Element> elements) {  
    Element? fifthElement = getFifthElement(elements);  
    if (fifthElement == null) {  
        displayMessage("Fifth element doesn't exist");  
        return; ❶  
    }  
    displayElement(fifthElement); ❷  
}
```

❶ Этот оператор `if` означает, что функция вернет значение раньше, если `fifthElement` равен `null`.

❷ Компилятор может сделать вывод, что эта строка доступна только в том случае, если `fifthElement` не равен `null`.

ПРЕДУПРЕЖДЕНИЯ КОМПИЛЯТОРА И ОШИБКИ

В C# небезопасное использование значения, допускающего `null`, приводит только к предупреждению компилятора, а не к ошибке компилятора. Если вы используете C# и включили `null`-безопасность, возможно, было бы разумно сконфигурировать ваш проект так, чтобы вместо предупреждений выдавались ошибки, чтобы они точно не остались незамеченными.

Как мы видим, с `null`-безопасностью мы можем использовать значения `null`, и компилятор будет отслеживать, когда значение логически может или не может быть `null`, а также следить за тем, чтобы значение `null` потенциально не использовалось небезопасно. Это позволяет нам извлечь выгоду из значений `null`, не подвергаясь опасности исключений нулевого указателя (и тому подобного).

Проверка на наличие значений `null`

Языки с `null`-безопасностью обычно предоставляют лаконичный синтаксис для проверки того, является ли значение нулевым, а также предоставляют доступ к функции-члену или ее свойству только в том случае, если оно не равно `null`. Это может устранить множество шаблонов, но соглашение о псевдокоде в этой книге будет придерживаться более подробной формы проверки на наличие значений `null`, чтобы оно было аналогично более широкому набору языков, которые не предлагают этот синтаксис.

Тем не менее, чтобы показать, что под этим подразумевается, представьте, что есть функция для поиска адреса, которая возвращает `null`, если адрес не найден:

```
Address? lookupAddress() {  
    ...  
    return null;  
    ...  
}
```

В коде, вызывающем эту функцию, может потребоваться проверить, является ли возвращаемое значение функции `lookupAddress()` равным `null`, а также потребоваться вызвать функцию `getCity()` на адрес, только если значение ненулевое. Примеры кода в этой книге делают это с помощью подробного оператора `if` для проверки на наличие значения `null`:

```
City? getCity() {
    Address? address = lookupAddress();
    if (address == null) {
        return null;
    }
    return address.getCity();
}
```

Но имейте в виду, что большинство языков, поддерживающих `null`-безопасность, также предоставляют более компактные синтаксисы для такого рода вещей. Например, мы, вероятно, смогли бы написать код, который мы только что видели, более лаконично, используя *условный оператор* `null`:

```
City? getCity() {
    return lookupAddress()?.getCity();
}
```

Как мы видим, использование `null`-безопасности имеет множество преимуществ. Это не только делает наш код менее подверженным ошибкам, но также может позволить нам использовать другие языковые функции, чтобы сделать код намного более лаконичным и при этом читабельным.

В.2. Использование `optional`

Если язык, который мы используем, не обеспечивает `null`-безопасность, или если по какой-то причине невозможно ее использование, то возврат `null` из функции может быть сюрпризом для вызывающего пользователя. Чтобы избежать этого, мы можем использовать такой тип, как `Optional`, чтобы заставить вызывающих пользователей осознать, что возвращаемое значение может отсутствовать. Код из предыдущего раздела выглядит следующим образом с использованием типа `Optional`:

```
Optional<Element> getFifthElement(List<Element> elements) {
    if (elements.size() < 5) {
        return Optional.empty();
    }
    return Optional.of(elements[4]);
}
```

Программист, использующий этот код, мог бы затем написать что-то вроде следующего:

```
void displayFifthElement(List<Element> elements) {
    Optional<Element> fifthElement = getFifthElement(elements);
    if (fifthElement.isPresent()) { ❶
```

```
displayElement(fifthElement.get()) ❷  
return;  
}  
displayMessage("Fifth element doesn't exist");  
}
```

❶ Проверяет наличие необязательного значения перед его использованием.

❷ Доступ к значению внутри необязательного осуществляется путем вызова функции `get()` для него.

Это, по общему признанию, немного неуклюже, но типы `Optional` обычно предоставляют различные функции-члены, которые могут сделать их использование намного более лаконичным в некоторых сценариях. Одним из примеров этого является функция `ifPresentOrElse()` (как можно увидеть в Java 9). Если мы перепишем функцию `displayFifthElement()` так, чтобы она могла использовать `Optional.ifPresentOrElse()`, она будет выглядеть следующим образом:

```
void displayFifthElement(List<Element> elements) {  
    getFifthElement(elements).ifPresentOrElse(  
        displayElment, ❶  
        () -> displayMessage("Fifth element doesn't exist")); ❷  
}
```

❶ Функция `displayElement()` вызывается вместе с элементом, если он присутствует.

❷ Функция `DisplayMessage()` вызывается, если элемент отсутствует.

В зависимости от нашего сценария использование типа `Optional` может быть слишком подробным и неуклюжим, но проблема необработанных значений `null` может быстро стать настолько распространенной, что стоимость дополнительной подробности и неуклюжести обычно более чем окупается с точки зрения повышения надежности кода и уменьшения ошибок.

OPTIONAL В C++

На момент написания книги версия `Optional` стандартной библиотеки C++ не поддерживает отсылки, что означает, что ее может быть сложно использовать для возврата объектов, подобных классам. Примечательной альтернативой является версия `optional` библиотеки `Boost`, которая поддерживает отсылки.

У каждого подхода есть свои плюсы и минусы (в которые мы здесь не будем вдаваться), но если вы планируете использовать `optional` в своем коде на C++, то стоит прочитать об этом более подробно. Стандартная библиотечная версия `optional`: <http://mng.bz/n2pe>

Расширенная библиотечная версия `optional`: +-

Приложение С

Дополнительные примеры кода

Шаблон проектирования

Глава 7 содержала упрощенную реализацию шаблона проектирования. В действительности программисты часто используют множество методов и языковых функций при реализации шаблона проектирования. Листинг С.1 ❶ демонстрирует более полную реализацию шаблона проектирования в Java. В этой реализации следует отметить следующие моменты.

- ◆ Конструктор класса `TextOptions` является приватным, чтобы заставить других программистов использовать шаблон проектирования.
 - ◆ Конструктор класса `TextOptions` принимает экземпляр `Builder` в качестве параметра. Это немного облегчает чтение и поддержку кода, поскольку позволяет избежать очень длинных списков параметров и аргументов.
 - ◆ Класс `TextOptions` предоставляет метод `toBuilder()`, который можно использовать для создания предварительно заполненного экземпляра класса `Builder` из экземпляра класса `TextOptions`.
 - ◆ Класс `Builder` является внутренним классом для класса `TextOptions`. Это служит двум целям:
 - интервал между именами становится немного приятнее, потому что теперь к `Builder` можно обращаться с помощью `TextOptions.Builder`;
 - в Java это позволяет классам `TextOptions` и `Builder` совместно иметь доступ к приватным переменным-членам и методам.
- ❶ Вдохновлено формами шаблонов проектирования, описанными в третьем издании книги *Effective Java* Джошуа Блоха (Эддисон-Уэсли, 2017), а также различными кодовыми базами, такими, как библиотеки Google Guava.

Листинг С.1. Реализация шаблона проектирования

```
public final class TextOptions {  
    private final Font font;  
    private final OptionalDouble fontSize;  
    private final Optional<Color> color;  
    private TextOptions(Builder builder) { ❶  
        font = builder.font;  
        fontSize = builder.fontSize;
```

```
        color = builder.color;
    }
    public Font getFont() {
        return font;
    }
    public OptionalDouble getFontSize() {
        return fontSize;
    }
    public Optional<Color> getColor() {
        return color;
    }
    public Builder toBuilder() { ❷
        return new Builder(this);
    }
    public static final class Builder { ❸
        private Font font;
        private OptionalDouble fontSize = OptionalDouble.empty();
        private Optional<Color> color = Optional.empty();
        public Builder(Font font) {
            this.font = font;
        }
        private Builder(TextOptions options) {❹
            font = options.font;
            fontSize = options.fontSize;
            color = options.color;
        }
        public Builder setFont(Font font) {
            this.font = font;
            return this;
        }
        public Builder setFontSize(double fontSize) {
            this.fontSize = OptionalDouble.of(fontSize);
            return this;
        }
        public Builder clearFontSize() {
            fontSize = OptionalDouble.empty();
            return this;
        }
        public Builder setColor(Color color) {
            this.color = Optional.of(color);
            return this;
        }
        public Builder clearColor() {
            color = Optional.empty();
            return this;
        }
    }
```

```
public TextOptions build() {  
    return new TextOptions(this);  
}  
}  
}
```

- ❶ Конструктор приватный и принимает `Builder` в качестве параметра.
- ❷ Функция `toBuilder()` позволяет создавать предварительно заполненный проект.
- ❸ Класс `Builder` — это внутренний класс `TextOptions`.
- ❹ Приватный конструктор `Builder` для копирования из некоторых `TextOptions`.

Ниже приведены некоторые примеры того, как можно использовать этот код.

```
TextOptions options1 = new TextOptions.Builder(Font.ARIAL)  
    .setFontSize(12.0)  
    .build();
```

```
TextOptions options2 = options1.toBuilder()  
    .setColor(Color.BLUE)  
    .clearFontSize()  
    .build();
```

```
TextOptions options3 = options2.toBuilder()  
    .setFont(Font.VERDANA)  
    .setColor(Color.RED)  
    .build();
```

Предметный указатель

A

Acceptance test–driven development, ATDD, 368
API, application programming interface, 340, 343, 345, 380
Assertion matcher, 405

B

Behavior-driven development, BDD, 367
Boilerplate code, 269

C

C#, 238, 251
C++, 217, 239
CamelCase, 155

D, E

Dart, 285
End-to-end, E2E tests, 412

F

Fakes, 347
Fuzz testing, 413

G

Golden testing, 413

H

HTML
◊ элемент, 189

I

IDE, integrated development environments, 165

J

Java, 178, 238, 251
Java Project Lombok, 281
JavaScript, 238, 251, 307

K, L

Kotlin, 281
Law of Demeter, 287

M

Mock, 353, 355
Mocks, 347

N, P, S

Null-безопасность, 183
PascalCase, 155
Stubs, 347

T

Test double, 347
Test-driven development, TDD, 367
Traits, 284

U

URL-адрес, 52

А

- Адаптивность кода, 266
- Альтернатива:
 - ◊ объектам данных, 290
- Анализ типов данных, 83
- Анонимные функции, 170, 171, 175
- Аргументы, 136
 - ◊ конструктора, 267

Б

- Безошибочные части соглашения, 89
- Безразличие к деталям реализации, 370
- Бит четности, 152
- Большие классы, 69
- Быстрая ошибка, 101, 104

В

- Важные параметры поведения, 340
- Ввод (предварительное условие), 85
- Вероятность:
 - ◊ ошибок, 16
 - ◊ сбоя кода, 97
- Взаимодействие с базой данных, 76
- Вложенность, 159
- Внедрение зависимостей, 65, 267, 271, 411
- Возврат:
 - ◊ значения, 89, 109, 119
 - ◊ магических чисел, 181
 - ◊ ответа, 76
 - ◊ пустой строки, 188
- Возвращаемое значение, 341
- Возвращено (постусловие), 85
- Вспомогательная функция, 169
- Вторые источники истины, 254
- Входные параметры, 203
- Выбраковка релиза, 30
- Вызовы функций, 161
- Выполняется на устройстве пользователя., 100
- Высококачественный код, 27

Г

- Глобальная переменная, 307
- Глобальное состояние, 307, 309, 311, 315, 397
- Глобальность, 309
- Громкая ошибка, 101, 106

Д

- Данные:
 - ◊ неинкапсулированные, 288
- Дано, когда, тогда, 334
- Действие, 333, 334
- Делегирование, 280
- Детали реализации, 56, 342
- Дискуссионный форум liveBook, 17
- Дихотомия, 107
- Документация, 82, 84, 150, 221, 249
- Дополнение, 278
- Допущение «невозможно сломать», 304

З

- Зависимость:
 - ◊ закодированная, 326
- Заглушки, 347, 355, 360
- Закон Деметры, 287
- Запрограммированные зависимости, 265, 409
- Запрос хранилища данных, 76
- Запуск, 340
- Защита:
 - ◊ внешнего мира, 347, 349
 - ◊ теста, 348, 351
- Защитное копирование, 236, 237
- Защищает от поломки, 335
- Значения по умолчанию, 320
- Золотое тестирование, 413

И

- Идентификатор, 191
- Идентификация, 143
- Изменение параметров, 205
- Изменчивость, 93
- Изменяемость, 222
- Изменяемые классы, 223
- Изменяемый код, 222
- Именованные:
 - ◊ аргументы, 162, 164
 - ◊ функции, 172
- Имитация, 347, 353, 360, 366
- Имитирование, 365
- Имитированные реализации, 347
- Инварианты, 85
- Инициализация, 100
- Интеграционные тесты, 45, 412

Интерпретация:

- ◊ данных, 76
- ◊ формата данных, 76

Интерфейс, 55, 69, 71, 73, 76, 265, 271, 272, 279, 353

Исключение, 299

- ◊ непроверенные, 87, 298
- ◊ проверенные, 87, 298

Исполнитель тестов, 334

Источники истины, 258

К

Качество кода, 16, 32, 36

Классицизм, 366

Классический подход, 366

Классы, 55, 60, 146, 280, 281, 285

Код, 17, 29, 32, 33, 37, 40, 64, 78, 101, 139, 177, 186, 269, 296, 301, 386

- ◊ адаптация, 277
- ◊ адаптируемый, 281
- ◊ взаимодействующий с сервером, 346
- ◊ взлом, 81
- ◊ вложенный, 157
- ◊ высококачественный, 46
- ◊ других программистов, 78
- ◊ многопоточный, 200
- ◊ многоразовый, 43, 62, 68
- ◊ модульный, 61, 67
- ◊ настроечный, 391
- ◊ ненадежный, 336
- ◊ обобщаемый, 43, 62
- ◊ переналадка, 392
- ◊ перенастраиваемый, 67
- ◊ понятный тестовый, 370
- ◊ проверяемый, 68
- ◊ тестируемый, 44, 62, 332, 365
- ◊ тестовый, 332, 339, 349
- ◊ улучшение, 65
- ◊ читабельный, 61, 153

Кодовая база, 29

Количество строк, 60

Комментарии, 144, 147, 148, 150

- ◊ встроенные, 165
- ◊ высокого уровня, 147
- ◊ документация, 87

Компиляция, 216, 279

Конструктор, 91, 92, 299

Контрактное программирование, 40

Контрольный пример, 332, 399, 406

Конфигурация, 401

- ◊ совместного использования, 393

Критический параметр, 209

Л

Линтеры, 156

Логика, 262

М

Магические числа, 185

Магическое значение, 129

Массив, 238

Мелкий шрифт, 91

- ◊ соглашения, 86

Метод:

- ◊ неявный, 113
- ◊ сигнализации об ошибке, 305, 307
- ◊ общий, 271
- ◊ явный, 113

Микросервис, 75

Многопоточность, 223

Модель поведения, 385

Модули, 55, 381

Модульная система, 41

Модульное тестирование, 16, 331, 413

Модульность, 41, 54

Модульные тесты, 45, 334, 340, 379

- ◊ изолированные, 366

Мониторинг, 351

Н

Названия функций и классов, 87

Названная константа, 167

Написание кода, 49

Написание модульных тестов, 16

Наследование, 277, 279

- ◊ классов, 274

Независим от деталей реализации, 335

Неизменяемые структуры данных, 238

Необъяснимые значения, 166, 167

Неправильное использование кода, 41

Непроверенное исключение, 117

- ◊ обработка, 118

- ◊ неявный, 117

Несовпадающие единицы измерения, 248

Неясные функции, 178
Нулевая безопасность, 50

О

Обнаружение поломок, 335
Обнаружитель совпадений в утверждениях, 405
Обобщаемость, 43, 325
Обработка:
 ◇ ошибок, 259
 ◇ перечислений, 211
 ◇ проверенного исключения, 116
Общедоступный API, 72, 341
Объекты данных, 246
Объем восстанавливаемости, 106
Одновременное тестирование, 385
Округление баланса, 360
Онлайн-журнал, 79
Оператор кода, 171
Описательные:
 ◇ имена, 143, 146
 ◇ типы, 163
Определение:
 ◇ кода, 32
 ◇ переменной, 307
Основные аргументы, 15
Отправка кода, 29
Ошибка, 113, 134, 275
 ◇ компилятора, 184
Ошибочная функция, 184

П

Пакеты, 55
Параметризованные тесты, 389
Парные типы, 243
Первичные данные, 253? 255
Переменные, 146? 309, 313
 ◇ глобальные, 309
Перечисления, 211, 295
Плохо объясненным, 405
Побочный эффект, 196, 198, 202, 206, 219
Повседневные решения, 28
Повторное использование кода, 16
Повышение производительности, 97
Подделки, 347, 362
Подзадачи, 73
Пользовательские имена, 406
Пользовательское приложение, 39

Пометка переменной, 307
Понятный тестовый код, 335
Последовательная связность, 60
Поставщик функций, 168
Постусловия, 85
Правильный способ, 47
Превращения кода, 30
Предварительные условия, 85
Предостережение, 218
Предотвращение отправки, 44
 ◇ неожиданных, 16
Предположение:
 ◇ необоснованное, 305
Представление времени, 247
Предупреждение компилятора, 138, 217
Преждевременная оптимизация, 303
Примеси, 284
Принцип инверсии зависимостей, 273
Проблема:
 ◇ с ромбовидным наследованием, 283
 ◇ хрупкого базового класса, 282
Проблемные иерархии, 283
Проверка кода, 30
 ◇ на разных языках, 95
 ◇ перед отправкой, 30
 ◇ постусловий, 94
 ◇ предварительных условий, 94
Программирование по соглашению, 85
Программное обеспечение, 30, 78, 99, 154, 332
Производные данные, 254
Производство, 30
Простота и быстрота запуска, 370
Пространства имен, 55
Противоположные крайности, 35
Протокол http, 52
Процесс:
 ◇ асинхронный, 127
 ◇ синхронный, 127
Псевдокод, 49, 50

Р

Работа со временем, 247
Разделение проблем, 60
Разработка:
 ◇ через тестирование, 367
 ◇ основанная на поведении, 367
 ◇ основанная на приемочных тестированиях, 368

Расчет остатка, 256
Расширение класса, 279
Реализация, 72, 122, 326, 343, 345

- ◊ нулевого объекта, 194

Регрессионное тестирование, 335, 413
Регрессия, 335
Резервная копия, 311
Результат по умолчанию, 216
Релиз, 30

- ◊ с ошибками, 44

Реструктуризация, 158
Ресурс, 100
Рефакторинг, 336, 337, 361

С

Сбои, 338
Связанные данные, 290
Связность, 60
Серверные фреймворки, 107
Сетевого ввода-вывода, 73
Сетевые ошибки, 100
Сигнализация, 122, 127

- ◊ об ошибках, 113, 130

Синтаксис, 170
Сквозные тесты, 45
Сложная форма шаблона, 193
Слои, 73

- ◊ абстракции, 16, 51, 52–55, 189, 297

Совместная конфигурация, 397
Соглашения по написанию кода, 16
Создание кода, 43

- ◊ модульного кода, 16

Соккрытие ошибки, 108
Состояние, 93

- ◊ совместного использования, 392

Специальный тип, 245
Статическая заводская функция, 91
Статическое цепляние, 270
Стиль кодирования:

- ◊ непоследовательный, 154
- ◊ последовательный, 154

Структурирование кода, 219
Структурное изменение кода, 336
Суперкласс, 277
Сценарии ошибок, 375
Сценарий, 111, 349

- ◊ Сценарий А, 34, 235
- ◊ Сценарий В, 34, 235

Т

Тест:

- ◊ написание, 366

Тестирование, 45, 46, 73, 344

- ◊ моделей, 387
- ◊ программного обеспечения, 412

Тестируемость, 45, 54
Тестовая константа:

- ◊ совместная, 401

Техника внедрения зависимостей, 312
Типажи, 284
Типы:

- ◊ возвращаемых данных, 87
- ◊ исключений, 295
- ◊ параметров, 87

Точное обнаружение:

- ◊ поломки, 334
- ◊ повреждений, 370

У

Уверенность в коде, 335
Универсальные типы, 326
Упорядочивание, 332, 334
Упрощение теста, 347, 348
Утверждение, 93, 96, 333, 334
Утечка:

- ◊ деталей, 292, 295
- ◊ сведений, 131

Ф

Фаззинг, 96, 413
Фрагмент кода, 101, 146, 376
Фреймворк, 107, 268, 269
Функции, 55, 57, 102, 146
Функциональная связность, 60
Функциональное:

- ◊ изменение, 336
- ◊ программирование, 171

Функциональность, 356
Функциональные изменения, 337
Функциональный стиль программирования, 174
Функция, 58, 117, 157, 198, 206, 224, 240, 288, 305, 310, 313, 322, 371

- ◊ анонимная, 171
- ◊ вспомогательная, 372, 402
- ◊ закрытая, 380
- ◊ контрольного примера, 390

- ◊ приватная, 376
- ◊ статистическая, 308

Х

Хакерский способ, 47
Хороший код, 28
Хорошо объясненные сбои, 335, 370, 405

Ц

Цвет по умолчанию, 90
Целочисленный тип, 249

Ч

Чистые слои абстракции, 277
Читабельность, 54
Чтение документации, 83

Ш

Шаблон:

- ◊ копирования, 227
- ◊ нулевого объекта, 119, 188, 190
- ◊ проектирования, 227

Шаблонный код, 269

Шрифт, 223

Э

Экземпляр, 404
Эмпирические правила, 61

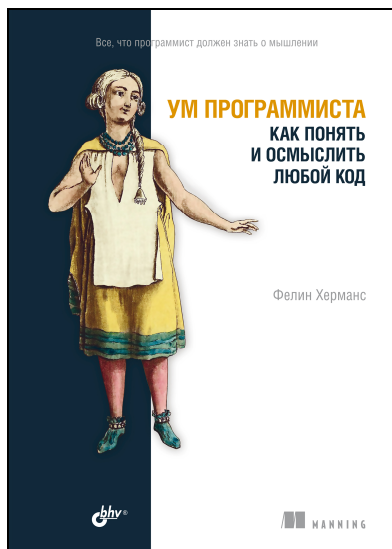
Я

Языки программирования, 17, 243
Языковая функция, 177, 178

Ум программиста. Как понять и осмыслить любой код

Отдел оптовых поставок:

e-mail: opt@bhv.ru



Прочитав эту книгу, вы

- узнаете, как именно ваш мозг видит код;
- научитесь бегло читать и быстро усваивать код;
- убедитесь, что есть простые приемы, позволяющие распутать самый сложный код;
- сможете навести порядок в любой, даже самой запущенной базе кода.

Книга освещает практические основы когнитивистики для программистов. Основные темы: осмысление и развитие чужого и собственного кода, изучение новых языков программирования, мнемонические приемы для программистов, поддержка кода в читаемом состоянии. Объяснено, как снижать когнитивную нагрузку при работе программиста, как делать код логичным и понятным для себя и коллег. Рассмотрены приемы именования функций, классов и переменных, подходы к ведению репозиторий, совместной разработке и доработке кода.

Доктор Фелин Херманс, приглашенный профессор в Лейденском университете в Нидерландах. Вот уже более 10 лет она исследует механизмы программирования, выясняя, как лучше учиться этому искусству и обучать других.



ИНТЕРНЕТ-МАГАЗИН

BHV.RU

КНИГИ, РОБОТЫ,
ЭЛЕКТРОНИКА

Интернет-магазин издательства «БХВ»

- Более 25 лет на российском рынке
- Книги и наборы по электронике и робототехнике по издательским ценам
- Электронные архивы книг и компакт-дисков
- Ответы на вопросы читателей



Скидка 15%

на бумажные книги
по промокоду: **JANE22**