

Роберт Мартин рекомендует

Пять строк кода

Кристиан Клаусен

Предисловие Роберта С. Мартина



Краткий обзор шаблонов рефакторинга

Извлечение метода (Extract method) (3.2.1) — часть метода извлекается в отдельный метод.

Замена кода типа классами (Replace type code with classes) (4.1.3) — преобразование перечисления в интерфейс. При этом значения перечислений становятся классами.

Перемещение кода в классы (Push code into classes) (4.1.5) — является продолжением шаблона «Замена кода типа классами» (4.1.3), так как перемещает в классы функциональность.

Встраивание метода (Inline method) (4.1.7) — удаление методов, которые уже не повышают читаемость программы.

Специализация метода (Specialize method) (4.2.2) — удаление из методов ненужного и проблемного обобщения.

Пробное удаление с последующей компиляцией (Try delete then compile) (4.5.1) — удаление из интерфейсов и классов неиспользуемых методов, когда известна вся их область видимости.

Объединение схожих классов (Unify similar classes) (5.1.1) — объединение двух или более классов, отличающихся друг от друга набором постоянных методов.

Совмещение инструкций if (Combine ifs) (5.2.1) — уменьшает повторяемость за счет объединения последовательных инструкций `if` с одинаковыми телами.

Введение паттерна «Стратегия» (Introduce strategy pattern) (5.4.2) — замена вариативности `if` инстанцированием (созданием экземпляра) классов.

Извлечение интерфейса из реализации (Extract interface from implementation) (5.4.4) — замена зависимостей в классе на интерфейс.

Удаление геттера или сеттера (Eliminate getter or setter) (6.1.3) — удаление геттеров и сеттеров за счет сближения их функциональности с данными.

Инкапсуляция данных (Encapsulate data) (6.2.3) — локализация инвариантов, относящихся к переменным, ведущая к более наглядной связности.

Обеспечение последовательности (Enforce sequence) (6.4.1) — с помощью компилятора обеспечивается выполнение действий в определенном порядке.

Five Lines of Code

HOW AND WHEN TO REFACTOR

CHRISTIAN CLAUSEN

FOREWORD BY ROBERT C. MARTIN



MANNING

SHELTER ISLAND

Пять строк кода

Роберт Мартин рекомендует

Кристиан Клаусен

Предисловие Роберта С. Мартина



Санкт-Петербург • Москва • Минск

2023

ББК 32.973.2-018-02
УДК 004.415
К47

Клаусен Кристиан

К47 Пять строк кода. Роберт Мартин рекомендует. — СПб.: Питер, 2023. — 368 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1959-2

В каждой кодовой базе есть ошибки и слабые места, которые нужно найти и исправить. Правильный рефакторинг сделает ваш код элегантным, удобным для чтения и простым в обслуживании.

Познакомьтесь с уникальным подходом, позволяющим реализовать любой метод в пяти строках кода. И не забывайте про тайну, хорошо известную большинству senior-разработчиков: иногда проще ухудшить код и вернуться к его исправлению позже.

«Пять строк кода» — это свежий взгляд на рефакторинг для разработчиков любого уровня. Вы узнаете, когда проводить рефакторинг, как использовать паттерны, а также научитесь определять признаки, которые говорят о том, что код необходимо удалить.

Для разработчиков всех уровней. В примерах используется доступный и понятный синтаксис TypeScript, который позволяет перейти к любому языку высокого уровня.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018-02
УДК 004.415

Права на издание получены по соглашению с Manning Publications. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1617298318 англ.

Original English language edition published by Manning Publications USA.
© 2021 by Manning Publications.

ISBN 978-5-4461-1959-2

© Перевод на русский язык ООО «Прогресс книга», 2022
© Издание на русском языке, оформление ООО «Прогресс книга», 2022
© Серия «Библиотека программиста», 2022

Краткое содержание

https://t.me/it_boooks

Предисловие	16
Введение.	18
Благодарности	23
Об авторе	25
Иллюстрация на обложке	26
От издательства	27
Глава 1. Рефакторинг рефакторинга	28
Глава 2. Суть рефакторинга.	43

Часть I

Учимся на рефакторинге компьютерной игры

Глава 3. Разбивка длинных функций	54
Глава 4. Пусть код типа работает	77
Глава 5. Совмещение схожего кода	121
Глава 6. Защита данных	174

Часть II

Применение полученных знаний в реальной жизни

Глава 7. Сотрудничество с компилятором.	216
Глава 8. Избегайте комментариев	242
Глава 9. Страсть к удалению кода	249
Глава 10. Никогда не бойтесь добавлять код	278
Глава 11. Соблюдение структуры в коде.	296
Глава 12. Избегайте оптимизаций и обобщенности	318
Глава 13. Пусть плохой код выглядит плохо	337
Глава 14. Подведение итогов	354
Приложение. Установка инструментов для части I	364

Оглавление

Предисловие	16
Введение.	18
Цель: избранные правила и шаблоны рефакторинга	19
Аудитория и план изложения.	20
О преподавании	20
О коде	22
Дополнительный проект.	22
Благодарности	23
Об авторе	25
Иллюстрация на обложке	26
От издательства	27
Глава 1. Рефакторинг рефакторинга	28
1.1. Что такое рефакторинг	30
1.2. Навыки: что требует рефакторинга.	31
1.2.1. Пример запаха кода	32
1.2.2. Пример правила.	32
1.3. Культура: когда проводить рефакторинг	33
1.3.1. Рефакторинг старых унаследованных систем	35
1.3.2. Когда рефакторинг делать не нужно	35
1.4. Инструменты: как проводить рефакторинг (безопасно)	36
1.5. Инструменты, необходимые для начала	37
1.5.1. Язык программирования: TypeScript.	37

1.5.2. Редактор: Visual Studio Code.	38
1.5.3. Контроль версий: Git	38
1.6. Общий пример: 2D-головоломка	39
1.6.1. Практика ведет к совершенству: вторая база кода	41
1.7. Примечание по реальным программам	41
Резюме	42
Глава 2. Суть рефакторинга.	43
2.1. Улучшение читаемости и обслуживаемости	43
2.1.1. Улучшение кода.	44
2.1.2. Обслуживание кода... без изменения его функциональности	47
2.2. Выработка скорости, гибкости и стабильности	47
2.2.1. Выбирайте композицию вместо наследования	48
2.2.2. Изменение кода путем добавления, а не изменения	49
2.3. Рефакторинг и повседневная работа.	50
2.3.1. Рефакторинг как метод для освоения	51
2.4. Определение «области» в контексте программного обеспечения.	52
Резюме	52

Часть I

Учимся на рефакторинге компьютерной игры

Глава 3. Разбивка длинных функций.	54
3.1. Определяем первое правило: почему пять строк?	55
3.1.1. Правило «Пять строк».	56
3.2. Шаблон проектирования для разбивки функций.	58
3.2.1. Шаблон рефакторинга «Извлечение метода»	63
3.3. Разбивка функций для уравнивания абстракций	66
3.3.1. Правило «Вызов или передача».	67
3.3.2. Применение правила	68
3.4. Свойства хорошего имени функции	69
3.5. Разбивка функций, делающих слишком много	72
3.5.1. Правило «if только в начале»	72
3.5.2. Применение правила	73
Резюме	76

Глава 4. Пусть код типа работает	77
4.1. Рефакторинг простой инструкции if	78
4.1.1. Правило «Никогда не использовать if с else»	78
4.1.2. Применение правила	80
4.1.3. Шаблон рефакторинга «Замена кода типа классами»	82
4.1.4. Перемещение кода в классы	86
4.1.5. Шаблон рефакторинга «Перемещение кода в классы»	89
4.1.6. Встраивание избыточного метода	93
4.1.7. Шаблон рефакторинга «Встраивание метода»	94
4.2. Рефакторинг большой инструкции if	97
4.2.1. Устранение обобщенности	100
4.2.2. Шаблон рефакторинга «Специализация метода».	102
4.2.3. Допускается только одна инструкция switch	104
4.2.4. Правило «Никогда не использовать switch»	106
4.2.5. Удаление if	108
4.3. Разбираемся с повторением кода	110
4.3.1. Разве нельзя было использовать вместо интерфейсов абстрактные классы?	112
4.3.2. Правило «Наследовать только от интерфейсов»	113
4.3.3. Зачем все это повторение кода?	114
4.4. Рефакторинг двух сложных выражений if	114
4.5. Удаление мертвого кода.	118
4.5.1. Шаблон рефакторинга «Пробное удаление с последующей компиляцией»	119
Резюме	120
Глава 5. Совмещение схожего кода	121
5.1. Объединение схожих классов	122
5.1.1. Шаблон рефакторинга «Объединение схожих классов»	130
5.2. Объединение простых условий	136
5.2.1. Шаблон рефакторинга «Совмещение инструкций if»	138
5.3. Объединение сложных условий.	139
5.3.1. Использование правил арифметики для условий.	140
5.3.2. Правило «Использовать чистые условия»	141
5.3.3. Применение условной арифметики	144
5.4. Объединение кода среди классов	146
5.4.1. Введение диаграмм классов UML для отражения связи классов	151

5.4.2. Шаблон рефакторинга «Введение паттерна “Стратегия”» . . .	153
5.4.3. Правило «Избегать интерфейсов с единственной реализацией»	161
5.4.4. Шаблон рефакторинга «Извлечение интерфейса из реализации»	162
5.5. Объединение похожих функций	165
5.6. Объединение схожего кода.	168
Резюме	173
Глава 6. Защита данных	174
6.1. Инкапсуляция с помощью геттеров	175
6.1.1. Правило «Не использовать геттеры или сеттеры»	175
6.1.2. Применение правила	178
6.1.3. Шаблон рефакторинга «Удаление геттера или сеттера».	181
6.1.4. Удаление последнего геттера.	183
6.2. Инкапсулирование простых данных.	187
6.2.1. Правило «Всегда избегать общих аффиксов»	187
6.2.2. Применение правила	189
6.2.3. Паттерн рефакторинга «Инкапсуляция данных».	194
6.3. Инкапсулирование сложных данных	197
6.4. Устранение инварианта последовательности	204
6.4.1. Шаблон рефакторинга «Обеспечение последовательности»	205
6.5. Устранение перечислений иным способом	208
6.5.1. Перечисление с помощью закрытых конструкторов	208
6.5.2. Переотображение чисел в классы.	211
Резюме	213

Часть II

Применение полученных знаний в реальной жизни

Глава 7. Сотрудничество с компилятором.	216
7.1. Близкое знакомство с компилятором	217
7.1.1. Слабость: проблема останова имеет ограниченную информативность во время компиляции.	218
7.1.2. Сильная сторона: достижимость гарантирует возвращение из методов	219
7.1.3. Сильная сторона: явное присваивание предотвращает обращение к неинициализированным переменным	220

7.1.4. Сильная сторона: контроль доступа помогает инкапсулировать данные.	.221
7.1.5. Сильная сторона: проверка типов подтверждает свойства . . .	.221
7.1.6. Слабость: разыменовывание null рушит приложение	.223
7.1.7. Слабость: арифметические ошибки вызывают переполнение или сбой.	.223
7.1.8. Слабость: ошибки выхода за допустимый диапазон вызывают сбой приложения.	.224
7.1.9. Слабость: бесконечные циклы стопорят приложение	.224
7.1.10. Слабость: взаимные блокировки и состояния гонки вызывают нежелательное поведение	.225
7.2. Использование компилятора	.227
7.2.1. Подключаем компилятор к работе	.228
7.2.2. Не перечьте компилятору.	.230
7.3. Доверие к компилятору.	.236
7.3.1. Учим компилятор инвариантам.	.236
7.3.2. Обращайте внимание на предупреждения	.239
7.4. Исключительное доверие к компилятору.	.240
Резюме	.240
Глава 8. Избегайте комментариев	.242
8.1. Удаление устаревших комментариев.	.244
8.2. Удаление закомментированного кода	.245
8.3. Удаление бессмысленных комментариев	.246
8.4. Преобразование комментариев в имена методов	.246
8.4.1. Использование комментариев для планирования	.247
8.5. Сохранение комментариев к инвариантам	.247
8.5.1. Инварианты в процессе	.248
Резюме	.248
Глава 9. Страсть к удалению кода	.249
9.1. Удаление кода может стать очередной вехой.	.251
9.2. Удаление кода для устранения ненужной сложности	.252
9.2.1. Техническое неведение ввиду неопытности	.252
9.2.2. Технические потери из-за нехватки времени.	.254
9.2.3. Технический долг под давлением обстоятельств	.254
9.2.4. Техническая задержка из-за роста	.255

9.3. Категоризация кода по степени его близости	256
9.4. Удаление кода в старых унаследованных системах.	257
9.4.1. Пояснение кода с помощью шаблона «Фигус-удавка»	257
9.4.2. Использование «Фигуса-душителя» для улучшения кода	260
9.5. Удаление кода из замороженного проекта	261
9.5.1. Получение желаемого результата по умолчанию	261
9.5.2. Минимизация затрат с помощью отрыва и стабилизации	262
9.6. Удаление веток в системе контроля версий	262
9.6.1. Минимизация затрат за счет ограничения количества веток.	263
9.7. Удаление документации кода	264
9.7.1. Алгоритм для определения необходимости документирования.	265
9.8. Удаление тестирующего кода	266
9.8.1. Удаление оптимистичных тестов	267
9.8.2. Удаление пессимистичных тестов	267
9.8.3. Исправление или удаление ненадежных тестов.	267
9.8.4. Рефакторинг кода для избавления от плохих тестов.	268
9.8.5. Специализация тестов для их ускорения	268
9.9. Удаление кода дополнительной конфигурации	269
9.9.1. Ограничение конфигурации настраиваемости во времени.	269
9.10. Удаление кода для сокращения числа библиотек	271
9.10.1. Ограничение использования внешних библиотек.	274
9.11. Удаление кода из работающего функционала	275
Резюме	276
Глава 10. Никогда не бойтесь добавлять код	278
10.1. Принятие неуверенности: встретить опасность лицом к лицу	279
10.2. Использование отрыва для преодоления страха создать что-то неправильно	280
10.3. Преодоление страха перед лишними затратами или риском установки фиксированного соотношения	281
10.4. Преодоление страха перед неудачей за счет постепенной разработки	283
10.5. Как копираст влияет на скорость	284
10.6. Изменение путем добавления через расширяемость	286

12 Оглавление

10.7. Изменение путем добавления поддерживает обратную совместимость	287
10.8. Изменение путем добавления с помощью переключателей функционала	288
10.9. Изменение путем добавления с помощью ветвления через абстрагирование	292
Резюме	295
Глава 11. Соблюдение структуры в коде	296
11.1. Категоризация структуры на основе области и источника	297
11.2. Три способа, которыми код отражает поведение.	298
11.2.1. Выражение поведения в потоке управления	298
11.2.2. Выражение поведения в структуре данных	300
11.2.3. Выражение поведения в данных	303
11.3. Добавление кода для раскрытия структуры	305
11.4. Наблюдение вместо прогнозирования и использование эмпирических техник	306
11.5. Обеспечение безопасности без понимания кода	307
11.5.1. Обеспечение безопасности через тестирование	308
11.5.2. Обеспечение безопасности за счет мастерства	308
11.5.3. Обеспечение безопасности с помощью инструментов	308
11.5.4. Обеспечение безопасности через формальную верификацию	309
11.5.5. Обеспечение безопасности через толерантность к ошибкам	309
11.6. Определение неэксплуатируемых структур	309
11.6.1. Эксплуатация пустого пространства с помощью извлечения и инкапсуляции	310
11.6.2. Эксплуатация дублирования с помощью объединения	311
11.6.3. Эксплуатация общих аффиксов с помощью инкапсуляции	314
11.6.4. Эксплуатация типа среды выполнения с помощью динамической диспетчеризации	316
Резюме	317
Глава 12. Избегайте оптимизаций и обобщенности	318
12.1. Стремление к простоте	319
12.2. Когда и как вносить обобщенность	321
12.2.1. Создание минимальной функциональности	322

12.2.2. Объединение компонентов с похожим уровнем стабильности	.323
12.2.3. Устранение ненужной обобщенности	.323
12.3. Когда и как оптимизировать	.323
12.3.1. Рефакторинг перед оптимизацией	.324
12.3.2. Оптимизация согласно теории ограничений	.326
12.3.3. Координирование оптимизации с помощью метрик	.329
12.3.4. Выбор удачных алгоритмов и структур данных	.330
12.3.5. Использование кэширования	.331
12.3.6. Изоляция оптимизированного кода	.333
Резюме	.335
Глава 13. Пусть плохой код выглядит плохо	.337
13.1. Проблемы привлечения внимания к плохому коду.	.338
13.2. Разделение на безупречный и legacy-код	.339
13.2.1. Теория разбитого окна	.340
13.3. Подходы к определению плохого кода.	.340
13.3.1. Правила в этой книге: простые и конкретные	.340
13.3.2. Запахи кода: полноценные и абстрактные.	.341
13.3.3. Цикломатическая сложность: алгоритмическая (объективная)	.342
13.3.4. Когнитивная сложность: алгоритмическая (субъективная)	.342
13.4. Правила безопасного ухудшения кода	.343
13.5. Методы безопасного ухудшения кода	.344
13.5.1. Использование перечислений	.344
13.5.2. Использование целых чисел и строк в качестве кода типа.	.345
13.5.3. Добавление в код магических чисел	.346
13.5.4. Добавление в код комментариев.	.346
13.5.5. Добавление в код пробелов	.347
13.5.6. Группировка элементов на основе имен	.348
13.5.7. Добавление контекста в имена.	.349
13.5.8. Создание длинных методов.	.350
13.5.9. Добавление в методы большого числа параметров	.351
13.5.10. Использование геттеров и сеттеров	.352
Резюме	.353

Глава 14. Подведение итогов	354
14.1. Краткий обзор пройденного	354
14.1.1. Введение: мотивация	355
14.1.2. Часть I: конкретизирование	355
14.1.3. Часть II: расширение горизонтов	355
14.2. Раскрытие внутренней философии.	356
14.2.1. Поиск все меньших шагов.	356
14.2.2. Поиск внутренней структуры	357
14.2.3. Использование правил для совместной работы	357
14.2.4. Интересы команды важнее личных интересов	358
14.2.5. Простота важнее универсальности	359
14.2.6. Использование объектов или функций высшего порядка. . .	360
14.3. Куда двигаться дальше.	361
14.3.1. Микроархитектура	361
14.3.2. Макроархитектура	361
14.3.3. Качество программного обеспечения.	361
Резюме	362
Приложение. Установка инструментов для части I	364
Node.js	364
TypeScript	364
Visual Studio Code.	365
Git.	365
Настройка проекта TypeScript	365
Создание проекта TypeScript	365
Как настроить уровень	366

Посвящается моим университетским наставникам, которые говорили мне:

*«Секрет постоянного успеха и превосходства
в тяжелом ежедневном труде».*

Оливье Данвиш

«Ты упускаешь суть».

Майер Голдберг

*Благодарю вас за то, что научили меня не стараться сделать правильно,
а просто делать правильно.*

Предисловие

Сталкивались ли вы со сложностями чтения литературы по программному обеспечению из-за нагромождения терминологии и запутанных принципов? Возникало ли у вас ощущение, что книга написана для круга посвященных, к которым вы не относитесь?

Так вот, эта книга написана иначе. Материал изложен простым языком и только по самой сути предмета.

Хотя и пособием для начинающих предлагаемая книга тоже не является. Изложение здесь не начинается с азов и утомительных основ программирования, и не стоит рассчитывать на то, что вас бережно проведут через тернии постижения предмета. Я гарантирую, что чтение этой книги станет для вас испытанием, но вполне посильным. Оно пройдет без пугающих сложностей и высокомерного жонглирования терминологией со стороны автора.

Рефакторинг, а именно ему посвящена книга, — это дисциплина, которая нацелена делать из плохого кода хороший без нарушения его функциональности. Если учесть, что существование современного цивилизованного мира уже немислимо без программного обеспечения, то становится очевидной исключительная актуальность этой темы.

Возможно, вы сочтете подобное утверждение преувеличением, но я буду настаивать на своем. Оглянитесь вокруг. Сколько в данный момент на вашем теле находится процессоров, выполняющих те или иные программы? Часы, телефон, ключи от машины, наушники... сколько их в радиусе 30 метров? Ваша микроволновка, кухонная плита, посудомойка, стиральная машина... автомобиль, наконец.

Сегодня в нашем обществе ничего не происходит без участия программ. Без них уже нельзя ни купить или продать что-либо, ни приготовить еду, ни посмотреть ТВ, ни позвонить друзьям...

А сколько из всех этих программ реально представляют собой хороший код? Подумайте о системах, с которыми работаете в данный момент. Прозрачна ли их реализация, понятна ли эксплуатация? А может, как и большинство других, они представляют собой мешанину, отчаянно жаждущую рефакторинга?

Эта книга рассказывает не о стерильном и отвлеченном рефакторинге, о котором вы могли слышать или читать до этого. Она знакомит вас с *реальным* рефакторингом в *реальных* проектах, рефакторингом унаследованных нашим поколением устаревших, но широко распространенных legacy-систем, рефакторингом в таких средах, с которыми мы все сталкиваемся чуть ли не каждый день.

Более того, я не стану никого винить в отсутствии автоматизированного тестирования программ. Ведь очевидно, что большинство унаследованных систем росли, развивались с очень давних пор, поэтому наличием подобных тестов они нас не балуют.

Итак, здесь изложены простые правила, которым вы сможете следовать, чтобы уверенно рефакторить сложные, запутанные, беспорядочные, непротестированные системы. Изучив эти правила и придерживаясь их, вы сможете эффективно улучшить обслуживаемые вами базы кода (совокупности листингов программ).

Но сразу оговорюсь: обучая рефакторингу, автор не вручает вам волшебную палочку. Рефакторинг старого, неработоспособного, непротестированного кода никогда не будет легким. Однако, вооружившись описанными в книге правилами и примерами, вы сможете пробиться через всю запутанность таких систем, решить проблемы с их сопровождением, которые долго не давали вам покоя.

Так что рекомендую читать внимательно. Изучайте примеры. Пытайтесь как следует понять, с какой именно целью они приведены. Работайте над усовершенствованием приводимого в книге кода, как это рекомендовано по ходу повествования. На это уйдет время. Это будет раздражать. Это станет испытанием. Но пройдите от начала до конца весь путь, переплывите широкую реку — и будете вознаграждены.

К другому берегу вы причалите уже с багажом навыков и знаний, которые будут служить верой и правдой в будущих проектах, с выработанным чутьем и умением отличать хороший код от плохого, с пониманием, что именно делает код ясным и понятным.

Роберт К. Мартин (Дядя Боб)

Введение

Мой отец учил меня писать программный код, когда я еще был совсем ребенком, так что я имею дело со структурой программ, сколько себя помню. А еще я всегда стремился помогать людям. Вставал по утрам именно с этим намерением. Поэтому неудивительно, что я увлекся преподаванием. И как только мне предложили должность ассистента кафедры в университете, я без раздумий согласился. У меня было несколько таких подработок, но наступил день, когда мое везение закончилось, и в один из семестров преподавать оказалось нечего.

Будучи предприимчивым, я решил создать студенческую организацию, в которой учащиеся могли бы учить друг друга. Возможность выступить давалась каждому, а тематика была самая разнообразная: от навыков, полученных в работе со сторонними проектами, до продвинутых знаний, вообще не входивших в учебный курс. На тот момент я предположил, что это позволит мне заниматься преподаванием, и не ошибся. Оказалось, что специалисты по информатике в большинстве своем робкие люди, так что мне пришлось самому вести лекции почти 60 недель подряд, а дальше уже все пошло по накатанному пути. За это время я глубоко вник в темы, которые преподавал, и в сам процесс преподавания. Эти лекции привлекли множество любознательных людей, и в их среде, в этом своеобразном сообществе я познакомился со своими лучшими друзьями.

Некоторое время спустя, уже покинув университет, я как-то отдыхал с одним из этих друзей. Нам было скучно, и он спросил, могу ли я сымпровизировать лекцию, как делал это раньше. Я ответил: «Давай попробуем». Мы открыли ноутбук, и я на одном дыхании настроил то, что, по существу, послужило затем основанием части I этой книги.

Когда я перестал печатать, мой друг был ошеломлен. Он решил, что это будет шуточное представление, но я-то думал иначе. Я действительно хотел научить его рефакторингу.

Моей целью было сделать так, чтобы через час он уже мог писать код, как если бы мастерски умел его рефакторить. Поскольку тема рефакторинга, как и любого усовершенствования программного кода, весьма непростая, было очевидно, что нам придется имитировать сам процесс. Я взглянул на код и постарался придумать правила, которые подскажут другу, что нужно делать, и при этом будут легко запоминаться. Хотя это и была спонтанная игра, выполняя предложенные мной упражнения, товарищ реально добился заметного улучшения кода. Полученный результат оказался настолько многообещающим, а внесенное улучшение настолько быстрым, что по возвращении в тот вечер домой я записал все, что мы разобрали. Я стал практиковать это при найме на работу молодых специалистов, и в результате постепенно мне удалось собрать, систематизировать и доработать правила и шаблоны рефакторинга и представить их в этой книге.

ЦЕЛЬ: ИЗБРАННЫЕ ПРАВИЛА И ШАБЛОНЫ РЕФАКТОРИНГА

Совершенство достигнуто не тогда, когда больше нечего добавить, а когда уже нечего убрать.

Антуан де Сент-Экзюпери

В мире существуют сотни шаблонов рефакторинга. Я же решил включить в книгу только 13. На мой взгляд, лучше в совершенстве овладеть небольшим количеством техник, чем поверхностно охватить большее их количество. К тому же я хотел сформировать полноценную связную систему, так как это привносит ощущение перспективы и упрощает организацию материала в уме. То же можно сказать и о правилах.

Нет ничего нового под Солнцем.

Книга Екклесиаста

Я не утверждаю, что вложил в книгу много нового, но считаю, что скомпоновал ее содержание не только интересно, но и эффективно. Многие правила были взяты из книги Роберта К. Мартина «Чистый код»¹, но при этом изменены для простоты

¹ *Мартин Р.* Чистый код. Создание, анализ и рефакторинг. — Питер, 2021.

понимания и применения. Многие шаблоны рефакторинга позаимствованы из книги Мартина Фаулера «Рефакторинг»¹, но адаптированы под использование компилятора вместо упора на сильные наборы тестов.

АУДИТОРИЯ И ПЛАН ИЗЛОЖЕНИЯ

Книга состоит из двух частей, сильно различающихся по стилю. Часть I закладывает прочный фундамент для выполнения рефакторинга и адресована каждому отдельно взятому разработчику. Не стремясь охватить вообще все стороны рефакторинга, я решил сделать упор на простоту изложения. Эта часть предназначена для тех, кому еще предстоит сформировать устойчивую основу для проведения рефакторинга, например для студентов, для начинающих программистов, в том числе самоучек. Если же, взглянув на исходный код в книге, вы решите, что улучшить его несложно, тогда часть I не для вас.

В части II фокус смещается на контекст применения рефакторинга и адресуется команде разработчиков. Здесь я выбрал наиболее ценные, на мой взгляд, уроки разработки программного обеспечения (ПО) в реальном, а не учебном мире. Некоторые темы состоят в основном из теории, например «Сотрудничайте с компилятором» и «Соблюдайте структуру в коде». Другие же в первую очередь практические, например «Страсть к удалению кода» и «Пусть плохой код выглядит плохо». Таким образом, часть II уже более универсальна, и даже опытные разработчики найдут здесь чему поучиться.

Поскольку во всех главах части I используется один общий пример, они тесно связаны между собой, и предполагается, что читать их будут в предложенной очередности. А вот часть II состоит уже из самостоятельных глав, связанных между собой только некоторыми ссылками из одной на другую. Если у вас нет времени на прочтение всей книги, можете просто выбрать наиболее интересные темы в части II и ознакомиться только с ними.

О ПРЕПОДАВАНИИ

Я много размышлял о преподавании. Передача знаний и навыков несет в себе много сложностей. Учитель должен стимулировать у студентов интерес, уверенность и отдачу. Но по факту мозг учащегося склонен к сохранению энергии, поэтому постоянно стремится отвлечься от процесса обучения.

Чтобы побороть эту склонность ума, сначала нужно пробудить мотивацию. Для этого я обычно привожу простое с виду упражнение. Когда студенты понимают,

¹ Фаулер М. Рефакторинг.

что не могут его выполнить, их охватывает естественное любопытство. В этом и состоит цель кода из сквозного примера части I. «Оптимизировать базу кода» — это на первый взгляд так просто. Но при ближайшем рассмотрении код оказывается уже настолько качественным, что многие просто не знают, что еще можно в нем улучшить.

На втором этапе нужно подтолкнуть студентов к уверенному экспериментированию и применению новых знаний. Всю важность этого я осознал во время факультативов по французскому языку. Когда преподавательница хотела научить нас новой фразе, она всегда делала это в три этапа.

1. Просила каждого повторить эту фразу дословно. На этом этапе чистой имитации мы были вынуждены повторить ее один раз.
2. Задавала каждому вопрос. Вопрос не всегда был понятен, но по интонации было очевидно, что это именно вопрос. Поскольку других вариантов у нас не было, приходилось еще раз повторять фразу. Такое повторение внушало уверенность и предоставляло нам первый пример контекста применения изучаемой фразы. На этом этапе зарождалось понимание.
3. Просила использовать фразу в диалоге. Умение синтезировать нечто новое — это сама цель преподавания, и для этого требуется не только понимание, но и уверенность.

Позже я узнал, что этот подход повторяет японский принцип *сюхари* из боевых искусств, он сейчас становится все более популярным. Состоит он из трех частей: «сю» означает полную имитацию, когда повторение производится без вопросов и собственного понимания; «ха» уже подразумевает некоторую вариацию, то есть выполнение изученного в несколько обновленной форме; «ри» же означает оригинальность, то есть полностью свободное применение, отличное от усвоенного автоматически.

Содержание части I книги полностью построено по принципу сюхари. Я рекомендую сначала следовать правилам без их понимания. Затем, когда вы осознаете их ценность, сможете придумывать вариации. В завершение же, когда освоите их в совершенстве, сможете переходить к пониманию запахов кода. Что касается шаблонов рефакторинга, я буду показывать, как применять каждый из них в коде, и нужно будет за мной повторять (это имитация). Затем те же шаблоны я буду демонстрировать в иных контекстах (стадия вариации), а в завершение предложу рассмотреть другое применение шаблона, призывая вас использовать его самостоятельно (стадия синтеза).

При этом вы сможете воспользоваться книгой для оценки самого процесса и тегами Git для оценки кода. Если же вы не будете отслеживать изменения кода примера и вникать в них, то все описанные действия будут казаться излишними повторениями одного и того же. Поэтому настоятельно советую в процессе чтения части I все же параллельно прорабатывать приводимые в ней примеры на компьютере.

О КОДЕ

Эта книга содержит множество примеров исходного кода, который приводится как в нумерованных листингах, так и в самом тексте. В обоих случаях шрифт кода имеет моноширинный формат, подобный этому, чтобы отличаться от прочего текста.

Во многих случаях оригинальный исходный код был переформатирован. Например, были добавлены разрывы строк и переработаны отступы, чтобы грамотно вписать все в доступное пространство книги. Помимо этого, во многих случаях, когда код описывался в самом тексте, соответствующие комментарии из него удалялись. В то же время множество листингов снабжено комментариями, подчеркивающими важные моменты.

Код примеров из этой книги доступен для скачивания на сайте издательства Manning (<https://www.manning.com/books/five-lines-of-code>), а также в моем репозитории на GitHub (<https://github.com/thedrlambda/five-lines>).

ДОПОЛНИТЕЛЬНЫЙ ПРОЕКТ

Чтобы вы могли лучше понять, как использовать приводимые в книге правила и шаблоны рефакторинга, я придумал дополнительный проект. Этот проект более продвинутый, и готовое решение к нему не прилагается. Всех заинтересованных он будет ждать на GitHub: <https://github.com/thedrlambda/bomb-guy>. Удачи!

Благодарности

Во-первых, я бы не стал тем, кем являюсь, и уж точно не написал бы эту книгу, если бы не два человека, которым она посвящается: Оливье Данви и Майер Голдберг. Моя благодарность вам безгранична. Вы научили меня теории типов и лямбда-исчислению, что и стало основой этой работы. Но, как и все превосходные учителя, вы сделали намного больше. Оливье, я знаю, для вас это стало неожиданностью, но для меня вовсе не удивительно, что вы больше всех в ученых кругах заслуживаете благодарности. Вы с легкостью даете советы, которые можно тут же применить и которые сохраняют свою актуальность даже спустя годы. Майер, ваш неутолимый энтузиазм, терпение и методика обучения всевозможным сложным темам программирования позволили мне выработать собственную модель восприятия и преподавания этой науки.

Хочу также выразить глубокую благодарность Роберту К. Мартину. Если кто-то сочтет эту книгу настолько же вдохновляющей, насколько я нахожу твою, то меня это безмерно порадует. Кроме того, я очень признателен за то, что ты не пожалел времени для составления обзора моей работы и решил написать к ней вступительное слово.

Еще одним основным участником создания книги стал дизайнер Ли Макгори. Спасибо тебе! Твоя креативность и мастерство вывели уровень иллюстраций на уровень содержания.

Глубочайшие благодарности всем сотрудникам команды Manning. Мой рецензент Эндрю Уолдрон дал потрясающую обратную связь и проявил энтузиазм, в результате чего я и решился работать именно с Manning. Мой редактор-консультант Хелен Стергиус стала моим проводником на пути, потребовавшем невероятных усилий по написанию этой книги. Без ее вдохновения и полезной

обратной связи такого уровня результата достичь бы точно не удалось. В качестве технического редактора выступил Марк Элстон, чьи комментарии всегда были мудры и точны. Его взгляд на раскрываемые в книге темы всегда дополнял мой. Благодарю также выпускающего редактора, команду маркетологов и само издательство Manning за сотрудничество и проявленное терпение.

Отдельную благодарность направляю людям, которые выступали наставниками в моей работе. Спасибо Джейкобу Блему. Ты на своем примере научил меня быть технически превосходным консультантом без ущерба для себя и своих ценностей. Твоя страсть к собственной деятельности проявляется в том, что ты способен узнать и вспомнить код, с которым работал десять лет назад, — меня это до сих пор поражает. Спасибо Клаусу Норрегаарду. Твой уровень внутреннего спокойствия и благодати восхищает меня. Спасибо Йохану Абильдсков. Я еще не встречал человека с настолько обширными и глубокими техническими знаниями, которые соперничают по своей силе только с твоей добротой. Без тебя эта книга никогда бы так и не покинула заповедные пространства моего жесткого диска. Благодарю также всех, кого я обучал и с кем мне довелось близко работать.

Помимо этого выражаю признательность тем, кто принял участие в подготовке этой книги через обратную связь и бесконечные технические беседы. Я принимал решение проводить время с вами, потому что вы делаете мою жизнь лучше. Спасибо Ганнибалу Кебловски. Твое любопытство породило саму идею написания этой книги. Спасибо Миккелю Крингельбаху. Ты помогал мне всякий раз, когда я просил, тренируя мой интеллект, а также делаясь своими идеями и опытом. Все это существенно помогло в создании книги. Спасибо Миккелю Бруну Якобсену. Твоя страсть и уровень компетенции в сфере программного обеспечения вдохновляют меня и подталкивают к совершенствованию. Спасибо всем, кто являлся частью сообщества взаимного обучения. Ваша неутолимая тяга к знаниям заставляла меня продолжать ими делиться. В частности, благодарю Суне Орт Сёренсен, Матиаса Воррейтера Педерсена, Йенса Йенсена, Каспера Фрексена, Матиаса Бака, Фредерика Бринка Трульсена, Кента Григо, Джона Смедегаарда, Ричарда Мёна, Кристофера Нёддебо Кнудсена, Кеннета Хансена, Расмуса Буххольдта и Кристофера Юста Андерсена.

В завершение говорю спасибо всем рецензентам: Бену Макнамара, Билли О'Каллагану, Бонни Малеку, Бренту Хонадель, Чарльзу Ламу, Кристиану Хасельбалху Тудалу, Клайву Харберу, Дэниэлу Васкесу, Дэвиду Тримму, Густаво Филипе Рамосу Гомесу, Джеффу Нойманну, Джоэлу Котарски, Джону Гатри, Джону Норкотту, Картикеяраджану Раджендрану, Киму Кьерсульффу, Луису Му, Марселю ван ден Бринку, Мареку Петаку, Матийсу Аффуртити, Орландо Мендесу Моралесу, Пауло Нуину, Рональду Харингу, Шону Мехаффи, Себастьяну Ларссону, Серджиу Попа, Тану Ви, Тейлору Долезалу, Тому Мэддену, Тайлеру Коваллису и Убальдо Пескаторе — ваши предложения помогли сделать эту книгу лучше.

Об авторе



Кристиан Клаусен имеет степень магистра компьютерных наук и специализируется на языках программирования, в частности на вопросах качества программного обеспечения и написания кода без ошибок. Кристиан был соавтором двух работ по теме качества программного обеспечения, опубликованных в самых престижных журналах. Он участвовал в проекте Coccinelle для исследовательской группы в Париже в роли инженера ПО, преподавал вводный и продвинутый курс программирования на объектно-ориентированных и функциональных языках в двух университетах, а также пять лет проработал консультантом и техническим руководителем.

Иллюстрация на обложке

Изображение на обложке называется *Femme Samojede en habit d'Été*, то есть «Самоедская женщина в летнем наряде». Эта иллюстрация взята из книги, рассказывающей о нарядах народов разных стран, под названием *Costumes Civils Actuels de Tous les Peuples Connus*, подготовленной Жаком Грассе де Сен-Совером (1757–1810) и опубликованной во Франции в 1788 году. Каждая иллюстрация в этом издании тщательно прорисована и раскрашена от руки. Богатое разнообразие экспонатов этой коллекции отчетливо напоминает, насколько велики были культурные различия между городами и регионами мира всего 200 лет назад. Отделенные друг от друга, люди говорили на разных языках и диалектах. На улицах городов или в сельской местности по одеянию человека можно было легко определить, где он живет и какое место в обществе занимает.

С тех пор наша манера одеваться изменилась, и различие нарядов в разных регионах постепенно сходит на нет. Сейчас уже стало сложно отличить друг от друга даже жителей разных континентов, не говоря уже о городах, регионах или странах. Возможно, мы променяли культурное разнообразие на более разнообразную личную жизнь — и уж точно на более разнообразную и ускоренную жизнь технологическую.

Во времена, когда сложно отличить одну книгу о программировании от другой, издательство Manning выделяется в компьютерном бизнесе изобретательностью и инициативностью, оформляя обложки своих книг примерами самобытной жизни регионов двухсотлетней давности, возвращенными к жизни Жаком Грассе де Сен-Совером.

От издательства

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

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.



Рефакторинг рефакторинга

https://t.me/it_boooks

В этой главе

- ✓ Разбор элементов рефакторинга.
- ✓ Внедрение рефакторинга в вашу повседневную работу.
- ✓ Важность безопасности при рефакторинге.
- ✓ Знакомство с общим примером для всей части I книги.

Ни для кого не секрет, что высокое качество кода подразумевает его более эффективное обслуживание, меньше ошибок и хорошее настроение разработчиков. Наиболее распространенный способ добиться от кода высокого качества — это рефакторинг. Тем не менее тот метод, которым обычно обучают рефакторингу — с *запахами кода* и *модульным тестированием*, — воздвигает для начинающих очень высокий порог вхождения. Я же считаю, что после небольшой практики любой сможет безопасно применить простые шаблоны рефакторинга.

В разработке ПО мы распределяем задачи по схеме, изображенной на рис. 1.1. Таким образом обозначаются необходимые для их решения предпосылки: навыки, инструменты, культура или их комбинация. Рефакторинг — это сложная технология, которая располагается в середине. Для его выполнения требуются все упомянутые компоненты.

- *Навыки* — нужны, чтобы выявить плохой код, требующий рефакторинга. Опытные программисты могут сделать это на основе собственных знаний

через запахи кода. Однако сами запахи зачастую не имеют четких границ и определений (требуют опытного анализа), их интерпретация субъективна, в связи с чем их сложно заучить. Поэтому для начинающего разработчика обнаружение запахов кода больше напоминает включение шестого чувства, чем навык.

- *Культура* — стремление обращаться к рефакторингу должно быть заложено в культуру и рабочую повседневность. Во многих случаях такая культура реализуется посредством известного цикла «красный — зеленый — рефакторинг», используемого в разработке через тестирование. Однако разработка через тестирование — это, как мне кажется, еще более сложное искусство. К тому же цикл «красный — зеленый — рефакторинг» не предоставляет удобного способа рефакторить старые (legacy) базы кода.
- *Инструменты* — нам требуется средство, которое бы позволило убедиться в безопасности выполняемых действий. Самый распространенный способ сделать это — автоматизированное тестирование. Но, как я уже сказал, выработка хорошего навыка в тестировании сама по себе представляет сложность.



Рис. 1.1. Навыки, культура и инструменты

В последующих разделах мы углубимся в каждую из этих областей и обрисуем, как можно начать путешествие в мир рефакторинга с упрощенных основ, без тестирования и абстрактных запахов кода. Изучение рефакторинга подобным образом способно быстро продвинуть качество кода начинающих разработчиков,

студентов и программистов-любителей на новый уровень. Технические руководители могут использовать методы из этой книги для обучения рефакторингу своих подчиненных, которые пока не привыкли применять его в производственном процессе.

1.1. ЧТО ТАКОЕ РЕФАКТОРИНГ

На этот вопрос я развернуто отвечу в следующей главе, хотя получить некоторое представление заранее тоже будет полезным. В простейшей форме *рефакторинг* означает «внесение в код изменений без изменения его действий». Для наглядности начнем с примера. В нем мы заменяем выражение локальной переменной.

Листинг 1.1. До

```
return pow(base, exp / 2) * pow(base, exp / 2);
```

Листинг 1.2. После

```
let result = pow(base, exp / 2);  
return result * result;
```

Для рефакторинга может быть много причин:

- ускорение работы кода (как в приведенном примере);
- уменьшение объема кода;
- придание коду универсальности или возможности повторного использования;
- облегчение восприятия и обслуживания кода.

Последняя причина является основной и очень важной, поэтому ее можно сформулировать как «создание хорошего кода».

ОПРЕДЕЛЕНИЕ

Хороший код — это такой код, который не просто корректно выполняет свои задачи, но также удобен для чтения и прост в обслуживании.

Поскольку рефакторинг не должен изменять функционал кода, то в течение книги мы сосредоточимся на его читаемости и обслуживаемости. Эти причины для рефакторинга будут разобраны более подробно в главе 2. В книге мы будем рассматривать только такой рефакторинг, который приводит к хорошему коду. В связи с этим для него будет актуальным следующее определение.

ОПРЕДЕЛЕНИЕ

Рефакторинг — изменение кода с целью повышения его читаемости и обслуживаемости без изменения функциональности.

Должен также сказать, что тип рассматриваемого нами рефакторинга во многом опирается на работу с языком объектно-ориентированного программирования.

Многие представляют себе программирование как написание кода. Однако обычно программисты больше времени проводят за чтением кода и его разбором, чем за написанием. А происходит это потому, что слишком сложна сфера, в которой они работают, и внесение изменений без досконального понимания кода может привести к катастрофическим сбоям.

Итак, первый аргумент для рефакторинга чисто экономический: время программиста стоит дорого, поэтому если мы сделаем нашу базу кода более читаемой, то выиграем время для реализации следующим программистом новых возможностей. Второй аргумент состоит в том, что повышение обслуживаемости кода означает меньшее число ошибок, которые к тому же проще исправить. Третий аргумент гласит, что с хорошей базой кода просто интереснее работать. Когда мы читаем код, то моделируем в своем сознании его действия. Чем больше информации нам приходится удерживать в голове одновременно, тем более утомительным становится этот процесс. Именно поэтому отладка может оказаться ужасной и возникает соблазн заново переделать все полностью.

1.2. НАВЫКИ: ЧТО ТРЕБУЕТ РЕФАКТОРИНГА

Распознать, где именно необходимо применить рефакторинг, — первая трудность. Обычно рефакторингу обучают в сочетании с так называемыми *запахами кода*. Эти запахи или заправки представляют описания того, как может проявлять себя плохое качество кода. Ориентирование по запахам является мощной техникой, но в то же время сами запахи довольно абстрактны, и начинать рефакторинг с этого сложно. Здесь требуется чутье, которое вырабатывается только с опытом.

Книга, которую вы держите в руках, использует другой подход, представляя легко узнаваемые и простые в применении правила для определения требующих рефакторинга фрагментов. Эти правила не только легко использовать, но и просто запомнить. При этом иногда они могут оказаться излишне строгими и потребуют от вас исправить код, который не пахнет. В редких же случаях, наоборот, мы можем последовать этим правилам и после их применения все равно получить код с заправкой.

Как показано на рис. 1.2, соответствие между запахами кода и правилами рефакторинга неидеально. Мои правила не являются необходимым и достаточным условием хорошего кода. Они представляют собой начало пути к выработке профессионального понимания того, какой именно код можно

посчитать хорошим. Предлагаю рассмотреть пример различия между оценками кода по запаху и по правилам этой книги.



Рис. 1.2. Правила и запахи кода

1.2.1. Пример запаха кода

К известным запахам относится, например, такой: функция должна выполнять *одно* действие. Это отличный ориентир, но бывает сложно понять, что именно является этим *одним*. Еще раз взглянем на приведенный выше код: он пахнет? Возможно, так как он делит, вычисляет экспоненту, а затем умножает. Означает ли это, что он совершает три действия? С другой стороны, он просто возвращает одно число и не меняет никакого состояния. Так, может, он все же совершает одно действие?

```
let result = pow(base, exp / 2);  
return result * result;
```

1.2.2. Пример правила

Сравним предыдущий запах со следующим правилом (подробно оно будет разобрано в главе 3): метод никогда не должен содержать больше *пяти строк кода*. Это можно определить на глаз, не задавая лишних вопросов. Правило простое, лаконичное и легко запоминается, особенно ввиду того, что так называется эта книга.

Запомните, что правила из этой книги — своеобразные подстраховки. Как уже говорилось, они не могут гарантировать хороший код во всех ситуациях. В некоторых случаях следование им может оказаться ошибочным. Однако эти

правила оказываются полезными, когда вы не знаете, с чего начать, и обычно иницируют красивый рефакторинг кода.

Заметьте, что все названия правил выражены в абсолютной категоричной форме и в них используются слова вроде «никогда»: это упрощает запоминание. Однако в их подробных описаниях зачастую указываются исключения — ситуации, когда эти правила *не* рекомендуется применять. В описаниях также приводятся значения правил. В начале изучения рефакторинга нужно использовать только абсолютную форму правил. После того как мы их усвоим, можно будет перейти к изучению исключений и лишь затем опираться уже на их базовое назначение — вот тогда мы станем настоящими гуру в написании кода.

1.3. КУЛЬТУРА: КОГДА ПРОВОДИТЬ РЕФАКТОРИНГ

Рефакторинг подобен принятию душа.

Кент Бек

Рефакторинг наиболее эффективен и наиболее экономичен при регулярном проведении. Поэтому, если у вас есть такая возможность, то советую внедрить эту практику в повседневную работу. В большинстве литературных источников предлагается использовать поток «красный — зеленый — рефакторинг». Но, как уже говорилось, этот подход привязывает рефакторинг к разработке через тестирование, а в данной книге нам нужно эти концепции разделить и сосредоточиться именно на рефакторинге, исключая тестирование из подробного рассмотрения. Следовательно, я советую использовать для решения любой задачи программирования более обобщенный рабочий поток, схематично показанный на рис. 1.3.

1. *Исследование.* Нередко в самом начале мы еще не до конца понимаем, что от нас требуется. Бывает, что заказчик сам не полностью осознает, чего именно хочет, или его требования прописаны неоднозначно. Еще бывает, что сразу и не разберешь, выполняемая ли вообще перед тобой задача. Поэтому всегда начинайте с экспериментирования. Реализуйте примерный образец, и у вас уже будет на чем строить диалог с заказчиком, уточняя его требования и ваши возможности.
2. *Специфицирование.* После того как вы утвердили задачу, оформите ее четкие критерии. В идеале это делается в форме автоматизированного тестирования.
3. *Реализация.* Реализуйте код.
4. *Тестирование.* Убедитесь, что код соответствует спецификации из шага 2.

5. *Рефакторинг.* Перед отправкой кода убедитесь, что в дальнейшем другим разработчикам будет удобно с ним работать (к тому же этим разработчиком можете снова стать вы).
6. *Доставка.* Для доставки кода существует множество способов; наиболее распространенный — это создание пула реквеста или отправка его в определенную ветку. Самое главное, чтобы ваш код попал к пользователям. Иначе какой вообще в нем смысл?



Рис. 1.3. Поток разработки

Поскольку мы проводим рефакторинг *на основе правил*, то рабочий поток весьма прост и к нему легко будет приступить. На рис. 1.4 подробно развернут шаг 5: *рефакторинг*.



Рис. 1.4. Подробное представление этапа рефакторинга

Я разработал правила так, чтобы они быстро запоминались и можно было легко обнаружить, где требуется их применение и как его провести. Это значит, что без дополнительных ориентиров несложно идентифицировать метод, нарушающий правило. При этом с каждым правилом связано несколько шаблонов рефакторинга, что помогает понять, как именно применить правило для исправления обнаруженной проблемы. Шаблоны рефакторинга содержат отчетливые пошаговые инструкции, чтобы вы по случайности ничего не сломали. Многие из приводимых в этой книге шаблонов намеренно задевают ошибки компиляции, чтобы вы гарантированно не внесли каких-либо нарушений. После того как мы немного попрактикуем, и правила, и шаблоны рефакторинга станут для вас обыденным делом.

1.3.1. Рефакторинг старых унаследованных систем

Предположим, что мы начинаем работать с обширной legacy-системой. И в этом случае есть грамотный способ внедрить в повседневную работу рефакторинг без необходимости прерывать весь рабочий поток, чтобы сначала полностью отрефакторить базу кода. Достаточно следовать такой чудесной рекомендации.

Сначала сделайте изменение простым, а затем уже внесите это простое изменение.

Кент Бек

Итак, если мы собираемся реализовать нечто новое, то должны начать с рефакторинга, это упрощает добавление дополнительного кода. Просматривается аналогия с подготовкой ингредиентов перед замешиванием теста для выпечки.

1.3.2. Когда рефакторинг делать не нужно

Чаще всего рефакторинг — это здорово, но есть у него и обратные стороны. Он может занимать много времени, особенно если проводить его нерегулярно. Ну а время программиста, как мы знаем, обходится недешево.

Есть три типа баз кода, в которых рефакторинг, скорее всего, себя не оправдает.

- Код, который вы планируете написать, выполнить один раз и удалить. В сообществе экстремального программирования (Extreme Programming) это называется «спайк» (*spike*).
- Код, который находится на обслуживании в преддверии завершения его эксплуатации.

- Код со строгими требованиями к производительности, например встраиваемая система или передовой физический движок в игре.

Во всех других случаях вложенные в рефакторинг силы и время определенно себя оправдают.

1.4. ИНСТРУМЕНТЫ: КАК ПРОВОДИТЬ РЕФАКТОРИНГ (БЕЗОПАСНО)

Мне, как и всем, нравятся автоматизированные тесты. Однако выработка хорошего навыка тестирования сама по себе представляется сложной. Поэтому, если вы уже умеете выполнять такие тесты, то можете смело использовать эту технику при прочтении книги. Если же нет, то ничего страшного.

Автоматизированные тесты в разработке ПО сродни тормозам у автомобилей. Мы оборудуем авто тормозной системой не потому, что хотим ехать медленно, а для того, чтобы быстрая езда была безопасной. То же верно и для программного обеспечения: автоматизированные тесты позволяют нам двигаться быстро и при этом безопасно. В этой книге мы изучаем абсолютно новый навык, значит, спешить нежелательно.

Вместо этого я предлагаю опереться на другие инструменты, например:

- детальные, поэтапные, структурированные шаблоны рефакторинга, подобные четким рецептам;
- контроль версий;
- компилятор.

На мой взгляд, если шаблоны рефакторинга спроектировать тщательно и выполнять мелкими шажками, то можно провести рефакторинг, ничего не нарушив. Это особенно актуально в случаях, когда наша IDE (интегрированная среда программирования, Integrated Development Environment) способна выполнять рефакторинг за нас.

Чтобы компенсировать отсутствие тестирования, в этой книге мы будем вылавливать многие распространенные ошибки с помощью компилятора и типов. Но я все равно рекомендую вам периодически открывать приложение, над которым вы работаете, и проверять его функциональность. Когда мы это проверим или компилятор не сообщит об ошибках, можно делать для себя коммит. Таким образом, если в дальнейшем работоспособность приложения нарушится и мы не сможем это с ходу исправить, можно будет откатиться к последнему коммиту.

Работая с реальными системами без автоматизированных тестов, мы все равно можем выполнять рефакторинг, но при этом наша уверенность должна будет на что-то опираться. В качестве такой опоры можно использовать для проведения рефакторинга IDE, выполнять ручные тесты, продвигаться очень мелкими шажками и т. п. Однако если учесть, что для этого необходимо затратить дополнительное время, то наверняка окажется целесообразнее использовать именно автоматизированное тестирование.

1.5. ИНСТРУМЕНТЫ, НЕОБХОДИМЫЕ ДЛЯ НАЧАЛА

Как я уже сказал, способы рефакторинга из этой книги требуют использования объектно-ориентированного языка программирования (ООП-языка). Это первое, что вам нужно, чтобы читать и понимать ее.

И написание кода, и рефакторинг представляют собой деятельность, которую мы осуществляем непосредственно на компьютере, своими руками. Следовательно, и осваивать эти навыки лучше всего, работая на компьютере, следуя приводимым примерам и экспериментируя. Для этого вам потребуются инструменты, описанные далее. Инструкции по их установке можно найти в приложении в конце книги.

1.5.1. Язык программирования: TypeScript

Все примеры кода книги написаны на TypeScript. Этот язык я выбрал по ряду причин. Самое главное — это то, что он выглядит и ощущается родственным многим распространенным языкам — Java, C#, C++ и JavaScript. Это позволит читателям, знакомым с любым из них, читать и понимать листинги без каких-либо проблем. Кроме того, TypeScript предоставляет способ перейти от полностью «не объектно-ориентированного» кода (то есть кода без классов) к его объектно-ориентированной альтернативе.

ПРИМЕЧАНИЕ

С целью оптимизации использования пространства в печатной версии книги в ней используется стиль программирования, в котором исключаются переводы строки, но сохраняется читаемость. Я не призываю вас следовать такому же стилю, если только вы по случайности тоже не пишете книгу с большим количеством TS-кода. По этой же причине отступы и скобки иногда имеют различное форматирование в книге и в коде проекта.

На случай, если TypeScript вам не знаком, по мере появления каких-то его нюансов я буду пояснять их вот в таких вставках.

В TYPESCRIPT...

Для проверки равенства используется оператор тождественности (===), потому что его действие больше походит на ожидаемый эффект равенства, чем действие нестрогого равенства (==). Рассмотрим пример:

```
0 == "" верно.
```

```
0 === "" ложно.
```

Несмотря на то что примеры приводятся на TypeScript, все шаблоны рефакторинга и правила являются общими и применимы к любому ООП-языку. В редких случаях TypeScript как таковой проявляет собственные нетипичные для других языков особенности, помогает или мешает нам. Эти случаи отмечены явно, и я объясняю, что делать в подобных ситуациях в других популярных языках.

1.5.2. Редактор: Visual Studio Code

Я не предполагаю, что вы будете использовать определенную программу-редактор. Тем не менее если у вас нет личных предпочтений, то советую выбрать Visual Studio Code. Он отлично работает с TypeScript, а также поддерживает работу `tsc -w` в фоновом режиме, автоматически выполняя компиляцию, чтобы мы не забыли о ней.

ПРИМЕЧАНИЕ

Visual Studio Code совершенно не то же самое, что *Visual Studio*.

1.5.3. Контроль версий: Git

Несмотря на то что для следования материалу книги использовать контроль версий не обязательно, я настоятельно его рекомендую, поскольку так будет намного проще отменить какие-либо действия, если вы запутаетесь в процессе.

СБРОС К ЭТАЛОННОМУ РЕШЕНИЮ

Вы можете в любой момент вернуть код к тому виду, в каком он был в начале основного раздела, с помощью, например, такой команды:

```
git reset --hard section-2.1
```

Предупреждение: при этом все внесенные вами изменения будут потеряны.

1.6. ОБЩИЙ ПРИМЕР: 2D-ГОЛОВОЛОМКА

В завершение опишу сам способ, которым буду обучать вас всем этим прекрасным правилам и крутым шаблонам рефакторинга. Книга построена вокруг одного общего примера: 2D-головоломки с перемещением блоков, она похожа на классическую игру Boulder Dash (рис. 1.5).

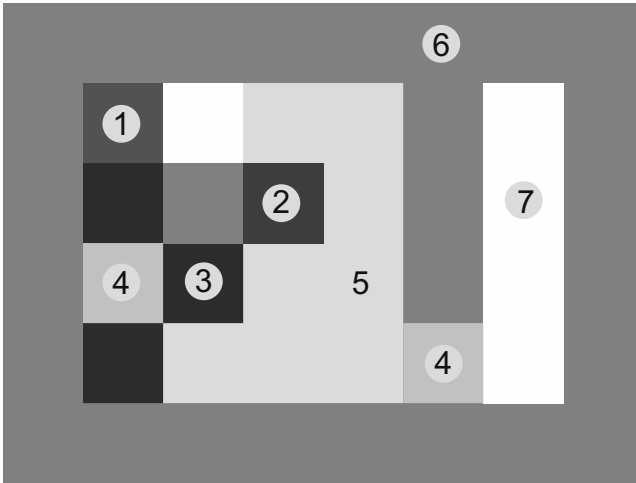


Рис. 1.5. Снимок экрана игры «из коробки»

Это означает, что на всю часть I книги у нас будет единая массивная база кода. Использование всего одного примера сэкономит время, потому что не придется в каждой главе знакомиться с новым.

Сам пример написан в реалистичном стиле, похожем на используемый в индустрии. Простым это упражнение назвать нельзя, если только вы уже не владеете навыками, описываемыми в данной книге. Код изначально подчиняется принципам *DRY* («Не повторяйся») и *KISS* («Не усложняй»). Но даже в таком виде он не приятнее сухого поцелуя¹.

Я выбрал компьютерную игру, потому что при ручном тестировании так будет проще заметить некорректное поведение кода на основе интуитивного понимания того, как должен происходить сам процесс игры. К тому же так тестирование окажется веселее по сравнению с просмотриванием, например, логов финансовых систем.

Пользователь управляет клеткой игрока с помощью клавиш-стрелок. Цель игры — передвинуть ящик (на рис. 1.5 обозначен как 2) в нижний правый угол.

¹ Авторская игра слов, так как в английском *dry* — это «сухой», а *kiss* — это «поцелуй». — *Здесь и далее примеч. пер.*

Несмотря на то что печатная версия книги черно-белая, на деле элементы представлены разными цветами.

1. Красная клетка — это игрок.
2. Коричневая клетка — это ящик.
3. Синие — это камни.
4. Желтые представляют ключи или замки — с этим мы разберемся позже.
5. Зеленоватые клетки называются *флаксами*.
6. Серые — это стены.
7. Белые — это воздух (пустота).

Если ящик или камень ничем не поддерживаются, то они падают. Игрок может передвинуть один камень или ящик за один ход при условии, что этот предмет не огражден и в результате не упадет. Путь между ящиком и нижним правым углом изначально закрыт на замок, поэтому игроку нужно сначала найти ключ. Флакс можно «съесть» (удалить), наступив на него.

Теперь самое время скачать игру и поэкспериментировать.

1. Перейдите на компьютере в расположение, куда хотите сохранить игру, и откройте консоль:
 - 1) команда `git clone https://github.com/thedrlambda/five-lines` скачает ее исходный код;
 - 2) команда `tsc -w` будет компилировать TypeScript в JavaScript при каждом его изменении.
2. Откройте `index.html` в браузере.

Здесь у вас есть возможность через код изменять уровни, так что смело экспериментируйте с созданием собственных карт путем обновления массива в переменной `map` (пример можете найти в приложении).
3. Откройте каталог в Visual Studio Code.
4. Выберите `Terminal`, затем `New Terminal`.
5. Выполните `tsc -w`.
6. Теперь TypeScript будет компилировать изменения в фоновом режиме, и терминал можно закрыть.
7. После внесения каждого изменения подождите секунду, пока TypeScript выполнит компиляцию, а затем обновите содержимое окна браузера.

Такую же процедуру вы будете использовать при написании кода в соответствии с примерами части I книги. Но прежде, чем к этому перейти, в следующем разделе мы заложим более детальную основу рефакторинга.

1.6.1. Практика ведет к совершенству: вторая база кода

Я свято верю в силу практики и поэтому создал еще один проект, для которого решения не привел. Этот проект вы можете использовать при перечитывании книги, для испытания собственных навыков или в качестве упражнения для студентов, если вы преподаватель. Он представляет собой 2D-игру в жанре экшена. В обеих базах кода (основного примера и дополнительного проекта) используются одинаковые стиль, структура, элементы. При этом их рефакторинг подразумевает выполнение одних и тех же шагов. Несмотря на то что вторая база кода более продвинутая, внимательно следуя правилам и шаблонам рефакторинга, вы должны добиться желаемого результата и в ней. Для скачивания второго проекта выполните все те же шаги, но со следующим URL: <https://github.com/thedrlambda/bomb-guy>.

1.7. ПРИМЕЧАНИЕ ПО РЕАЛЬНЫМ ПРОГРАММАМ

Важно еще раз повторить, что основной задачей этой книги является введение в работу рефакторинга. Эта задача не подразумевает указание конкретных правил, которые вы могли бы применять в продакшене в той или иной ситуации. Для их использования сначала нужно выучить названия правил и повторять их. Когда для вас это станет простым, изучите описания, отражающие исключения из правил. Наконец, используйте все это для выработки понимания лежащих в основе правил запахов кода. Предстоящий вам путь схематично показан на рис. 1.6.

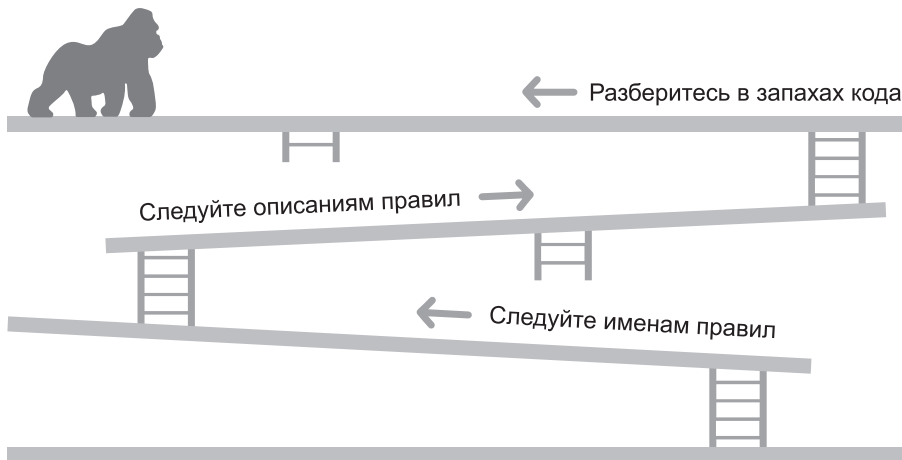


Рис. 1.6. Как использовать правила

Необходимость именно такого постижения смысла правил и неформальный подход к рефакторингу объясняет невозможность создания программы автоматического рефакторинга. (Хотя можно разработать плагин, который на основе правил будет выделять *предположительно проблематичные* области кода.) Цель же самих правил — выработать понимание. Если коротко: следуйте правилам, пока сами не начнете понимать, как лучше.

Имейте также в виду, что благодаря концентрации исключительно на изучении рефакторинга и наличию безопасной среды при чтении книги можно обойтись без автоматизированных тестов, но вряд ли это пройдет при работе в реальных системах. Здесь мы поступаем так, потому что изучать рефакторинг и автоматизированное тестирование гораздо эффективнее по отдельности.

РЕЗЮМЕ

- Проведение рефакторинга требует наличия трех предпосылок: *навыков* для определения фрагментов кода, требующих улучшения, *культуры* для понимания, когда нужно это делать, и *инструментов*, позволяющих понять, как именно нужно это делать.
- Условно для определения того, что именно нужно рефакторить, используются запахи кода. Начинаящим программистам сложно разобраться в них, не имея опыта, потому что они слишком абстрактны. В этой книге приводятся конкретные правила, способные на время обучения заменить практику определения запахов кода. Эти правила имеют три уровня абстракции: конкретные названия, собственно описания, касающиеся смысловых нюансов в виде исключений, и, наконец, сама суть запахов.
- Я верю, что для снижения порога вхождения в автоматизированное тестирование и рефакторинг нужно изучать эти дисциплины по отдельности. Вместо автоматизированного тестирования мы задействуем компилятор, контроль версий и ручное тестирование.
- Рабочий поток рефакторинга обычно связан с разработкой через тестирование в цикле «красный — зеленый — рефакторинг». Но это опять же подразумевает зависимость от автоматизированных тестов. Вместо этого я предлагаю использовать рабочий поток из шести шагов (исследование, специфицирование, реализация, тестирование, рефакторинг, доставка) как для нового кода, так и для рефакторинга непосредственно перед изменением старого.

На протяжении части I книги мы будем изменять исходный код 2D-головоломки с использованием Visual Studio Code, TypeScript и Git.

2

Суть рефакторинга

https://t.me/it_boooks

В этой главе

- ✓ Читаемость как средство для передачи намерения.
- ✓ Локализация инвариантов для повышения обслуживаемости.
- ✓ Ускорение разработки за счет техники внесения изменений путем добавления.
- ✓ Внедрение рефакторинга в повседневную работу.

В предыдущей главе мы рассмотрели элементы, участвующие в рефакторинге. В текущей же мы углубимся в технические детали, чтобы сформировать устойчивое представление о рефакторинге и уяснить, почему он важен с технической точки зрения.

2.1. УЛУЧШЕНИЕ ЧИТАЕМОСТИ И ОБСЛУЖИВАЕМОСТИ

Начнем с повторения определения рефакторинга, которое используем в этой книге: «*Рефакторинг* — это улучшение кода без изменения его функциональности». Разберем подробнее каждую из двух частей этого определения: «*улучшение кода*» и «*без изменения его функциональности*».

2.1.1. Улучшение кода

Мы уже видели, что более качественный код выигрывает в читаемости и обслуживаемости, а также поняли, почему это важно. Но тогда мы не разбирали, что именно означает читаемость и обслуживаемость и как на них влияет рефакторинг.

Читаемость

Читаемость — это способность кода передать заложенное в него намерение. То есть если мы предполагаем, что код работает должным образом, то при чтении его самого должно быть легко понять, что именно он делает. Для передачи намерения в коде есть много способов: использование и следование соглашениям; написание комментариев; именование переменных, методов, классов и файлов; использование пробелов и отступов.

Относительно друг друга эти способы могут быть более или менее эффективны, и мы рассмотрим их чуть позже. Сейчас же взглянем на простую вымышленную функцию, код которой нарушает все принципы передачи намерения, упомянутые выше. Справа от нее приводится тот же метод, но его код уже подчиняется этим принципам. Первую версию читать сложно, зато вторая воспринимается намного легче.

Листинг 2.1. Пример реально нечитаемого кода

```
function checkValue(str: boolean) {
  // Проверка значения
  if (str !== false)
    // return
    return true;
  else; // в ином случае
    return str;
}
```

← Плохое имя метода: булев параметр назван str

← Комментарий просто повторяет код

← Точка с запятой (;), которую легко упустить, и тривиальный комментарий

← Двойное отрицание трудно читать

← Комментарий просто повторяет имя

← Сбивающий с толку отступ; и в этом месте str может быть лишь false, поэтому будет понятнее просто указать это

Листинг 2.2. Тот же код, но уже в читаемом виде

```
function isTrue(bool: boolean) {
  if (bool)
    return true;
  else
    return false;
}
```

После подобной чистки кода очевидно, что можно просто написать следующее.

Листинг 2.3. Тот же код в упрощенной форме

```
function isTrue(bool: boolean) {
  return bool;
}
```

Обслуживаемость

Когда нужно изменить функциональность, например исправить баг или добавить новую возможность, мы зачастую начинаем с анализа контекста, в который, на наш взгляд, должен вписаться новый код. Мы пытаемся понять, что код делает в своем текущем виде и как можно безопасно, быстро и легко изменить его для реализации нашей новой цели. *Обслуживаемость* — это выражение того, как долго нам приходится с этим разбираться.

При этом очевидно, что чем больше кода нам нужно прочесть и проанализировать, тем больше это требует времени и тем выше вероятность упустить что-либо. Следовательно, обслуживаемость тесно связана с риском, который возникает при каждом внесении изменений.

Многие программисты на всех уровнях очень щепетильно и осторожно относятся к фазе анализа. Всем доводилось время от времени по случайности упускать что-либо и наблюдать последствия. Осторожность также означает, что если мы не можем с ходу определить, является ли нечто важным, то чаще склоняемся к более безопасному варианту. Длительная фаза анализа считается признаком плохой обслуживаемости кода и указывает на необходимость его доработки.

В некоторых системах при изменении чего-либо в одном месте вдруг ломается что-то, казалось бы, совсем с ним не связанное. Представьте себе онлайн-магазин, в котором внесение изменений в механизм рекомендаций нарушает работу платежной подсистемы. Такие системы называются *хрупкими*.

Причиной этой хрупкости обычно является *глобальное состояние*. Здесь *глобальное* означает «находящееся вне рассматриваемой области и категории». С точки зрения метода глобальными являются поля. Принцип *состояния*, в свою очередь, чуть более абстрактен. *Состояние* — это «все, что может изменяться в процессе выполнения программы». Сюда входят все переменные, содержимое базы данных, файлы на жестком диске и само оборудование. (Технически даже намерение пользователя и вся реальность в некотором смысле представляют собой состояние, просто для наших целей они неважны.)

Для простоты при работе можно определять глобальное состояние по фигурным скобкам: { ... }. Все находящееся вне этих скобок считается глобальным состоянием по отношению к тому, что находится внутри них.

Проблема с глобальным состоянием в том, что мы зачастую ассоциируем свойства с данными, считая их объектами исключительно в нашем собственном ведении. Но фактически, если данные глобальны, они оказываются доступными для изменения другими людьми, которые могут связать с ними другие свойства, тем самым ненамеренно нарушив наши. Свойства, которые мы не проверяем в коде явно (или проверяем только с помощью утверждений), называются

инвариантами. Утверждения «Это число никогда не будет отрицательным» и «Этот файл точно существует» являются примерами инвариантов. К сожалению, практически нереально гарантировать, что инварианты останутся валидными, особенно ввиду того, что по ходу изменения системы программисты склонны о них забывать, плюс в команду приходят новые люди.

КАК НАРУШАЮТСЯ НЕЛОКАЛЬНЫЕ ИНВАРИАНТЫ

Представим, что работаем над приложением для продовольственного магазина. В нем продаются фрукты и овощи, значит, в нашей системе все элементы имеют свойство `daysUntilExpiry`¹. Мы реализуем функционал, который срабатывает каждый день, вычитает один из `daysUntilExpiry` и автоматически удаляет товар, когда соответствующее значение достигает нуля. Теперь у нас есть инвариант, что `daysUntilExpiry` всегда положительно.

Мы также хотим, чтобы в нашей системе присутствовало свойство `urgency`, отражающее степень важности продажи каждого товара. Повышенное значение `urgency` должны иметь товары с более высоким значением `value`², а также с более низким `daysUntilExpiry`. Следовательно, мы реализуем инструкцию `urgency = value / daysUntilExpiry`. Она не может дать сбой, ведь мы *знаем*, что `daysUntilExpiry` всегда положительно.

Два года спустя нас просят обновить систему, потому что магазин начал продавать лампы накаливания. Мы быстренько добавляем в ассортимент эти лампы, но у них нет срока годности (`daysUntilExpiry`), и мы помним функционал, который вычитает дни, удаляя товары, когда их значение `daysUntilExpiry` достигает нуля, но при этом совершенно забыли про инвариант. В итоге мы решаем изначально установить `daysUntilExpiry` на нулевое значение. Таким образом, когда функция вычитет единицу, оно уже не будет равно нулю.

Так мы нарушили инвариант, что привело к сбою системы при попытке вычислить `urgency` для любой лампы накаливания: `Error: Division by zero`.

Обслуживаемость можно повысить путем явной проверки свойств, тем самым удалив инварианты. Однако внесение подобной доработки приведет к изменению функциональности кода, к чему, как мы увидим в следующем разделе, рефакторинг приводить не должен. Напротив, рефакторинг стремится повысить обслуживаемость путем перемещения инвариантов туда, где их будет проще заметить. Это называется *локализацией инвариантов*: элементы, которые изменяются вместе, должны располагаться вместе.

¹ Дней до конца срока годности.

² Ценности.

2.1.2. Обслуживание кода... без изменения его функциональности

Вопрос «Что делает этот код?» весьма интересен, хотя и несколько метафизичен. Инстинктивно мы должны воспринимать программный код как *черный ящик* и принять как аксиому, что мы можем менять все в его конструкции, но только если это не будет проявлять себя вне самого ящика. Если мы помещаем на его входе значение, то должны получить на выходе после рефакторинга (изменения его внутренней конструкции) тот же результат, что и до его проведения, даже если это будет ошибка.

Такой принцип работает практически всегда, за одним исключением: допускается изменять быстродействие кода. На это есть много причин. Во-первых, в большинстве систем быстродействие менее ценно, чем читаемость и обслуживаемость. Во-вторых, если быстродействие важно, оно должно обрабатываться отдельным от рефакторинга этапом с помощью средств профилирования или с привлечением соответствующих экспертов. Оптимизацию мы будем более подробно изучать в главе 12.

При выполнении рефакторинга нам нужно учесть границы нашего черного ящика. Сколько кода мы планируем изменить? Чем больше кода мы включим в конструкцию ящика, тем больше элементов сможем изменить. Определиться с этим особенно важно при работе в команде, потому что, если кто-то внесет изменения в код, который мы рефакторим, могут возникнуть конфликты слияния. На деле нам нужно *резервировать* код, который находится в рефакторинге, чтобы никто больше его не изменил одновременно с нами. Чем меньше кода мы зарезервируем, тем ниже риск появления конфликтов изменений. Таким образом, определение подходящей области для рефакторинга становится сложным и важным актом поиска баланса.

Подытоживая, обозначим три кита рефакторинга.

1. Повышение читаемости за счет передачи намерения.
2. Повышение обслуживаемости путем локализации инвариантов.
3. Выполнение пунктов 1 и 2 без влияния на код вне рассматриваемой области.

2.2. ВЫРАБОТКА СКОРОСТИ, ГИБКОСТИ И СТАБИЛЬНОСТИ

Я уже упоминал преимущества работы с грамотной и чистой базой кода: это повышает продуктивность, снижает количество ошибок и в целом намного приятнее. Хорошая обслуживаемость, в свою очередь, добавляет еще несколько плюшек, о которых мы и поговорим в этом разделе.

Существует несколько уровней шаблонов рефакторинга, начиная от конкретных и локальных (например, смена имени переменной) до абстрактных и глобальных (например, введение шаблонов проектирования). Я согласен с тем, что именование переменных может повлиять на читаемость, но все же наиболее значительно влияют на качество кода его архитектурные изменения. В этой книге ближе всего к уровню внутреннего рефакторинга методов мы подойдем при изучении их грамотного именования.

2.2.1. Выбирайте композицию вместо наследования

Тот факт, что нелокальные инварианты сложно обслуживать, уже не новость. Группа инженеров под очаровательным именем «Банда четырех» (Gang of Four) (Эрих Гамма, Ричард Хелм, Ральф Джонсон и Джон Влиссидес) в далеком 1999 году опубликовала книгу *Design Patterns*¹. В те давние времена они выступали против случайного введения нелокальных инвариантов: наследования. Самое известное их высказывание даже содержит рекомендацию, как этого избежать: «Выбирайте композицию объектов вместо наследования».

Этот совет является краеугольным камнем всех построений книги, которую вы сейчас читаете, а большинство рассматриваемых в ней шаблонов рефакторинга и правил предназначены именно для того, чтобы помочь создать *композицию объектов*, то есть набор объектов, содержащих ссылки на другие объекты. Возьмем, к примеру, небольшую библиотеку птиц (орнитологические детали здесь неважны). Слева в ней используется наследование, а справа — композиция.

Листинг 2.4. Использование наследования

```
interface Bird {
    hasBeak(): boolean;
    canFly(): boolean;
}
class CommonBird implements Bird {
    hasBeak() { return true; }
    canFly() { return true; }
}
class Penguin extends CommonBird {
    canFly() { return false; }
}
```

← Наследование

Листинг 2.5. Использование композиции

```
interface Bird {
    hasBeak(): boolean;
    canFly(): boolean;
}
class CommonBird implements Bird {
    hasBeak() { return true; }
    canFly() { return true; }
}
class Penguin implements Bird {
    private bird = new CommonBird();
    hasBeak() { return bird.hasBeak(); }
    canFly() { return false; }
}
```

← Композиция

← Необходимо вручную перенаправить вызовы

¹ Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2021.

В этой книге мы будем еще много говорить о преимуществах правого варианта. Но для того, чтобы получить уже сейчас начальное понимание, представьте, что необходимо добавить новый метод с названием `canSwim` в `Bird`. В обоих случаях мы добавляем его в `CommonBird`.

Листинг 2.6. Использование наследования

```
class CommonBird implements Bird {
    // ...
    canSwim() { return false; }
}
```

В листинге 2.5 в примере композиции мы все равно получаем ошибку компиляции в `Penguin`, потому что он не реализует новый метод `canSwim`. В итоге приходится вручную добавить этот метод и решить, может пингвин плавать или нет. В случае, когда нам просто нужно, чтобы `Penguin` вел себя аналогично другим птицам, это несложно реализовать, например, с помощью `hasBeak`. В отличие от поведения композиции, пример с наследованием без всякого сигнала для нас заключает, что пингвин плавать не может, и нам самим в итоге нужно не забыть переопределить `canSwim`. А человеческая память уже устойчиво зарекомендовала себя как ненадежный и хрупкий инструмент, особенно когда наше внимание поглощено новым функционалом, над которым мы работаем.

Гибкость

Система, построенная на основе композиции, позволяет нам комбинировать и многократно использовать код намного более дискретно и детально, чем в других случаях. Работа с системами, опирающимися на композицию, подобна игре в LEGO. Когда все собрано в единый комплекс, становится удивительно легко переставлять фрагменты или создавать новые композиции путем совмещения существующих составных частей. Причем важность подобной гибкости в нашем понимании намного возрастает, когда мы осознали, что большинство систем в конечном итоге используются так, как их изначальные разработчики и не предполагали.

2.2.2. Изменение кода путем добавления, а не изменения

Самым же большим преимуществом композиции, пожалуй, является то, что она позволяет вносить *изменения путем добавления*. Это подразумевает возможность добавления или изменения определенной функциональности без влияния на остальные части ПО — в некоторых случаях не приходится даже менять имеющийся код. Далее в книге мы еще вернемся к разбору того, как это делается. Здесь же мы рассмотрим некоторые последствия внесения изменений путем добавления. Это свойство программных систем также иногда называют

принципом открытости/закрытости, подразумевающим, что системы должны быть открыты для расширения (дополнения), но закрыты для изменения.

Скорость программирования

Как уже отмечалось, одним из первых действий при реализации чего-то нового или исправления бага является анализ окружающего контекста, позволяющий избежать его нарушения. Однако если мы сможем вносить изменения, не затрагивая никакой другой код, то это позволит сэкономить затрачиваемое на анализ время.

Конечно, если просто добавлять код, то база кода быстро разрастется, что тоже может стать проблемой. Нужно уделить дополнительное внимание тому, какой код используется, а какой — нет. Неиспользуемый код следует удалять как можно быстрее. К этой идее мы еще вернемся ниже.

Стабильность

Если следовать технике изменения путем добавления, у нас всегда будет возможность сохранить существующий код. Станет легко реализовывать функциональность и откатываться к старой ее версии, если новая не сработает. Таким образом, мы сможем гарантировать, что никогда не внесем новых ошибок в добавляемый код. А если добавить к этому еще и уменьшение числа ошибок, связанных с локализацией инвариантов, то на выходе будут получаться гораздо более стабильные системы.

2.3. РЕФАКТОРИНГ И ПОВСЕДНЕВНАЯ РАБОТА

Еще во введении я говорил, что рефакторинг должен быть частью ежедневной работы любого программиста. Поставляя неотрефакторенный код, мы обрекаем следующего программиста, который будет работать с ним, на дополнительное расходование драгоценного времени. Более того, с отрицательными факторами, описанными до этого момента, в плохой архитектуре ПО оказывается связан так называемый принцип займа, обычно его называют *техническим долгом*. Этот принцип мы более подробно разберем в главе 9.

Я уже отмечал два рекомендуемых варианта ежедневного рефакторинга.

- В legacy-системах начинать с рефакторинга и только затем приступать к внесению изменений, следуя стандартному рабочему потоку.
- Проводить рефакторинг после внесения любых изменений в код.

Что касается обязательного проведения рефакторинга перед отправкой кода, то про него иногда говорят так.

Всегда оставляйте за собой место в лучшем виде, чем вы его нашли.

Правило бойскаута

2.3.1. Рефакторинг как метод для освоения

Завершая характеристику рефакторинга, скажу, что, как и многие другие техники, он требует длительного освоения. Хотя в конечном итоге навык рефакторинга доводится до автоматизма. Получив опыт в обнаружении более качественного кода, прочувствовав его преимущества, программисты постепенно совершенствуют свое мастерство в написании и понимании программ. В свою очередь, рост стабильности ПО находит отражение, например, в том, что к стабильным кодам все чаще обращаются (явление роста частоты развертывания — *deployment frequency*), а это обычно еще больше повышает стабильность. За счет гибкости становится возможным выстраивать системы управления конфигурацией или системы с переключением функциональности, обслуживание которых без этой самой гибкости просто невозможно.

Рефакторинг предполагает способ анализа и понимания кода, отличающийся от всего, к чему мы привыкли. Ведь конечная цель уникальна. Иногда нам достается код, на понимание которого может уйти от нескольких часов до нескольких дней. Следующая глава демонстрирует, что рефакторинг позволяет нам улучшать код даже без его понимания. Таким образом, в процессе работы с кодом мы можем перерабатывать небольшие его фрагменты формально, пока окончательный результат не станет легким для восприятия.

РЕФАКТОРИНГ КАК ВСТУПИТЕЛЬНОЕ ЗАДАНИЕ

Рефакторинг часто используют в качестве вводного задания для новых членов команды, чтобы они могли работать с кодом и обучаться в безопасной среде без необходимости сразу иметь дело с клиентами. Это, безусловно, хорошая практика, но возможна она, только если пренебрегать ежедневной надлежащей проверкой кода, что я определенно не одобряю.

Я уже много говорил о множестве преимуществ как в изучении, так и в применении рефакторинга. Надеюсь, что мне удалось вас вдохновить и вы готовы совершить со мной это путешествие в мир рефакторинга!

2.4. ОПРЕДЕЛЕНИЕ «ОБЛАСТИ» В КОНТЕКСТЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Программное обеспечение представляет собой модель конкретных аспектов реальной жизни, будь то код для автоматизации процесса, отслеживание или моделирование (симуляция) событий либо что-то другое. ПО всегда связано с неким компонентом реальной жизни. Этот компонент реальности и называется *областью* применения программного обеспечения. Как правило, области включают в себя пользователей и экспертов, собственный язык и даже культуру.

В части I книги рассматриваемой областью станет двухмерная игра-головоломка. Пользователи в ней выступают игроками, а эксперты области представлены самой игрой или дизайнерами уровней. Мы уже видели, что игра использует собственный язык, вводя такие слова, как «флакс», который игрок может «съесть». Наконец, видеоигры подразумевают определенную культуру. Она выражается в форме ожиданий того, как с ними можно взаимодействовать. Например, люди, знакомые с видеоиграми, по умолчанию принимают тот факт, что некоторые игровые объекты подвержены гравитации (камни и ящики), а некоторые — нет (ключи и сам игрок).

При разработке ПО зачастую приходится тесно сотрудничать с экспертами области, что означает необходимость изучения их языка и культуры. Языки программирования не допускают двусмысленности. Следовательно, иногда нам нужно исследовать и определять пограничные случаи, неизвестные даже экспертам. Таким образом, программирование в первую очередь основывается на постоянном обучении и коммуникации.

РЕЗЮМЕ

- Провести рефакторинг кода означает более четко выразить через листинг его назначение и локализовать инварианты, не изменив при этом изначальной функциональности кода.
- Выбор в пользу композиции вместо наследования открывает возможность вносить изменения путем добавления, что повышает скорость разработки ПО, увеличивает гибкость и стабильность кода.
- Чтобы предотвратить накопление технического долга, рефакторинг следует сделать частью ежедневной работы.
- Практикуя рефакторинг, мы получаем возможность взглянуть на код с точки зрения уникальной перспективы и найти лучшие решения.

Часть I

Учимся на рефакторинге компьютерной игры

В части I мы пройдемся по, казалось бы, и без того неплохой базе кода и шаг за шагом ее улучшим. В ходе этого процесса мы познакомимся с набором правил и создадим небольшой каталог мощных шаблонов рефакторинга.

Код мы будем улучшать в четыре этапа, для описания каждого из которых выделена своя глава: «Разбивка длинных функций», «Пусть код типа работает», «Совмещение схожего кода» и, наконец, «Защита данных». Каждая глава строится на основе предыдущей, поэтому некоторые изменения будут временными. Если код или инструкция кажутся вам странными или смотрятся некрасиво, то проявите терпение — скорее всего, это будет исправлено.

Не паникуйте.

*Дуглас Адамс.
Автостопом по галактике*

3

Разбивка длинных функций

В этой главе

- ✓ Определение слишком длинных методов с помощью правила «Пять строк».
- ✓ Работа с кодом без изучения его специфики.
- ✓ Разбивка длинных методов с помощью шаблона «Извлечение метода».
- ✓ Уравновешивание уровней абстракции с помощью правила «Вызов или передача».
- ✓ Изолирование инструкций `if`, в которых `if` находится только в начале.

Код запросто может стать запутанным и сбивающим с толку, даже если следовать принципам DRY и KISS. Причинами подобного беспорядка обычно выступают:

- методы, выполняющие несколько разных действий;
- использование низкоуровневых элементарных операций (обращение к массивам, арифметика и т. д.);
- недостаток понятного человеку текста, например комментариев, грамотных имен переменных и методов.

К сожалению, простого знания об этих причинах недостаточно для конкретного определения проблемных мест в коде, не говоря уже о понимании, как их исправить.

В этой главе я предлагаю четкий способ определения методов, которые с большой вероятностью имеют слишком много ответственностей. В качестве примера рассмотрим конкретный метод в 2D-головоломке, который выполняет слишком много: `draw`. Я покажу вам структурированный и безопасный способ улучшить этот метод с попутным удалением комментариев. Затем мы обобщим этот процесс до построения многократно используемого шаблона рефакторинга: «Извлечение метода» (3.2.1). На примере того же метода `draw` мы научимся выявлять и решать также с помощью шаблона «Извлечение метода» еще одну проблему — смешивание различных уровней абстракции. В ходе этого процесса нам станет ясно, в чем состоит хороший тон при именовании методов.

Завершив работу с `draw`, перейдем еще к одному примеру — методу `update` — и повторим тот же процесс, уточнив, как можно работать с кодом, не вникая в его детали. Этот пример научит вас определять различные признаки наличия у метода излишних задач. А с помощью «Извлечения метода» мы прочувствуем, как повысить его читаемость путем изменения имен переменных.

Стоит заметить, что нередко мы проводим различие между методами (определяемыми в объектах) и функциями (статичными или находящимися вне классов). Это может несколько запутать. К счастью, TypeScript здесь нам помогает, так как при определении функции в нем необходимо использовать ключевое слово `function`, чего не нужно делать при определении метода. Если вас все равно сбивает это с толку, то можете просто заменить *функцию* *методом*, поскольку все правила и виды рефакторинга равно применимы к ним обоим.

Предполагая, что вы уже настроили необходимые инструменты и скачали код, как описывалось в приложении, перейдем к содержимому файла `index.ts`. Напомню, что вы всегда можете сверить свой код с любым базовым разделом книги, выполнив, к примеру, `git diff section-3.1`. Если же вы вдруг в коде запутаетесь, то можете использовать, например, `git reset --hard section-3.1`, чтобы вернуть его к форме из начала последнего основного раздела. Итак, мы видим перед собой код, появляется желание улучшить его качество, но одновременно возникает вопрос: с чего начать?

3.1. ОПРЕДЕЛЯЕМ ПЕРВОЕ ПРАВИЛО: ПОЧЕМУ ПЯТЬ СТРОК?

Чтобы ответить на этот вопрос, мы начнем с ознакомления с основным правилом всей книги — «Пять строк». Эта простая аксиома утверждает, что никакой метод не должен содержать более пяти строк кода. В данной книге пять строк — это конечная цель, потому что соблюдение этого правила уже является огромным улучшением кода.

3.1.1. Правило «Пять строк»

Утверждение

Метод не должен содержать более пяти строк кода, исключая { и }.

Объяснение

Строка, иногда называемая *инструкцией*, относится к `if`, `for`, `while` или к иным ключевым конструкциям, оканчивающимся точкой с запятой: сюда входят присваивания, вызовы методов, `return` и т. д. При этом мы не считаем пробелы и скобки: { и }.

Любой метод можно привести в соответствие с этим правилом. Вот легкий способ, демонстрирующий, как это возможно сделать: если у нас есть метод с 20 строками, то можно создать вспомогательный метод с первыми десятью строками и другой метод с остальными десятью. Теперь исходный метод состоит всего из двух строк: одна вызывает первый вспомогательный метод, а другая — второй. Этот процесс можно повторять до тех пор, пока у нас не получится всего по две строки в каждом методе.

Конкретная величина этого ограничения (5) не так важна, как сам факт соблюдения ограничения. По моему опыту, вполне сработает установка любого значения величины ограничения, необходимого для реализации прохода по фундаментальной структуре данных.

В этой книге мы работаем в 2D-режиме, то есть наша фундаментальная структура данных — это двухмерный массив. Две нижеприведенные функции перебирают 2D-массив: одна проверяет, содержит ли он четное число, а вторая находит в нем минимальный элемент. Причем каждая реализует свою задачу ровно в пяти строках кода.

Листинг 3.1. Функция, ищущая в 2D-массиве четное число

```
function containsEven(arr: number[][]) {  
  for (let x = 0; x < arr.length; x++) {  
    for (let y = 0; y < arr[x].length; y++) {  
      if (arr[x][y] % 2 === 0) {  
        return true;  
      }  
    }  
  }  
  return false;  
}
```

В TYPESCRIPT...

Не существует отдельных типов для целочисленных значений и значений с плавающей точкой. В этом языке и те и другие относятся к одному типу: `number`.

Листинг 3.2. Функция, ищущая в 2D-массиве минимальный элемент

```
function minimum(arr: number[][]): number {
    let result = Number.POSITIVE_INFINITY;
    for (let x = 0; x < arr.length; x++) {
        for (let y = 0; y < arr[x].length; y++) {
            result = Math.min(arr[x][y], result);
        }
    }
    return result;
}
```

В TYPESCRIPT...

Для объявления переменных используется `let`. `let` пытается самостоятельно вывести тип, но существует и возможность указать его явно, например, так: `let a: number = 5;`. В нем *никогда* не используется `var` ввиду странности работы области видимости этого оператора, а именно, возможности определения переменных после их использования. Ниже левый вариант кода валиден, но отражает не то, что нам нужно. А вот код справа выдает ошибку, что мы от него и ожидаем.

Плохо

```
a = 5;
var a: number;
```

Хорошо

```
a = 5;
let a: number;
```

Чтобы прояснить подсчет строк, я приведу пример, который уже встречался нам в начале главы 2. Здесь мы насчитываем четыре строки: по одной для каждой `if` (включая `else`) и каждой точки с запятой.

Листинг 3.3. Четырехстрочный метод из главы 2

```
function isTrue(bool: boolean) {
    if (bool)
        return true;
    else return false;
}
```

Запах

Наличие длинных методов уже говорит о запахе. Все дело в том, что с ними сложно работать, так как приходится одновременно удерживать всю логику метода в голове. Однако что же стоит за выражением «длинный метод»? Что значит *длинный*?

Чтобы ответить на этот вопрос, обратимся к другому признаку запаха: методы должны делать что-то одно. Если для выполнения одного значительного действия как раз достаточно пяти строк, то данное ограничение ограждает нас и от

этого запаха. Иногда мы работаем в условиях, где фундаментальная структура данных в разных частях кода различна. Как только мы освоимся с рассматриваемым правилом, сможем начать варьировать количество строк кода для подгонки под конкретные примеры. Это нормально, но на практике количество строк в методе без запаха зачастую приближается именно к пяти.

Намерение

Оставаясь без проверки, по мере добавления новой функциональности методы склонны постепенно разрастаться. Это сильно усложняет их понимание, и ограничение размера призвано предотвратить подобную ситуацию и тем самым оградить нас от необходимости разбираться с путаницей кодов.

Уверяю вас, что четыре метода, каждый из которых состоит из пяти строк кода, можно намного быстрее понять, чем один, состоящий из 20 строк. Дело в том, что название каждого метода позволяет передать намерение кода. По своей сути, именование методов равнозначно добавлению комментария не реже одного раза в пять строк. К тому же если небольшие методы названы грамотно, то и для большой функции найти хорошее имя будет несложно.

Дополнительная информация

Для помощи в освоении этого правила смотрите шаблон «Извлечение метода». Подробнее о запахе, относящемся к правилу «Методы должны делать что-то одно», хорошо написано в книге Роберта К. Мартина «Чистый код», а о запахе «Длинные методы» — в книге Мартина Фаулера «Рефакторинг».

3.2. ШАБЛОН ПРОЕКТИРОВАНИЯ ДЛЯ РАЗБИВКИ ФУНКЦИЙ

Хотя правило «Пять строк» просто понять, следовать ему куда сложнее. Поэтому мы будем неоднократно к нему возвращаться, прорабатывая все более сложные примеры части I.

Вооружившись этим правилом, подготовились к погружению в код и... Начнем с функции `draw`. Для понимания кода всегда первым делом оцениваем имя функции. Опасность здесь в том, чтобы не увязнуть в попытке разобрать каждую строчку кода — это займет очень много времени и окажется непродуктивным. Вместо этого приступим к анализу «формы» кода.

Попытаемся определить группы строк, относящиеся к чему-то одному. Чтобы было проще их выявить, будем добавлять пустые строки, обособляя предполагаемую группу. Иногда можно добавить комментарии, которые помогают запомнить, к чему относится та или иная группа. Как правило, лучше комментариев

избегать, поскольку они устаревают либо используются подобно дезодоранту против запаха плохого кода. В данном же случае, как вы скоро увидите, указание дополнительных комментариев — мера временная.

Зачастую при изначальном создании программы разработчики удерживают группы инструкций в уме и вставляют пустые строки для отслеживания логики плюс иногда добавляют комментарии. Но как только новый специалист захочет узнать, что же код делает, выяснится, что этот самый код уже не в своем перво-зданном виде, и это намерение очевидно контрпродуктивно. Вы могли слышать фразу: «Проще всего съесть слона маленькими кусочками». Именно это мы сейчас и делаем. Не разбирая сразу всю функцию, мы делим ее и обрабатываем каждый фрагмент, который достаточно мал и понятен.

На рис. 3.1 скрыты все несущественные строки, чтобы вы могли сосредоточиться только на структуре, не отвлекаясь на детали. (Так мы поступили только здесь, в начале процесса анализа.) Даже без возможности увидеть подробности, мы замечаем две группы строк, каждая из них начинается с комментария соответственно: `// Рисуем карту` и `// Рисуем игрока`.

```
function draw() {
    [redacted]
    [redacted]
    [redacted]

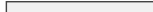
    // Рисуем карту
    [redacted] {
        [redacted] {
            [redacted]
            [redacted]
            [redacted]
            [redacted]
            [redacted]
            [redacted]
            [redacted]
            [redacted]
            [redacted]
            [redacted]
            [redacted]
            [redacted]
            [redacted]
        }
    }

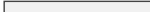
    // Рисуем игрока
    [redacted]
    [redacted]
}
```

Рис. 3.1. Начальная функция draw

Можно воспользоваться этими комментариями и проделать следующее.

1. Создать новый (пустой) метод `drawMap`.
2. В месте комментария поместить вызов к `drawMap`.
3. Выбрать все строки в найденной группе, затем вырезать и вставить их в качестве тела `drawMap`.
4. Повторив тот же процесс для `drawPlayer`, получаем измененную версию (рис. 3.2, 3.3).

```
function draw() {
    
    
    
    // Рисуем карту
     {
         {
            
            
            
            
            
            
            
            
            
            
            
            
            
            
        }
    }
}

// Рисуем игрока


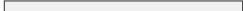
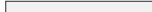
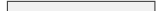
}
```

Рис. 3.2. До

```
function draw() {
    
    

    drawMap(g);
    drawPlayer(g);
}

function drawMap(g: Canvas  ) {
     {
         {
            
            
            
            
            
            
            
            
            
            
            
            

            
            
        }
    }
}

function drawPlayer(g: Canvas  ) {
    
    
}
```

Рис. 3.3. После

Теперь посмотрим, как это работает в реальном коде. Начнем с листинга 3.4. Обратите внимание, что мы видим ту же структуру, по-прежнему не вникая в действия каждой отдельной строки.

Листинг 3.4. Начальная форма

```
function draw() {
  let canvas = document.getElementById("GameCanvas") as HTMLCanvasElement;
  let g = canvas.getContext("2d");
```

```

g.clearRect(0, 0, canvas.width, canvas.height);

// Рисуем карту
for (let y = 0; y < map.length; y++)
  for (let x = 0; x < map[y].length; x++)
    if (map[y][x] === Tile.FLUX)
      g.fillStyle = "#ccffcc";
    else if (map[y][x] === Tile.UNBREAKABLE)
      g.fillStyle = "#999999";
    else if (map[y][x] === Tile.STONE || map[y][x] === Tile.FALLING_STONE)
      g.fillStyle = "#0000cc";
    else if (map[y][x] === Tile.BOX || map[y][x] === Tile.FALLING_BOX)
      g.fillStyle = "#8b4513";
    else if (map[y][x] === Tile.KEY1 || map[y][x] === Tile.LOCK1)
      g.fillStyle = "#ffcc00";
    else if (map[y][x] === Tile.KEY2 || map[y][x] === Tile.LOCK2)
      g.fillStyle = "#00ccff";

    if (map[y][x] !== Tile.AIR && map[y][x] !== Tile.PLAYER)
      g.fillRect(x * TILE_SIZE, y * TILE_SIZE, TILE_SIZE, TILE_SIZE);
  }
}

// Рисуем игрока
g.fillStyle = "#ff0000";
g.fillRect(playerx * TILE_SIZE, playery * TILE_SIZE, TILE_SIZE, TILE_SIZE);
}

```

Комментарии, отмечающие начало логической группировки строк

Комментарии, отмечающие начало логической группировки строк

В TYPESCRIPT...

Преобразование типов, подобное приведению в других языках, выполняется с помощью `as`. В случае недействительности преобразования `null` не возвращается, как это происходит при работе с `as` в C#.

Теперь повторяем шаги, описанные ранее.

1. Создаем новый (пустой) метод `drawMap`.
2. В месте комментария помещаем вызов `drawMap`.
3. Выбираем все соответствующие строки в найденной группе, вырезаем и вставляем их в качестве тела `drawMap`.

Если попробовать выполнить компиляцию сейчас, то выскочит несколько ошибок. Причина в том, что переменная `g` больше не находится в области видимости. Это можно исправить. Сначала наведите курсор на `g` в оригинальном методе `draw`. Так мы узнаем ее тип, который используем для введения в `drawMap` параметра `g`: `CanvasRenderingContext2D`.

Очередная компиляция сообщит нам, что в месте вызова `drawMap` есть ошибка, потому что здесь не хватает параметра `g`. Опять же это несложно исправить: мы передаем `g` в качестве аргумента.

Теперь те же действия мы повторяем для `drawPlayer` и, как видно ниже, получаем ровно то, что ожидали. Обратите внимание: при этом по-прежнему не требуется выяснять, что делает код на более глубоких уровнях, достаточно именования методов.

Листинг 3.5. После «Извлечения метода»

```
function draw() {
  let canvas = document.getElementById("GameCanvas") as HTMLCanvasElement;
  let g = canvas.getContext("2d");

  g.clearRect(0, 0, canvas.width, canvas.height);

  drawMap(g);
  drawPlayer(g);
}

function drawMap(g: CanvasRenderingContext2D) {
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length; x++) {
      if (map[y][x] === Tile.FLUX)
        g.fillStyle = "#ccffcc";
      else if (map[y][x] === Tile.UNBREAKABLE)
        g.fillStyle = "#999999";
      else if (map[y][x] === Tile.STONE || map[y][x] === Tile.FALLING_STONE)
        g.fillStyle = "#0000cc";
      else if (map[y][x] === Tile.BOX || map[y][x] === Tile.FALLING_BOX)
        g.fillStyle = "#8b4513";
      else if (map[y][x] === Tile.KEY1 || map[y][x] === Tile.LOCK1)
        g.fillStyle = "#ffcc00";
      else if (map[y][x] === Tile.KEY2 || map[y][x] === Tile.LOCK2)
        g.fillStyle = "#00ccff";

      if (map[y][x] !== Tile.AIR && map[y][x] !== Tile.PLAYER)
        g.fillRect(x * TILE_SIZE, y * TILE_SIZE, TILE_SIZE, TILE_SIZE);
    }
  }
}

function drawPlayer(g: CanvasRenderingContext2D) {
  g.fillStyle = "#ff0000";
  g.fillRect(playerx * TILE_SIZE, playery * TILE_SIZE, TILE_SIZE, TILE_SIZE);
}
```

Новая функция и вызов, соответствующие первому комментарию

Новая функция и вызов, соответствующие второму комментарию

Вот и проведены два первых рефакторинга. Поздравляю! Процесс, через который мы только что прошли, соответствует стандартному шаблону рефакторинга, который называется «Извлечение метода».

ПРИМЕЧАНИЕ

Поскольку мы просто перемещаем строки, то риск внедрить ошибку минимален, особенно с учетом того, что компилятор сигнализировал нам о забытых параметрах.

Мы использовали комментарии в качестве имен методов. Следовательно, имена функций и комментариев передают одинаковую информацию. Теперь можно избавиться от комментариев и от ставших теперь бесполезными пустых строк, которые прежде использовались для группировки.

3.2.1. Шаблон рефакторинга «Извлечение метода»**Описание**

«Извлечение метода» выделяет часть одного метода и помещает ее в отдельный метод. Это можно делать механически, и многие современные IDE по умолчанию содержат в себе данный шаблон рефакторинга. Уже одно только это делает его безопасным, ведь компьютеры редко ошибаются в таких вещах. Но можно делать это и вручную, не понижая при этом уровня безопасности.

Подобный подход может вызвать сложности, если делать присваивание нескольким параметрам или выполнять `return` не во всех ветках. Но рассмотрение подобных ситуаций выходит за рамки этой книги, поскольку они очень редки и обычно их можно упростить перестановкой или повторением строк в методах.

СОВЕТ

Поскольку присутствие `return` не во всех ветках `if` может помешать извлечению метода, я советую начинать с конца метода и продвигаться вверх. В результате операция `return` как бы окажется впереди — будет обработана одной из первых и по окончании будет выполнена для всех веток.

Процесс

1. Отмечаем строки для извлечения, отделяя их пустыми строками. При необходимости добавляем комментарий.
2. Создаем новый (пустой) метод с нужным именем.
3. В начале группы размещаем вызов этого нового метода.
4. Выбираем все строки в группе, вырезаем их и вставляем в качестве тела созданного метода.
5. Компилируем.
6. Вносим параметры, вызывая тем самым ошибки компиляции.
7. Если мы делаем присваивание одному из этих параметров (назовем его `p`):
 - помещаем `return p`; в конце нового метода;
 - помещаем инструкцию присваивания `p = newMethod(...)`; в месте вызова.

8. Компилируем.
9. Передаем аргументы, исправляя таким образом ошибки компиляции.
10. Удаляем ненужные более пустые строки и комментарии.

Пример

Рассмотрим полномасштабный пример работы этого процесса. Здесь у нас снова функция, находящая минимальный элемент в 2D-массиве. Мы выяснили, что она слишком длинная, следовательно, нужно извлечь ее часть, находящуюся между пустыми строками.

Листинг 3.6. Функция для поиска минимального элемента в 2D-массиве

```
function minimum(arr: number[][]) {
  let result = Number.POSITIVE_INFINITY;
  for (let x = 0; x < arr.length; x++)
    for (let y = 0; y < arr[x].length; y++)

      if (result > arr[x][y])
        result = arr[x][y];

  return result;
}
```

Строки, которые
нужно извлечь

Делаем следующее.

1. Отмечаем строки для извлечения, обособляя их пустыми строками. При необходимости добавляем комментарии.
2. Создаем новый метод `min`.
3. В начале группы помещаем вызов к `min`.
4. Вырезаем и вставляем строки группы в тело нового метода.

Листинг 3.7. До

```
function minimum(arr: number[][]) {
  let result = Number.POSITIVE_INFINITY;
  for (let x = 0; x < arr.length; x++)
    for (let y = 0; y < arr[x].length; y++)

      if (result > arr[x][y])
        result = arr[x][y];

  return result;
}
```

Листинг 3.8. После (1/3)

```
function minimum(arr: number[][]) {
  let result = Number.POSITIVE_INFINITY;
  for (let x = 0; x < arr.length; x++)
    for (let y = 0; y < arr[x].length; y++)

      min();

  return result;
}

function min() {
  if (result > arr[x][y])
    result = arr[x][y];
}
```

Новый метод
и вызов

Извлеченные строки
из версии «до»

5. Компилируем.
6. Вводим параметры для `result`, `arr`, `x` и `y`.
7. Извлеченная функция делает присваивание `result`. Значит, нужно:
 - поместить `return result`; в конце `min`;
 - поместить инструкцию присваивания `result = min(...)`; в месте вызова.

Листинг 3.9. До

```
function minimum(arr: number[][]){
  let result = Number.POSITIVE_INFINITY;
  for (let x = 0; x < arr.length; x++)
    for (let y = 0; y < arr[x].length; y++)

      min();

  return result;
}

function min() {

  if (result > arr[x][y])
    result = arr[x][y];
}
```

8. Компилируем.
9. Передаем вызывающие ошибки аргументы `result`, `arr`, `x` и `y`.
10. В завершение удаляем ненужные пустые строки.

Листинг 3.11. До

```
function minimum(arr: number[][]){
  let result = Number.POSITIVE_INFINITY;
  for (let x = 0; x < arr.length; x++)
    for (let y = 0; y < arr[x].length; y++)
      result = min();
  return result;
}

function min(
  result: number, arr: number[][],
  x: number, y: number)
{
  if (result > arr[x][y])
    result = arr[x][y];
  return result;
}
```

Листинг 3.10. После (2/3)

```
function minimum(arr: number[][]){
  let result = Number.POSITIVE_INFINITY;
  for (let x = 0; x < arr.length; x++)
    for (let y = 0; y < arr[x].length; y++)

      result = min(); ← Присваивание
                        result

  return result;
}

function min(
  result: number, arr: number[][],
  x: number, y: number)
{
  if (result > arr[x][y])
    result = arr[x][y];
  return result; ← Добавленная
                  инструкция return
}
```

Добавленные
параметры

Листинг 3.12. После (3/3)

```
function minimum(arr: number[][]){
  let result = Number.POSITIVE_INFINITY;
  for (let x = 0; x < arr.length; x++)
    for (let y = 0; y < arr[x].length; y++)
      result = min(result, arr, x, y); ←
  return result;
}

function min(
  result: number, arr: number[][],
  x: number, y: number)
{
  if (result > arr[x][y])
    result = arr[x][y];
  return result;
}
```

Аргументы добавлены,
а пустые строки удалены

Может показаться, что будет лучше использовать вместо трех отдельных аргументов встроенные `Math.min` или `arr[x][y]`. Если сможете сделать это безопасно, то такой подход действительно может оказаться более удачным. Но помните, что можно легко перемудрить, стараясь быть умнее, что обычно себя не оправдывает. А из приведенного примера можно понять, что предложенная выше трансформация, хотя и немного громоздкая, заведомо *безопасна*.

В ней можно быть уверенным, что данный процесс ничего не нарушит. И такая уверенность окажется ценнее, чем идеальный результат, особенно с учетом того, что мы не изучали функциональность кода. Чем больше элементов нам нужно отслеживать, тем выше шанс что-нибудь забыть. Компилятор же ничего не забывает, и приведенный процесс построен именно на основе этого факта. Лучше уж создать необычный с виду код, который безопасен, чем красивый, в безопасности которого можно усомниться. (Если бы мы могли полагаться, например, на обширное автоматизированное тестирование, то можно было бы пойти на риски. Но у нас не тот случай.)

Дополнительные материалы

Для получения красивого результата можно было бы совместить несколько других шаблонов рефакторинга. Мы не станем разбирать их подробно, поскольку в этой книге рассматриваются только межметодные шаблоны рефакторинга. Но я опишу здесь этот вариант коротко, на случай, если вам захочется разобратся в нем самостоятельно.

1. Выполнить другой небольшой шаблон рефакторинга, «Извлечение общего подвыражения», который в данном случае вводит временную переменную `let tmp = arr[x][y]`; вне группы и заменяет вхождения `arr[x][y]` в этой группе на `tmp`.
2. Использовать «Извлечение метода», как описывалось ранее.
3. Выполнить «Встраивание локальной переменной», которое отменит действия «Извлечения общего подвыражения», заменив `tmp` на `arr[x][y]` и удалив временную переменную `tmp`.

Подробнее об этих шаблонах, включая «Извлечение метода», можно почитать в уже упомянутой здесь книге «Рефакторинг» Мартина Фаулера.

3.3. РАЗБИВКА ФУНКЦИЙ ДЛЯ УРАВНОВЕШИВАНИЯ АБСТРАКЦИЙ

Мы добились от нашей исходной функции `draw` размера пять строк. Да, `drawMap` конфликтует с этим правилом, и это будет исправлено в главе 4. Но при этом с `draw` мы еще не закончили: она тоже конфликтует, правда, уже с другим правилом.

3.3.1. Правило «Вызов или передача»

Утверждение

Функция должна либо вызывать методы в объекте, либо передавать этот объект в виде аргумента, но не то и другое одновременно.

Объяснение

Когда мы начинаем вводить больше методов и передавать элементы в качестве параметров, можем столкнуться с неравным распределением ответственностей. Например, функция может выполнять и низкоуровневые операции, такие как установка индекса в массиве, и одновременно передавать этот массив более сложной функции в качестве аргумента. Подобный код будет сложно читать, потому что потребуется переключаться между низкоуровневыми операциями и высокоуровневыми именами методов. Гораздо проще придерживаться единственного уровня абстракции.

Взгляните на следующую функцию, которая находит среднее значение в массиве. Обратите внимание, что в ней используется и высокоуровневая абстракция `sum(arr)`, и низкоуровневая `arr.length`.

Листинг 3.13. Функция для поиска среднего по массиву

```
function average(arr: number[]) {
  return sum(arr) / arr.length;
}
```

Этот код нарушает названное правило. Здесь лучше подойдет реализация, абстрагирующая поиск длины.

Листинг 3.14. До

```
function average(arr: number[]) {
  return sum(arr) / arr.length;
}
```

Листинг 3.15. После

```
function average(arr: number[]) {
  return sum(arr) / size(arr);
}
```

Запах

Утверждение «Содержимое функции должно находиться на единственном уровне абстракции» настолько значительно, что само по себе выражает запах. Тем не менее, как и в случае с большинством других запахов, сложно определить его количественно, не говоря уже о поиске способа решения. Зато нетрудно определить, передается ли что-то в качестве аргумента, а также заметить наличие `.` рядом с ним.

Намерение

Когда мы вводим абстракцию, извлекая некоторые детали метода, это правило вынуждает нас также извлечь и другие детали. Таким образом, мы гарантируем, что внутренний уровень абстракции метода всегда останется одинаков.

Дополнительная информация

В помощь для понимания и реализации этого правила читайте о шаблоне рефакторинга «Извлечение метода». Подробнее о запахе «Содержимое функции должно находиться на одном уровне абстракции» можете узнать из книги Роберта К. Мартина «Чистый код. Создание, анализ и рефакторинг».

3.3.2. Применение правила

Опять же, если оценить метод `draw` в его текущем виде, который представлен на рис. 3.4, легко заметить, что в нем это правило нарушается. Ведь мы передаем переменную в качестве параметра и в то же время вызываем для нее метод.

```
function draw() {
    [ ]
    [ ]

    g. [ ]

    [ ](g);
    [ ](g);
}
```

Рис. 3.4. `g` передается и вызывается

Эти нарушения правила мы исправим с помощью «Извлечения метода». Но что мы будем извлекать? Здесь нужно обратить внимание на некоторые детали. Мы видим пустые строки кода, но если извлечь строку с `g.clearRect`, то в итоге мы будем передавать `canvas` в качестве аргумента, а также вызывать `canvas.getContext` — снова налицо нарушение правила.

Листинг 3.16. Так выглядит `draw` сейчас

```
function draw() {
    let canvas = document.getElementById("GameCanvas") as HTMLCanvasElement;
    let g = canvas.getContext("2d");

    g.clearRect(0, 0, canvas.width, canvas.height); ← Вызываем для g метод

    drawMap(g);
    drawPlayer(g); | g передается в качестве аргумента
}
```

Более уместно извлечь совместно первые три строки. Мы так и сделаем. Но прежде, чем извлекать эти строки, обсудим, что значит хорошее имя метода, ведь при каждом выполнении извлечения метода есть отличная возможность повысить читаемость кода введением хорошего имени.

3.4. СВОЙСТВА ХОРОШЕГО ИМЕНИ ФУНКЦИИ

Я не могу предоставить универсальные правила для создания *хорошего имени*, но могу назвать несколько свойств, которыми оно должно обладать:

- оно должно быть правдивым и описывать реальное назначение функции;
- оно должно быть полноценным, то есть охватывать все, что делает функция;
- оно должно быть понятно для работающих в этой области людей, поэтому следует использовать соответствующую терминологию. В качестве дополнительного преимущества это повышает эффективность коммуникации и упрощает обсуждение кода с коллегами или заказчиками.

Здесь впервые нам требуется принять в рассмотрение действия кода, потому что другие указатели, на которые мы могли бы ориентироваться, отсутствуют. К счастью, мы уже существенно снизили количество строк, которые нужно проанализировать: их осталось всего три.

Первая строка получает HTML-элемент, на котором нужно произвести отрисовку, вторая инстанцирует графические элементы, а третья очищает холст. Если коротко, то код создает графический объект.

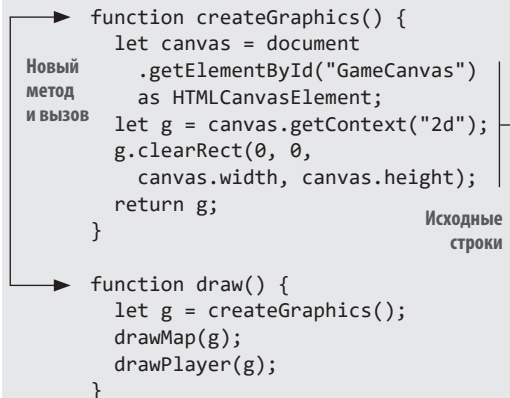
Листинг 3.17. До

```
function draw() {
  let canvas = document
    .getElementById("GameCanvas")
    as HTMLCanvasElement;
  let g = canvas.getContext("2d");

  g.clearRect(0, 0,
    canvas.width, canvas.height);

  drawMap(g);
  drawPlayer(g);
}
```

Листинг 3.18. После



```
function createGraphics() {
  let canvas = document
    .getElementById("GameCanvas")
    as HTMLCanvasElement;
  let g = canvas.getContext("2d");
  g.clearRect(0, 0,
    canvas.width, canvas.height);
  return g;
}

function draw() {
  let g = createGraphics();
  drawMap(g);
  drawPlayer(g);
}
```

Обратите внимание, что нам больше не требуются пустые строки, так как код легко понять и без них.

На этом отработка `draw` закончена, и можно двигаться дальше. Вернемся к началу и повторим аналогичный процесс для другой длинной функции: `update`. В этом случае снова даже без прочтения кода можно определить две очевидные группы строк, разделенные пустой строкой.

Листинг 3.19. Начальная форма

```
function update() {
  while (inputs.length > 0) {
    let current = inputs.pop();
    if (current === Input.LEFT)
      moveHorizontal(-1);
    else if (current === Input.RIGHT)
      moveHorizontal(1);
    else if (current === Input.UP)
      moveVertical(-1);
    else if (current === Input.DOWN)
      moveVertical(1);
  }

  for (let y = map.length - 1; y >= 0; y--) {
    for (let x = 0; x < map[y].length; x++) {
      if ((map[y][x] === Tile.STONE || map[y][x] === Tile.FALLING_STONE)
        && map[y + 1][x] === Tile.AIR) {
        map[y + 1][x] = Tile.FALLING_STONE;
        map[y][x] = Tile.AIR;
      } else if ((map[y][x] === Tile.BOX || map[y][x] === Tile.FALLING_BOX)
        && map[y + 1][x] === Tile.AIR) {
        map[y + 1][x] = Tile.FALLING_BOX;
        map[y][x] = Tile.AIR;
      } else if (map[y][x] === Tile.FALLING_STONE) {
        map[y][x] = Tile.STONE;
      } else if (map[y][x] === Tile.FALLING_BOX) {
        map[y][x] = Tile.BOX;
      }
    }
  }
}
```

← Пустая строка, разделяющая две группы

Можно естественным образом разделить этот код на две меньшие функции. Как их назовем? Обе группы все еще достаточно сложны, поэтому стоит отложить их разбор на потом. Чисто поверхностно можно заметить, что в первой группе преобладает слово `input`, а во второй — слово `map`. Мы знаем, что разделяем функцию под названием `update`, значит, в качестве черногового варианта можем совместить эти слова, чтобы получить имена функций `updateInput` и `updateMap`.

`updateMap` — это осмысленно и в порядке, а вот для входных данных (`Input`) нам вряд ли нужно «обновление» — `update`. Поэтому мы используем другой прием именования и заменим `update` на `handle`: `handleInputs`.

ПРИМЕЧАНИЕ

При выборе подобных имен всегда возвращайтесь к ним позже, когда функции станут меньше, чтобы оценить их обоснованность и возможность улучшения.

Листинг 3.20. После применения «Извлечения метода»

```

function update() {
  handleInputs();
  updateMap();
}

function handleInputs() {
  while (inputs.length > 0) {
    let current = inputs.pop();
    if (current === Input.LEFT)
      moveHorizontal(-1);
    else if (current === Input.RIGHT)
      moveHorizontal(1);
    else if (current === Input.UP)
      moveVertical(-1);
    else if (current === Input.DOWN)
      moveVertical(1);
  }
}

function updateMap() {
  for (let y = map.length - 1; y >= 0; y--) {
    for (let x = 0; x < map[y].length; x++) {
      if ((map[y][x] === Tile.STONE || map[y][x] === Tile.FALLING_STONE) &&
        map[y + 1][x] === Tile.AIR) {
        map[y + 1][x] = Tile.FALLING_STONE;
        map[y][x] = Tile.AIR;
      } else if ((map[y][x] === Tile.BOX || map[y][x] ===
        Tile.FALLING_BOX) && map[y + 1][x] === Tile.AIR) {
        map[y + 1][x] = Tile.FALLING_BOX;
        map[y][x] = Tile.AIR;
      } else if (map[y][x] === Tile.FALLING_STONE) {
        map[y][x] = Tile.STONE;
      } else if (map[y][x] === Tile.FALLING_BOX) {
        map[y][x] = Tile.BOX;
      }
    }
  }
}
  
```

Извлеченная первая группа и вызов

Извлеченная вторая группа и вызов

Теперь `update` вписывается в наши правила, и мы с ней заканчиваем. Результат кажется далеким от идеала, но мы постепенно приближаемся к заветным магическим пяти строкам.

3.5. РАЗБИВКА ФУНКЦИЙ, ДЕЛАЮЩИХ СЛИШКОМ МНОГО

С `update` мы закончили, и можно продолжить, например, с одной из только что введенных функций: `updateMap`. В этой функции добавлять дополнительные пробелы будет неестественным. Следовательно, нам нужно использовать другое правило: помещать `if` *только в начале функции*.

3.5.1. Правило «if только в начале»

Утверждение

Если в функции есть инструкция `if`, то она должна идти первой.

Объяснение

Ранее уже указывалось, что функции должны выполнять одно действие. Проверка чего-либо — это и есть одно действие. Значит, если в функции есть `if`, то эта инструкция должна идти в ней первой. Она также должна быть *единственным* действием в том смысле, что после нее ничего выполняться не должно. Мы можем избежать наличия чего-либо после нее, если извлечем эту часть в отдельный фрагмент, как уже делали несколько раз.

Когда мы говорим, что `if` должна быть единственным действием, которое выполняет метод, подразумеваем, что не требуется извлекать ее тело, равно как отделять от нее компонент `else`. И тело, и `else` являются частью структуры кода, на которую мы опираемся в качестве ориентира, чтобы исключить необходимость разбираться в коде. Поведение и структура тесно связаны, а раз мы выполняем рефакторинг, то поведение изменять не должны — значит, и структуру тоже.

В примере ниже показана функция, выводящая на печать простые числа от 2 до n .

Листинг 3.21. Функция для вывода на печать всех простых чисел от 2 до n

```
function reportPrimes(n: number) {
  for (let i = 2; i < n; i++)
    if (isPrime(i))
      console.log(`${i} is prime`);
}
```

Здесь очевидно присутствуют как минимум две ответственности:

- перебор чисел;
- проверка, является ли число простым.

Следовательно, и функций здесь должно получиться не менее двух.

Листинг 3.22. До

```
function reportPrimes(n: number) {
  for (let i = 2; i < n; i++)
    if (isPrime(i))
      console.log(`${i} is prime`);
}
```

Листинг 3.23. После

```
function reportPrimes(n: number) {
  for (let i = 2; i < n; i++)
    reportIfPrime(i);
}

function reportIfPrime(n: number) {
  if (isPrime(n))
    console.log(`${n} is prime`);
}
```

Всякий раз, когда мы что-то проверяем, это является ответственностью и должно обрабатываться отдельной функцией. На этом-то и основывается данное правило.

Запах

Задача этого правила, как и правила «Пять строк», — помочь исключить появление запаха функций, выполняющих более одного действия.

Намерение

Это правило подразумевает изолирование инструкций `if`, так как они уже сами по себе имеют одну ответственность, в то же время цепочка из `else if` представляет неделимую единицу, которую разрывать нельзя. Это означает, что наименьшее количество строк, достижимое с помощью извлечения метода в контексте `if` и `else if`, может получиться при извлечении этой `if` вместе с ее `else if`.

Дополнительная информация

Для помощи в реализации этого правила смотрите технику рефакторинга «Извлечение метода». Подробнее о запахе «Методы должны выполнять одно действие» читайте в книге Роберта К. Мартина «Чистый код».

3.5.2. Применение правила

Нарушения этого правила легко обнаружить, даже не вникая в специфику кода. На рис. 3.5 есть одна большая группа `if` в середине функции.


```

function updateMap() {
  [redacted] {
    [redacted] {
      if [redacted]
        [redacted] {
          [redacted]
          [redacted]
        } else if [redacted]
          [redacted] {
            [redacted]
            [redacted]
          } else if [redacted] {
            [redacted]
            [redacted]
          } else if [redacted] {
            [redacted]
            [redacted]
          }
        }
      }
    }
  }
}

```

Рис. 3.5. if в середине функции

Чтобы подобрать грамотное название для функции, которую мы хотим извлечь, нужно поверхностно взглянуть на извлекаемый код. В группе этих строк присутствует два основных слова: `map` и `tile`. У нас уже есть функция `updateMap`, значит, новую мы назовем `updateTile`.

Листинг 3.24. После извлечения метода

```

function updateMap() {
  for (let y = map.length - 1; y >= 0; y--) {
    for (let x = 0; x < map[y].length; x++) {
      updateTile(x, y);
    }
  }
}

function updateTile(x: number, y: number) {
  if ((map[y][x] === Tile.STONE || map[y][x] === Tile.FALLING_STONE)
    && map[y + 1][x] === Tile.AIR) {
    map[y + 1][x] = Tile.FALLING_STONE;
    map[y][x] = Tile.AIR;
  } else if ((map[y][x] === Tile.BOX || map[y][x] === Tile.FALLING_BOX)
    && map[y + 1][x] === Tile.AIR) {
    map[y + 1][x] = Tile.FALLING_BOX;
  }
}

```

Извлеченный
метод и вызов

```

    map[y][x] = Tile.AIR;
  } else if (map[y][x] === Tile.FALLING_STONE) {
    map[y][x] = Tile.STONE;
  } else if (map[y][x] === Tile.FALLING_BOX) {
    map[y][x] = Tile.BOX;
  }
}

```

Теперь `updateMap` вписывается в предел пяти строк, и это хорошо. Далее на гребне успеха сразу выполним аналогичную трансформацию с `handleInputs`.

Листинг 3.25. До

```

function handleInputs() {
  while (inputs.length > 0) {
    let current = inputs.pop();
    if (current === Input.RIGHT)
      moveHorizontal(1);
    else if (current === Input.LEFT)
      moveHorizontal(-1);
    else if (current === Input.DOWN)
      moveVertical(1);
    else if (current === Input.UP)
      moveVertical(-1);
  }
}

```

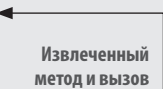
Листинг 3.26. После

```

function handleInputs() {
  while (inputs.length > 0) {
    let current = inputs.pop();
    handleInput(current);
  }
}

function handleInput(input: Input) {
  if (input === Input.RIGHT)
    moveHorizontal(1);
  else if (input === Input.LEFT)
    moveHorizontal(-1);
  else if (input === Input.DOWN)
    moveVertical(1);
  else if (input === Input.UP)
    moveVertical(-1);
}

```



Извлеченный метод и вызов

На этом завершаем работу с `handleInputs`. Нам удалось добиться дополнительной оптимизации читаемости. Она была получена с помощью так называемой техники «Извлечение метода». Эта техника в числе прочего предполагает присвоение параметрам новых имен, которые оказываются более информативными в новом контексте кода. К примеру, `current` отлично подходит в качестве имени для переменной в цикле, но в новой функции `handleInput` более подходящим названием будет `input`.

Мы ввели функцию, которая выглядит проблематичной. `handleInput` уже компактна, поэтому трудно определить, как вписать ее в правило пяти строк. В данной главе мы рассмотрели только технику «Извлечение метода» и правила, помогающие определить ситуации для ее применения. Но поскольку тело каждой `if` уже представляет одну строку и нельзя расчленять цепочки `if else`, то к `handleInput` применять «Извлечение метода» не рекомендуется. Тем не менее, как мы увидим в следующей главе, и для этого случая есть элегантное решение.

РЕЗЮМЕ

- Правило «Пять строк» утверждает, что методы должны содержать не более пяти строк. Оно помогает выявить те из них, которые выполняют более одного действия. Для разбивки таких длинных методов мы применяем шаблон рефакторинга «Извлечение метода» и попутно удаляем комментарии, преобразуя их в имена методов.
- Правило «Вызов или передача» утверждает, что метод должен либо вызывать методы для объекта, либо передавать этот объект в виде параметра, но не то и другое сразу. Оно помогает выявить методы, которые смешивают несколько уровней абстракции. В данном случае для разделения этих уровней и соотнесения их с разными структурными единицами снова используем «Извлечение метода».
- Имена методов должны быть правдивы, полноценны и понятны. «Извлечение метода» позволяет нам присваивать параметрам осмысленные имена для дополнительного повышения читаемости.
- Правило «if только в начале» утверждает, что проверка условия с помощью `if` выполняет одно действие, значит, метод больше ничего делать не должен. Это правило также помогает выявить методы, выполняющие более одного действия. Для изолирования таких конструкций `if` мы снова используем «Извлечение метода».

4

Пусть код типа работает

В этой главе

- ✓ Устранение ранней привязки с помощью правил «Никогда не использовать `if c else`» и «Никогда не использовать `switch`».
- ✓ Удаление инструкции `if` с помощью правил «Замена кода типов классами» и «Перемещение кода в классы».
- ✓ Избавление от ненужной обобщенности с помощью шаблона «Специализация метода».
- ✓ Предотвращение зацепления с помощью шаблона «Наследование только от интерфейсов».
- ✓ Удаление методов с помощью правил «Встраивание метода» и «Пробное удаление с последующей компиляцией»

В конце предыдущей главы мы просто ввели функцию `handleInput`, для которой не могли использовать «Извлечение метода» (3.2.1), потому что не хотели разрывать цепочку `else if`. К сожалению, `handleInput` не вписывается в наше основополагающее правило «Пять строк» (3.1.1), поэтому оставлять все как есть нельзя.

Вот эта функция.

Листинг 4.1. Начальная форма

```
function handleInput(input: Input) {  
    if (input === Input.LEFT) moveHorizontal(-1);  
    else if (input === Input.RIGHT) moveHorizontal(1);  
    else if (input === Input.UP) moveVertical(-1);  
    else if (input === Input.DOWN) moveVertical(1);  
}
```

4.1. РЕФАКТОРИНГ ПРОСТОЙ ИНСТРУКЦИИ IF

Здесь мы немного застряли. Чтобы показать вам, как обрабатывать подобные цепочки `else if`, я начну с введения нового правила.

4.1.1. Правило «Никогда не использовать `if` с `else`»

Утверждение

Никогда не используйте `if` с `else`, если только не выполняете проверку в отношении типа данных, который не контролируете.

Объяснение

Принимать решения бывает непросто. В реальной жизни многие люди склонны этого избегать и постоянно откладывают решение на потом. А вот в коде мы используем инструкции `if-else` активно. Я не стану утверждать, как лучше действовать в жизни, но в коде ожидание определенно является более удачной тактикой. Если мы используем `if-else`, то фиксируем точку, в которой программа принимает решение. Это снижает гибкость кода, поскольку исключает возможность внесения вариативности после блока `if-else`.

Конструкции `if-else` можно рассматривать как жестко закодированные решения. Однако подобно тому, как нам не нравятся жестко прописанные в коде константы, так же не нравятся и жестко прописанные решения.

Лучше никогда не прописывать решение жестко, то есть никогда не использовать `if` с `else`. К сожалению, при этом необходимо обращать внимание на то, относительно чего выполняется проверка. Например, с помощью `e.key` мы проверяем, какая клавиша нажата, здесь у нас используется тип `string`. Реализацию `string` мы изменить не можем, значит, не можем избежать и цепочки `else if`.

Но это не повод расстраиваться, потому что такие случаи обычно происходят на границах программы при получении входных данных извне приложения: пользователь что-то вводит, выполняется запрос значения из базы данных и т. д.

В таких случаях первым делом нужно отобразить сторонние типы данных в типы данных, которые мы контролируем. В нашем примере с игрой одна такая цепочка `else if` считывает ввод, сделанный пользователем, и отображает его в наши типы.

Листинг 4.2. Отображение пользовательского ввода в управляемые типы данных

```
window.addEventListener("keydown", e => {
  if (e.key === LEFT_KEY || e.key === "a") inputs.push(Input.LEFT);
  else if (e.key === UP_KEY || e.key === "w") inputs.push(Input.UP);
  else if (e.key === RIGHT_KEY || e.key === "d") inputs.push(Input.RIGHT);
  else if (e.key === DOWN_KEY || e.key === "s") inputs.push(Input.DOWN);
});
```

Мы не имеем контроля над любым из двух типов данных в этих условиях: `KeyboardEvent` и `string`. Как и говорилось, эти цепочки `else if` должны быть напрямую связаны с вводом/выводом, который, в свою очередь, должен быть отделен от остальной части приложения.

Обратите внимание, что мы считаем отдельные `if` *проверками*, а `if-else` — *решениями*. Это позволяет проводить простую проверку в начале методов, где было бы сложно извлечь ранний возврат `return`, как в следующем примере. То есть это правило конкретно нацелено на `else`.

Помимо этого, оно легко проверяется: достаточно просто найти `else`. Вернемся к более ранней функции, которая получает массив чисел и находит их среднее. Если вызвать предыдущую реализацию с пустым массивом, то мы получим ошибку деления на нуль. В этом есть смысл, потому что мы эту реализацию знаем, но для пользователя такая ошибка окажется бесполезной. Значит, желательнее более широко идентифицировать (выбросить) ошибку через `throw`. Вот два способа исправить это.

Листинг 4.3. До

```
function average(ar: number[]) {
  if (size(ar) === 0)
    throw "Empty array not allowed";
  else
    return sum(ar) / size(ar);
}
```

Листинг 4.4. После

```
function assertNotEmpty(ar: number[]) {
  if (size(ar) === 0)
    throw "Empty array not allowed";
}
function average(ar: number[]) {
  assertNotEmpty(ar);
  return sum(ar) / size(ar);
}
```

Запах

Это правило относится к *раннему связыванию*, которое является запахом. Когда мы компилируем программу, то поведение, подобное решениям `if-else`, разрешается и фиксируется в нашем приложении, не позволяя внести изменения без повторной компиляции. Противоположным этому является *позднее связывание*, когда поведение определяется в последний возможный момент уже при выполнении кода.

Раннее связывание не позволяет делать изменение путем добавления, потому что мы можем изменить инструкцию `if`, только модифицировав ее с последующей компиляцией. В свою очередь, позднее связывание позволяет использовать простое добавление, что намного предпочтительнее. Об этом мы говорили в главе 2.

Намерение

Инструкции `if` выступают в качестве операторов потока управления. Это означает, что они определяют, какой код должен выполняться далее. Но в объектно-ориентированном программировании есть намного более сильные операторы потока управления: объекты. Если использовать интерфейс с двумя реализациями, то мы сможем при выполнении решить, какой код выполнять, в зависимости от инстанцируемого класса. По сути, это правило вынуждает нас искать способы использовать объекты, которые являются более эффективными и гибкими инструментами управления.

Дополнительная информация

Позднее связывание будет более подробно рассмотрено при знакомстве с шаблонами рефакторинга «Замена кода типа классами» (4.1.3) и «Введение паттерна “Стратегия”» (5.4.2).

4.1.2. Применение правила

Первым шагом для избавления от `if-else` в `handleInput` будет замена перечисления *Input интерфейсом* `Input`. После этого значения заменяются классами. В завершение — это самая восхитительная часть — из-за того, что теперь значения являются объектами, становится возможно переместить код внутри `if` в методы каждого из классов. Но для этого нам предстоит преодолеть несколько разделов книги, так что наберитесь терпения. Мы будем идти к заветной цели шаг за шагом.

1. Введем новый интерфейс с временным именем `Input2`, содержащий методы для четырех значений в нашем перечислении.

Листинг 4.5. Новый интерфейс

```
enum Input {
    RIGHT, LEFT, UP, DOWN
}
interface Input2 {
    isRight(): boolean;
    isLeft(): boolean;
    isUp(): boolean;
    isDown(): boolean;
}
```

- Создадим четыре класса, соответствующие этим четырем значениям перечисления. Все методы, за исключением соответствующего конкретному классу, должны возвращать `false`. Заметьте: эти методы временные, в чем мы позже убедимся.

Листинг 4.6. Новые классы

```
class Right implements Input2 {
  isRight() { return true; }
  isLeft() { return false; }
  isUp() { return false; }
  isDown() { return false; }
}
class Left implements Input2 { ... }
class Up implements Input2 { ... }
class Down implements Input2 { ... }
```

isRight возвращает true
в правильном классе

Другие методы
возвращают false

- Переименуем перечисление в `RawInput`, после этого компилятор будет выдавать ошибку во всех местах, где используется это перечисление.

Листинг 4.7. До

```
enum Input {
  RIGHT, LEFT, UP, DOWN
}
```

Листинг 4.8. После (1/3)

```
enum RawInput {
  RIGHT, LEFT, UP, DOWN
}
```

- Изменим типы с `Input` на `Input2` и заменим проверки равенства новыми методами.

Листинг 4.9. До

```
function handleInput(input: Input) {
  if (input === Input.LEFT)
    moveHorizontal(-1);
  else if (input === Input.RIGHT)
    moveHorizontal(1);
  else if (input === Input.UP)
    moveVertical(-1);
  else if (input === Input.DOWN)
    moveVertical(1);
}
```

Листинг 4.10. После (2/3)

```
function handleInput(input: Input2) {
  if (input.isLeft())
    moveHorizontal(-1);
  else if (input.isRight())
    moveHorizontal(1);
  else if (input.isUp())
    moveVertical(-1);
  else if (input.isDown())
    moveVertical(1);
}
```

Изменяем тип
для использования
интерфейса

Используем вместо проверок
равенства новые методы

- Исправим последние ошибки внесением изменений.

Листинг 4.11. До

```
Input.RIGHT
Input.LEFT
Input.UP
Input.DOWN
```

Листинг 4.12. После (3/3)

```
new Right()
new Left()
new Up()
new Down()
```


6. В завершение везде переименуем Input2 в Input. На этом этапе код будет выглядеть так.

Листинг 4.13. До

```

window.addEventListener("keydown", e =>
{
    if (e.key === LEFT_KEY
        || e.key === "a")
        inputs.push(Input.LEFT);
    else if (e.key === UP_KEY
        || e.key === "w")
        inputs.push(Input.UP);
    else if (e.key === RIGHT_KEY
        || e.key === "d")
        inputs.push(Input.RIGHT);
    else if (e.key === DOWN_KEY
        || e.key === "s")
        inputs.push(Input.DOWN);
});

function handleInput(input: Input) {
    if (input === Input.LEFT)
        moveHorizontal(-1);
    else if (input === Input.RIGHT)
        moveHorizontal(1);
    else if (input === Input.UP)
        moveVertical(-1);
    else if (input === Input.DOWN)
        moveVertical(1);
}

```

Листинг 4.14. После

```

window.addEventListener("keydown", e =>
{
    if (e.key === LEFT_KEY
        || e.key === "a")
        inputs.push(new Left());
    else if (e.key === UP_KEY
        || e.key === "w")
        inputs.push(new Up());
    else if (e.key === RIGHT_KEY
        || e.key === "d")
        inputs.push(new Right());
    else if (e.key === DOWN_KEY
        || e.key === "s")
        inputs.push(new Down());
});

function handleInput(input: Input) {
    if (input.isLeft())
        moveHorizontal(-1);
    else if (input.isRight())
        moveHorizontal(1);
    else if (input.isUp())
        moveVertical(-1);
    else if (input.isDown())
        moveVertical(1);
}

```

В шаблоне «Замена кода типа классами» включаем процесс создания перечислений в классы.

4.1.3. Шаблон рефакторинга «Замена кода типа классами»

Описание

Этот шаблон рефакторинга преобразует перечисление в интерфейс, при этом значения перечисления становятся классами. Подобное действие позволяет нам добавлять каждому значению свойства и локализовать функциональность, относящуюся к данному конкретному значению. Совместно с другим шаблоном рефакторинга, который рассмотрим следующим («Перемещение кода в классы», 4.1.5), это дает возможность вносить изменения путем добавления. Дело в том, что зачастую используются перечисления посредством `switch` или цепочек `else if`, разбросанных по всему приложению. Инструкция `switch` определяет, как каждое возможное значение перечисления должно обрабатываться в данном месте.

Когда мы трансформируем значения в классы, получаем возможность вместо этого сгруппировать функциональность, относящуюся к этому значению, без необходимости учитывать какие-либо другие значения перечисления. Этот процесс объединяет функциональность с данными. Он локализует функциональность относительно данных, то есть конкретного значения данных. Добавление нового значения в перечисление означает проверку логики, связанной с этим перечислением, во многих файлах, тогда как добавление нового класса, реализующего интерфейс, требует от нас создания методов только в этом файле — никакой модификации другого кода не требуется (конечно, пока нам не понадобится использовать этот новый класс).

Обратите внимание, что *код типа* также оформляется иначе, чем перечисления. Любой целочисленный тип или любой тип, поддерживающий проверку тождественности `===`, может выступать как код типа. Чаще всего используются `int` и `enum`. Вот пример подобного кода типа для размеров футболок.

Листинг 4.15. Начальный

```
const SMALL = 33;
const MEDIUM = 37;
const LARGE = 42;
```

В случае с `int` отслеживать использование кода типа сложнее, потому что при создании кода разработчик мог использовать число без ссылки на центральную константу. Поэтому всегда, встретив код типа, следует преобразовать его в перечисления. Только так получится применить этот шаблон рефакторинга безопасно.

Листинг 4.16. До

```
const SMALL = 33;
const MEDIUM = 37;
const LARGE = 42;
```

Листинг 4.17. После

```
enum TShirtSizes {
  SMALL = 33,
  MEDIUM = 37,
  LARGE = 42
}
```

Процесс

1. Вводим новый интерфейс с временным именем. Этот интерфейс должен содержать методы для каждого из значений перечисления.
2. Создаем классы, соответствующие каждому значению перечисления. Все методы из этого интерфейса, кроме одного, соответствующего классу, должны делать `return false`.
3. Переименовываем перечисление. В результате компилятор сообщает об ошибке везде, где оно используется.
4. Изменяем старое имя типа на временное и заменяем проверки тождественности новыми методами.

5. Заменяем оставшиеся ссылки значениями перечислений инстанцированием новых классов.
6. Когда ошибок больше нет, везде переименовываем интерфейс, заменяя его имя постоянным.

Пример

Рассмотрим небольшой пример с перечислением сигналов светофора и функцией для определения момента, когда можно начинать движение.

Листинг 4.18. Начальный

```
enum TrafficLight {
    RED, YELLOW, GREEN
}
const CYCLE = [TrafficLight.RED, TrafficLight.GREEN, TrafficLight.YELLOW];
function updateCarForLight(current: TrafficLight) {
    if (current === TrafficLight.RED)
        car.stop();
    else
        car.drive();
}
```

Следуя описанному процессу, мы делаем так.

1. Вводим новый интерфейс с временным именем. Этот интерфейс должен содержать методы для каждого значения перечисления.

Листинг 4.19. Новый интерфейс

```
interface TrafficLight2 {
    isRed(): boolean;
    isYellow(): boolean;
    isGreen(): boolean;
}
```

2. Создаем классы, соответствующие каждому значению перечисления. Все методы интерфейса, кроме одного, соответствующего классу, должны осуществлять `return false`.

Листинг 4.20. Новые классы

```
class Red implements TrafficLight2 {
    isRed() { return true; }
    isYellow() { return false; }
    isGreen() { return false; }
}
class Yellow implements TrafficLight2 {
    isRed() { return false; }
    isYellow() { return true; }
    isGreen() { return false; }
}
```

```
class Green implements TrafficLight2 {
    isRed() { return false; }
    isYellow() { return false; }
    isGreen() { return true; }
}
```

3. Переименовываем перечисление. В результате компилятор сообщает об ошибках во всех местах использования этого перечисления.

Листинг 4.21. До

```
enum TrafficLight {
    RED, YELLOW, GREEN
}
```

Листинг 4.22. После (1/4)

```
enum RawTrafficLight {
    RED, YELLOW, GREEN
}
```

4. Изменяем имя типов со старого на временное и заменяем проверки тождественности новыми методами.

Листинг 4.23. До

```
function updateCarForLight(
    current: TrafficLight)
{
    if (current === TrafficLight.RED)
        car.stop();
    else
        car.drive();
}
```

Листинг 4.24. После (2/4)

```
function updateCarForLight(
    current: TrafficLight2)
{
    if (current.isRed())
        car.stop();
    else
        car.drive();
}
```

5. Вместо оставшихся ссылок на значения перечисления используем инстанцированные новые классы.

Листинг 4.25. До

```
const CYCLE = [
    TrafficLight.RED,
    TrafficLight.GREEN,
    TrafficLight.YELLOW
];
```

Листинг 4.26. После (3/4)

```
const CYCLE = [
    new Red(),
    new Green(),
    new Yellow()
];
```

6. В завершение, когда ошибок уже нет, везде даем интерфейсу постоянное имя.

Листинг 4.27. До

```
interface TrafficLight2 {
    // ...
}
```

Листинг 4.28. После (4/4)

```
interface TrafficLight {
    // ...
}
```

Этот шаблон рефакторинга сам по себе не вносит кардинальных улучшений в код, но создает основу для существенных улучшений в дальнейшем. Наличие методов `is` для всех значений тоже считается запахом, так что пока мы заменили

один запах другим. Но эти методы можно затем обработать по одному, тогда как значения перечисления были тесно связаны между собой и обрабатывать их по отдельности было невозможно. Важно отметить, что большинство методов `is` являются временными и существуют недолго — в примере мы избавимся от некоторых из них в текущей главе и от многих других в главе 5.

Дополнительные материалы

Этот шаблон рефакторинга также можно найти в книге Мартина Фаулера «Рефакторинг».

4.1.4. Перемещение кода в классы

Вот теперь подошло время магии. Все условия в `handleInput` связаны с параметром `input`, то есть код должен находиться в этом классе. К счастью, реализовать это несложно. Для этого нужно сделать следующее.

1. Скопировать `handleInput` и вставить во все классы. Удалить `function`, потому что теперь это метод, и заменить параметр `input` на `this`. Имя у него по-прежнему неверное, поэтому компилятор продолжает выдавать ошибки.

Листинг 4.29. После

```
class Right implements Input {
    // ...
    handleInput() {
        if (this.isLeft())
            moveHorizontal(-1);
        else if (this.isRight())
            moveHorizontal(1);
        else if (this.isUp())
            moveVertical(-1);
        else if (this.isDown())
            moveVertical(1);
    }
}
```

Удаляем function и параметр

Изменяем input на this

2. Скопировать сигнатуру метода в интерфейс `Input` и дать ему имя, немного отличающееся от исходного `handleInput`. В данном случае мы уже находимся в `Input`, поэтому нет смысла писать его дважды.

Листинг 4.30. Новый интерфейс

```
interface Input {
    // ...
    handle(): void;
}
```

3. Пройти по методам `handleInput` во всех четырех классах. Поскольку процесс для всех классов будет одинаков, то я продемонстрирую его на одном:

1) встроить возвращаемые значения `isLeft`, `isRight`, `isUp` и `isDown`;

Листинг 4.31. До

```
class Right implements Input {
    // ...
    handleInput() {
        if (this.isLeft())
            moveHorizontal(-1);
        else if (this.isRight())
            moveHorizontal(1);
        else if (this.isUp())
            moveVertical(-1);
        else if (this.isDown())
            moveVertical(1);
    }
}
```

Листинг 4.32. После (1/4)

```
class Right implements Input {
    // ...
    handleInput() {
        if (false)
            moveHorizontal(-1);
        else if (true)
            moveHorizontal(1);
        else if (false)
            moveVertical(-1);
        else if (false)
            moveVertical(1);
    }
}
```



После встраивания
методов is

2) удалить все `if (false) { ... }` и часть `if` условия `if (true)`;

Листинг 4.33. До

```
class Right implements Input {
    // ...
    handleInput() {
        if (false)
            moveHorizontal(-1);
        else if (true)
            moveHorizontal(1);
        else if (false)
            moveVertical(-1);
        else if (false)
            moveVertical(1);
    }
}
```

Листинг 4.34. После (2/4)

```
class Right implements Input {
    // ...
    handleInput() {
        moveHorizontal(1);
    }
}
```

3) изменить имя на `handle`, чтобы указать, что с этим методом мы закончили. Теперь здесь нет ошибок и компилятор должен принять его.

Листинг 4.35. До

```
class Right implements Input {
    // ...
    handleInput() { moveHorizontal(1); }
}
```

Листинг 4.36. После (3/4)

```
class Right implements Input {
    // ...
    handle() { moveHorizontal(1); }
}
```

4. Заменить тело `handleInput` на вызов нового метода.

Листинг 4.37. До

```
function handleInput(input: Input) {
  if (input.isLeft())
    moveHorizontal(-1);
  else if (input.isRight())
    moveHorizontal(1);
  else if (input.isUp())
    moveVertical(-1);
  else if (input.isDown())
    moveVertical(1);
}
```

Листинг 4.38. После (4/4)

```
function handleInput(input: Input) {
  input.handle();
}
```

Пройдя весь описанный процесс, мы достигаем нашей великой цели: получаем то улучшение кода, к которому стремились, — все `if` исчезли и получившиеся методы легко вписываются в пять строк.

Листинг 4.39. До

```
function handleInput(input: Input) {
  if (input.isLeft())
    moveHorizontal(-1);
  else if (input.isRight())
    moveHorizontal(1);
  else if (input.isUp())
    moveVertical(-1);
  else if (input.isDown())
    moveVertical(1);
}
```

Листинг 4.40. После

```
function handleInput(input: Input) {
  input.handle();
}

interface Input {
  // ...
  handle(): void;
}

class Left implements Input {
  // ...
  handle() { moveHorizontal(-1); }
}

class Right implements Input {
  // ...
  handle() { moveHorizontal(1); }
}

class Up implements Input {
  // ...
  handle() { moveVertical(-1); }
}

class Down implements Input {
  // ...
  handle() { moveVertical(1); }
}
```

Это мой любимый шаблон рефакторинга: он настолько структурирован, что его можно выполнять с минимальной мыслительной нагрузкой, а в итоге получать очень красивый код. Я называю его «Перемещение кода в классы».

4.1.5. Шаблон рефакторинга «Перемещение кода в классы»

Описание

Этот шаблон рефакторинга является естественным продолжением «Замены кода типа классами», поскольку перемещает в классы функциональность. В результате инструкции `if` зачастую удаляются и функциональность оказывается ближе к данным. Как уже говорилось, это помогает локализовать инварианты, так как функциональность, связанная с конкретным значением, перемещается в класс, соответствующий этому значению.

В самой простой форме шаблона мы всегда предполагаем, что перемещать в классы надлежит весь метод. Это не составит сложности, потому что начинаем всегда с извлечения методов. Код можно переместить и без его предварительного извлечения, но это потребует большей внимательности, потому что возрастет риск что-нибудь нарушить.

Процесс

1. Скопировать исходную функцию и вставить ее во все классы. Удалить `function`, так как теперь это метод. Заменить контекст на `this` и удалить неиспользуемые параметры. У метода по-прежнему неверное имя, поэтому компилятор продолжит сообщать об ошибках.
2. Скопировать сигнатуру метода в целевой интерфейс. Дать ему имя, немного отличающееся от исходного метода.
3. Пройти по новому методу во всех классах:
 - 1) встроить методы, которые возвращают константное выражение;
 - 2) выполнить все возможные вычисления заранее, что обычно подразумевает удаление `if (true)` и `if (false) { ... }`, но также может потребовать и предварительного упрощения условий (например, `false || true` становится `true`);
 - 3) изменить имя на соответствующее, тем самым указав, что с этим методом мы закончили. Теперь компилятор должен его принять.
4. Заменить тело исходной функции вызовом нового метода.

Пример

Поскольку этот шаблон рефакторинга тесно связан с «Заменой кода типа классами», продолжим предыдущий пример со светофором.

Листинг 4.41. Начальный

```
interface TrafficLight {
    isRed(): boolean;
```



```

    isYellow(): boolean;
    isGreen(): boolean;
}
class Red implements TrafficLight {
    isRed() { return true; }
    isYellow() { return false; }
    isGreen() { return false; }
}
class Yellow implements TrafficLight {
    isRed() { return false; }
    isYellow() { return true; }
    isGreen() { return false; }
}
class Green implements TrafficLight {
    isRed() { return false; }
    isYellow() { return false; }
    isGreen() { return true; }
}
function updateCarForLight(current: TrafficLight) {
    if (current.isRed())
        car.stop();
    else
        car.drive();
}

```

Следуя процессу, мы делаем так.

1. Создаем в целевом интерфейсе новый метод. Даем ему имя, немного отличающееся от исходного.

Листинг 4.42. Новый метод

```

interface TrafficLight {
    // ...
    updateCar(): void;
}

```

2. Копируем исходную функцию и вставляем ее во все классы. Удаляем `function`, поскольку теперь это метод. Заменяем контекст на `this` и удаляем неиспользуемые параметры. Метод по-прежнему имеет неверное имя, поэтому компилятор продолжает сообщать об ошибках.

Листинг 4.43. Дублирование метода в классах

```

class Red implements TrafficLight {
    // ...
    updateCarForLight() {
        if (this.isRed())
            car.stop();
        else
            car.drive();
    }
}

```

```

class Yellow implements TrafficLight {
    // ...
    updateCarForLight() {
        if (this.isRed())
            car.stop();
        else
            car.drive();
    }
}
class Green implements TrafficLight {
    // ...
    updateCarForLight() {
        if (this.isRed())
            car.stop();
        else
            car.drive();
    }
}

```

3. Проходим через новый метод во всех классах:

- 1) встраиваем методы, возвращающие константное выражение;
- 2) выполняем все возможные вычисления заранее;

Листинг 4.44. До

```

class Red implements TrafficLight {
    // ...
    updateCarForLight() {
        if (this.isRed())
            car.stop();
        else
            car.drive();
    }
}
class Yellow implements TrafficLight {
    // ...
    updateCarForLight() {
        if (this.isRed())
            car.stop();
        else
            car.drive();
    }
}
class Green implements TrafficLight {
    // ...
    updateCarForLight() {
        if (this.isRed())
            car.stop();
        else
            car.drive();
    }
}

```

Листинг 4.45. После (1/4)

```

class Red implements TrafficLight {
    // ...
    updateCarForLight() {
        if (true)
            car.stop();
        else
            car.drive();
    }
}
class Yellow implements TrafficLight {
    // ...
    updateCarForLight() {
        if (false)
            car.stop();
        else
            car.drive();
    }
}
class Green implements TrafficLight {
    // ...
    updateCarForLight() {
        if (false)
            car.stop();
        else
            car.drive();
    }
}

```

Листинг 4.46. До

```

class Red implements TrafficLight {
    // ...
    updateCarForLight() {
        if (true)
            car.stop();
        else
            car.drive();
    }
}
class Yellow implements
    TrafficLight {
    // ...
    updateCarForLight() {
        if (false)
            car.stop();
        else
            car.drive();
    }
}
class Green implements
    TrafficLight {
    // ...
    updateCarForLight() {
        if (false)
            car.stop();
        else
            car.drive();
    }
}

```

Листинг 4.47. После (2/4)

```

class Red implements TrafficLight {
    // ...
    updateCarForLight() {

        car.stop();

    }
}
class Yellow implements
    TrafficLight {
    // ...
    updateCarForLight() {

        car.drive();

    }
}

class Green implements
    TrafficLight {
    // ...
    updateCarForLight() {

        car.drive();

    }
}

```

3) возвращаем методу подходящее имя, указывая таким образом, что закончили с ним.

Листинг 4.48. До

```

class Red implements TrafficLight {
    // ...
    updateCarForLight() { car.stop(); }
}
class Yellow implements
    TrafficLight {
    // ...
    updateCarForLight() { car.drive(); }
}
class Green implements TrafficLight {
    // ...
    updateCarForLight() { car.drive(); }
}

```

Листинг 4.49. После (3/4)

```

class Red implements TrafficLight {
    // ...
    updateCar() { car.stop(); }
}
class Yellow implements
    TrafficLight {
    // ...
    updateCar() { car.drive(); }
}
class Green implements TrafficLight {
    // ...
    updateCar() { car.drive(); }
}

```

4. Заменяем тело исходной функции вызовом нового метода.

Листинг 4.50. До

```
function updateCarForLight(
  current: TrafficLight)
{
  if (current.isRed())
    car.stop();
  else car.drive();
}
```

Листинг 4.51. После (4/4)

```
function updateCarForLight(
  current: TrafficLight)
{
  current.updateCar();
}
```

Ранее мы отмечали, что если не убрать методы `is`, то они становятся запахами, тем более отратно, что в нашем миниатюрном примере они не нужны. Это дополнительное преимущество данного шаблона рефакторинга.

Дополнительные материалы

В данном простом виде этот рефакторинг, по сути, аналогичен «Перемещению метода», описываемому Мартином Фаулером. Тем не менее я думаю, что наша слегка измененная интерпретация лучше передает его намерение и демонстрирует эффективность.

4.1.6. Встраивание избыточного метода

На этом этапе можно наблюдать еще один удивительный эффект рефакторинга. Несмотря на то что мы только что ввели функцию `handleInput`, это не значит, что она должна остаться. Рефакторинг зачастую цикличен и подразумевает добавление необходимых для его выполнения компонентов с последующим их удалением. Так что никогда не бойтесь добавления кода.

Когда мы вводили `handleInput`, у нее было ясное предназначение. Однако теперь она больше не повышает читаемость программы, лишь занимает место, поэтому можно ее удалить. Для этого необходимо сделать следующее.

1. Изменить имя метода на `handleInput2`. В результате возникнут ошибки компиляции во всех местах его использования.
2. Скопировать тело `input.handle()`; и обратить внимание, что `input` является параметром.
3. Заменить вызов ее тела там, где эта функция используется, а это единственное место.

Листинг 4.52. До

```
handleInput(current);
```

Листинг 4.53. После

```
current.handle();
```

После этого и после переименования `current` в `input` обновленная `handleInputs` выглядит так.

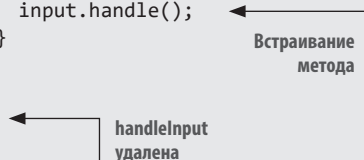
Листинг 4.54. До

```
function handleInputs() {
  while (inputs.length > 0) {
    let current = inputs.pop();
    handleInput(current);
  }
}

function handleInput(input: Input) {
  input.handle();
}
```

Листинг 4.55. После

```
function handleInputs() {
  while (inputs.length > 0) {
    let input = inputs.pop();
    input.handle();
  }
}
```



Этот шаблон рефакторинга — «Встраивание метода» — является точной противоположностью «Извлечения метода» (3.2.1), который мы изучали в главе 3.

4.1.7. Шаблон рефакторинга «Встраивание метода»

Описание

Два ключевых момента этой книги — добавление кода (обычно для поддержки классов) и удаление кода. Данный шаблон рефакторинга относится к последнему: удаляет методы, которые больше не повышают читаемости программы. Делает он это путем перемещения кода из метода во все точки вызова. В результате метод становится неиспользуемым и может быть безболезненно удален.

Обратите внимание, что подразумевается строгое различие между встраиванием методов и действием шаблона рефакторинга «Встраивание метода». В предыдущих разделах мы встраивали методы `is` в процессе перемещения кода в классы, после чего использовали «Встраивание метода» для удаления исходной функции. Когда мы встраиваем методы (без акцента на шаблоне как таковом), проводим это не в каждой точке вызова, поэтому исходный метод сохраняем. Обычно цель этого — упрощение точки вызова. Когда мы встраиваем метод (с акцентом на использовании шаблона), делаем это в каждой точке вызова, после чего сам метод удаляем.

В данной книге мы иногда применяем шаблон, когда методы содержат всего одну строку. Причина этого в нашем строгом ограничении в пять строк. Встраивание метода с одной строкой не может нарушить это правило. Можно также применить этот шаблон рефакторинга к методам, содержащим более одной строки.

При этом нужно проверить, не является ли метод излишне сложным для встраивания. Приведенный ниже пример дает абсолютное значение числа.

Мы оптимизировали его для повышения производительности, поэтому веток в нем нет и состоит он из одной строки. Этот метод достигает своей цели за счет низкоуровневых операций, поэтому его добавление повышает читаемость и встраивать его не стоит. В данном случае его встраивание будет противодействовать запаху «Операции должны находиться на одном уровне абстракции», из которого происходит правило «Вызов или передача» (3.3.1).

Листинг 4.56. Метод, который не нужно встраивать

```
const NUMBER_BITS = 32;
function absolute(x: number) {
  return (x ^ x >> NUMBER_BITS-1) - (x >> NUMBER_BITS-1);
}
```

Процесс

1. Изменить имя метода на временное. В результате в местах использования этой функции возникнут ошибки.
2. Скопировать тело метода и обратить внимание на его параметры.
3. Везде, где компилятор сообщает об ошибке, заменить вызов скопированным телом и отобразить аргументы в параметры.
4. Когда компиляция будет выполняться без ошибок и станет очевидно, что исходный метод не используется, удалить его.

Пример

Поскольку пример кода игры мы уже видели, используем для проверки код из другой области. Представим, что мы разделили две части банковской транзакции: снятие денежных средств с одного счета и их последующее внесение на другой. Это означает, что стало возможно случайно внести средства, не сняв их, если вызвать не тот метод. Чтобы исправить ситуацию, мы решаем объединить эти два метода.

Листинг 4.57. Начальный

```
function deposit(to: string, amount: number) {
  let accountId = database.find(to);
  database.updateOne(accountId, { $inc: { balance: amount } });
}

function transfer(from: string, to: string, amount: number) {
  deposit(from, -amount);
  deposit(to, amount);
}
```

В TYPESCRIPT...

Символ \$ рассматривается как любой другой знак, аналогично, например, _. Таким образом, он может являться частью имени и особой смысловой нагрузки не несет. \$inc может с тем же успехом быть do_inc.

Следуя процессу, мы делаем следующее.

1. Изменяем имя метода на временное. В результате компилятор выдает ошибку во всех местах его использования.

Листинг 4.58. До

```
function deposit(to: string,
    amount: number) {
    // ...
}
```

Листинг 4.59. После (1/2)

```
function deposit2(to: string,
    amount: number) {
    // ...
}
```

2. Копируем тело метода и обращаем внимание на его параметры.
3. Везде, где компилятор выдает ошибку, заменяем вызов скопированным телом и отображаем аргументы в параметры.

Листинг 4.60. До

```
function transfer(
    from: string,
    to: string,
    amount: number)
{
    deposit(from, -amount);

    deposit(to, amount);
}
```

Листинг 4.61. После (2/2)

```
function transfer(
    from: string,
    to: string,
    amount: number)
{
    let fromAccountId = database.find(from);
    database.updateOne(fromAccountId,
        { $inc: { balance: -amount } });
    let toAccountId = database.find(to);
    database.updateOne(toAccountId,
        { $inc: { balance: amount } });
}
```

4. Когда компиляция проходит без ошибок, понимаем, что исходный метод не используется. Удаляем его.

Теперь деньги не смогут появляться из ниоткуда по причине использования не того фрагмента кода. Конечно, можно усомниться, что подобное повторение кода является хорошей идеей. Что ж, в главе 6 я покажу и другое решение, где используется инкапсуляция.

Дополнительные материалы

Этот шаблон рефакторинга можно найти в книге Мартина Фаулера «Рефакторинг».

4.2. РЕФАКТОРИНГ БОЛЬШОЙ ИНСТРУКЦИИ IF

Повторим тот же процесс, но теперь с методом подлиннее: `drawMap`.

Листинг 4.62. Начальный

```
function drawMap(g: CanvasRenderingContext2D) {
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length; x++) {
      if (map[y][x] === Tile.FLUX)
        g.fillStyle = "#ccffcc";
      else if (map[y][x] === Tile.UNBREAKABLE)
        g.fillStyle = "#999999";
      else if (map[y][x] === Tile.STONE || map[y][x] === Tile.FALLING_STONE)
        g.fillStyle = "#0000cc";
      else if (map[y][x] === Tile.BOX || map[y][x] === Tile.FALLING_BOX)
        g.fillStyle = "#8b4513";
      else if (map[y][x] === Tile.KEY1 || map[y][x] === Tile.LOCK1)
        g.fillStyle = "#ffcc00";
      else if (map[y][x] === Tile.KEY2 || map[y][x] === Tile.LOCK2)
        g.fillStyle = "#00ccff";

      if (map[y][x] !== Tile.AIR && map[y][x] !== Tile.PLAYER)
        g.fillRect(x * TILE_SIZE, y * TILE_SIZE, TILE_SIZE, TILE_SIZE);
    }
  }
}
```

Сразу отметим серьезное нарушение правила «if только в начале» (3.5.1) из предыдущей главы: здесь у нас длинная цепочка `else if` прямо в середине кода. Поэтому первым делом извлекаем эту цепочку в отдельный метод.

Листинг 4.63. После «Извлечения метода» (3.2.1)

```
function drawMap(g: CanvasRenderingContext2D) {
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length; x++) {
      colorOfTile(g, x, y);
      if (map[y][x] !== Tile.AIR && map[y][x] !== Tile.PLAYER)
        g.fillRect(x * TILE_SIZE, y * TILE_SIZE, TILE_SIZE, TILE_SIZE);
    }
  }
}

function colorOfTile(g: CanvasRenderingContext2D, x: number, y: number) {
  if (map[y][x] === Tile.FLUX)
    g.fillStyle = "#ccffcc";
  else if (map[y][x] === Tile.UNBREAKABLE)
    g.fillStyle = "#999999";
  else if (map[y][x] === Tile.STONE || map[y][x] === Tile.FALLING_STONE)
    g.fillStyle = "#0000cc";
  else if (map[y][x] === Tile.BOX || map[y][x] === Tile.FALLING_BOX)
    g.fillStyle = "#8b4513";
  else if (map[y][x] === Tile.KEY1 || map[y][x] === Tile.LOCK1)
```

Извлеченный
метод и вызов


```

    g.fillStyle = "#ffcc00";
    else if (map[y][x] === Tile.KEY2 || map[y][x] === Tile.LOCK2)
        g.fillStyle = "#00ccff";
}

```

Сейчас `drawMap` вписывается в наше правило «Пять строк», поэтому переходим к `colorOfTile`. Заметно, что `colorOfTile` нарушает правило «Никогда не использовать `if` с `else`». Как и ранее, для решения этой проблемы заменяем перечисление `Tile` интерфейсом `Tile`.

1. Вводим новый интерфейс с временным именем `Tile2`, который содержит методы для всех значений нашего перечисления.

Листинг 4.64. Новый интерфейс

```

interface Tile2 {
    isFlux(): boolean;
    isUnbreakable(): boolean;
    isStone(): boolean;
    // ...
}

```

Методы для всех значений перечисления

2. Создаем классы, соответствующие каждому значению перечисления.

Листинг 4.65. Новые классы

```

class Flux implements Tile2 {
    isFlux() { return true; }
    isUnbreakable() { return false; }
    isStone() { return false; }
    // ...
}
class Unbreakable implements Tile2 { ... }
class Stone implements Tile2 { ... }
// ...

```

Аналогичные классы для оставшихся значений перечисления

3. Переименовываем перечисление в `RawTile`, и компилятор индикацией ошибок показывает нам все места его использования.

Листинг 4.66. До

```

enum Tile {
    AIR,
    FLUX,
    UNBREAKABLE,
    PLAYER,
    STONE, FALLING_STONE,
    BOX, FALLING_BOX,
    KEY1, LOCK1,
    KEY2, LOCK2
}

```

Листинг 4.67. После (1/2)

```

enum RawTile {
    AIR,
    FLUX,
    UNBREAKABLE,
    PLAYER,
    STONE, FALLING_STONE,
    BOX, FALLING_BOX,
    KEY1, LOCK1,
    KEY2, LOCK2
}

```

Изменение имени для получения ошибок компиляции

4. Заменяем проверки тождественности новыми методами. Это изменение нужно внести во многих местах по всему приложению, здесь же приводится только `colorOfTile`.

Листинг 4.68. До

```
function colorOfTile(
  g: CanvasRenderingContext2D,
  x: number, y: number)
{
  if (map[y][x] === Tile.FLUX)
    g.fillStyle = "#ccffcc";
  else if (map[y][x] === Tile.UNBREAKABLE)
    g.fillStyle = "#999999";
  else if (map[y][x] === Tile.STONE
    || map[y][x] === Tile.FALLING_STONE)
    g.fillStyle = "#0000cc";
  else if (map[y][x] === Tile.BOX
    || map[y][x] === Tile.FALLING_BOX)
    g.fillStyle = "#8b4513";
  else if (map[y][x] === Tile.KEY1
    || map[y][x] === Tile.LOCK1)
    g.fillStyle = "#ffcc00";
  else if (map[y][x] === Tile.KEY2
    || map[y][x] === Tile.LOCK2)
    g.fillStyle = "#00ccff";
}
```

Листинг 4.69. После (2/2)

```
function colorOfTile(
  g: CanvasRenderingContext2D,
  x: number, y: number)
{
  if (map[y][x].isFlux())
    g.fillStyle = "#ccffcc";
  else if (map[y][x].isUnbreakable())
    g.fillStyle = "#999999";
  else if (map[y][x].isStone()
    || map[y][x].isFallingStone())
    g.fillStyle = "#0000cc";
  else if (map[y][x].isBox()
    || map[y][x].isFallingBox())
    g.fillStyle = "#8b4513";
  else if (map[y][x].isKey1()
    || map[y][x].isLock1())
    g.fillStyle = "#ffcc00";
  else if (map[y][x].isKey2()
    || map[y][x].isLock2())
    g.fillStyle = "#00ccff";
}
```

Используем
новые методы
вместо проверок
тождественности

ПРЕДУПРЕЖДЕНИЕ

Позабойтесь, чтобы `map[y][x] === Tile.FLUX` стала `map[y][x].isFlux()`, а `map[y][x] !== Tile.AIR` стала `!map[y][x].isAir()`. Обратите внимание на !.

5. Заменяем `Tile.FLUX` на `new Flux()`, `Tile.AIR` на `new Air()` и т. д.

В прошлый раз в аналогичной ситуации у нас ошибок не было и можно было переименовать временный `Tile2` в постоянный `Tile`. Но теперь ситуация изменилась: у нас по-прежнему есть два места с ошибками, показывающими, что мы используем `Tile`. Именно поэтому назначим временное имя. В противном случае мы бы наверняка не увидели проблемы в `remove` и предположили, что она работает, — но это не так.

Листинг 4.70. Последние две ошибки

```
let map: Tile[][] = [
  [2, 2, 2, 2, 2, 2, 2, 2],
  [2, 3, 0, 1, 1, 2, 0, 2],
  [2, 4, 2, 6, 1, 2, 0, 2],
  [2, 8, 4, 1, 1, 2, 0, 2],
  [2, 4, 1, 1, 1, 9, 0, 2],
  [2, 2, 2, 2, 2, 2, 2, 2],
];

function remove(tile: Tile) {
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length; x++) {
      if (map[y][x] === tile) {
```

Ошибки ввиду
того, что мы
ссылаемся на Tile

```

        map[y][x] = new Air();
    }
}
}
}

```

Каждая из этих двух ошибок требует особого подхода, поэтому разберем их по очереди.

4.2.1. Устранение обобщенности

Проблема `remove` в том, что при получении типа клетки она удаляет его по всей карте, при этом не осуществляет проверок относительно конкретного экземпляра `tile`. Вместо этого проверка выполняется на сходство экземпляров.

Листинг 4.71. Начальный

```

function remove(tile: Tile) {
    for (let y = 0; y < map.length; y++) {
        for (let x = 0; x < map[y].length; x++) {
            if (map[y][x] === tile) {
                map[y][x] = new Air();
            }
        }
    }
}

```

Иначе говоря, проблема в излишней обобщенности `remove`. Она может удалять любой тип клетки. Эта обобщенность делает ее менее гибкой и усложняет изменение. Следовательно, предпочтительнее задействовать специализацию: создать менее универсальную версию и переключиться на ее использование.

Прежде чем мы сможем создать специализированную версию, необходимо понять, как она применяется. Опять же нужно сделать параметр менее общим, следовательно, надо определить, какие аргументы передаются ему сейчас. Используем уже знакомый процесс и изменяем имя `remove` на временное, `remove2`. Таким образом, выясняем, что `remove` используется в четырех местах.

Листинг 4.72. До

```

/// ...
remove(new Lock1());
/// ...
remove(new Lock2());
/// ...
remove(new Lock1());
/// ...
remove(new Lock2());
/// ...

```

Несмотря на то что `remove` поддерживает удаление любого типа, удаляет она только `Lock1` или `Lock2`. Этим можно воспользоваться.

1. Повторяем `remove2`.

Листинг 4.73. До

```
function remove2(tile: Tile) {
  // ...
}
```

Листинг 4.74. После (1/4)

```
function remove2(tile: Tile) {
  // ...
}
function remove2(tile: Tile) {
  // ...
}
```

← Функции имеют одинаковое тело

2. Переименовываем одну из них в `removeLock1`, удаляем ее параметр и временно заменяем `=== tile` на `=== Tile.LOCK1`. Делаем именно это, несмотря на то что переименовали `Tile` в `RawTile`, поскольку так получим код, идентичный тому, что мы обрабатывали ранее.

Листинг 4.75. До

```
function remove2(tile: Tile) {
  for (let y = 0; y < map.length; y++)
    for (let x = 0; x < map[y].length; x++)
      if (map[y][x] === tile)
        map[y][x] = new Air();
}
```

Листинг 4.76. После (2/4)

```
function removeLock1() {
  for (let y = 0; y < map.length; y++)
    for (let x = 0; x < map[y].length; x++)
      if (map[y][x] === Tile.LOCK1)
        map[y][x] = new Air();
}
```

← Заменяем tile на Tile.LOCK1
Переименовываем и удаляем параметр

3. Такой тип равенства мы удалять умеем. Как уже делали ранее, заменяем его вызовом метода.

Листинг 4.77. До

```
function removeLock1() {
  for (let y = 0; y < map.length; y++)
    for (let x = 0; x < map[y].length; x++)
      if (map[y][x] === Tile.LOCK1)
        map[y][x] = new Air();
}
```

Листинг 4.78. После (3/4)

```
function removeLock1() {
  for (let y = 0; y < map.length; y++)
    for (let x = 0; x < map[y].length; x++)
      if (map[y][x].isLock1())
        map[y][x] = new Air();
}
```

← Используем метод вместо проверки равенства

4. В этой функции ошибок больше нет, значит, можно заменить старые вызовы новыми.

Листинг 4.79. До

```
remove(new Lock1());
```

Листинг 4.80. После (4/4)

```
removeLock1();
```

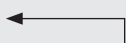
То же самое сделаем для `removeLock2`. После этого и `removeLock1`, и `removeLock2` останутся без ошибок. В `remove2` ошибка сохраняется, но эта функция больше не вызывается, поэтому мы ее просто удаляем. Итак, мы произвели следующее изменение.

Листинг 4.81. До

```
function remove(tile: Tile) {
  for (let y = 0; y < map.length; y++)
    for (let x = 0; x < map[y].length; x++)
      if (map[y][x] === tile)
        map[y][x] = new Air();
}
```

Листинг 4.82. После

```
function removeLock1() {
  for (let y = 0; y < map.length; y++)
    for (let x = 0; x < map[y].length; x++)
      if (map[y][x].isLock1())
        map[y][x] = new Air();
}
function removeLock2() {
  for (let y = 0; y < map.length; y++)
    for (let x = 0; x < map[y].length; x++)
      if (map[y][x].isLock2())
        map[y][x] = new Air();
}
```


 Исходная remove удалена

Рассмотренный процесс введения менее обобщенных версий функции называется специализацией метода.

4.2.2. Шаблон рефакторинга «Специализация метода»

Описание

Этот тип рефакторинга более загадочный, потому что идет вразрез с интуицией многих программистов. Последние обладают естественной тягой к обобщению и повторному использованию кода, но это оказывается чревато проблемами, так как в результате размываются границы ответственности и в готовой программе вызов обобщенного кода может происходить из разных мест. Данный шаблон рефакторинга устраняет эти эффекты. Более специализированные методы вызываются из меньшего числа точек и раньше, чем обобщенные, могут стать неиспользуемыми, что позволит их удалить.

Процесс

1. Продублировать метод, который нужно специализировать.
2. Присвоить одному из методов-копий новое постоянное имя и удалить (или заменить) параметр, используемый в качестве основы специализации.
3. Соответствующим образом исправить метод, чтобы исключить в нем ошибки.
4. Переключиться с использования старых вызовов на новые.

Пример

Представьте, что мы реализуем игру в шахматы. В составе нашего модуля проверки верности ходов нами разработано отличное обобщенное выражение, оценивающее, соответствует ли ход правилам игры и продвижения фигуры.

Листинг 4.83. Начальный

```
function canMove(start: Tile, end: Tile, dx: number, dy: number) {
    return dx * abs(start.x - end.x) === dy * abs(start.y - end.y)
    || dy * abs(start.x - end.x) === dx * abs(start.y - end.y);
}

/// ...
if (canMove(start, end, 1, 0)) // Rook
/// ...
if (canMove(start, end, 1, 1)) // Bishop
/// ...
if (canMove(start, end, 1, 2)) // Knight
/// ...
```

Следуя описанному процессу, мы делаем так.

1. Дублируем метод, который хотим специализировать.

Листинг 4.84. До

```
function canMove(start: Tile, end: Tile,
    dx: number, dy: number)
{
    return dx * abs(start.x - end.x)
    === dy * abs(start.y - end.y)
    || dy * abs(start.x - end.x)
    === dx * abs(start.y - end.y);
}
```

Листинг 4.85. После (1/4)

```
function canMove(start: Tile, end: Tile,
    dx: number, dy: number)
{
    return dx * abs(start.x - end.x)
    === dy * abs(start.y - end.y)
    || dy * abs(start.x - end.x)
    === dx * abs(start.y - end.y);
}

function canMove(start: Tile, end: Tile,
    dx: number, dy: number)
{
    return dx * abs(start.x - end.x)
    === dy * abs(start.y - end.y)
    || dy * abs(start.x - end.x)
    === dx * abs(start.y - end.y);
}
```

2. Даем одному из методов-копий новое постоянное имя и удаляем (или заменяем) параметры, используемые в качестве основы специализации.

Листинг 4.86. До

```
function canMove(start: Tile, end: Tile,
    dx: number, dy: number)
{
    return dx * abs(start.x - end.x)
    === dy * abs(start.y - end.y)
    || dy * abs(start.x - end.x)
    === dx * abs(start.y - end.y);
}
```

Листинг 4.87. После (2/4)

```
function rookCanMove(
    start: Tile, end: Tile)
{
    return 1 * abs(start.x - end.x)
    === 0 * abs(start.y - end.y)
    || 0 * abs(start.x - end.x)
    === 1 * abs(start.y - end.y);
}
```

3. Соответствующим образом корректируем метод, чтобы в нем не было ошибок. Поскольку ошибок не остается, мы просто вносим упрощение.

Листинг 4.88. До

```
function rookCanMove(
  start: Tile, end: Tile)
{
  return 1 * abs(start.x - end.x)
    === 0 * abs(start.y - end.y)
    || 0 * abs(start.x - end.x)
    === 1 * abs(start.y - end.y);
}
```

Листинг 4.89. После (3/4)

```
function rookCanMove(
  start: Tile, end: Tile)
{
  return abs(start.x - end.x)
    === 0
    || 0
    === abs(start.y - end.y);
}
```

4. Переключаемся с использования старых вызовов на новые.

Листинг 4.90. До

```
if (canMove(start, end, 1, 0)) // Пешка
```

Листинг 4.91. После (4/4)

```
if (rookCanMove(start, end))
```

Обратите внимание, что комментарий больше не нужен. При этом `rookCanMove` еще и легче понять: пешка может сделать ход, если изменение по оси `x` или `y` равно нулю. Можно даже удалить часть `abs`, еще больше упростив выражение.

Я предоставляю вам возможность самостоятельно провести аналогичный рефакторинг для остальных фигур в исходном коде. Являются ли их методы настолько же понятными, как этот?

Дополнительные материалы

Насколько я знаю, процесс для этого шаблона рефакторинга проведен и описан здесь впервые, хотя Джонатан Блоу уже обсуждал преимущества специализации методов по сравнению с их обобщенными альтернативами в своей речи «Как писать независимые игры» на конференции в UC Berkeley Computer Science Undergraduate Association 2011.

4.2.3. Допускается только одна инструкция `switch`

Остается всего одна ошибка: мы создаем нашу карту, используя индексы перечисления, но этот прием больше не работает. Подобные индексы обычно применяются для хранения элементов в базах данных или файлах. В случае с игрой логично хранить уровни в файлах, используя индексы, поскольку их проще упорядочить в последовательность (сериализовать), чем объекты.

На практике же зачастую нет возможности изменить существующие внешние данные для проведения рефакторинга. Поэтому вместо изменения всей карты лучше создать новую функцию, которая переведет нас от индексов перечислений в новые классы. К счастью, реализовать это легко.

Листинг 4.92. Введение transformTile

```
let rawMap: RawTile[][] = [ [2, 2, 2, 2, 2, 2, 2, 2],
    [2, 3, 0, 1, 1, 2, 0, 2],
    [2, 4, 2, 6, 1, 2, 0, 2],
    [2, 8, 4, 1, 1, 2, 0, 2],
    [2, 4, 1, 1, 1, 9, 0, 2],

    [2, 2, 2, 2, 2, 2, 2, 2],
];
let map: Tile2[][];
function assertExhausted(x: never): never {
    throw new Error("Unexpected object: " + x);
}
function transformTile(tile: RawTile) {
    switch (tile) {
        case RawTile.AIR: return new Air();
        case RawTile.PLAYER: return new Player();
        case RawTile.UNBREAKABLE: return new Unbreakable();
        case RawTile.STONE: return new Stone();
        case RawTile.FALLING_STONE: return new FallingStone();
        case RawTile.BOX: return new Box();
        case RawTile.FALLING_BOX: return new FallingBox();
        case RawTile.FLUX: return new Flux();
        case RawTile.KEY1: return new Key1();
        case RawTile.LOCK1: return new Lock1();
        case RawTile.KEY2: return new Key2();
        case RawTile.LOCK2: return new Lock2();
        default: assertExhausted(tile);
    }
}
function transformMap() {
    map = new Array(rawMap.length);
    for (let y = 0; y < rawMap.length; y++) {
        map[y] = new Array(rawMap[y].length);
        for (let x = 0; x < rawMap[y].length; x++) {
            map[y][x] = transformTile(rawMap[y][x]);
        }
    }
}
window.onload = () => {
    transformMap();
    gameLoop();
}
```

Примечание: Прием из TypeScript, скоро будет пояснен

Новый метод для преобразования перечисления RawTile в объект Tile2

Новый метод для отображения всей карты

Не забываем вызвать новый метод

В TYPESCRIPT...

Перечисление — это название для числа, как в C#, а не класса, как в Java. Поэтому нам не нужно выполнять никакое преобразование между числами и перечислениями и мы можем просто использовать индексы последних, как в предыдущем коде.

`transformMap` четко вписывается в пятистрочный лимит. С ней наше приложение компилируется без ошибки. Теперь можно проверить работоспособность игры, везде переименовать `Tile2` в `Tile` и выполнить коммит внесенных изменений.

При этом `transformTile` нарушает правило «Пять строк», а также почти нарушает еще одно — «Никогда не использовать `switch`», — но тут мы буквально втискиваемся в рамки исключения из правил, об этом ниже.

4.2.4. Правило «Никогда не использовать `switch`»

Утверждение

Так и есть: никогда не использовать `switch`. Исключением могут стать случаи, когда в этих инструкциях нет `default` и возвращение происходит в *каждом* `case`.

Объяснение

Переключатели — это зло, так как они допускают два так называемых удобства, каждое из которых ведет к багам. Во-первых, при выполнении анализа кейсов с помощью `switch` нам не всегда нужно что-то делать для каждого значения. Ведь в `switch` есть вариант `default` — по умолчанию. С помощью `default` можно разобраться со многими значениями без повторения их оценки. Теперь и обрабатываемые, и не обрабатываемые нами случаи становятся инвариантом. Однако, как и при использовании любого предустановленного (`default`) значения, компилятор сам принимает решения и перестает просить нас перепроверять этот инвариант при добавлении новых значений. Теперь для него безразлично, забыли мы обработать новое значение или же решили оставить его как `default`.

Еще одно неудачное «удобство» `switch` заключается в его сквозной логике, подразумевающей продолжение выполнения программы до момента достижения `break`. А ведь так легко можно забыть включить эту инструкцию и не заметить ее отсутствия.

Обычно я настоятельно рекомендую держаться подальше от `switch`. Но, как говорится в подробном описании правила, часть недостатков `switch` можно

исправить. Первый способ прост: не помещать функциональность в `default`. В большинстве языков `default` использовать не стоит. Однако не все языки позволяют опускать `default`, и если используемый нами язык относится к таким, функциональность `switch` нужно исключать полностью.

Проблему сквозной логики мы решаем с помощью возвращения (`return`) в каждом кейсе. В результате сквозного прохода по коду не происходит, значит, и `break` мы не потеряем.

В TYPESCRIPT...

Переключатели особенно полезны, так как мы можем проинструктировать компилятор проверять, все ли значения перечисления были отображены. Чтобы это реализовать, нам потребуется ввести «магическую функцию», но данная тема уже относится конкретно к TypeScript и в программу нашей книги не входит. К счастью, эта функция никогда не меняется и данный паттерн в TypeScript всегда работает.

Листинг 4.93. Прием `assertExhausted`

```
function assertExhausted(x: never): never {
  throw new Error("Unexpected object: " + x);
}
/// ...
switch (t) {
  case ...: return ...;
  // ...
  default: assertExhausted(t);
}
```

Этот тип функции является одним из немногих, которые не получится перестроить для соответствия правилу пяти строк, если нам нужно, чтобы компилятор проверял, отображали ли мы все значения.

Запах

В книге Мартина Фаулера «Рефакторинг» `switch` отнесен к запахам. `Switch` фокусируется на контексте: «как обработать значение *X* *здесь*». В противоположность этому перемещение функциональности в классы фокусируется на данных: «как это значение (объект) обрабатывает ситуацию *X*». Первый случай — фокус на контексте — означает большую удаленность инвариантов от связанных с ними данных, что ведет к их глобализации.

Намерение

Эlegantным побочным эффектом этого правила является то, что мы преобразуем `switch` в цепочки `else if`, которые затем делаем классами. Происходит добавление кода, удаляющего `if`, и они исчезают, при этом функциональность сохраняется, а добавление новых значений становится более простым и безопасным.

Дополнительная информация

Как уже говорилось, об этом запахе можно подробнее почитать в книге Мартина Фаулера «Рефакторинг».

4.2.5. Удаление `if`

Итак, на чем мы остановились? Мы работаем над функцией `colorOfTile`, которая сейчас выглядит так.

Листинг 4.94. Начальный

```
function colorOfTile(g: CanvasRenderingContext2D, x: number, y: number) {
  if (map[y][x].isFlux())
    g.fillStyle = "#ccffcc";
  else if (map[y][x].isUnbreakable())
    g.fillStyle = "#999999";
  else if (map[y][x].isStone()
    || map[y][x].isFallingStone())
    g.fillStyle = "#0000cc";
  else if (map[y][x].isBox()
    || map[y][x].isFallingBox())
    g.fillStyle = "#8b4513";
  else if (map[y][x].isKey1()
    || map[y][x].isLock1())
    g.fillStyle = "#ffcc00";
  else if (map[y][x].isKey2()
    || map[y][x].isLock2())
    g.fillStyle = "#00ccff";
}
```

`colorOfTile` нарушает правило «Никогда не использовать `if` с `else`». Видно, что все условия в этой функции оценивают `map[y][x]`. Подобные условия мы уже наблюдали ранее, поэтому мы снова применяем «Перемещение кода в классы».

1. Копируем `colorOfTile` и вставляем ее во все классы. Удаляем `function`. В данном случае удаляем параметры `y` и `x` и заменяем `map[y][x]` на `this`.
2. Копируем сигнатуру метода в интерфейс `Tile` и попутно переименовываем его в `color`.

3. Проходим по новому методу во всех классах:
 - 1) встраиваем все методы `is`;
 - 2) удаляем `if (true)` и `if (false) { ... }`. В большинстве новых методов остается по одной строке, а `Air` и `Player` вообще пусты;
 - 3) изменяем имя на `color`, отмечая таким образом, что с этим методом мы закончили.
4. Заменяем тело `colorOfTile` на вызов `map[y][x].color`. Вот теперь инструкция `if` полностью удалена, и никакие правила больше не нарушаются.

Листинг 4.95. До

```
function colorOfTile(
  g: CanvasRenderingContext2D,
  x: number, y: number)
{
  if (map[y][x].isFlux())
    g.fillStyle = "#ccffcc";
  else if (map[y][x].isUnbreakable())
    g.fillStyle = "#999999";
  else if (map[y][x].isStone()
    || map[y][x].isFallingStone())
    g.fillStyle = "#0000cc";
  else if (map[y][x].isBox()
    || map[y][x].isFallingBox())
    g.fillStyle = "#8b4513";
  else if (map[y][x].isKey1()
    || map[y][x].isLock1())
    g.fillStyle = "#ffcc00";
  else if (map[y][x].isKey2()
    || map[y][x].isLock2())
    g.fillStyle = "#00ccff";
}
```

Листинг 4.96. После

```
function colorOfTile(
  g: CanvasRenderingContext2D,
  x: number, y: number)
{
  map[y][x].color(g);
}
interface Tile {
  // ...
  color(g: CanvasRenderingContext2D): void;
}
class Air implements Tile {
  // ...
  color(g: CanvasRenderingContext2D) {
    ← color в Air и Player пуст, потому что
      все if оказались ложными
  }
}
class Flux implements Tile {
  // ...
  color(g: CanvasRenderingContext2D) {
    g.fillStyle = "#ccffcc"; ←
  }
}
}

Остальные классы содержат
только свой конкретный цвет
```

Метод `colorOfTile` содержит всего одну строку, поэтому мы используем «Встраивание метода».

1. Изменяем имя метода на `colorOfTile2`.
2. Копируем тело `map[y][x].color(g)`; и отмечаем параметры `x`, `y` и `g`.
3. Эта функция используется только в одном месте, и там мы заменяем вызов ее телом.

Листинг 4.97. До

```
colorOfTile(g, x, y);
```

Листинг 4.98. После

```
map[y][x].color(g);
```

В итоге у нас получается следующее.

Листинг 4.99. До

```
function drawMap(
  g: CanvasRenderingContext2D)
{
  for (let y = 0; y < map.length; y++) {
    for (let x = 0;
        x < map[y].length; x++){
      colorOfTile(g, x, y);
      if (map[y][x] !== Tile.AIR
          && map[y][x] !== Tile.PLAYER)
        g.fillRect(
          x * TILE_SIZE,
          y * TILE_SIZE,
          TILE_SIZE,
          TILE_SIZE);
    }
  }
}
function colorOfTile(
  g: CanvasRenderingContext2D,
  x: number, y: number)
{
  map[y][x].color(g);
}
```

Листинг 4.100. После

```
function drawMap(
  g: CanvasRenderingContext2D)
{
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length; x++){
      map[y][x].color(g); ← Встроенное тело
      if (!map[y][x].isAir()
          && !map[y][x].isPlayer())
        g.fillRect(
          x * TILE_SIZE,
          y * TILE_SIZE,
          TILE_SIZE,
          TILE_SIZE);
    }
  }
}
← colorOfTile удалена
```

Нам удалось успешно избавиться от большой конструкции `if` в `drawMap`. Но `drawMap` по-прежнему не вписывается в наши правила, так что продолжим.

4.3. РАЗБИРАЕМСЯ С ПОВТОРЕНИЕМ КОДА

Недостаток `drawMap` заключается в том, что у нее в середине находится `if`. Проблему можно решить, как и раньше, извлечением этой `if`. Но текущая глава посвящена «Перемещению кода в классы», поэтому можно пойти на авантюру и попробовать именно такой прием. Его применение имеет смысл, потому что и `if`, и предшествующая ей строка связаны с `map[y][x]`.

СОВЕТ

Если вы хотите действовать немного смелее, то можете вместо извлечения метода и его встраивания в следующий процесс перемещать этот метод непосредственно в классы. Только сначала обязательно сделайте коммит, чтобы в случае возникновения неполадок можно было сделать откат назад.

Эта процедура будет одинаковой для `handleInput` и `colorOfTile`, за исключением того, что мы не просто извлекаем `if`. Мы начинаем с применения «Извлечения метода» (3.2.1) для тела инструкций `for`.

Листинг 4.101. До

```
function drawMap(
  g: CanvasRenderingContext2D
) {
  for (let y = 0; y < map.length; y++) {
    for (let x = 0;
      x < map[y].length; x++){
      map[y][x].color(g);
      if (!map[y][x].isAir()
        && !map[y][x].isPlayer())
        g.fillRect(
          x * TILE_SIZE, y * TILE_SIZE,
          TILE_SIZE, TILE_SIZE);
    }
  }
}
```

Листинг 4.102. После

```
function drawMap(
  g: CanvasRenderingContext2D
) {
  for (let y = 0; y < map.length; y++) {
    for (let x = 0;
      x < map[y].length; x++){
      drawTile(g, x, y);
    }
  }
}

function drawTile(
  g: CanvasRenderingContext2D,
  x: number, y: number
) {
  map[y][x].color(g);
  if (!map[y][x].isAir()
    && !map[y][x].isPlayer())
    g.fillRect(
      x * TILE_SIZE, y * TILE_SIZE,
      TILE_SIZE, TILE_SIZE);
}
```

Теперь можно использовать «Перемещение кода в классы» для перемещения этого метода в классы `Tile`.

Листинг 4.103. До

```
function drawTile(
  g: CanvasRenderingContext2D,
  x: number, y: number
) {
  map[y][x].color(g);
  if (!map[y][x].isAir()
    && !map[y][x].isPlayer())
    g.fillRect(
      x * TILE_SIZE,
      y * TILE_SIZE,
      TILE_SIZE,
      TILE_SIZE);
}
```

Листинг 4.104. После

```
function drawTile(
  g: CanvasRenderingContext2D,
  x: number, y: number
) {
  map[y][x].draw(g, x, y);
}

interface Tile {
  // ...
  draw(g: CanvasRenderingContext2D,
    x: number, y: number): void;
}

class Air implements Tile {
  // ...
  draw(g: CanvasRenderingContext2D,
    x: number, y: number)
  {
    ← В итоге draw в Air и Player
    оказывается пуста
  }
}

class Flux implements Tile {
  // ...
  draw(g: CanvasRenderingContext2D,
    x: number, y: number)
  {
```

```

g.fillStyle = "#ccffcc";
g.fillRect(
  x * TILE_SIZE,
  y * TILE_SIZE,
  TILE_SIZE,
  TILE_SIZE);
}
}

```

Все остальные классы в итоге содержат по две строки после встраивания color и isAir, а также удаления if (true)

Как обычно, после «Перемещения кода в классы» получаем функцию всего с одной строкой: `drawTile`. Поэтому используем «Встраивание метода».

Листинг 4.105. До

```

function drawMap(
  g: CanvasRenderingContext2D)
{
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length; x++){
      drawTile(g, x, y);
    }
  }
}
function drawTile(
  g: CanvasRenderingContext2D,
  x: number, y: number)
{
  map[y][x].draw(g);
}

```

Листинг 4.106. После

```

function drawMap(
  g: CanvasRenderingContext2D)
{
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length; x++){
      map[y][x].draw(g, x, y);
    }
  }
}

```

← Встроенное тело

← drawTile удалена

Сейчас вас может заинтересовать: «Зачем все это повторение кода в классах? Разве нельзя было использовать вместо интерфейсов абстрактные классы и поместить весь общий код в них?» Ответу на эти вопросы по очереди.

4.3.1. Разве нельзя было использовать вместо интерфейсов абстрактные классы?

Начну с короткого ответа «да». Да, мы могли так сделать, избежав в результате повторения кода. Однако подобный подход также имеет существенные недостатки. Во-первых, использование интерфейса, как мы сделали это выше, вынуждает нас предпринимать действия для каждого нового вводимого класса. При таком подходе невозможно случайно забыть свойство или переопределить что-то, что переопределению подлежать не должно. А вот если бы мы отдали предпочтение использованию классов, столкнулись бы с тем, что, вернувшись к коду через полгода для добавления нового типа клеток, обнаружили бы наверняка, что напрочь забыли, как в нем и что работает.

Этот принцип настолько силен и убедителен, что даже был оформлен в правило, которое запрещает нам использовать абстрактные классы: «Наследовать только от интерфейсов».

4.3.2. Правило «Наследовать только от интерфейсов»

Утверждение

Реализовывать наследование только от интерфейсов.

Объяснение

Это правило утверждает, что мы можем наследовать только от интерфейсов, но не от классов или их абстрактных аналогов. Чаще всего разработчики используют абстрактные классы, чтобы обеспечить предустановленную (default) реализацию для некоторых методов, оставив другие абстрактными. Это уменьшает повторяемость и оказывается удобным для тех, кто попросту ленив.

К сожалению, недостатки этого подхода перевешивают достоинства. Совместно используемый код ведет к зацеплению. В этом случае зацепление — это код в абстрактном классе. Представьте, что в этом классе реализованы два метода: `methodA` и `methodB`. Мы выясняем, что одному подклассу требуется только `methodA`, а второму — только `methodB`. В такой ситуации единственным вариантом становится переопределить один из методов пустой версией.

Когда у нас есть метод с предустановленной (default) реализацией, возможны два сценария: либо каждый подкласс нуждается в этом методе, в этом случае мы можем легко вынести его вонне класса; либо *некоторым* подклассам нужно переопределить этот метод, но из-за предустановленной (default) реализации компилятор сам не напомним нам о методе, когда мы станем добавлять новый подкласс.

Это еще один пример возможных проблем с предустановленными (default) реализациями, о некоторых я уже упоминал ранее. В этой ситуации будет лучше оставить методы полностью абстрактными, потому что тогда нам потребуется обрабатывать все случаи явно.

Когда нескольким классам нужно использовать общий код, можно поместить этот код в отдельный общий класс. Мы еще вернемся к этой теме в главе 5 при обсуждении «Введения паттерна “Стратегия”» (5.4.2).

Запах

Я вывел это правило из принципа «Выбирайте композицию объектов вместо наследования», приведенного в книге Эриха Гаммы, Ричарда Хельма, Ральфа Джонсона и Джона Влсидиса «Паттерны объектно-ориентированного проектирования». Эта же книга ввела в сферу ООП понятие *паттернов проектирования*.

Намерение

Этот запах отчетливо утверждает, что мы должны разделять код, ссылаясь на другие объекты, а не наследуя от них. Данное правило чрезвычайно строго, поскольку очень редко бывает так, что задача требует наследования. А в случаях,

когда наследование не требуется, композиция дает намного более гибкие и стабильные решения.

Дополнительная информация

Как уже говорилось, это правило взято из книги «Паттерны объектно-ориентированного проектирования». Более удачное решение, дающее нужное совместное использование кода, будет разбираться в главе 5 при обсуждении техники рефакторинга «Введение паттерна “Стратегия”» (5.4.2).

4.3.3. Зачем все это повторение кода?

Во многих случаях повторение кода считается плохим признаком. Об этом все знают, но подумаем, на чем основано это мнение. Повторение кода плохо в случаях, когда есть необходимость сопровождать его, проводя изменения по всей программе.

Если у нас дублируется код и мы изменим его только в одном месте из двух, то в итоге получим две разные функции. Иными словами, повторение кода плохо, потому что создает угрозу расхождений между его репликами.

В большинстве случаев мы не стремимся к повторению кода. Однако в нашей конкретной ситуации оно будет уместно. Мы предполагаем, что графическое оформление клеток должно со временем меняться, а клетки должны постоянно различаться между собой. То есть расхождение кодов двух копий со временем окажется обоснованным и полезным. Чтобы понять смысл этого, подумайте, насколько легко было бы сделать ключи в нашем примере с игрой круглыми.

Ну а если бы код должен был сходиться, а не расходиться, как в случае выше, то как бы мы это обработали, не прибегая к наследованию? Обработаем эту ситуацию в следующей главе.

4.4. РЕФАКТОРИНГ ДВУХ СЛОЖНЫХ ВЫРАЖЕНИЙ IF

Остаются две функции, которые все еще нарушают наши правила, это `moveHorizontal` и `moveVertical`. Так как они почти одинаковы, я представил обработку только наиболее сложной из них, оставив вторую вам в качестве самостоятельного упражнения. Сейчас `moveHorizontal` выглядит очень громоздкой, но большую ее часть на время можно проигнорировать.

Для начала обратите внимание, что у нас здесь два оператора `||`. Они выражают информацию о внутренней области. Значит, эту структуру нам нужно не только сохранить, но и выделить. Для этого мы перенесем в классы только эту часть.

Листинг 4.107. Начальный

```
function moveHorizontal(dx: number) {
  if (map[playery][playerx + dx].isFlux()
    || map[playery][playerx + dx].isAir()) {
    moveToTile(playerx + dx, playery);
  } else if ((map[playery][playerx + dx].isStone()
    || map[playery][playerx + dx].isBox()
    && map[playery][playerx + dx + dx].isAir()
    && !map[playery + 1][playerx + dx].isAir()) {
    map[playery][playerx + dx + dx] = map[playery][playerx + dx];
    moveToTile(playerx + dx, playery);
  } else if (map[playery][playerx + dx].isKey1()) {
    removeLock1();
    moveToTile(playerx + dx, playery);
  } else if (map[playery][playerx + dx].isKey2()) {
    removeLock2();
    moveToTile(playerx + dx, playery);
  }
}
```

||, которые
нужно сохранить

Данный подход несколько отличается от того, что мы делали ранее, поскольку перемещается не весь метод, хотя сам процесс остается прежним. Самое сложное — это придумать хорошее имя. Теперь необходимо посмотреть, что делает код, и соблюсти осторожность. Нам нужно принять за аксиому, что между флаксом и воздухом есть что-то общее. Речь идет конкретно об игре, так что я не буду особо заморачиваться, а просто скажу, что они оба *съедобны* (*isEdible*).

1. Вводим в интерфейс *Tile* метод *isEdible*.
2. В каждом классе добавляем метод с немного измененным именем: *isEdible2*.
3. В качестве тела помещаем `return this.isFlux() || this.isAir();`.
4. Встраиваем значения *isFlux* и *isAir*.
5. Удаляем временную 2 в имени.
6. Заменяем `map[playery][playerx + dx].isFlux() || map[playery][playerx + dx].isAir()` *только здесь*. Нельзя заменять ее везде, потому что мы не знаем, относятся ли другие `||` к тому же свойству (то есть к съедобности).

Для других `||` ситуация аналогична. Здесь ящики и камни разделяют свойство *pushable* (возможность перемещения) в данном контексте. Следуя тому же шаблону, мы получаем такой код.

Листинг 4.108. До

```
function moveHorizontal(dx: number) {
  if (map[playery][playerx + dx].isFlux()
    || map[playery][playerx + dx].isAir()) {
    moveToTile(playerx + dx, playery);
  } else if ((map[playery][playerx + dx].isStone()
    || map[playery][playerx + dx].isBox())
```

||, которые
нужно извлечь

```

    && map[playery][playerx + dx + dx].isAir()
    && !map[playery + 1][playerx + dx].isAir()) {
    map[playery][playerx + dx + dx] = map[playery][playerx + dx];
    moveToTile(playerx + dx, playery);
} else if (map[playery][playerx + dx].isKey1()) {
    removeLock1();
    moveToTile(playerx + dx, playery);
} else if (map[playery][playerx + dx].isKey2()) {
    removeLock2();
    moveToTile(playerx + dx, playery);
}
}

```

Листинг 4.109. После

```

function moveHorizontal(dx: number) {
    if (map[playery][playerx + dx].isEdible()) {
        moveToTile(playerx + dx, playery);
    } else if (map[playery][playerx + dx].isPushable()) {
        && map[playery][playerx + dx + dx].isAir()
        && !map[playery + 1][playerx + dx].isAir()) {
            map[playery][playerx + dx + dx] = map[playery][playerx + dx];
            moveToTile(playerx + dx, playery);
        } else if (map[playery][playerx + dx].isKey1()) {

            removeLock1();
            moveToTile(playerx + dx, playery);
        } else if (map[playery][playerx + dx].isKey2()) {
            removeLock2();
            moveToTile(playerx + dx, playery);
        }
    }
}

interface Tile {
    // ...
    isEdible(): boolean;
    isPushable(): boolean;
}

class Box implements Tile {
    // ...
    isEdible() { return false; }
    isPushable() { return true; }
}

class Air implements Tile {
    // ...
    isEdible() { return true; }
    isPushable() { return false; }
}

```

← Новые вспомогательные методы

← Вок и Stone схожи

← Air и Flux схожи

Сохранив поведение `||`, продолжаем как обычно и смотрим на контекст. В этом коде им является `map[playery][playerx + dx]`, поскольку она используется в каждой `if`. Здесь мы видим, что «Перемещение кода в классы» применимо не только к серии проверок тождественности, но также ко всему, что имеет отчетливый

контекст: то есть много вызовов методов в одном экземпляре ([.] слева с одинаковым содержимым).

Итак, мы снова перемещаем код в `map[playery][playerx + dx]; tile`. После применения «Перемещения кода в классы» в целом получаем следующее.

Листинг 4.110. После применения «Перемещения кода в классы»

```
function moveHorizontal(dx: number) {
  map[playery][playerx + dx].moveHorizontal(dx);
}
interface Tile {
  // ...
  moveHorizontal(dx: number): void;
}
class Box implements Tile { ←
  // ...
  moveHorizontal(dx: number) {
    if (map[playery][playerx + dx + dx].isAir()
        && !map[playery + 1][playerx + dx].isAir()) {
      map[playery][playerx + dx + dx] = this;
      moveToTile(playerx + dx, playery);
    }
  }
}
class Key1 implements Tile { ←
  // ...
  moveHorizontal(dx: number) {
    removeLock1();
    moveToTile(playerx + dx, playery);
  }
}
class Lock1 implements Tile { ←
  // ...
  moveHorizontal(dx: number) { }
}
class Air implements Tile { ←
  // ...
  moveHorizontal(dx: number) {
    moveToTile(playerx + dx, playery);
  }
}
```

Box и Stone
схожи

Key1 и Key2
схожи

Остальные
пусты

Air и Flux
схожи

Как обычно, исходный метод `moveHorizontal` содержит всего одну строку, поэтому мы его встраиваем. Обратите внимание: из-за того что `if` была довольно сложной, от нее остались артефакты в `Box` и `Stone`. К счастью, они по-прежнему соответствуют правилам. Теперь вы можете проделать то же самое для метода `moveVertical`.

Единственный метод, который все еще конфликтует с новым правилом «Никогда не использовать `if с else`», — это `updateTile`. Но у него есть особенности структуры, которую более подробно мы изучим в следующей главе.

4.5. УДАЛЕНИЕ МЕРТВОГО КОДА

В завершение главы проведем небольшую чистку. Мы ввели множество новых методов, некоторые, наоборот, удалили после встраивания, но все еще кое-что в коде можем усовершенствовать.

Многие IDE, включая Visual Studio Code, указывают на функции, которые не используются. Когда мы встречаем подобное указание, нам следует отложить все и сразу такую функцию удалить. Удаление кода сэкономит время и силы, потому что в будущем, возвратясь к коду, не придется уделять ему внимание и размышлять над ним.

К сожалению, из-за того, что интерфейсы являются публичными, никакая IDE не сообщит о том, есть ли там неиспользуемые методы. Мало ли, зачем мы могли их сохранить: задел на будущее, применение для чего-то выходящего за область видимости — IDE не может этого знать. Как правило, методы из интерфейсов нельзя так просто удалять.

Однако интерфейсы, рассмотренные в этой главе, введены нами, вся их область видимости известна. Так что можем свободно делать с ними что нам пожелаем: в частности, можем удалить оттуда неиспользуемые методы. Вот техника для обнаружения подобных методов.

1. Выполнить компиляцию. Ошибок быть не должно.
2. Удалить метод из интерфейса.
3. Выполнить компиляцию:
 - 1) если компилятор начнет ругаться, то отменить удаление и переходить дальше;
 - 2) в противном случае пройти по каждому классу и проверить, можно ли удалить из него тот же метод, не получив ошибок.

Эта техника проста, но очень полезна. После чистки интерфейсов в одном из них остается один метод, а во втором десять. Я настолько люблю удалять код, что выработал на основе этого процесса целый шаблон рефакторинга: «Пробное удаление с последующей компиляцией».

4.5.1. Шаблон рефакторинга «Пробное удаление с последующей компиляцией»

Описание

Этот шаблон в первую очередь применяется для удаления неиспользуемых методов из интерфейсов, когда вся область видимости этих интерфейсов известна. Его также можно применять для поиска и удаления любых неиспользуемых методов. Выполнить «Пробное удаление с последующей компиляцией» не сложнее, чем говорит о том само имя шаблона: сначала мы удаляем метод, а затем смотрим на реакцию компилятора. Этот шаблон рефакторинга интересен именно своим назначением, а не замысловатостью. Обратите внимание, что не стоит выполнять такой рефакторинг при реализации новых возможностей в коде, поскольку можно случайно удалить методы, которые *пока* не используются.

Наличие неактуального кода существенно тормозит всю базу кода. Будучи частью программы, он выполняется или игнорируется, но в том и другом случае требует внимания, такой фрагмент кода замедляет компиляцию и анализ, затрудняет тестирование. Чем быстрее мы его удалим, тем экономнее распорядимся своим временем и ресурсами системы.

Многие программы-редакторы определенным образом выделяют такие фрагменты кода. Но, когда метод расположен в интерфейсе, даже если он не используется, программы-редакторы оказываются бессильны указать на это. Они не будут судить, есть ли обращение к коду изнутри зоны видимости или извне. Редакторы эту разницу отличить не могут. По их мнению, единственное безопасное для целостности программы решение — предположить, что все методы интерфейса предназначены для использования вне нашей области видимости.

Когда нам известно, что интерфейс используется только в этой области, необходимо почистить его вручную. Это и является целью описываемого шаблона рефакторинга.

Процесс

1. Выполнить компиляцию. Ошибок быть не должно.
2. Удалить метод из интерфейса.
3. Выполнить компиляцию:
 - 1) если компилятор выдает ошибку, отменить удаление и переходить далее;
 - 2) в противном случае пройти по каждому классу и проверить, можно ли удалить из него этот метод, не вызвав ошибку.

Пример

В приведенном ниже учебном фрагменте присутствует три неиспользуемых метода, но программа-редактор выделяет не все из них. А в некоторых редакторах вообще ни один не выделяется.

Листинг 4.111. Начальный

```
interface A {  
    m1(): void;  
    m2(): void;  
}  
class B implements A {  
    m1() { console.log("m1"); }  
    m2() { this.m3(); }  
    m3() { console.log("m3"); }  
}  
let a = new B();  
a.m1();
```

Ну что, сможете обнаружить и устранить эти три неиспользуемых метода, выполняя описанный процесс?

РЕЗЮМЕ

- Правила «Никогда не использовать `if с else`» (4.1.1) и «Никогда не использовать `switch`» (4.2.4) утверждают, что `else` и `switch` следует использовать только на границах программы. И `else`, и `switch` являются низкоуровневыми операторами потока управления. Перерабатывая коды приложений, следует применять шаблоны рефакторинга «Замена кода типа классами» (4.1.3) и «Перемещение кода в классы» (4.1.5) для замены операторов `switch` и цепочек `else if` высокоуровневыми классами и методами.
- Слишком обобщенные методы могут мешать проводить рефакторинг. В таких случаях можно использовать шаблон «Специализация метода» (4.2.2) для устранения излишней обобщенности.
- Правило «Наследовать только от интерфейсов» (4.3.2) не позволяет повторно использовать код с помощью абстрактных классов и наследования классов, потому что эти виды наследования вызывают чрезмерное зацепление.
- Мы добавили два шаблона рефакторинга для очистки кода после его проведения — «Встраивание метода» (4.1.7) и «Пробное удаление с последующей компиляцией» (4.5.1). Они оба способны удалять методы, которые больше не повышают читаемость кода.

5

Совмещение схожего кода

В этой главе

- ✓ Знакомство с шаблоном «Объединение схожих классов».
- ✓ Раскрытие структуры с помощью условной арифметики.
- ✓ Объяснение простых классовых диаграмм UML.
- ✓ Объединение схожего кода с помощью «Введения паттерна “Стратегия”» (5.4.2).
- ✓ Устранение загруженности с помощью правила «Избегать интерфейсов с единственной реализацией» (5.4.3).

В предыдущей главе я говорил, что мы еще вернемся к функции `updateTile`. Она нарушает несколько правил, в особенности «Никогда не использовать `if` с `else`» (4.1.1). К тому же мы уже поработали над сохранением `||` в коде, потому что они выражают структуру. Пришло время научиться выражать в коде больше подобных структур. Этим мы и займемся в главе 5.

Вот как сейчас выглядит `updateTile`.

Листинг 5.1. Начальный

```
function updateTile(x: number, y: number) {  
  if ((map[y][x].isStone() || map[y][x].isFallingStone())  
      && map[y + 1][x].isAir()) {  
    map[y + 1][x] = new FallingStone();  
  }  
}
```



```

        map[y][x] = new Air();
    } else if ((map[y][x].isBox() || map[y][x].isFallingBox())
        && map[y + 1][x].isAir()) {
        map[y + 1][x] = new FallingBox();
        map[y][x] = new Air();
    } else if (map[y][x].isFallingStone()) {
        map[y][x] = new Stone();
    } else if (map[y][x].isFallingBox()) {
        map[y][x] = new Box();
    }
}

```

5.1. ОБЪЕДИНЕНИЕ СХОЖИХ КЛАССОВ

Для начала, как уже делали выше, заключим в скобки инструкции (то есть `(map[y][x].isStone() || map[y][x].isFallingStone())`), выражающие связь, которую мы хотим не только сохранить, но и усилить. Следовательно, первым шагом для нас будет введение функции для каждого заключенного в скобки выражения с `||`. В данном случае признаки `stone` и `box` нужно понимать как «ведет себя как камень» и «ведет себя как ящик» соответственно.

Листинг 5.2. До

```

function updateTile(x: number, y: number)
{
    if ((map[y][x].isStone()
        || map[y][x].isFallingStone())
        && map[y + 1][x].isAir()) {
        map[y + 1][x] = new FallingStone();
        map[y][x] = new Air();
    } else if ((map[y][x].isBox()
        || map[y][x].isFallingBox())
        && map[y + 1][x].isAir()) {
        map[y + 1][x] = new FallingBox();
        map[y][x] = new Air();
    } else if (map[y][x].isFallingStone()) {
        map[y][x] = new Stone();
    } else if (map[y][x].isFallingBox()) {
        map[y][x] = new Box();
    }
}

```

Листинг 5.3. После

```

function updateTile(x: number, y: number) {
    if (map[y][x].isStony()
        && map[y + 1][x].isAir()) {
        map[y + 1][x] = new FallingStone();
        map[y][x] = new Air();
    } else if (map[y][x].isBoxy()
        && map[y + 1][x].isAir()) {
        map[y + 1][x] = new FallingBox();
        map[y][x] = new Air();
    } else if (map[y][x].isFallingStone()) {
        map[y][x] = new Stone();
    } else if (map[y][x].isFallingBox()) {
        map[y][x] = new Box();
    }
}

interface Tile {
    // ...
    isStony(): boolean;
    isBoxy(): boolean;
}

```

Новые
вспомогательные
методы

```
class Air implements Tile {
    // ...
    isStony() { return false; }
    isBoxy() { return false; }
}
```

Новые вспомогательные
методы

Разобравшись с `||`, можно переместить код в классы, а можно пока не спешить, сначала взглянуть на эти классы и припомнить то множество методов, которые мы ввели в предыдущей главе. Итак, в данный момент «Пробное удаление с последующей компиляцией» (4.5.1) позволяет нам удалить `isStone` и `isBox`.

Очевидно, что единственное различие между `Stone` и `FallingStone` заключается в результате методов `isFallingStone` и `moveHorizontal`.

Листинг 5.4. Stone

```
class Stone implements Tile {
    isAir() { return false; }
    isFallingStone() { return false; }
    isFallingBox() { return false; }
    isLock1() { return false; }
    isLock2() { return false; }
    draw(g: CanvasRenderingContext2D,
        x: number, y: number)
    {
        // ...
    }
    moveVertical(dy: number) { }
    isStony() { return true; }
    isBoxy() { return false; }
    moveHorizontal(dx: number) {
        // ...
    }
}
```

Упомянутые
немногочисленные
различия

Листинг 5.5. FallingStone

```
class FallingStone implements Tile {
    isAir() { return false; }
    isFallingStone() { return true; }
    isFallingBox() { return false; }
    isLock1() { return false; }
    isLock2() { return false; }
    draw(g: CanvasRenderingContext2D,
        x: number, y: number)
    {
        // ...
    }
    moveVertical(dy: number) { }
    isStony() { return true; }
    isBoxy() { return false; }
    moveHorizontal(dx: number) {
        // ...
    }
}
```

Когда метод возвращает константу, мы называем его *константным методом*. Два класса, с которыми мы сейчас работаем, можно объединить, потому что они разделяют один константный метод, а он в каждом кейсе возвращает значение, отличное от возвращаемого значения в другом кейсе. Подобное объединение двух классов выполняется в две фазы, а сам процесс напоминает алгоритм сложения дробей. Первым шагом при сложении дробей уравниваются знаменатели. Аналогичным образом при объединении классов изначально они выравниваются по всем методам, кроме константных. Возвращаясь к примеру с дробями: второй фазой идет их фактическое сложение. В случае с классами также происходит их фактическое объединение. Посмотрим, как это выглядит на практике.

1. На первом этапе уравниваем две `moveHorizontal`s.

1) в теле каждой `moveHorizontal` заключаем имеющийся код в `if (true) { }`;

Листинг 5.6. До

```
class Stone implements Tile {
    // ...
    moveHorizontal(dx: number) {
        if (map[playery][playerx+dx+dx]
            .isAir()
            && !map[playery+1][playerx+dx]
            .isAir())
        {
            map[playery]
                [playerx+dx + dx] = this;
            moveToTile(playerx+dx, playery);
        }
    }
}
class FallingStone implements Tile {
    // ...
    moveHorizontal(dx: number) {
    }
}
```

Листинг 5.7. После (1/8)

```
class Stone implements Tile {
    // ...
    moveHorizontal(dx: number) {
        if (true) {
            if (map[playery][playerx+dx+dx].isAir()
                && !map[playery+1][playerx+dx].isAir())
            {
                map[playery][playerx+dx + dx] = this;
                moveToTile(playerx+dx, playery);
            }
        }
    }
}
class FallingStone implements Tile {
    // ...
    moveHorizontal(dx: number) {
        if (true) {
        }
    }
}
```

← Новые if(true)

2) заменяем `true` на `isFallingStone() === true` и `isFallingStone() === false` соответственно;

Листинг 5.8. До

```
class Stone implements Tile {
    // ...
    moveHorizontal(dx: number) {
        if (true) {
            if (map[playery][playerx+dx+dx]
                .isAir()
                && !map[playery+1][playerx+dx]
                .isAir())
            {
                map[playery]
                    [playerx+dx + dx] = this;
                moveToTile(playerx+dx, playery);
            }
        }
    }
}
class FallingStone implements Tile {
    // ...
    moveHorizontal(dx: number) {
        if (true) {
        }
    }
}
```

Листинг 5.9. После (2/8)

```
class Stone implements Tile {
    // ...
    moveHorizontal(dx: number) {
        if (this.isFallingStone() === false) {
            if (map[playery][playerx+dx+dx].isAir()
                && !map[playery+1][playerx+dx].isAir())
            {
                map[playery][playerx+dx + dx] = this;
                moveToTile(playerx+dx, playery);
            }
        }
    }
}
class FallingStone implements Tile {
    // ...
    moveHorizontal(dx: number) {
        if (this.isFallingStone() === true) {
        }
    }
}
```

← Специализированные условия

3) копируем тело каждой `moveHorizontal` и вместе с `else` вставляем его в другую `moveHorizontal`.

Листинг 5.10. До

```
class Stone implements Tile {
    // ...
    moveHorizontal(dx: number) {
        if (this.isFallingStone() === false) {
            if (map[playery][playerx+dx+dx]
                .isAir()
                && !map[playery+1][playerx+dx]
                .isAir())
            {
                map[playery]
                    [playerx+dx + dx] = this;
                moveToTile(playerx+dx, playery);
            }
        }
    }
}
class FallingStone implements Tile {
    // ...
    moveHorizontal(dx: number) {
        if (this.isFallingStone() === true)
        {
        }
    }
}
```

Тело из другого
метода

Листинг 5.11. После (3/8)

```
class Stone implements Tile {
    // ...
    moveHorizontal(dx: number) {
        if (this.isFallingStone() === false) {
            if (map[playery][playerx+dx+dx].isAir()
                && !map[playery+1][playerx+dx].isAir())
            {
                map[playery][playerx+dx + dx] = this;
                moveToTile(playerx+dx, playery);
            }
        }
        else if (this.isFallingStone() === true)
        {
        }
    }
}
class FallingStone implements Tile {
    // ...
    moveHorizontal(dx: number) {
        if (this.isFallingStone() === false) {
            if (map[playery][playerx+dx+dx].isAir()
                && !map[playery+1][playerx+dx].isAir())
            {
                map[playery][playerx+dx + dx] = this;
                moveToTile(playerx+dx, playery);
            }
        }
        else if (this.isFallingStone() === true)
        {
        }
    }
}
```

2. Теперь, когда отличаются друг от друга только константные методы `isFallingStone`, переходим ко второй фазе, начав с введения поля `falling` и присвоения ему значения в конструкторе.

Листинг 5.12. До

```
class Stone implements Tile {
    // ...
    isFallingStone() { return false; }
}
class FallingStone implements Tile {
    // ...
    isFallingStone() { return true; }
}
```

Присваиваем новому полю
значение по умолчанию

Листинг 5.13. После (4/8)

```
class Stone implements Tile {
    private falling: boolean;
    constructor() {
        this.falling = false;
    }
    // ...
    isFallingStone() { return false; }
}
class FallingStone implements Tile {
    private falling: boolean;
    constructor() {
        this.falling = true;
    }
    // ...
    isFallingStone() { return true; }
}
```

Новое
поле

3. Изменяем `isFallingStone` на возвращение нового поля `falling`.

Листинг 5.14. До

```
class Stone implements Tile {
    // ...
    isFallingStone() { return false; }
}
class FallingStone implements Tile {
    // ...
    isFallingStone() { return true; }
}
```

Листинг 5.15. После (5/8)

```
class Stone implements Tile {
    // ...
    isFallingStone() { return this.falling; }
}
class FallingStone implements Tile {
    // ...
    isFallingStone() { return this.falling; }
}
```

Возвращает поле вместо константы

4. Компилируем, чтобы убедиться в сохранении работоспособности кода.

5. Для каждого класса:

1) копируем предустановленное значение `falling` и делаем его параметром;

Листинг 5.16. До

```
class Stone implements Tile {
    private falling: boolean;
    constructor() {
        this.falling = false;
    }
    // ...
}
```

Листинг 5.17. После (6/8)

```
class Stone implements Tile {
    private falling: boolean;
    constructor(falling: boolean) {
        this.falling = falling;
    }
    // ...
}
```

Делаем `falling` параметром

2) проходим по всем ошибкам компилятора и вставляем это предустановленное значение в качестве аргумента.

Листинг 5.18. До

```
/// ...
new Stone();
/// ...
```

Листинг 5.19. После (7/8)

```
/// ...
new Stone(false);
/// ...
```

Производит вызов с предустановленным значением

6. Удаляем все, кроме одного из объединяемых классов, и исправляем все наведенные ошибки компилятора, переключаясь на этот оставшийся класс.

Листинг 5.20. До

```
/// ...
new FallingStone(true);
/// ...
```

Листинг 5.21. После (8/8)

```
/// ...
new Stone(true);
/// ...
```

Заменяет удаленный класс объединенным

В результате объединение приводит к следующей трансформации.

Листинг 5.22. До

```
function updateTile(x: number,
    y: number) {
    if (map[y][x].isStony()
        && map[y + 1][x].isAir()) {
```

Листинг 5.23. После

```
function updateTile(x: number, y:
    number) {
    if (map[y][x].isStony()
        && map[y + 1][x].isAir()) {
```

```

    map[y + 1][x] = new FallingStone();
    map[y][x] = new Air();
} else if (map[y][x].isBoxy()
    && map[y + 1][x].isAir()) {
    map[y + 1][x] = new FallingBox();
    map[y][x] = new Air();
} else if (map[y][x].isFallingStone()) {
    map[y][x] = new Stone();
} else if (map[y][x].isFallingBox()) {
    map[y][x] = new Box();
}
}
class Stone implements Tile {
    // ...
    isFallingStone() { return false; }
    moveHorizontal(dx: number) {
        if (map[playery][playerx+dx+dx].isAir()
            && !map[playery+1][playerx+dx].isAir())
        {
            map[playery][playerx+dx + dx] = this;
            moveToTile(playerx+dx, playery);
        }
    }
}
class FallingStone implements Tile {
    // ...
    isFallingStone() { return true; }
    moveHorizontal(dx: number) { }
}

```

```

    map[y + 1][x] = new Stone(true);
    map[y][x] = new Air();
} else if (map[y][x].isBoxy()
    && map[y + 1][x].isAir()) {
    map[y + 1][x] = new FallingBox();
    map[y][x] = new Air();
} else if (map[y][x].isFallingStone()) {
    map[y][x] = new Stone(false);
} else if (map[y][x].isFallingBox()) {
    map[y][x] = new Box();
}
}
class Stone implements Tile {
    constructor(private falling: boolean) { }
    // ...
    isFallingStone() { return this.falling; }
    moveHorizontal(dx: number) {
        if (this.isFallingStone() === false) {
            if (map[playery][playerx+dx+dx].isAir()
                && !map[playery+1][playerx+dx].isAir())
            {
                map[playery][playerx+dx + dx] = this;
                moveToTile(playerx+dx, playery);
            }
        } else if(this.isFallingStone() === true)
        {
        }
    }
}

```

Закрытое поле, установленное в конструкторе

isFallingStone возвращает none this

moveHorizontal содержит объединенные тела

FallingStone удален

В TYPESCRIPT...

Конструкторы действуют несколько иначе, нежели во многих других языках. Во-первых, у нас может быть только один конструктор, и он всегда называется `constructor`.

Во-вторых, размещение `public` или `private` перед параметром конструктора автоматически создает переменную экземпляра и присваивает ей значение аргумента. Это значит, что следующие фрагменты кода будут эквивалентны:

До

```

class Stone implements Tile {
    private falling: boolean;
    constructor(falling: boolean) {
        this.falling = falling;
    }
}

```

После

```

class Stone implements Tile {
    constructor(
        private falling: boolean) { }
}

```

В этой книге мы, как правило, отдаем предпочтение варианту «После» (справа).

Глядя на полученную `moveHorizontal`, можно отметить несколько интересных моментов. Самое очевидное — это то, что она содержит пустую `if`. Более того, теперь она содержит `else`, то есть нарушает правило «Никогда не использовать `if` с `else`». Типичный эффект объединения классов использованным нами способом — это раскрытие потенциально скрытого кода типов. В данном случае таким кодом является булев параметр `falling`. Раскрыть этот код типа можно путем перевода его в перечисление.

Листинг 5.24. До

```
/// ...
new Stone(true);
/// ...
new Stone(false);
/// ...
class Stone implements Tile {
    constructor(private falling:
        boolean)
    { }
    // ...
    isFallingStone() {
        return this.falling;
    }
}
```

Листинг 5.25. После

```
enum FallingState {
    FALLING, RESTING
}
/// ...
new Stone(FallingState.FALLING);
/// ...
new Stone(FallingState.RESTING);
/// ...
class Stone implements Tile {
    constructor(private falling:
        FallingState)
    { }
    // ...
    isFallingStone() {
        return this.falling
            === FallingState.FALLING;
    }
}
```

Это изменение уже сделало код более читаемым, потому что мы стали обходиться использованием неименованных булевых аргументов в `Stone`. Что еще лучше, мы знаем, как разобраться с перечислениями, а именно использовать «Замену кода типа классами» (4.1.3).

Листинг 5.26. До

```
enum FallingState {
    FALLING, RESTING
}

new Stone(FallingState.FALLING);
new Stone(FallingState.RESTING);
class Stone implements Tile {
    constructor(private falling:
        FallingState)
    { }
```

Листинг 5.27. После

```
interface FallingState {
    isFalling(): boolean;
    isResting(): boolean;
}
class Falling implements FallingState {
    isFalling() { return true; }
    isResting() { return false; }
}
class Resting implements FallingState {
    isFalling() { return false; }
    isResting() { return true; }
}
new Stone(new Falling());
new Stone(new Resting());
class Stone implements Tile {
    constructor(private falling:
        FallingState)
    { }
```

```
// ...
isFallingStone() {
    return this.falling
        === FallingState.FALLING;
}
}
```

```
// ...
isFallingStone() {
    return this.falling.isFalling();
}
}
```

Если нас беспокоит, что `new` работают несколько медленнее, то можно извлечь их в константы. Но здесь нужно помнить: оптимизация производительности программы должна проходить с использованием инструментов профилирования. Если мы встроим `isFallingStone` в метод `moveHorizontal`, то увидим, что наверняка должны использовать «Перемещение кода в классы» (4.1.5).

Листинг 5.28. До

```
interface FallingState {
    // ...
}

class Falling implements FallingState {
    // ...
}

class Resting implements FallingState {
    // ...
}

class Stone implements Tile {
    // ...
    moveHorizontal(dx: number) {
        if (!this.falling.isFalling()) {
            if (map[playery][playerx+dx+dx]
                .isAir()
                && !map[playery+1][playerx+dx]
                .isAir())
            {
                map[playery]
                    [playerx+dx + dx] = this;
                moveToTile(playerx+dx, playery);
            }
        } else if (this.falling
            .isFalling()) {
        }
    }
}
```

Листинг 5.29. После

```
interface FallingState {
    // ...
    moveHorizontal(
        tile: Tile, dx: number): void;
}

class Falling implements FallingState {
    // ...
    moveHorizontal(tile: Tile,
        dx: number) {
    }
}

class Resting implements FallingState {
    // ...
    moveHorizontal(tile: Tile, dx: number) {
        if (map[playery][playerx+dx+dx]
            .isAir()
            && !map[playery+1][playerx+dx]
            .isAir())
        {
            map[playery]
                [playerx+dx + dx] = tile;
            moveToTile(playerx+dx, playery);
        }
    }
}

class Stone implements Tile {
    // ...
    moveHorizontal(dx: number) {
        this.falling
            .moveHorizontal(this, dx);
    }
}
```

В завершение, поскольку мы ввели новый интерфейс, можно использовать технику «Пробное удаление с последующей компиляцией» для удаления `isResting`. Реализацию того же самого для `Box` и `FallingBox` я оставляю вам на самостоятельную проработку. Имейте в виду, что можете повторно использовать `FallingState`. Подобное объединение двух схожих классов мы так и называем: «Объединение схожих классов».

5.1.1. Шаблон рефакторинга «Объединение схожих классов»

Описание

Если имеется два или более класса, которые различаются между собой набором константных методов, их можно объединить с помощью данного шаблона. Набор константных методов называется *базисом*. Базис с двумя методами называется *двухточечным базисом*. Нам же нужно, чтобы в базисе было как можно меньше методов. Когда мы хотим объединить X классов, мы нуждаемся максимум в $(X - 1)$ -точечном базисе. Объединение классов хорошо тем, что уменьшение их числа обычно означает большее раскрытие структуры.

Процесс

1. На первом этапе уравниваем все небазисные методы. Для каждого из них выполняются следующие шаги:
 - 1) в теле каждой версии метода заключаем имеющийся код в `if (true) { }`;
 - 2) заменяем `true` выражением, вызывающим все базисные методы и сравнивающим результаты с их константными значениями;
 - 3) копируем тело каждой версии и вместе с `else` вставляем его во все другие версии.
2. Теперь, когда различаются только базисные методы, переходим ко второму этапу, начиная с введения поля для каждого метода в базисе и присвоения ему констант в конструкторе.
3. Изменяем методы на возвращение новых полей вместо констант.
4. Выполняем компиляцию, проверяя, не нарушили ли мы код.
5. В каждом классе для всех полей по очереди:
 - 1) копируем предустановленное значение поля и делаем его параметром;
 - 2) проходим по всем ошибкам компиляции и вставляем это предустановленное значение в качестве аргумента.
6. Когда объединенные классы будут идентичны, удаляем их все, кроме одного, и исправляем ошибки компиляции переключением на этот оставшийся класс.

Пример

В этом примере у нас светофор с тремя достаточно похожими классами, это означает, что их можно объединить.

Листинг 5.30. Начальный

```
function nextColor(t: TrafficColor) {
  if (t.color() === "red") return new Green();
  else if (t.color() === "green") return new Yellow();
  else if (t.color() === "yellow") return new Red();
}
```

```
interface TrafficColor {
    color(): string;
    check(car: Car): void;
}
class Red implements TrafficColor {
    color() { return "red"; }
    check(car: Car) { car.stop(); }
}
class Yellow implements TrafficColor {
    color() { return "yellow"; }
    check(car: Car) { car.stop(); }
}
class Green implements TrafficColor {
    color() { return "green"; }
    check(car: Car) { car.drive(); }
}
```

Далее строго следуем процессу.

1. Базисным методом является `color`, так как в каждом классе он возвращает константу, отличную от возвращаемой константы в другом классе. Значит, уравнивать нужно методы `check`. Для каждого из этих методов выполняем следующие шаги:

- 1) в теле каждой версии `check` заключаем имеющийся код в `if (true) { }`;

Листинг 5.31. До

```
class Red implements TrafficColor {
    // ...
    check(car: Car) {

        car.stop();

    }
}
class Yellow implements TrafficColor {
    // ...
    check(car: Car) {

        car.stop();

    }
}
class Green implements TrafficColor {
    // ...
    check(car: Car) {

        car.drive();

    }
}
```

Листинг 5.32. После (1/8)

```
class Red implements TrafficColor {
    // ...
    check(car: Car) {
        if (true) {
            car.stop();
        }
    }
}
class Yellow implements TrafficColor {
    // ...
    check(car: Car) {
        if (true) {
            car.stop();
        }
    }
}
class Green implements TrafficColor {
    // ...
    check(car: Car) {
        if (true) {
            car.drive();
        }
    }
}
```

Добавленная
if (true){}

- 2) заменяем `true` выражением, вызывающим базисный метод и сравнивающим результат с константными значениями;

Листинг 5.33. До

```
class Red implements TrafficColor {
    color() { return "red"; }
    check(car: Car) {
        if (true) {
            car.stop();
        }
    }
}
class Yellow implements
    TrafficColor {
    color() { return "yellow"; }
    check(car: Car) {
        if (true) {
            car.stop();
        }
    }
}
class Green implements
    TrafficColor {
    color() { return "green"; }
    check(car: Car) {
        if (true) {
            car.drive();
        }
    }
}
```

Листинг 5.34. После (2/8)

```
class Red implements TrafficColor {
    color() { return "red"; }
    check(car: Car) {
        if (this.color() === "red") {
            car.stop();
        }
    }
}
class Yellow implements
    TrafficColor {
    color() { return "yellow"; }
    check(car: Car) {
        if (this.color() === "yellow") {
            car.stop();
        }
    }
}
class Green implements
    TrafficColor {
    color() { return "green"; }
    check(car: Car) {
        if (this.color() === "green") {
            car.drive();
        }
    }
}
```

Проверяем
базисный метод

- 3) теперь копируем тело каждой версии и вместе с `else` вставляем его во все остальные версии.

Листинг 5.35. До

```
class Red implements TrafficColor {
    // ...
    check(car: Car) {
        if (this.color() === "red") {
            car.stop();
        }
    }
}
class Yellow implements
    TrafficColor {
    // ...
    check(car: Car) {
        if (this.color() === "red") {
            car.stop();
        }
    }
}
```

Листинг 5.36. После (3/8)

```
class Red implements TrafficColor {
    // ...
    check(car: Car) {
        if (this.color() === "red") {
            car.stop();
        } else if (this.color() === "yellow") {
            car.stop();
        } else if (this.color() === "green") {
            car.drive();
        }
    }
}
class Yellow implements
    TrafficColor {
    // ...
    check(car: Car) {
        if (this.color() === "red") {
            car.stop();
        } else if (this.color() === "yellow") {
            car.stop();
        } else if (this.color() === "green") {
            car.drive();
        }
    }
}
```

Копируем методы
друг в друга

```
// ...
check(car: Car) {
    if (this.color() ==
        "yellow") {
        car.stop();
    }
}

class Green implements
    TrafficColor {
    // ...
    check(car: Car) {

        if (this.color() ==
            "green") {
            car.drive();
        }
    }
}
```

```
// ...
check(car: Car) {
    if (this.color() == "red") {
        car.stop();
    } else if (this.color() == "yellow") {
        car.stop();
    } else if (this.color() == "green") {
        car.drive();
    }
}

class Green implements TrafficColor {
    // ...
    check(car: Car) {
        if (this.color() == "red") {
            car.stop();
        } else if (this.color() == "yellow") {
            car.stop();
        } else if (this.color() == "green") {
            car.drive();
        }
    }
}
```

Копируем методы
друг в друга

- Сейчас методы `check` стали равны и обработки требует только базисный метод. Второй этап начинается с введения поля для метода `color` и присваивания ему константы в конструкторе.

Листинг 5.37. До

```
class Red implements TrafficColor {

    color() { return "red"; }
    // ...
}

class Yellow implements
    TrafficColor {

    color() { return "yellow"; }
    // ...
}

class Green implements
    TrafficColor {

    color() { return "green"; }
    // ...
}
```

Листинг 5.38. После (4/8)

```
class Red implements TrafficColor {
    constructor(
        private col: string = "red") { }
    color() { return "red"; }
    // ...
}

class Yellow implements TrafficColor {
    constructor(
        private col: string = "yellow") { }
    color() { return "yellow"; }
    // ...
}

class Green implements TrafficColor {
    constructor(
        private col: string = "green") { }
    color() { return "green"; }
    // ...
}
```

Добавленные
конструкторы

3. Изменяем методы на возвращение новых полей вместо констант.

Листинг 5.39. До

```
class Red implements TrafficColor {
    // ...
    color() { return "red"; }
}
class Yellow implements TrafficColor {
    // ...
    color() { return "yellow"; }
}
class Green implements TrafficColor {
    // ...
    color() { return "green"; }
}
```

Листинг 5.40. После (5/8)

```
class Red implements TrafficColor {
    // ...
    color() { return this.col; }
}
class Yellow implements TrafficColor {
    // ...
    color() { return this.col; }
}
class Green implements TrafficColor {
    // ...
    color() { return this.col; }
}
```

Возвращает поле
вместо константы

4. Выполняем компиляцию, убеждаясь, что все по-прежнему работает.

5. В каждом классе для всех полей по очереди:

- 1) копируем предустановленное значение поля и делаем его параметром;

Листинг 5.41. До

```
class Red implements TrafficColor {
    constructor(
        private col: string = "red") { }
    // ...
}
```

Листинг 5.42. После (6/8)

```
class Red implements TrafficColor {
    constructor(
        private col: string) { }
    // ...
}
```

Вырезаем предустановленное
значение

- 2) прорабатываем все ошибки компилятора и вставляем предустановленное значение в качестве аргумента.

Листинг 5.43. До

```
function nextColor(t: TrafficColor) {
    if (t.color() === "red")
        return new Green();
    else if (t.color() === "green")
        return new Yellow();
    else if (t.color() === "yellow")
        return new Red();
}
```

Листинг 5.44. После (7/8)

```
function nextColor(t: TrafficColor) {
    if (t.color() === "red")
        return new Green();
    else if (t.color() === "green")
        return new Yellow();
    else if (t.color() === "yellow")
        return new Red("red");
}
```

Исправляем ошибку,
вставляя значение

6. Когда объединенные классы станут идентичны, удаляем их все, за исключением одного, и исправляем ошибки компиляции, переключаясь на этот оставшийся класс.

Листинг 5.45. До

```
function nextColor(t: TrafficColor) {
  if (t.color() === "red")
    return new Green();
  else if (t.color() === "green")
    return new Yellow();
  else if (t.color() === "yellow")
    return new Red();
}
class Yellow implements TrafficColor { ... }
class Green implements TrafficColor { ... }
```

Листинг 5.46. После (8/8)

```
function nextColor(t: TrafficColor) {
  if (t.color() === "red")
    return new Red("green");
  else if (t.color() === "green")
    return new Red("yellow");
  else if (t.color() === "yellow")
    return new Red("red");
}

```

Удаление классов
Yellow и Green

Итак, отметим два момента: на этой стадии интерфейс нам не нужен и следует не забыть переименовать Red. Необходимо также поработать над удалением if с else, возможно, с помощью шаблона рефакторинга, который нам предстоит рассмотреть очень скоро. Тем не менее обращаю ваше внимание, что мы уже успешно объединили три класса.

Листинг 5.47. До

```
function nextColor(t: TrafficColor) {
  if (t.color() === "red")
    return new Green();
  else if (t.color() === "green")
    return new Yellow();
  else if (t.color() === "yellow")
    return new Red();
}
interface TrafficColor {
  color(): string;
  check(car: Car): void;
}
class Red implements TrafficColor {
  color() { return "red"; }
  check(car: Car) { car.stop(); }
}
class Yellow implements TrafficColor {
  color() { return "yellow"; }
  check(car: Car) { car.stop(); }
}
class Green implements TrafficColor {
  color() { return "green"; }
  check(car: Car) { car.drive(); }
}
```

Листинг 5.48. После

```
function nextColor(t: TrafficColor) {
  if (t.color() === "red")
    return new Red("green");
  else if (t.color() === "green")
    return new Red("yellow");
  else if (t.color() === "yellow")
    return new Red("red");
}
interface TrafficColor {
  color(): string;
  check(car: Car): void;
}
class Red implements TrafficColor {
  constructor(private col: string) {}
  color() { return this.col; }
  check(car: Car) {
    if (this.color() === "red") {
      car.stop();
    } else if (this.color() === "yellow") {
      car.stop();
    } else if (this.color() === "green") {
      car.drive();
    }
  }
}
```

На данном этапе разумно извлечь эти три цвета в константы, чтобы избежать необходимости в их многократном повторном инстанцировании (создании экземпляров). К счастью, сделать это несложно.

Дополнительные материалы

Насколько мне известно, это первое описание данного процесса в качестве шаблона рефакторинга, поэтому некуда вас адресовать.

5.2. ОБЪЕДИНЕНИЕ ПРОСТЫХ УСЛОВИЙ

Для продолжения работы с `updateTile` нам нужно сделать тела некоторых `if` более похожими между собой. Взглянем на код.

Листинг 5.49. Начальный

```
function updateTile(x: number, y: number) {
  if (map[y][x].isStony()
      && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Stone(new Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isBoxy()
      && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Box(new Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isFallingStone()) {
    map[y][x] = new Stone(new Resting());
  } else if (map[y][x].isFallingBox()) {
    map[y][x] = new Box(new Resting());
  }
}
```

Введем методы для установки и сброса нового поля `falling`.

Листинг 5.50. После введения `drop` и `rest`

```
interface Tile {
  // ...
  drop(): void;
  rest(): void;
}
class Stone implements Tile {
  // ...
  drop() { this.falling = new Falling(); }
  rest() { this.falling = new Resting(); }
}
class Flux implements Tile {
  // ...
  drop() { }
  rest() { }
}
```

Новый метод для установки нового поля;
в большинстве классов пуст

Новый метод для сброса нового поля;
в большинстве классов пуст

Новый метод для сброса нового поля;
в большинстве классов пуст

Новый метод для установки нового поля; в большинстве классов пуст

Первым делом разберемся с `rest`, а потом и с `drop`. Можно использовать `rest` непосредственно в `updateTile`.

Листинг 5.51. До

```
function updateTile(x: number, y: number) {
  if (map[y][x].isStony()
    && map[y + 1][x].isAir()) {
    map[y+1][x] = new Stone(new Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isBoxy()
    && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Box(new Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isFallingStone()) {
    map[y][x] = new Stone(new Resting());
  } else if (map[y][x].isFallingBox()) {
    map[y][x] = new Box(new Resting());
  }
}
```

Листинг 5.52. После

```
function updateTile(x: number, y: number) {
  if (map[y][x].isStony()
    && map[y + 1][x].isAir()) {
    map[y+1][x] = new Stone(new Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isBoxy()
    && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Box(new Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isFallingStone()) {
    map[y][x].rest();
  } else if (map[y][x].isFallingBox()) {
    map[y][x].rest();
  }
}
```

Использует новый
вспомогательный метод

Итак, мы добились, чтобы тела последних двух `if` были одинаковыми. Когда две инструкции `if`, расположенные рядом, имеют одинаковые тела, их можно объединить простым добавлением `||` между двумя этими условиями.

Листинг 5.53. До

```
function updateTile(x: number, y: number) {
  if (map[y][x].isStony()
    && map[y + 1][x].isAir()) {
    map[y+1][x] = new Stone(new Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isBoxy()
    && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Box(new Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isFallingStone()) {
    map[y][x].rest();
  } else if (map[y][x].isFallingBox()) {
    map[y][x].rest();
  }
}
```

Листинг 5.54. После

```
function updateTile(x: number, y: number) {
  if (map[y][x].isStony()
    && map[y + 1][x].isAir()) {
    map[y+1][x] = new Stone(new Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isBoxy()
    && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Box(new Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isFallingStone()
    || map[y][x].isFallingBox()) {
    map[y][x].rest();
  }
}
```

Объединенное
условие

За это время мы уже привыкли к `||`, значит, не должно вызывать удивление, что выражение `||` сразу же помещается в классы, которые именуются согласно общей составляющей обоих имен методов: `isFalling`.

Я хочу повторить важный момент из главы 2. На протяжении этого процесса мы ничего не анализируем в программе, а просто следуем существующей структуре кода. Использованные приемы рефакторинга не требуют полного понимания фактических действий кода. А это, в свою очередь, чрезвычайно важно, потому

что если сначала разбираться во всем коде, то рефакторинг может оказаться весьма затратным. Тот факт, что некоторые шаблоны рефакторинга применимы без предварительного изучения кода, может сэкономить уйму драгоценного времени.

Получившийся же код будет выглядеть так.

Листинг 5.55. До

```
function updateTile(x: number, y: number) {
  if (map[y][x].isStony()
    && map[y + 1][x].isAir()) {
    map[y+1][x] = new Stone(new
Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isBoxy()
    && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Box(new
Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isFallingStone()
    || map[y][x].isFallingBox()) {
    map[y][x].rest();
  }
}
```

Листинг 5.56. После

```
function updateTile(x: number, y: number) {
  if (map[y][x].isStony()
    && map[y + 1][x].isAir()) {
    map[y+1][x] = new Stone(new
Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isBoxy()
    && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Box(new
Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isFalling()) {
    map[y][x].rest();
  }
}
```

Использует новый
вспомогательный метод

Несмотря на то что этот шаблон рефакторинга является одним из простейших в книге, именно его возможности позволяют использовать более мощные шаблоны. Без длительных вступлений с моей стороны встречайте один из таких шаблонов: «Совмещение инструкций if».

5.2.1. Шаблон рефакторинга «Совмещение инструкций if»

Описание

Этот шаблон уменьшает повторяемость за счет объединения последовательных if с одинаковыми телами. Обычно мы сталкиваемся с подобной ситуацией только в процессе целенаправленного рефакторинга, при котором намеренно создаем такую ситуацию, иначе просто было бы неестественно писать if с одинаковыми телами друг за другом. Польза данного шаблона в том, что он раскрывает находящуюся в двух выражениях связь, добавляя ||, а это, как мы уже могли убедиться, очень нам на руку.

Процесс

1. Убедиться, что тела действительно одинаковые.
2. Выбрать код между закрывающей скобкой первого if и открывающей скобкой else if, нажать Delete и вставить ||. Вставить после if открывающую скобку и закрывающую перед {. Скобки вокруг выражений мы всегда сохраняем, чтобы сохранить поведение кода.

Листинг 5.57. До

```
if (expression1) {
  // body
} else if (expression2) {
  // same body
}
```

Листинг 5.58. После

```
if ((expression1) || (expression2)) {
  // body
}
```

3. Если выражения простые, лишние скобки можно позже удалить или настроить редактор на автоматическое выполнение этого действия.

Пример

В этом примере прописана логика для определения, какие действия выполнять с накладной.

Листинг 5.59. Начальный

```
if (today.getDate() === 1 && account.getBalance() > invoice.getAmount()) {
  account.pay(bill);
} else if (invoice.isLastDayOfPayment() && invoice.isApproved()) {
  account.pay(bill);
}
```

Листинг 5.60. До

```
if (today.getDate() === 1
    && account.getBalance()
    > invoice.getAmount())
{
  account.pay(bill);
} else if (invoice.isLastDayOfPayment()
    && invoice.isApproved())
{
  account.pay(bill);
}
```

Листинг 5.61. После

```
if ((today.getDate() === 1
    && account.getBalance()
    > invoice.getAmount())
    || (invoice.isLastDayOfPayment()
    && invoice.isApproved()))
{
  account.pay(bill);
}
```

Условие первого if (в скобках)

Условие второго if (в скобках)

Дополнительные материалы

Многие разработчики в индустрии программирования рассматривают описанный процесс как само собой разумеющиеся рутинные операции. Поэтому думаю, что здесь эта техника впервые была представлена как официальный шаблон рефакторинга.

5.3. ОБЪЕДИНЕНИЕ СЛОЖНЫХ УСЛОВИЙ

Глядя на первый if в `updateTile`, мы понимаем, что он просто заменяет один камень воздухом и один воздух камнем. Это эквивалентно перемещению камня и определению его как падающего с помощью функции `drop`. То же касается и случая с ящиком.

Листинг 5.62. До

```
function updateTile(x: number, y:number) {
  if (map[y][x].isStony()
    && map[y + 1][x].isAir()) {
    map[y+1][x] = new Stone(new Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isBoxy()
    && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Box(new Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isFalling()) {
    map[y][x].rest();
  }
}
```

Листинг 5.63. После

```
function updateTile(x: number, y:
number) {
  if (map[y][x].isStony()
    && map[y + 1][x].isAir()) {
    map[y][x].drop();
    map[y + 1][x] = map[y][x];
    map[y][x] = new Air();
  } else if (map[y][x].isBoxy()
    && map[y + 1][x].isAir()) {
    map[y][x].drop();
    map[y + 1][x] = map[y][x];
    map[y][x] = new Air();
  } else if (map[y][x].isFalling()) {
    map[y][x].rest();
  }
}
```

Устанавливаем камень или ящик
как падающий, меняем местами
клетки и добавляем новый воздух

Теперь тела первых двух `if` одинаковы. Можно снова использовать «Совмещение инструкций `if`» для объединения этих `if` с помощью размещения `||` между условиями.

Листинг 5.64. До

```
function updateTile(x: number, y:number) {
  if (map[y][x].isStony()
    && map[y + 1][x].isAir()) {
    map[y][x].drop();
    map[y + 1][x] = map[y][x];
    map[y][x] = new Air();
  } else if (map[y][x].isBoxy()
    && map[y + 1][x].isAir()) {
    map[y][x].drop();
    map[y + 1][x] = map[y][x];
    map[y][x] = new Air();
  } else if (map[y][x].isFalling()) {
    map[y][x].rest();
  }
}
```

Листинг 5.65. После

```
function updateTile(x: number, y:
number) {
  if (map[y][x].isStony()
    && map[y + 1][x].isAir()
    || map[y][x].isBoxy()
    && map[y + 1][x].isAir()) {
    map[y][x].drop();
    map[y + 1][x] = map[y][x];
    map[y][x] = new Air();
  } else if (map[y][x].isFalling()) {
    map[y][x].rest();
  }
}
```

Совмещенные условия

Итоговое условие несколько усложнилось по сравнению с предыдущей похожей ситуацией. Значит, самое время рассмотреть обработку подобных условий.

5.3.1. Использование правил арифметики для условий

Условным выражением можно управлять тем же способом, который используется для большей части кода в книге: не разбираясь в его действиях. Если не вдаваться в теорию, то можно просто сказать, что `||` и `|` действуют подобно `+` (сложению),

а `&&` (и `&`) подобно $\times$ (умножению). Чтобы лучше это запомнить, можно использовать такой мнемонический прием: две полосы (`||`) способны составить $+$, а внутри `&` заключен $\times$ (знак умножения). Наглядно это показано на рис. 5.1. Так проще разобраться, когда нужно использовать скобки вокруг `||` и начать применять все стандартные арифметические правила.

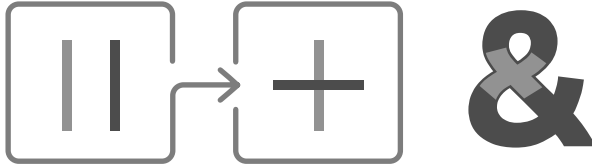


Рис. 5.1. Мнемоника для запоминания приоритета выполнения действий

Правила на рис. 5.2 применимы во всех случаях, когда условия имеют побочные эффекты. Чтобы иметь возможность применять эти правила ожидаемым образом, нужно всегда избегать использования в условиях побочных эффектов, сформулируем это как «использовать чистые условия».

$$a + b + c = (a + b) + c = a + (b + c)$$

$$a \cdot b \cdot c = (a \cdot b) \cdot c = a \cdot (b \cdot c)$$

$$a + b = b + a$$

$$a \cdot b = b \cdot a$$

$$a \cdot (b + c) = a \cdot b + a \cdot c$$

$$(a + b) \cdot c = a \cdot c + b \cdot c$$

Рис. 5.2. Арифметические правила

5.3.2. Правило «Использовать чистые условия»

Утверждение

Условия всегда должны быть чистыми.

Объяснение

Условия — это то, что следует за `if` или `while` и находится в середине циклов `for`. Их ответственность — принимать решения в зависимости от анализа заданных параметров любого типа.

«Чистые» означает, что условия не имеют побочных эффектов. *Побочные эффекты* — сопутствующие действия, например, когда условия одновременно с основной своей функцией присваивают значения переменным, выбрасывают

(throw) исключения либо взаимодействуют с вводом-выводом, то есть выводят что-либо, производят запись в файлы и т. п.

Чистота условий важна по нескольким причинам. Во-первых, как уже упоминалось, условия с побочными эффектами не позволяют нам использовать ранее описанные правила. Во-вторых, побочные эффекты нетипичны для условий, и мы не ожидаем, что они, эффекты, будут располагаться именно в условиях. Не ожидаем, но получается, что нам их нужно находить, затрачивая время и мыслительные ресурсы, отслеживать те условия, в которых такие побочные эффекты присутствуют.

Код, подобный приведенному ниже, довольно типичен. В нем `readLine` и возвращает следующую строку, и продвигает указатель. Продвижение указателя относится к побочным эффектам, значит, наше условие не является чистым. Более качественная его реализация справа разделяет ответственность получения строки и продвижения указателя. Эту реализацию можно дополнительно улучшить внедрением метода, который вместо возвращения `null` будет проверять, есть ли еще данные для считывания, но это уже отдельная тема.

Листинг 5.66. До

```
class Reader {
  private data: string[];
  private current: number;

  readLine() {
    this.current++;
    return this.data[this.current]
      || null;
  }
}
// ...
let br = new Reader();
let line: string | null;
while ((line = br.readLine())
  != null) {
  console.log(line);
}
```

Листинг 5.67. После

```
class Reader {
  private data: string[];
  private current: number;
  nextLine() {
    this.current++;
  }
  readLine() {
    return this.data[this.current] || null;
  }
}
// ...
let br = new Reader();
for(;br.readLine() != null;br.nextLine()){
  let line = br.readLine();
  console.log(line);
}
```

Новый метод с побочным эффектом

Побочный эффект устранен из метода

Преобразуем в цикл for, чтобы не забыть вызвать nextLine

Второй вызов для получения текущей строки

Заметьте, что `readLine` можно вызывать любое количество раз без побочных эффектов.

В случаях, когда у нас нет контроля над реализацией и, следовательно, мы не можем отделить возврат значения от выполнения побочных эффектов, можно использовать кэш. Реализовать кэш можно многими способами. Так что, не вдаваясь в подробности реализации, мы рассмотрим общий вид кэша, он может получать любой метод и отделять часть с выполнением действий побочных эффектов от возврата значения.

Листинг 5.68. Кэш

```

class Cacher<T> {
  private data: T;
  constructor(private mutator: () => T) {
    this.data = this.mutator();
  }
  get() {
    return this.data;
  }
  next() {
    this.data = this.mutator();
  }
}

let tmpBr = new Reader();
let br = new Cacher(() => tmpBr.readLine());
for (; br.get() !== null; br.next()) {
  let line = br.get();
  console.log(line);
}

```

Стандартное инстанцирование Reader,
но с временным именем

Обертывание конкретного
вызова в кэш

Запах

Это правило происходит из общего запаха, который гласит: «Отделять запросы от команд». Его можно найти в книге Ричарда Митчела и Джима Маккина *Design by contrast, by Example* (Addison-Wesley, 2001). Этот запах не похож на другие, так как выработать для него чутье несложно, в отличие от чутья на множество других запахов. В нем «команды» означают все действия с побочными эффектами, а «запросы» — все чистые коды. Самый простой способ следовать описанному условию запаха — это допускать побочные эффекты только в методах `void`: они либо имеют побочные эффекты, либо что-то возвращают, но не то и другое одновременно.

То есть единственным отличием общего запаха от правила является то, что мы сосредотачиваем внимание только на точке вызова, а не на точке определения. В оригинальном труде Митчел и Макким выстраивают на основе этого множество принципов, которые опираются на строгое отделение `return` от побочного эффекта во всех случаях. Мы же ослабили запах, чтобы сосредоточиться на условиях, потому что смешивание запросов и команд вне условий не влияет на возможность рефакторинга. Подчинение требованиям запаха больше относится к личным предпочтениям и хорошему стилю. При этом также довольно широко распространена и противоположная позиция: использование методов, которые и возвращают, и изменяют что-либо. Что ж, это позволяет нам попрактиковаться в обнаружении подобных моментов. По факту одним из самых используемых операторов в программировании является `++`, который и выполняет инкремент, и возвращает значение.

И отметим также, что можно держать пари: это правило уходит корнями в правило «Методы должны выполнять одно действие» из книги Роберта К. Мартина «Чистый код». Наличие побочного эффекта — это одно действие, а возвращение чего-либо — уже другое.

Намерение

Намерение состоит в разделении получения данных и изменения данных. Это делает код более чистым и предсказуемым. Кроме того, это позволяет использовать более удачное именование, потому что методы упрощаются. Побочные эффекты относятся к компонентам, способным изменять глобальное состояние, что весьма опасно, об этом говорилось в главе 2. Следовательно, отделение действий, направленных на изменение, упрощает управление ими.

Дополнительные материалы

Почитать о запросах и командах, а также об их использовании для создания утверждений, иногда называемых *контрактами*, вы можете в уже упомянутой книге Ричарда Митчела и Джима Маккима *Design by Contrast, by Example*.

5.3.3. Применение условной арифметики

Использование условий согласно правилам, представленным на рис. 5.2, очень эффективно. Вот, например, наше условие из `updateTile`: сначала мы преобразуем его в математическое уравнение, после чего можем легко использовать знакомые арифметические правила, чтобы упростить это условие и затем преобразовать обратно в код. Весь процесс преобразований показан ниже, на рис. 5.3.

$$\begin{array}{c}
 \begin{array}{ccc}
 \overbrace{\text{map}[y][x].\text{isStony}()}^a & \cdot & \overbrace{\text{map}[y+1][x].\text{isAir}()}^b \\
 \underbrace{|| \text{map}[y][x].\text{isBoxy}()}_c & \cdot & \underbrace{\text{map}[y+1][x].\text{isAir}()}_b \\
 + & & \cdot
 \end{array} \\
 = a \cdot b + c \cdot b \\
 = (a + c) \cdot b \\
 \hline
 \overbrace{(\text{map}[y][x].\text{isStony}() || \text{map}[y][x].\text{isBoxy}()))}^a & \cdot & \overbrace{\text{map}[y+1][x].\text{isAir}()}^b
 \end{array}$$

Рис. 5.3. Применение правил арифметики

Мысленно практикуя процесс преобразования условия в математическое уравнение, его упрощения и обратного изменения в код, можно выработать очень полезный навык, который окажется бесценным при решении более сложных задач в реальной жизни. Эта техника также позволит легче находить в условиях коварные ошибки со скобками.

ИСТОРИЯ ИЗ РЕАЛЬНОЙ ЖИЗНИ

Я так долго практиковал этот процесс, что довел его до автоматизма. Несколько раз за всю свою карьеру внештатного специалиста я получал приглашения в проект с единственным назначением — обнаруживать в условиях ошибки со скобками. Если вы не освоите данный прием, то подобные баги будет сложно выявить, а их эффекты окажутся непредсказуемыми.

Применив полученное ранее упрощение к коду, мы приходим к следующему.

Листинг 5.69. До

```
function updateTile(x: number, y:
  number) {
  if (map[y][x].isStony()
    && map[y + 1][x].isAir()
    || map[y][x].isBoxy()
    && map[y + 1][x].isAir()) {
    map[y][x].drop();
    map[y + 1][x] = map[y][x];
    map[y][x] = new Air();
  } else if (map[y][x].isFalling()) {
    map[y][x].rest();
  }
}
```

Листинг 5.70. После

```
function updateTile(x: number, y:
  number) {
  if (map[y][x].isStony()
    || map[y][x].isBoxy()
    && map[y + 1][x].isAir()) {
    map[y][x].drop();
    map[y + 1][x] = map[y][x];
    map[y][x] = new Air();
  } else if (map[y][x].isFalling()) {
    map[y][x].rest();
  }
}
```

Условие
упрощено

Теперь мы находимся в ситуации, аналогичной рассмотренной ранее: у нас есть оператор `||`, которой нужно поместить в класс. В главе 4 была связь между камнями и ящиками, и мы называли этот метод `pushable`. Но в текущей ситуации это имя смысла уже не имеет. Здесь важно не использовать прямолинейно старое имя только потому, что оно относится к той же связи: имя должно отражать контекст. Поэтому в данном случае лучше напишем новый метод — `canFall`.

После применения «Перемещения кода в классы» получаем очередное симпатичное упрощение.

Листинг 5.71. До

```
function updateTile(x: number, y:
  number) {
  if ((map[y][x].isStony()
    || map[y][x].isBoxy())
    && map[y + 1][x].isAir()) {
    map[y][x].drop();
    map[y + 1][x] = map[y][x];
    map[y][x] = new Air();
  } else if (map[y][x].isFalling()) {
    map[y][x].rest();
  }
}
```

Листинг 5.72. После

```
function updateTile(x: number, y:
  number) {
  if (map[y][x].canFall()
    && map[y + 1][x].isAir()) {
    map[y][x].drop();
    map[y + 1][x] = map[y][x];
    map[y][x] = new Air();
  } else if (map[y][x].isFalling()) {
    map[y][x].rest();
  }
}
```

Используем новый
вспомогательный метод

5.4. ОБЪЕДИНЕНИЕ КОДА СРЕДИ КЛАССОВ

Продолжим работу с `updateTile`. Нет ничего, что бы мешало переместить ее в классы.

Листинг 5.73. До

```
function updateTile(x: number, y:
    number) {
    if (map[y][x].canFall()
        && map[y + 1][x].isAir()) {
        map[y][x].drop();
        map[y + 1][x] = map[y][x];
        map[y][x] = new Air();
    } else if (map[y][x].isFalling()) {
        map[y][x].rest();
    }
}
```

Листинг 5.74. После

```
function updateTile(x: number, y:
    number) {
    map[y][x].update(x, y);
}
interface Tile {
    // ...
    update(x: number, y: number): void;
}
class Air implements Tile {
    // ...
    update(x: number, y: number) { }
}
class Stone implements Tile {
    // ...
    update(x: number, y: number) {
        if (map[y + 1][x].isAir()) {
            this.falling = new Falling();
            map[y + 1][x] = this;
            map[y][x] = new Air();
        } else if (this.falling.isFalling())
        {
            this.falling = new Resting();
        }
    }
}
```

Встроим `updateTile` с целью почистить код. Ведь к этому моменту мы переместили множество методов в классы, ввели множество методов в интерфейс. Самое время для выполнения промежуточной чистки с помощью «Пробного удаления с последующей компиляцией». Обратите внимание, что это удалит почти все методы `isX`, которые были введены. Те же, что останутся, имеют особое функциональное назначение, например, `isLockX` и `isAir` влияют на поведение других клеток.

Сейчас и в `Stone`, и в `Box` имеется в точности следующий код. В противоположность более ранней ситуации (см. раздел 4.6) сейчас нам не нужно расхождение кода. Поведение «падения» должно оставаться согласованным, к тому же мы сможем использовать его и далее, если будем вводить дополнительные клетки.

1. Сначала создаем класс `FallStrategy`.

Листинг 5.75. Новый класс

```
class FallStrategy {
}
```

2. Инстанцируем `FallStrategy` в конструкторе `Stone` и `Box`.

Листинг 5.76. До

```
class Stone implements Tile {
    constructor(
        private falling: FallingState)
    {
    }
    // ...
}
```

Листинг 5.77. После (1/5)

```
class Stone implements Tile {
    private fallStrategy: FallStrategy;
    constructor(
        private falling: FallingState)
    {
        this.fallStrategy = new FallStrategy();
    }
    // ...
}
```

Новое поле

Инициализируем новый метод

3. Мы перемещаем `update` так же, как делали это при «Перемещении кода в классы».

Листинг 5.78. До

```
class Stone implements Tile {
    // ...
    update(x: number, y: number) {
        if (map[y + 1][x].isAir()) {
            this.falling = new Falling();
            map[y + 1][x] = this;
            map[y][x] = new Air();
        } else if
            (this.falling.isFalling()) {
            this.falling = new Resting();
        }
    }
}
class FallStrategy {
}
```

Листинг 5.79. После (2/5)

```
class Stone implements Tile {
    update(x: number, y: number) {
        this.fallStrategy.update(x, y);
    }
}
class FallStrategy {
    update(x: number, y: number) {
        if (map[y + 1][x].isAir()) {
            this.falling = new Falling();
            map[y + 1][x] = this;
            map[y][x] = new Air();
        } else if (this.falling.isFalling()) {
            this.falling = new Resting();
        }
    }
}
```

4. Мы зависим от поля `falling`, поэтому поступаем следующим образом:

1) перемещаем поле `falling` и создаем ссылку доступа к нему в `FallStrategy`;

Листинг 5.80. До

```
class Stone implements Tile {
    private fallStrategy: FallStrategy;
    constructor(
        private falling: FallingState)
    {
        this.fallStrategy =
            new FallStrategy();
    }
    // ...
}
```

Листинг 5.81. После(3/5)

```
class Stone implements Tile {
    private fallStrategy: FallStrategy;
    constructor(
        falling: FallingState)
    {
        this.fallStrategy =
            new FallStrategy(falling);
    }
    // ...
}
```

Удаляем private

Добавляем аргумент

```
class FallStrategy {
    // ...
}
```

```
class FallStrategy {
    constructor(
        private falling: FallingState) {
        // ...
    }
    getFalling() { return this.falling; }
    // ...
}
```

Добавляем конструктор с параметром

Новая ссылка доступа для этого поля

3) исправляем ошибки в исходном классе, используя новые ссылки доступа.

Листинг 5.82. До

```
class Stone implements Tile {
    // ...
    moveHorizontal(dx: number) {
        this.falling
        .moveHorizontal(this, dx);
    }
}
```

Листинг 5.83. После (4/5)

```
class Stone implements Tile {
    // ...
    moveHorizontal(dx: number) {
        this.fallStrategy
        .getFalling()
        .moveHorizontal(this, dx);
    }
}
```

Используем новую ссылку доступа

5. Добавляем параметр `tile`, чтобы заменить `this` для оставшихся ошибок в `FallStrategy`.

Листинг 5.84. До

```
class Stone implements Tile {
    // ...
    update(x: number, y: number) {
        this.fallStrategy.update(x, y);
    }
}
class FallStrategy {
    update(x: number, y: number) {
        if (map[y + 1][x].isAir()) {
            this.falling = new Falling();
            map[y + 1][x] = this;
            map[y][x] = new Air();
        } else if (this.falling.isFalling()) {
            this.falling = new Resting();
        }
    }
}
```

Листинг 5.85. После (5/5)

```
class Stone implements Tile {
    // ...
    update(x: number, y: number) {
        this.fallStrategy.update(this, x, y);
    }
}
class FallStrategy {
    update(tile: Tile, x: number, y: number) {
        if (map[y + 1][x].isAir()) {
            this.falling = new Falling();
            map[y + 1][x] = tile;
            map[y][x] = new Air();
        } else if (this.falling.isFalling()) {
            this.falling = new Resting();
        }
    }
}
```

Добавление параметра на замену this

В итоге получаем такую трансформацию.

Листинг 5.86. До

```
class Stone implements Tile {
    constructor(private falling:
        FallingState)
    {
    }
    // ...
    update(x: number, y: number) {
        if (map[y + 1][x].isAir()) {
            this.falling = new Falling();
            map[y + 1][x] = this;
            map[y][x] = new Air();
        } else if
            (this.falling.isFalling()) {
            this.falling = new Resting();
        }
    }
}
```

Листинг 5.87. После

```
class Stone implements Tile {
    private fallStrategy: FallStrategy;
    constructor(falling: FallingState)
    {
        this.fallStrategy =
            new FallStrategy(falling);
    }
    // ...
    update(x: number, y: number) {
        this.fallStrategy.update(this, x, y);
    }
}
class FallStrategy {
    constructor(private falling: FallingState)
    { }
    isFalling() { return this.falling; }
    update(tile: Tile, x: number, y: number) {
        if (map[y + 1][x].isAir()) {
            this.falling = new Falling();
            map[y + 1][x] = tile;
            map[y][x] = new Air();
        } else if (this.falling.isFalling()) {
            this.falling = new Resting();
        }
    }
}
```

При внимательном рассмотрении `else if` в `FallStrategy.update` можно заметить, что если `falling` является `true`, то условие устанавливает значение `false`; в противном случае оно уже установлено как `false`. Следовательно, это условие можно удалить.

Листинг 5.88. До

```
class FallStrategy {
    // ...
    update(tile: Tile, x: number, y:
        number) {
        if (map[y + 1][x].isAir()) {
            this.falling = new Falling();
            map[y + 1][x] = tile;
            map[y][x] = new Air();
        } else if
            (this.falling.isFalling()) {
            this.falling = new Resting();
        }
    }
}
```

Листинг 5.89. После

```
class FallStrategy {
    // ...
    update(tile: Tile, x: number, y:
        number) {
        if (map[y + 1][x].isAir()) {
            this.falling = new Falling();
            map[y + 1][x] = tile;
            map[y][x] = new Air();
        } else {
            this.falling = new Resting();
        }
    }
}
```

Условие
удалено

Теперь код присваивает `falling` по всем путям, значит, его можно вынести за границы `if`. Мы также удаляем пустую инструкцию `else`. Получилось условие `if`, проверяющее значение, совпадающее с переменной. В подобных случаях предпочтительно использовать непосредственно переменную.

Листинг 5.90. До

```
class FallStrategy {
    // ...
    update(tile: Tile, x: number, y:
        number) {
        if (map[y + 1][x].isAir()) {
            this.falling = new Falling();
            map[y + 1][x] = tile;
            map[y][x] = new Air();
        } else {
            this.falling = new Resting();
        }
    }
}
```

Листинг 5.91. После

```
class FallStrategy {
    // ...
    update(tile: Tile, x: number, y:
        number) {
        this.falling = map[y + 1][x].isAir()
            ? new Falling()
            : new Resting();
        if (this.falling.isFalling()) {
            map[y + 1][x] = tile;
            map[y][x] = new Air();
        }
    }
}
```

Вынесение `this.falling`
за пределы `if`

Мы вписались в пять строк! Но это еще не конец. Помните, что есть правило, утверждающее «`if` только в начале» (3.5.1). Нам по-прежнему нужно ему следовать, поэтому выполним простое «Извлечение метода» (3.2.1).

Листинг 5.92. До

```
class FallStrategy {
    // ...
    update(tile: Tile, x: number, y:
        number) {
        this.falling = map[y + 1][x].isAir()
            ? new Falling()
            : new Resting();
        if (this.falling.isFalling()) {
            map[y + 1][x] = tile;
            map[y][x] = new Air();
        }
    }
}
```

Листинг 5.93. После

```
class FallStrategy {
    // ...
    update(tile: Tile, x: number, y:
        number) {
        this.falling = map[y + 1][x].isAir()
            ? new Falling()
            : new Resting();
        this.drop(tile, x, y);
    }
    private drop(tile: Tile,
        x: number, y: number)
    {
```

Извлеченный
метод

Встраиваем `updateTile`, выполняем компиляцию, тестируем, делаем коммит и... делаем перерыв.

Шаблон рефакторинга, который был применен для объединения «кода падения», называется «Введение паттерна «Стратегия»». Это самый запутанный шаблон в книге. Здесь и далее мы прибегаем к нему во многих местах. Стоит сказать, что обычно принято, кроме всего прочего, демонстрировать эффект этого шаблона с помощью диаграмм. Не хочется нарушать традицию и сейчас, но сначала нужно немного отвлечься для знакомства с диаграммами классов UML.

5.4.1. Введение диаграмм классов UML для отражения связи классов

Иногда нам требуется описать характеристики кода, такие как его архитектура или порядок выполнения действий. Некоторые из этих характеристик легче всего формулировать с помощью диаграмм. Для формирования диаграмм служит фреймворк под названием Unified Modelling Language (UML).

UML включает много типов стандартных диаграмм для передачи конкретных характеристик кода. Вот лишь некоторые примеры: диаграммы последовательностей действий, диаграммы классов и диаграммы деятельности. Описание всех возможных типов не является целью этой книги. Характеристики паттерна «Стратегия» и ряда других паттернов чаще всего демонстрируются с помощью определенного типа UML-диаграммы, а именно *диаграммы классов*. Я поставил перед собой цель добиться, чтобы после прочтения этой книги вы могли взять любую другую, посвященную чистому коду или рефакторингу, и смогли разобраться в ней. Поэтому в данном разделе я опишу, как работают диаграммы классов.

Диаграммы классов отражают структуры интерфейсов и классов, а также связи между ними. Классы в этих диаграммах представляются с помощью прямоугольных блоков, в которых прописывается название и иногда методы, но очень редко поля. Интерфейсы представляются аналогично классам, но над названием дополнительно указывается слово `interface`. Можно также обозначить, являются ли методы и поля `private (-)` или `public (+)`. На рис. 5.4 показано, как небольшой класс с полями и методами оформляется на диаграмме.

Листинг 5.94. Завершенный класс

```
class Cls {
    private text: string = "Hello";
    public name: string;
    private getText() { return this.text;
}
    printText() {
console.log(this.getText()); }
}
```

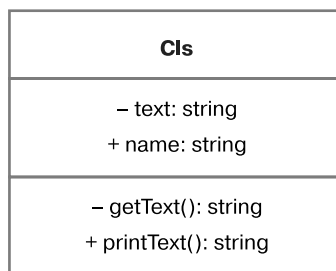


Рис. 5.4. Диаграмма класса

В большинстве случаев интересно говорить только о публичном интерфейсе класса. Поэтому никакие закрытые (внутренние, частные) объекты мы обычно не включаем в рассмотрение. В данном же примере большинство полей закрыты по веской причине, о которой говорится в следующей главе. Поскольку мы чаще всего отражаем только публичные методы, то и видимость добавлять не требуется.

Важнейшей частью диаграммы классов являются связи между классами и интерфейсами. Они разделяются на три категории: «X использует Y», «X является Y» и «X содержит Y» или «X содержит несколько Y». В каждой из этих категорий два типа стрелок обозначают несколько различающиеся между собой понятия. Все виды связей, отражаемые на диаграмме классов, показаны на рис. 5.5.



Рис. 5.5. Связи в UML

Эту систему обозначений можно немного упростить. Правило «Наследовать только от интерфейсов» (4.3.2) не позволяет нам использовать стрелку наследования. Стрелки «Использует» обычно задействуются, когда нам неизвестно, или неинтересно, какой по сути является связь. Разница между композицией и агрегацией, по большому счету, только эстетическая. Следовательно, чаще всего можно обойтись двумя типами связей: композицией и реализацией (рис. 5.6, 5.7). Вот два простых случая применения классов и диаграмм.

Листинг 5.95. Реализует

```
interface A {
    m(): void;
}
class B implements A {
    m() { console.log("Hello"); }
}
```

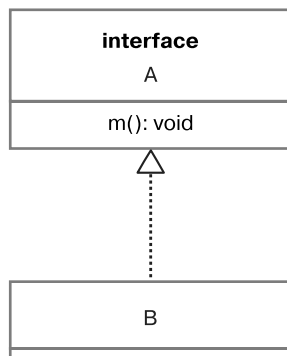


Рис. 5.6. Реализация

Заметьте, не нужно показывать, что В также содержит метод `m`, так как интерфейс об этом уже сообщает.

Листинг 5.96. Составной

```
class A {
    private b: B;
}
class B {
}
```



Рис. 5.7. Композиция

Диаграммы классов для целой программы оказываются чрезмерно перегруженными и, следовательно, бесполезны. Визуальные представления с помощью диаграмм используются в основном для демонстрации паттернов проектирования или небольших частей архитектуры ПО, поэтому включать в них целесообразно только важные методы. На рис. 5.8 показана диаграмма классов, сфокусированная на `FallStrategy`.

Теперь, вооружившись знанием об использовании диаграмм классов, можно проиллюстрировать эффект от «Введения паттерна «Стратегия»».

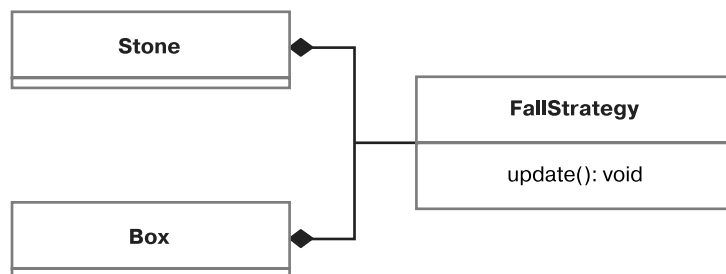


Рис. 5.8. Диаграмма класса с фокусом на `FallStrategy`

5.4.2. Шаблон рефакторинга «Введение паттерна «Стратегия»»

Описание

В предыдущих главах уже упоминалось, что инструкция `if` является низкоуровневым оператором потока управления. Были также отмечены преимущества использования объектов. Принцип введения вариативности путем инстанцирования другого класса называется паттерном «Стратегия». Обычно он выражается в виде диаграммы классов, аналогичной предыдущим (рис. 5.9).

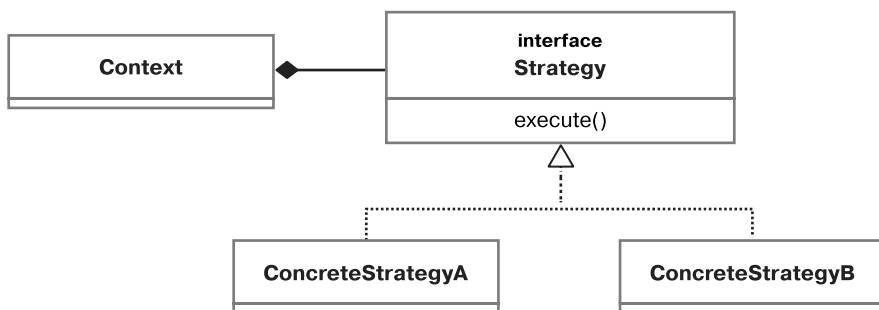


Рис. 5.9. Паттерн «Стратегия» в виде диаграммы классов

Многие паттерны являются вариациями паттерна «Стратегия». Если у нашей стратегии есть поля, то это уже другой паттерн — *паттерн «Состояние»*. Различия между этими двумя паттернами скорее академические — их понимание придает внешнюю солидность нашим высказываниям, а на деле редко оказывается пригодным. Практическая основная идея остается одна: получить возможность вносить изменения путем добавления классов (преимущества этого мы обсуждали в главе 2). Так что для описания перемещения любого кода в отдельный класс оказывается достаточно термина «*паттерн «Стратегия»*». И даже если в итоге эта возможность внесения вариативности не найдет применения, мы все-таки будем осознавать, что она у нас имеется.

Обратите внимание, что эта техника отличается от преобразования кода типов в классы. В том случае классы представляли собой данные, и поэтому мы могли передавать в них множество методов. Что же касается классов «Стратегии», то в них редко добавляются методы после завершения создания ПО. Вместо этого в случае изменения функциональности предпочтительным оказывается создание нового класса.

Поскольку целью паттерна «Стратегия» является внесение вариативности, он всегда реализуется с помощью наследования: обычно от интерфейса, но иногда и от абстрактного класса. Мы уже разбирали недостатки наследования, но не использовали его.

Вариативность паттерна «Стратегия» представляет крайнюю форму динамического связывания. В процессе выполнения этот паттерн позволяет загружать классы, которые нашему коду совершенно неизвестны, плавно интегрируя их в поток управления, — и не возникает необходимости даже выполнять перекомпиляцию. Если брать из этой книги лишь что-то одно, то лучше всего будет освоить именно паттерн «Стратегия».

Поводом для его введения могут быть две ситуации. Во-первых, мы можем выполнять рефакторинг для внесения в код вариативности. В этом случае в конце

у нас должен получиться интерфейс. Чтобы максимально ускорить такой рефакторинг, рекомендуется интерфейс на время отложить в сторонку. Во-вторых, как, например, в ситуации с «кодом падения», мы можем не предполагать внесение вариативности. Нам просто бывает нужно объединить поведение кода между классами.

У нас есть правило «Избегать интерфейсов с единственной реализацией» (5.4.3). Когда окажется нужен интерфейс — сразу же или позднее, — мы используем шаблон «Извлечение интерфейса из реализации» (5.4.4). И правило, и шаблон я объясню ниже.

Процесс

1. Выполняем извлечение метода для кода, который намереваемся изолировать. Если нужно объединить его с чем-то еще, то стоит убедиться в идентичности методов.
2. Создаем новый класс.
3. Инстанцируем этот новый класс в конструкторе.
4. Перемещаем метод в новый класс.
5. Если в каких-либо полях есть зависимости:
 - 1) перемещаем эти поля в новый класс, создавая для них ссылки доступа;
 - 2) исправляем ошибки в исходном классе, используя новые ссылки доступа.
6. Добавляем параметр на замену `this` для оставшихся в новом классе ошибок.
7. Встраиваем метод (4.1.7), чтобы обратить извлечение из шага 1.

Пример

В данном сценарии представим, что у нас есть два класса, которые могут обрабатывать массив по пакетам, то есть мы можем передавать им небольшие массивы, являющиеся срезами — или пакетами — более крупного массива. Эта ситуация типична при работе с объемами данных, превышающими вместимость RAM, или при их стриминге. В таком случае у нас есть один обработчик пакетов для поиска минимального элемента и еще один для поиска суммы.

Листинг 5.97. Начальный

```
class ArrayMinimum {
  constructor(private accumulator: number) {
  }
  process(arr: number[]) {
    for (let i = 0; i < arr.length; i++)
```

```

        if (this.accumulator > arr[i])
            this.accumulator = arr[i];
        return this.accumulator;
    }
}
class ArraySum {
    constructor(private accumulator: number) {
    }
    process(arr: number[]) {
        for (let i = 0; i < arr.length; i++)
            this.accumulator += arr[i];
        return this.accumulator;
    }
}

```

Эти обработчики пакетов похожи, но не идентичны. Я продемонстрирую, как одновременно извлекать «Стратегию» из обоих, чтобы подготовить классы к последующему объединению.

1. Выполним извлечение метода для кода, который хотим изолировать. Поскольку в итоге наша цель — объединить два класса, то убеждаемся, что их методы идентичны.

Листинг 5.98. До

```

class ArrayMinimum {
    constructor(private accumulator: number) {
    }
    process(arr: number[]) {
        for (let i = 0; i < arr.length; i++)
            if (this.accumulator > arr[i])
                this.accumulator = arr[i];
        return this.accumulator;
    }
}
class ArraySum {
    constructor(private accumulator: number) {
    }
    process(arr: number[]) {
        for (let i = 0; i < arr.length; i++)
            this.accumulator += arr[i];
        return this.accumulator;
    }
}

```

Листинг 5.99. После (1/7)

```

class ArrayMinimum {
    constructor(private accumulator: number) {
    }
    process(arr: number[]) {
        for (let i = 0; i < arr.length; i++)
            this.processElement(arr[i]);
        return this.accumulator;
    }
    processElement(e: number) {
        if (this.accumulator > e)
            this.accumulator = e;
    }
}
class ArraySum {
    constructor(private accumulator: number) {
    }
    process(arr: number[]) {
        for (let i = 0; i < arr.length; i++)
            this.processElement(arr[i]);
        return this.accumulator;
    }
    processElement(e: number) {
        this.accumulator += e;
    }
}

```

Извлеченные метод и вызов

Извлеченные метод и вызов

2. Создаем новые классы.

Листинг 5.100. После (2/7)

```
class MinimumProcessor {
}
class SumProcessor {
}
```

3. Инстанцируем созданные классы в конструкторах.

Листинг 5.101. До

```
class ArrayMinimum {
    constructor(private accumulator:
        number) {
    }
    // ...
}
class ArraySum {
    constructor(private accumulator:
        number) {
    }
    // ...
}
```

Листинг 5.102. После (3/7)

```
class ArrayMinimum {
    private processor: MinimumProcessor;
    constructor(private accumulator: number) {
        this.processor = new MinimumProcessor();
    }
    // ...
}
class ArraySum {
    private processor: SumProcessor;
    constructor(private accumulator: number) {
        this.processor = new SumProcessor();
    }
    // ...
}
```

Добавление поля
и инициализация его
в конструкторе

4. Перемещаем методы в `MinimumProcessor` и `SumProcessor` соответственно.

Листинг 5.103. До

```
class ArrayMinimum {
    // ...
    processElement(e: number) {
        if (this.accumulator > e)
            this.accumulator = e;
    }
}
class ArraySum {
    // ...
    processElement(e: number) {
        this.accumulator += e;
    }
}
class MinimumProcessor {
}
class SumProcessor {
}
```

Листинг 5.104. После (4/7)

```
class ArrayMinimum {
    // ...
    processElement(e: number) {
        this.processor.processElement(e);
    }
}
class ArraySum {
    // ...
    processElement(e: number) {
        this.processor.processElement(e);
    }
}
class MinimumProcessor {
    processElement(e: number) {
        if (this.accumulator > e)
            this.accumulator = e;
    }
}
}
```

Вызываем метод
в классе

Новый метод

```
class SumProcessor {
    processElement(e: number) {
        this.accumulator += e;
    }
}
```

Новый метод

6. Поскольку в обоих случаях наблюдается зависимость от поля `accumulator`, выполняем следующие шаги:

- 1) перемещаем поле `accumulator` в классы `MinimumProcessor` и `SumProcessor`, создавая для них ссылки доступа;

Листинг 5.105. До

```
class ArrayMinimum {
    private processor: MinimumProcessor;
    constructor(private accumulator: number) {
        this.processor =
            new MinimumProcessor();
    }
    // ...
}
class ArraySum {
    private processor: SumProcessor;
    constructor(private accumulator: number) {
        this.processor =
            new SumProcessor();
    }
    // ...
}
class MinimumProcessor {
    // ...
}
class SumProcessor {
    // ...
}
```

Листинг 5.106. После (5/7)

```
class ArrayMinimum {
    private processor: MinimumProcessor;
    constructor(accumulator: number) {
        this.processor =
            new MinimumProcessor(accumulator);
    }
    // ...
}
class ArraySum {
    private processor: SumProcessor;
    constructor(accumulator: number) {
        this.processor =
            new SumProcessor(accumulator);
    }
    // ...
}
class MinimumProcessor {
    constructor(private accumulator: number) {
    }
    getAccumulator() {
        return this.accumulator;
    }
    // ...
}
class SumProcessor {
    constructor(private accumulator: number) {
    }
    getAccumulator() {
        return this.accumulator;
    }
    // ...
}
```

Ссылка доступа для получения поля

Перемещаем поле

- 2) исправляем ошибки в исходном классе, используя новые ссылки доступа.

Листинг 5.107. До

```
class ArrayMinimum {
  // ...
  process(arr: number[]) {
    for (let i = 0; i < arr.length; i++)
      this.processElement(arr[i]);
    return this.accumulator;
  }
}
class ArraySum {
  // ...
  process(arr: number[]) {
    for (let i = 0; i < arr.length; i++)
      this.processElement(arr[i]);
    return this.accumulator;
  }
}
```

Листинг 5.108. После (6/7)

```
class ArrayMinimum {
  // ...
  process(arr: number[]) {
    for (let i = 0; i < arr.length; i++)
      this.processElement(arr[i]);
    return
    this.processor.getAccumulator(); }
}
class ArraySum {
  // ...
  process(arr: number[]) {
    for (let i = 0; i < arr.length; i++)
      this.processElement(arr[i]);
    return
    this.processor.getAccumulator(); }
}
```

Использование ссылки
доступа для получения поля

6. Добавляем параметр на замену `this` для исправления оставшихся в новых классах ошибок (хотя в данном примере это не обязательно, поскольку ошибок в этих классах нет).
7. Встраиваем метод для обращения извлечения из шага 1.

Листинг 5.109. До

```
class ArrayMinimum {
  // ...
  process(arr: number[]) {
    for (let i = 0; i < arr.length; i++)
      this.processElement(arr[i]);
    return
    this.processor.getAccumulator();
  }
  processElement(e: number) {
    this.processor.processElement(e);
  }
}
class ArraySum {
  // ...
  process(arr: number[]) {
    for (let i = 0; i < arr.length; i++)
      this.processElement(arr[i]);
    return
    this.processor.getAccumulator();
  }
  processElement(e: number) {
    this.processor.processElement(e);
  }
}
```

Листинг 5.110. После (7/7)

```
class ArrayMinimum {
  // ...
  process(arr: number[]) {
    for (let i = 0; i < arr.length; i++)
      this.processor
        .processElement(arr[i]);
    return this.processor
      .getAccumulator();
  }
}
class ArraySum {
  // ...
  process(arr: number[]) {
    for (let i = 0; i < arr.length; i++)
      this.processor
        .processElement(arr[i]);
    return this.processor
      .getAccumulator();
  }
}
```

Метод
processElement
встроен

processElement удален

Сейчас два исходных класса, `ArrayMinimum` и `ArraySum`, идентичны, за исключением разве что инстанцирования в конструкторе. Это можно исправить, используя шаблон «Извлечение интерфейса из реализации», с которым мы вскоре познакомимся, и передачей его в качестве параметра.

Листинг 5.111. До

```
class ArrayMinimum {
    constructor(private accumulator:
        number) {

    }
    process(arr: number[]) {
        for (let i = 0; i < arr.length; i++)
            if (this.accumulator > arr[i])
                this.accumulator = arr[i];
        return this.accumulator;
    }
}
class ArraySum {
    constructor(private accumulator:
        number) {

    }
    process(arr: number[]) {
        for (let i = 0; i < arr.length; i++)
            this.accumulator += arr[i];
        return this.accumulator;
    }
}
```

Листинг 5.112. После

```
class ArrayMinimum {
    private processor: MinimumProcessor;
    constructor(accumulator: number) {
        processor =
            new MinimumProcessor(accumulator);
    }
    process(arr: number[]) {
        for (let i = 0; i < arr.length; i++)
            this.processor.processElement(arr[i]);
        return this.processor.getAccumulator();
    }
}
class ArraySum {
    private processor: SumProcessor;
    constructor(accumulator: number) {
        processor =
            new SumProcessor(accumulator);
    }
    process(arr: number[]) {
        for (let i = 0; i < arr.length; i++)
            this.processor.processElement(arr[i]);
        return this.processor.getAccumulator();
    }
}
class MinimumProcessor {
    constructor(private accumulator: number) {
    }
    getAccumulator() {
        return this.accumulator;
    }
    processElement(e: number) {
        if (this.accumulator > e)
            this.accumulator = e;
    }
}
class SumProcessor {
    constructor(private accumulator: number) {
    }
    getAccumulator() {
        return this.accumulator;
    }
    processElement(e: number) {
        this.accumulator += e;
    }
}
```

Дополнительные материалы

Впервые паттерн «Стратегия» был введен в книге «Паттерны объектно-ориентированного проектирования» за авторством «Банды четырех». Ввиду его эффективности этот паттерн встречается довольно часто. А вот идея внедрения паттерна «Стратегия» в код впервые высказана в книге Мартина Фаулера «Чистый код». Обе книги уже упоминались ранее.

5.4.3. Правило «Избегать интерфейсов с единственной реализацией»

Утверждение

Никогда не использовать интерфейсы, у которых всего одна реализация.

Объяснение

Это правило утверждает, что у нас не должно быть интерфейсов с единственной реализацией. Эти интерфейсы могут появляться, если следовать распространенному совету «Всегда писать код на уровне интерфейсов». Тем не менее такой подход зачастую может оказаться невыгодным.

Объясняется это просто, а именно так: интерфейс с одной реализацией не повышает читаемости. Еще хуже то, что интерфейс сигнализирует о вариативности. А если ее нет, то он лишь нагружает нашу ментальную модель. Он также может замедлять работу в случае, когда появляется необходимость изменить реализующий класс, поскольку тогда приходится обновлять и интерфейс, что требует особой осторожности. Эти соображения подобны аргументам из описания шаблона «Специализация метода» (4.2.2). Интерфейсы с одним реализующим классом представляют форму обобщения, которая не несет никакой пользы.

Во многих языках мы помещаем интерфейсы в их собственный отдельный файл. Так вот, присутствие интерфейса с одним реализующим классом требует двух файлов, в то время как наличие только реализующего класса занимает всего один файл. Разница в один файл не представляет, казалось бы, серьезной проблемы, но если наша база имеет склонность к интерфейсам всего с одним потомком, то количество файлов может увеличиться в два раза, а это уже существенно повысит умственную нагрузку.

Бывают случаи, когда есть смысл использовать интерфейсы без реализаций. Они оказываются нужны, когда стоит задача создать анонимные классы, чаще всего используемые для компараторов или же для более строгой инкапсуляции посредством анонимных внутренних классов. Инкапсуляцию мы будем разбирать в следующей главе. А что касается анонимных внутренних классов, то, поскольку на практике они используются редко, их описание выходит за рамки этой книги.

Запах

Существует известное утверждение: «Любую задачу в компьютерной науке можно решить внесением дополнительного уровня абстракции». Именно в этом и состоит суть интерфейсов. Мы прячем детали за абстракцией. Джон Кармак был превосходным ведущим программистом *Doom*, *Quake* и нескольких других игр. Он разъяснил запах, из которого проистекает это правило, в одном из твитов: «Абстракция — это компромисс в виде повышения реальной сложности ради уменьшения воспринимаемой». Подразумевается, что с абстракциями следует быть осторожными.

Намерение

Намерение состоит в ограничении ненужного шаблонного кода. Интерфейсы — это типичный его источник. Они особенно опасны, так как многих разработчиков учили, что как раз интерфейсам нужно отдавать предпочтение. В результате эти разработчики теперь склонны раздувать свои приложения.

Дополнительная информация

Фред Джордж упомянул похожее правило в своем выступлении на конференции GOTO 2015 — он назвал его *The Secret Assumption of Agile*.

5.4.4. Шаблон рефакторинга «Извлечение интерфейса из реализации»

Описание

Это еще один простой шаблон. Его польза состоит в том, что он позволяет нам откладывать создание интерфейсов до момента необходимости (когда нам потребуется внести вариативность).

Процесс

1. Создаем новый интерфейс с тем же именем, что и класс, из которого будем производить извлечение.
2. Переименовываем класс, из которого извлекаем интерфейс, и прописываем в нем реализацию этого нового интерфейса.
3. Выполняем компиляцию и проходим по ошибкам:
 - 1) если ошибка вызвана `new`, изменяем инстанцирование на новое имя класса;
 - 2) в противном случае добавляем вызывающий ошибку метод в интерфейс.

Пример

Продолжим работать с предыдущим примером, сосредоточившись на `SumProcessor`.

Листинг 5.113. Начальный

```

class ArraySum {
  private processor: SumProcessor;
  constructor(accumulator: number) {
    processor = new SumProcessor(accumulator);
  }
  process(arr: number[]) {
    for (let i = 0; i < arr.length; i++)
      this.processor.processElement(arr[i]);
    return this.processor.getAccumulator();
  }
}
class SumProcessor {
  constructor(private accumulator: number) { }
  getAccumulator() { return this.accumulator; }
  processElement(e: number) {
    this.accumulator += e;
  }
}

```

Следуя процессу, мы делаем так.

1. Создаем новый интерфейс с тем же именем, что и класс, из которого производим извлечение.

Листинг 5.114. Добавление нового интерфейса

```

interface SumProcessor {
}

```

2. Переименовываем класс, из которого хотим извлечь интерфейс, и прописываем в нем реализацию нового интерфейса.

Листинг 5.115. До

```

class SumProcessor {
  // ...
}

```

Листинг 5.116. После (1/3)

```

class TmpName implements SumProcessor {
  // ...
}

```

3. Выполняем компиляцию и рассматриваем ошибки:

- 1) если ошибка в `new`, изменяем instantiation на новое имя класса;

Листинг 5.117. До

```

class ArraySum {
  private processor: SumProcessor;
  constructor(accumulator: number) {
    processor =
      new SumProcessor(accumulator);
  }
  // ...
}

```

Листинг 5.118. После (2/3)

```

class ArraySum {
  private processor: SumProcessor;
  constructor(accumulator: number) {
    processor =
      new TmpName(accumulator);
  }
  // ...
}

```

←
Инстанцируем класс
вместо интерфейса

2) в противном случае добавляем вызывающий ошибку метод в интерфейс.

Листинг 5.119. До

```
class ArraySum {
    // ...
    process(arr: number[]) {
        for (let i = 0; i < arr.length; i++)
            this.processor.processElement(arr[i]);
        return this.processor.getAccumulator();
    }
}
interface SumProcessor {
}
```

Листинг 5.120. После (3/3)

```
class ArraySum {
    // ...
    process(arr: number[]) {
        for (let i = 0; i < arr.length; i++)
            this.processor.processElement(arr[i]);
        return this.processor.getAccumulator();
    }
}
interface SumProcessor {
    processElement(e: number): void;
    getAccumulator(): number;
}
```

Добавление методов
в интерфейс

Теперь, когда все работает, нужно дать интерфейсу более подходящее имя, например `ElementProcessor`, и переименовать класс обратно в `SumProcessor`. Кроме того, можно прописать в уже знакомом нам `MinimumProcessor` реализацию этого интерфейса, а затем заменить процессором (`MinimumProcessor`) параметр `accumulator` в `ArraySum` и переименовать его в `BatchProcessor`. В итоге два обработчика пакетов станут идентичными, и один из них можно будет удалить. Выполнение всех описанных действий приведет к следующему коду.

Листинг 5.121. После

```
class BatchProcessor {
    constructor(private processor: ElementProcessor) { }
    process(arr: number[]) {
        for (let i = 0; i < arr.length; i++)
            this.processor.processElement(arr[i]);
        return this.processor.getAccumulator();
    }
}
interface ElementProcessor {
    processElement(e: number): void;
    getAccumulator(): number;
}
class MinimumProcessor implements ElementProcessor {
    constructor(private accumulator: number) { }
    getAccumulator() { return this.accumulator; }
    processElement(e: number) {
        if (this.accumulator > e)
            this.accumulator = e;
    }
}
class SumProcessor implements ElementProcessor {
    constructor(private accumulator: number) { }
    getAccumulator() { return this.accumulator; }
    processElement(e: number) {
        this.accumulator += e;
    }
}
```

Дополнительные материалы

Насколько мне известно, данная техника в качестве шаблона рефакторинга описывается впервые.

5.5. ОБЪЕДИНЕНИЕ ПОХОЖИХ ФУНКЦИЙ

Еще одно место, где у нас есть схожий код, — это две функции, `removeLock1` и `removeLock2`.

Листинг 5.122. `removeLock1`

```
function removeLock1() {
  for (let y = 0; y < map.length; y++) {
    for
      (let x = 0; x < map[y].length; x++){
        if (map[y][x].isLock1()) {
          map[y][x] = new Air();
        }
      }
    }
  }
}
```

Листинг 5.123. `removeLock2`

```
function removeLock2() {
  for (let y = 0; y < map.length; y++) {
    for
      (let x = 0; x < map[y].length; x++){
        if (map[y][x].isLock2()) {
          map[y][x] = new Air();
        }
      }
    }
  }
}
```

Единственное
отличие

Оказывается, их также можно объединить с помощью «Введения паттерна “Стратегия”». Эти функции не идентичны, поэтому мы обработаем их, представив, что у нас уже есть первая и нужно ввести вторую, то есть мы хотим внести вариативность.

1. Сначала применяем «Извлечение метода» для кода, который хотим изолировать.

Листинг 5.124. До

```
function removeLock1() {
  for (let y = 0; y < map.length; y++)
    for (let x = 0; x < map[y].length; x++)
      if (map[y][x].isLock1())
        map[y][x] = new Air();
}
```

Листинг 5.125. После (1/3)

```
function removeLock1() {
  for (let y = 0; y < map.length; y++)
    for (let x = 0; x < map[y].length; x++)
      if (check(map[y][x]))
        map[y][x] = new Air();
}
function check(tile: Tile) {
  return tile.isLock1();
}
```

Новые метод
и вызов

2. Создаем новый класс.

Листинг 5.126. Новый класс

```
class RemoveStrategy {
}
```

3. В этом случае у нас нет конструктора, в котором можно было бы инстанцировать этот новый класс. Вместо этого мы инстанцируем его прямо в функции.

Листинг 5.127. До

```
function removeLock1() {
  for (let y = 0; y < map.length; y++)
    for (let x = 0; x < map[y].length; x++)
      if (check(map[y][x]))
        map[y][x] = new Air();
}
```

4. Перемещаем метод.

Листинг 5.129. До

```
function removeLock1() {
  let shouldRemove = new RemoveStrategy();
  for (let y = 0; y < map.length; y++)
    for (let x = 0; x < map[y].length; x++)
      if (check(map[y][x]))
        map[y][x] = new Air();
}

function check(tile: Tile) {
  return tile.isLock1();
}
```

Листинг 5.128. После (2/3)

```
function removeLock1() {
  let shouldRemove = new RemoveStrategy();
  for (let y = 0; y < map.length; y++)
    for (let x = 0; x < map[y].length; x++)
      if (check(map[y][x]))
        map[y][x] = new Air();
}
```

Инициализация
нового класса

Листинг 5.130. После (3/3)

```
function removeLock1() {
  let shouldRemove = new RemoveStrategy();
  for (let y = 0; y < map.length; y++)
    for (let x = 0; x < map[y].length; x++)
      if (shouldRemove.check(map[y][x]))
        map[y][x] = new Air();
}

class RemoveStrategy {
  check(tile: Tile) {
    return tile.isLock1();
  }
}
```

Перемещенный
метод

5. В новом классе нет ни зависимостей от каких-либо полей, ни ошибок.

После «Введения паттерна «Стратегия»» можно использовать «Извлечение интерфейса из реализации» в качестве подготовки к внесению вариативности.

1. Создаем новый интерфейс с тем же именем, что и класс, из которого производим извлечение.

Листинг 5.131. До

```
interface RemoveStrategy {
}
```

2. Переименовываем класс, из которого будем извлекать интерфейс, и прописываем в нем реализацию нового интерфейса.

Листинг 5.132. До

```
class RemoveStrategy {
  // ...
}
```

Листинг 5.133. После (1/3)

```
class RemoveLock1 implements
RemoveStrategy
{
  // ...
}
```

3. Выполняем компиляцию и проходим по ошибкам:

1) если ошибка в new, прописываем новое имя класса;

Листинг 5.134. До

```
function removeLock1() {
  let shouldRemove = new RemoveStrategy();
  for (let y = 0; y < map.length; y++)
    for (let x = 0; x < map[y].length; x++)
      if (shouldRemove.check(map[y][x]))
        map[y][x] = new Air();
}
```

Листинг 5.135. После (2/3)

```
function removeLock1() {
  let shouldRemove = new RemoveLock1();
  for (let y = 0; y < map.length; y++)
    for (let x = 0; x < map[y].length; x++)
      if (shouldRemove.check(map[y][x]))
        map[y][x] = new Air();
}
```

←
Инстанцирование
класса вместо интерфейса

2) в противном случае добавляем вызывающий ошибку метод в интерфейс.

Листинг 5.136. До

```
interface RemoveStrategy {
}
```

Листинг 5.137. После (3/3)

```
interface RemoveStrategy {
  check(tile: Tile): boolean;
}
```

На этом этапе можно без проблем создать RemoveLock2 из копии RemoveLock1. Потребуется лишь вынести shouldRemove в качестве параметра. Я не стану грузить вас подробностями, но в целом мы делаем следующее.

1. Извлекая из removeLock1 все, кроме первой строки, мы получаем remove.
2. Локальная переменная shouldRemove используется всего раз, поэтому мы ее встраиваем.
3. Применяем «Встраивание метода» для removeLock1.

Этот рефакторинг приводит к тому, что у нас остается только одна функция remove.

Листинг 5.138. До

```
function removeLock1() {
  for (let y = 0; y < map.length; y++)
    for (let x = 0; x < map[y].length; x++)
      if (map[y][x].isLock1())
        map[y][x] = new Air();
}
class Key1 implements Tile {
  // ...
  moveHorizontal(dx: number) {
    removeLock1();
    moveToTile(playerx + dx, playery);
  }
}
```

Листинг 5.139. После

```
function remove(
  shouldRemove: RemoveStrategy) {
  for (let y = 0; y < map.length; y++)
    for (let x = 0; x < map[y].length; x++)
      if (shouldRemove.check(map[y][x]))
        map[y][x] = new Air();
}
class Key1 implements Tile {
  // ...
  moveHorizontal(dx: number) {
    remove(new RemoveLock1());
    moveToTile(playerx + dx, playery);
  }
}
interface RemoveStrategy {
  check(tile: Tile): boolean;
}
class RemoveLock1
  implements RemoveStrategy {
  check(tile: Tile) {
    return tile.isLock1();
  }
}
```

Как и ранее, в результате наших действий `remove` становится более обобщенной, но на сей раз это нас не ограничивает. К тому же теперь можно вносить изменения путем добавления: если нужно удалить любой тип клетки, мы можем просто создать еще один класс, реализующий `RemoveStrategy`, ничего при этом не меняя.

В некоторых приложениях предпочтительнее избегать вызова `new` внутри цикла, так как это может замедлять их выполнение. Если это будет актуально в данной ситуации, мы можем легко сохранить «Стратегию» `RemoveLock` в переменной экземпляра и инициализировать ее в конструкторе.

Казалось бы, финиш, но мы еще не закончили с `Key1`.

5.6. ОБЪЕДИНЕНИЕ СХОЖЕГО КОДА

У нас еще есть повторяющийся код в `Key1` и `Key2`, а также в `Lock1` и `Lock2`. В обоих случаях классы-двойники оказываются почти идентичными.

Листинг 5.140. `Key1` и `Lock1`

```
class Key1 implements Tile {
    // ...
    draw(g: CanvasRenderingContext2D,
        x: number, y: number)
    {
        g.fillStyle = "#ffcc00";
        g.fillRect(x * TILE_SIZE, y * TILE_SIZE,
            TILE_SIZE, TILE_SIZE);
    }
    moveHorizontal(dx: number) {
        remove(new RemoveLock1());
        moveToTile(playerx + dx, playery);
    }
}

class Lock1 implements Tile {
    // ...
    isLock1() { return true; }
    isLock2() { return false; }
    draw(g: CanvasRenderingContext2D,
        x: number, y: number)
    {
        g.fillStyle = "#ffcc00";
        g.fillRect(x * TILE_SIZE, y * TILE_SIZE,
            TILE_SIZE, TILE_SIZE);
    }
}
```

Листинг 5.141. `Key2` и `Lock2`

```
class Key2 implements Tile {
    // ...
    draw(g: CanvasRenderingContext2D,
        x: number, y: number)
    {
        g.fillStyle = "#00ccff";
        g.fillRect(x * TILE_SIZE, y * TILE_SIZE,
            TILE_SIZE, TILE_SIZE);
    }
    moveHorizontal(dx: number) {
        remove(new RemoveLock2());
        moveToTile(playerx + dx, playery);
    }
}

class Lock2 implements Tile {
    // ...
    isLock1() { return false; }
    isLock2() { return true; }
    draw(g: CanvasRenderingContext2D,
        x: number, y: number)
    {
        g.fillStyle = "#00ccff";
        g.fillRect(x * TILE_SIZE, y * TILE_SIZE,
            TILE_SIZE, TILE_SIZE);
    }
}
```

Сначала для обоих замков и для обоих ключей мы используем «Объединение схожих классов».

Листинг 5.142. До

```
class Key1 implements Tile {

    // ...
    draw(g: CanvasRenderingContext2D,
        x: number, y: number)
    {
        g.fillStyle = "#ffcc00";
        g.fillRect(x * TILE_SIZE, y * TILE_SIZE,
            TILE_SIZE, TILE_SIZE);
    }
    moveHorizontal(dx: number) {
        remove(new RemoveLock1());
        moveToTile(playerx + dx, playery);
    }
}

class Lock1 implements Tile {

    // ...

class Key1 implements Tile {

    // ...
    draw(g: CanvasRenderingContext2D,
        x: number, y: number)
    {
        g.fillStyle = "#ffcc00";
        g.fillRect(x * TILE_SIZE, y * TILE_SIZE,
            TILE_SIZE, TILE_SIZE);
    }
    moveHorizontal(dx: number) {
        remove(new RemoveLock1());
        moveToTile(playerx + dx, playery);
    }
}

class Lock1 implements Tile {
```

Листинг 5.143. После

```
class Key implements Tile {
    constructor(
        private color: string,
        private removeStrategy: RemoveStrategy)
    { }
    // ...
    draw(g: CanvasRenderingContext2D,
        x: number, y: number)
    {
        g.fillStyle = this.color;
        g.fillRect(x * TILE_SIZE, y * TILE_SIZE,
            TILE_SIZE, TILE_SIZE);
    }
    moveHorizontal(dx: number) {
        remove(this.removeStrategy);
        moveToTile(playerx + dx, playery);
    }
}

class Lock implements Tile {
    constructor(
        private color: string,
        private lock1: boolean,
        private lock2: boolean) { }
    // ...
    isLock1() { return this.lock1; }
    isLock2() { return this.lock2; }
}

class Key implements Tile {
    constructor(
        private color: string,
        private removeStrategy: RemoveStrategy)
    { }
    // ...
    draw(g: CanvasRenderingContext2D,
        x: number, y: number)
    {
        g.fillStyle = this.color;
        g.fillRect(x * TILE_SIZE, y * TILE_SIZE,
            TILE_SIZE, TILE_SIZE);
    }
    moveHorizontal(dx: number) {
        remove(this.removeStrategy);
        moveToTile(playerx + dx, playery);
    }
}

class Lock implements Tile {
```



```
// ...

draw(g: CanvasRenderingContext2D,
     x: number, y: number)
{
    g.fillStyle = "#ffcc00";
    g.fillRect(x * TILE_SIZE, y * TILE_SIZE,
               TILE_SIZE, TILE_SIZE);
}
}
function transformTile(tile: RawTile) {
    switch (tile) {
        // ...
        case RawTile.KEY1:
            return new Key1();

        case RawTile.LOCK1:
            return new Lock1();
    }
}
```

```
constructor(
    private color: string,
    private lock1: boolean,
    private lock2: boolean) { }
// ...
isLock1() { return this.lock1; }
isLock2() { return this.lock2; }
draw(g: CanvasRenderingContext2D,
     x: number, y: number)
{
    g.fillStyle = this.color;
    g.fillRect(x * TILE_SIZE, y * TILE_SIZE,
               TILE_SIZE, TILE_SIZE);
}
}
function transformTile(tile: RawTile) {
    switch (tile) {
        // ...
        case RawTile.KEY1:
            return new Key("#ffcc00",
                           new RemoveLock1());
        case RawTile.LOCK1:
            return new Lock("#ffcc00",
                            true, false);
    }
}
```

Этот код работает, но для его усовершенствования уместно воспользоваться преимуществом уже знакомой нам структуры. Нами введены методы `isLock1` и `isLock2`: они происходят из значений в перечислении, поэтому мы знаем, что только один из этих методов может вернуть `true` для любого из заданных классов. Следовательно, для представления обоих методов нужен всего один параметр. То же касается и методов `Lock`. Внесем соответствующие изменения.

Листинг 5.144. До

```
class Lock implements Tile {
    constructor(
        private color: string,
        private lock1: boolean,
        private lock2: boolean) { }
    // ...
    isLock1() { return this.lock1; }
    isLock2() { return this.lock2; }
}
```

Листинг 5.145. После

```
class Lock implements Tile {
    constructor(
        private color: string,
        private lock1: boolean
    ) { }
    // ...
    isLock1() { return this.lock1; }
    isLock2() { return !this.lock1; }
}
```

Похоже, что между параметрами `color`, `lock1` и `removeStrategy` наших конструкторов в `Key` и `Lock` наблюдается связь. С целью объединить компоненты

двух классов, используем наш новый полюбившийся прием: «Вводим паттерн “Стратегия”».

Листинг 5.146. До

```
class Key implements Tile {
  constructor(
    private color: string,
    private removeStrategy: RemoveStrategy)
  { }
  // ...
  draw(g: CanvasRenderingContext2D,
    x: number, y: number)
  {
    g.fillStyle = this.color;
    g.fillRect(x * TILE_SIZE, y * TILE_SIZE,
      TILE_SIZE, TILE_SIZE);
  }
  moveHorizontal(dx: number) {
    remove(this.removeStrategy);
    moveToTile(playerx + dx, playery);
  }

  moveVertical(dy: number) {
    remove(this.removeStrategy);
    moveToTile(playerx, playery + dy);
  }
}

class Lock implements Tile {
  constructor(
    private color: string,
    private lock1: boolean) { }
  // ...
  isLock1() { return this.lock1; }
  isLock2() { return !this.lock1; }
  draw(g: CanvasRenderingContext2D,
    x: number, y: number)
  {
    g.fillStyle = this.color;
    g.fillRect(x * TILE_SIZE, y * TILE_SIZE,
      TILE_SIZE, TILE_SIZE);
  }
}

function transformTile(tile: RawTile) {
  switch (tile) {
    // ...
    case RawTile.KEY1:
```

Листинг 5.147. После

```
class Key implements Tile {
  constructor(
    private keyConf: KeyConfiguration)
  { }
  // ...
  draw(g: CanvasRenderingContext2D,
    x: number, y: number)
  {
    g.fillStyle = this.keyConf.getColor();
    g.fillRect(x * TILE_SIZE, y * TILE_SIZE,
      TILE_SIZE, TILE_SIZE);
  }
  moveHorizontal(dx: number) {
    remove(this.keyConf
      .getRemoveStrategy());
    moveToTile(playerx + dx, playery);
  }
  moveVertical(dy: number) {
    remove(this.keyConf
      .getRemoveStrategy());
    moveToTile(playerx, playery + dy);
  }
}

class Lock implements Tile {
  constructor(
    private keyConf: KeyConfiguration) { }
  // ...
  isLock1() { return this.keyConf.is1(); }
  isLock2() { return !this.keyConf.is1(); }
  draw(g: CanvasRenderingContext2D,
    x: number, y: number)
  {
    g.fillStyle = this.keyConf.getColor();
    g.fillRect(x * TILE_SIZE, y * TILE_SIZE,
      TILE_SIZE, TILE_SIZE);
  }
}

class KeyConfiguration {
  constructor(
    private color: string,
    private _1: boolean,
```

```

        return new Key("#ffcc00",
            new RemoveLock1());
    case RawTile.LOCK1:
        return new Lock("#ffcc00", true);
    }
}

```

```

        private removeStrategy: RemoveStrategy)
    {
    }
    getColor() { return this.color; }
    is1() { return this._1; }
    getRemoveStrategy() {
        return this.removeStrategy;
    }
}
const YELLOW_KEY =
    new KeyConfiguration("#ffcc00", true,
        new RemoveLock1());
function transformTile(tile: RawTile) {
    switch (tile) {
        // ...
        case RawTile.KEY1:
            return new Key(YELLOW_KEY);
        case RawTile.LOCK1:
            return new Lock(YELLOW_KEY);
    }
}

```

Представьте, что на этом этапе нужно ввести третью и четвертую пару «ключ + замок». Для этого достаточно изменить `boolean` в `keyConfiguration` на `number` и методы `isLock` на один `fits(id: number)`. Очевидно, что теперь можно ввести столько пар «ключ + замок», сколько захочется. Конечно же, после этого обязательно видоизменим `number`, представляя его в виде перечисления, и используем «Замену кода типа классами». Ну а остальное вы уже знаете.

Опять же заметьте, что эта трансформация сделала явным момент, которого мы до сих пор не касались в явной форме: цвета и ID замков связаны. Это можно было ожидать ввиду интуитивной сути самого примера. Однако, даже если бы мы работали не над игровым примером, а, скажем, над сложной финансовой системой, то и там в обязательном порядке постепенно раскрывали бы подобные связи, заложенные в существующую структуру кода. Некоторые обнаруживаемые таким образом связи оказываются случайными, поэтому нужно быть внимательными и постоянно задаваться вопросом: имеет ли смысл подобное связывание элементов структуры. Такие связи также могут выявлять каверзные баги в коде, возникающие из-за связывания элементов, которые на самом деле связаны быть не должны.

Введенный нами класс `KeyConfiguration` на данный момент пуст и довольно невыразителен. Однако в следующей главе ситуация изменится, кроме того, и связи продолжат раскрываться и будут использованы с помощью инкапсулирования данных.

РЕЗЮМЕ

- Когда у нас есть похожие фрагменты кода, который должен сходиться, их следует объединить. Классы мы объединяем с помощью «Объединения схожих классов» (5.1.1), условия `if` — с помощью «Совмещения инструкций `if`» (5.2.1), а методы — с помощью «Введения паттерна “Стратегия”» (5.4.2).
- Правило «Использовать чистые условия» (5.3.2) утверждает, что условия не должны иметь побочных эффектов, поскольку для чистых условий мы можем использовать арифметику для работы с условиями. Было показано, как можно отделять побочные эффекты от условий с помощью `Cache`.
- UML-диаграммы классов обычно используются для иллюстрации архитектурных изменений в базе кода.
- Интерфейсы с одним реализующим классом представляют собой форму излишней обобщенности. Правило «Избегать интерфейсов всего с одной реализацией» (5.4.3) гласит, что у нас их быть не должно. Вместо них и гораздо позднее следует вводить интерфейс с помощью шаблона рефакторинга «Извлечение интерфейса из реализации» (5.4.4).

6

Защита данных

В этой главе

- ✓ Обеспечение инкапсуляции с помощью правила «Не использовать геттеры и сеттеры».
- ✓ Устранение геттеров с помощью шаблона «Удаление геттера или сеттера».
- ✓ Использование шаблона «Инкапсулирование данных» (6.2.3), чтобы всегда избегать общих аффиксов.
- ✓ Удаление инварианта с помощью шаблона «Обеспечение последовательности».

В главе 2 мы обсудили преимущество локализации инвариантов. Потом уже осуществляли это практически, когда вводили классы, потому что они притягивают функциональность, касающуюся одних и тех же данных, а значит, также притягивают инварианты и локализуют их. В текущей главе сосредоточимся на инкапсуляции — ограничении доступа к данным и функциональности, в результате инварианты смогут нарушаться только локально, что будет гораздо проще предотвратить.

6.1. ИНКАПСУЛЯЦИЯ С ПОМОЩЬЮ ГЕТТЕРОВ

На данном этапе наш код подчиняется правилам и уже намного лучше читается и расширяется. Однако можно добиться еще большего, введя очередное правило: «Не использовать геттеры или сеттеры».

6.1.1. Правило «Не использовать геттеры или сеттеры»

Утверждение

Не использовать сеттеры или геттеры для небулевых полей.

Объяснение

Когда мы говорим «*сеттеры*» или «*геттеры*», имеем в виду методы, которые напрямую присваивают (сеттеры) или получают (геттеры) небулево поле. Для C#-программистов мы в это определение также включаем свойства. Заметьте, что это никак не связано с именем метода — оно может быть как `getX`, так и любым другим.

Геттеры и сеттеры зачастую изучаются параллельно с инкапсуляцией в качестве стандартного способа обхода закрытых (`private`) полей. Однако если мы используем геттеры для полей объекта, то тут же нарушаем инкапсуляцию и делаем инвариант глобальным. После возвращения объекта получатель может распространить его по программе далее, на что мы не в силах повлиять. Любой фрагмент программы, который получает объект, может вызывать публичные методы этого объекта и изменить его неожиданным для нас образом.

Сеттеры представляют схожую проблему. Теоретически они вводят очередную абстракцию, в которой мы можем модифицировать внутреннюю структуру данных и изменить сеттер так, чтобы он по-прежнему имел ту же сигнатуру. Следуя нашему определению, такие методы больше не являются сеттерами, а значит, и проблемы как будто не представляют. Однако на деле мы изменяем геттер для возвращения новой структуры данных. В таком случае модуль-получатель также необходимо изменить, чтобы вместить эту новую структуру данных. Это и представляет форму сильной связанности, которой мы хотим избежать.

Данная проблема касается только изменяемых объектов. Тем не менее правило определяет в качестве исключений только булевы поля ввиду еще одного эффекта закрытых полей, который касается и неизменяемых полей: тесной связи их с предполагаемой ими же архитектурой. Одно из важнейших преимуществ преобразования полей в закрытые состоит в том, что это способствует формированию архитектуры на основе передачи (`push-based architecture`). В этой архитектуре вычисления перемещаются максимально близко к данным, в то время как в архитектуре на основе запросов (`pull-based architecture`) данные запрашиваются и затем производятся вычисления в центральной точке, откуда запрос данных был произведен.

Следуя архитектуре на основе запросов, мы получаем примитивные классы данных без каких-либо интересных методов и большие управляющие классы, которые выполняют всю работу и смешивают данные из многих мест. Этот подход формирует сильную связанность между данными и управляющим кодом, а также подспудно способствует возникновению связи между классами данных.

В архитектуре на основе передачи вместо получения данных они передаются в качестве аргументов. В результате все наши классы начинают обладать функциональностью, а код распространяется согласно ее свойствам.

Рассмотрим пример. В нем нужно сгенерировать ссылку на пост в блоге. Оба варианта предлагаемого кода выполняют одно и то же, но один написан в соответствии с архитектурой на основе запросов, а второй — на основе передачи. Структура вызовов первого варианта показана на рис. 6.1, а второго — на рис. 6.2.

Листинг 6.1. Архитектура на основе запросов

```
class Website {
  constructor (private url: string) { }
  getUrl() { return this.url; }
}
class User {
  constructor (private username:
    string) { }
  getUsername() { return this.username; }
}
class BlogPost {
  constructor (private author: User,
    private id: string) { }
  getId() { return this.id; }
  getAuthor() { return this.author; }
}
function generatePostLink(website: Website,
  post: BlogPost)
{
  let url = website.getUrl();
  let user = post.getAuthor();
  let name = user.getUsername();
  let postId = post.getId();
  return url + name + postId;
}
```

Листинг 6.2. Архитектура на основе передачи

```
class Website {
  constructor (private url: string) { }
  generateLink(name: string, id: string) {
    return this.url + name + id;
  }
}
class User {
  constructor (private username: string) { }
  generateLink(website: Website, id: string)
  {
    return website.generateLink(
      this.username,
      id);
  }
}
class BlogPost {
  constructor (private author: User,
    private id: string) { }
  generateLink(website: Website) {
    return this.author.generateLink(
      website,
      this.id);
  }
}
function generatePostLink(website: Website,
  post: BlogPost)
{
  return post.generateLink(website);
}
```

В примере на основе передачи, скорее всего, лучше встроить `generatorPostLink`, поскольку это всего одна строка без дополнительной информации.

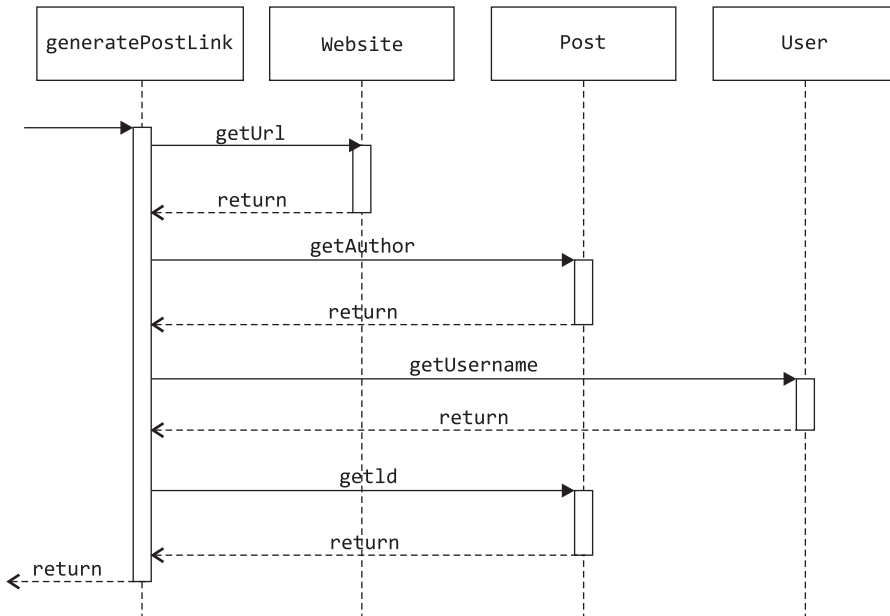


Рис. 6.1. Структура вызовов на основе запросов

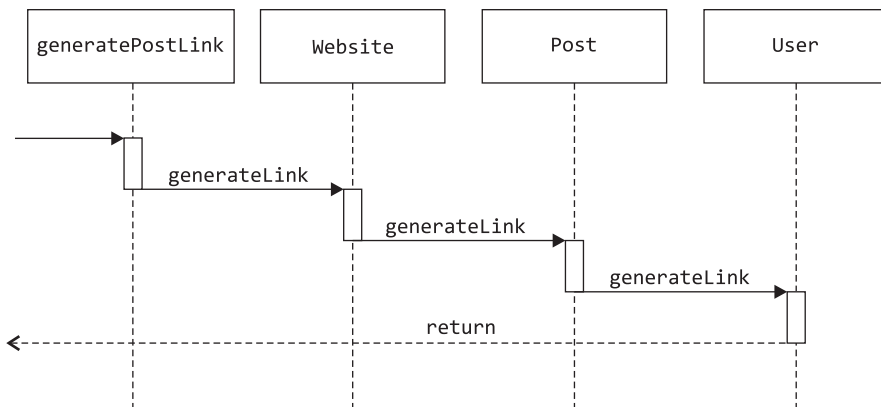


Рис. 6.2. Структура вызовов на основе передачи

Запах

Рассматриваемое правило основано на так называемом *законе Деметры*, который обычно формулируется как «Не разговаривай с незнакомцами». В данном контексте *незнакомец* — это объект, к которому мы не хотим иметь прямого доступа, но можем получить ссылку на него. В объектно-ориентированных языках это чаще всего выполняется с помощью геттеров — вот так и возникает формулировка этого правила.

Намерение

Проблема взаимодействия с объектами, на которые можно получить ссылку, заключается в том, что мы оказываемся сильно привязаны к способу получения объекта. Что касается структуры кода-владельца объекта, нам необходимо быть уверенными, что этот владелец поля не может изменить структуру данных, не сохранив поддержку способа получения нами старой структуры. В противном случае код будет нарушен.

В архитектуре на основе передачи методы используются подобно службам. Пользователи этих методов не должны беспокоиться о внутренней структуре данных или о том, как ссылки доставляются в вызывающий код.

Дополнительная информация

Подробное описание закона Деметры можно найти в Сети. В качестве же хорошего упражнения по его использованию я рекомендую задание по рефакторингу Fantasy Battle Сэмуела Иттербринка, оно доступно по адресу <https://github.com/Neppord/FantasyBattle-Refactoring-Kata>.

6.1.2. Применение правила

В коде у нас всего три геттера, два из которых находятся в `KeyConfiguration`: `getColor` и `getRemoveStrategy`. К счастью, разобраться с ними будет нетрудно. Начнем с `getRemoveStrategy`.

1. Делаем `getRemoveStrategy` закрытым (`private`), чтобы получить ошибки во всех местах его использования.

Листинг 6.3. До

```
class KeyConfiguration {
    // ...
    getRemoveStrategy() {
        return this.removeStrategy;
    }
}
```

Листинг 6.4. После (1/3)

```
class KeyConfiguration {
    // ...
    private getRemoveStrategy() { ←
        return this.removeStrategy;
    }
}
```

Метод
стал private

2. Чтобы исправить ошибки, используем «Перемещение кода в классы» (4.1.5) на соответствующих строках.

Листинг 6.5. До

```
class Key implements Tile {
    // ...
    moveHorizontal(dx: number) {

remove(this.keyConf.getRemoveStrategy());
    moveToFile(playerx + dx, playery);
    }
    moveVertical(dy: number) {

remove(this.keyConf.getRemoveStrategy());
    moveToFile(playerx, playery + dy);
    }
}
class KeyConfiguration {
    // ...
}
```

Листинг 6.6. После (2/3)

```
class Key implements Tile {
    // ...
    moveHorizontal(dx: number) {
        this.keyConf.removeLock(); ←
        moveToFile(playerx + dx, playery);
    }
    moveVertical(dy: number) {
        this.keyConf.removeLock(); ←
        moveToFile(playerx, playery + dy);
    }
}
class KeyConfiguration {
    // ...
    removeLock() { ←
        remove(this.removeStrategy);
    }
}
```

Строки,
выдававшие
ошибки

Новый метод

3. В процессе выполнения «Перемещения кода в классы» `getRemoveStrategy` встраивается. Следовательно, он оказывается больше не задействован и можно его удалить, чтобы исключить возможное использование другими фрагментами кода в дальнейшем.

Листинг 6.7. До

```
class KeyConfiguration {
    // ...
    private getRemoveStrategy() {
        return this.removeStrategy;
    }
}
```

Листинг 6.8. После (3/3)

```
class KeyConfiguration {
    // ...
    ← getRemoveStrategy удален
}
```

После повторения этого процесса для `getColor` получаем следующее.

Листинг 6.9. До

```
class KeyConfiguration {
    // ...
    getColor() {
        return this.color;
    }
    getRemoveStrategy() {
        return this.removeStrategy;
    }
}
class Key implements Tile {
    // ...
    draw(g: CanvasRenderingContext2D,
        x: number, y: number)
    {
        g.fillStyle = this.keyConf.getColor();
        g.fillRect(x * TILE_SIZE, y *
            TILE_SIZE, TILE_SIZE, TILE_SIZE);
    }
    moveHorizontal(dx: number) {
        remove(this.keyConf.getRemoveStrategy());
        moveToTile(playerx + dx, playery);
    }
    moveVertical(dy: number) {
        remove(this.keyConf.getRemoveStrategy());
        moveToTile(playerx, playery + dy);
    }
}
class Lock implements Tile {
    // ...
    draw(g: CanvasRenderingContext2D,
        x: number, y: number)
    {
        g.fillStyle = this.keyConf.getColor();
        g.fillRect(x * TILE_SIZE, y *
            TILE_SIZE, TILE_SIZE, TILE_SIZE);
    }
}
```

Листинг 6.10. После

```
class KeyConfiguration {
    // ...
    setColor(g: CanvasRenderingContext2D) { ←
        g.fillStyle = this.color;
    }
    removeLock() {
        remove(this.removeStrategy);
    }
}
class Key implements Tile {
    // ...
    draw(g: CanvasRenderingContext2D,
        x: number, y: number)
    {
        this.keyConf.setColor(g); ←
        g.fillRect(x * TILE_SIZE, y * TILE_SIZE,
            TILE_SIZE, TILE_SIZE);
    }
    moveHorizontal(dx: number) {
        this.keyConf.removeLock(); ←
        moveToTile(playerx + dx, playery);
    }
    moveVertical(dy: number) {
        this.keyConf.removeLock(); ←
        moveToTile(playerx, playery + dy);
    }
}
class Lock implements Tile {
    // ...
    draw(g: CanvasRenderingContext2D,
        x: number, y: number)
    {
        this.keyConf.setColor(g); ←
        g.fillRect(x * TILE_SIZE, y * TILE_SIZE,
            TILE_SIZE, TILE_SIZE);
    }
}
```

Метод, замещающий `getColor`

Метод, замещающий `getRemoveStrategy`

Заметьте, что `setColor` не является сеттером в описанном ранее смысле. Обратите также внимание, что в результирующем коде нарушается правило «Вызов или передача» (3.3.1), поскольку и передается `g`, и вызывается `g.fillRect`. Это можно

исправить либо путем перемещения `fillRect` в `KeyConfiguration` вместе с цветом, либо путем извлечения `fillRect` в метод. Если так будет сделано, то, скорее всего, `g` окажется инкапсулирован в более поздней точке и этот метод перемещен в кастомный графический объект вместо `CanvasRenderingContext2D`. Оставлю разбор этой ситуации в качестве упражнения для наиболее заинтересованных читателей.

Удаление геттера является очередным несложным, но очень важным процессом. О его важности говорит уже наличие двух имен, означающих, что мы должны избавиться от геттеров. Данный шаблон рефакторинга называется «Удаление геттера или сеттера».

6.1.3. Шаблон рефакторинга «Удаление геттера или сеттера»

Описание

Этот шаблон позволяет удалять геттеры и сеттеры, перемещая функциональность ближе к данным. К нашему удобству благодаря сходству геттеров и сеттеров можно избавиться от тех и других с помощью одного и того же процесса. Однако для простоты чтения до конца описания будем работать в предположении, что речь идет только о геттерах.

Мы уже много раз локализовывали инварианты, перемещая код ближе к данным. Здесь решение будет аналогичным. Обычно, когда мы так делаем, то вместо геттера вводим множество похожих функций. Их количество напрямую связано с количеством контекстов, в которых задействовался геттер. Наличие множества методов означает, что их можно именовать на основе конкретного контекста вызова, а не контекста данных.

Мы уже встречали это явление в главе 4. В примере `TrafficLight` у машины был публичный метод `drive`, который в итоге вызывала `TrafficLight`. Метод `drive` назван в соответствии с эффектом, который он оказывает на машину, но можно было назвать его на основе контекста вызова: `notifyGreenLight`. Для машины эффект будет тот же.

Листинг 6.11. До

```
class Green implements TrafficLight {
    // ...
    updateCar() { car.drive(); }
}
```

Листинг 6.12. После

```
class Green implements TrafficLight {
    // ...
    updateCar() { car.notifyGreenLight(); }
}
```

После переименования
метода на основе контекста

Процесс

1. Делаем геттер или сеттер закрытым, чтобы получить ошибки во всех местах его присутствия.
2. Исправляем ошибки перемещением кода в классы.
3. В процессе перемещения кода в классы геттер или сеттер встраивается. В результате он больше не задействуется, и мы его удаляем, чтобы исключить возможное использование другими фрагментами кода в дальнейшем.

Пример

Продолжая предыдущий пример, можно выбрать любой геттер для удаления.

Листинг 6.13. Начальный

```
class Website {
  constructor (private url: string) { }
  getUrl() { return this.url; }
}
class User {
  constructor (private username: string) { }
  getUsername() { return this.username; }
}
class BlogPost {
  constructor (private author: User, private id: string) { }
  getId() { return this.id; }
  getAuthor() { return this.author; }
}
function generatePostLink(website: Website, post: BlogPost) {
  let url = website.getUrl();
  let user = post.getAuthor();
  let name = user.getUsername();
  let postId = post.getId();
  return url + name + postId;
}
```

Здесь я продемонстрирую удаление `getAuthor`. Следуя описанному выше процессу, поступим так.

1. Сделаем геттер закрытым (`private`), вызывая ошибки во всех местах его использования.

Листинг 6.14. До

```
class BlogPost {
  // ...
  getAuthor() {
    return this.author;
  }
}
```

Листинг 6.15. После (1/3)

```
class BlogPost {
  // ...
  private getAuthor() { ← Добавили private
    return this.author;
  }
}
```

2. Исправим ошибки с помощью шаблона «Перемещение кода в классы».

Листинг 6.16. До

```
function generatePostLink(website:
Website,
    post: BlogPost)
{
    let url = website.getUrl();
    let user = post.getAuthor();
    let name = user.getUsername();
    let postId = post.getId();
    return url + name + postId;
}
class BlogPost {
    // ...
}
```

Листинг 6.17. После (2/3)

```
function generatePostLink(website:
Website,
    post: BlogPost)
{
    let url = website.getUrl();
    let name = post.getAuthorName();
    let postId = post.getId();
    return url + name + postId;
}
class BlogPost {
    // ...
    getAuthorName() { ← Новый
                        метод
        return this.author.getUsername();
    }
}
```

3. При перемещении кода в классы геттер встраивается. В результате он перестает использоваться и его можно безболезненно удалить, чтобы исключить возможное применение другими фрагментами кода в дальнейшем.

Листинг 6.18. До

```
class BlogPost {
    // ...
    private getAuthor() {
        return this.author;
    }
}
```

Листинг 6.19. После (3/3)

```
class BlogPost {
    // ...
    ← getAuthor удален
}
```

Следуя тому же процессу для других геттеров, получаем версию на основе перемещения, описанную в начале раздела 6.1.

6.1.4. Удаление последнего геттера

Последним геттером является `FallStrategy.getFalling`. Избавимся от него тем же путем.

1. Сделаем геттер закрытым (`private`), вызывая ошибки во всех местах его использования.

Листинг 6.20. До

```
class FallStrategy {
    // ...
    getFalling() {
        return this.falling;
    }
}
```

Листинг 6.21. После (1/3)

```
class FallStrategy {
    // ...
    private getFalling() { ←
        return this.falling;
    }
    Добавили private
}
```

2. Исправим ошибки с помощью перемещения кода в классы.

Листинг 6.22. До

```
class Stone implements Tile {
    // ...
    moveHorizontal(dx: number) {
        this.fallStrategy.getFalling()
        .moveHorizontal(this, dx);
    }
}
class Box implements Tile {
    // ...
    moveHorizontal(dx: number) {
        this.fallStrategy.getFalling()
        .moveHorizontal(this, dx);
    }
}
class FallStrategy {
    // ...
}
```

Листинг 6.23. После (2/3)

```
class Stone implements Tile {
    // ...
    moveHorizontal(dx: number) {
        this.fallStrategy
        .moveHorizontal(this, dx);
    }
}
class Box implements Tile {
    // ...
    moveHorizontal(dx: number) {
        this.fallStrategy
        .moveHorizontal(this, dx);
    }
}
class FallStrategy {
    // ...
    moveHorizontal(tile: Tile, dx: number) {
        this.falling
        .moveHorizontal(tile, dx);
    }
}
```

Новый метод

3. В процессе перемещения кода в классы геттер оказался встроен. В результате он перестает использоваться, и мы удалим его, чтобы исключить возможное применение другими фрагментами кода в дальнейшем.

Листинг 6.24. До

```
class FallStrategy {
    // ...
    private getFalling() {
        return this.falling;
    }
}
```

Листинг 6.25. После (3/3)

```
class FallStrategy {
    // ...
    ← getFalling удален
}
```

В итоге FallStrategy теперь выглядит так.

Листинг 6.26. До

```
class Stone implements Tile {
    // ...
    moveHorizontal(dx: number) {
        this.fallStrategy.getFalling()
        .moveHorizontal(this, dx);
    }
}
class Box implements Tile {
    // ...
```

Листинг 6.27. После

```
class Stone implements Tile {
    // ...
    moveHorizontal(dx: number) {
        this.fallStrategy
        .moveHorizontal(this, dx);
    }
}
class Box implements Tile {
    // ...
```

Новый перемещенный код

```

moveHorizontal(dx: number) {
    this.fallStrategy.getFalling()
    .moveHorizontal(this, dx);
}
}
class FallStrategy {
    constructor(private falling: FallingState)
    { }
    getFalling() { return this.falling; }
    update(tile: Tile, x: number, y: number) {
        this.falling = map[y + 1][x].isAir()
            ? new Falling()
            : new Resting();
        this.drop(tile, x, y);
    }
    private drop(tile: Tile,
        x: number, y: number)
    {
        if (this.falling.isFalling()) {
            map[y + 1][x] = tile;
            map[y][x] = new Air();
        }
    }
}

```

```

moveHorizontal(dx: number) {
    this.fallStrategy
        .moveHorizontal(this, dx); ← Новый
    }                                     перемещенный код
}
class FallStrategy {
    constructor(private falling: FallingState)
    { }
    ← getFalling удален
    update(tile: Tile, x: number, y: number) {
        this.falling = map[y + 1][x].isAir()
            ? new Falling()
            : new Resting();
        this.drop(tile, x, y);
    }
    private drop(tile: Tile,
        x: number, y: number)
    {
        if (this.falling.isFalling()) {
            map[y + 1][x] = tile;
            map[y][x] = new Air();
        }
    }
    moveHorizontal(tile: Tile, dx: number) { ← Новый
        this.falling.moveHorizontal(tile, dx);
    }
}

```

Если внимательно взглянуть на `FallStrategy`, то можно заметить, что здесь уместны еще кое-какие улучшения. Во-первых, тернарный оператор `? :` нарушает правило «Никогда не использовать `if` с `else`» (4.1.1). Во-вторых, похоже, что условие `if` в `drop` оказалось больше связано с `falling`. Если начать с тернарного оператора, то избавиться от него можно перемещением соответствующей строки в `Tile`.

Листинг 6.28. До

```

interface Tile {
    // ...
}
class Air implements Tile {
    // ...
}

class Stone implements Tile {
    // ...
}

```

Листинг 6.29. После

```

interface Tile {
    // ...
    getBlockOnTopState(): FallingState; ←
}
class Air implements Tile {
    // ...
    getBlockOnTopState() {
        return new Falling();
    }
}
class Stone implements Tile {
    // ...
    getBlockOnTopState() {
        return new Resting();
    }
}

```

← Перемещенный код


```

class FallStrategy {
    // ...
    update(tile: Tile, x: number, y:
        number) {
        this.falling = map[y + 1][x].isAir()
            ? new Falling()
            : new Resting();
        this.drop(tile, x, y);
    }
}

```

```

    }
}
class FallStrategy {
    // ...
    update(tile: Tile, x: number, y:
        number) {
        this.falling =
            map[y + 1][x].getBlockOnTopState(); ←
        this.drop(tile, x, y);    Перемещенный
    }                             код
}

```

В `FallStrategy.drop` можно полностью избавиться от `if`, переместив этот метод в `FallingState` и встроив `FallStrategy.drop`.

Листинг 6.30. До

```

interface FallingState {
    // ...
}

class Falling {
    // ...
}

class Resting {
    // ...
}

class FallStrategy {
    // ...
    update(tile: Tile, x: number, y:
        number) {
        this.falling =
            map[y + 1][x].getBlockOnTopState();
        this.drop(tile, x, y);
    }
    private drop(tile: Tile,
        x: number, y: number)
    {
        if (this.falling.isFalling()) {
            map[y + 1][x] = tile;
            map[y][x] = new Air();
        }
    }
}

```

Листинг 6.31. После

```

interface FallingState {
    // ...
    drop(
        tile: Tile, x: number, y: number): void; ←
}

class Falling {
    // ...
    drop(tile: Tile, x: number, y: number) { ←
        map[y + 1][x] = tile;
        map[y][x] = new Air();
    }
}                                     Перемещенный
class Resting {                               код
    // ...
    drop(tile: Tile, x: number, y: number) { } ←
}

class FallStrategy {
    // ...
    update(tile: Tile, x: number, y:
        number) {
        this.falling =
            map[y + 1][x].getBlockOnTopState();
        this.falling.drop(tile, x, y) ←
    }
    ← drop удален
}

```

6.2. ИНКАПСУЛИРОВАНИЕ ПРОСТЫХ ДАННЫХ

Снова мы находимся в ситуации, когда наш код подчиняется всем описанным ранее правилам. Значит, пора вводить новое.

6.2.1. Правило «Всегда избегать общих аффиксов»

Утверждение

В коде не должно быть методов или переменных с общими префиксами или суффиксами.

Объяснение

Мы нередко добавляем в начало или конец имен методов и переменных нечто уточняющее их контекст, например `username` для имени пользователя или `startTimer` для действия запуска таймера. Таким образом мы сообщаем контекст. Это, конечно, повышает читаемость кода, но при этом наличие у нескольких элементов одинакового аффикса указывает на их связанность. А для выражения подобной структуры есть более удачный способ — классы.

Преимущество использования классов состоит в группировке таких методов и переменных, которая позволяет получить полный контроль над внешним интерфейсом. Мы можем скрыть вспомогательные методы, чтобы они не загружали глобальную область видимости. Это особенно ценно, поскольку правило пяти строк приводит к возникновению множества методов.

Бывает и так, что не каждый метод можно безопасно вызвать отовсюду. Если мы извлечем среднюю часть сложного вычисления, то для ее работы может потребоваться дополнительная калибровка. В нашей игре этот случай относится к `updateMap` и `drawMap`, они обе требуют вызова `transformMap`.

Еще более важно, что за счет сокрытия данных мы гарантируем сохранение инвариантов в классах. Это позволяет делать их локальными и упрощает их поддержку.

Рассмотрим пример с банком из главы 4, где путем прямого вызова `deposit` можно было внести деньги на счет без их снятия с другого счета. Поскольку нам никогда не нужно вызывать `deposit` напрямую, более грамотной реализацией этой функциональности было бы помещение обоих методов в класс и закрытие `deposit` в категорию `private`.

Листинг 6.32. Плохой

```
function accountDeposit(
  to: string, amount: number)
{
  let accountId = database.find(to);
  database.updateOne(
    accountId,
    { $inc: { balance: amount } });
}
function accountTransfer(amount: number,
  from: string, to: string)
{
  accountDeposit(from, -amount);
  accountDeposit(to, amount);
}
```

Листинг 6.33. Удачный

```
class Account {
  private deposit(
    to: string, amount: number)
  {
    let accountId = database.find(to);
    database.updateOne(
      accountId,
      { $inc: { balance: amount } });
  }
  transfer(amount: number,
    from: string, to: string)
  {
    this.deposit(from, -amount);
    this.deposit(to, amount);
  }
}
```

Запах

Запах, из которого происходит это правило, называется «*Принцип единственной ответственности*». Он аналогичен запаху «Методы должны выполнять одно действие», о котором мы говорили ранее, но предназначен для классов. У классов должна быть одна ответственность.

Намерение

Проектирование классов с одной ответственностью требует дисциплины и бдительности. Это правило помогает определять подответственности. Структура, на которую указывает общий аффикс, предполагает, что эти методы и переменные разделяют ответственность, обозначенную этим аффиксом. Следовательно, эти методы должны находиться в отдельном классе, посвященном общей ответственности.

Это правило также помогает определять появление новых ответственностей, даже когда они возникают по ходу роста приложения. Классам свойственно со временем разрастаться.

Дополнительная информация

Принцип единственной ответственности очень подробно описан в интернете и является стандартным для построения классов. К сожалению, это означает, что он зачастую представлен как нечто, что должно быть жестко задано и заложено в проект заранее. Здесь же мы используем иной подход и сосредоточимся на признаке, который можно обнаружить в коде на любой стадии его создания или изменения.

6.2.2. Применение правила

У нас есть очевидная группа с одинаковым аффиксом, а именно метод и переменные:

- `playerx`;
- `playery`;
- `drawPlayer`.

Это предполагает, что следует поместить их в класс `Player`. Класс `Player` у нас уже есть, но назначение у него совсем иное. Можно предложить два выхода из этой, казалось бы, тупиковой ситуации. Первый — заключить все типы клеток в пространство имен и сделать их публичными. Несмотря на то что это предпочтительное решение, оно ведет к большому количеству манипуляций с TypeScript. А поскольку эта книга не о TypeScript, то воспользуемся другим выходом и просто переименуем существующий класс `Player`.

Листинг 6.34. До

```
class Player implements Tile { ... }
```

Листинг 6.35. После

```
class PlayerTile implements Tile { ... }
```

Добавляем `Tile` к имени

Теперь для упомянутой группы можно создать новый класс `Player`.

1. Создаем класс `Player`.

Листинг 6.36. Новый класс

```
class Player { }
```

2. Перемещаем переменные `playerx` и `playery` в `Player`, заменяя `let` на `private`. Удаляем `player` из их имен. Создаем также для переменных геттеры и сеттеры, с которыми разберемся позже.

Листинг 6.37. До

```
let playerx = 1;
let playery = 1;
```

Листинг 6.38. После (1/4)

```
class Player {
  private x = 1;
  private y = 1;
  getX() { return this.x; }
  getY() { return this.y; }
  setX(x: number) { this.x = x; }
  setY(y: number) { this.y = y; }
}
```

Новый класс

Удаляем `player`
из их имен

Новые
геттеры
и сеттеры

3. Поскольку `playerx` и `playery` больше не находятся в глобальной области видимости, компилятор выдает ошибки, помогая нам и указывая на все

связанные с ними ссылки. Исправлением ошибок мы займемся в следующих пяти шагах:

- 1) выбираем подходящее имя переменной для экземпляра класса `Player`: это будет `player`;
- 2) представляя, что у нас есть переменная `player`, используем ее геттеры и сеттеры;

Листинг 6.39. До

```
function moveToTile(
    newx: number, newy: number)
{
    map[playerx][playerx] =
        new Air();
    map[newx][newx] = new PlayerTile();
    playerx = newx;
    playery = newy;
}
/// ...
```

Листинг 6.40. После (2/4)

```
function moveToTile(
    newx: number, newy: number)
{
    map[player.getY()][player.getX()] =
        new Air();
    map[newx][newx] = new PlayerTile();
    player.setX(newx);
    player.setY(newy);
}
/// ...
```

Обращение изменено на геттеры

Присваивание изменено на сеттеров

Обращение и присваивание везде изменены

- 3) если ошибки замечены в двух или более разных методах, то добавляем `player: Player` в качестве первого параметра и `player` в качестве аргумента, вызывая новые ошибки;

Листинг 6.41. До

```
interface Tile {
    // ...
    moveHorizontal(
        dx: number): void;
    moveVertical(
        dy: number): void;
}
```

Листинг 6.42. После (3/4)

```
interface Tile {
    // ...
    moveHorizontal(
        player: Player, dx: number): void;
    moveVertical(
        player: Player, dy: number): void;
}
```

player добавлен в качестве параметра во многие методы, даже в те, которые находятся в интерфейсах

- 4) повторяем процесс, пока ошибка не останется только в одном методе;
- 5) поскольку мы инкапсулировали переменные, помещаем `let player = new Player();` в точку, где они находились.

Листинг 6.43. После (4/4)

```
let player = new Player();
```

Эта трансформация внесла изменения во всю базу кода. Ниже показаны некоторые из важнейших оказанных ею эффектов.

Листинг 6.44. До

```
interface Tile {
  // ...
  moveHorizontal(
    dx: number): void;
  moveVertical(
    dy: number): void;
}
/// ...
function moveToTile(
  newx: number, newy: number)
{
  map[playery][playerx] =
    new Air();
  map[newy][newx] = new PlayerTile();
  playerx = newx;
  playery = newy;
}
/// ...

let playerx = 1;
let playery = 1;
```

Листинг 6.45. После

```
interface Tile {
  // ...
  moveHorizontal(
    player: Player, dx: number): void; ←
  moveVertical(
    player: Player, dy: number): void; ←
}
/// ...
function moveToTile(
  newx: number, newy: number)
{
  map[player.getY()][player.getX()] = ←
    new Air();
  map[newy][newx] = new PlayerTile();
  player.setX(newx); ←
  player.setY(newy); ←
}
/// ...
class Player { ←
  private x = 1;
  private y = 1;
  getX() { return this.x; }
  getY() { return this.y; }
  setX(x: number) { this.x = x; }
  setY(y: number) { this.y = y; }
}
let player = new Player(); ←
```

player добавлен
в качестве параметра
во многие методы

Обращение
изменено
на геттеры

Присваивание
изменено на сеттеры

Новый класс
с геттерами
и сеттерами

Новое объявление
вместо
инкапсулированных
переменных

После введения класса можно без проблем переместить в него любой метод с аффиксом `Player`. В данном случае нужно переместить только `drawPlayer`.

Листинг 6.46. До

```
function drawPlayer(player: Player,
  g: CanvasRenderingContext2D)
{
  g.fillStyle = "#ff0000";
  g.fillRect(
    player.getX() * TILE_SIZE,
    player.getY() * TILE_SIZE,
    TILE_SIZE,
    TILE_SIZE);
}
class Player {
  // ...
}
```

Листинг 6.47. После

```
function drawPlayer(player: Player,
  g: CanvasRenderingContext2D)
{
  player.draw(g);
}
class Player {
  // ...
  draw(g: CanvasRenderingContext2D) {
    g.fillStyle = "#ff0000";
    g.fillRect(
      this.x * TILE_SIZE,
      this.y * TILE_SIZE,
      TILE_SIZE,
      TILE_SIZE);
  }
}
```

Заметьте, что мы
встроили геттеры

Как обычно, выполняем «Встраивание метода» (4.1.7) для `drawPlayer`. Созданный класс нарушает наше новое правило «Не использовать геттеры или сеттеры». Значит, надо применить к нему соответствующий рефакторинг, «Удаление геттера или сеттера». Начнем с `getX`.

1. Делаем этот геттер закрытым (`private`), вызывая ошибки во всех местах его использования.

Листинг 6.48. До

```
class Player {
  // ...
  getX() { return this.x; }
}
```

Листинг 6.49. После (1/3)

```
class Player {
  // ...
  private getX() { return this.x; } ← Делаем геттер закрытым
}
```

2. Исправляем ошибки с помощью перемещения кода в классы.

Листинг 6.50. До

```
class Right implements Input {
  handle(player: Player) {
    map[player.getY()][player.getX() + 1]
      .moveHorizontal(player, 1);
  }
}
class Resting {
  // ...
  moveHorizontal(
    player: Player, tile: Tile, dx: number)
  {
    if (map[player.getY()]
      [player.getX()+dx + dx].isAir()
      && !map[player.getY() + 1]
        [player.getX()+dx].isAir())
    {
      map[player.getY()]
        [player.getX()+dx + dx] = tile;
      moveToTile(player,
        player.getX()+dx,
        player.getY());
    }
  }
}
/// ...
moveToTile(player,
  player.getX(), player.getY() + dy);
/// ...
function moveToTile(player: Player,
  newx: number, newy: number)
{
  map[player.getY()][player.getX()] =
    new Air();
  map[newy][newx] = new PlayerTile();
  player.setX(newx);
  player.setY(newy);
}
```

Листинг 6.51. После (2/3)

```
class Right implements Input {
  handle(player: Player) {
    player.moveHorizontal(1);
  }
}
class Resting {
  // ...
  moveHorizontal(
    player: Player, tile: Tile, dx: number)
  {
    player.pushHorizontal(tile, dx); ← Методы, переданные в Player
  }
}
/// ...
player.move(0, dy); ←
/// ...
function moveToTile(player: Player,
  newx: number, newy: number)
{
  player.moveToTile(newx, newy);
}
/// ...
class Player {
  // ...
  moveHorizontal(dx: number) {
    map[this.y][this.x + dx]
      .moveHorizontal(this, dx);
  }
  move(dx: number, dy: number) {
    this.moveToTile(this.x+dx, this.y+dy);
  }
  pushHorizontal(tile: Tile, dx: number) {
    if (map[this.y]
      [this.x+dx + dx].isAir()
      && !map[this.y + 1]
        [this.x+dx].isAir()) {
```

```

}
/// ...
class Player {
  // ...
}

```

```

        map[this.y][this.x+dx + dx] = tile;
        this.moveToTile(this.x+dx, this.y);
      }
    }
    moveToTile(newx: number, newy: number) {
      map[this.y][this.x] = new Air();
      map[newy][newx] = new PlayerTile();
      this.x = newx;
      this.y = newy;
    }
  }
}

```

В процессе перемещения кода в классы геттер встраивается. Следовательно, он больше не задействован и его можно и нужно удалить, чтобы исключить дальнейшее использование другими фрагментами кода.

Листинг 6.52. До

```

class Player {
  // ...
  getX() { return this.x; }
}

```

Листинг 6.53. После (3/3)

```

class Player {
  // ...
  ← Удаляем getX
}

```

К счастью, `getX` и `getY` были настолько близко связаны, что `getY` просто исчез вместе с `getX`, а с ними, что удивительно, пропали и два сеттера. Теперь у нас получается следующее.

Листинг 6.54. До

```

class Player {
  // ...
  getX() { return this.x; }
  getY() { return this.y; }
  setX(x: number) { this.x = x; }
  setY(y: number) { this.y = y; }
}

```

Листинг 6.55. После

```

class Player {
  // ...
  ← Геттеры и сеттеры удалены

  moveHorizontal(dx: number) {
    map[this.y][this.x + dx]
      .moveHorizontal(this, dx);
  }
  move(dx: number, dy: number) {
    moveToTile(this.x + dx, this.y + dy);
  }
  pushHorizontal(tile: Tile, dx: number) {
    if (map[this.y][this.x + dx + dx].isAir()
      && !map[this.y + 1][this.x + dx].isAir())
    {
      map[this.y][this.x + dx + dx] = tile;
      moveToTile(this.x + dx, this.y);
    }
  }
  moveToTile(newx: number, newy: number) {
    map[this.y][this.x] = new Air();
    map[newy][newx] = new PlayerTile();
    this.x = newx;
    this.y = newy;
  }
}

```

← Новые методы, переданные в Player

Поскольку метод `moveToTile` был полностью перемещен в `Player`, применяем для его исходного варианта «Встраивание метода», тем самым удаляя его из глобальной области видимости. Теперь новый метод `Player.moveToTile` вызывается только внутри класса `Player`, значит, можно сделать его `private`, тем самым в значительной степени очистить растущий интерфейс для `Player`.

Этот процесс перемещения переменных и методов в класс называется инкапсулированием данных.

6.2.3. Паттерн рефакторинга «Инкапсуляция данных»

Описание

Как уже говорилось, инкапсулировать переменные и методы необходимо, чтобы ограничить к ним доступ и сделать структуру явной. Инкапсуляция методов помогает упростить их имена и делает связность более отчетливой. Это ведет к более стройным классам и нередко к их уменьшению, что также очень кстати. Судя по моему опыту, люди излишне сдержанно относятся к созданию классов.

Однако самый значительный выигрыш возникает за счет инкапсулирования переменных. Как говорилось в главе 2, мы зачастую предполагаем наличие у наших данных некоторых свойств. И чем больше точек возможного доступа к данным имеется, тем сложнее становится поддерживать эти свойства. Ограничение области означает, что изменять данные могут только методы внутри класса, а значит, и на свойства данных могут влиять только эти методы. В таком случае для проверки инварианта достаточно проверить код внутри класса.

Обратите внимание, что в некоторых ситуациях у нас могут быть только методы с общим аффиксом без переменных. При этом все равно есть смысл использовать данный шаблон рефакторинга, но тогда нужно переместить методы в класс до выполнения внутренних шагов.

Процесс

1. Создаем класс.
2. Перемещаем в него переменные, заменяя `let` на `private`. Упрощаем имена переменных, создаем для этих переменных геттеры и сеттеры.
3. Поскольку переменные больше не находятся в глобальной области видимости, компилятор выдает ошибки, помогая найти все ссылки на них. Исправляем эти ошибки следующими пятью шагами:
 - 1) подбираем подходящее имя переменной для экземпляра созданного класса;
 - 2) заменяем обращение к предполагаемой переменной геттерами или сеттерами;

- 3) если ошибки возникают в двух и более разных методах, то в качестве первого параметра добавляем параметр с ранее придуманным именем переменной (см. выше) и помещаем ту же переменную в качестве первого аргумента в точках вызова;
- 4) повторяем процесс, пока ошибка не останется только в одном методе;
- 5) если мы инкапсулировали переменные, инстанцируем новый класс в точке, где они были объявлены. В противном случае помещаем его экземпляр в метод, выдающий ошибку.

Пример

Приведу специально созданный для этого процесса пример, код ниже просто инкрементирует переменную 20 раз, на каждом шагу выводя ее значение. Но даже этих нескольких строк достаточно, чтобы показать возможные подводные камни при выполнении описанного рефакторинга.

Листинг 6.56. Начальный

```
let counter = 0;
function incrementCounter() {
  counter++;
}
function main() {
  for (let i = 0; i < 20; i++) {
    incrementCounter();
    console.log(counter);
  }
}
```

Следуя процессу, мы делаем так.

1. Создаем класс.

Листинг 6.57. Новый класс

```
class Counter { }
```

2. Перемещаем переменные в этот новый класс, заменяя `let` на `private`. Упрощаем имена переменных, а также создаем для этих переменных геттеры и сеттеры.

Листинг 6.58. До

```
let counter = 0;
class Counter {
}
```

Листинг 6.59. После (1/4)

```
class Counter {
  private counter = 0;
  getCounter() { return this.counter; }
  setCounter(c: number) {
    this.counter = c;
  }
}
```

Инкапсулированная переменная

Новый сеттер

Новый геттер

3. Поскольку `counter` больше не находится в глобальной области видимости, компилятор выдает ошибки, помогая нам найти все соответствующие ссылки. Эти ошибки исправляем, выполняя следующие пять шагов:

- 1) придумываем подходящее имя переменной для экземпляра нового класса: `counter`;
- 2) заменяем обращение к предполагаемой переменной геттерами или сеттерами;

Листинг 6.60. До

```
function incrementCounter() {
    counter++;
}
function main() {
    for (let i = 0; i < 20; i++) {
        incrementCounter();
        console.log(counter);
    }
}
```

Листинг 6.61. После (2/4)

```
function incrementCounter() {
    counter.setCounter(
        counter.getCounter() + 1); ← Присваивание, замененное
                                     сеттером
}
function main() {
    for (let i = 0; i < 20; i++) {
        incrementCounter();
        console.log(counter.getCounter()); ← Обращение,
                                             замененное геттером
    }
}
```

- 3) если ошибки обнаружены в двух и более разных методах, то добавляем в качестве первого параметра параметр с назначенным выше именем переменной и помещаем ту же переменную в качестве первого аргумента во всех точках вызова;

Листинг 6.62. До

```
function incrementCounter()
{
    counter.setCounter(
        counter.getCounter() + 1);
}
function main() {
    for (let i = 0; i < 20; i++) {
        incrementCounter();
        console.log(counter.getCounter());
    }
}
```

Листинг 6.63. После (3/4)

```
function incrementCounter(counter: Counter) ← Добавлен параметр
{
    counter.setCounter(
        counter.getCounter() + 1);
}
function main() {
    for (let i = 0; i < 20; i++) {
        incrementCounter(counter); ← Искусственная
        console.log(counter.getCounter()); переменная передана
    }                                     в качестве аргумента
}
```

- 4) повторяем процесс, пока ошибка не останется только в одном методе. В данном случае у нас всего одна ошибка;
- 5) теперь можно по неосторожности внести ошибку, инициализировав класс внутри цикла. Не всегда легко заметить, выполняется ли код внутри цикла. Обратите внимание, что приведенный ниже пример, хотя и пройдет компиляцию, корректно работать не будет.

Листинг 6.64. Ошибочный

```
function main() {
  for (let i = 0; i < 20; i++) {
    let counter = new Counter();
    incrementCounter(counter);
    console.log(counter.getCounter());
  }
}
```

← Неверное место
инстанцирования

Чтобы оградить себя от совершения подобной ошибки, необходимо определить, инкапсулированы ли переменные. В данном примере мы сами уже провели инкапсуляцию, значит, инстанцируем новый класс в точке, где была переменная.

Листинг 6.65. До

```
class Counter { ... }
```

Листинг 6.66. После (4/4)

```
class Counter { ... }
let counter = new Counter();
```

← Инстанцирование переменной
в точке, где была предыдущая
переменная

Теперь легко выполнить перемещение в `incrementCounter` с тем же суффиксом. Итоговый код в этом примере также нарушает одно из наших правил. Сможете сами определить, какое именно, и исправить? Подсказка: обратите внимание на то, как мы используем `counter` в листинге 6.63.

Дополнительные материалы

Этот шаблон рефакторинга очень тесно связан с шаблоном под названием «Инкапсуляция поля» из книги Мартина Фаулера «Рефакторинг. Улучшение существующего кода». «Инкапсуляция поля» делает публичное поле закрытым (`private`) и вводит для него геттер и сеттер. Разница между шаблонами в том, что наша версия заменяет публичный доступ к полю параметрами и после этого также инкапсулирует методы без поля.

Преобразование в параметры принесло пользу, теперь будет проще перемещать операцию инстанцирования, если это понадобится. Из-за параметров мы вынуждены инстанцировать класс до его использования, тем самым избегая возможной ошибки нулевой ссылки, которая могла возникнуть, когда к нему существовал глобальный доступ.

6.3. ИНКАПСУЛИРОВАНИЕ СЛОЖНЫХ ДАННЫХ

В нашей базе кода игры отчетливо выделяется еще одна группа методов и переменных:

- `map`;
- `transformMap`;

- updateMap;
- drawMap.

Эта группа прямо-таки просится в класс Map, значит, будет резонно использовать шаблон «Инкапсуляция данных».

1. Создаем класс Map.

Листинг 6.67. Новый класс

```
class Map { }
```

1. Перемещаем переменную map в Map и заменяем let на private. В данном случае имя упростить не получится. Вдобавок создадим для map геттер и сеттер.

Листинг 6.68. До

```
let map: Tile[][];
```

Листинг 6.69. После (1/4)

```
class Map {
  private map: Tile[][];
  getMap() { return this.map; }
  setMap(map: Tile[][]) { this.map = map; }
}
```

Добавляем геттер и сеттер для map

Перемещаем переменную, изменяя let на private

2. Поскольку Map больше не находится в глобальной области видимости, компилятор выдает ошибки, тем самым помогая нам найти все соответствующие им ссылки. Найденные ошибки исправим, выполнив следующие пять шагов:

- 1) подбираем удачное имя переменной для экземпляра класса Map: map;
- 2) заменяем обращение к предполагаемой переменной геттерами и сеттерами;

Листинг 6.70. До

```
function remove(
  shouldRemove: RemoveStrategy)
{
  for (let y = 0;
    y < map.length;
    y++)
    for (let x = 0;
      x < map[y].length;
      x++)
      if (shouldRemove.check(
        map[y][x]))
        map[y][x] = new Air();
}
```

Листинг 6.71. После (2/4)

```
function remove(
  shouldRemove: RemoveStrategy)
{
  for (let y = 0;
    y < map.getMap().length;
    y++)
    for (let x = 0;
      x < map.getMap()[y].length;
      x++)
      if (shouldRemove.check(
        map.getMap()[y][x]))
        map.getMap()[y][x] = new Air();
}
```

Обращение к map через getMap

- 3) если ошибки возникли в двух и более разных методах, то добавляем в качестве первого параметра параметр с именем переменной, придуманное ранее, и помещаем эту же переменную в качестве первого аргумента во всех точках вызова;

Листинг 6.72. До

```
interface Tile {
  // ...
  moveHorizontal(
    player: Player, dx: number): void;
  moveVertical(
    player: Player, dy: number): void;
  update(
    x: number, y: number): void;
}
/// ...
```

Листинг 6.73. После (3/4)

```
interface Tile {
  // ...
  moveHorizontal(map: Map,
    player: Player, dx: number): void;
  moveVertical(map: Map,
    player: Player, dy: number): void;
  update(map: Map,
    x: number, y: number): void;
}
/// ...
```

← тар добавлена в качестве аргумента

← тар добавлена во многих местах

- 4) повторяем процесс, пока ошибка не останется лишь в одном методе;
- 5) переменную мы инкапсулировали, значит, можно поместить `let map = new Map();` в точку, где находилась `map`.

Листинг 6.74. После (4/4)

```
let map = new Map();
```

Результатом будет следующая трансформация.

Листинг 6.75. До

```
interface Tile {
  // ...
  moveHorizontal(player: Player, dx: number): void;
  moveVertical(player: Player, dy: number): void;
  update(x: number, y: number): void;
}
/// ...
function remove(shouldRemove: RemoveStrategy)
{
  for (let y = 0; y < map.length; y++)
    for (let x = 0; x < map[y].length; x++)
      if(shouldRemove.check(map[y][x])) map[y][x] = new Air();
}
/// ...
let map: Tile[][];
```

Листинг 6.76. После

```
interface Tile {
  // ...
  moveHorizontal(map: Map, player: Player, dx: number): void;
  moveVertical(map: Map, player: Player, dy: number): void;
  update(map: Map, x: number, y: number): void;
}
/// ...
function remove(map: Map, shouldRemove: RemoveStrategy)
{
```

← тар добавлена в качестве аргумента

```

    for (let y = 0; y < map.getMap().length; y++)
      for (let x = 0; x < map.getMap()[y].length; x++)
        if (shouldRemove.check(map.getMap()[y][x]))
          map.getMap()[y][x] = new Air();
  }
  /// ...
class Map {
  private map: Tile[][];
  getMap() { return this.map; }
  setMap(map: Tile[][]) { this.map = map; }
}

```

Обращаемся к map
через getMap

Новый класс с геттером
и сеттером для map

Теперь обработать упомянутые выше методы будет просто: «Перемещение кода в классы» по ходу дела упрощает их имена и становится возможно использовать «Встраивание метода», как уже много раз приходилось делать.

Листинг 6.77. До

```

function transformMap(map: Map) {
  map.setMap(new Array(rawMap.length));
  for (let y = 0; y < rawMap.length; y++) {
    map.getMap()[y] = new Array(rawMap[y].length);
    for (let x = 0; x < rawMap[y].length; x++)
      map.getMap()[y][x] = transformTile(rawMap[y][x]);
  }
}
function updateMap(map: Map) {
  for (let y = map.getMap().length - 1; y >= 0; y--)
    for (let x = 0; x < map.getMap()[y].length; x++)
      map.getMap()[y][x].update(map, x, y);
}
function drawMap(map: Map, g: CanvasRenderingContext2D) {
  for (let y = 0; y < map.getMap().length; y++)
    for (let x = 0; x < map.getMap()[y].length; x++)
      map.getMap()[y][x].draw(g, x, y);
}

```

Листинг 6.78. После

```

class Map {
  // ...
  transform() {
    this.map = new Array(rawMap.length);
    for (let y = 0; y < rawMap.length; y++) {
      this.map[y] = new Array(rawMap[y].length);
      for (let x = 0; x < rawMap[y].length; x++)
        this.map[y][x] = transformTile(rawMap[y][x]);
    }
  }
  update() {
    for (let y = this.map.length - 1; y >= 0; y--)
      for (let x = 0; x < this.map[y].length; x++)
        this.map[y][x].update(this, x, y);
  }
}

```

```

    }
    draw(g: CanvasRenderingContext2D) {
        for (let y = 0; y < this.map.length; y++)
            for (let x = 0; x < this.map[y].length; x++)
                this.map[y][x].draw(g, x, y);
    }
}

```

Как и в случае с `Player`, мы ввели геттер и сеттер, значит, снова нужно «Удалить геттер или сеттер». Удачно, что сеттер не используется и удалить его будет легко. А вот геттер требует перемещения кода, поэтому я разделил его представление в листингах «до» и «после» на несколько частей.

Листинг 6.79. До

```

class Falling {
    // ...
    drop(map: Map, tile: Tile,
        x: number, y: number)
    {
        map.getMap()[y + 1][x] = tile;
        map.getMap()[y][x] = new Air();
    }
}
class Map {
    // ...
}

```

Листинг 6.81. До

```

class FallStrategy {
    // ...
    update(map: Map, tile: Tile,
        x: number, y: number)
    {
        this.falling =
            map.getMap()[y + 1][x].isAir()
            ? new Falling()
            : new Resting();
        this.falling.drop(map, tile, x, y);
    }
}
class Map {
    // ...
}

```

Листинг 6.80. После

```

class Falling {
    // ...
    drop(map: Map, tile: Tile,
        x: number, y: number)
    {
        map.drop(tile, x, y);
    }
}
class Map {
    // ...
    drop(tile: Tile, x: number, y: number) {
        this.map[y + 1][x] = tile;
        this.map[y][x] = new Air();
    }
}

```

Код перемещен
в Map

Листинг 6.82. После

```

class FallStrategy {
    // ...
    update(map: Map, tile: Tile,
        x: number, y: number)
    {
        this.falling =
            map.getBlockOnTopState(x, y + 1);
        this.falling.drop(map, tile, x, y);
    }
}
class Map {
    // ...
    getBlockOnTopState(x: number, y: number) {
        return this.map[y][x]
            .getBlockOnTopState();
    }
}

```

Код перемещен
в Map

Листинг 6.83. До

```

class Player {
    // ...
    moveHorizontal(map: Map, dx:
        number) {
        map.getMap()[this.y][this.x + dx]
            .moveHorizontal(map, this, dx);
    }
    moveVertical(map: Map, dy: number) {
        map.getMap()[this.y + dy][this.x]
            .moveVertical(map, this, dy);
    }
    pushHorizontal(map: Map, tile: Tile,
        dx: number)
    {
        if (map.getMap()
            [this.y][this.x + dx + dx].isAir()
            && !map.getMap()
            [this.y + 1][this.x + dx].isAir())
        {
            map.getMap()[this.y][this.x +
                dx + dx] = tile;
            this.moveToTile(
                map, this.x + dx, this.y);
        }
    }
    private moveToTile(map: Map,
        newX: number, newY: number)
    {
        map.getMap()[this.y][this.x] =
            new Air();
        map.getMap()[newY][newX] =
            new PlayerTile();
        this.x = newX;
        this.y = newY;
    }
}
class Map {
    // ...
}

```

Листинг 6.84. После

```

class Player {
    // ...
    moveHorizontal(map: Map, dx: number) {
        map.moveHorizontal(this,
            this.x, this.y, dx);
    }
    moveVertical(map: Map, dy: number) {
        map.moveVertical(this,
            this.x, this.y, dy);
    }
    pushHorizontal(map: Map, tile: Tile,
        dx: number)
    {
        if (map.isAir(this.x + dx + dx, this.y)
            && !map.isAir(this.x + dx, this.y + 1))
        {
            map.setTile(this.x + dx + dx, this.y,
                tile);
            this.moveToTile(
                map, this.x + dx, this.y);
        }
    }
    private moveToTile(map: Map,
        newX: number, newY: number)
    {
        map.movePlayer(this.x, this.y,
            newX, newY);
        this.x = newX;
        this.y = newY;
    }
}
class Map {
    // ...
    isAir(x: number, y: number) {
        return this.map[y][x].isAir();
    }
    setTile(x: number, y: number, tile: Tile)
    {
        this.map[y][x] = tile;
    }
    movePlayer(x: number, y: number,
        newX: number, newY: number)
    {
        this.map[y][x] = new Air();
        this.map[newY][newX] = new
PlayerTile();
    }
    moveHorizontal(player: Player,
        x: number, y: number, dx: number)
    {
        this.map[y][x + dx]
            .moveHorizontal(this, player, dx);
    }
}

```

Код
перемещен
в Map

```

moveVertical(player: Player,
  x: number, y: number, dy: number)
{
  this.map[y + dy][x].moveVertical(
    this, player, dy);
}
}

```

Листинг 6.85. До

```

function remove(map: Map,
  shouldRemove: RemoveStrategy)
{
  for (let y = 0;
    y < map.getMap().length;
    y++)
    for (let x = 0;
      x < map.getMap()[y].length;
      x++)
      if (shouldRemove.check(
        map.getMap()[y][x]))
        map.getMap()[y][x] = new Air();
}
class Map {
  // ...
  getMap() {
    return this.map;
  }
}

```

Листинг 6.86. После

```

class Map {
  // ...
  ← getMap удален
  remove(shouldRemove: RemoveStrategy) { ←
    for (let y = 0;
      y < this.map.length;
      y++)
      for (let x = 0;
        x < this.map[y].length;
        x++)
        if (shouldRemove.check(
          this.map[y][x]))
          this.map[y][x] = new Air();
    }
  }
}

```

Код
перемещен
в Map

Исходный `remove` теперь составляет одну строку, значит, можно и нужно использовать «Встраивание метода».

Обычно мы не склонны вводить в наш публичный интерфейс сильные методы вроде `setTile`. Это дает почти полный контроль закрытому (`private`) полю `map`. Однако сейчас добавление кода нас не пугает, и мы действуем уверенно.

Далее замечаем, что все строки, кроме одной в `Player.pushHorizontal`, используют `Map`, следовательно, нужно переместить этот код в `Map`.

Листинг 6.87. До

```

class Player {
  // ...
  pushHorizontal(map: Map, tile: Tile,
    dx: number)
  {
    if (map.isAir(this.x + dx
      + dx, this.y)
      && !map.isAir(this.x + dx,
        this.y + 1))
    {
      map.setTile(this.x + dx + dx,
        this.y,

```

Листинг 6.88. После

```

class Player {
  // ...
  pushHorizontal(map: Map, tile: Tile,
    dx: number)
  {
    map.pushHorizontal(
      this, tile, this.x, this.y, dx); ←
  }
  moveToTile(map: Map, ←
    newX: number, newY: number) ←
  {
    map.movePlayer(this.x, this.y,

```

Код перемещен
в Map

Метод стал
публичным

```

        tile);
        this.moveToTile(
            map, this.x + dx, this.y);
    }
}
private moveToTile(map: Map,
    newx: number, newy: number)
{
    map.movePlayer(this.x, this.y,
        newx, newy);
    this.x = newx;
    this.y = newy;
}
}
}

```

```

        newx, newy);
        this.x = newx;
        this.y = newy;
    }
}
class Map {
    // ...
    pushHorizontal(player: Player, tile: Tile,
        x: number, y: number, dx: number)
    {
        if (this.map[y][x + dx + dx].isAir()
            && !this.map[y + 1][x + dx].isAir())
        {
            this.map[y][x + dx + dx] = tile;
            player.moveToTile(this, x + dx, y);
        }
    }
}
}

```

Этот `setTile` используется только внутри `Map`. Можно сделать его закрытым (`private`) или, еще лучше, удалить, ведь мы любим удалять код.

6.4. УСТРАНЕНИЕ ИНВАРИАНТА ПОСЛЕДОВАТЕЛЬНОСТИ

Теперь видно, что карта инициализируется вызовом `map.transform`. Но в объектно-ориентированном сеттинге применяется другой механизм инициализации: конструктор. В данном случае нам повезло, поскольку можно заменить `transform` на `constructor`, а вызов `transform` удалить.

Листинг 6.89. До

```

class Map {
    // ...
    transform() {
        // ...
    }
}
/// ...
window.onload = () => {
    map.transform();
    gameLoop(map);
}

```

Листинг 6.90. После

```

class Map {
    // ...
    constructor() {
        // ...
    }
}
/// ...
window.onload = () => {
    gameLoop(map);
}

```

transform стал constructor

Вызов transform удален

Благодаря этому действию мы получили существенный эффект, устранив инвариант необходимости вызова `map.transform` до обращения к другим методам.

Такую ситуацию, в которой для вызова одного фрагмента кода необходимо вызвать другой, мы называем *инвариантом последовательности*. Теперь уже стало невозможно вызвать конструктор в начале, поэтому инвариант устранен. Эту технику также можно использовать для обеспечения выполнения действий в определенном порядке. Данный шаблон рефакторинга называется «Обеспечение последовательности».

6.4.1. Шаблон рефакторинга «Обеспечение последовательности»

Описание

Я считаю, что самый крутой тип рефакторинга, это когда можно «обучить» компилятор обеспечивать нужный нам порядок выполнения программы. И сейчас мы имеем именно такую ситуацию.

Отличительное свойство объектно-ориентированных языков состоит в том, что конструкторы всегда вызываются до методов в объектах. Есть возможность воспользоваться этим свойством, чтобы обеспечить нужный порядок действий. При этом сделать это совсем не сложно, хотя и потребует вводить по одному классу на каждом шаге, выполнение которого мы хотим обеспечить. Зато после внесения всех этих изменений последовательность перестанет быть инвариантом, потому что она станет обусловленной. Нам не нужно будет помнить о необходимости вызова одного метода перед другим, потому что иначе сделать уже не получится: порядок вызова окажется четко заданным. И ведь это то, что нам надо!

При использовании конструктора для обеспечения выполнения нужного кода гарантом его выполнения становится экземпляр класса. Ведь нельзя получить экземпляр, не создав предварительно конструктор.

Этот пример показывает, как с помощью данной техники обеспечить перевод строки в верхний регистр до ее вывода на печать.

Листинг 6.91. До

```
function print(str: string) {
    // string should be capitalized
    console.log(str);
}
```

Листинг 6.92. После

```
class CapitalizedString {
    private value: string;
    constructor(str: string) {
        this.value = capitalize(str);
    }
    print() {
        console.log(this.value);
    }
}
```

← Инвариант исчез

Преобразование «Обеспечение последовательности» может быть двух типов: внутренним и внешним. В предыдущем примере показана внутренняя его версия: целевая функция перемещается в новый класс. Ниже приводится сравнение упомянутых версий, которые по большей части предлагают одинаковые усовершенствования кода.

Листинг 6.93. Внутренняя версия

```
class CapitalizedString {
  private value: string;
  constructor(str: string) {
    this.value = capitalize(str);
  }

  print() {
    console.log(this.value);
  }
}
```

private
против
public

Метод против
функции
с конкретным
типом
параметра

Листинг 6.94. Внешняя версия

```
class CapitalizedString {
  public readonly value: string;
  constructor(str: string) {
    this.value = capitalize(str);
  }
}

function print(str: CapitalizedString) {
  console.log(str.value);
}
```

Этот шаблон рефакторинга фокусируется на внутренней версии, потому что она ведет к более уверенной инкапсуляции без использования геттера или публичного поля.

Процесс

1. Используем шаблон «Инкапсуляция данных» для метода, который должен выполняться последним.
2. Вызываем в конструкторе первый метод.
3. Если аргументы двух методов связаны, делаем их полями и удаляем из метода.

Пример

Рассмотрим пример, похожий на уже знакомый нам случай с банковскими операциями. Мы хотим обеспечить, чтобы деньги всегда сначала снимались у отправителя и только после этого уже зачислялись получателю. Таким образом, сначала в последовательности будет идти `deposit` с отрицательной денежной суммой (`amount`), а затем `deposit` с положительной денежной суммой (`amount`).

Листинг 6.95. Начальный

```
function deposit(
  to: string, amount: number)
{
  let accountId = database.find(to);
  database.updateOne(
    accountId,
    { $inc: { balance: amount } });
}
```

1. Используем шаблон «Инкапсуляция данных» для метода, который должен выполняться последним.

Листинг 6.96. До

```
function deposit(
  to: string, amount: number)
{
  let accountId = database.find(to);
  database.updateOne(
    accountId,
    { $inc: { balance: amount } });
}
```

Листинг 6.97. После (1/2)

```
class Transfer { ← Новый класс
  deposit(
    to: string, amount: number)
  {
    let accountId = database.find(to);
    database.updateOne(
      accountId,
      { $inc: { balance: amount } });
  }
}
```

2. Вызываем в конструкторе первый метод.

Листинг 6.98. До

```
class Transfer {

  deposit(to: string, amount: number) {
    let accountId = database.find(to);
    database.updateOne(
      accountId,
      { $inc: { balance: amount } });
  }
}
```

Листинг 6.99. После (2/2)

```
class Transfer {
  constructor(
    from: string, amount: number) {
    {
      this.deposit(from, -amount);
    }
  }
  deposit(to: string, amount: number) {
    let accountId = database.find(to);
    database.updateOne(
      accountId,
      { $inc: { balance: amount } });
  }
}
```

Новый конструктор, вызывающий первый метод

Теперь мы обеспечили вызов `deposit` с отрицательным `amount` от отправителя, но можно пойти еще дальше. Мы можем связать два `amount`, сделав аргумент полем и удалив из него `amount`. Поскольку нужно, чтобы в одном случае количество

было отрицательным, резонно ввести вспомогательный метод, и в результате получаем следующее.

Листинг 6.100. После

```
class Transfer {
  constructor(from: string, private amount: number) {
    this.depositHelper(from, -this.amount);
  }
  private depositHelper(to: string, amount: number) {
    let accountId = database.find(to);
    database.updateOne(accountId, { $inc: { balance: amount } });
  }
  deposit(to: string) {
    this.depositHelper(to, this.amount);
  }
}
```

Мы гарантировали невозможность создания денег из ниоткуда при отчислении их, но при этом деньги могут исчезнуть, если забыть вызвать `deposit` с получателем. Следовательно, нам нужно обернуть (`wrap`) этот класс в другой класс, чтобы гарантировать также и зачисление средств.

Дополнительные материалы

Мне неизвестны какие-либо формальные описания подобных паттернов. Без сомнения, есть люди, знакомые с этим способом использования объектов для обеспечения нужного порядка выполнения, но я подобных описаний и обсуждений не встречал.

6.5. УСТРАНЕНИЕ ПЕРЕЧИСЛЕНИЙ ИНЫМ СПОСОБОМ

Последним явно обозначенным методом является `transformTile`, и выделяется он суффиксом `tile`. У нас уже есть класс (или, говоря точнее, перечисление) с таким же суффиксом: `RawTile`. Имя `transformTile` предполагает, что данный метод нужно переместить в перечисление `RawTile`. Однако это невозможно реализовать во многих языках, включая TypeScript, так как в них перечисления не могут содержать методов.

6.5.1. Перечисление с помощью закрытых конструкторов

Если наш язык не поддерживает методы в перечислениях, есть техника, которая позволит обойти это ограничение с помощью закрытого (`private`) конструктора. Каждый объект должен создаваться через вызов конструктора. Если сделать конструктор `private`, объекты можно будет создавать только внутри класса.

В частности, тогда мы сможем управлять количеством существующих экземпляров. Если поместить эти экземпляры в публичные константы, то можно использовать их в качестве перечислений.

Листинг 6.101. Перечисление

```
enum TShirtSize {
    SMALL,
    MEDIUM,
    LARGE,
}

function sizeToString(s: TShirtSize) {
    if (s === TShirtSize.SMALL)
        return "S";
    else if (s === TShirtSize.MEDIUM)
        return "M";
    else if (s === TShirtSize.LARGE)
        return "L";
}
```

Листинг 6.102. Закрытый (private) конструктор

```
class TShirtSize {
    static readonly SMALL = new
    TShirtSize();
    static readonly MEDIUM = new
    TShirtSize();
    static readonly LARGE = new
    TShirtSize();
    private constructor() { }
}

function sizeToString(s: TShirtSize) {
    if (s === TShirtSize.SMALL)
        return "S";
    else if (s === TShirtSize.MEDIUM)
        return "M";
    else if (s === TShirtSize.LARGE)
        return "L";
}
```

Единственное исключение состоит в том, что при такой конструкции мы не можем использовать `switch`, но потеря невелика, ведь мы как раз и стремимся выполнять правило, которое запрещает нам так делать. Заметьте, что при сериализации/десериализации (работе с последовательностями) данных возникает несколько странное поведение кода, но этот вопрос уже выходит за рамки книги.

Теперь `TShirtSize` является классом (что очень здорово), и мы можем переместить в него код. К сожалению, в данной конфигурации нельзя исключить `if`, потому что, в отличие от прошлого раза, у нас нет класса для каждого значения: есть всего один класс. Для получения полноценных преимуществ выполнения этих шаблонов необходимо исправить ситуацию, а именно заменить код типа классами (4.1.3).

Листинг 6.103. Классы заменяют значения кода типов

```
interface SizeValue { }
class SmallValue implements SizeValue { }
class MediumValue implements SizeValue { }
class LargeValue implements SizeValue { }
```

Опять же можно упростить эти имена с помощью пространств имен или пакетов. На этот раз допустимо пропустить методы `is`, потому что мы никогда не создаем новые экземпляры на лету, следовательно, использования `===` будет достаточно. Затем будем использовать эти новые классы в качестве аргумента для каждого

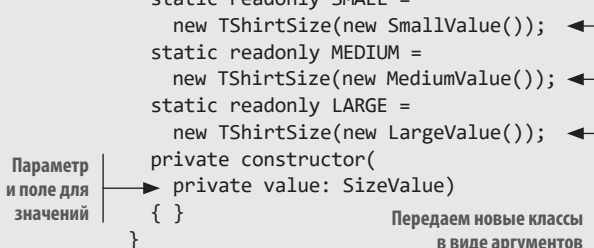
значения в классе закрытого (`private`) конструктора. При этом сохраняется данный аргумент в виде поля.

Листинг 6.104. До

```
class TShirtSize {
    static readonly SMALL =
        new TShirtSize();
    static readonly MEDIUM
        = new TShirtSize();
    static readonly LARGE
        = new TShirtSize();
    private constructor() { }
```

Листинг 6.105. После

```
class TShirtSize {
    static readonly SMALL =
        new TShirtSize(new SmallValue());
    static readonly MEDIUM =
        new TShirtSize(new MediumValue());
    static readonly LARGE =
        new TShirtSize(new LargeValue());
    private constructor(
        private value: SizeValue)
    { }
```



Параметр
и поле для
значений

Передаем новые классы
в виде аргументов

Теперь при каждом перемещении чего-либо в `TShirtSize` можно переместить это дальше во все классы и разрешить `=== TShirtSize.`, тем самым избавившись от `if`. Этот прием можно было бы назвать отдельным шаблоном, но я решил не включать его в их список по двум причинам. Во-первых, этот процесс неприменим равнозначно ко всем языкам программирования, в частности, к числу исключений относится Java. Во-вторых, у нас уже есть шаблон для устранения перечислений, которому надлежит отдавать предпочтение.

В нашей игре остается одно перечисление: `RawTile`. Мы уже выполнили для него замену кода типа классами, но полностью устранить это перечисление не смогли, потому что используем его индексы в некоторых местах кода программы. Однако с помощью предыдущей трансформации его все-таки можно с успехом удалить.

Введем новый класс `RawTile2` с закрытым (`private`) конструктором, содержащим поле для каждого значения перечисления. Теперь создадим новый интерфейс `RawTileValue` и классы для каждого значения перечисления, которые в качестве аргументов передадим в поля `RawTile2`.

Листинг 6.106. Новый класс

```
interface RawTileValue { }
class AirValue implements RawTileValue { }
// ...
class RawTile2 {
    static readonly AIR = new RawTile2(new AirValue());
    // ...
    private constructor(private value: RawTileValue) { }
}
```

Так мы приблизились на шаг к удалению этого перечисления. Далее логично перейти к использованию классов вместо перечислений.

6.5.2. Переотображение чисел в классы

В некоторых языках перечисления не могут иметь методов, потому что обрабатываются как именованные целые числа. В нашей игре мы сохраняем `rawMap` в виде целых значений и можем интерпретировать их как перечисления. Чтобы заменить эти перечисления, необходим способ преобразования чисел в наши новые экземпляры `RawTile2`. Проще всего для этого будет создать массив, где все значения будут идти в том же порядке, что и в перечислении.

Листинг 6.107. До

```
enum RawTile {
  AIR,
  FLUX,
  UNBREAKABLE,
  PLAYER,
  STONE, FALLING_STONE,
  BOX, FALLING_BOX,
  KEY1, LOCK1,
  KEY2, LOCK2
}
```

Листинг 6.108. После

```
const RAW_TILES = [
  RawTile2.AIR,
  RawTile2.FLUX,
  RawTile2.UNBREAKABLE,
  RawTile2.PLAYER,
  RawTile2.STONE, RawTile2.FALLING_STONE,
  RawTile2.BOX, RawTile2.FALLING_BOX,
  RawTile2.KEY1, RawTile2.LOCK1,
  RawTile2.KEY2, RawTile2.LOCK2
];
```

Таким образом можно легко отобразить числа в верные экземпляры. После избавления от `RawTile2` изменим оставшиеся ссылки `RawTile` на `RawTile2` — или, если это возможно, на `number`.

Листинг 6.109. До

```
let rawMap: RawTile[][] = [
  // ...
];
class Map {
  private map: Tile[][];
  constructor() {
    this.map = new Array(rawMap.length);
    for (let y = 0;
        y < rawMap.length;
        y++)
    {
      this.map[y] =
        new Array(rawMap[y].length);
      for (let x = 0;
          x < rawMap[y].length;
          x++)
        this.map[y][x] =
          transformTile(
            rawMap[y][x]);
    }
  }
  // ...
}
```

Листинг 6.110. После

```
let rawMap: number[][] = [
  // ...
];
class Map {
  private map: Tile[][];
  constructor() {
    this.map = new Array(rawMap.length);
    for (let y = 0;
        y < rawMap.length;
        y++)
    {
      this.map[y] =
        new Array(rawMap[y].length);
      for (let x = 0;
          x < rawMap[y].length;
          x++)
        this.map[y][x] =
          transformTile(
            RAW_TILES[rawMap[y][x]]);
    }
  }
  // ...
}
```

Невозможно поместить RawTile2

Отображает число в класс

```

}
function transformTile(tile: RawTile) {
  // ...
}

```

```

}
/// ...
function transformTile(tile: RawTile2) {
  // ...
}

```

Параметр изменен
на класс

Теперь мы получили ошибку в `transformTile`. Проблема состоит в сохранившемся `switch`, потому что, как и говорилось, методы закрытого (`private`) конструктора со `switch` не работают. Все сделанные нами действия были нацелены на удаление перечисления, а с ним и этого `switch`. Следовательно, следующее действие — перемещение кода в классы через `RawTile2`, причем во все классы.

Листинг 6.111. До

```

interface RawTileValue { }
class AirValue implements RawTileValue { }
class StoneValue implements RawTileValue { }
class Key1Value implements RawTileValue { }
/// ...
class RawTile2 {
  // ...
}
/// ...
function assertExhausted(x: never): never {
  throw new Error(
    "Unexpected object: " + x);
}
function transformTile(tile: RawTile2) {
  switch (tile) {
    case RawTile.AIR:
      return new Air();
    case RawTile.STONE:
      return new Stone(new Resting());
    case RawTile.KEY1:
      return new Key(YELLOW_KEY);
    // ...
    default: assertExhausted(tile);
  }
}

```

Листинг 6.112. После

```

interface RawTileValue {
  transform(): Tile;
}
class AirValue implements RawTileValue {
  transform() {
    return new Air();
  }
}
class StoneValue implements RawTileValue {
  transform() {
    return new Stone(new Resting());
  }
}
class Key1Value implements RawTileValue {
  transform() {
    return new Key(YELLOW_KEY);
  }
}
/// ...
class RawTile2 {
  // ...
  transform() {
    return this.value.transform();
  }
}
function transformTile(tile: RawTile2) {
  return tile.transform();
}

```

Код перемещен
прямо через
значения

Магический `assertExhausted`
больше не нужен

Наконец-то `switch` исчез. Теперь `transformTile` представляет собой одну строку, значит, проведем «Встраивание метода». И в завершение переименуем `RawTile2` в его постоянное имя `RawTile`.

РЕЗЮМЕ

- Для обеспечения лучшей инкапсуляции следует избегать раскрытия данных. Правило «Не использовать геттеры или сеттеры» (6.1.1) утверждает, что, кроме всего прочего, нельзя раскрывать закрытые (**private**) поля косвенно через геттеры или сеттеры. Избавиться от геттеров или сеттеров можно с помощью шаблона «Удаление геттера или сеттера» (6.1.3).
- Правило «Всегда избегать общих аффиксов» (6.2.1) утверждает, что если у нас есть методы и переменные с общим префиксом или суффиксом, то их необходимо поместить в один класс. Реализовать это можно с помощью шаблона «Инкапсуляция данных» (6.2.3).
- Шаблон «Обеспечение последовательности» (6.4.1) позволяет с помощью классов заставить компилятор обеспечить нужную последовательность выполнения действий, тем самым устранив инвариант последовательности.
- Еще один способ избавиться от перечислений — использовать класс с закрытым (**private**) конструктором. Подобное решение позволит в дальнейшем устранить перечисления и инструкции **switch**.

На этом завершается часть I книги. Здесь еще можно было бы продолжить инкапсулировать компоненты, такие как **inputs** и **handleInputs**. Можно даже инкапсулировать **player** и **map** в классе **Game**, но все это я оставляю вам на самостоятельную проработку.

А сверх этого, скажу вам, можно извлечь константы, улучшить именование переменных и методов, а также ввести пространства имен или налечь на код типов и преобразовать некоторые или даже все булевы параметры в перечисления, после чего заменить код типа классами (4.1.3) — таким образом вызвав сход снежной лавины с горы, лавины дальнейших преобразований кода. Суть в том, что описанное в части I — далеко не конец рефакторинга. Наоборот, это лишь его уверенное начало! В части II книги мы разберем общие принципы, которые позволят нам проводить истинно крутой рефакторинг.

Можно смело утверждать, что все сделанное нами в примере с игрой из части I уже существенно улучшило архитектуру кода по трем основным причинам.

1. Теперь расширение игры новыми видами **Tile** можно будет выполнять намного безопаснее и быстрее.
2. Стало намного проще анализировать код, потому что связанные переменные и функциональность теперь сгруппированы в классы и методы с понятными именами.
3. В обновленном коде у нас также есть возможность контролировать область видимости данных намного более точно, до мельчайших фрагментов данных.

Следовательно, тем программистам, которые будут работать с этим кодом, будет сложнее запрограммировать код, нарушающий нелокальные инварианты, которые, как было сказано в главе 2, являются причиной большинства багов.

В нескольких местах части I нам приходилось немного погружаться в анализ кода, чтобы дать компонентам подходящие имена или решить, нужно ли оставлять элементы вместе в одной конструктивной единице. Однако такой анализ всегда был краток: нам ни разу не пришлось тратить время на разбор каких-либо странных особенностей кода, например, почему один из циклов `for` в `update` выполняется назад или почему мы перемещаем входные данные в стек, а не выполняем ходы напрямую (мы могли даже и не заметить стек). Чтобы разобраться все подобные тонкости кода, необходимо было бы провести намного больше времени за углубленным его анализом, но нам для выполнения рефакторинга этого не потребовалось.

Часть II

Применение полученных знаний в реальной жизни

В части II мы подробнее разберем возможное использование правил и шаблонов рефакторинга в реальных задачах, добавив контекст. Попутно познакомимся с практиками, позволяющими полноценно использовать доступные инструменты, и обсудим принципы, на которых они построены.

Чтобы реализовать поставленные цели, перейдем на более высокий уровень абстракции. Вместо разбора конкретных правил и шаблонов мы будем изучать социотехнические аспекты, влияющие на рефакторинг и качество кода. Параллельно с этим я буду давать действенные советы касательно навыков, культуры и инструментов (рис. II).



Рис. II. Навыки, культура и инструменты



Сотрудничество с компилятором

В этой главе

- ✓ Разбор сильных и слабых сторон компиляторов.
- ✓ Использование сильных сторон компиляторов для устранения инвариантов.
- ✓ Разделение ответственности с компилятором.

Когда программист еще только учится писать код, компилятор может показаться ему скучным занудой и придирааться-то по, казалось бы, совершенным мелочам. Он, компилятор, все воспринимает слишком буквально, не дает поблажек и ругается, видите ли, даже на малейшие промахи. Однако при правильном использовании компилятор становится одним из наиболее мощных инструментов повседневной работы программиста. Он не только трансформирует программный код из высокоуровневого языка в низкоуровневый, но также проверяет ряд его свойств и гарантирует отсутствие ошибок определенного вида при дальнейшем выполнении программы.

В этой главе мы начнем знакомиться с компилятором ближе, чтобы получить возможность активно им пользоваться, опираясь на его сильные стороны. При этом узнаем и то, на что он не способен, чтобы когда-нибудь по незнанию не возложить слишком большие необоснованные надежды на его слабости и не потерпеть закономерное фиаско.

Команде разработчиков, близко знакомых с компилятором, следует сделать его частью этой команды и поделиться с ним ответственностью за соблюдение корректности кода, позволить ему помогать в создании правильных программ. Если же программисты начинают противиться его рекомендациям, стремятся обходить его советы, то, как правило, это выливается в непомерное возрастание рисков получить глобальные ошибки в будущем ради получения минимальной сиюминутной сомнительной выгоды.

Но гораздо логичнее принять нового коллегу, компилятор в команду, начать доверять ему: прикладывать усилия, чтобы в коде появлялось как можно меньше опасных инвариантов, внимательно вчитываться в его сообщения и предупреждения.

И наконец после ознакомления программистов с компилятором и принятия его в команду должно восторжествовать четкое осознание и принятие того факта, что компилятор лучше справляется с прогнозированием поведения программы, чем программист-человек. Фактически он просто робот. Он не устает, даже когда работает с сотнями тысяч строк кода. Он может проверять свойства, которые не смог бы реально проверить ни один человек-разработчик. Это очень мощный инструмент, и его необходимо грамотно использовать!

7.1. БЛИЗКОЕ ЗНАКОМСТВО С КОМПИЛЯТОРОМ

В мире уже существует больше компиляторов, чем я могу сосчитать, и все время появляются новые. Поэтому вместо того, чтобы разбирать какой-то конкретный, мы обсудим свойства, присущие большинству из них, включая передовые варианты для Java, C# и TypeScript.

Компилятор — это программа, которая хороша для определенных задач, таких как обеспечение логики и стабильности. Вопреки распространенным легендам, при выполнении компиляции одного и того же кода несколько раз подряд мы не получим разных результатов. При этом нужно понимать, что компилятор не всесилен и в определенных вещах, наоборот, плох — к примеру, от него никак не стоит ожидать успеха в принятии решения, выработке суждения о том, что лучше. Эти инструменты — компиляторы — в процессе работы следуют общей идиоме: «Если сомневаешься, спроси».

Фундаментальная цель компилятора состоит в генерации программы на определенном исполняемом языке таким образом, чтобы она была функционально эквивалентна нашей исходной программе. Однако в современные версии компиляторов для помощи разработчику заложена возможность проверки вероятных ошибок даже на позднейшей стадии — выполнения программы. Вот на этом

уникальном функционале компиляторов я и собираюсь сфокусировать посыл этой главы.

Как это обычно бывает с программированием, наилучшее понимание приходит через практику. Нам необходимо хорошо разобраться в том, что компилятор может и чего не может, а также понять, как его можно обхитрить. Для этого у меня всегда в запасе имеются экспериментальные проекты, позволяющие проверить, как компилятор обрабатывает тот или иной момент. Может ли он гарантировать инициализацию этого компонента? Сообщит ли он мне о том, что `x` здесь может быть `null`?

В последующих разделах мы постараемся найти ответы на эти и множество других вопросов, подробно разобрав некоторые из наиболее распространенных сильных и слабых черт современных компиляторов.

7.1.1. Слабость: проблема останова имеет ограниченную информативность во время компиляции

Причина, по которой мы не можем сказать точно, что произойдет при выполнении, называется *проблемой останова*. Ее смысл в том, что без запуска программы на выполнение мы не можем знать, как конкретно она себя поведет, и даже при ее выполнении мы наблюдаем всего один из многих путей прохода по нашей программе.

ПРОБЛЕМА ОСТАНОВА

Как правило, программы в своей основе непредсказуемы.

В качестве быстрой демонстрации верности данного утверждения рассмотрим следующую программу.

Листинг 7.1. Программа без ошибок среды выполнения

```
if (new Date().getDay() === 35)
    5.foo();
```

Известно, что `getDay` никогда не вернет 35. Значит, что бы ни находилось внутри `if`, эта инструкция никогда не выполнится. То есть ее можно проигнорировать, даже принимая во внимание, что она и так провалится, ведь в числе 5 не определен метод `foo`.

Некоторые программы будут определенно давать сбой или отклоняться компилятором. Некоторые, наоборот, точно к сбою не приведут и будут допущены

компилятором к выполнению. Проблема останова означает, что компилятор должен решить, как обрабатывать программу в промежуточных ситуациях, между этими вариантами. Иногда компилятор допускает программы, которые могут вести себя ожидаемым образом, включая сбои при выполнении. В других случаях компилятор запрещает программу, если не может гарантировать ее безопасное выполнение. Компилятор анализирует предложенный код. Этот анализ называется *консервативным*.

Консервативный анализ подтверждает, что в программе отсутствует возможность конкретного сбоя. Полагаться мы можем только на такую формулировку результата анализа.

Обратите внимание, что проблема останова не является специфичной для какого-либо компилятора или языка. Это характерная черта всех языков программирования. На деле подверженность проблеме останова лежит глубоко в сути любого языка программирования как такового. Различаются же языки и компиляторы тем, насколько и когда они проявляют консервативность, а когда — нет.

7.1.2. Сильная сторона: достижимость гарантирует возвращение из методов

В одном из видов консервативного анализа проверяется, выполняет ли метод `return` (возврат и корректное завершение) по каждому пути. Обычно мы не можем выйти из метода, не достигнув инструкции `return`.

В TypeScript допускается завершать метод и по-другому. Но если использовать `assertExhausted` из главы 4, как раз можно получить нужную корректную исходную форму программного кода. И хотя следующий ниже по тексту код будет выполняться с ошибкой, ключевое слово `never` вынуждает компилятор проанализировать, есть ли иная возможность достичь `assertExhausted`. А именно: в данном примере компилятор выяснит, что мы не проверили все значения перечисления.

Листинг 7.2. Ошибка компиляции ввиду достижимости

```
enum Color {
  RED, GREEN, BLUE
}
function assertExhausted(x: never): never {
  throw new Error("Unexpected object: " + x);
}
function handle(t: Color) {
  if (t === Color.RED) return "#ff0000";
  if (t === Color.GREEN) return "#00ff00";
  assertExhausted(t);
}
```

Компилятор выдает ошибку, так как мы не обрабатывали `Color.BLUE`

С помощью этой проверки в подразделе 4.3.2 мы узнавали, все ли кейсы покрыл оператор `switch`. В типизированных языках, где она обычно присутствует по стандарту языка, такая проверка называется *проверкой полноты*.

Как правило, воспользоваться этим анализом сложно, особенно при соблюдении правила пяти строк, так как возможность определить, сколько `return` у нас есть и где они находятся, становится завуалированной.

7.1.3. Сильная сторона: явное присваивание предотвращает обращение к неинициализированным переменным

Еще одно сильное свойство компиляторов в том, что они качественно выполняют проверку на предмет присваивания переменной значения до ее использования. Такие переменные, конечно, не обязательно содержат что-то полезное, но им определенно были явно присвоены *какие-то значения*.

Эта проверка применима ко всем локальным переменным, в частности, в случаях, когда мы хотим инициализировать переменные внутри `if`. Иногда мы рискуем не инициализировать переменную по какому-то пути. Рассмотрим следующий код, который ищет, кому принадлежит имя (`name`) `John`. При достижении инструкции `return` ничто не гарантирует, что переменная `result` была инициализирована. Обнаружив это, компилятор программу не пропустит.

Листинг 7.3. Неинициализированная переменная

```
let result;
for (let i = 0; i < arr.length; i++)
  if (arr[i].name === "John")
    result = arr[i];
return result;
```

Мы сами можем знать, что в этом коде `arr` определенно содержит элемент, чьим `name` является `John`. Но компилятор в данном случае перестраховывается. Оптимальным выходом из сложившейся ситуации будет научить его тому, что нам самим уже известно, то есть дать понять, что программа найдет элемент с именем `John`.

Обучить компилятор можно с помощью другого направления анализа явного присваивания: анализа полей только для чтения (или заключительных). Поле только для чтения должно быть инициализировано перед закрытием конструктора. Это значит, что компилятор может быть уверен: мы обязательно присвоим полю значение либо в конструкторе, либо непосредственно в месте объявления.

С помощью этого строгого подхода можно гарантировать компилятору существование конкретного значения поля. В примере выше мы можем обернуть

массив в класс с полем только для чтения для объекта с именем `John`. Таким образом, можно даже избежать необходимости перебирать весь список в поиске. Конечно, внесение этого изменения вынудит изменить процесс создания списка. Тем не менее так будет исключена возможность того, что кто-нибудь по случайности оставит объект `John` незамеченным, тем самым устранив инвариант.

7.1.4. Сильная сторона: контроль доступа помогает инкапсулировать данные

Компилятор также превосходно справляется с контролем доступа, который мы используем, когда у нас есть инкапсулированные данные. Если сделать их компонент закрытым (`private`), то можно быть уверенными, что он по случайности не проскочит вне внутреннего контента, компилятор отловит эту ситуацию. В главе 6 мы видели много примеров того, как и почему стоит использовать данную технику, поэтому здесь дополнительно углубляться в нее не станем. Однако хотелось бы развеять одно распространенное среди начинающих программистов заблуждение, что `private` относится к классу, а *не* к объекту. То есть на самом деле мы можем инспектировать закрытые (`private`) компоненты другого объекта, если он находится в том же классе.

Если у нас есть методы, чувствительные к инвариантам, то можно защитить их, сделав закрытыми. Например, так.

Листинг 7.4. Ошибка компиляции из-за доступа

```
class Class {
  private sensitiveMethod() {
    // ...
  }
}
let c = new Class();
c.sensitiveMethod();
```

Здесь возникла
ошибка компиляции

7.1.5. Сильная сторона: проверка типов подтверждает свойства

Последняя сильная из затронутых нами сторона компиляторов, она же самая значительная — проверка типов. Соответствующий модуль отвечает за проверку существования переменных и компонентов данных, и мы использовали эту функциональность, когда переименовывали что-либо для получения ошибок в части I книги. «Обеспечение последовательности» (6.4.1) мы также реализовывали с помощью модуля проверки типов.

В этом примере закодирован список, который не может быть пуст, потому что состоять он может только из одного элемента или элемента, сопровождаемого списком.

Листинг 7.5. Ошибка компиляции из-за типов

```
interface NonEmptyList<T> {
  head: T;
}
class Last<T> implements NonEmptyList<T> {
  constructor(public readonly head: T) { }
}
class Cons<T> implements NonEmptyList<T> {
  constructor(
    public readonly head: T,
    public readonly tail: NonEmptyList<T>) { }
}
function first<T>(xs: NonEmptyList<T>) {
  return xs.head;
}
first([]);
```

Ошибка
типа

Вопреки распространенному жаргону, строгая типизированность не является двоичным свойством (есть/нет). Степень строгости типизации языков программирования может варьироваться от слабой до сильной в широком диапазоне. Подмножество TypeScript, которое мы рассматриваем в этой книге, ограничивает свои возможности по заданию типов, чтобы быть наравне с Java и C#. Такой степени строгости типизации достаточно, чтобы научить компилятор отслеживать сложные свойства, например невозможность извлечения чего-либо из пустого стека. Однако для этого необходимо хорошо знать теорию. В нескольких языках есть еще более строгие системы типов, и ниже я перечислил наиболее интересные из них (с указанием языков) в порядке возрастания строгости:

- одалживание типов (Rust);
- вывод полиморфных типов (OCaml и F#);
- классы типов (Haskell);
- тип «объединение» и тип «пересечение» (TypeScript);
- зависимые типы (Coq и Agda).

В языках с хорошим модулем проверки типов в компиляторе наивысшим возможным уровнем безопасности будет обучить такой модуль свойствам конкретной программы, что равноценно использованию самых запутанных статических анализаторов или ручному подтверждению свойств, что оказывается наиболее сложным способом, часто наводящим ошибки. Эта книга не имеет целью обучение данной технике, но сама техника очень перспективна, предоставляет множество удобств, поэтому я надеюсь, что побудил в вас интерес к самостоятельному ее изучению.

7.1.6. Слабость: разыменовывание null рушит приложение

В противоположном конце диапазона у нас `null`. Опасность `null` в том, что при попытке вызова для него методов он приводит к сбою. Некоторые инструменты способны обнаружить часть таких случаев, но они редко замечают все, то есть полагаться на эти инструменты полностью нельзя.

Если отключить в TypeScript строгую проверку на `null`, то он будет вести себя подобно другим популярным языкам, в которых этот код допускается. То есть он пропустит присутствие `null`, попустительствуя, например, тому, что мы можем вызвать `average(null)`, сломав тем самым выполнение программы.

Листинг 7.6. Потенциальное разыменовывание `null`, но без ошибки компиляции

```
function average(arr: number[]) {  
    return sum(arr) / arr.length;  
}
```

Риск возникновения ошибок среды выполнения означает, что при работе с допускающими `null` значениями необходима особая осторожность. Я обычно говорю так: «Если вы не видите необходимости проверки переменной на `null`, значит, она, вероятно, `null` и есть». Чтобы избежать аварийных остановок выполнения программы, лучше перестраховаться.

Некоторые IDE могут указывать на избыточность проверки переменных на `null`, и я понимаю, насколько эта половинчатое предупреждение об ошибке может раздражать. Тем не менее настоятельно рекомендую не игнорировать эти проверки, пока вы не будете абсолютно уверены, что они слишком затратны или излишни и никогда не перехватят ошибку.

7.1.7. Слабость: арифметические ошибки вызывают переполнение или сбой

Еще одна слабость компиляторов в том, что они иногда не проверяют критическую ситуацию деления (или деления с остатком) на нуль. Компилятор даже не проверяет, может ли что-либо вызвать переполнение. Все это называется *арифметическими ошибками*. Деление целого числа на нуль приводит к сбою программы. Что еще хуже, переполнения просто молча ведут к нарушению поведения программы.

Обратимся к предыдущему примеру: даже если мы знаем, что наш код не вызывает `average` с `null`, практически ни один компилятор не заметит потенциальное деление на нуль, если в программе будет этот вызов с пустым массивом.

Листинг 7.7. Потенциальное деление на нуль, но ошибки компиляции нет

```
function average(arr: number[]) {  
    return sum(arr) / arr.length;  
}
```

Поскольку компилятор здесь не особо помогает, при программировании арифметики нужно быть очень внимательными. Убедитесь, что делитель не может быть нулевым, а также что не добавляете или отнимаете числа, способные вызвать положительное или отрицательное переполнение. Либо используйте какую-нибудь вариацию `BigIntegers`.

7.1.8. Слабость: ошибки выхода за допустимый диапазон вызывают сбой приложения

Еще одно дело, в котором на компилятор нельзя положиться, — это обращение к данным напрямую. Если программа пытается обратиться к индексу, который находится вне границ структуры данных, это приводит к ошибке выхода из допустимого диапазона.

Представьте, что у нас есть функция поиска индекса первого простого числа в массиве. С ее помощью это можно сделать так.

Листинг 7.8. Потенциальное обращение к элементу вне допустимого диапазона, но ошибки компиляции не возникает

```
function firstPrime(arr: number[]) {
    return arr[indexOfPrime(arr)];
}
```

Однако, если в массиве простого числа не окажется, такая функция вернет `-1`, что вызовет ошибку выхода за допустимый диапазон.

Разобраться с этим ограничением можно двумя способами: либо задать в программе обход всей структуры данных (это если есть риск не найти предполагаемый элемент), либо использовать уже знакомый нам подход явного присваивания с целью подтвердить обязательное существование элемента.

7.1.9. Слабость: бесконечные циклы стопорят приложение

Совершенно иное проявление сбоя программы — когда вообще ничего не происходит. Такое случается, когда код молча входит в бесконечный цикл, а мы при этом видим просто пустой экран. Компиляторы, как правило, подобных ошибок не замечают.

В примере ниже требуется определить, находимся ли мы внутри строки. Но мы по ошибке забыли передать предыдущую `quotePosition` во второй вызов `indexOf`. И вот пожалуйста: если `s` содержит кавычку, то получается бесконечный цикл, а компилятор этого не замечает.

Листинг 7.9. Потенциально бесконечный цикл, но ошибки компиляции нет

```
let insideQuote = false;
let quotePosition = s.indexOf("\"");
while(quotePosition >= 0) {
    insideQuote = !insideQuote;
    quotePosition = s.indexOf("\"");
}
```

Вероятность возникновения подобных проблем уменьшается за счет отказа от `while` в пользу `for` или `foreach`. А недавно появилась возможность использовать высокоуровневые конструкции, такие как `forEach` в TypeScript или потоковые операции в Java и LINQ в C#, их тоже можно использовать.

7.1.10. Слабость: взаимные блокировки и состояния гонки вызывают нежелательное поведение

Заключительная категория сложностей кроется в многопоточности программ. При наличии множества выполняемых потоков, использующих общие изменяемые данные, могут возникать разнообразные проблемы: состояние гонки, взаимные блокировки, состояние голодания и т. д.

TypeScript не поддерживает многопоточность, поэтому я не могу привести примеры подобных ошибок в этом языке. Однако я могу продемонстрировать их с помощью псевдокода.

Состояние гонки — это первая проблема, с которой мы сталкиваемся при работе с потоками. Она возникает, когда два или более потока соперничают за право прочесть/записать общую переменную. В результате может получиться так, что они оба прочтут эту переменную до ее обновления.

Листинг 7.10. Псевдокод для состояния гонки

```
class Counter implements Runnable {
    private static number = 0;
    run() {
        for (let i = 0; i < 10; i++)
            console.log(this.number++);
    }
}
let a = new Thread(new Counter());
let b = new Thread(new Counter());
a.start();
b.start();
```

Листинг 7.11. Пример вывода

```
1
2
3
4
5      | Повтор чисел...
5      |
7 ←-----| ...и их пропуск
8
...
```

Чтобы исправить эту ошибку, вводятся так называемые замки. В следующем примере мы пропишем для каждого потока такой замок и прежде, чем продолжать выполнение в одном потоке, убедимся в том, что замок другого потока действительно свободен.

Листинг 7.12. Псевдокод для взаимной блокировки

```

class Counter implements Runnable {
    private static number = 0;
    constructor(
        private mine: Lock, private other: Lock) { }
    run() {
        for (let i = 0; i < 10; i++) {
            mine.lock();
            other.waitFor();
            console.log(this.number++);
            mine.free();
        }
    }
}
let aLock = new Lock();
let bLock = new Lock();
let a = new Thread(new Counter(aLock, bLock));
let b = new Thread(new Counter(bLock, aLock));
a.start();
b.start();

```

Листинг 7.13. Пример вывода

```

1
2
3
4

```

← Ничего не происходит

Проблема, проиллюстрированная в данном случае, называется *взаимной блокировкой*: оба потока заблокированы в ожидании разблокирования друг друга для продолжения. Типичная бытовая аналогия для данной ситуации — это когда у двери встречаются два человека, которые оба стремятся уступить друг другу дорогу.

И наконец, последняя категория возникающих при многопоточности ошибок. Сделаем циклы бесконечными и просто будем выводить информацию о том, какой поток выполняется.

Листинг 7.14. Псевдокод для голодания

```

class Printer implements Runnable {
    constructor(private name: string,
        private mine: Lock, private other: Lock) { }
    run() {
        while(true) {
            other.waitFor();
            mine.lock();
            console.log(this.name);
            mine.free();
        }
    }
}
let aLock = new Lock();
let bLock = new Lock();
let a = new Thread(
    new Printer("A", aLock, bLock));
let b = new Thread(
    new Printer("B", bLock, aLock));
a.start();
b.start();

```

Листинг 7.15. Пример вывода

```

A
A
A
A

```

← Продолжается бесконечно

Проблема здесь в том, что В никогда не удастся начать выполнение. Это достаточно редкая ситуация, но технически вполне возможная. Называется она *голоданием*. В качестве аналогии можно привести мост с единственной полосой движения, где машины с одной стороны обязаны пропускать встречные, поток которых никак не заканчивается.

Противодействию подобным проблемам посвящено множество книг. Я же в качестве оптимального совета рекомендую вам избегать использования множественности потоков, обрабатывающих общие изменяемые данные. На счет того, нужно ли вообще всегда и полностью избегать «множественности потоков», «общих данных» или «изменяемости», уже надо решать в конкретной ситуации.

7.2. ИСПОЛЬЗОВАНИЕ КОМПИЛЯТОРА

Теперь, когда с компилятором мы познакомились, пришло время включить его в дело. Он должен, как было сказано выше, стать частью команды. Понимая то, как он способен помочь, мы должны спроектировать ПО, задействовав сильные стороны компилирования и избежав слабых. Бороться с ним или же его обманывать нам точно не стоит.

Люди зачастую уподобляют разработку программ их конструированию. Но, как заметил Мартин Фаулер в своем блоге, это одна из наиболее вредных аналогий в нашей области. Программирование никак не является конструированием. Это взаимодействие, причем на нескольких уровнях.

- Мы взаимодействуем с компьютером, когда говорим ему, что делать.
- Мы взаимодействуем с другими разработчиками, когда они читают наш код.
- Мы взаимодействуем с компилятором, когда просим его обработать наш код.

В таком ракурсе у программирования куда больше общего с литературой. Мы собираем знания об интересующей нас области, формируем в сознании модель, после чего выражаем ее в коде. Здесь можно привести прекрасную цитату.

Структуры данных — это алгоритмы, застывшие во времени.

Не могу вспомнить имя

Дэн Норт выразил похожую мысль, что программы — это коллективное знание команды разработчиков об их области, застывшее во времени. Программа представляет завершенное однозначное описание всего, что разработчики считают действительным относительно конкретной области. В этой аналогии компилятор выступает в роли литературного редактора, который гарантирует соответствие наших текстов определенным критериям.

7.2.1. Подключаем компилятор к работе

Как уже неоднократно было показано в этой книге, есть несколько способов разрабатывать программы с учетом возможностей компилятора. Вот небольшой список некоторых из этих способов.

Применение компилятора для генерации списка «надо сделать»

Чаще всего мы задеиствовали компилятор для обнаружения всех точек, где необходимо внести изменения. В этих случаях мы просто переименовывали оригинальный метод, и компилятор выдавал ошибки во всех местах использования этого метода со старым именем. Таким образом мы обеспечивали безопасность, будучи уверенными, что компилятор не упустит ни одну ссылку. Это отлично работает, но только когда у нас нет других ошибок.

Представьте, что нам нужно найти каждую точку использования перечисления, чтобы локализовать использование `default`. Все вхождения перечисления, включая те, что задеиствуют `default`, можно найти путем присоединения к имени, например, `_handled`. Теперь компилятор выдаст ошибку в каждой точке использования перечисления. Далее при завершении редактирования каждой такой точки можно просто добавлять к имени перечисления `_handled`, устранив таким образом ошибку.

Листинг 7.16. Поиск вхождений перечислений с помощью ошибок компиляции

```
enum Color_handled {
    RED, GREEN, BLUE
}
function toString(c: Color) {
    switch (c) {
        case Color.RED: return "Red";
        default: return "No color";
    }
}
```



Ошибки компиляции

После завершения можно легко удалить отовсюду `_handled`.

Гарантия безопасности за счет обеспечения последовательности

Шаблон «Обеспечение последовательности» подразумевает информирование компилятора об инварианте в нашей программе, что позволяет сделать такой инвариант свойством. В результате его уже нельзя будет по случайности нарушить, потому что компилятор гарантирует сохранение этого свойства при каждой компиляции программы.

В главе 6 мы разбирали внутреннюю и внешнюю версии использования классов для соблюдения последовательности. Обе эти версии гарантируют, что строка будет заранее переведена в верхний регистр.

Листинг 7.17. Внутренний

```
class CapitalizedString {
  private value: string;
  constructor(str: string) {
    this.value = capitalize(str);
  }

  print() {
    console.log(this.value);
  }
}
```

Листинг 7.18. Внешний

```
class CapitalizedString {
  public readonly value: string;
  constructor(str: string) {
    this.value = capitalize(str);
  }
}

function print(str: CapitalizedString) {
  console.log(str.value);
}
```

Повышение безопасности за счет инкапсуляции

Обеспечивая строгую инкапсуляцию посредством реализуемого компилятором контроля доступа, можно локализовать наши инварианты. Инкапсулируя данные, мы получаем уверенность, что они сохранятся в ожидаемой форме.

Как уже было показано, можно предотвратить случайный вызов вспомогательного метода `depositHelper`, сделав его закрытым (`private`).

Листинг 7.19. Закрытый вспомогательный метод

```
class Transfer {
  constructor(from: string, private amount: number) {
    this.depositHelper(from, -this.amount);
  }
  private depositHelper(to: string, amount: number) {
    let accountId = database.find(to);
    database.updateOne(accountId, { $inc: { balance: amount } });
  }
  deposit(to: string) {
    this.depositHelper(to, this.amount);
  }
}
```

Повышение уровня безопасности за счет обнаружения компилятором неиспользуемого кода

Изучая шаблон «Пробное удаление с последующей компиляцией» (4.5.1), мы задействовали компилятор для проверки факта действительного использования кода. При одновременном удалении нескольких методов компилятор может быстро просканировать всю базу кода, сообщив, какие из них используются.

С помощью этого подхода мы избавляемся от методов в интерфейсах. Компилятор, конечно, не может знать, будут ли они использованы или же не используются вовсе. Но если нам известно, что интерфейс задействуется только внутренне, то можно просто попробовать удалить из него методы и посмотреть, примет ли компилятор в итоге программу и сообщит ли об ошибках.

В коде из главы 4 можно безопасно удалить оба метода `m2` и даже метод `m3`.

Листинг 7.20. Пример с методами, которые можно удалить

```
interface A {
    m1(): void;
    m2(): void;
}
class B implements A {
    m1() { console.log("m1"); }
    m2() { m3(); }
    m3() { console.log("m3"); }
}
let a = new B();
a.m1();
```

Повышение безопасности с помощью явных значений

И последнее. Ранее в этой главе приводилась структура данных списка, которая не может быть пустой. Факт этот гарантируется с помощью полей только для чтения. Такие поля проходят через анализ явного присваивания и при завершении конструктора должны иметь значение. Даже в языке, поддерживающем несколько конструкторов, нельзя получить объект с неинициализированными полями только для чтения.

Листинг 7.21. Непустой список из-за полей только для чтения

```
interface NonEmptyList<T> {
    head: T;
}
class Last<T> implements NonEmptyList<T> {
    constructor(public readonly head: T) { }
}
class Cons<T> implements NonEmptyList<T> {
    constructor(
        public readonly head: T,
        public readonly tail: NonEmptyList<T>) { }
}
```

7.2.2. Не перечьте компилятору

Есть и обратная сторона вопроса: досадно каждый раз видеть, как кто-то намеренно противится компилятору и не дает ему выполнять свою работу. Это можно делать несколькими способами, и ниже я приведу краткий обзор наиболее распространенных. Причиной несогласия с компилятором обычно служат три недочета разработчика, каждому из которых будет посвящен отдельный раздел: непонимание типов, лень и непонимание архитектуры.

Типы

Как уже говорилось, модуль проверки типов является сильнейшей стороной компилятора. Следовательно, его обман или отключение относятся к самым серьезным нарушениям. Есть три вида неверного использования этого функционала.

Приведение типов

Первый — это использование приведения типов, с помощью которого мы как бы говорим компилятору, что знаем работу с типами лучше его. Приведение типов не позволяет ему оказать нам помощь и, по сути, отключает его от проверки конкретной переменной или выражения. Типы не интуитивны. Навык их понимания нужно вырабатывать. Необходимость в приведении является показателем того, что либо мы не понимаем типов, либо их не понял кто-то другой.

Приведение типов мы используем, когда перед нами не тот тип, который нужен. Можно сравнить такой подход с приемом болеутоляющего для устранения хронической боли: это поможет сейчас, но никак не исправит суть проблемы.

Зачастую приведение используется, когда мы получаем нетипизированный JSON из веб-сервиса. В примере ниже разработчик был уверен, что JSON в переменной всегда был числом.

Листинг 7.22. Приведение типа

```
let num = <number> JSON.parse(variable);
```

Здесь возможны две ситуации: либо получить входные данные из другого источника, который мы контролируем, например из собственного веб-сервиса, либо — что будет временным решением — использовать на стороне отправителя те же типы, что и на стороне получателя. В этом нам могут оказать содействие несколько библиотек. Если данные поступают от стороннего источника, то самым безопасным решением будет спарсить их с помощью кастомного парсера. Вот как мы обработали ввод клавиш в части I книги.

Листинг 7.23. Парсинг ввода из строки в кастомные классы

```
window.addEventListener("keydown", e => {
  if (e.key === LEFT_KEY || e.key === "a") inputs.push(new Left());
  else if (e.key === UP_KEY || e.key === "w") inputs.push(new Up());
  else if (e.key === RIGHT_KEY || e.key === "d") inputs.push(new Right());
  else if (e.key === DOWN_KEY || e.key === "s") inputs.push(new Down());
});
```

Динамические типы

Хуже косвенной «нейтрализации» проверки типов может быть только ее *фактическое* отключение. Это происходит при использовании динамических типов: в TypeScript с помощью `any` (`dynamic` в C#). Несмотря на то что такой прием

может показаться полезным, особенно при обмене объектами JSON через HTTP, он может спровоцировать великое множество потенциальных ошибок, таких как обращение к несуществующим полям или к полям, типы которых отличаются от ожидаемых, в результате мы можем попытаться перемножить две строки.

Недавно я столкнулся с проблемой, когда код TypeScript выполнялся в версии ES6, а компилятор был настроен на ES5, из-за чего о некоторых методах ES6 он был не осведомлен. Если конкретно, то он не знал о `findIndex` для массивов. Стремясь исправить эту проблему, разработчик привел переменную к типу `any`, чтобы компилятор допустил для нее любой вызов.

Листинг 7.24. Использование `any`

```
(<any> arr).findIndex(x => x === 2);
```

Отсутствие этого метода при выполнении являлось маловероятным, так что особой опасности этот прием не представлял. Тем не менее обновление конфигурации до ES6 оказалось бы более безопасным и уже постоянным решением.

Типы среды выполнения

Третий способ, которым люди пытаются обмануть компилятор, — это перемещение понимания контекста в среду выполнения. Такое решение полностью противоречит рекомендациям данной книги. Вот довольно распространенный пример того, как это происходит. Представьте, что есть метод с десятью параметрами. Такое количество параметров может вызывать путаницу, к тому же при каждом добавлении или удалении параметра необходимо исправить метод во всех местах его вызова. Поэтому вместо использования десяти параметров принимается решение использовать только один: `Map` из строк в значения. Тогда станет возможно с легкостью добавлять в него значения без необходимости менять какой-либо код. И фактически это будет ужасной идеей, потому что окажется отброшенным в сторону понимание контекста. Компилятор не может знать, какие ключи существуют в `Map`, и, следовательно, не может проверить, есть ли шанс, что мы обратимся к несуществующему ключу. Мы перешли от сильной стороны «проверки типов» к слабости появления «ошибок выхода из допустимого диапазона». Фигурально выражаясь: «Устали от стирки? Простейшее решение: сожгите всю одежду».

В этом примере вместо передачи трех отдельных аргументов мы передаем одну карту, после чего сможем извлекать значения с помощью `get`.

Листинг 7.25. Типы среды выполнения

```
function stringConstructor(
  conf: Map<string, string>,
  parts: string[]) {
  return conf.get("prefix")
    + parts.join(conf.get("joiner"))
    + conf.get("postfix");
}
```

Более безопасным решением будет создать объект с этими конкретными полями.

Листинг 7.26. Статические типы

```
class Configuration {
    constructor(
        public readonly prefix: string,
        public readonly joiner: string,
        public readonly postfix: string) { }
}
function stringConstructor(
    conf: Configuration,
    parts: string[]) {

    return conf.prefix
    + parts.join(conf.joiner)
    + conf.postfix;
}
```

Лень

Второй серьезный недостаток разработчика — это лень. Я не считаю, что программистов нужно винить за леность, поскольку именно она и приводит большинство людей в эту профессию. Мы самоотверженно проводим часы или недели, неустанно трудясь над автоматизацией чего-то, что не хотим делать сами. Лень как качество делает нас более эффективными программистами, а вот лень как привычка, наоборот, может резко снизить качество кода.

Еще одна причина моей толерантности к этой черте характера в том, что разработчики зачастую находятся в состоянии сильного давления и жестких дедлайнов. В таком состоянии любой будет искать самые короткие пути и возможности их еще больше сократить. Вот только проблема в том, что зачастую в ход идут краткосрочные меры или компромиссы.

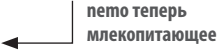
Установки по умолчанию

Значения по умолчанию мы немного обсуждали в части I. При использовании предустановленного (default) варианта рано или поздно кто-то добавит значение, которое не должно быть предустановленным, и забудет внести исправление. Вместо использования предустановок нужно возложить на разработчика ответственность за каждое добавление или изменение чего-либо. Для этого не нужно вообще использовать значение по умолчанию, чтобы компилятор подталкивал разработчика к принятию решения. Ведь, помимо всего прочего, это может помочь выявить пробелы в понимании решаемой задачи, когда компилятор задает вопрос, на который у нас нет ответа.

В коде ниже разработчик хотел воспользоваться тем фактом, что большинство животных являются млекопитающими, и предустановил это значение. Однако мы запросто можем забыть переопределить эту установку, особенно из-за отсутствия помощи от компилятора.

Листинг 7.27. Ошибка из-за предустановленных аргументов

```
class Animal {
  constructor(name: string, isMammal = true) { ... }
}
let nemo = new Animal("Clown fish");
```



немо теперь
млекопитающее

Наследование

Через правило «Наследовать только от интерфейсов» (4.3.2) я ясно выразил свое мнение и доводы в отношении совместного использования кода через наследование. Наследование — это форма предустановленного поведения, о котором мы говорили в предыдущем разделе. Более того, наследование добавляет связанность между реализующими ее классами.

В примере ниже если добавим еще один метод в `Mammal`, то нужно будет помнить о необходимости проверки валидности этого метода во всех классах-потомках. А о такой проверке легко забыть или просто упустить ее из вида. В коде примера мы добавили в суперкласс `Mammal` метод `layEggs`, который сработает для большинства потомков, кроме `Platypus` (утконос).

Листинг 7.28. Проблема, вызванная наследованием

```
class Mammal {
  laysEggs() { return false; }
}
class Dolphin extends Mammal { }
/// ...
class Platypus extends Mammal {
}
```



Нужно переписать
`laysEggs`

Непроверенные исключения

Исключения зачастую появляются в двух обликах: те, которые мы вынуждены обработать, и те, которые не нуждаются в обработке. Но если исключение может произойти, оно должно быть обработано где-либо или хотя бы вызывающий код проинформирован о том, что это не сделано. В этом и заключается суть проверенных исключений. Непроверенные исключения нужно использовать только для того, что не может произойти, например, когда мы знаем, что некий инвариант будет верен, но не можем выразить его в языке. Наличие одного непроверенного исключения под названием `Impossible` кажется достаточным. Хотя, как и с любым инвариантом, мы рискуем, что однажды он будет нарушен и возникнет необработанное исключение `Impossible`.

В примере ниже наблюдается проблема с использованием непроверенного исключения для чего-то, что не является невозможным. Следуя логике, мы проверяем входной массив на пустоту, потому что это может вызвать арифметическую

ошибку. Однако из-за того, что мы используем непроверенное исключение, вызывающий компонент по-прежнему может вызвать наш метод с пустым массивом и программа продолжит давать сбой.

Листинг 7.29. Использование непроверенного исключения

```
class EmptyArray extends RuntimeException { }
function average(arr: number[]) {
  if (arr.length === 0) throw new EmptyArray();
  return sum(arr) / arr.length;
}
/// ... console.log(average([]));
```

Более удачным решением будет использовать проверенное исключение. Если локальный инвариант в точке вызова гарантирует, что это исключение не произойдет, то мы можем легко использовать упомянутое выше исключение `Impossible`. Ситуация представлена в виде псевдокода, поскольку TypeScript, к сожалению, не предоставляет проверенных исключений.

Листинг 7.30. Использование непроверенного исключения

```
class Impossible extends RuntimeException { }
class EmptyArray extends CheckedException { }
function average(arr: number[]) throws EmptyArray {
  if (arr.length === 0) throw new EmptyArray();
  return sum(arr) / arr.length;
}

/// ...
try {
  console.log(average(arr));
} catch (EmptyArray e) {
  throw new Impossible();
}
```

Архитектура

Третий способ, которым люди не позволяют компилятору приносить пользу, связан с недостаточным пониманием архитектуры, в частности микроархитектуры. *Микроархитектура* — это архитектура, которая относится только к данной команде, но не к другим.

Основную причину этого мы обсудили в части I: это нарушение инкапсуляции геттерами и сеттерами. Подобные действия создают связанность между получателем и полем, лишая компилятор возможности контроля доступа.

В нижеприведенной реализации стека мы нарушаем инкапсуляцию, раскрывая внутренний массив. Это означает, что от него может зависеть внешний код. Что еще хуже, внешний код может изменять стек через изменение массива.

Листинг 7.31. Плохая микроархитектура с геттером

```
class Stack<T> {
    private data: T[];
    getArray() { return this.data; }
}
stack.getArray()[0] = newBottomElement; ← Эта строка
                                         изменяет стек
```

Иначе это еще может произойти при передаче закрытого поля в качестве аргумента, что вызовет аналогичный эффект. В данном примере метод, получающий массив, может делать с ним что угодно, включая изменение стека. Не обращайте внимания на странное имя функции.

Листинг 7.32. Плохая микроархитектура с параметром

```
class Stack<T> {
    private data: T[];
    printLast() { printFirst(this.data); }
}
function printFirst<T>(arr: T[]) {
    arr[0] = newBottomElement; ← Эта строка
                               изменяет стек
}
```

Вместо этого следует передать `this`, что позволит сохранить инварианты локальными.

7.3. ДОВЕРИЕ К КОМПИЛЯТОРУ

Теперь мы умеем активно использовать компилятор и создавать программы с учетом его участия. Обладая знаниями о его сильных и слабых сторонах, мы редко вступаем с компилятором в спор и уже можем начать ему доверять.

Можно отказаться от контрпродуктивного чувства, что мы знаем что-то лучше компилятора, и пора начать больше прислушиваться к его мнению. Взамен мы получим отдачу большую, чем вложение. А судя по последнему разделу, вкладываем мы в него многое.

В завершение разберем два последних тонких момента, в работе с которыми люди склонны не доверять компилятору: инварианты и предупреждения.

7.3.1. Учим компилятор инвариантам

Зловредность глобальных инвариантов на протяжении книги подробно обсуждалась, так что сейчас они должны быть под вашим чутким контролем. А что насчет локальных инвариантов?

Их проще поддерживать, поскольку такие инварианты ограничены и прозрачны для понимания. Однако они провоцируют все те же конфликты с компилятором, ведь разработчики знают о своей программе то, чего не знает он.

Рассмотрим более обширный пример, где все это будет наглядно. Здесь мы создаем структуру данных для подсчета элементов. В результате при добавлении элементов эта структура отслеживает, сколько элементов каждого типа было добавлено. Для удобства код также отслеживает общее число добавленных элементов.

Листинг 7.33. Счетный механизм

```
class CountingSet {
  private data: StringMap<number> = { };
  private total = 0;
  add(element: string) {
    let c = this.data.get(element);
    if (c === undefined)
      c = 0;
    this.data.put(element, c + 1);
    this.total++;
  }
}
```

Отслеживание
общего числа

Далее появляется намерение добавить метод для выбора из этой структуры случайного элемента. Для этого можно выбирать случайное число величиной меньше общего количества и возвращать элемент, который бы находился в этой позиции, если бы перед нами был массив. Поскольку массив мы не храним, то вместо этого нужно перебирать ключи и перескакивать вперед на заданное число индексов к нужному элементу.

Листинг 7.34. Выбор случайного элемента (ошибка)

```
class CountingSet {
  // ...
  randomElement(): string {
    let index = randomInt(this.total);
    for (let key in this.data.keys()) {
      index -= this.data[key];
      if (index <= 0)
        return key;
    }
  }
}
```

Ошибка из-за достижимости

Этот метод не проходит компиляцию успешно, поскольку анализ достижимости, описанный ранее, сигнализирует об ошибке. Компилятор не знает, что мы будем всегда выбирать успешно элемент из перечня, так как ему неизвестен инвариант, что `total` — это количество элементов в структуре данных, и программа не выйдет за его рамки. Это локальный инвариант, остающийся верным при завершении каждого метода в этом классе.

В данном случае можно исправить ошибку, добавив невозможное исключение.

Листинг 7.35. Выбор случайного элемента (исправлено)

```
class Impossible { }
class CountingSet {
  // ...
  randomElement(): string {
    let index = randomInt(this.total);
    for (let key in this.data.keys()) {
      index -= this.data[key];
      if (index <= 0)
        return key;
    }
    throw new Impossible(); ← Исключение
                              для избежания ошибки
  }
}
```

Тем не менее так мы решаем проблему только здесь и сейчас, поскольку не добавили никаких мер безопасности, гарантирующих, что этот инвариант не окажется нарушен позднее. Представьте, что сами будете реализовывать функцию `remove` и забудете уменьшить `total`. Так что компилятор обоснованно недолюбливает наш метод `randomElement`, потому что он опасен.

Всякий раз, когда в программе есть инварианты, следуем адаптированной вариации известной тактики «Если не можешь победить, присоединяйся». Ее развернутая форма относительно инвариантов такова.

1. Удаляем их.
2. Если не можем, то рассказываем о них компилятору.
3. Если не можем, знакомим с ними среду выполнения через автоматизированный тест.
4. Если не можем, рассказываем о них команде разработчиков через подробную документацию.
5. Если не можем, рассказываем о них тестировщику и тестируем их вручную.
6. Если и это не получается, то начинаем молиться высшим силам, потому что больше никакая земная нам не поможет.

В данном контексте «не можем» означает больше непрактичность действия, нежели его фактическую невозможность. Для каждого из этих решений требуется свой промежуток времени. Однако имейте в виду: чем ниже мы спускаемся по этому списку, тем дольше нам придется впоследствии тратить времени на поддержание и обслуживание кода. Документация требует более тщательного и длительного обслуживания, чем тесты, потому что тесты сообщают вам, когда выходят из синхронизации с ПО, а документация такой любезности не оказывает. Чем выше решение по списку, тем оно менее затратно в длительной

перспективе. Это должно исключать очень частую отговорку, что у нас нет времени на написание тестов, ведь по факту без них в перспективе мы затратим гораздо больше времени.

Имейте в виду, что если вы изначально создаете короткоживущее ПО, то можете позволить себе выбрать вариант из конца списка: например, когда создаете прототип, который будет выброшен сразу после ручного тестирования.

7.3.2. Обращайте внимание на предупреждения

Еще одна область, в которой люди склонны сомневаться, — это сообщения и предупреждения компилятора. В больничной среде существует понятие *«усталость от сигналов тревоги»*: когда медицинские работники утрачивают чувствительность к внешним шумам, потому что они становятся нормой, а не исключением. Тот же эффект возникает и в мире ПО: чем чаще мы игнорируем предупреждения, ошибки среды выполнения или баги, тем меньше внимания уделяем им в дальнейшем. Еще одна аналогия для усталости от сигналов тревоги — это теория разбитых окон, согласно которой если что-то находится в нетронутом состоянии, то люди склонны так это и оставлять и не уделять должного внимания. Но и тогда, когда с этим объектом происходит нечто плохое (например, разбивается окно у заброшенного дома), у людей проявляется склонность продолжить эту тенденцию и ничего не предпринимать, хотя, согласно теории и практике, неприятности будут множиться и позже это может вызвать крупные разрушения.

Даже если некоторые предупреждения оказываются неоправданными, игнорировать их нельзя. Опасность заключается в том, что мы можем упустить существенное просто из-за того, что оно затеряется среди относительно маловажных сообщений. Эту опасность необходимо четко осознавать. Несущественные ошибки или предупреждения могут завуалированно сигнализировать о довольно серьезных ситуациях.

Дело в том, что у предупреждений компилятора есть цель: помочь нам совершать меньше ошибок. Следовательно, допустимым и здоровым можно считать лишь их нулевое количество. В некоторых базах кода это кажется невозможным, потому что предупреждения накапливались и распространялись по ним бесконтрольно и без ограничений слишком долго. В таких ситуациях мы устанавливаем для их количества верхнюю границу числа допустимых и затем постепенно месяц за месяцем снижаем ее. Это утомительная задача, особенно из-за того, что мы не ощутим никаких существенных выгод, пока количество предупреждений не сойдет к минимуму. Как только они полностью исчезнут, следует принять за правило придерживаться конфигурации языка, не допускающей игнорирования предупреждений, и тем более не допускать роста их количества до прежнего устрашающе огромного числа.

7.4. ИСКЛЮЧИТЕЛЬНОЕ ДОВЕРИЕ К КОМПИЛЯТОРУ

А это работает?

Каждый программист

Красивый итог длинного пути — получившаяся в результате наших усилий чистая база кода. А все потому, что мы научились прислушиваться к рекомендациям компилятора и проектировать программу с их учетом. И более того: освоив сильные и слабые стороны компилирования, вместо основного упора на собственное рассуждение мы стали полностью полагаться на суждение компилятора. Вместо постоянных собственных мучительных сомнений, сработает ли то или иное наше действие в базе кода, предпочитаем просто спросить об этом компилятор.

Если к тому же мы обучили его структуре нашей области работы, закодировали инварианты и привыкли дорабатывать программу до того, что никаких предупреждений во время компилирования не возникает, если полностью доверились этому, то успешная компиляция должна внушать нам большую уверенность, чем мы могли получить от простого чтения кода. Конечно же, компилятор не может знать, решает ли наш код возлагаемую на него целевую задачу, но он способен сообщить нам, может ли программа дать сбой, чего мы точно никогда в нее не закладываем специально, но можем получить, совершив ошибки при разработке кода.

Большая уверенность от компилирования программы, чем от простого самостоятельного чтения кода, не вырабатывается в одночасье. Для этого нужны усердная практика и дисциплина, а также подходящие технологии (имеется в виду язык программирования). Цитата ниже подразумевает в том числе и отношения между разработчиком кода и компилятором.

Если вы являетесь умнейшим человеком в комнате, значит, вы находитесь не в той комнате.

Автор неизвестен

РЕЗЮМЕ

- Помните типичные сильные и слабые стороны современных компиляторов. Мы можем подстраивать код так, чтобы избегать слабых и полагаться на сильные, поставив компилятор себе на службу.
 - Используйте достижимость, чтобы обеспечить покрытие инструкцией `switch` всех кейсов.

- Используйте явное присваивание, чтобы гарантировать наличие у переменных значений.
- Используйте контроль доступа для защиты методов с чувствительными инвариантами.
- Убеждайтесь в том, что переменные не `null`, прежде чем разыменовывать их.
- Убеждайтесь, что числа не являются нулями, прежде чем делить на них.
- Проверяйте, чтобы математические операции не вызывали переполнения или антипереполнения, либо используйте `BigIntegers`.
- Избегайте ошибок выхода из допустимого диапазона путем перебора всей структуры данных или через использование явного присваивания.
- Избегайте бесконечных циклов за счет использования высокоуровневых конструкций.
- Избегайте проблем с многопоточностью, не позволяя нескольким потокам обрабатывать общие изменяемые данные.
- Научитесь эффективно использовать компилятор, а не сопротивляться ему. Это позволит добиться более высокой степени безопасности.
 - Используйте ошибки компилирования в качестве списка точек для внесения исправлений в программу при рефакторинге.
 - Используйте компилятор, чтобы обеспечить инварианты последовательности.
 - Используйте компилятор для обнаружения неиспользуемого кода.
 - Не применяйте приведение типов, а также динамические типы или типы среды выполнения.
 - Не используйте предустановленные (`default`) значения, наследование от классов или непроверенные исключения.
 - Передавайте `this` вместо закрытых (`private`) полей, чтобы избежать нарушения инкапсуляции.
- Доверяйте компилятору, цените его листинги и предупреждения, избегайте появления усталости от непомерно большого числа сигналов тревоги, подерживая базу кода в чистом виде.
- Полагайтесь на мнение компилятора о работоспособности кода.

Избегайте комментариев

В этой главе

- ✓ Опасность комментариев.
- ✓ Определение стоящих комментариев.
- ✓ Работа с разными типами комментариев.

Комментарии — это, вероятно, одна из наиболее противоречивых тем книги, поэтому для начала стоит уточнить: эта глава рассматривает комментарии, которые находятся внутри методов и *не* используются внешними инструментами, такими как Javadoc:

```
interface Color {  
    /**  
     * Метод для преобразования цвета в hex-строку.  
     * возвращает 6-цифренное hex-число с хештегом в начале  
     */  
    toHex(): string;  
}
```

Мое мнение: комментарии — это форма искусства. И хотя не все с ним согласятся, оно почти в точности совпадает с мнением многих видных программистов. К сожалению, не так много разработчиков умеют правильно комментировать свои программы, что в конечном итоге снижает ценность создаваемого ими кода. Лучше вообще обойтись без комментариев, чем портить код. Отсюда и происходит общее правило, согласно которому я рекомендую их избегать. Роб Пайк

высказал аналогичные аргументы в серии своих эссе *Notes on Programming in C*, вышедшей еще в 1989 году.

[Комментарии — это] деликатный случай, требующий вкуса и грамотного рассуждения. Я же склоняюсь к тому, чтобы их исключать, и на то вижу несколько причин. Во-первых, если код вполне ясен и задействует грамотные имена типов и переменных, то он должен сам себя объяснять. Во-вторых, комментарии не проверяются компилятором, поэтому нет гарантии, что они всегда будут корректны, особенно после изменения кода. Сбивающие с толку комментарии могут реально вводить в заблуждение. Третья же причина заключается в типографике, так как комментарии загромождают код.

Роб Пайк

Мартин Фаулер дополняет это мнение, причисляя комментарии к запахам. Один из его аргументов в том, что они зачастую используются подобно дезодоранту для сильно пахнущего кода. В таких случаях следует не прятать запахи за комментариями, а просто почистить сам код.

Многие преподаватели требуют от студентов объяснения их кода через комментарии, чтобы учащиеся приобретали навыки писать их правильно с самого начала. Это подобно включению промежуточных вычислений в процесс присваивания: хорошо для обучения, но не так полезно в реальной практике. Перенос идеи комментирования кода в будущие реальные проекты вызывает сложности. Как поделился в своем твите Кевлин Хенни, проблемы с труднопонимаемым кодом вряд ли удастся решить добавлением комментариев от того же разработчика.

Распространенным заблуждением является предположение, что авторы невразумительного кода каким-то образом смогут выразить свои мысли отчетливо и понятно через комментарии.

Кевлин Хенни

Комментарии не проверяются компилятором, поэтому писать их проще, чем код, ведь комментатора ничто не ограничивает. Тем не менее именно из-за того, что компилятор о них не знает, в системах с продолжительным жизненным циклом они имеют тенденцию устаревать или, что еще хуже, начинают вводить в заблуждение.

Для комментариев есть множество вариантов применения, включая планирование работы, указание на «хаки», документирование кода и его удаление. В своей

книге «Чистый код» Мартин Фаулер называет около 20 типов комментариев. Отслеживать такое количество категорий может оказаться слишком сложно, поэтому в книге комментарии подразделяются на пять видов, каждый из которых подразумевает определенный способ его использования.

В большинстве случаев нам следует избегать их применения в отправляемом коде. При этом до отправки на промежуточных стадиях создания и отладки кода комментарии вполне приветствуются. Следовательно, с ними нужно разбираться на стадии рефакторинга. Прежде чем отправлять какой-либо комментарий, нужно всегда подумать, есть ли более удачный способ выразить его смысл. Я бы хотел создать правило, которое бы вообще их запрещало. Но в некоторых случаях комментарий может спасти нас от серьезных ошибок и оказывается стоящим. Есть свойства, которые бывает сложно или дорого передавать через код, но гораздо проще выразить в комментарии буквально за секунды. Такой подход к комментариям вторит идее Кевлина Хенни (<https://medium.com/@kevinhenney/comment-only-what-the-code-cannot-say-dfdb7b8595ac>).

Комментировать нужно только то, чего не может сказать код.

Кевлин Хенни

Ниже перечисляются те самые пять категорий комментариев в порядке возрастания их сложности.

8.1. УДАЛЕНИЕ УСТАРЕВШИХ КОММЕНТАРИЕВ

Здесь мы не скупимся на строгость формулировки, поскольку эта категория также включает, кроме просто старых, необходимость удаления откровенно ошибочных или вводящих в заблуждение комментариев. Делаем мы это с тем основанием, что такие комментарии, хотя и были грамотными и осмысленными, когда их писали, с течением времени утратили согласованность с базой кода.

Заметьте, как в данном примере комментарий и условие расходятся в выражении «или/и», что является довольно опасным.

Листинг 8.1. Устаревший комментарий

```
if (element.hasSelection() || element.isMultiSelect()) {  
    // Is содержит выбор и позволяет множественный выбор  
    // ...  
}
```

Проще всего разобраться именно с устаревшими комментариями, которые либо утратили значимость, либо стали ошибочными. Такие комментарии не экономят наше время работы с кодом, но однозначно отвлекают внимание: мы должны их как минимум прочитать. Поэтому лучше удалять их без колебаний.

Более пагубный эффект в том, как я уже говорил, что старые комментарии могут вводить нас в заблуждение. Ведь в таком случае мы уже не просто тратим время на их чтение, но и подвергаем себя риску проделать огромную работу с кодом, исходя из неверных предпосылок, ошибочного понимания. И наконец, самая плохая перспектива состоит в том, что из-за устаревших комментариев мы сами можем допустить ошибки в перерабатываемом коде.

8.2. УДАЛЕНИЕ ЗАКОММЕНТИРОВАННОГО КОДА

Иногда мы экспериментируем с удалением фрагментов кода — можно легко и быстро закомментировать такие фрагменты и посмотреть, к чему это приведет. Это хороший способ экспериментирования, но после такого приема весь закомментированный код обязательно нужно удалять. Тем более что, поскольку наш код находится в системе контроля версий, его можно легко восстановить даже после удаления.

В примере ниже несложно увидеть, почему здесь добавлены комментарии: первые наброски кода были рабочими, но еще не оптимальными. Разработчик подумал, что может сделать лучше, но не был уверен — это можно понять, поскольку здесь непростой алгоритм, — и при этом не воспользовался возможностями системы контроля версий либо из-за недостатка опыта, либо из-за дороговизны ветвления. Следовательно, вместо удаления старого алгоритма разработчик просто закомментировал его, чтобы можно было быстро вернуться к изначальному варианту, если новый алгоритм не сработает. Чтобы проверить работоспособность нового кода, разработчику могло потребоваться выполнить его слияние с основной веткой. Когда же проверка прошла успешно, код уже находился там и на упорядочение работающего фрагмента не хватило то ли времени, то ли желания.

Листинг 8.2. Закомментированный код

```
const PHI = (1 + Math.sqrt(5)) / 2;  
const PHI_ = (1 - Math.sqrt(5)) / 2;  
const C = 1 / Math.sqrt(5);  
function fib(n: number) {  
  // if(n <= 1) return n;  
  // else return fib(n-1) + fib(n-2);  
  return C * (Math.pow(PHI, n) - Math.pow(PHI_, n));  
}
```

Правильный сценарий должен был разыгрываться следующим образом. Разработчик создает ветку в Git, удаляет старый код и начинает работать с новым. Если в итоге оказывается, что новый код не работает, разработчик переключается на основную ветку и удаляет экспериментальную. Если же все работает, далее он выполняет слияние с основной веткой, и все остается чистым. Этой процедуре надо следовать даже при необходимости предварительного слияния нового кода с основной веткой. Тогда, если код вдруг не заработает, можно просто восстановить его изначальный вариант из истории версий.

8.3. УДАЛЕНИЕ БЕССМЫСЛЕННЫХ КОММЕНТАРИЕВ

Еще одна категория — это комментарии, которые ничего не дают. Когда код так же легко прочесть, как и комментарий, и он так же информативен, такой комментарий можно посчитать *бессмысленным*.

Листинг 8.3. Бессмысленный комментарий

```
/// Выводит ошибку
Logger.error(errorMessage, e);
```

В эту категорию также входят комментарии, которые мы игнорируем при сканировании кода. Если никто никогда не читает комментарий, то он только занимает пространство и от него можно спокойно избавиться.

8.4. ПРЕОБРАЗОВАНИЕ КОММЕНТАРИЕВ В ИМЕНА МЕТОДОВ

Некоторые комментарии не добавляют функциональности, а документируют код. Проще всего показать это на примере.

Листинг 8.4. Комментарий, документирующий код

```
/// Создание url запроса
if (queryString)
    fullUrl += "?" + queryString;
```

В таких случаях можно просто извлечь этот блок в метод с таким же именем, что и комментарий. Как видно в данном примере, после этой операции комментарий становится бессмысленным, и мы поступаем с ним соответственно, то есть удаляем. Такую технику мы уже дважды применяли в главе 3.

Листинг 8.5. До

```
/// Создаем url запроса
if (queryString)
    fullUrl += "?" + queryString;
```

Листинг 8.6. После

```
/// Создаем url запроса
fullUrl = buildRequestUrl(
    fullUrl, queryString);
/// ...
function buildRequestUrl(
    fullUrl: string, queryString: string)
{
    if (queryString)
        fullUrl += "?" + queryString;
    return fullUrl;
}
```

Теперь комментарий бессмысленен

Люди, как правило, не любят такие длинные имена методов. Но фактически это является проблемой только для часто вызываемых методов. У языков есть свойство: наиболее используемые нами слова чаще всего являются короткими.

То же должно касаться и базы кода: в часто вызываемых методах следует употреблять короткие осмысленные наименования. Это тем более верно, что для постоянно используемых элементов нам реже требуется пояснение.

8.4.1. Использование комментариев для планирования

Такие комментарии чаще всего возникают, когда мы используем их для планирования работы и разделения большой задачи. Это отличный способ создания дорожной карты. Лично я всегда планирую код с помощью комментариев примерно так.

Листинг 8.7. Комментарии в планировании

```
/// Запрос данных
/// Проверка чего-то
/// Преобразование
/// В противном случае
/// Отправка
```

Некоторые из этих комментариев обычно после реализации кода утрачивают смысл, например *В противном случае*. Другие же преобразуются в методы. Что именно с ними делать, зависит от предпочтения разработчика. Важно просто отметить, что после написания кода мы должны критически оценить, насколько они важны и стоит ли их оставлять.

8.5. СОХРАНЕНИЕ КОММЕНТАРИЕВ К ИНВАРИАНТАМ

Заключительный вид комментариев — это те, которые документируют нелокальный инвариант. Как уже неоднократно говорилось, именно здесь могут возникать ошибки. В качестве способа их обнаружения можно просто спрашивать: «Предостережет ли этот комментарий разработчика от внесения ошибки в код?»

Когда мы встречаем такие комментарии, несмотря на их «нелокальность», при первом подходе к их оценке следует проверять, можно ли их встроить в код. В некоторых случаях оказывается возможно удалить эти комментарии с помощью компилятора, как это описывалось в главе 7. Но так бывает редко, поэтому при следующем подходе нужно подумать, можно ли создать автоматизированный тест для проверки этого инварианта. Если оба подхода не принесут успешного результата, то комментарий нужно сохранить.

В следующем примере показана подозрительная инструкция, `session.logout`, сопровождаемая комментарием, который объясняет ее причину. Аутентификация или аналогичные замысловатые взаимодействия могут оказаться ужасно сложными для тестирования или моделирования, и тогда комментарий оказывается полностью оправдан.

Листинг 8.8. Комментарий документирует инвариант

```
/// Выход из сеанса производится для обязательной  
/// аутентификации при очередном запросе  
session.logout();
```

8.5.1. Инварианты в процессе

Нечто невыполненное (нужно сделать) или, быть может, ошибочное (исправить), или обходной путь стороннего ПО (хак) являются инвариантами: не инвариантами в коде, но инвариантами процесса. Некоторые разработчики пренебрегают ими и справедливо утверждают, что они должны находиться не в коде, а в системе тикетов. Я согласен с этим утверждением, хотя сам предпочитаю локальное использование комментариев прямо в коде. Тем не менее, если они находятся в коде, необходимо визуально обозначить, сколько их в нем, причем это количество должно постепенно сокращаться. Следует стремиться исправить или выполнить то, о чем сообщает комментарий, чтобы можно было его удалить, а не откладывать отмеченное им действие на потом.

Лучшее время для посадки дерева было двадцать лет назад. Второе лучшее время сейчас.

Китайская пословица

РЕЗЮМЕ

- Комментарии могут быть полезны в процессе разработки, но перед отправкой кода нужно постараться их удалить.
- Существует пять типов комментариев:
 - устаревшие, которые нужно удалить, так как они могут привести к ошибкам понимания кода;
 - закомментированный код, который должен быть удален, поскольку он уже находится в системе контроля версий;
 - пустые (бессмысленные) комментарии, которые нужно удалять, потому что они не повышают читаемость;
 - комментарии, которые можно преобразовать в имена методов, должны стать именами методов;
 - комментарии, которые документируют нелокальный инвариант, нужно преобразовывать в код или автоматизированный тест. Если это невозможно, то их можно оставить.



Стрась к удалению кода

В этой главе

- ✓ Как код замедляет разработку.
- ✓ Установление границ, чтобы исключить ненужные затраты.
- ✓ Обработка переходов с помощью шаблона «Фигус-душитель».
- ✓ Минимизация затрат с помощью шаблона «Отрыв и стабилизация».
- ✓ Удаление всего ненужного для обеспечения полноты функциональности кода.

Все системы создаются для осуществления определенной функциональности. Сама же функциональность напрямую определяется кодом, поэтому может создаться впечатление, что код сам по себе является невяной ценностью. Но это иллюзия, забудьте о ней. Код представляет собой необходимость. Это неизбежное зло, с которым нам приходится мириться, чтобы система выполняла свои функции.

Еще одна причина, по которой мы склонны воспринимать код как нечто ценное, состоит в том, что его сложно создавать. Написание кода требует привлечения опытных разработчиков, которые затратят уйму времени (и при этом проглотят тонны кофеина). Присвоение ценности чему-то из-за затраченных на его создание усилий называется *учетом невозвратных затрат*. Ценность никогда

не складывается из одних только вложений, она определяется в первую очередь величиной отдачи от этих вложений. Работая с кодом, это необходимо понимать, так как нам приходится непрерывно вкладывать усилия не только в его разработку, но и в его поддержание, независимо от представляемой им ценности.

Каждый программист, утомившись от ручного выполнения рутинных задач, неоднократно приходил к необходимости их автоматизации. Во многом именно поэтому мы и стали программистами. Тем не менее очень легко утратить бдительность из-за автоматизации разработки кода, которая отвлекает от решения изначальной задачи, и начать больше времени проводить за автоматизацией разработки, чем затратили бы на ручное программирование.

Писать код интересно, к тому же это развивает как нашу креативность, так и навыки решения задач. Но сам код в перспективе представляет собой определенную статью расходов, достаточно вспомнить все то время, которое необходимо для его поддержания. Чтобы удовлетворить творческий зуд и в то же время получить работающую систему, на протяжении всей карьеры мы тренируемся: выполняем задания, совершаем рывки и экспериментируем с кодом, после чего подчас сразу его удаляем.

В 1998 году Кристофер Хси провел исследование, которое назвал *Less is Better: When Low-Value Options Are Valued More Highly than High-Value Options* (Journal of Behavioral Decision Making, vol. 11, pp. 107–121, Dec. 1998, <http://mng.bz/l2Do>). В этом исследовании он установил ценность обеденного сервиза из 24 предметов. Потом он добавил в набор несколько разбитых предметов и выяснил, что при этом общая ценность сервиза снизилась. Итак, размер сервиза был увеличен, но ценность его уменьшилась. А все потому, что появились «слабые, дефектные» места в системе.

В наших системах нужен долгосрочный код. Естественно, его объем от случая к случаю варьируется, это зависит от сложности предметной области, для которой он реализуется. Но при работе с кодом любой длины самое важное — не пренебрегать его простотой и лаконичностью: чем меньше, тем лучше. И это основной посыл данной главы.

В ней мы сначала рассмотрим то, как можно столкнуться с проблемами в коде из-за технического неведения, потерь, долга или задержек. После этого разберем несколько видов кода, которые вызывают задержку в разработке, например ветвление в системе контроля версий, необходимость составления и поддержки в актуальном состоянии документации и множественность разных возможностей, запрограммированных в коде. Затем обсудим, каким образом можно обойти подобные задержки или же вовсе от них избавиться.

9.1. УДАЛЕНИЕ КОДА МОЖЕТ СТАТЬ ОЧЕРЕДНОЙ ВЕХОЙ

Сфера программирования пережила множество фаз развития, и, чтобы спрогнозировать перспективы, нам нужно оглянуться на то, что мы уже прошли. Однако перебор всех изобретений и людей, которые привели нас к текущей ситуации, окажется чересчур обширным. Вместо этого я выстроил краткую хронологию, на мой взгляд, самых существенных прорывов, которые совершила информатика.

- 1944 — компьютеры использовались для выполнения вычислений без каких-либо абстракций.
- 1952 — Грейс Хоппер изобрела первый компоновщик, позволивший компьютерам работать с символами, а не с чистыми вычислениями.
- 1957 — предыдущий прорыв привел к изобретению компилятора, в частности Fortran. Теперь появилась возможность писать код, используя языки высокого уровня с операторами управления вроде циклов.
- 1972 — очередной важной проблемой, которую удалось решить, стали абстракции данных. Здесь появляется новое поколение языков, таких как Си — а позже C++ и Java, — они работают с данными не напрямую, а через указатели и ссылки.
- 1994 — очередной существенный скачок совершила «Банда четырех» (Эрих Гамма, Ричард Хельм, Ральф Джонсон, Джон Влиссидес), они создали набор паттернов проектирования с возможностью многократного использования. В процессе проектирования ПО эти паттерны выступают в качестве высокоуровневых строительных блоков.
- 1999 — Мартин Фаулер составил каталог стандартных шаблонов рефакторинга. В отличие от шаблонов проектирования они не требовали предварительного выстраивания, а позволяли улучшать существующий код.
- 2011 — последним важным прорывом в мире программирования, по моему мнению, стала архитектура микросервисов, популяризованная Сэмом Ньюманом. Эта архитектура, хотя и основана на давно известном принципе слабой связанности, при этом решает современную проблему масштабируемости. Она также позволяет создавать новые архитектурные версии посредством косвенного взаимодействия. Таким образом, можно совершенствовать дизайн уже работающих систем.

Теперь мы стали профессионалами в написании кода и построении систем. При этом производимые системы могут быть настолько сложными, что полностью

их не сможет охватить и понять ни один человек. Это создает проблемы при удалении каких-либо компонентов, так как для выяснения возможности удаления чего-то сначала нужно затратить время, чтобы понять, какой именно код выполняется, как часто и в каких версиях. В удалении кода мы еще не достигли высот, и я считаю, что это могло бы стать очередной важной вехой в развитии программирования.

9.2. УДАЛЕНИЕ КОДА ДЛЯ УСТРАНЕНИЯ НЕНУЖНОЙ СЛОЖНОСТИ

Это очень характерно для сложных программных систем — естественно разрастаться по мере добавления в них новых возможностей, экспериментирования и обработки все большего числа граничных случаев. Когда мы что-то реализуем, нам нужно создать в уме модель поведения системы, после чего внести изменения, чтобы на нее повлиять. Чем больше база кода, тем сложнее модель, так как появляется больше связей и отслеживается более обширная библиотека утилит.

Эта сложность встречается в двух видах: сложность области и ненужная сложность. *Сложность области* обусловлена прикладной областью, в которой мы работаем. То есть решаемая нами задача является по определению сложной. Например, система для вычисления налогов будет сложной, что бы мы ни делали, потому что сложен сам закон о налогах. *Ненужная сложность* — это любая сложность, которая не обусловлена прикладной областью задачи, а оказывается привнесена случайно.

Ненужная сложность обычно используется в качестве синонима для *технического долга*. Но я считаю, что лучше применять более точные термины. В своей практике мне встречались четыре типа ненужной сложности, каждый со своей причиной и со своим способом удаления: техническое неведение, технические потери, технический долг и техническая задержка. Разберем каждый по очереди.

9.2.1. Техническое неведение ввиду неопытности

Самый простой тип ненужной сложности — это *техническое неведение*. Он возникает вследствие принятия по незнанию ошибочных решений при создании ПО, от чего в результате страдает архитектура. Это происходит, когда нам не хватает навыков для решения задачи без внесения ненужной связанности либо из-за

того, что мы не знаем, не хотим что-то знать или не имеем времени на изучение этого. К счастью, данная книга призвана облегчить вам жизнь. Единственным состоятельным решением здесь может послужить первая половина одного из принципов манифеста методики гибкой разработки ПО.

Непрерывное внимание к техническому совершенству и хорошему дизайну повышает гибкость.

*Манифест методики
гибкой разработки*

Каждому необходимо непрерывно стремиться совершенствоваться в ремесле создания программ для сложных систем, читать книги и статьи в блогах, просматривать конференции и видеоуроки, делаясь полученными знаниями через коллективное программирование и, что самое главное, обдуманно практикуя — ибо ничто не заменит практику.

КОЛЛЕКТИВНОЕ ПРОГРАММИРОВАНИЕ

В некоторых ситуациях требуется дополнительный когнитивный заряд, например, когда мы сталкиваемся со сложной задачей, багом, который нужно срочно исправить, или когда мы просто учимся. Такой заряд можно получить через практику коллективного программирования.

Ллевеллин Фалько однажды прекрасно сформулировал основной принцип коллективного программирования: «Любая идея, прежде чем попасть в код, должна пройти еще через чей-то мозг» (*Llewellyn's strong-style pairing*, пост в блоге от 30 июня 2014 года). На практике это означает, что человек за клавиатурой не должен делать что-то сверх или помимо того, о чем договорились два совместно работающих программиста. К примерам можно отнести *парное программирование* — когда это делают два человека — и *ансамблевое программирование* (иногда называемое *моб-программированием*), когда в процессе участвует уже больше людей.

Коллективное программирование ведет к непосредственному обмену знаниями. Оно раскрывает все возможные микрозатраты и высвобождает когнитивный потенциал человека, мастерство которого еще нужно наращивать, увеличивая эффективность его обучения. Оно также обычно ведет к повышению качества, так как код проходит проверку в реальном времени или даже синхронно. Кроме того, коллективное программирование исключает необходимость дополнительной асинхронной оценки кода, делая процесс его доставки заказчику более экономичным.

9.2.2. Технические потери из-за нехватки времени

Итак, простейшим видом ненужной сложности являются *технические потери*. Они тоже возникают ввиду принятия при написании кода неудачных решений, которые вредят архитектуре.

Чаще всего такие потери обусловлены нехваткой времени. Иногда программисты недостаточно хорошо понимают задачу или модель и оказываются слишком заняты, чтобы с этим разобраться. В других случаях они пропускают тестирование или рефакторинг, опять же оправдываясь дефицитом времени. А иногда какой-либо процесс оказывается просто сознательно пропущенным из-за навигирующегося дедлайна.

Эти плохие решения принимаются намеренно. Ведь во всех случаях именно разработчик решает пожертвовать лучшим пониманием вопроса, пусть даже под давлением внешних обстоятельств. Это, по сути, сродни саботажу.

РЕАЛЬНАЯ ИСТОРИЯ

Одно время я работал техническим лидером на проекте, в котором мы постепенно применяли ряд практик, чтобы избежать повторения ошибок прошлого. Сроки отправки очередной части проекта сильно поджимали, поэтому я спросил одного из разработчиков, может ли функция X быть готова к завтрашнему дню. Он ответил: «Да, если я пропущу тестирование». Прикусив язык, я сказал ему, что «готова» означает следование всем нашим практикам, включая обязательное тестирование.

В качестве выхода из положения нужно обучать разработчиков тому, что пропускать самые нужные этапы и практики нельзя ни под каким предлогом. Нужно приучить проектных менеджеров, клиентов и других участников процесса, что ПО жизненно важно создавать именно правильно. Я для этого периодически задаю своим специалистам вопрос из разряда: «Хотели бы вы получить новую машину на три недели раньше, если бы это означало пропуск тестирования тормозов или подушек безопасности?» В некоторых сферах существуют процедуры и регламенты, жестко задающие порядок действий. У разработчиков тоже есть набор практик, которых необходимо придерживаться, даже находясь под давлением людей и обстоятельств.

9.2.3. Технический долг под давлением обстоятельств

В то время как и техническое неведение, и потери можно и нужно устранять, технический долг уже более тонкая категория. *Технический долг* — это когда мы временно отдаем предпочтение субоптимальному решению ради определенной выгоды. Такое решение тоже оказывается намеренным, как и предыдущее, но

здесь ключевое слово «*временно*». Если выбранное нами решение не является временным, то оно не является долгом, а это уже потери.

К примеру, это тот случай, когда мы реализуем быстрый патч без учета правильности архитектуры и отправляем его для исправления критической проблемы — после этого нам нужно начать все заново и реализовать уже выверенный постоянный патч, отвечающий всем стандартам и вписывающийся в окружение. Здесь я хочу подчеркнуть, что формирование технического долга является стратегическим решением и, по существу, не является преступным при условии, что для него устанавливается срок годности.

9.2.4. Техническая задержка из-за роста

Заключительный тип ненужной сложности самый неоднозначный. *Техническая задержка* — это все, что замедляет разработку. Сюда относятся все прочие категории, помимо описанных в предыдущих пунктах, вместе с документацией, тестами, да и вообще всем кодом.

Автоматизированные тесты усложняют (намеренно) изменение кода, поскольку при этом приходится менять и сами тесты. Это не обязательно плохо, например, в критических системах, где предпочтение обычно отдается размеренности и стабильности, а не скорости. Но гораздо меньше это подходит к ситуациям, которые опираются на высокий уровень экспериментирования над кодом, например, во время авралов.

Документация замедляет нас, потому что при изменении чего-либо ее приходится также обновлять. Даже сам код представляет техническую задержку, потому что нам необходимо учитывать влияние его изменений на все приложение, а также тратить время на его обслуживание.

Техническая задержка является побочным эффектом создания чего-либо. Сама по себе она не плоха, но негативно сказывается в ситуациях, когда приходится поддерживать редко используемую документацию, функционал или код. В таких случаях может оказаться экономичнее удалить подобный компонент, чтобы исключить задержку.

РЕАЛЬНАЯ ИСТОРИЯ

Как-то раз, когда я работал над одним из проектов в качестве разработчика, меня попросили создать специфичную подсистему. Я это сделал, но по завершении клиент оказался не готов ее задействовать. Техлид дал мне указание оставить ее в системе, чтобы она дождалась готовности клиента. С того дня во всем, что мы создавали, нам приходилось учитывать то, как новый код отреагирует, *если* клиент внезапно начнет использовать эту отложенную подсистему. В итоге он, конечно же, так ее и не использовал.

В данном случае избитый аргумент — «Ничего страшного не произойдет, если этот компонент просто здесь оставить» — оказывается ошибочен. Решением будет удалить столько, сколько возможно без нарушения функционала, и ничто другое. Все, что не оправдывает себя, нужно исключать, даже если оно немного используется. Удаляйте все неиспользуемое или необязательное: функционал, фрагмент кода, документацию, wiki-страницу, тест, флаг конфигурации, интерфейс, ветку контроля версий и т. д.

Используй или отбрось.

Пословица

Определившись с тем, что все перечисленное вызывает задержку в реализации проекта, остаток главы мы посвятим подробному изучению типичной ситуации в работе с кодом и на ней поучимся, как можно избавиться от разных негативных составляющих без потери общей ценности.

9.3. КАТЕГОРИЗАЦИЯ КОДА ПО СТЕПЕНИ ЕГО БЛИЗОСТИ

Прежде чем заняться удалением конкретных элементов, нужно немного отвлечься. На конференции GOTO 2016 Дэн Норт выступил с речью *Software, faster*. В ней он подразделил код на три категории по степени его близости к разработчику. Программисты обычно очень близко знакомы с кодом, который создали недавно. Они также хорошо знакомы с библиотеками и утилитами, которые часто используют. Все, что не входит ни в одну из этих категорий, является им незнакомым (далеким от них), а значит, и дорогим в обслуживании, так как программистам приходится всякий раз при обращении изучать это заново.

Если отнести эту идею к технической задержке, то код, который хорошо знаком ввиду его частого использования, обычно необходимо оставить. Но идея Дэна Норта также подчеркивает, что чем чаще мы работаем с кодом, тем он нам более знаком. И в свою очередь, удаление хорошо знакомого кода окажется дешевле и безопаснее по сравнению с удалением того, который еще предстоит изучить и понять.

Дэн Норт даже шутливо утверждал, что спустя шесть недель близость свежего кода к разработчику начинает утрачиваться и код быстро покидает все три категории хорошо знакомых фрагментов программы. Я думаю, что конкретный промежуток времени здесь неважен. Автор некоего кода естественным образом имеет преимущество в его понимании по сравнению со слабым сторонним программистом. Но важно учитывать, что это преимущество утрачивается

и в определенный момент даже автор кода теряет свою фору. Так что имеет смысл согласиться с идеей Дэна Норта о том, что промежуток должен измеряться в неделях, а не в месяцах. Поэтому при дальнейшем упоминании этой концепции я все-таки буду подразумевать промежуток в шесть недель.

9.4. УДАЛЕНИЕ КОДА В СТАРЫХ УНАСЛЕДОВАННЫХ СИСТЕМАХ

Стандартное определение *legacy-кода* звучит как: «Код, который мы боимся изменять». Такие коды часто появляются из-за фактора цирка. *Фактор цирка* выражает, сколько людей должно убежать и присоединиться к цирку, чтобы в результате утраты их знаний и памяти часть разработки остановилась. Если мы слышим утверждение вроде: «Только Джон знает, как развертывать эту систему», то говорим, что фактор цирка в этой системе равен одному.

Мы никогда не захотим останавливать разработку, поэтому нужно минимизировать риск, сохраняя фактор цирка достаточно большим по величине. Тем не менее, даже если вся команда знает о коде все, иногда и целую команду могут уволить или заменить внештатными сотрудниками. Когда мы утрачиваем фактор цирка, получаем неизвестный код, с которым стараемся не связываться, — *legacy-код*.

Этот код вполне может работать, но самого факта, что мы не чувствуем себя уверенными в его редактировании, уже достаточно, чтобы захотеть что-то в этом изменить. Если команда специалистов хочет быть продуктивной, то в работе с кодом она должна чувствовать себя комфортно и брать на себя ответственность за него. Если часть кода представляет собой темный лес, это означает, что мы понятия не имеем, когда или как он вдруг может сломаться. А что, если он вдруг сломается в 03:00 в субботу? Кто его исправит?

9.4.1. Прояснение кода с помощью шаблона «Фикус-удавка»

Первым делом в разрешении такой ситуации надо будет определить, сколько *legacy-кода* используется. Если он почти не применяется, то у нас есть шанс просто его удалить без дальнейшего расследования. Если активно используется лишь небольшая часть, то можно просто исправить именно ее и избавиться от остального. Если же в активном использовании находится весь *legacy-код*, то придется его хорошенько освоить и по возможности добиться стабильности его работы.

Когда мы пытаемся разобраться в legacy-коде, нужно прежде всего понять, как часто каждая из его частей вызывается. Но этого недостаточно. Нужно знать, сколько из этих вызовов оказываются успешными. Некоторый код вызывается, но дает сбой, и его результат никогда не используется. Это довольно распространенный случай для legacy-кода. Наконец, необходимо выяснить, насколько тесно связан такой код с остальной частью ПО. Я рекомендую начинать с последнего.

Помощником в этом процессе может выступить предложенный Мартином Фаулером шаблон «*Фигус-удавка*». Назван он был в честь фикусового дерева, семена которого прорастают на существующем дереве, затем постепенно растущий из семени фикус окутывает своего носителя до тех пор, пока полностью его не «удушает». Для нас аналогом носителя выступает legacy-система. Реализацию же самого шаблона разберем ниже.

Листинг 9.1. Legacy-код

```
class LegacyA {
    static a() { ... }
}
class LegacyB {
    b() { ... }
}

LegacyA.a();
let b = new LegacyB();
b.b();
```

Чтобы выяснить, насколько тесно связан фрагмент кода со всей системой, его можно изолировать, сделав так, чтобы все процессы проходили через виртуальный шлюз. Для этого инкапсулируем классы в новый пакет/пространство имен. Затем в этом новом пакете создадим новый класс шлюза (*Gate*). Все публичные модификаторы ограничим областью данного пакета и исправим ошибки, добавляя публичную функцию в класс шлюза.

Листинг 9.2. До

```
class LegacyA {
    static a() { ... }
}
class LegacyB {
    b() { ... }
}
LegacyA.a();
let b = new LegacyB();
b.b();
```

Листинг 9.3. После

```
namespace Legacy {
    class LegacyA {
        static a() { ... }
    }
    class LegacyB {
        b() { ... }
    }
    export class Gate {
        a() { return LegacyA.a(); }
        bClass() { return new LegacyB(); }
    }
}
```

```
let gate = new Legacy.Gate();
gate.a();
let b = gate.bClass();
b.b();
```

В этот момент точно известно, сколько точек контакта имеет legacy-код, потому что они все являются функциями в классе шлюза. У нас также есть удобная возможность добавить мониторинг, поместив его туда же, в класс шлюза: мы логируем каждый вызов и информацию об успешности его завершения. Это лишь необходимый минимум действий, который далее можно усложнить по своему усмотрению.

Листинг 9.4. До

```
namespace Legacy {
  // ...
  export class Gate {
    a() { return LegacyA.a(); }
    bClass() { return new LegacyB(); }
  }
}
```

Листинг 9.5. После добавления мониторинга

```
namespace Legacy {
  // ...
  export class Gate {
    a() {
      try {
        let result = LegacyA.a();
        Logger.log("a success");
        return result;
      } catch (e) {
        Logger.log("a fail");
        throw e;
      }
    }
    bClass() {
      try {
        let result = new LegacyB();
        Logger.log("bClass success");
        return result;
      } catch (e) {
        Logger.log("bClass fail");
        throw e;
      }
    }
  }
}
```

Этот код мы помещаем в продакшен и ждем. Команде нужно решить, сколько должно длиться такое ожидание, но я не думаю, что ее члены согласятся обслуживать функционал, который не используется хотя бы раз в месяц. (Определенные вещи выполняются по расписанию, например ежеквартальные, полугодовые или ежегодные финансовые отчеты. Но я не считаю, что они являются исключением для предложенной ниже доработки.) Когда legacy-код побыл в продакшене какое-то время, мы уже знаем, как активно используется каждая его часть и какие из вызовов всегда проваливаются.

9.4.2. Использование «Фигуса-душителя» для улучшения кода

Частота вызова фрагмента кода обычно является хорошим индикатором того, насколько окажется критическим провал такого вызова. Я предпочитаю начинать с простых решений: самые вызываемые компоненты практически однозначно нужно перемещать, а самые редковываемые — удалять. Так что сначала я разбираюсь с этими крайностями, после чего перехожу к остальным, над которыми уже нужно подумать. Код, который вызывается реже всего или постоянно проваливается, нужно тщательно анализировать, чтобы определить, насколько он критичен и обеспечивает ли стратегическую функциональность.

Если какой-то legacy-код оказывается критическим или стратегическим, то сначала нужно убедиться в том, что количество его вызовов этот факт отражает. Можно увеличить число обращений к этому коду путем доработки UI, с помощью обучения или маркетинга. Когда использование кода начинает отражать его важность, нужно в нем разобраться. Для этого есть два варианта. Первый — отрефакторить эту часть legacy-кода, тем самым удалив зацепление и хрупкость, после чего переместить код в категорию «свежего». Второй — переделывать эту часть и по ее готовности переключиться на новую версию, изменив один раз шлюз.

Если какой-то legacy-код оказывается не критическим и не стратегическим, то удаляем в шлюзе соответствующий метод. Подобные действия иногда могут приводить большие куски legacy-кода в неиспользуемое состояние, что можно выяснить с помощью IDE с применением методов и «Пробного удаления с последующей компиляцией» (4.5.1) для методов интерфейсов. Кроме того, устранение зацеплений упрощает вызывающий код, а иногда позволяет вообще его удалить. На рис. 9.1 приводится схематичное обобщение работы с legacy-кодом.



Рис. 9.1. Как поступать с legacy-кодом

9.5. УДАЛЕНИЕ КОДА ИЗ ЗАМОРОЖЕННОГО ПРОЕКТА

Представим, что заказчик продукта просит добавить серьезный функционал. Мы начинаем над ним работать, но к моменту завершения возникают организационные задержки: получение необходимого доступа, обучение пользователей и пр. Вместо того чтобы тратить время на ожидание, мы просто переходим к следующей крупной задаче. Но теперь у нас висит *замороженный проект*.

Замороженные проекты не ограничиваются кодом. Они могут включать таблицы баз данных, интеграции, сервисы и множество внешних компонентов. Как только автор-разработчик забывает о проекте, становится практически невозможно даже заметить его существование, особенно если организационная задержка произошла только из-за обучения пользователей. В системе от него не остается и следа, и никакое расследование его не отыщет.

В основной ветке кода может быть неиспользуемый фрагмент. При этом в коде ничто не указывает на то, что он не используется, а при любом внесении изменений его нужно учитывать, продолжать его поддержку. Это повышает ответственную нагрузку нас, специалистов. Кроме того, такой код рискует попасть в категорию legacy. Еще одна проблема замороженных проектов в том, что на момент устранения организационной задержки созданная функциональность может оказаться уже неактуальной.

9.5.1. Получение желаемого результата по умолчанию

В зависимости от того, находится ли проект исключительно в коде или же связан с базами данных, сервисами, интеграцией и пр., для этого есть немного разные решения. Рассмотрим их по очереди.

Если проект не имеет влияния вне базы кода, можно перевести его в отдельную ветку. Затем нужно добавить к нему тег, отметив период шесть недель, после которого этот тег нужно будет удалить. В результате, если мы не начнем использовать проект в течение этих шести недель, он будет удален.

Если же проект включает изменения, внешние по отношению к коду, то мы не можем поместить его в отдельную ветку. Вместо этого нужно создать тикет в инструменте управления проектами, отметив все подлежащие удалению компоненты, и запланировать этот тикет на шестинедельный срок. Если это происходит часто, то будет нелишним создать скрипты, которые устанавливают и удаляют наиболее часто используемые типы компонентов.

В обоих случаях заметно, что при отсутствии намеренных действий с кодом он будет исчезать. Следовательно, в таких сценариях окажется невозможным случайно внести техническую задержку — ее можно будет добавить только намеренно.

9.5.2. Минимизация затрат с помощью отрыва и стабилизации

Еще один способ сэкономить усилия при внесении крупных изменений — это использование предложенного Дэном Нормом шаблона «Отрыв и стабилизация» (именно из него и появилось правило шести недель). В этом шаблоне мы рассматриваем проект в отрыве — то есть реализуем его максимально отдельно от самого приложения без внимания к качеству: никакого автоматизированного тестирования и рефакторинга. Но при этом обязательно включаем мониторинг, чтобы понимать, сколько кода используется.

Спустя шесть недель мы возвращаемся к коду и смотрим, использовался ли он. Если да, то повторно его реализуем, но уже как положено — с рефакторингом и всем прочим. Если же код не использовался, то просто его удаляем, что сделать довольно легко, потому что он уже в отрыве, то есть уже имеет минимальную интеграцию с основной системой. Таким образом мы экономим не только время на удаление, но и то время, которое потратили бы на рефакторинг или тестирование кода, еще не зная, будет ли он вообще использоваться. На рис. 9.2 показана обобщенная схема работы с замороженными проектами.

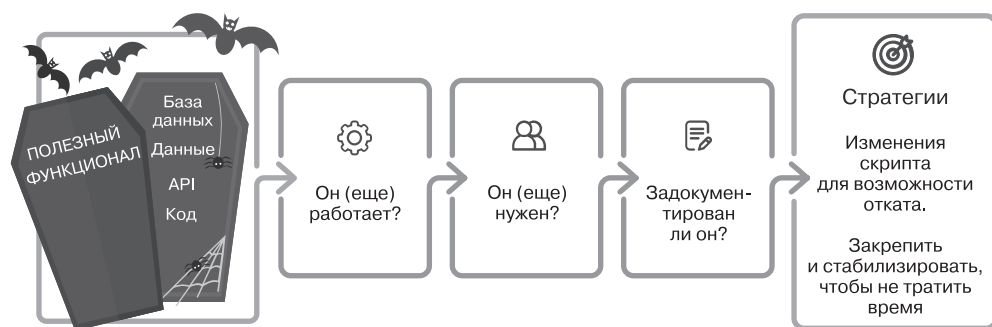


Рис. 9.2. Как поступать с замороженными проектами

9.6. УДАЛЕНИЕ ВЕТОК В СИСТЕМЕ КОНТРОЛЯ ВЕРСИЙ

В разных реализациях систем контроля версий ветвление работает по-разному. В централизованных системах, таких как Subversion, ветки дублируют всю базу кода, стоимость таких систем очень высока. В то же время размер веток в Git ограничивается байтами, независимо от размера базы кода. В текущем разделе мы рассмотрим только такие ветки, потому что в дорогостоящих альтернативах обсуждаемые нами проблемы не возникают.

Если говорить в общем, для создания веток может быть много разных причин. Основные же разделим на три категории:

- для внесения быстрого патча;
- для пометки коммитов, к которым нам может потребоваться вернуться позже, например релизов;
- для того чтобы работать над частью проекта, не влияя на работу коллег.

При использовании дешевых веток мы не столь тщательно отслеживаем их удаление, и в итоге с течением времени они неизбежно скапливаются.

Ветки из первой и третьей категории после слияния с основной веткой нужно удалять. Во второй категории вместо этого нужно использовать встроенный в Git метод для установки тегов. Но если мы это знаем, то почему же ветки продолжают накапливаться? Иногда это происходит вследствие элементарной невнимательности, когда мы забываем, например, отметить опцию **Delete Branch** при слиянии пул-реквестов или удалить экспериментальную ветку по завершении работы с ней. Иногда в ветках размещаются замороженные проекты, отрывы или прототипы, потому что мы считаем, что этот код однажды нам может понадобиться.

С подобными ситуациями можно довольно легко разобраться. Более сложный же тип веток — это такие, которые ожидают слияния, но при этом заблокированы, потому что не могут пройти условный шлюз для попадания в основную ветку. Это происходит, если шлюз включает в себя человеческую составляющую, например получение дополнительной команды интегрироваться или проведение асинхронного анализа кода. Оба эти условия затрудняют процесс непрерывной интеграции и могут легко стать узким местом, замедляющим разработку. Но если ветки стоят нам всего по несколько байт, то чем плохо их оставлять?

Подобно коду, ветки в Git почти свободны технически, но влекут сильную ответственную нагрузку: их надо помнить, различать, обрабатывать. По-хорошему, у нас должна быть только основная ветка и, возможно, ветка релиза. Любые другие не должны задерживаться больше нескольких дней. Оставляя их надолго, мы подвергаем себя дорогостоящим, утомительным, уязвимым для ошибок конфликтам слияния. А беспорядок порождает еще больший беспорядок.

9.6.1. Минимизация затрат за счет ограничения количества веток

Чтобы решить эту проблему, можно взять на вооружение элемент из методики разработки канбан. В канбан используется принцип работы с установкой ограничений на объем выполняемой работы (WIP). Это означает установку верхней планки для количества тикетов, одновременно исполняемых

командой. Такой подход помогает выявить узкие места в разработке, потому что они в конечном счете упираются в этот лимит WIP, не позволяя следующим участникам процесса начать свою работу. Когда возникает такая ситуация, что следующие сотрудники-разработчики не могут заняться новыми тикетами, им поневоле приходится выявлять узкое место и искать способ его устранения. Это стимулирует командную работу и непрерывное совершенствование процесса.

Обсуждаемая в этом разделе проблема с излишеством веток в точности отражает проблему узких мест канбан, значит, здесь подойдет такое же, как в канбан, решение: ввести жесткий лимит на количество веток. Разберем вещи, о которых нужно помнить при определении предела для WIP или числа веток. Этот лимит не должен быть меньше числа рабочих станций, чтобы все они могли работать параллельно. Здесь *рабочая станция* представляет собой единицу, которая может работать независимо, например один ансамбль в случае ансамблевого программирования, одна пара в случае парного или один разработчик в иных случаях. Установка более высокого лимита создаст эффект буфера, который внесет в систему задержку, но иногда может и пригодиться, если какая-то часть работы окажется существенно больше остальных в объеме. Мы стремимся максимально сократить элемент задержки в системе. Когда же лимит установлен, то самое важное — это ему следовать и не сдвигать ни при каких условиях, за исключением разве что изменения размера команды. Рисунок 9.3 схематично обобщает работу с ветками в системе контроля версий.



Рис. 9.3. Как работать с ветками

9.7. УДАЛЕНИЕ ДОКУМЕНТАЦИИ КОДА

Документация кода бывает представлена в разных формах, таких как wiki-страницы, Javadoc, проектные документы, руководства и пр. Здесь мы не будем затрагивать комментарии к внутренним методам. О них мы уже говорили в предыдущей главе.

Документация оказывается очень полезной, если она отвечает трем требованиям:

- *релевантности* — она должна давать ответ на соответствующий вопрос;
- *точности* — ответ на искомый в документах вопрос должен быть корректен;
- *доступности* — поиск ответа не должен вызывать трудности.

Если одного из этих качеств не хватает, то ценность такой документации сильно падает. Писать хорошее документальное сопровождение сложно, требуются затраты усилий для обеспечения его релевантности и точности в каждый момент времени. Очевидно, что документация должна использоваться в работе не реже, чем меняется описываемый ею предмет. В противном случае ее поддержание не будет оправдывать вложенных сил. Сохранение актуальности достигается за счет частого внесения в документы корректировок или заблаговременного обобщения формулировок. Обобщенные формулировки — это те, которые абстрагируются от часто изменяющихся элементов.

Суть опасности сохранения утратившей актуальность документации зависит от того, какое из трех качеств она нарушает. Меньшее из всех зол — это когда ответ на вопрос просто сложно найти. В этом случае мы тратим только время на поиски и написание. Хуже, когда мы сталкиваемся с нерелевантной документацией: здесь мы не только тратим время на поиск, но еще и вынуждены при каждом поиске ответа отсеивать ненужную информацию, в результате нам снова приходится заниматься изысканиями. Самым же печальным сценарием можно назвать неточную документацию: в лучшем случае она может вызвать непонимание и сомнения, а в худшем привести к совершению ошибок.

9.7.1. Алгоритм для определения необходимости документирования

Может показаться, что документация избавляет нас от повторяющегося поиска одного и того же, но так ли это? Да, если документация не устарела, действительно от обращения к ней есть практическая отдача. А на самом деле документация может со временем утрачивать свою релевантность или точность.

Еще одно замечание: документировать нужно не все. Когда мне приходится решать для себя, есть ли смысл документировать что-либо, я слеую таким рассуждениям.

1. Если объект документирования (код) изменяется часто, то пользы от его документирования мало.
2. Если он используется редко, то нужно задокументировать.
3. Если можно автоматизировать его тестирование, то нужно так и сделать.
4. В противном случае следует осваивать его до принятия на интуитивном уровне.

Заметьте, что помочь в актуализации документации может частое использование кода, но оно же приведет к упомянутым ранее частым доработкам. Для них вполне логично задействовать новичков в команде, поручая им просмотр и исправление неточностей в комплекте документов. При возникновении ошибок в работе с кодом лучше всегда анализировать: ошибка происходит из-за неверной документации, или просто сотрудник допустил ее без всякой связи с документами. Если возникают сомнения или обнаруживается, что документация нуждается в пересмотре, сотруднику следует разобраться и заявить об этом во всеуслышание.

На рис. 9.4 приводится обобщенная схема поддержки документации. В качестве способной сохранять точность альтернативы документации можно задействовать автоматизированные тесты, с чем мы далее и познакомимся.

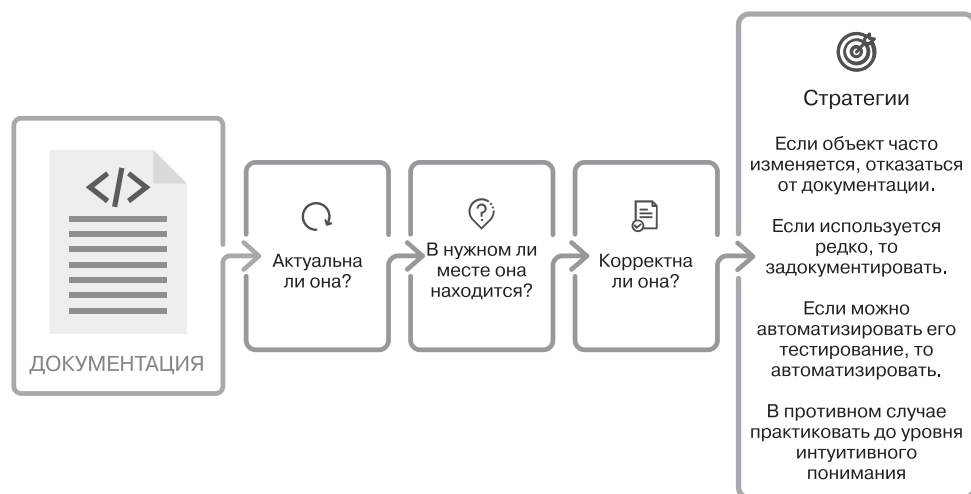


Рис. 9.4. Как поддерживать документацию

9.8. УДАЛЕНИЕ ТЕСТИРУЮЩЕГО КОДА

Автоматизированные тесты (в этом разделе просто *тесты*) в своих разных формах гораздо богаче по свойствам, чем документация. Тесты различаются по типам, каждый тип охватывает свой набор свойств. Какие именно могут быть свойства тестов, можно узнать из книги Кента Бека *Test Desiderata* (<http://mng.bz/BKW2>), там он описывает 12 свойств тестов. Я не стану разбирать каждый тип и выделю лишь те, которые вредят разработке.

9.8.1. Удаление оптимистичных тестов

Иногда мы пишем код вроде функции `hash` и хотим его протестировать, для чего придумываем тест типа «Если $a = b$, тогда $\text{hash}(a) = \text{hash}(b)$ ». Здесь мы вроде бы проверяем что-то, что должно быть истинным, но по факту спотыкаемся о тавтологию: проверяем что-то, что истинно всегда.

Одно из обязательных свойств тестов в том, что они должны внушать уверенность. Зеленый тест после своего прохождения должен повышать нашу уверенность в том, что протестированный код работает. Значит, тесты должны что-то проверять. Если же тест никогда не может провалиться, как показано выше, то в нем нет смысла и уверенность в работе кода он может вызвать только ложную.

В сообществе тестирующих бывает хороший принцип: «Никогда не доверяй тесту, провал которого ты не видел». Скажем, мы находим в коде ошибку. Создав тест до ее исправления, можно проверить, будет ли он корректно давать сбой. Дал — значит, тест работает! Если же создать его после исправления ошибки, то мы лишь увидим, как он благополучно проходит и завершается.

9.8.2. Удаление пессимистичных тестов

По той же логике красный тест должен означать неисправность чего-то, что нужно исправить. Именно поэтому терпимость к проваливающимся тестам должна полностью отсутствовать. Провал теста — сигнал: требуется смотреть код! Если у нас есть тесты, которые всегда оказываются красными, то мы рискуем выработать усталость от сигналов тревоги и упустить критическую ошибку, даже когда тесты ее перехватят.

9.8.3. Исправление или удаление ненадежных тестов

И оптимистичные, и пессимистичные тесты представляют крайние точки, которые всегда проходят либо всегда проваливаются. Но проблемы могут возникнуть и с тестами, которые могут непредсказуемо оказаться красными или зелеными. Именно их непредсказуемость обуславливает то, что их классифицируют как *ненадежные тесты*. Как и оба рассмотренных выше типа, они тоже не ведут к выполнению каких-либо действий, кроме, разве что, выполнения самих тестов несколько раз дополнительно. Мы предпринимаем действия, только если тест красный и это связано с необходимостью корректировки кода. Всем остальным тестам, которые ненадежны, в базе кода делать нечего.

9.8.4. Рефакторинг кода для избавления от плохих тестов

Совершенно другая категория состоит из тестов, которые требуют внимательной настройки или содержат излишние повторения, в связи с чем их необходимо рефакторить или создать сложные конфигурации тестирования. Эти тесты опасны тем, что вызывают ощущение выполнения ценной работы: мы упрощаем, локализуем, делаем все правильно. К сожалению, делать все правильно в неправильном месте все равно будет неправильным. Если тест оказывается сложнее кода, то откуда нам знать, в коде ошибка или в нем? Необходимость в рефакторинге тестов указывает на то, что архитектура именно тестируемого кода не совершенна. Любой рефакторинг должен проводиться в коде, но не в тесте.

9.8.5. Специализация тестов для их ускорения

В некоторых местах мы используем сквозные тесты для проверки работоспособности определенной функциональности. Эта техника по-своему полезна, но такие тесты могут оказаться медленными, а наличие их большого числа повлияет на то, как часто мы сможем их запускать. Если некоторые тесты вынуждают нас отказаться или отсрочить выполнение других тестов, то первые очевидно вредят разработке, и с этой ситуацией необходимо разобраться. Для этого есть два способа: отделить медленные тесты от быстрых и продолжить выполнять последние как можно чаще. Либо выяснить, что вызывает провал медленного теста и, если ответ «ничто», удалить его (это оптимистичный тест). В глубине системы есть, вероятно, всего несколько мест, где может возникнуть сбой, и есть реальная возможность сделать тесты именно для этих мест. Такие узкоспециализированные тесты окажутся быстрее, что позволит исправлять ошибки более оперативно. На рис. 9.5 приводится обобщенная схема работы с автоматизированными тестами.



Рис. 9.5. Как обрабатывать автоматизированные тесты

9.9. УДАЛЕНИЕ КОДА ДОПОЛНИТЕЛЬНОЙ КОНФИГУРАЦИИ

Большинство программистов знают, что жестко прописанный код — это плохо.

В качестве первого противоядия такому явлению мы осваиваем извлечение жестко прописанного значения в константу, после чего по мере набора опыта знакомимся с такой максимой.

Если не можешь довести что-то до совершенства, сделай это хотя бы настраиваемым.

Максима

Возможность изменения конфигурации, проще говоря, настраиваемость, может повысить практическую ценность нашего ПО при увеличении числа пользователей, не вызывая существенного разрастания базы кода. Когда настраиваемость реализуется в виде переключателей функционала, она позволяет нам отделить *развертывание* от *релиза*, увеличить частоту развертывания (полезных и осознанных обращений к коду) и в то же время переводит релиз из разряда технических решений в разряд бизнес-решений.

Но есть здесь и обратная сторона: в каждом месте, куда мы добавляем дополнительную конфигурацию, код становится сложнее. Хуже того, в большинстве случаев мы удваиваем область тестирования, потому что нам нужно проверить каждую опцию относительно всех переключателей. Область тестирования разрастается экспоненциально. Хорошо, когда некоторые переключатели оказываются независимыми: их можно протестировать одновременно. Параллельное тестирование упрощает ситуацию, но увеличивает риск появления ошибок, связанных с взаимодействиями между этими переключателями.

9.9.1. Ограничение конфигурации настраиваемости во времени

В качестве решения проблемы с повышением сложности кода из-за внедрения его настраиваемости я предлагаю как можно большую часть настраиваемости делать временной. С этой целью я категоризирую добавляемую конфигурацию кода на основе ее предполагаемого срока службы. Так вырисовываются следующие категории: экспериментальная, переходная и постоянная.

Экспериментальная конфигурация

В этой книге уже появлялись примеры экспериментальной конфигурации — переключатели функционала. Их необходимо удалять сразу после релиза нового функционала. А для того, чтобы в итоге не произошло наведение новых проблем,

сделать это нужно в течение шести недель, о чем уже говорилось. Еще один тип экспериментальной конфигурации позволяет подтвердить преимущество вносимых изменений относительно прежней версии. Иногда это называется *бета-тестированием* или *тестированием А/В*. В коде оба эти вида очень похожи между собой, но преследуют разные цели. В этом случае конфигурация позволяет опробовать изменение лишь на некоторой доле пользователей. Они могут оценить, какой вариант лучше — прежний или текущий. Эта техника позволяет нам подстроиться под обратную связь от этих пользователей или отказаться от изменения кода, не доставляя неудобств всем пользователям вообще.

По своему опыту могу сказать, что тестовая конфигурация часто просачивается из экспериментальной стадии, становясь постоянной, разделяя базу пользователей на тех, у кого функционал включен, и тех, у кого нет. При этом возрастает лишь сложность, но не удобство использования. Это плохо, и во избежание подобных сценариев нам нужно действовать на упреждение: с самого начала определять, что является экспериментальным, и создавать напоминание об удалении этой конфигурации сразу после тестирования (вписываясь все в те же шесть недель).

Переходная конфигурация

Переходная конфигурация нужна, когда бизнес или база кода претерпевают сильные изменения. Примером может служить переход от legacy-системы к новой. Мы не можем ожидать или настаивать, чтобы внесение столь масштабных изменений произошло в течение шести недель, поэтому здесь придется иметь дело с более длительным увеличением сложности и более затратным процессом чистки. К нашему счастью, долгосрочные переходы обычно имеют два свойства, которыми можно воспользоваться.

Во-первых, многие типы переходов оказываются невидимы для пользователя. Следовательно, нас вполне устроит связывание *релиза* и *развертывания*. Это означает, что мы можем сделать конфигурацию частью кода, а не чем-то внешним. В свою очередь, это позволит нам собрать всю связанную с переходом конфигурацию в центральной точке, отделенной от всего остального. Таким образом мы явно укажем, что эти флаги конфигурации являются более связанными и должны рассматриваться как единая коллекция.

Во-вторых, рано или поздно наступает момент, в котором переход завершается и переходную часть можно удалять. Пользуясь этим, можно избежать траты времени на копание в коде и удаление его небольших частей (своеобразный список «нужно сделать»), просто дождавшись, когда можно будет удалить сразу все. Чтобы обезопасить такой подход, нужно снова использовать шаблон «Фигурус-удавка», отделив шлюзом *весь* доступ к legacy-компоненту. При удалении целого кода получим хороший ориентир: когда возникнет возможность удалить шлюз, не вызвав ошибок в коде, станет возможно спокойно удалять

весь legacy-компонент. Делается это либо через «Пробное удаление с последующей компиляцией», либо путем постепенного удаления методов в шлюзе по мере отказа от их использования. Когда шлюз опустеет, его также можно будет удалить.

Постоянная конфигурация

Последняя категория — это постоянная конфигурация настраиваемости кода. Ее можно назвать особенной, потому что она должна обладать следующими качествами: либо часто использоваться, либо оказываться простой в поддержке. Примером конфигурации с возросшей частотой использования является повторное задействование большей части одного и того же ПО для двух разных клиентов за счет вынесения их отличий за находящийся в коде переключатель конфигурации. К этой же категории может быть отнесена конфигурация, позволяющая задействовать разные уровни использования кода, давая возможность подстраиваться под разные бизнес-масштабы. Оба этих примера потенциально могут удвоить число пользователей, что, безусловно, увеличит частоту обращения к самому коду и тем самым сделает оправданным увеличение объемов его сопровождения.

Примером подкатегории простых в поддержании конфигураций является предоставление пользователям переключателя между светлой или темной темой. Он повлияет только на внешние части кода (стилизацию), а значит, обслуживание не усложнит. Тем не менее это позволит предоставить заинтересованным пользователям возможность использовать альтернативный режим эксплуатации продукта.

Необходимо проявлять бдительность и очень тщательно оценивать, что попадает в постоянную категорию. Если это не ведет к увеличению использования и не оказывается простым в обслуживании, то есть вероятность, что оно того не стоит и лучше отказаться от разрастания постоянной конфигурации.

На рис. 9.6 приводится обобщенная схема работы с дополнительными конфигурациями в коде.

9.10. УДАЛЕНИЕ КОДА ДЛЯ СОКРАЩЕНИЯ ЧИСЛА БИБЛИОТЕК

Если нужно быстро и недорого получить много функциональности, то можно использовать сторонние библиотеки. Некоторые из них избавляют от написания тысяч строк кода, одновременно обеспечивая более высокое качество или безопасность, чем можно было бы получить собственными усилиями. Я всегда рекомендую оставлять вопросы безопасности тем специалистам, кто посвятил

этой стезе свою жизнь, например разработчикам этих библиотек. Ведь, будучи недостаточно искушенными в этой области людьми, обычные программисты неспособны противостоять злоумышленникам, которые зачастую также посвящают освоению этого мастерства целую жизнь.

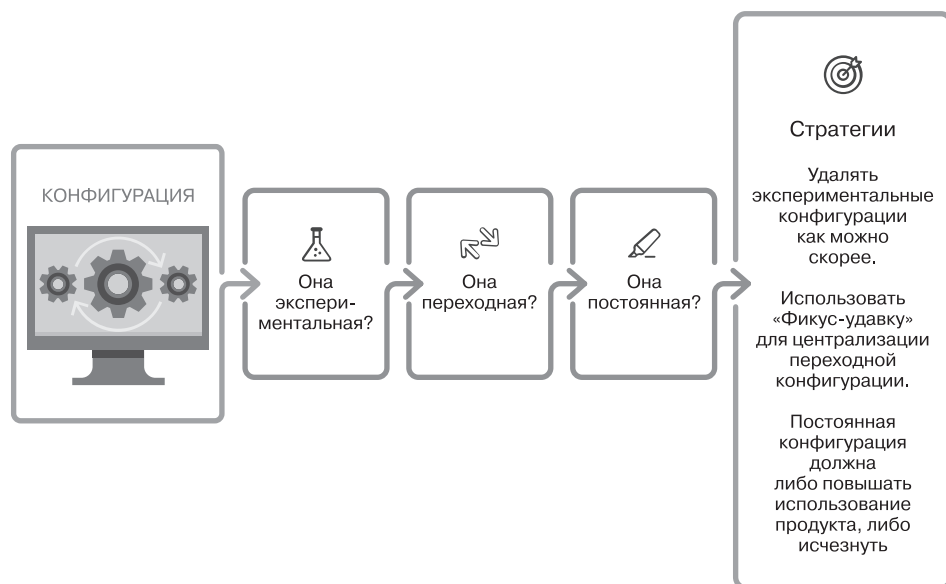


Рис. 9.6. Как работать с кодом дополнительной конфигурации

Еще одно соображение безопасности, которое подталкивает нас к использованию сторонней безопасной библиотеки, состоит в том, что качество ее реализации прямо влияет на жизнеспособность ПО, увеличивая ее. Ведь если в программе вдруг произойдет серьезный инцидент, касающийся безопасности, то это может разрушить доверие пользователей, а с ним погубить и сам продукт. Вероятность этого уменьшается с использованием сторонних библиотек.

Следующая причина обратиться к внешней библиотеке в том, что она дает возможности, которых иначе мы не имеем, например позволяет использовать фронтенд-фреймворки вроде Swing (Java), React (TypeScript) или WPF (C#). Реализация всех этих возможностей представляет собой большой объем кода, для создания которого нужны специализированные навыки — знание графического программирования. Этими навыками наша команда может не обладать, а в функционале внешних библиотек приобретет готовые реализации и сможет ими воспользоваться.

К сожалению, использование библиотек — это палка о двух концах, потому что, хотя нам и не приходится поддерживать код внутри библиотек, появляется необходимость обновлять сами библиотеки, что иногда ведет также и к адаптации нашего кода. Выполнение этого может требовать немало времени и повышает риск возникновения ошибок. Использование библиотек также увеличивает общую когнитивную нагрузку на команду, потому что ее члены должны иметь хотя бы минимум необходимых знаний о них.

В случае применения библиотек мы частично утрачиваем прогнозируемость, поскольку уже не можем предсказать событие выпуска обновления или то, сколько времени уйдет на корректировку базы кода. Иногда задействуемый нами функционал устаревает или удаляется, и на его замену нужно создавать нечто новое. Иногда возникают ошибки, и нам приходится реализовывать временные обходные пути или хаки, чтобы вернуть работоспособность программе. Наконец, когда ошибки в библиотеках исправлены, нам нужно избавиться от уже ненужных обходных решений, чтобы они не мешались в коде. Приходится также выбирать, будем ли мы вникать в исходный код библиотеки, разбираться в нем или же согласимся на ущербность системы безопасности, так как библиотека является еще одним потенциальным вектором атаки, за который мы можем поручиться, только если будем относиться к нему как к собственному коду.

Опасность использования внешних библиотек возрастает также потому, что в большинстве современных языков имеется менеджер пакетов, который сильно упрощает добавление зависимостей. И, как иллюстрирует предыдущий сценарий, нам необходимо беспокоиться не только о вводимых нами зависимостях, но и обо всех зависимостях этих зависимостей и т. д.

ИЗВЕСТНЫЙ МЫСЛЕННЫЙ ЭКСПЕРИМЕНТ

В одном из своих постов Дэвид Гилбертсон приводит наводящий на размышления выдуманный сценарий, в котором он выпускает небольшую JS-библиотеку, добавляющую выводимым в консоль сообщениям цвета. Он говорит так: «Люди любят красивые цвета» и «Мы живем во времена, когда люди устанавливают пакеты npm с таким рвением, будто бы принимают обезболивающее». Используя минимум социальной инженерии (пул-реквесты), он внедряет свою библиотеку в другие библиотеки, и она начинает получать сотни тысяч загрузок ежемесячно. Однако пользователи не догадываются, что эта библиотека содержит вредоносный код, крадущий данные с использующих его сайтов.

9.10.1. Ограничение использования внешних библиотек

Один из способов избежать описанных проблем — выбирать библиотеки от проверенных поставщиков, чтобы можно было довериться их внутреннему качеству и требованиям безопасности. Такие поставщики стараются избегать внесения кардинальных перемен в свои продукты. Перепроверять качество безопасности и корректировать код нам нужно только при обновлениях, так что в случае редкого изменения библиотек мы эти затраты сокращаем.

Еще одним способом противодействия озвученным проблемам является частое обновление кода. В DevOps говорят так.

Если что-то причиняет боль, делай это чаще.

Поговорка DevOps

Если мы делаем что-то часто, то у нас возникает больший стимул рационализировать это и сократить сопутствующие неприятности. Этот аргумент стоит за такими процессами, как непрерывная интеграция и доставка. Еще одно преимущество более частого выполнения чего-либо в том, что так мы сокращаем размер отдельных единиц работы, что ведет к распределению затрат, снижению рисков и общей нагрузке.

Тем не менее это не помогает разобраться с рисками нарушения безопасности, упомянутыми в предыдущем разделе. Последним и самым простым решением, которое я предлагаю, будет такое: сделать зависимости видимыми, после чего категоризировать каждую библиотеку как *расширяющую* или *критическую*. Этот подход уменьшит зависимость от закрытой библиотеки и сведет к минимуму необходимость слепо доверять ей.

Когда расширяющая библиотека вдруг выйдет из строя, просто удаляйте ее, восстанавливайте работу приложения и уже потом ищите ей замену. Будьте осторожны с переводом библиотеки из категории расширяющих в критические. Если в базе притаились неиспользуемые библиотеки, удаляйте их. Конечно, их функции относительно сложно реализовать самостоятельно, но зачастую оно стоит затраченных усилий, так как в конце концов придает уверенности в работоспособности кода.

Скажем, мы установили библиотеку jQuery с сотнями функций, но используем ее только для выполнения вызовов Ajax. Намного лучше будет либо найти библиотеку попроще, которая удовлетворит наши потребности, либо реализовать собственную функцию, которая будет делать то же самое. В плане безопасности

нам необходимо анализировать весь код в библиотеке, даже если не используем всю ее напрямую. Рисунок 9.7 схематично обобщает описанный принцип работы со сторонними библиотеками.



Рис. 9.7. Как работать со сторонними библиотеками

9.11. УДАЛЕНИЕ КОДА ИЗ РАБОТАЮЩЕГО ФУНКЦИОНАЛА

Код — это своего рода обуза. Он требует времени на обслуживание, изобилует непредсказуемостями, отсюда возникают риски. Компенсирует же весь этот негатив его практическая полезность. Это частое заблуждение, что количество функционала в продукте напрямую определяет его практическую полезность. Некоторые считают, что увеличение возможностей ПО обязательно приведет к увеличению его ценности. К сожалению, здесь все не так просто.

На протяжении этой главы я старался пояснить, что баланс между затратностью кода и выгодой от его функциональности не так прямолинейно однозначен. Кроме этого, имеется множество влияющих факторов: как долго мы согласны мириться с увеличением сложности, насколько мы ценим прогнозируемость, как тестируем новые возможности, как тщательно знакомим с ними людей и т. д. Есть два способа повысить ценность кода в соотношении затрат/выгоды. И если выгоду от добавления функционала повысить и оценить сложно, то найти способы сократить затраты путем рефакторинга или, что еще лучше, удаления кода оказывается проще. Это верно, даже если речь идет об удалении работающего функционала, затраты на обслуживание которого выше приносимой им пользы.

По той же логике все, что не используется, независимо от своего потенциала, является лишь затратами. Именно поэтому нужно любить удалять код, ведь каждым разумным удалением мы моментально повышаем ценность всей базы кода. На рис. 9.8 показана обобщенная схема оптимизации рабочего функционала.



Рис. 9.8. Как оптимизировать рабочий функционал

РЕЗЮМЕ

- Техническое неведение, технические потери, технический долг и техническая задержка являются факторами, замедляющими и усложняющими разработку. Техническое неведение обычно возникает ввиду неопытности и прорабатывается только путем непрерывного стремления к техническому совершенству. Технические потери зачастую возникают из-за недостатка времени, но поскольку они не приносят никаких выгод, то являются чистым саботажем. Технический долг формируется ввиду обстоятельств и является вполне приемлемым при условии, что он будет временным. Техническая задержка является побочным эффектом разрастания базы кода. Она оказывается необходимой и неизбежной, потому что наше ПО моделирует очень сложный мир.
- Для прояснения legacy-систем и удаления из них лишнего кода можно использовать шаблон «Фигура-удавка». С его же помощью можно централизовать конфигурацию на время переходного периода.
- Использование шаблона «Отрыв и стабилизация» помогает сократить часть затрат, возникающих из-за замороженного проекта. Более того, устанавливая срок для удаления проекта вместо его сохранения, мы не позволяем ему оказаться причиной задержки в дальнейшем.

- Удаляя несовершенные автоматизированные тесты, мы повышаем собственную уверенность в тестировании, а значит, и общую пользу от него. Плохие тесты могут быть оптимистичными, пессимистичными или ненадежными. Можно избавиться от сложных тестов путем рефакторинга кода тестируемого ПО. А введение специализации медленных тестов, отказ от их универсальной пригодности повысит их скорость. Устанавливая лимит на ветвление, можно сократить умственную нагрузку, время, затрачиваемое на отслеживание устаревших веток в системе контроля версий.
- Устанавливая строгие временные границы для существования дополнительной конфигурации кода, мы минимизируем увеличение его сложности.
- Ограничивая использование внешних библиотек, мы экономим время на обновление ПО и анализ качества кода, одновременно повышая предсказуемость поведения создаваемой системы.
- Чтобы документация кода была полезной, она должна быть релевантной, точной и хорошо организованной для поиска в ней информации. Для принятия правильных решений относительно ведения документации можно использовать удобный алгоритм.

10

Никогда не бойтесь добавлять код

В этой главе

- ✓ Симптомы боязни добавить код.
- ✓ Как побороть этот страх.
- ✓ Компромиссы при дублировании кода.
- ✓ Обратная совместимость и как ее придерживаться.
- ✓ Снижение рисков с помощью выключателей функциональности.

После разбора всех больных мест кода в прошлой главе можно начать бояться вообще писать его. Ведь в конце той главы мы обозначили, что код увеличивает затраты. Еще одним источником страха выступает опасение написать несовершенный код. Совершенство становится нереалистичной целью, учитывая то, что разрушить его можно легко и множеством способов. Понятие идеального кода включает в себя соблюдение требований по множеству аспектов, таких как быстродействие, структура, уровень абстракции, простота использования, простота обслуживания, новшество, креативность, корректность, безопасность и т. д. Удержать все это в голове, одновременно стремясь решить непростые практические задачи, оказывается просто невозможным.

Я начал писать код еще до того, как получил формальное образование в компьютерных науках. В те времена я был очень продуктивен и креативен, потому что меня заботило лишь, чтобы мой код заработал. После этого я пошел в университет

и узнал все те способы, которыми код может провалиться, или ввиду чего он может оказаться плохим. Моя продуктивность упала. Я получал задание и засиживался над ним в раздумьях в течение часов и даже дней и только потом выдавал первую строчку. Когда я осознал, какой эффект возымел на меня этот страх перед написанием кода, начал бороться с ним — с тех самых пор я веду непрерывную борьбу со страхом писать код — своим собственным и других разработчиков.

В этой главе я поведаю о симптомах, с помощью которых определяю подобные ситуации, и предложу рекомендации для противодействия им. По сути, добавление кода безопаснее, чем его изменение, стоит использовать этот факт в повседневной работе, например, через повторение кода или расширяемость.

10.1. ПРИНЯТИЕ НЕУВЕРЕННОСТИ: ВСТРЕТИТЬ ОПАСНОСТЬ ЛИЦОМ К ЛИЦУ

Мы не можем работать эффективно, если напуганы. Разработка ПО связана с изучением предметной области и кодированием этого знания на языке программирования. Наиболее эффективный способ выработки знаний — это экспериментирование, но для него требуется смелость: мы должны открыто показывать те детали, в которых больше всего сомневаемся, и привлекать к этому внимание. Именно поэтому в популярном фреймворке Scrum смелость выступает одной из пяти ценностей. В ходе крупного исследования специалисты Google выяснили, что самой весомой предпосылкой успешности команды является чувство психологической безопасности: имеется в виду доверие членов команды друг к другу и решительное принятие рисков.

Страх крепчает еще больше, когда мы сталкиваемся с областью задачи, в которой не особенно уверены, хотя именно такие случаи лучше всего подходят для обучения. В импровизационном театре есть принцип, который звучит так: *«встретить опасность лицом к лицу»*. Этот принцип признает, что все мы естественным образом склонны избегать неудобных ситуаций, но лучший театр строится на смелом противодействии таким ситуациям и их использовании для собственного совершенствования. Патрик Ленчиони говорит об этом принципе как об одном из наиболее важных уроков для эффективного консалтинга (подкаст *At the Table*, апрель 2017 года). Я думаю, что это в такой же степени применимо и к разработке ПО.

Неважно, насколько мы хороши как профессионалы, если мы ничего не производим. Я вошел в индустрию сразу же после университета. Наполненный юношеской надменностью, я испытывал чувство превосходства над остальными — чувство, которое испарилось, как только мне пришлось отправлять свой

первый код в продакшен. Масштаб всех тех вещей, которые вдруг могли пойти не так, просто зашкаливал. Будучи внештатным сотрудником, я работал с разными компаниями, а первый проект на новом месте всегда сопряжен со страхом. Вот почему я выработал стратегию, которая также используется во многих крупных компаниях, вместе с принципом «смотреть опасности в глаза». Суть стратегии очень проста: на новом месте я должен заявить о себе в продакшене в первый же день. Это моментально избавляет от страха и обеспокоенности, а также показывает мне, как я могу принести ценность в ту или иную команду.

Страх — это форма психологической боли. Как я сказал еще в первой главе, если что-то причиняет боль, делайте это чаще! Если что-то пугает, работайте с этим чаще, пока страх перед этим не пройдет вовсе.

10.2. ИСПОЛЬЗОВАНИЕ ОТРЫВА ДЛЯ ПРЕОДОЛЕНИЯ СТРАХА СОЗДАТЬ ЧТО-ТО НЕПРАВИЛЬНО

Будучи внештатным сотрудником в разных организациях, я часто вижу, что страх перед неудачей препятствует продуктивности. Страх заставляет людей обсуждать, проектировать или думать о том, какими способами создать что-то, прежде чем они изучат, что именно им надлежит создавать. Такое происходит, когда страх сделать что-то с ошибкой превозмогает страх не реализовать назначения самой разработки. Когда такое видишь, необходимо бить тревогу.

Я рекомендую рабочий поток программирования начать с отрыва: это поможет решить проблему. На рис. 10.1 первым идет этап исследования, которое обычно принимает именно форму отрыва. Код, создаваемый на этом этапе, может и не дожить до реализации основного предназначения ПО, но будет не так важно, окажется ли он ошибочным. Это и послужит избавлению от страха перед самим кодированием.



Рис. 10.1. Рекомендованный поток разработки

Отрыв дает нам осознание того, что мы вполне можем доработать нашу первую версию на следующих подходах, а также внушает уверенность для этого. Отрывы — это мощная техника, но их введение может вызывать сложности, так как для этого требуется дисциплина. Внутренняя политика работы в компании должна позволять и даже приветствовать написание кода, который в итоге будет выброшен. Все заинтересованные лица проекта — стейкхолдеры — должны понимать, что продукт — это *знание*, а не код или функциональность.

У стейкхолдеров, а иногда и у разработчиков есть соблазн использовать код из отрыва в результирующем варианте продукта. Это сигнализирует о том, что господствует противоположное понимание, как будто продукт — это код, а не знание. Этот взгляд имеет катастрофический побочный эффект в том, что появляется стремление сделать код отрывов максимально качественным. В результате у разработчиков снова формируется такой же страх, как если бы они писали код для продакшена, ведь они действительно начали работать на чистовую в том же дефиците знания. Для сохранения преимущества использования отрывов нужно строго следить за тем, чтобы их код не отправлялся в продакшен, а также продвигать идею, что продукт — это знание. Отрывы можно использовать для проверки предположений, оценки удобства использования и экспериментирования.

Чтобы культивировать в команде идею приоритета знания, будет уместно представлять результаты в формате, обычно ассоциируемом с продуктами знаний, такими как слайдовые презентации или техническая документация. Представление результата отрыва в качестве одного слайда, показывающего три самых важных момента, в сопровождении скриншота или макета наглядно покажет стейкхолдерам, что время было потрачено с пользой. Впоследствии такие слайды можно использовать на еженедельных встречах команды с целью обмена знаниями, что приведет к снижению фактора цирка, укрепит дух команды и еще больше сфокусирует внимание на аспекте знания.

10.3. ПРЕОДОЛЕНИЕ СТРАХА ПЕРЕД ЛИШНИМИ ЗАТРАТАМИ ИЛИ РИСКОМ УСТАНОВКИ ФИКСИРОВАННОГО СООТНОШЕНИЯ

Еще один симптом страха перед кодом — чрезмерное внимание к инфраструктуре и вспомогательным средствам разработки, когда окружающие инструменты и конвейер разработки оказываются намного сложнее, чем сам итоговый код продакшена. Прежде чем написать одну строку бизнес-логики, некоторые команды проводят огромное количество времени за настройкой сред тестирования, продвинутых стратегий ветвления и структур репозитория, систем выключателей функционала, фронтенд-фреймворков, автоматизированных сборок и развертываний. Все эти вещи требуют времени и места в создании ПО. Но они должны

способствовать сокращению рисков или лишних затрат и не являются основной целью. Уделение им слишком много времени предполагает, что страх затрат или риска оказывается сильнее желания разработать и отправить готовый продукт. Когда у нас нет кода, то нет и рисков или затрат, а значит, трата времени на все эти усилия представляет чистой воды прокрастинацию. Настройка этих вспомогательных систем может быть интересной и трудной, вызывая ощущение важности. Однако если мы не отправляем через них код в продакшен, то и сокращать затраты оказывается не для чего, а значит, и фактической ценности в таких инструментах нет. Худший сценарий — это когда поддержка или разработка этих систем требует слишком много усилий и мы уже не можем заняться тем, что действительно имеет значение.

РЕАЛЬНАЯ ИСТОРИЯ

Однажды я присоединился к команде, которая никак не могла дойти до этапа отправки. Меня в общих чертах ввели в курс дела относительно компании и проекта. Я же, как обычно, поинтересовался, какую процедуру развертывания кода они используют. На это ведущий разработчик начал вздохнуть и расписывать поразительно запутанный конвейер сборки и развертывания. Он мог все, разве что не заваривал кофе. Недоумевая, почему у них проблемы с доставкой кода, я перечислил список возможных причин. Предположив, что сложности заключались в излишнем зацеплении внутри архитектуры, я попросил разрешения взглянуть на базу кода. «У нас еще нет кода. Мы работали над выстраиванием этого конвейера».

Рекомендуемое решение по избеганию подобных ситуаций я почерпнул из книги *The DevOps Handbook*, написанной Джиним Кимом в соавторстве с другими специалистами (IT Revolution Press, 2016). Ее авторы советуют выделять 20 % времени разработчиков на нефункциональные требования, такие как обслуживание и разработка вспомогательных инструментов. Установка такого лимита работает в двух направлениях: гарантирует, что задачи по обслуживанию не затеряются на фоне работы с функционалом, и, в обратном смысле, сохранит правильное соотношение сложностей. Выделение всего 20 % времени на разработку чего-либо означает, что это никогда не станет важнее и сложнее разработки кода продакшена. Думаю, соотношение 80:20 (усилий, затраченных на создание кода, к усилиям на настройку вспомогательной инфраструктуры) достаточно разумно.

Для реализации этого решения есть множество способов. Можно прибавлять 20 % к предварительному плану на каждый тикет или просто отводить несколько часов в день. К сожалению, на моем опыте большинство небольших временных отрезков либо игнорируются, либо тратятся впустую ввиду переключения

контекста и прочей дополнительной нагрузки. В качестве альтернативы можно зарезервировать каждый пятый рабочий интервал для атаки на рефакторинг и работы по обслуживанию. Такой подход тоже нельзя назвать идеальным. Работа по настройке инфраструктуры в авральном варианте оказывается слишком напряженной и скучной, поскольку и разработчики, и стейкхолдеры обычно недополучают ощущение прогресса, которого так не хватает во время подобных спринтов. Еще один недостаток второго способа в том, что за время четырех предшествовавших интервалов код становится более зацепленным и запутанным, что усложняет работу с ним.

Наиболее успешным вариантом реализации этой техники из тех, с которыми я сталкивался, было резервирование пятниц для работы вне тикетов, то есть над всем, что не определяется запросом стейкхолдера. В такие пятницы разработчики экспериментируют, проводят крупный рефакторинг или выполняют задачи по автоматизированному тестированию, чтобы сократить затраты и повысить качество. Одного дня как раз достаточно для реализации существенных задач без негативной дополнительной нагрузки на общий процесс. Зачастую такие дни оказываются желанной передышкой от рутинной работы с тикетами, что оказывает оживляющий эффект на команду.

10.4. ПРЕОДОЛЕНИЕ СТРАХА ПЕРЕД НЕУДАЧЕЙ ЗА СЧЕТ ПОСТЕПЕННОЙ РАЗРАБОТКИ

Синдром самозванца — это когда человек чувствует себя неквалифицированным и боится, что кто-нибудь обвинит его в этом и назовет самозванцем. В нашей индустрии это ощущение не редкость, хотя такой страх иррационален и почти всегда оказывается безосновательным. При этом он сильно сказывается на продуктивности, так как из стремления защитить себя мы стремимся сделать код идеальным, чтобы не к чему было придраться. Написать идеальный код для непростой задачи крайне сложно, если вообще возможно, а в результате такого перфекционизма программист либо впадает в прокрастинацию, либо берется только за простые задачи.

Разработчики иногда неосторожны в необдуманной критике чужого кода. Зачастую можно слышать, как они жалуются на юзабилити чьего-то решения, его производительность, стабильность, архитектуру, да что угодно. Когда такое слышишь, то невольно думаешь про себя: «Неужели кто-то говорит такое и о моем коде?» либо «А что, если этот парень увидит мой код? А может, он на него и смотрит?». Такие размышления лишь усиливают в нас проявления синдрома самозванца.

Я утратил веру в идеальный код. Повышение эффективности ПО требует навыков и профилирования. Повышение простоты в использовании требует

тестирования и экспериментирования. Повышение расширяемости требует рефакторинга и прогнозирования. А увеличение стабильности требует тестирования или типизации. Все это занимает время. А ведь зачастую не менее важным оказывается еще одно свойство: стоимость производства. Это означает, что нам нужно выбирать, на чем делать акцент и когда можно допустить и принять некоторое несовершенство.

Как правильно расставить акценты, учитывая, что при написании кода нужно иметь в виду столько требований и то, что невозможно оптимизировать его по всем им сразу? Я лично выделил среди всех метрик-требований одну наиболее существенную: оптимизация под жизнь разработчика. Это значит, что нужно стараться затратить как можно меньше времени на то, чтобы перейти от получения задания к какому-то готовому рабочему решению. В таком случае мы сможем больше времени своей жизни уделять любимому делу, например писать больше кода.

Оптимизация под жизнь разработчика имеет и дополнительные выгоды: максимизирует практику и минимизирует промежуток времени до получения первых откликов от тестов, тестировщиков, стейкхолдеров и пользователей. Работа с короткими циклами обратной связи повышает качество продукта, потому что обратная связь помогает нам правильно направлять усилия по улучшению наших решений. Неважно, откуда мы начинаем. Если мы будем совершенствовать продукт быстрее конкурентов, то в конечном счете их превзойдем.

Эта же философия стоит за предложенным Дэном Нортон шаблоном «Отрыв и стабилизация», о котором шла речь в главе 9. В этом шаблоне мы рассматриваем задачу в отрыве и создаем код без учета метрик. Затем мы добавляем к нему мониторинг и спустя шесть недель видим, используется ли он. Если не используется, то удаляем. В противном случае переписываем уже на основе собранных за время мониторинга данных. Здесь мы делаем оптимизацию под жизнь разработчика, потому что тратим время на написание лишь такого кода, который используется. Помимо этого, у нас есть обратная связь, которая направляет наши усилия в нужное русло, в результате чего мы уже не тратим времени на соответствие ненужным метрикам.

10.5. КАК КОПИПАСТ ВЛИЯЕТ НА СКОРОСТЬ

В этой книге мы уже несколько раз рассматривали один из основных способов добавления кода: его повторение. В качестве наиболее примечательных случаев можно вспомнить дублирование кода `draw` во всех классах `Tile` в разделе 4.3 и кода `update` в `Stone` и `Box` в разделе 5.4. В случае с `draw` мы пришли к выводу, что сходство оказалось случайным и что эти коды должны различаться, поэтому оставили фрагмент повторяющимся. Для реплик кода в конечном итоге были приняты разные решения. В случае же с `update` мы сделали вывод, что коды

связаны и должны быть скомпонованы вместе, поэтому фрагменты объединили. Повторение кода — это метод, позволяющий содействовать или препятствовать расхождению в коде. Но здесь нужно учитывать два важных свойства этого процесса. Разберем их по очереди.

Во-первых, когда мы делаем код общего доступа, можем легко повлиять на все места его использования. Это означает, что мы можем быстро внести в поведение глобальные изменения. Однако поменять его только в одном месте вызова оказывается не так просто и не так однозначно: изменения отражаются во всех точках вызова. С другой стороны, если мы разделяем доступ к коду путем создания его копий и организации вызова из каждой точки своей отдельной реплики кода, то каждая точка вызова отделяется, а значит, можно без проблем изменять их по отдельности. Но только теперь оказывается, наоборот, сложно повлиять на поведение глобально, поскольку сам код придется обновлять во всех местах. Итак, делая один код общего доступа, мы повышаем скорость внесения изменений в его глобальное поведение, в то время как создание его копий повышает скорость изменения локального поведения в месте вызова.

Во-вторых, быстрота глобального изменения поведения означает, что мы можем повлиять на множество разных мест одновременно. В главе 2 мы определили хрупкость как особенность системы, которая при внесении изменений склонна ломаться в совершенно несвязанных местах кода. Можно легко представить, что каждая точка вызова общей функции имеет разные локальные инварианты. Когда мы изменяем этот общий код, то рискуем нарушить любое число этих инвариантов, поскольку для этого кода они не являются локальными. Таким образом, использование общего кода повышает *хрупкость* системы.

Но иногда высокая скорость изменения глобального поведения может оказаться очень кстати, благодаря ей наш код сможет при необходимости быстро адаптироваться. Повышенная хрупкость системы оказывается нашей платой за это, как и риск внесения в общий код неверных изменений, способных навредить глобально. Эти два недостатка усиливают необходимость в тестировании, испытании или мониторинге.

Поскольку скопированный код полностью отделен, то с ним можно смело экспериментировать и изменять его без опасений, так как в этом случае мы не рискуем сломать что-либо где-то еще. Во время реализации отрыва я призываю прибегать к дублированию как можно чаще, потому что это позволяет быстро проверять различные предположения. Это верно даже во время шестинедельного периода «отрыва и стабилизации». Когда код будет настроен и мы еще о нем не успеем забыть, нужно вернуться и проверить, есть ли смысл объединить его с источником копии, используя шаблон рефакторинга из главы 5. Иначе говоря, необходимо ставить такие вопросы: «Нужно ли его связать с источником? Когда он изменяется, должен ли меняться источник? Владеет ли моя команда этим объединенным кодом?» Если ответом хоть на один из этих вопросов будет «нет», код наверняка должен остаться разделенным.

10.6. ИЗМЕНЕНИЕ ПУТЕМ ДОБАВЛЕНИЯ ЧЕРЕЗ РАСШИРЯЕМОСТЬ

Еще один вариант добавления кода подразумевает использование расширяемости. Если нам известно, что определенный код восприимчив к изменениям, можно сделать его расширяемым. Это означает вынесение его вариаций в отдельные классы. В этом случае добавить новую вариацию окажется так же просто, как создать очередной класс. Если наша область достаточно постоянна, то участки, которые обычно изменяются, должны со временем становиться все более и более восприимчивыми к дальнейшему изменению.

Точки добавления вариативности усложняют код. В результате логика его выполнения становится менее понятной, что затрудняет внесение изменений в будущем. Получается, что внесение расширяемости может оказаться слишком затратным, поскольку это без необходимости усложнит код. А так как код отражает явления из реальной жизни, то часть сложности наследуется как раз от прикладной области и называется *неотъемлемой сложностью*. А сложность, которая не является выражением внутренней предметной области, называется *ненужной сложностью*.

Для ограничения ненужной сложности нам следует откладывать введение точек вариативности до того момента, когда они потребуются. На протяжении этой книги везде, где у нас была вариация, мы следовали одному и тому же процессу из трех шагов.

1. Дублировали код.
2. Прорабатывали его и адаптировали.
3. Если в том был смысл, то объединяли этот код с его источником.

Этот сценарий дает нам много свободы при работе с кодом, так как он оказывается отделенным от остальной части программы. По завершении его обработки можно легко объединить отделенный код с исходником, чтобы сделать структуру более явно выраженной.

Этот рабочий поток напоминает еще один распространенный шаблон «Расширение — контракт». Название отражает две самые короткие его фазы. Всего в этом шаблоне их три, и они очень похожи на вышеописанный процесс.

1. На фазе расширения мы добавляем новую функциональность. Это вполне безопасно, так как мы просто делаем добавление. Но теперь у нас получается две копии одного поведения, которые нужно поддерживать.
2. Мы выполняем перенос, постепенно переводя клиентов на новую функциональность. Это самая длинная фаза.
3. В заключительной фазе — контракте — после перемещения всех клиентов удаляем исходную версию поведения.

В этой книге мы видели два существенных способа придания коду расширяемости: «Замена кода типа классами» (4.1.3) и «Введение паттерна “Стратегия”» (5.4.2). Оба этих шаблона преобразуют статическую структуру в динамическую. «Замена кода типа классами» преобразует статический поток управления, представленный в виде `if` и `switch`, в вызовы методов в интерфейсе. Поток управления через интерфейс является динамическим, поскольку его без проблем можно в любой момент расширить, то есть у нас есть возможность изменять поведение простым добавлением реализующего класса. «Введение паттерна “Стратегия”» объединяет две копии фрагмента кода, позволяя нам добавлять новые копии динамически через добавление новых стратегий.

10.7. ИЗМЕНЕНИЕ ПУТЕМ ДОБАВЛЕНИЯ ПОДДЕРЖИВАЕТ ОБРАТНУЮ СОВМЕСТИМОСТЬ

Нередко мы раскрываем функциональность внешне через публичные интерфейсы или API. Если люди полагаются на наш код, то мы оказываемся ответственны за их защиту от непредвиденных побочных эффектов при его обновлении. Стандартным решением в данном случае является версионирование. При создании новой версии кода пользователям необходимо предоставить возможность продолжить использовать знакомый им старый вариант без опасений, что он будет изменен. То есть обеспечить обратную совместимость кода.

ОБЯЗАТЕЛЬСТВО MICROSOFT ПО ОБЕСПЕЧЕНИЮ ОБРАТНОЙ СОВМЕСТИМОСТИ

Как известно, компания Microsoft прилагает невероятные усилия по обеспечению обратной совместимости, что в немалой степени обусловило ее успех. В серии видео *One Dev Question with Raymond Chen* Раймонд Чен описывает, как код из Windows 95 по-прежнему используется в Windows 10. Осознание того факта, что код, которому больше 20 лет, все еще работает, может весьма впечатлить. В доступном на YouTube видео *Why You Can't Name a File CON in Windows* Том Скотт демонстрирует мой любимый пример — успешно работающий поиск окна выбора файла из Windows 3.1 в современных системах Windows 10.

- Нажмите **Start**, введите **ODBC** и нажмите **ODBC Data Source Administrator (32-bit)**.
- Нажмите кнопку **Add**.
- Выберите **Driver Do Microsoft Access (*.mdb)** и нажмите **Finish**.
- Нажмите кнопку **Select**.

Мне нравится напоминать людям, что безопаснее всего просто ничего в коде не менять. Несмотря на то что это отчасти шутка, в ней кроется глубокое

наблюдение. Если мы действительно стремимся предлагать клиентам максимальную безопасность, наш код должен сохранять обратную совместимость на протяжении всего срока своей службы. То есть при любом внесении изменений появляется в публичном интерфейсе новый метод, новая конечная точка в API или новое событие в событийной системе. Исходная функциональность метода при этом остается нетронутой.

Вести разработку таким образом на удивление легко. Мы просто следуем тому же процессу, что описывался ранее. Начинаем с дублирования существующей конечной точки, которую хотим изменить. Далее реализуем нужные изменения, будучи уверенными, что они не затронут окружения. Затем производим объединение с кодом исходной конечной точки. Поначалу добавляется ненужная сложность, причем «ненужность» обусловлена несовершенством предыдущей версии 1.0. Чтобы избавиться от этой сложности, нам нужно постараться перевести людей на новую версию, пояснив ее преимущества перед старой, адаптировав под нее руководства и громко заявив об изменениях. Аналогично тому, как мы работали с legacy-кодом в предыдущей главе, нам следует сопровождать исходную версию мониторингом. Когда по его результатам мы увидим, что эту версию уже никто не использует, можно будет безопасно ее удалить.

Здесь все еще остается вопрос о том, как указать, какую версию использовать. Поскольку я склоняюсь к самым простым решениям, то рекомендую размещать информацию о версиях в именах точек входа. Обратите внимание, что мы версионируем только внешний слой: интерфейс между нами и пользователями. Нам не нужно версионировать методы, которые контролируются, поскольку их можно протестировать и убедиться, что все в порядке. С пользовательским же кодом этого сделать нельзя. Рекомендуется использовать последовательную схему именования, чтобы можно было легко понять, какая версия новее. Пример того, как не нужно называть функции, можно найти в РНР при поиске способа очищения входных данных для SQL.

Листинг 10.1. Три версии экранирования строки в РНР

```
mysql_escape_string
mysql_real_escape_string
mysqli_real_escape_string
```

10.8. ИЗМЕНЕНИЕ ПУТЕМ ДОБАВЛЕНИЯ С ПОМОЩЬЮ ПЕРЕКЛЮЧАТЕЛЕЙ ФУНКЦИОНАЛА

Слияние нашего кода с кодом коллег называется *интеграцией*. Мы знаем, что, если интегрировать код часто и небольшими частями, это сократит ошибки и в итоге сэкономит нам время, а также избавит от ужасных конфликтов слияния. Лучше уж проделывать это несколько раз в день, или непрерывно практикуя ансамблевое программирование. Тут же появляются вопросы вроде:

«Что, если код еще не готов?» или «Что, если пользователи не готовы к новой функциональности?».

Такой ход мысли типичен, когда мы думаем о развертывании кода как о его релизе. В базе кода вполне могут находиться его части, которые не выполняются. Можно даже его развернуть без ведома остальных. Самый простой способ сделать код игнорируемым — это поместить его в `if (false)`. Тогда станет возможно без опаски добавлять что угодно и до тех пор, пока это будет успешно компилироваться, можно так же безопасно интегрировать данный фрагмент в основную ветку или даже его развернуть. Но имейте в виду, что здесь есть минимальное требование: он должен компилироваться без ошибок.

На этой идее основано то, что называется *переключением функционала*. Для обработки этой возможности созданы чрезвычайно сложные системы, но в качестве начальной точки я всегда советую на время обучения использовать самый простой вариант. Новые принципы требуют практики, а использование готовых инструментов может оказаться отвлекающим и в то же время непосильным. Я рекомендую начинать применение переключателей функционала, выполняя такую последовательность операций.

1. Если он еще не существует, создать класс `FeatureToggle`.

Листинг 10.2. Новый класс

```
class FeatureToggle {
}
```

2. Добавить для решаемой задачи статический метод, возвращающий `false`. Он называется *флагом функциональности (feature-флагом)*. В нашем примере это `featureA`.

Листинг 10.3. До

```
class FeatureToggle {
}
```

Листинг 10.4. После (1/4)

```
class FeatureToggle {
    static featureA() { return false; }
}
```

Новый feature-флаг

3. Найти место, где нужно реализовать изменение. Добавить туда пустую `if (FeatureToggle.featureA())` и заключить существующий код в `else`.

Листинг 10.5. До

```
class Context {
    foo() {
        code();
    }
}
```

Исходный код, без изменений

Листинг 10.6. После (2/4)

```
class Context {
    foo() {
        if (FeatureToggle.featureA()) {
        } else {
            code();
        }
    }
}
```

Новая if (false)

Исходный код, без изменений

4. Продублировать код из `else` в `if`.**Листинг 10.7.** До

```
class Context {
    foo() {
        if (FeatureToggle.featureA()) {
        } else {
            code();
        }
    }
}
```

Листинг 10.8. После (3/4)

```
class Context {
    foo() {
        if (FeatureToggle.featureA()) {
            code();
        } else {
            code();
        }
    }
}
```

Тот же код

5. Внести желаемые изменения в код внутри `if`. Когда новый код будет подготовлен для тестирования, необходимо изменить `FeatureToggle.featureA` на возвращение значения переменной среды: `false`, если такая переменная не существует.

Листинг 10.9. До

```
class FeatureToggle {
    static featureA() {
        return false;
    }
}
```

Листинг 10.10. После (4/4)

```
class FeatureToggle {
    static featureA() {
        return Env.isSet("featureA");
    }
}
```

Feature-флаг
использует среду

Теперь на локальной машине можно установить переменную для тестирования нового функционала, но сам он по-прежнему останется не виден для других. Можно безопасно развернуть код и, когда клиенты будут готовы, просто установить переменную среды в продакшене, чтобы он запустился. Становится возможным, работая таким образом, интегрировать и разворачивать код так часто, как того потребует разработка. Тем не менее и здесь нужно учитывать некоторые подвохи.

Первый состоит в том, что если забыть осуществить этот процесс или выполнить его некорректно, то можно отправить что-нибудь в продакшен случайно. Как разработчик, который гордится своей работой, я мало чего боюсь так же панически, как утраты контроля над тем, что отправляется в продакшен. Это одна из причин, по которым я рекомендую использовать именно простую версию переключения функционала. Упрощенный процесс снижает риск ошибки. В начале можно просто добавить переключение функционала как завершающий штрих стандартного рабочего потока и внести в процедуру развертывания установку всех переменных среды сразу. В результате процесс будет аналогичен нашим стандартным развертываниям. Таким образом можно практиковаться и совершенствовать свои навыки, не раскрывая для пользователей вносимых изменений, но в то же время обеспечивая возможность отката функционала, который

корректно переключается без повторного развертывания. Это, в свою очередь, является еще одним преимуществом приведенной техники.

Еще один подвох в том, что теперь у нас выполняется две копии одного кода. Более того: у нас есть `if`. Как уже говорилось, `if` добавляют фактическую сложность, и если у нас возникает зависимый функционал, то необходимо поместить `if` в обе ветки. Такой подход быстро приводит к разрастанию кода, и ПО становится неуправляемым. Но эти `if` особенные. Они являются техническим долгом, потому что присутствуют здесь временно. Каждый раз, когда закрывается задача, породившая переключение функционала с помощью `if`, необходимо создавать запланированное задание для удаления этого переключения. И здесь я снова рекомендую устанавливать срок запуска этого задания на выполнение в промежутке до шести недель. На этот раз, если функционал будет активирован в продакшене, мы удалим часть `else`. В противном случае мы удалим часть `if`. Это может означать, что код так и не увидел свет, но в таком случае мы рассматриваем его как замороженный проект и удаляем из основной ветки. Нельзя допускать, чтобы feature-флаги застаивались, поскольку они будут загрязнять базу кода и могут вызвать катастрофические сбои.

ИНЦИДЕНТ С ПЕРЕКЛЮЧАТЕЛЯМИ В KNIGHT

В своем посте *Knightmare: A DevOps Cautionary Tale* (<http://mng.bz/dm0w>) Дэг Севен рассказывает, что в 2012 году быстрорастущая трейдинговая компания Knight выпустила новую версию программного обеспечения. В тот день сошлось сразу несколько событий, сделав его, вероятно, худшим днем за всю историю компании. Поскольку развертывание выполнялось вручную одним инженером, то ПО развертывалось не на всех серверах, поэтому одновременно работали две версии. Аварийный выключатель в системе реализован не был, как и процесс для обработки ситуаций, в которых что-то могло пойти не так. Единственной сработавшей функцией защиты оказалось предупреждающее письмо по электронной почте, которое было успешно проигнорировано.

Такой подход к организации работы был рискованным изначально, но сам по себе он не вызвал проблемы. Лавинный процесс начался из-за несовместимости двух работающих версий. Новый код переназначил флаг конфигурации, который не использовался уже несколько лет. К сожалению, привязанный к этому флагу код все еще находился в базе и выполнялся на некоторых серверах. Это привело к тому, что программа начала вести спонтанные неуправляемые торги. В результате компания Knight потеряла около 400 миллионов долларов за 45 минут, которые потребовались на то, чтобы остановить программу.

Когда два обозначенных подвоха будут проработаны и появится крепкая уверенность, что переключения везде выполнены корректно, надлежит должным образом их удалить. И только потом — но ни в коем случае не раньше — можно перейти к дальнейшему освоению этой превосходной технологии. Первым шагом, вероятно, будет перемещение переключателей в базу данных и создание для них небольшого UI, чтобы бизнес мог включать/отключать нужные компоненты. Можно также предусмотреть медленное внедрение: сначала новый функционал увидят только 10 % пользователей, что позволит убедиться в его корректной работе, после чего уже постепенно увеличивать число таких пользователей. Можно пойти еще дальше и привязать переключатели к какой-нибудь метрике вроде «Купил ли пользователь что-либо?»: если количество покупающих пользователей будет расти, то внедрение будет резонно ускорить. Эта техника называется *A/B-тестированием*, она может оказаться очень выгодной.

A/B-ТЕСТИРОВАНИЕ САЙТА ПРЕДВЫБОРНОЙ КАМПАНИИ ОБАМЫ

Во время проведения выборов президента США в 2008 году на сайте кампании Барака Обамы требовалось добавить фото и кнопку **Sign Up** (Подписаться). Заранее никак нельзя было угадать, какие фото или текст кнопки сработают лучше всего, поэтому его команда решила на эксперимент. С помощью метода A/B-тестирования они показывали одну комбинацию одним посетителям и другую — другим. В результате было замечено, что фото Обамы с семьей имело наилучший эффект вместе с кнопкой **Learn More** (Узнать больше). Такой комплексный подход показал на 40 % лучший результат по сравнению с периодом до A/B-тестирования, что, по приблизительным оценкам, привело к увеличению взносов на 60 миллионов долларов (<http://mng.bz/rmxy>).

Имейте в виду, что такой подход в полной мере применим и к теме из предыдущей главы, потому что данный алгоритм автоматически отсеивает код, который оказывается менее выгодным или ошибочным. После чего остается лишь вручную выполнить фактическое удаление, посмотрев, включены ли флаги.

10.9. ИЗМЕНЕНИЕ ПУТЕМ ДОБАВЛЕНИЯ С ПОМОЩЬЮ ВЕТВЛЕНИЯ ЧЕРЕЗ АБСТРАГИРОВАНИЕ

Сейчас вполне уместно будет задать законный вопрос: «А разве переключение функционала не нарушает правило “Никогда не использовать `if c else`” (4.1.1)?» Действительно, нарушает, и разрешить эту ситуацию можно двумя способами. Самый простой — он лежит на поверхности — это сказать, что `if` используются временно. Их несложно удалить, что мы и планируем, и они не должны

разрастаться, что является основной проблемой конструкций `if`. Я прибегаю к этому оправданию, если `feature`-флаг используется только в одном месте.

Иногда функционал требует внесения изменения в нескольких местах кода. Значит, если мы используем несколько `if`, то связанные с этим изменением инварианты распространятся на них. В таких случаях, прежде чем отправлять код, я применяю «Замену кода типа классами» для булева элемента внутри `feature`-флага. Вместо возвращения `true` либо `false` я возвращаю `NewA` или `OldA`. Этот прием обычно называется *ветвлением через абстрагирование*: классы представляют собой абстракцию, а наличие именно двух классов (больше одного) является ветвлением. Представление в виде классов позволяет избавиться от `if` перемещением их в эти классы, как мы уже неоднократно делали в части I книги.

В данном примере показаны две версии одной программы: в одной используются переключатели функционала, а во второй — ветвление через абстрагирование.

Листинг 10.11. Переключатели функционала

```
class FeatureToggle {
    static featureA() {
        return Env.isSet("featureA");
    }
}
class ContextA {
    foo() {
        if (FeatureToggle.featureA()) {
            aCodeV2();
        } else {
            aCodeV1();
        }
    }
}
class ContextB {
    bar() {
        if (FeatureToggle.featureA()) {
            bCodeV2();
        } else {
            bCodeV1();
        }
    }
}
```

Листинг 10.12. Ветвление через абстрагирование

```
class FeatureToggle {
    static featureA() {
        return Env.isSet("featureA")
            ? new Version2()
            : new Version1();
    }
}
class ContextA {
    foo() {
        FeatureToggle.featureA().aCode();
    }
}
class ContextB {
    bar() {
        FeatureToggle.featureA().bCode();
    }
}
interface FeatureA {
    aCode(): void;
    bCode(): void;
}
class Version1 implements FeatureA {
    aCode() { aCodeV1(); }
    bCode() { bCodeV1(); }
}
class Version2 implements FeatureA {
    aCode() { aCodeV2(); }
    bCode() { bCodeV2(); }
}
```

Данный подход локализует инварианты изменения функциональности в этих классах. Затем, когда приходит время удалять переключатель, поступаем так.

1. Удаляем один из классов.

Листинг 10.13. До

```
class Version1 implements FeatureA {
    aCode() { aCodeV1(); }
    bCode() { bCodeV1(); }
}
class Version2 implements FeatureA {
    aCode() { aCodeV2(); }
    bCode() { bCodeV2(); }
}
```

Листинг 10.14. После (1/4)



```
class Version2 implements FeatureA {
    aCode() { aCodeV2(); }
    bCode() { bCodeV2(); }
}
```

2. Затем, следуя правилу «Избегать интерфейсов с единственной реализацией» (5.4.3), также удаляем интерфейс.

Листинг 10.15. До

```
interface FeatureA {
    aCode(): void;
    bCode(): void;
}
```

Листинг 10.16. После (2/4)



3. В завершение встраиваем методы в оставшийся класс и feature-флаг.

Листинг 10.17. До

```
class ContextA {
    foo() {
        FeatureToggle.featureA().aCode();
    }
}
class ContextB {
    bar() {
        FeatureToggle.featureA().bCode();
    }
}
class Version2 implements FeatureA {
    aCode() { aCodeV2(); }
    bCode() { bCodeV2(); }
}
```

Листинг 10.18. После (3/4)



```
class ContextA {
    foo() {
        aCodeV2();
    }
}
class ContextB {
    bar() {
        bCodeV2();
    }
}
class Version2 implements FeatureA {
    aCode() { aCodeV2(); }
    bCode() { bCodeV2(); }
}
```

4. Далее также удаляем этот класс.

Листинг 10.19. До

```
class Version2 implements FeatureA {
    aCode() { aCodeV2(); }
    bCode() { bCodeV2(); }
}
```

Листинг 10.20. После (4/4)



У нас остался следующий код, лишенный всяческих следов переключения функционала.

Листинг 10.21. После

```
class FeatureToggle {  
}  
class ContextA {  
    foo() {  
        aCodeV2();  
    }  
}  
class ContextB {  
    bar() {  
        bCodeV2();  
    }  
}
```

РЕЗЮМЕ

- Внедрение в рабочий поток отрывов помогает преодолеть страх создать что-то не так.
- Необходимо согласиться на некоторые дополнительные затраты, чтобы большую часть времени можно было посвятить реализации действительно актуальных для стейкхолдеров задач.
- Установка основного ориентира производственного процесса на удобство для разработчика — оптимизация под жизнь программиста — увеличивает практичность и продуктивность программирования.
- Дублирование кода способствует безопасному экспериментированию с его фрагментами, в то время как его совместное использование повышает хрупкость.
- Более обширное тело кода лучше раскрывает внутреннюю структуру ПО и дает нам более точный ориентир для рефакторинга.
- Цель рефакторинга — сократить ненужную сложность. Неотъемлемая же сложность является необходимой для осмысленного моделирования лежащей в основе предметной области.
- Изменение путем добавления поддерживает обратную совместимость версий систем, сокращая риски.
- Переключатели функционала поддерживают интеграцию кода, что также снижает риски.
- Ветвление путем абстрагирования помогает в работе со сложными переключателями функционала.

11

Соблюдение структуры в коде

В этой главе

- ✓ Кодирование поведения в потоке управления.
- ✓ Перемещение поведения в структуры данных.
- ✓ Использование данных для кодирования поведения.
- ✓ Определение неиспользуемых структур в коде.

Программное обеспечение — это модель некоего аспекта реального мира. Реальный мир по мере нашего роста и обучения изменяется, и нам нужно адаптировать свои программы, чтобы они шли в ногу с этими переменами. Получается, что до тех пор, пока программное обеспечение используется, оно никогда не оказывается законченным. Ну и коль скоро программная модель мира должна быть точной и выражать закодированную структуру реальности, код должен отражать связи, существующие в реальном мире.

В этой главе мы сначала разберем, откуда происходят различные типы структур кода. Затем познакомимся с тремя способами кодирования поведения и узнаем, как переключать поведение между этими способами. Определив типы структуры, с которой работаем, перейдем к рефакторингу: постараемся понять, когда он оказывается кстати, а когда может навредить. В завершение примем в рассмотрение разные типы неиспользуемых структур и разберем, как их задействовать с помощью изученных нами шаблонов рефакторинга.

11.1. КАТЕГОРИЗАЦИЯ СТРУКТУРЫ НА ОСНОВЕ ОБЛАСТИ И ИСТОЧНИКА

В разработке ПО мы имеем дело с несколькими видами структур (то есть узнаваемых паттернов). Структура может представлять два схожих метода или нечто, что люди выполняют каждый день. Структура присутствует в предметной области, она есть в поведении программы, в нашем взаимодействии и, конечно, в коде.

Я предпочитаю разделять область структуры на четыре категории, они сведены в таблицу ниже: измерение по горизонтали показывает, влияет ли структура на одну команду или человека (внутрикомандная) либо на несколько команд или нескольких людей (межкомандная). Другое измерение, по вертикали, показывает, относится структура к коду или к человеку (табл. 11.1).

Таблица 11.1. Категории пространства структуры

	Межкомандная	Внутрикомандная
В коде	Внешний API	Данные и функции, большая часть рефакторинга
Среди людей	Штатное расписание, процессы	Поведение, эксперты в области

Макроархитектура относится к межкомандной структуре: она дает понимание, чем является наш продукт и как другой код с ним взаимодействует. Она устанавливает то, как должен выглядеть наш внешний API, какими данными располагает каждая команда. Макроархитектура определяет наши программные платформы.

Микроархитектура относится уже к внутрикомандной структуре: это сведения о том, что может сделать команда для привнесения ценности, какие сервисы мы используем, как организовать данные и как выглядит код. Приводимые в данной книге шаблоны рефакторинга относятся именно к этой категории.

Не забудем, что мы работаем в рамках процессов и определенной иерархии внутри компании. Здесь понятие «*процессы*» относится к Scrum, Kanban, моделям проектов и т. д. «*Иерархия*» означает принципы, которые устанавливают, какая команда и с кем должна взаимодействовать.

Наконец, есть структура, которую определяют эксперты предметной области. Такие эксперты знакомы с шаблонами предметной области, так как эти шаблоны повторяют в конечном счете их поведение. Эти эксперты определяют то, как должно функционировать ПО, то есть система в итоге должна отражать их поведение.

Удивляет же здесь то, что структура организации склонна проецироваться вдоль горизонтального измерения и определять, как выглядит внешний API. Это называется *законом Конвея*. Аналогичным образом структура в поведении

экспертов предметной области склонна просачиваться в код. Это одновременно и поражает, и оказывается кстати, так как если мы замечаем неэффективные моменты в коде, то зачастую можем отыскать их реальный источник в работе экспертов, в процессах либо еще в чем-то. Понимание этого может стать очень мощным инструментом для совершенствования разработанного кода.

Я об этом говорю, потому что пользовательское поведение тоже влияет на структуру кода. А некоторые изменения в структуре кода требуют изменения в пользовательском поведении. Мы можем рассматривать пользователей как еще одну часть кода. Если у нас нет возможности взаимодействовать с ними, то они являются внешними. Следовательно, при оценке с позиции рефакторинга получается, что они нас ограничивают. Если же существует возможность переобучить наших пользователей, то они находятся в области рефакторинга. Помните: хотя изменение поведения людей кажется более простым, чем изменение кода, в больших организациях или среди обширных баз пользователей это зачастую оказывается сложным и медленным процессом. Поэтому чаще оказывается целесообразно сначала смоделировать пользовательское поведение как оно есть, включая все его слабые стороны, а затем уже постепенно предоставлять более эффективную функциональность кода, сопровождая ее обучением. Таким образом, станет возможно проводить рефакторинг пользовательского поведения.

11.2. ТРИ СПОСОБА, КОТОРЫМИ КОД ОТРАЖАЕТ ПОВЕДЕНИЕ

Независимо от источника поведения встроить его в код можно тремя способами:

- в поток управления;
- в структуру данных;
- в сами данные.

Далее мы разберем каждый из них. Чтобы обозначить различия между ними, я продемонстрирую с их использованием известную программу FizzBuzz. Я также покажу, как кодировать бесконечные циклы, поскольку они представляют собой интересный особый случай. Имейте в виду: так как рефакторинг не изменяет поведение, то мы либо реализуем повторение, либо переносим структуру из одного подхода в другой.

11.2.1. Выражение поведения в потоке управления

Поток управления выражается в тексте кода через операторы управления, вызовы методов или просто строками кода. В качестве примера приведу один и тот же цикл, использующий эти три наиболее распространенных типа потока управления.

Листинг 11.1. Оператор управления

```
let i = 0;
while (i < 5) {
  foo(i);
  i++;
}
```

Листинг 11.2. Вызов метода

```
function loop(i: number) {
  if (i < 5) {
    foo(i);
    loop(i + 1)
  }
}
```

Листинг 11.3. Строки

```
foo(0);
foo(1);
foo(2);
foo(3);
foo(4);
```

Каждый раз, когда мы говорим о повторении кода, мы почти всегда имеем в виду перемещение между этими тремя подкатегориями поведения и чаще всего в направлении от самого правого из них, а именно от строк. Эти три подкатегории имеют тонкие различия. Вызовы метода и строки могут выражать нелокальную структуру, в то время как цикл может действовать лишь локально.

Листинг 11.4. Вызов метода

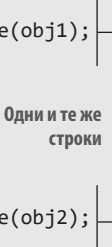
```
function insert(data: object) {
  let db = new Database();
  let normalized = normalize(data);
  db.insert(normalized);
}
function a() {
  // ...
  insert(obj1);
  // ...
}
function b() {
  // ...
  insert(obj2);
  // ...
}
```



Один и тот же вызов метода

Листинг 11.5. Строки

```
function a() {
  // ...
  let db = new Database();
  let normalized = normalize(obj1);
  db.insert(normalized);
  // ...
}
function b() {
  // ...
  let db = new Database();
  let normalized = normalize(obj2);
  db.insert(normalized);
  // ...
}
```



Одни и те же строки

ЗНАКОМСТВО С FIZZBUZZ

FizzBuzz — это детская игра на изучение таблицы умножения. Вы выбираете два числа, после чего игроки один за другим последовательно называют числа. Если очередное число в последовательности делимо на ваше первое число, то ребенок говорит Fizz. Если оно делимо на второе ваше число, то он говорит Buzz. Если же оно делимо на оба числа, то произносится FizzBuzz. Игра продолжается до тех пор, пока кто-нибудь не допустит ошибку.

Реализация игры в виде кода принимает такую форму: написать программу, получающую на входе число N и выводящую все числа от 0 до N . Но при этом если число окажется делимым на 3, то выводить вместо него Fizz. Если оно окажется делимым на 5, выводить Buzz. А если число будет делимо на оба этих значения, выводить FizzBuzz.

С другой стороны, операторы управления и вызовы метода способны выполнять то, чего не могут строки, — создавать бесконечные циклы.

Листинг 11.6. Оператор управления

```
for (;;) { }
```

Листинг 11.7. Вызов метода

```
function loop() {  
  loop();  
}
```

Работая с поведением в потоке управления, легко вносить существенные изменения, поскольку мы можем изменять поток простым перемещением инструкций. Зачастую мы предпочитаем стабильность и небольшие изменения, поэтому обычно извлекаем поведение из потока управления. Но в некоторых ситуациях требуется вносить серьезные корректировки. В подобных случаях может оказаться полезно встроить поведение в поток управления, затем внести изменения, после чего вынести поведение обратно.

Многие шаблоны рефакторинга в данной книге работают именно на этом уровне. К примерам можно отнести «Извлечение метода» (3.2.1) и «Совмещение инструкций if» (5.2.1).

Большинство людей предпочитают реализовать FizzBuzz, кодируя ее в поток управления.

Листинг 11.8. FizzBuzz с использованием потока управления

```
function fizzBuzz(n: number) {  
  for (let i = 0; i < n; i++) {  
    if (i % 3 === 0 && i % 5 === 0) {  
      console.log("FizzBuzz");  
    } else if (i % 5 === 0) {  
  
      console.log("Buzz");  
    } else if (i % 3 === 0) {  
      console.log("Fizz");  
    } else {  
      console.log(i);  
    }  
  }  
}
```

11.2.2. Выражение поведения в структуре данных

Еще один способ — кодирование поведения в структуре данных. Выше уже использовалось поэтическое сравнение структур данных с алгоритмами, застывшими во времени. Мой любимый пример — это связь между двоичной функцией поиска и структурой данных «двоичное дерево поиска» (BST).

Если не углубляться в детали, то двоичный поиск — это алгоритм поиска элемента в упорядоченном списке. Реализуется он путем повторяющегося деления

области поиска пополам — поскольку список упорядоченный, то по результату сравнения ключа поиска со средним элементом можно отбрасывать половину этого списка. BST — это структура дерева, состоящая из узлов. Каждый узел содержит значение и может иметь до двух прямых потомков. Инвариант (или поведение, вложенное в эту структуру данных) состоит в том, что все левые потомки меньше значения узла, а все правые — больше. При поиске элемента в BST мы сравниваем его со значением в корне, а затем рекурсивно спускаемся по соответствующему дочернему дереву. Получается, что в структуре BST выражается поведение двоичного поиска (рис. 11.1).

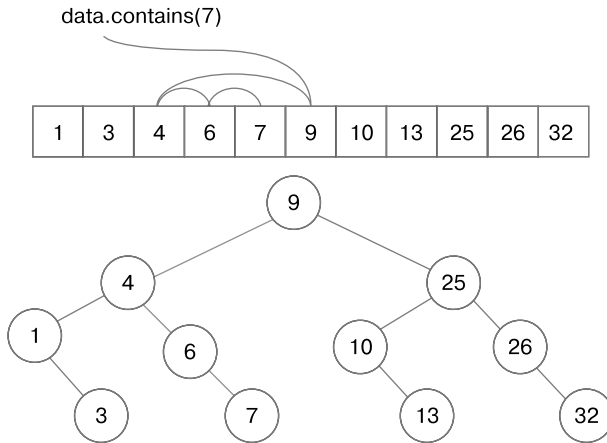


Рис. 11.1. Двоичный поиск и BST

Посмотрим, как можно использовать вместо `for`, `while` и рекурсивных функций типы для определения бесконечных циклов. В этом примере мы задействуем рекурсивную структуру данных. У `Rec` есть поле `f`, чей тип содержит `Rec`. В связи с этим данная структура и называется рекурсивной. Поскольку поле `f` является функцией, мы можем определить функцию `helper`, которая получает объект `Rec`, извлекает из него функцию и вызывает ее с тем же объектом `Rec`. Теперь можно инстанцировать объект `Rec` с функцией `helper` и передать его в функцию `helper`. Заметьте, что в этом примере никакая функция не вызывает себя напрямую: `helper` вызывает себя через структуру данных `Rec`.

Листинг 11.9. Рекурсивная структура данных

```
class Rec {
    constructor(public readonly f: (_: Rec) => void) {}
}

function loop() {
    let helper = (r: Rec) => r.f(r);
    helper(new Rec(helper));
}
```

Если сравнивать с поведением в потоке управления, то становится заметно, что с помощью структур данных стало сложнее вносить существенные изменения, если они не совпадают с существующими точками вариации. Зато с помощью этого подхода проще и быстрее вносить небольшие изменения. Причина в том, что мы получаем улучшенную безопасность типов и локализацию. В некоторых случаях можно получить и прирост быстродействия, когда структура данных позволяет нам кэшировать и многократно использовать информацию, как в примере сравнения двоичного поиска с BST. Шаблоны рефакторинга «Замена кода типа классами» (4.1.3) и «Введение паттерна “Стратегия”» (5.4.2) перемещают структуру из потока управления в структуры данных.

Кодирование FizzBuzz в структуру данных представляет трудоемкую задачу, потому что нам нужно выразить в коде циклическое поведение %. Также можно реализовать в виде структуры данных натуральные числа, чтобы избавиться от оператора управления `for`, но я оставляю это упражнение для вас. К счастью, код достаточно легко читаем.

Листинг 11.10. FizzBuzz с использованием структур данных

<pre>interface FizzAction { num(n: number): void; buzz(): void; } class SayFizz implements FizzAction { num(n: number) { console.log("Fizz"); } buzz() { console.log("FizzBuzz"); } } class FizzNumber implements FizzAction { num(n: number) { console.log(n); } buzz() { console.log("Buzz"); } }</pre>	Кодирование поведения fizz
<pre>interface BuzzAction { num(n: number, act: FizzAction): void; } class SayBuzz implements BuzzAction { num(n: number, act: FizzAction) { act.buzz(); } } class BuzzNumber implements BuzzAction { num(n: number, act: FizzAction) { act.num(n); } }</pre>	Кодирование поведения buzz
<pre>interface FizzNum { next(): FizzNum; action(): FizzAction; } class FizzNum1 implements FizzNum { next() { return new FizzNum2(); }</pre>	Кодирование % 3

```

    action() { return new FizzNumber(); }
}
class FizzNum2 implements FizzNum {
    next() { return new Fizz(); }
    action() { return new FizzNumber(); }
}
class Fizz implements FizzNum {
    next() { return new FizzNum1(); }
    action() { return new SayFizz(); }
}

interface BuzzNum {
    next(): BuzzNum;
    action(): BuzzAction;
}
class BuzzNum1 implements BuzzNum {
    next() { return new BuzzNum2(); }
    action() { return new BuzzNumber(); }
}
class BuzzNum2 implements BuzzNum {
    next() { return new BuzzNum3(); }
    action() { return new BuzzNumber(); }
}
class BuzzNum3 implements BuzzNum {
    next() { return new BuzzNum4(); }
    action() { return new BuzzNumber(); }
}
class BuzzNum4 implements BuzzNum {
    next() { return new Buzz(); }
    action() { return new BuzzNumber(); }
}
class Buzz implements BuzzNum {
    next() { return new BuzzNum1(); }
    action() { return new SayBuzz(); }
}

function fizzBuzz(n: number) {
    let f = new Fizz();
    let b = new Buzz();
    for (let i = 0; i < n; i++) {
        b.action().num(i, f.action());
        f = f.next();
        b = b.next();
    }
}

```

Кодирование %3

Кодирование %5

11.2.3. Выражение поведения в данных

Последний подход заключается в кодировании поведения в данных. Он наиболее сложен, потому что здесь мы быстро попадаем в мертвую зону проблемы останова (halt) (рассмотренной в разделе 7.1), когда мы не получаем никакой поддержки и сигналов от инструментов и компиляторов.

В нашей индустрии мы чаще всего встречаем структуру в данных при их дублировании. Это может вести к проблемам с согласованностью обращения к данным, особенно если эти данные изменяемы. Подобные сложности могут быть оправданы приростом в производительности. Однако и они могут стать источником ошибок и лишних затрат.

Чтобы создать бесконечный цикл посредством данных, нам нужно использовать ссылки в массивах TypeScript, Java и C#, при этом объекты тоже обрабатываются как ссылки. Идея состоит в том, чтобы поместить в память функцию, ищущую ссылку, которая будет являться ею самой, и вызывающую ее. Заметьте, что эта функция опять вызывает себя не напрямую, а косвенно, через кучу (heap).

Листинг 11.11. Рекурсивные данные

```
function loop() {  
  let a = [() => { }];  
  a[0] = () => a[0]();  
  a[0]();  
}
```

В отличие от двух других методов здесь мы не получаем поддержки от компиляторов, что делает работу с данным методом небезопасной. Одно из решений — это использование инструментов для извлечения данных и генерирование из этих данных структур данных. В результате мы дублируем поведение и должны либо поддерживать этот инструмент сами, либо добавлять стороннюю зависимость.

Поскольку с этой структурой так сложно работать, я обычно рекомендую стремиться преобразовывать ее в одну из двух других. Тем не менее в разделе 6.5.2 мы видели пример рефакторинга, состоявший в перемещении этой структуры из потока управления в данные.

Кодирование игры FizzBuzz в данные, вероятно, выглядит намного проще, чем ее кодирование в структуру данных. Отчасти из-за того, что мы вернулись к наличию циклического поведения в операторе %. Это означает, что циклическое поведение закодировано в потоке управления. Однако при желании мы можем реализовать его с помощью указателей или ссылок. Я оставляю это в качестве упражнения для самых упорных читателей.

Листинг 11.12. FizzBuzz с использованием структур данных

```
interface FizzAction {  
  num(n: number): void;  
  buzz(): void;  
}  
class SayFizz implements FizzAction {  
  num(n: number) { console.log("Fizz"); }
```

```

    buzz() { console.log("FizzBuzz"); }
  }
  class FizzNumber implements FizzAction {
    num(n: number) { console.log(n); }
    buzz() { console.log("Buzz"); }
  }
  interface BuzzAction {
    num(n: number, act: FizzAction): void;
  }
  class SayBuzz implements BuzzAction {
    num(n: number, act: FizzAction) {
      act.buzz();
    }
  }
  class BuzzNumber implements BuzzAction {
    num(n: number, act: FizzAction) {
      act.num(n);
    }
  }
  const FIZZ = [
    new SayFizz(),
    new FizzNumber(),
    new FizzNumber()
  ];
  const BUZZ = [
    new SayBuzz(),
    new BuzzNumber(),
    new BuzzNumber(),
    new BuzzNumber(),
    new BuzzNumber()
  ];
  function fizzBuzz(n: number) {
    for (let i = 0; i < n; i++) {
      BUZZ[i % BUZZ.length].num(i, FIZZ[i % FIZZ.length]);
    }
  }

```

Кодирование 3

Кодирование 5

11.3. ДОБАВЛЕНИЕ КОДА ДЛЯ РАСКРЫТИЯ СТРУКТУРЫ

При выполнении рефакторинга некоторые изменения упрощаются, а некоторые усложняются. Мы производим рефакторинг для поддержания конкретного вектора изменений: направления, в котором, на наш взгляд, движется ПО. Чем больше у нас кода, тем выше вероятность, что мы знаем этот вектор и то, как код склонен изменяться, поскольку имеем больше данных. Как говорилось в главе 1, если код не должен изменяться, то и причины для рефакторинга нет.

Рефакторинг укрепляет текущую структуру и делает ее более восприимчивой к похожим изменениям. Он помещает точки вариации в места, где мы видели или ожидаем эту вариацию. В стабильных (под)системах это оказывается бесценным, поскольку ускоряет разработку и повышает качество продукта. С другой стороны, в (под)системах с обширной функциональностью для нас важнее экспериментирование, а не укрепление.

Когда мы не уверены во внутренней структуре, нужно отложить задачи рефакторинга и сосредоточиться на ее корректности. Естественно, мы никогда не должны жертвовать продуктивностью команды, значит, хрупкость кода повышать нельзя. Нам по-прежнему нужно избегать нелокальных инвариантов. Когда мы откладываем рефакторинг, то должны инкапсулировать неотрефакторенный код, чтобы он не оказал непредвиденного воздействия на остальную базу. Но мы не должны добавлять точки вариации, потому что одновременно с обеспечиваемой ими простотой вариативности они привносят сложность — сложность, которая затрудняет экспериментирование и, что еще важнее, может скрывать другие структуры.

При реализации новых возможностей или подсистем неизбежно возникает неопределенность. В таких ситуациях есть смысл использовать вместо классов перечисления и циклы, потому что их можно быстро изменить. Новый код обычно подвергается интенсивному тестированию, поэтому риск внести ошибки, не перехватив их, снижается. Когда код достигает зрелости и структура становится более стабильной, стабилизироваться должен и сам код. С помощью рефакторинга мы должны подстраивать структуру. В результате основательность кода должна выражать то, насколько мы уверены в его направлении.

11.4. НАБЛЮДЕНИЕ ВМЕСТО ПРОГНОЗИРОВАНИЯ И ИСПОЛЬЗОВАНИЕ ЭМПИРИЧЕСКИХ ТЕХНИК

По аналогии с предыдущим разделом если мы попробуем спрогнозировать вектор изменений, то можем не улучшить базу кода, а лишь навредить. Как это характерно для нашей индустрии, следует не ставить код в зависимость от догадок и предположений, а опираться на эмпирические техники. Наша сфера деятельности, программирование, движется в направлении более научного подхода с использованием методов для непрерывного совершенствования через структурированные эксперименты, таких как Тойота Ката, управление на основе фактических данных (Evidence-Based Management) и Popcorn Flow.

Можно легко попасть в ловушку собственного стремления оказаться умным и искушенным программистом. Когда мы замечаем возможность сделать что-то расширяемым или обобщенным, решаем более сложную задачу или осознаем нечто гениальное, то непременно стремимся воспользоваться этим открытием.

Если написание более удачного кода требует ничтожно мало времени, то хочется воспользоваться такой возможностью. Но если нет твердой уверенности, что это обобщение в итоге будет использоваться, то нужно подумать, стоит ли добавлять ненужный код и ненужную сложность.

РЕАЛЬНАЯ ИСТОРИЯ

Однажды я обсуждал с разработчиком создание компьютерной игры в шахматы. Я спросил его, как бы он реализовал фигуры. Будучи хорошо знакомым с объектно-ориентированным программированием, он ответил: «С помощью интерфейса и классов». Подводя его к сути, я спросил: «А не будет ли проще прописать их жестко?» Он сказал: «Конечно, но тогда мне жаль тех, кто должен будет обслуживать такое решение» — и рассмеялся, на что я ответил: «А его вовсе не придется обслуживать, ведь шахматы не менялись уже 500 лет». На его лице возникло изумление.

Эта история показывает, что, даже имея под рукой мощные инструменты, не всегда нужно их использовать. Нам следует наблюдать, как именно код склонен изменяться:

- если он не меняется, ничего не делать;
- если он меняется непредсказуемо, то проводить рефакторинг только во избежание хрупкости;
- в других случаях проводить рефакторинг для приспособления к прошлым изменениям.

11.5. ОБЕСПЕЧЕНИЕ БЕЗОПАСНОСТИ БЕЗ ПОНИМАНИЯ КОДА

Вы наверняка помните, что в главе 3 я выступал за рефакторинг без понимания кода. Как мы недавно разобрали, рефакторинг перемещает поведение между потоком управления, структурами данных и данными. Это верно независимо от внутренней предметной области или структуры, потому что структура находится в коде. Чтобы с ней работать, нам не нужно ее понимать при условии, что мы следуем структуре, которая уже есть в коде, и используем надежные шаблоны рефакторинга, не допуская ошибок.

Последняя часть этого утверждения скрывает в себе подвох, потому что люди склонны допускать ошибки. К счастью, нам не нужно снова изобретать колесо, поскольку уже известен перечень возможных действий, проделав которые мы можем себя защитить. Заметьте, что ни один из шаблонов не является безотказным,

и мы обычно используем всего понемногу. Как и в большинстве случаев реальной жизни, существует точка, в которой нам приходится остановиться в рефакторинге и принять оставшуюся долю риска.

11.5.1. Обеспечение безопасности через тестирование

Наиболее распространенный подход к обеспечению безопасности — это тестирование кода. Я считаю, что это надо делать всегда. Не только для проверки его корректности, но также для того, чтобы поставить себя на место пользователей. Мы создаем программное обеспечение, чтобы сделать чей-то мир лучше. Но разве можно этого добиться, если не знать, как выглядит этот мир? Проблема с надежным тестированием кода в том, что оно быстро становится неуправляемым, чрезвычайно времязатратным и само может содержать ошибки, поскольку и системы тестирования делает и эксплуатирует человек. Как и со многими другими однообразными задачами в разработке ПО, решением здесь будет автоматизация: в частности, выполнение тестов на корректность, также известных как *функциональные тесты*. Риск же состоит в том, что и эти тесты могут не охватить область, где возникает ошибка, либо не протестировать то, что мы от них ожидаем.

11.5.2. Обеспечение безопасности за счет мастерства

Еще один способ снизить вероятность возникновения ошибок — это сосредоточиться на выполняющем рефакторинг человеке. Во-первых, нужно разделить рефакторинг на небольшие шаги — настолько небольшие, чтобы риск провала был незначителен. Когда шаги будут достаточно малы, проблемой станет уже риск пропустить некоторые из них. Этот риск можно снизить наработкой опыта. Повторение за повторением — выполняйте рефакторинг в безопасной среде так часто, чтобы он стал для вас механическим. В этом случае риск будет постепенно сокращаться и определяться лишь человеком, проводящим рефакторинг.

11.5.3. Обеспечение безопасности с помощью инструментов

К слову о механистичности, можно сократить человеческие ошибки, вообще исключив человеческий фактор. Многие современные IDE предлагают встроенную возможность рефакторинга посредством вспомогательных инструментов. Так что вместо выполнения шагов для извлечения метода можно попросить программу-редактор сделать это за нас. Потребуется лишь указать, какой код нужно извлечь. В этом случае риск состоит в том, что инструмент может содержать баг. К счастью, если этот инструмент широко используется, то баги обычно быстро исправляются, а значит, и риск уменьшается.

11.5.4. Обеспечение безопасности через формальную верификацию

Если мы создаем ПО, для которого сбои окажутся чрезвычайно убыточными, например, для самолета или марсохода, то можем пойти на крайность и проверить код на отсутствие ошибок формально. Можно даже использовать систему интерактивного доказательства теорем, чтобы механически проверять корректность всех наших доводов, что сегодня является эталонным решением в плане проверки качества. Поскольку это просто еще один метод с использованием инструмента, то риск остается тем же, что и в предыдущем подходе: в системе может оказаться баг, который совпадет с ошибкой в наших доводах.

11.5.5. Обеспечение безопасности через толерантность к ошибкам

И последнее. Мы можем создать код так, что даже в случае возникновения ошибки он будет самокорректироваться. Один из примеров — это переключение функционала: как говорилось в предыдущей главе, мы можем добавить автоматический откат при сбое. Таким образом, даже если будет допущена ошибка во время рефакторинга и код даст сбой, то система переключения функционала автоматически вернется к старому коду.

Этот метод может не сработать, если системе не удастся отличить корректный ответ от ошибки. Примером может быть возвращение функцией `-1` вместо выбрасывания исключения при сбое. Система может ожидать целое число, и `-1` вполне отвечает ее ожиданиям.

11.6. ОПРЕДЕЛЕНИЕ НЕЭКСПЛУАТИРУЕМЫХ СТРУКТУР

Структура присутствует во всех наших действиях. Она определяется предметной областью, тем, как мы друг с другом взаимодействуем, и нашей моделью мышления (предубеждениями и предпочтениями). Большая часть этой структуры просачивается в базы кода. Как я уже говорил, мы можем обработать эту структуру посредством рефакторинга, чтобы сделать наш код стабильным даже при высокой скорости изменения.

Эксплуатация структуры, которая является случайной или краткосрочной, зачастую ведет к снижению скорости. Всегда нужно учитывать, насколько прочна ее основа и окажется ли эта структура устойчивой. Как правило, внутренняя область оказывается старше программного обеспечения, в связи с чем является более зрелой и менее подверженной радикальным изменениям. Следовательно, основанную на такой области структуру зачастую можно использовать безопасно.

Наши процессы и, к сожалению, наши команды имеют намного более короткий жизненный цикл, чем ПО. Они менее стабильны, поэтому если мы решим встроить их в систему, то, скорее всего, будем вынуждены заново разбирать весь код процессов, только чтобы встроить новый, продолжая так до бесконечности.

Прежде чем мы сможем понять, стоит ли задействовать структуру, ее нужно найти. Так что далее мы познакомимся с наиболее распространенными местами для поиска подходящих структур в коде и узнаем, как ими можно воспользоваться.

11.6.1. Эксплуатация пустого пространства с помощью извлечения и инкапсуляции

Разработчики зачастую используют для обозначения структуры кода пустые строки, потому что нам свойственна умственная группировка инструкций, полей и пр. Когда нам нужно реализовать нечто сложное, мы разделяем это на мелкие части. Между этими частями мы размещаем пустые строки и иногда комментарий, который впоследствии служит в качестве первого наброска для имени группы.

Как уже описывалось в главе 1, если имеются группы инструкций, разделенные пустым пространством, то необходимо подумать об использовании «Извлечения метода». Конечно, когда разработчики пишут новый код, они должны сами извлекать методы. Но это требует усилий, и если таковой навык не наработан практикой, то многие склонны пропускать этот элемент рефакторинга. Добавление пустой строки не представляет особых затрат и рисков, в связи с чем к нему прибегают почти все. В результате это помогает понять ход мысли автора кода во время решения данной задачи. При этом очень кстати, что теперь у вас уже есть опыт извлечения методов и вы способны легко укрепить такую структуру. В приведенном ниже примере функция извлекает минимальное значение массива из каждого его элемента. В этом коде два раздела, отделенные пустой строкой.

Листинг 11.13. До

```
function subMin(arr: number[]) {
  let min = Number.POSITIVE_INFINITY;
  for (let x = 0; x < arr.length; x++)
  {
    min = Math.min(min, arr[x]);
  }

  for (let x = 0; x < arr.length; x++)
  {
    arr[x] -= min;
  }
}
```

Листинг 11.14. После

```
function subMin(arr: number[]) {
  let min = findMin(arr);
  subtractFromEach(min, arr);
}
function findMin(arr: number[]) {
  let min = Number.POSITIVE_INFINITY;
  for (let x = 0; x < arr.length; x++) {
    min = Math.min(min, arr[x]);
  }
  return min;
}
function subtractFromEach(min: number, |
```

Извлеченные
методы

```

arr: number[]))
{
  for (let x = 0; x < arr.length; x++) {
    arr[x] -= min;
  }
}

```

Второе распространенное место для поиска неэксплуатируемых пробелов — это области, где они применяются для группировки полей. В этом случае пустое пространство предполагает, что элементы данных являются тесно связанными (то есть изменяются вместе). Данную структуру мы эксплуатировали через шаблон рефакторинга «Инкапсуляция данных» (6.2.3). В следующем примере у нас есть класс `Particle` с полями для `x`, `y` и `color`. Наличие пробела указывает на то, что `x` и `y` тесно связаны, чем мы можем воспользоваться.

Листинг 11.15. До

```

class Particle {
  private x: number;
  private y: number;

  private color: number;
  // ...
}

```

Листинг 11.16. После

```

class Vector2D {
  private x: number;
  private y: number;
  // ...
}
class Particle {
  private position: Vector2D;
  private color: number;
  // ...
}

```

11.6.2. Эксплуатация дублирования с помощью объединения

Мы очень много говорили о дублировании кода. Оно наблюдается в инструкциях, методах, классах и прочих компонентах, потому что, как и пустые строки, не требует особых усилий и не несет особых рисков. По аналогии с теми же пустыми строками известно, как поступать с каждым видом дублирования. Будем следовать структуре, лежащей в основе части I книги: инструкции в методы, а методы в классы.

Повторяющиеся инструкции у нас могут либо находиться рядом друг с другом, либо быть разбросаны по нескольким методам в разных классах. В обоих случаях мы начинаем с использования фундаментального шаблона рефакторинга «Извлечение метода». В примере ниже приведены две программы форматирования. Их общие потоки выполнения различаются, поэтому мы решаем разобраться с инструкцией `result +=`, которая есть в обоих. Для начала мы ее извлекаем.

Листинг 11.17. До

```
class XMLFormatter {
    format(vals: string[]) {
        let result = "";
        for (let i = 0;
            i < vals.length;
            i++) {
            result +=
`<Value>${vals[i]}</Value>`;
        }
        return result;
    }
}
class JSONFormatter {
    format(vals: string[]) {
        let result = "";
        for (let i = 0;
            i < vals.length;
            i++) {
            if (i > 0) result += ",";
            result += `{ value:
                "${vals[i]}" }`;
        }
        return result;
    }
}
```

Листинг 11.18. После

```
class XMLFormatter {
    format(vals: string[]) {
        let result = "";
        for (let i = 0; i < vals.length; i++) {
            result += this.formatSingle(vals[i]);
        }
        return result;
    }
    formatSingle(val: string) {
        return `<Value>${val}</Value>`;
    }
}
class JSONFormatter {
    format(vals: string[]) {
        let result = "";
        for (let i = 0; i < vals.length; i++) {
            if (i > 0) result += ",";
            result += this.formatSingle(vals[i]);
        }
        return result;
    }
    formatSingle(val: string) {
        return `{ value: "${val}" }`;
    }
}
```

Извлеченный
метод

Если извлеченные методы разбросаны по разным классам, то на этот раз можно центрировать их, используя «Инкапсуляцию данных» для методов.

Листинг 11.19. До

```
class XMLFormatter {
    formatSingle(val: string) {
        return `<Value>${val}</Value>`;
    }
    // ...
}
class JSONFormatter {
    formatSingle(val: string) {
        return `{ value: "${val}" }`;
    }
    // ...
}
```

Листинг 11.20. После

```
class XMLFormatter {
    formatSingle(val: string) {
        return new XMLFormatSingle()
            .format(val);
    }
    // ...
}
class JSONFormatter {
    formatSingle(val: string) {
        return new JSONFormatSingle()
            .format(val);
    }
    // ...
}
class XMLFormatSingle {
    format(val: string) {
        return `<Value>${val}</Value>`;
    }
}
class JSONFormatSingle {
    format(val: string) {
        return `{ value: "${val}" }`;
    }
}
```

Инкапсулированный
метод

Если методы идентичны, то эти классы также идентичны, значит, можно просто удалить все, кроме одного. Если эти инкапсулирующие классы просто похожи, то при обнаружении повторяющихся классов можно использовать «Объединение схожих классов» (5.1.1).

Листинг 11.21. До

```
class XMLFormatSingle {
    format(val: string) {
        return `<Value>${val}</Value>`;
    }
}
class JSONFormatSingle {
    format(val: string) {
        return `{ value: "${val}" }`;
    }
}
```

Листинг 11.22. После

```
class XMLFormatter {
    formatSingle(val: string) {
        return new FormatSingle("<Value>",
            "</Value>")
            .format(val);
    }
    // ...
}
class JSONFormatter {
    formatSingle(val: string) {
        return new FormatSingle("{ value: '"',"' }"),
            .format(val);
    }
    // ...
}
class FormatSingle {
    constructor(
        private before: string,
        private after: string) { }
    format(val: string) {
        return `${before}${val}${after}`;
    }
}
```



Если инструкции похожи только в потоке выполнения, но не в операторах, можно сделать их идентичными с помощью «Введения паттерна “Стратегия”». Именно в этом заключается сила данного шаблона рефакторинга: он может раскрывать структуру даже там, где она скрыта.

Листинг 11.23. До

```
class XMLFormatter {
    format(vals: string[]) {
        let result = "";
        for (let i = 0;
            i < vals.length;
            i++) {
            result +=
                new
                FormatSingle("<Value>", "</Value>")
                    .format(vals[i]);
        }
        return result;
    }
}
class JSONFormatter {
    format(vals: string[]) {
```

Листинг 11.24. После

```
class XMLFormatter {
    format(vals: string[]) {
        return new Formatter(
            new
            FormatSingle("<Value>", "</Value>"),
            new None()).format(vals);
    }
}
class JSONFormatter {
    format(vals: string[]) {
        return new Formatter(
            new FormatSingle("{ value: '"',"' }"),
            new Comma()).format(vals);
    }
}
class Formatter {
```



```

let result = "";
for (let i = 0;
    i < vals.length;
    i++) {
    if (i > 0) result += ",";
    result +=
        new FormatSingle
            ("{ value: '",' }")
            .format(vals[i]);
}
return result;
}
}

```

```

constructor(
    private single: FormatSingle,
    private sep: Separator) { }
format(vals: string[]) {
    let result = "";
    for (let i = 0; i < vals.length; i++) {
        result = this.sep.put(i, result);
        result += this.single.format(vals[i]);
    }
    return result;
}
}
interface Separator {
    put(i: number, str: string): string;
}
class Comma implements Separator {
    put(i: number, result: string) {
        if (i > 0) result += ",";
        return result;
    }
}
class None implements Separator {
    put(i: number, result: string) {
        return result;
    }
}
}

```

Паттерн
«Стратегия»

К этому моменту две исходные программы форматирования различаются только константными значениями, значит, можно легко их объединить.

11.6.3. Эксплуатация общих аффиксов с помощью инкапсуляции

Еще один способ, которым мы обнаруживаем структуру в данных, методах и классах, настолько очевиден и надежен, что для него есть правило: «Всегда избегать общих аффиксов» (6.2.1). Аналогично пустой строке с комментарием здесь видно разделение по группам и предполагаемое имя. Этот подход также не требует особых усилий и не представляет больших рисков. И опять же известно, как укрепить структуру, потому что независимо от того, реализовано ли группирование через пробелы, повторение или именование, решение остается прежним: инкапсулировать данные.

До этого момента упоминалось только, как применять это правило к полям и методам. Но его также можно использовать и для группировки классов с похожими именами. Я не стремился акцентироваться на этом, потому что механизм для каждого языка будет свой. В Java можно инкапсулировать

классы внутри других классов или пакетов. В Си у нас есть пространства имен, а в TypeScript пространства имен или модули. Я рекомендую вам самим поэкспериментировать и определить, какой механизм лучше всего подойдет для вашей команды.

В примере ниже имеется несколько протоколов для кодирования/декодирования данных, которые были, вероятно, получены вследствие внедрения паттерна «Стратегия». Содержимое этих протоколов значения не имеет.

Листинг 11.25. До

```
interface Protocol { ... }
class StringProtocol implements Protocol { ... }
class JSONProtocol implements Protocol { ... }
class ProtobufProtocol implements Protocol { ... }
/// ...
let p = new StringProtocol();
/// ...
```

У всех классов есть общий суффикс `Protocol`, который нарушает правило «Всегда избегать общих аффиксов». В данном случае мы не можем удалить `Protocol` напрямую, потому что `String` будет конфликтовать со встроенным классом. Однако этого не произойдет, если сначала мы инкапсулируем эти три класса и интерфейс в пространство имен.

Листинг 11.26. После

```
namespace protocol {
  export interface Protocol { ... }
  export class String implements Protocol { ... }
  export class JSON implements Protocol { ... }
  export class Protobuf implements Protocol { ... }
}
/// ...
let p = new protocol.String();
/// ...
```

В TYPESCRIPT...

В TypeScript для управления доступом на разных уровнях используются разные ключевые слова. Внутри классов поля и методы являются публичными по умолчанию, и можно использовать `private` для ограничения их доступа. Все, что находится вне класса, является закрытым по умолчанию, значит, можно использовать `export`, чтобы расширить доступ для этих компонентов (функций, классов и т. д.).

11.6.4. Эксплуатация типа среды выполнения с помощью динамической диспетчеризации

Ранее я лишь вскользь упоминал заключительный тип структуры, на котором хочу остановиться сейчас и который является распространенным признаком неэксплуатируемой структуры. Речь идет об инспектировании типов среды выполнения с помощью `typeof`, `instanceof`, отражения или приведения типов.

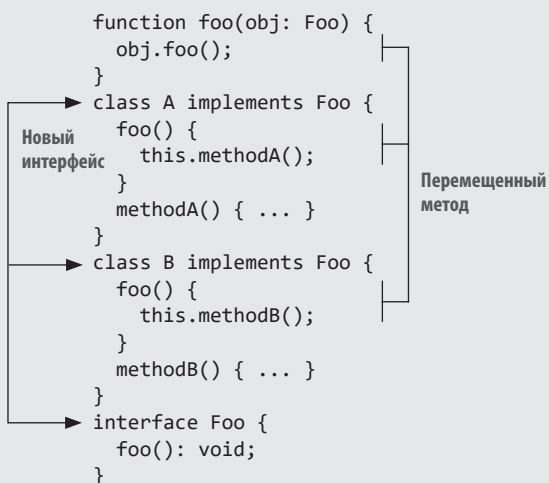
Объектно-ориентированное программирование задумывалось без каких-либо средств инспектирования типа среды выполнения, потому что в нем имеется более мощный механизм: динамическая диспетчеризация через интерфейсы. Используя интерфейсы, можно помещать в переменную разные типы классов. Затем при вызове для этой переменной метода активировать его в соответствующем классе. Это также позволяет избежать использования инспектирования типов среды выполнения и является частным случаем правила «Никогда не использовать `if` с `else`» (4.1.1).

А теперь представим, что у нас есть переменная, которая может содержать что-то с типом `A` или `B`, и мы инспектируем этот тип напрямую, чтобы определить, в каком именно кейсе находимся. Если мы контролируем `A` и `B`, то решение будет простым: создать новый интерфейс, изменить эту переменную на его тип и в обоих классах реализовать этот интерфейс. Теперь можно использовать «Перемещение кода в классы» (4.1.5) — и, как это уже неоднократно случалось, `if` исчезнет.

Листинг 11.27. До

```
function foo(obj: any) {
  if (obj instanceof A) {
    obj.methodA();
  } else if (obj instanceof B) {
    obj.methodB();
  }
}
class A {
  methodA() { ... }
}
class B {
  methodB() { ... }
}
```

Листинг 11.28. После



```
function foo(obj: Foo) {
  obj.foo();
}
class A implements Foo {
  foo() {
    this.methodA();
  }
  methodA() { ... }
}
class B implements Foo {
  foo() {
    this.methodB();
  }
  methodB() { ... }
}
interface Foo {
  foo(): void;
}
```

Новый интерфейс

Перемещенный метод

Если же контроля над источником `A` и `B` у нас нет, то нужно вынести инспектирование типа на границу кода, чтобы обеспечить чистоту ядра базы кода. Та же рекомендация приводится в правиле «Никогда не использовать `if` с `else`».

РЕЗЮМЕ

- Код отражает поведение людей, участвующих в его разработке, а также процессы и внутреннюю предметную область.
- Поведение, встроенное в поток управления, упрощает внесение крупных изменений.
- Поведение, встроенное в структуру данных, предлагает такие преимущества, как безопасность типов, локализация, быстрое действие, упрощает внесение небольших изменений.
- Поведение, встроенное в данные, может быть использовано в качестве последнего средства и должно быть ограничено, поскольку его сложно поддерживать с соблюдением безопасности из-за отсутствия поддержки со стороны компилятора.
- С помощью рефакторинга мы либо обрабатываем дублирование внутри одного из перечисленных подходов, либо переносим структуры из одного подхода в другой.
- Используйте код для раскрытия структуры, чтобы сделать ее податливой с помощью рефакторинга, тем самым совершенствуя структурирование.
- Координируйте усилия по рефакторингу с помощью эмпирических техник и старайтесь не опираться в нем на изменчивую основу.
- Ищите неиспользуемые структуры, которые обычно появляются в результате избегания рисков. Их признаками, как правило, являются пустые пространства, дублирование кода, общие аффиксы или инспектирование типа среды выполнения.

12

Избегайте оптимизаций и обобщенности

В этой главе

- ✓ Минимизация обобщенности для уменьшения зацепления.
- ✓ Представление оптимизации через инварианты.
- ✓ Проработка хрупкости, вызываемой оптимизациями.

Оптимизация производительности и обобщенность — это две своеобразные игры, очень занимательные, по мнению программистов, которые обычно больше вредят, чем помогают. Под *оптимизацией* в данной главе мы будем подразумевать *оптимизацию производительности*, то есть увеличение пропускной способности кода или сокращение времени его выполнения. Под *обобщенностью* будем понимать исполнение кодом более обширной функциональности, что обычно достигается посредством более обобщенных параметров. Чтобы проиллюстрировать обобщенность и ее возможный вред, рассмотрим следующий пример.

Представьте, что у вас попросили нож. Если человек находится в ситуации выживания в природных условиях, то ему подойдет швейцарский армейский ножик. Если же этот инструмент просит повар, то куда более уместен будет разделочный нож. То есть универсальный обобщенный нож никому не будет нужен. В данном примере, как и в коде, вмещающая обобщенность структура может доставить больше хлопот, чем сама обобщенность — пользы, потому что контекстом в таких случаях выступает все.

Текущую главу мы начнем с изучения вредных последствий использования этих практик. Потом перейдем к более глубокому рассмотрению обобщенности и оптимизации и разберем, когда они уместны, а когда — нет.

В разделе 12.2, который посвящен обобщенности, после обсуждения причин ее использования будет рассказано, как избегать ненужной обобщенности. Она может проникать в код, когда в программное обеспечение добавляются возможности, о которых заказчики не просили. Кроме того, она может стать следствием объединения старого кода с новым до достижения новой стадии готовности. От обоих этих видов обобщенности сложно избавиться, поэтому мы будем говорить о том, как не допустить их появления в принципе. Поскольку обобщенность просачивается даже в самые продуманные базы кода, то завершу я данный раздел объяснением того, как находить и нейтрализовать ее ненужные проявления.

В разделе 12.3, посвященном оптимизации, поступим так же, как с обобщенностью: сначала разберемся, когда стоит ее избегать, а когда — нет. После этого рассмотрим подготовительные шаги, предшествующие реализации любой оптимизации:

- первым делом убедиться, что код хорошо отрефакторен;
- затем проверить, чтобы потоки были распланированы грамотно, без лишних затрат, и отыскать узкое место в системе;
- найдя это узкое место, с помощью профилирования определить подходящие для оптимизации методы;
- далее оценить наиболее безопасный из этих методов, а именно: выбор подходящих структур данных и алгоритмов или кэширование.

В завершение же разберемся, почему важно изолировать любую необходимую калибровку производительности.

12.1. СТРЕМЛЕНИЕ К ПРОСТОТЕ

В основе этой главы, а на деле и всей книги лежит идея, что всегда необходимо стремиться к простоте. Этот принцип настолько важен, что был включен в число идеалов сферы разработки ПО в бизнес-сказке Джина Кима *The Unicorn Project* (IT Revolution Press, 2019 год). Простота важна по той причине, что люди имеют ограниченную когнитивную емкость. Невозможно удерживать в голове одновременно очень много информации. При работе с кодом область мышления человека быстро заполняются две вещи: связанные компоненты, поскольку нужно их удерживать во внимании одновременно, и инварианты, которые необходимо отслеживать, чтобы понять их функциональность. Эти источники проблем зачастую связывают с двумя указанными выше известными упражнениями по программированию. Когда мы делаем нечто более общим, то увеличиваем

число его возможных применений, в результате чего к нему может быть при-
вязано больше компонентов. При работе с обобщенным кодом разработчику
необходимо учитывать как можно больше возможных способов, которыми он
может быть вызван.

В главе 4 мы непосредственно столкнулись с проблемой обобщенности. Глядя
на приведенную ниже функцию, невозможно определить, вызывается ли она со
всеми возможными значениями для `Tile` или только с частью из них. Без по-
нимания этого упростить данную функцию никак не получится.

Листинг 12.1. Функция, обобщенная без необходимости

```
function remove(tile: Tile) {
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length; x++) {
      if (map[y][x] === tile) {
        map[y][x] = new Air();
      }
    }
  }
}
```

Вторым источником неприятностей выступает оптимизация, которая опирается
на использование инвариантов. В результате при работе с оптимизированным
кодом мы вынуждены всегда удерживать эти детали в уме. Это занятная игра
и здоровое упражнение — искать инварианты, когда мы работаем с алгоритмами
или структурами данных. Приведу пример: легко заметить, что инвариантом
двоичного поиска является упорядоченность структуры данных, но еще легче
упустить инвариант, что можно эффективно обращаться к элементам не по
порядку.

В главе 7 приведен пример того, как оптимизации вносят инварианты. Там была
кратко затронута реализация счетного механизма. Этот механизм выполняет
подсчет каждого элемента. Чтобы равномерно выбирать случайный элемент
из этой структуры данных, мы генерировали случайное число, которое меньше
общего количества элементов.

Вычисление общего количества элементов делается просто, но необходимость
его постоянного повторения оказывается слишком затратной. Эту затрату
можно оптимизировать, введя поле `total` для отслеживания общего количества
элементов. Вместе с этим полем появляется инвариант, что оно всегда должно
обновляться при добавлении или удалении элементов. В противном случае есть
риск нарушить метод `randomElement`. С другой стороны, в неоптимизированной
версии добавление нового метода не может нарушить существующие.

Наше стремление к упрощению не означает, что код никогда нельзя оптимизиро-
вать и обобщать. Опытные математики утверждают: «Иногда для доказательства
теоремы нам *необходима* более общая лемма». Но при этом всегда должны быть

12.2.1. Создание минимальной функциональности

Упомянутый трехэтапный метод, состоящий из дублирования, преобразования и объединения, гарантирует минимальную обобщенность только в случае минимальной функциональности. Если мы добавляем больше возможностей, чем требуется, или делаем их излишне универсальными, то никакой метод нас уже не спасет. Единственный способ противодействовать этому — придерживаться принципа минималистичной разработки.

Максимизируйте объем несделанной работы.

Кент Бек

Рекомендация «создавать по минимуму» не является новой. Ее уже проговаривали тысячи раз тысячами разных способов. Мне больше всего нравится версия, высказанная Кентом Бекем. Это, вероятно, самая сложная для соблюдения рекомендация данной главы, но, так как она очень важна, ее стоит повторить.

Чтобы создавать функционал по минимуму, для начала нужно понять контекст — область поведения, которую необходимо реализовать. Когда мы что-то понимаем не до конца, наш мозг стремится охватить все возможные случаи. В таких ситуациях мы склонны думать, что, предоставляя нашим пользователям функцию, решающую больше задач, мы делаем им подарок.

Как продемонстрировал пример со швейцарским ножиком, проектирование кода, вмещающего в себя обобщенность, может создать больше хлопот, чем эта обобщенность принесет пользы. Еще одна причина создавать исключительно только то, что заказано, состоит в том, что по мере развития ПО требования склонны изменяться. Поэтому все бонусы от усилия, затраченного на реализацию и поддержание ненужной универсальности, легко сводятся на нет. Следовательно, нам нужно решать лишь ту задачу, которая перед нами фактически стоит, но никак не воображаемую общую.

РЕАЛЬНАЯ ИСТОРИЯ

Недавно я работал над системой для вычисления и отслеживания рейтингов игроков в настольный теннис. Закончив с начальным дизайном и функциональностью, я понял, что могу использовать эти данные для генерации набора команд, которые с наибольшей вероятностью сыграют самые зрелищные матчи. Будучи уверенным, что этот функционал будет востребован пользователями постоянно, я его реализовал. Но, как можно было ожидать, у них уже были методы для определения потенциально интересных встреч, так что потребности в новой реализации не было — ее использовали всего несколько раз, и то из чистого любопытства.

12.2.2. Объединение компонентов с похожим уровнем стабильности

В описанной выше ситуации я мог исправить большую часть ошибок с помощью своей излюбленной практики удаления кода. Но для добавления дополнительной функциональности мне пришлось обобщить часть вспомогательных функций и кода бэкенда. От такой обобщенности гораздо сложнее избавиться, но, поскольку она повышает когнитивную нагрузку, мне пришлось заняться этим.

Чтобы избежать подобных проблем, следует быть очень внимательными при объединении компонентов. Как правило, лучше не спешить объединять что-то новое с чем-то старым. Вместо этого стоит подождать, пока эти компоненты достигнут схожего уровня стабильности. Для этого им не обязательно находиться в употреблении одинаковое количество времени. Чаще всего второй экземпляр чего-либо стабилизируется намного быстрее, а третий и подавно.

12.2.3. Устранение ненужной обобщенности

В качестве последнего средства защиты от ненужной обобщенности рекомендуется регулярно проводить мониторинг на предмет ее присутствия и при обнаружении сразу удалять. В предыдущих главах уже описаны два шаблона рефакторинга, нацеленных конкретно на удаление ненужной обобщенности: «Специализация метода» и «Пробное удаление с последующей компиляцией». Из ознакомительных примеров понятно, что эти шаблоны необходимо применять после обширного рефакторинга. Так мы поступали в учебных целях. Но на деле «Пробное удаление с последующей компиляцией» вряд ли позволит обнаружить всю обобщенность, от которой можно избавиться.

Более эффективным способом ее поиска будет мониторинг передаваемых в функции аргументов среды выполнения. Добавить код для логирования этих параметров несложно при условии, что наши объекты являются достаточно сериализуемыми. После этого можно будет проинспектировать последние N вызовов каждого метода и посмотреть, не вызывается ли какой-либо параметр постоянно с одним значением, в случае чего можно будет использовать для этого параметра «Специализацию метода». Даже если он вызывается с несколькими разными значениями, может быть смысл в создании специализированной копии этой функции для каждого из них.

12.3. КОГДА И КАК ОПТИМИЗИРОВАТЬ

Еще одним типичным источником повышенной когнитивной нагрузки является оптимизация. Как и в случае с обобщенностью, прежде чем делать что-либо, нужно объяснить необходимость этого. В отличие от обобщенности здесь у нас нет простого процесса, который бы автоматически определял необходимость

в оптимизации. Тем не менее в нашем распоряжении есть другой инструмент: для обоснования оптимизации я всегда рекомендую настраивать автоматические тесты производительности и прибегать к ней, только когда они проваливаются. Вот список наиболее распространенных видов подобных тестов.

- «Этот метод должен завершаться в течение 14 мс». Данный вид называется *бенчмарком*. Его нередко используют во встраиваемых решениях или системах реального времени, где требуется предоставить ответ в рамках заданного дедлайна или интервала. Несмотря на простоту написания, подобные тесты тесно привязаны к среде выполнения. Если у нас есть сборщик мусора или антивирусный сканер, то они могут повлиять на абсолютную производительность и мы получим ложные отрицательные результаты. Следовательно, надежно выполнять бенчмарки можно только в продакшен-средах.
- «Этот сервис должен успевать обрабатывать 1000 запросов в секунду». В *нагрузочных тестах* мы оцениваем пропускную способность. Они обычно используются в веб- или облачных системах. В сравнении с бенчмарками нагрузочные тесты намного устойчивее к внешним факторам, но нам все равно может потребоваться продакшен-среда.
- «Текущее выполнение теста не должно оказаться более чем на 10 % медленнее предыдущего выполнения». Наконец, *тест подтверждения быстрого действия* гарантирует, что достигнутая производительность внезапно не обрушится. Подобные тесты полностью отключены от внешних факторов до тех пор, пока они дают согласованные результаты между выполнениями. И все же они способны обнаруживать, когда кто-то добавляет излишне медлительный компонент в основной цикл или случайно переключает одну структуру данных на другую, вызывая увеличение пропусков кэша.

Перефразируя фразу из мира юриспруденции, «код эффективен, пока не доказано обратное». Когда наши тесты подтвердили необходимость оптимизации, мы должны понимать, как сократить будущую когнитивную нагрузку до минимума.

12.3.1. Рефакторинг перед оптимизацией

Первым делом нужно убедиться, что код адекватно отрефакторен. Одна из целей рефакторинга — локализовать инварианты, сделав их более наглядными. Поскольку оптимизация опирается на инварианты, это означает, что будет проще оптимизировать хорошо отрефакторенный код.

Когда в главе 3 мы знакомились с правилом «Вызов или передача» (3.3.1), то делали предварительный рефакторинг, извлекая `length` в отдельную функцию, чтобы избежать нарушения данного правила.

Листинг 12.4. До

```
function average(arr: number[]) {
  return sum(arr) / arr.length;
}
```

Листинг 12.5. После

```
function average(arr: number[]) {
  return sum(arr) / size(arr);
}
```

Тогда этот рефакторинг мог показаться чрезмерным или надуманным. Однако при нынешнем понимании, что следующим шагом будет инкапсулирование методов в класс, становится понятно, что они определяют для нашей новой структуры данных очень аккуратный и минималистичный публичный интерфейс. Этот интерфейс упрощает реализацию описанных ранее оптимизаций. Добавить внутреннее кэширование становится так же просто, как добавить новое поле в новый класс. В качестве альтернативы, если есть желание изменить структуру данных, можно извлечь интерфейс из реализации (5.4.4), а затем создать новый класс, реализующий этот интерфейс, который будет использовать нужную структуру данных.

Листинг 12.6. Инкапсулированный

```
class NumberSequence {
  constructor(private arr: number[]) { }
  sum() {
    let result = 0;
    for(let i = 0; i < this.arr.length; i++)
      result += this.arr[i];
    return result;
  }
  size() { return this.arr.length; }
  average() {
    return this.sum() / this.size();
  }
}
```

Листинг 12.7. Всего кэша

```
class NumberSequence {
  private total = 0;
  constructor(private arr: number[]) {
    for(let i = 0; i < this.arr.length; i++)
      this.total += this.arr[i];
  }
  sum() { return this.total; }
  size() { return this.arr.length; }
  average() {
    return this.sum() / this.size();
  }
}
```

Поручите это компилятору

Еще одна причина делать предварительный рефакторинг заключается в том, что компиляторы постоянно работают над генерацией более оптимального кода. Разработчики этих инструментов обычно решают, что нужно оптимизировать, путем изучения распространенных идиом и примеров использования, сосредотачиваясь в итоге на наиболее актуальных ситуациях и закладывая это в работу инструментов. Следовательно, начиная оригинальничать, мы непреднамеренно замедляем выполнение кода просто потому, что компилятор перестает понимать, что мы стараемся сделать. Здесь уместно вспомнить основную идею главы 7: сотрудничайте с компилятором, а не противьтесь ему.

В примере из главы 1 было показано, что хороший компилятор может автоматически устранить повторяющееся подвыражение `pow(base, exp / 2)`, убедившись в отсутствии побочных эффектов. Таким образом, обе программы должны продемонстрировать одинаковое быстроедействие.

Листинг 12.8. Неоптимизированный

```
return pow(base, exp / 2) * pow(base, exp / 2);
```

Листинг 12.9. Оптимизированный

```
let result = pow(base, exp / 2);
return result * result;
```

Вносимые компилятором улучшения должны означать, что наш код со временем автоматически становится быстрее, если мы пишем его идиоматически.

Это хороший аргумент для откладывания оптимизации в самый долгий ящик. Конечно, против этого работает человеческое желание выглядеть умнее, оно проявляется, когда мы стремимся показать, как ловко разобрались со сложным кодом, или продемонстрировать свою креативность через необычные паттерны и решения. Я и сам это делаю, когда чувствую свою интеллектуальную неуверенность, но только не в общих базах кода! Мой любимый способ выделиться — это заменить две типичные операции необычными, низкоуровневыми аналогами, которые выглядят быстрее.

Листинг 12.10. Идиоматичный

```
function isEven(n: number) {
    return n % 2 === 0;
}
```

Листинг 12.12. Идиоматичный

```
function half(n: number) {
    return n / 2;
}
```

Листинг 12.11. Показуха

```
function isEven(n: number) {
    return (n & 1) === 0;
}
```

Листинг 12.13. Показуха

```
function half(n: number) {
    return n >> 1;
}
```

Код в листингах 12.11 и 12.13 выглядит намного круче, но выражения в листингах 12.10 и 12.12 являются настолько типичными, что все передовые компиляторы смогут автоматически их оптимизировать. А значит, единственным эффектом показухи будет усложненное чтение кода.

12.3.2. Оптимизация согласно теории ограничений

Если после рефакторинга тесты по-прежнему не проходят, нужно начинать оптимизацию. При работе в параллельной системе, будь то с сотрудничающими потоками, процессами или сервисами, мы попадаем под действие *теории ограничений*. Элияху Голдрат в своем гениальном романе *The Goal*¹ иллюстрирует, как стремление к сокращению локальной неэффективности влияет на глобальную эффективность.

Чтобы продемонстрировать теорию ограничений, я обычно использую аналогию из реальной жизни, показанную на рис. 12.1. Система — это дорожное движение, где *задачи* представлены автомобилями, которым нужно проехать слева направо. На своем пути задачи проходят через *рабочие станции*, которые подобны перекресткам со светофорами. Каждый перекресток позволяет автомобилям проезжать с разными скоростями, которые могут изменяться. Между перекрестками есть участок дороги, где автомобили скапливаются в очереди: в теории ограничений этот участок называется *буфером*. Если правый буфер

¹ Голдрат Э. Цель.

перекрестка оказывается практически пуст при почти заполненном левом, такой перекресток становится *узким местом*.

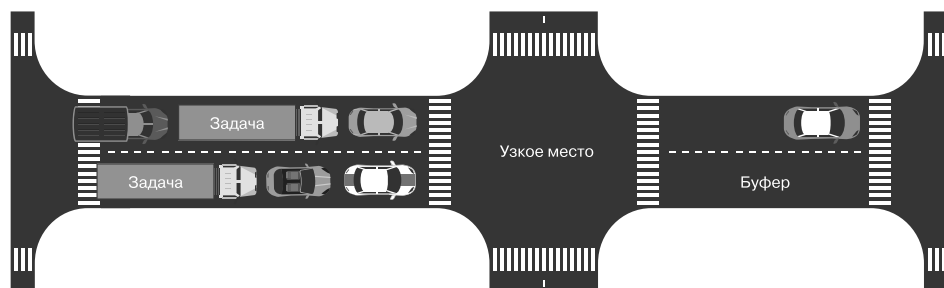


Рис. 12.1. Иллюстрация системы

Независимо от того, рассматриваем мы автомобили, обрабатываемый кусок металла или фрагмент данных, теория ограничений работает для любой системы, состоящей из последовательно связанных рабочих станций. Для разработчиков система — это приложение, а рабочие станции — это последовательно связанные между собой работники. Каждый работник выполняет определенную работу и передает ее результат другому работнику через буфер.

В потоке от входа до выхода в любой момент времени есть ровно один медленный работник, то есть узкое место. Оптимизация работника, предшествующего узкому месту, приводит лишь к ускорению заполнения буфера перед этим узким местом. Оптимизация работника, следующего после узкого места, не влияет на общую производительность, поскольку этот работник не может получать входные данные достаточно быстро. Выходит, что оказать влияние на производительность всей системы может лишь оптимизация самого узкого места.

Однако, оптимизируя одно узкое место, мы создаем другое. Может оказаться, что следующий за бывшим узким местом работник не будет поспевать за его возросшей пропускной способностью либо предшествующий этому месту работник не сможет достаточно быстро производить собственный выход.

К счастью, в сфере ПО для этой ситуации есть изысканное решение, которое называется *объединением ресурсов*. Объединение ресурсов означает, что все доступные обрабатывающие ресурсы помещаются в общий пул, откуда их может взять для обработки любой, кому они потребуются. Таким образом, узкому месту предоставляется максимально возможная производительность. Этот подход можно реализовать внешне, на уровне сервиса через балансировщики нагрузки, либо внутренне, в нашем приложении через объединение потоков.

Независимо от того, объединяем мы ресурсы внешне или внутренне, эффекты для производительности окажутся одинаковыми, поэтому кратко рассмотрим

лишь пример внутреннего объединения. Напомню, что в TypeScript нет потоков, так что ниже приведен псевдокод, ориентированный на Java. В данном примере у нас есть двухэтапная система, в которой этап В занимает вдвое больше времени, чем этап А. Как нам известно, порядок значения не имеет. Для сообщения между потоками мы используем блокирующие очереди, при этом работники и потоки никогда не завершаются. В данной простейшей реализации на каждом этапе у нас по одному работнику; обратите внимание на два бесконечных цикла. Когда мы вводим объединение ресурсов, то выносим этот бесконечный цикл за границы этапов, тем самым делая их задачами.

Листинг 12.14. Простейшая реализация потоков

```
interface Runnable { run(): void; }
class A implements Runnable {
  // ...
  run() {
    while (true) {
      let result = this.input.dequeue();
      Thread.sleep(1000);
      this.output.enqueue(result);
    }
  }
}
class B implements Runnable {
  // ...
  run() {
    while (true) {
      let result = this.input.dequeue();
      Thread.sleep(2000);
      this.output.enqueue(result);
    }
  }
}
let enter = new Queue();
let between = new Queue();
let exit = new Queue();
let a = new A(enter, between);
let b = new B(between, exit);
let aThread = new Thread(a);
let bThread = new Thread(b);
aThread.start();
bThread.start();
```

Листинг 12.15. Объединение ресурсов

```
interface Runnable { run(): void; }
interface Task { execute(): void; }
class A implements Task {
  // ...
  execute() {
    let result = this.input.dequeue();
    Thread.sleep(1000);
    this.output.enqueue(result);
  }
}
class B implements Task {
  // ...
  execute() {
    let result = this.input.dequeue();
    Thread.sleep(2000);
    this.output.enqueue(result);
  }
}
class Worker implements Runnable {
  run() {
    while (true) {
      let task = this.tasks.dequeue();
      task.run();
    }
  }
}
let enter = new Queue();
let between = new Queue();
let exit = new Queue();
let tasks = new Queue();
enter.onEnqueue(element => tasks.enqueue(
  new A(enter, between)));
between.onEnqueue(element =>
  tasks.enqueue(
    new B(between, exit)));
let pool = [
  new Thread(new Worker()),
  new Thread(new Worker());
pool.forEach(t => t.start());
```

Абстрагирование
нового задания

Запускаемый
работник

Планировка
заданий

Объединение
потоков

Как видите, структуры этих фрагментов кода почти идентичны. Настройка несколько усложняется, поскольку нам нужно создавать задание каждый раз, когда образуется работа. Однако решение с объединением ресурсов обеспечивает существенно большую пропускную способность. Обработка 100 запросов программой из листинга 12.14 занимает примерно 201 секунду, в то время как программа из листинга 12.15 может справиться с ними за 150 секунд.

Еще более важно, что даже с тривиальной реализацией объединения ресурсов нам не нужно думать об оркестровке потоков; система следит за этим автоматически. Можно даже впоследствии изменить потоковое поведение, не повлияв на этапы. В данном случае очевидно, что для каждого *A* у нас есть два потока *B*, но на практике у нас есть десятки тысяч небольших этапов и колеблющаяся среда выполнения. Расплачиваемся мы за это необходимостью поддерживать код объединения ресурсов или ПО, что увеличивает когнитивную нагрузку системы. Однако важно, что при этом мы не увеличили когнитивную нагрузку кода предметной области в этапах.

12.3.3. Координирование оптимизации с помощью метрик

Если после оптимизации системы с помощью объединения ресурсов мы по-прежнему не вписываемся в требования по быстродействию, нужно производить оптимизацию внутри узкого места. Здесь ситуация с одним потоком, то есть нам нужно сделать так, чтобы этот поток выполнял задачу быстрее. Однако рассчитывать на полномасштабную оптимизацию всего не стоит. Мало того, что это потребовало бы невероятных усилий, так еще и колоссально усложнило бы работу с базой кода. Вместо этого лучше сосредоточиться на тех частях кода, которые окажут наиболее значительное влияние.

Для этого нужно определить в коде хот-споты. *Хот-споты* — это методы, в которых поток проводит больше всего времени. Образованию хот-спота способствуют два фактора: метод, требующий времени на завершение, и метод, находящийся внутри цикла. Единственным надежным способом обнаружения хот-спотов является профилирование. *Профилирование* подразумевает отслеживание того, сколько всего времени проводится в методе. Для этого существует великое множество инструментов. В качестве альтернативы можно легко добавлять отсчитывающий время код, начиная с верхнего уровня, и затем итеративно углубляться в те 20 % кода, которые занимают 80 % времени.

То, что известное соотношение 80:20 применимо к коду, также подтверждает мое убеждение в том, что оптимизация не должна являться частью ежедневной работы, поскольку она выполняется в ущерб более ценному ресурсу — продуктивности команды. Единственное исключение — это разработчики, чья повседневная

работа связана с хот-спотами, например специалисты по производительности или люди, работающие со встроенными системами либо системами реального времени.

Когда производительность становится ключевым вопросом, для использования профилирования есть еще одна причина. Многие программисты знакомы с базовой алгоритмикой, включая асимптотический анализ (обычно нотация « O большое»). Притом что знание подобных принципов может принести много пользы, здесь важно понимать, что показатель асимптотического роста упрощен. Следовательно, переход на алгоритм или структуру данных с лучшей асимптотикой на деле может привести к снижению производительности ввиду тех же факторов, для абстрагирования которых предназначен анализ, таких, например, как пропуски кэша. Выявить эти эффекты можно лишь с помощью измерений. Подтверждается это тем фактом, что большинство библиотечных функций сортировки используют для небольших данных сортировку вставками со сложностью $O(n^2)$ вместо более асимптотически эффективной быстрой сортировки, выполняющейся за $O(n \lg(n))$.

12.3.4. Выбор удачных алгоритмов и структур данных

Определив хот-спот в узком месте, можно переходить к поиску способов его оптимизации. Наиболее безопасным способом оптимизации будет замена одной структуры данных на другую, имеющую идентичный интерфейс. Такая оптимизация безопасна, поскольку код нашей области не должен меняться для адаптации новой структуры данных. В этом случае вводимый нами инвариант относится к использованию, то есть в случае его нарушения мы рискуем снижением производительности.

Используемые нами тесты производительности сразу же перехватят ее падение, и поменять структуру данных или алгоритм в этот момент будет несложно. Поэтому я обычно не возражаю против введения подобных инвариантов. Я советую разработчикам учитывать поведение при выборе между существующими структурами данных или алгоритмами. Если мы реализуем их сами и не находимся в хот-споте, то все равно должны отдавать предпочтение простоте реализации.

Иногда можно получить пользу от локальной смены структуры данных. Это типичная практика, когда мы используем внутри хот-спота данные, которые доступны вне его. Представьте, что у нас есть определенные данные и мы хотим извлечь их элементы по порядку в хот-споте. Это можно сделать путем повторяющегося извлечения минимального элемента, что окажется операцией в линейном времени $O(n)$. Но если у нас есть данные вне хот-спота, то можно поместить их в структуру данных вроде кучи для минимума и извлекать их оттуда за логарифмическое время $O(\lg(n))$. А еще лучше, если у нас есть возможность

упорядочить эти данные перед их входом в хот-спот. Тогда можно будет извлекать минимальный элемент за постоянное время $O(1)$.

Как я уже говорил, это типичный случай, который предоставляет аргумент в пользу применения структур данных вместо алгоритмов. Однако эту идею можно развить и далее. В разных частях кода данные можно использовать по-разному. Рассмотрим случай, когда наши инварианты поведения не согласуются по всему коду. Тогда можно сменить структуру данных локально, чтобы она подходила под конкретное использование. Эта идея выглядит очевидной, но по своему опыту я знаю, что ей следуют довольно редко.

В качестве примера представим, что реализовали структуру данных связанный список и нам нужно, чтобы в ней был метод `sort`. Сортировку можно реализовать, управляя связанным списком напрямую. Ввиду особенности поведения кэша будет эффективнее преобразовать этот список в массив, затем его упорядочить и преобразовать обратно в связанный список.

Листинг 12.16. Упорядочение связанного списка

```
interface Node<T> { element: T, next: Node<T> }
class LinkedList<T> {
  private root: Node<T> | null;
  // ...
  sort() {
    let arr = this.toArray(); Array.sort(arr);
    let list = new LinkedList<T>(arr);
    this.root = list.root;
  }
}
```

ПРИМЕЧАНИЕ

Напомню, что мы можем обращаться к `list.root` другого объекта, потому что `private` относится к классу, а не к объекту.

Этот метод очень эффективен, и нам потребовалось лишь написать код для преобразования в массив и обратно, а это нам и так пришлось бы делать. Если же вдобавок мы захотим, чтобы наш связанный список был неизменяемым, то можно просто заменить последнюю строку с присваиванием на инструкцию `return`.

12.3.5. Использование кэширования

Кэширование — это еще одна оптимизация, которую зачастую можно использовать безопасно. Его идея проста: вместо многократного выполнения вычислений делать их один раз, сохранять результат и потом его использовать. В главе 5 есть пример кэширующего класса, который может обертыть любую функцию

с целью отделения побочных эффектов от возвращаемого значения. Инвариантом, характерным для всех операций кэширования, является многократный вызов функции с одними и теми же аргументами.

Листинг 12.17. Кэш для отделения побочных эффектов от возвращаемого значения

```
class Cacher<T> {
  private data: T;
  constructor(private mutator: () => T) {
    this.data = this.mutator();
  }
  get() {
    return this.data;
  }
  next() {
    this.data = this.mutator();
  }
}
```

Кэширование максимально безопасно при совмещении с инвариантом идемпотентности. Это значит, что вызов его с одинаковыми аргументами всегда будет давать один и тот же результат. В таких случаях можно выполнять кэширование внешне. Ниже приведен пример. В целях простоты он получает всего один аргумент, но может быть расширен для работы с многоаргументными функциями. Единственное требование — чтобы аргументы содержали метод `hashCode`, который встречается во многих языках.

Листинг 12.18. Кэш для идемпотентных функций

```
interface Cacheable { hashCode(): string; }
class Cacher<G extends Cacheable, T> {
  private data: { [key: string]: T } = { };
  constructor(private func: (arg: G) => T) { }
  call(arg: G) {
    let hashCode = arg.hashCode();
    if (this.data[hashCode] === undefined) {
      this.data[hashCode] = this.func(arg);
    }
    return this.data[hashCode];
  }
}
```

Кэширование становится менее безопасным, когда наша функция идемпотентна лишь временно. Временная идемпотентность типична для изменяемых данных: например, цена продукта вряд ли будет изменяться с каждым вызовом. Этот инвариант менее устойчив. Представьте, что цена изменяется, притом что в кэше остается ее прежнее значение, которое теперь становится неверным. В качестве

стандартной реализации для приведенного выше внешнего кэша можно добавить срок действия. Заметьте, что этот инвариант более хрупок, потому что изменение установленного срока менее вероятно, чем нарушение основного свойства вроде идемпотентности.

Листинг 12.19. Кэш для временно идемпотентных функций

```
interface Cacheable { hashCode(): string; }
class Cacher<G extends Cacheable, T> {
  private data: { [key: string]: { result: T, expiry: number } } = { };
  constructor(private func: (arg: G) => T,
    private duration: number) { }
  call(arg: G) {
    let hashCode = arg.hashCode();
    if (this.data[hashCode] === undefined
      || this.data[hashCode].expiry < Date.now()) {
      this.data[hashCode] = {
        result: this.func(arg),
        expiry: Date.now() + this.duration
      };
    }
    return this.data[hashCode].result;
  }
}
```

Даже без идемпотентности мы все равно можем реализовать кэширование. Однако тогда оно должно быть внутренним. К примерам такого кэша можно отнести поле `total` из листинга 12.7. Как мы уже говорили, это опаснее всего, поскольку нам потребуется обслуживать его во всем классе в течение всего срока существования.

12.3.6. Изоляция оптимизированного кода

Существуют редкие случаи, когда алгоритмов, параллельности и кэширования оказывается недостаточно для выполнения требований тестов производительности. В подобных случаях мы прибегаем к ее *калибровке*, иногда называемой *микрооптимизациями*. В ходе этого процесса мы ищем небольшие инварианты во взаимосвязи между средой выполнения и желаемым поведением.

К примерам калибровки относится использование *паттернов магических битов*. Это магические числа, обычно они записываются в шестнадцатеричном виде, что еще больше усложняет их чтение. Паттерны магических битов зачастую соответствуют тонкому нюансу используемого алгоритма: нам нужно либо понять его ценой высокой когнитивной нагрузки, либо оставить код в покое. Чтобы понять смысл сказанного, рассмотрим функцию на Си, вычисляющую обратный квадратный корень. Эта функция вместе с оригинальными комментариями была

позаимствована из базы кода видеоигры *Quake III Arena*. Чувствовали бы вы себя уверенно, внося изменения в эту функцию?

Листинг 12.20. Функция вычисления обратного квадратного корня с помощью паттерна магических битов

```
float Q_rsqrt( float number )
{
    long i; float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y = number;
    i = * ( long * ) &y; // злостный взлом плавающей точки на уровне битов
    i = 0x5f3759df - ( i >> 1 ); // какого черта? ← Паттерн магических битов
    y = * ( float * ) &i;
    y = y * ( threehalfs - ( x2 * y * y ) ); // Первая итерация
    // y = y * ( threehalfs - ( x2 * y * y ) ); // Вторая итерация, можно удалить
    return y;
}
```

Использование методов и классов для минимизации блокируемой области

Мы не можем внести какие-либо существенные изменения в откалиброванную функцию, не поняв ее, что обычно оказывается непросто (то есть дорого в когнитивном смысле). В итоге такой код оказывается, по сути, заблокирован. Осознавая это, мы должны изолировать откалиброванный код, стараясь свести блокируемую область до минимума с сохранением эффективности калибровки. Когда в калибровке задействованы данные, нужно использовать для их изоляции класс. В противном случае можно извлечь ее код в отдельный метод.

Относительно именования в подобных ситуациях я считаю, что можно доработать его позднее, когда мы лучше разберемся в коде. Вот только в случае с откалиброванным кодом маловероятно, что кто-либо когда-либо разберется в нем лучше, чем мы после его извлечения. Поэтому нам следует хорошенько постараться подобрать для этого метода или класса удачное имя, грамотно его задокументировать и тщательно проверить качество. Если все это будет сделано на совесть, то ни у кого не возникнет желания лезть в его исходник.

Предупреждение будущих разработчиков через пакеты

Будет также нелишним сообщить будущим разработчикам, что этот код откалиброван, в связи с чем углубляться в него не стоит. Поскольку мы только что изолировали его в методы и классы, потребуется очередной уровень абстракции: пакеты или пространства имен. Как я уже говорил, в разных языках для этого

есть разные механизмы, но описываемая в данном разделе идея будет работать для любого из них.

Я рекомендую использовать для откалиброванного кода отдельный пакет. Дело в том, что, когда мы его импортируем и используем, пакет становится невидимым. Это оказывается очевидным при поверхностном осмотре кода, поскольку он находится на первой строке файла и отображается в большинстве механизмов автозавершения кода. Грамотные же предупреждающие знаки проявляются только при необходимости, чтобы не отвлекать программиста в повседневной работе.

Если вам нужен совет по именованию подобного пакета, то я предпочитаю использовать слово `magic`. Мое отношение к калибровке производительности выражается известным изречением «Достаточно продвинутая технология неотличима от магии». Причем оно интересно согласуется с тем фактом, что в самой калибровке многое опирается на магические константы, такие как ранее рассмотренный паттерн магических битов.

Помимо указания на сложность чтения определенного кода, объединение всех откалиброванных фрагментов также говорит о том, что к этой области применяются особые требования качества. Ни при каких обстоятельствах этот пакет не должен становиться мусорной кучей кода, которую никто не сможет понять. Наоборот, это должен быть алтарь для того кода, который исключительно хорошо понимают лишь некоторые разработчики, хотя бы в период его осмысления. Это окажется полезным для пользователей, поскольку они будут знать, что в этой области баги менее вероятны, а также повлияет на автора, которому потребуются соблазны более высокие требования к качеству или нарушить священность алтаря. Вряд ли кому-то захочется пересекать магическую черту. В следующей главе мы разберем этот феномен более подробно.

РЕЗЮМЕ

- Простота подразумевает снижение когнитивной нагрузки, которую для нас создает код.
- Обобщенность повышает риск зацепления.
- Внося обобщенность через объединение и построение функционала по принципу минимализма, мы избегаем появления ее излишков.
- При объединении только такого кода, который имеет схожую стабильность, риск необходимости удаления обобщенности снижается.
- Для обнаружения ненужной обобщенности или претендентов на оптимизацию используются мониторинг и профилирование.

- Любая оптимизация должна обосновываться спецификацией, которая на практике обычно принимает форму тестов производительности. В повседневной работе оптимизации лучше избегать.
- Рефакторинг локализует инварианты, а оптимизация на них опирается. Значит, рефакторинг оказывается оптимизирующим.
- Объединение ресурсов способно производить оптимизацию без увеличения хрупкости кода предметной области.
- Выбор между существующими алгоритмами и структурами данных является достойной внимания оптимизацией.
- Кэширование может оказаться дешевой и безопасной оптимизацией, вносящей мало инвариантов.
- При использовании калибровки производительности ее необходимо изолировать, чтобы другие разработчики не тратили время на попытки ее понять.

13

Пусть плохой код выглядит плохо

В этой главе

- ✓ Причины отделения хорошего кода от плохого.
- ✓ Виды плохого кода.
- ✓ Правила безопасного ухудшения кода.
- ✓ Применение этих правил, чтобы сделать плохой код еще хуже.

В конце последней главы мы разобрали преимущества от наглядного выражения качества кода, которое будет понятно с первого взгляда. В контексте оптимизации мы делали это путем его изоляции в отдельном пространстве имен или пакете. В данной же главе мы изучим, как более наглядно выражать качество кода, делая плохой код очевидно плохим. Называется этот процесс *антирефакторингом*.

Первым делом мы рассмотрим, в чем заключается польза антирефакторинга, сначала с позиции процессов, а затем с позиции обслуживания. После того как разберемся с мотивацией его использования, рассмотрим признаки плохого кода, познакомившись с некоторыми наиболее распространенными метриками качества. Последним подготовительным шагом перед самим антирефакторингом будет определение фундаментальных правил, гарантирующих, что мы

не повредим безвозвратно структуру кода, а лишь изменим ее представление. Изучив правила, завершив главу знакомством с несколькими безопасными практическими методами придания нужному коду заметности. В этом практическом разделе вы также увидите, как использовать пройденные правила для разработки техник, подходящих вашей команде.

13.1. ПРОБЛЕМЫ ПРИВЛЕЧЕНИЯ ВНИМАНИЯ К ПЛОХОМУ КОДУ

Иногда приходится читать или писать код, который очевидно не так хорош, как должен быть. Однако ввиду таких ограничений, как сложность этого кода, задачи, или банально из-за нехватки времени нельзя отрефакторить код до желаемого уровня. В подобных ситуациях мы иногда проводим небольшой рефакторинг «просто, чтобы код не выглядел так ужасно». Делаем мы это, потому что гордость не позволяет нам отправлять низкокачественную работу. Хотя фактически такие действия являются ошибочными. В этой ситуации будет лучше отправить страшную неразбериху, чем запрятать проблемы под пресловутый ковер.

Сохранение плохого кода имеет два преимущества: его будет легко снова отыскать и он будет сигнализировать о том, что требования к ПО не выполняются. Чтобы программист мог решиться отправить плохой код, сигнализируя о проблеме, в команде должен быть установлен высокий уровень психологической безопасности. Необходимо верить, что мы лишь выступаем в роли посланника и нам за это не влетит. А вот отсутствие чувства безопасности наверняка окажется большей проблемой, чем качество нашего кода. В своем проекте *Project Aristotle* компании Google и re:Work показали, что психологическая безопасность является наиболее значительным фактором продуктивности. Как бывший техлид, я всегда следовал убеждению «Быть проинформированным всегда лучше», то есть к посланникам нужно относиться с уважением. Мне просто нужно было знать, что, например, мы движемся в неустойчивом русле и качество продукта снижается. Поскольку я тоже был занят, код среднего качества мог проскакивать мимо меня незамеченным, но вот очевидно плохой код уже бы не проскочил. Рассмотрите приведенные ниже два примера с одинаковой функциональностью. Какой из них больше нуждается в рефакторинге?

Ответ: «Оба». Несмотря на то что методы в листинге 13.1 малы, они плохо извлечены, скрыто повторение `banner.state`. В результате стало сложно увидеть, что этот метод нужно переместить в класс `State`. Оставляю это в качестве упражнения для усердного читателя.

Листинг 13.1. Удовлетворительный

```
function animate() {
  handleChosen();
  handleDisplaying();
  handleCompleted();
  handleMoving();
}
function handleChosen() {
  if (value >= threshold
    && banner.state === "chosen") {
    // ...
  }
}
function handleDisplaying() {
  if (value >= target
    && banner.state === "displaying") {
    // ...
  }
}
function handleCompleted() {
  if (banner.state === "completed") {
    // ...
  }
}
function handleMoving() {
  if (banner.state === "moving"
    && banner.target === banner.current)
  {
    // ...
  }
}
```

Встроенная функция
с добавленными
пустыми строками

Встроенная функция
с добавленными
пустыми строками

Встроенная функция
с добавленными
пустыми строками

Встроенная функция
с добавленными
пустыми строками

Листинг 13.2. Намеренно плохой

```
function animate() {
  // FIXME: All concern banner.state
  if (value >= threshold
    && banner.state === State.Chosen) {
    // ...
  }
  if (value >= target
    && banner.state === State.Displaying)
  {
    // ...
  }
  if (banner.state === State.Completed) {
    // ...
  }
  if (banner.state === State.Moving
    && banner.target === banner.current)
  {
    // ...
  }
}
enum State {
  Chosen, Displaying, Completed, Moving
}
```

Новый комментарий

Новое перечисление

13.2. РАЗДЕЛЕНИЕ НА БЕЗУПРЕЧНЫЙ И LEGACY-КОД

Чем хуже код, тем легче его обнаружить. Простота обнаружения важна, так как разработчики зачастую большую часть внимания уделяют решению задач. Если что-то не оказывается отчетливо выраженным, то они запросто могут это пропустить. Если же код очевидно плох, то не заметить его сложно, и кто-нибудь обязательно займется исправлением ситуации в подходящее время. Я люблю говорить: «Если не можете сделать что-то хорошо, то сделайте это выделяющимся».

Я не имею ввиду, что весь код должен быть совершенен, но если подразделять код в базе на отличный, удовлетворительный или плохой, то я бы предпочел, чтобы в ней был плохой код, а не удовлетворительный. Если у нас не хватает времени

или навыков для его поднятия до «отличного» уровня, то следует сделать его плохим. Такой подход разделит наш код на безупречный и legacy.

Когда можно с ходу определить, является код безупречным или же legacy, становится легко оценить соотношение хорошего и плохого кода в файле. А это очень полезно знать для грамотной организации рефакторинга. В частности, мне нравится начинать с файлов, которые ближе всего к безупречности. Делаю я так по двум причинам. Во-первых, рефакторинг зачастую представляет каскадную деятельность, то есть для того, чтобы сделать что-то качественным, нам нужно сделать таковым и окружающий код. Если же окружающий код уже хорош, то риск погрузиться в кроличью нору рефакторинга снижается. Вторая же причина — это *теория разбитого окна*.

13.2.1. Теория разбитого окна

Согласно теории разбитого окна, если в здании разбить одно окно, то вскоре люди устремятся бить и другие. И хотя эту теорию оспаривают, я все равно нахожу ее ценной по меньшей мере в качестве аналогии. Чисто интуитивно каждый может отыскать в ней смысл: пока я ношу новые ботинки, я стараюсь за ними тщательно следить. Но, как только они пачкаются, моя заботливость уменьшается и ботинки стремительно утрачивают свой прежний лоск. Аналогичный эффект наблюдается и при разработке кода. Когда мы видим плохой код, то с большей легкостью готовы добавить рядом такой же. Если же мы делаем безупречными целые файлы, то они обычно остаются в таком состоянии дольше.

13.3. ПОДХОДЫ К ОПРЕДЕЛЕНИЮ ПЛОХОГО КОДА

Прежде чем разобрать, как можно ухудшить плохой код, сначала освоим несколько методов для его нахождения. Как говорилось еще во вступлении, нельзя определить, насколько код плох, просто на него посмотрев. Причина в том, что читаемость, хоть и является одной из составляющих хорошего кода, при этом оказывается субъективной. Однако есть несколько других методов оценки качества. Ниже мы разберем самые распространенные из них, чтобы выделить наиболее заметные его черты.

13.3.1. Правила в этой книге: простые и конкретные

Представление о плохом коде вырабатывалось в части I книги, для чего были введены выразительные правила. Эти правила созданы так, чтобы привлекать наше внимание, даже когда оно занято чем-то другим или у нас еще мало опыта.

Несмотря на то что они очень эффективны в период, пока мы еще вырабатываем свое шестое чувство, универсальными эти правила не назовешь. Программисты,

которые не читали эту книгу, наверняка не будут рассматривать передачу чего-либо в качестве параметра и вызов метода для того же объекта как нечто привлекающее внимание и даже не сочтут это плохим. Если в вашей команде есть общий набор правил, подобных приведенным в этой книге, то обычно для выделения кода нужно просто делать противоположное им.

Пример ниже нарушает два правила. Можете понять какие?

Листинг 13.3. Нарушение двух правил

```
function minimum(arr: number[][]) {
  let result = 99999;
  for (let x = 0; x < arr.length; x++) {
    for (let y = 0; y < arr[x].length; y++) {
      if (arr[x][y] < result)
        result = arr[x][y];
    }
  }
  return result;
}
```

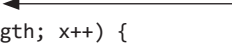
Ответ: правила «Пять строк» (3.1.1) и «if только в начале» (3.5.1).

13.3.2. Запахи кода: полноценные и абстрактные

Мои правила появились не на ровном месте. Я их добыл из запахов кода, собранных по множественным ресурсам, таким как книги «Рефакторинг» Мартина Фаулера и «Чистый код» Роберта Мартина. Так что использование запахов кода — это еще один подход для определения признаков плохого кода. Мои личные наблюдения учат, что большинство запахов начинают привлекать внимание программиста, только когда тот уже имеет за спиной определенный опыт. Хотя некоторые запахи достаточно легко осваиваются даже на вводных курсах и являются заметными для всех, например «Магические константы» и «Повторяющийся код».

Листинг 13.4. Пример запаха кода

```
function minimum(arr: number[][]) {
  let result = 99999;
  for (let x = 0; x < arr.length; x++) {
    for (let y = 0; y < arr[x].length; y++) {
      if (arr[x][y] < result)
        result = arr[x][y];
    }
  }
  return result;
}
```



Магическое число

13.3.3. Цикломатическая сложность: алгоритмическая (объективная)

Притом что два предыдущих метода предназначены для людей, также были попытки научить компьютеры обнаруживать плохой код. Примененные для этого алгоритмы опять же являются лишь аппроксимациями. Однако поскольку они основаны на конкретных расчетах, то представляют ценность и для человека, который может на их основе принять решение о необходимости рефакторинга. Самой известной метрикой оценки качества кода, вероятно, является *цикломатическая сложность*.

Если коротко, то эта метрика подсчитывает количество путей в коде. Это можно подсчитать на уровне инструкций, где `if` имеет два пути: один для истинного результата и один для ложного. То же касается `for` и `while`, поскольку мы в них либо входим, либо пропускаем. Можно также сделать подсчет на уровне выражений, где каждый оператор `||` или `&&` разделяет путь надвое: один пропускает правую сторону, а второй — нет. Интересно то, что эта метрика также указывает минимальное число необходимых тестов, поскольку для каждого пути нужен хотя бы один.

Листинг 13.5. Цикломатическая сложность: 4

```
function minimum(arr: number[][]) {           +1
  let result = 99999;
  for (let x = 0; x < arr.length; x++) {       +1
    for (let y = 0; y < arr[x].length; y++) {   +1
      if (arr[x][y] < result)                  +1
        result = arr[x][y];
    }
  }
  return result;
}
```

=4

Цикломатическая сложность вычисляется для потока управления метода. Для человека она не всегда очевидна, особенно на уровне выражений. Когда он оценивает цикломатическую сложность на глаз, то обычно опирается на отступы, поскольку делает их для каждого `if`, `for` и т. д.

13.3.4. Когнитивная сложность: алгоритмическая (субъективная)

Намного более молодой по времени возникновения метрикой вычисления оценки качества кода является *когнитивная сложность*. Как и следует из ее названия, она оценивает, сколько информации человек должен удерживать в голове, читая данный метод. Эта метрика жестче относится к вложенности,

чем цикломатическая, поскольку разработчику нужно помнить каждое условие, через которое проходит код. Когнитивная сложность близка к оценке того, насколько сложно человеку прочесть что-либо. Однако при поиске заметных глазу признаков это также сводится к подсчету отступов.

Листинг 13.6. Когнитивная сложность: 6

```
function minimum(arr: number[][]) {
  let result = 99999;
  for (let x = 0; x < arr.length; x++) {      +1
    for (let y = 0; y < arr[x].length; y++) { +2
      if (arr[x][y] < result)                  +3
        result = arr[x][y];
    }
  }
  return result;
}
```

=6

13.4. ПРАВИЛА БЕЗОПАСНОГО УХУДШЕНИЯ КОДА

Когда мы ухуждаем код (то есть делаем плохой код выделяющимся), нужно придерживаться трех правил.

1. Никогда не уничтожать верную информацию.
2. Не усложнять будущий рефакторинг.
3. Результат должен привлекать внимание.

Первое, оно же самое важное правило в том, что мы должны сохранить любую корректную информацию. Например, если у метода подходящее имя, но само тело оставляет желать лучшего, не нужно портить имя, чтобы метод выделялся еще больше. Допускается удаление неверной или избыточной информации, к примеру устаревших или тривиальных комментариев.

Второе правило гласит, что своими усилиями мы не должны усложнить работу следующему разработчику, тем более что мы и сами можем оказаться в этой роли. Таким образом, нужно обозначить любую имеющуюся информацию, включая предположения о том, как бы мы отрефакторили этот код. Например, поместить пустые строки там, где нужно извлечь методы. По возможности желательно упростить дальнейший рефакторинг.

Третье правило гласит, что получающийся в итоге код должен привлекать внимание. Таким образом мы гарантируем, что его заметят, и указываем на серьезный недочет, не позволяющий ему считаться безупречным кодом. Следуя этим трем правилам, мы не создадим дополнительных проблем, поскольку все, что делается согласно им, является легко обратимым.

13.5. МЕТОДЫ БЕЗОПАСНОГО УХУДШЕНИЯ КОДА

Познакомившись с правилами игры, пора рассмотреть общие методы, которые я использую для выделения плохого кода. Я советую вам выработать для этого собственные способы, которые будут соответствовать тому, что считает запахами ваша команда. Главное — старайтесь не нарушать перечисленные правила.

Все описанные ниже способы просты и вполне обратимы. Безопасность и обратимость очень важны: эти методы я придумал для ситуаций, когда оказываюсь сильно погружен в другую работу и не всегда могу верно оценить код. Они фокусируются на признаках кода, которые либо заметны большинству людей, либо очень полезны для дальнейшего рефакторинга.

13.5.1. Использование перечислений

Мой любимый метод для выделения требующего рефакторинга кода — это замена кода типа, например, логического, на перечисление. Добавляются перечисления обычно быстро и легко, к тому же, являются очень заметными. В главе 4 было показано, что удаление перечислений хоть и занимает время, но сложностей не вызывает. Еще одно преимущество перечислений в простоте их чтения, так как они именованы.

Учитывая три перечисленных правила, сначала нужно понять, может ли такой подход уничтожить информацию. Если заменить логический тип, то единственная возможная информация окажется в форме именованных констант. В таком случае можно сохранить эти имена в качестве имен значений перечисления. Кроме этого, превращая логический тип в перечисление, приходится добавлять информацию в сигнатуры типов переменных и методов.

Листинг 13.7. До

```
class Package {
    private priority: boolean;
    scheduleDispatch() {
        if (this.priority)
            dispatchImmediately(this);
        else
            queue.push(this);
    }
}
```

Листинг 13.8. После

```
class Package {
    private priority: Importance;
    scheduleDispatch() {
        if (this.priority === Importance.Priority)
            dispatchImmediately(this);
        else
            queue.push(this);
    }
}
enum Importance {
    Priority, Regular
}
```

Передан
в перечисление

Второе правило утверждает, что изменения не должны усложнять дальнейший рефакторинг. Здесь же он даже упрощается, поскольку для устранения перечислений имеется стандартный поток: «Замена кода типа классами» (4.1.3), затем «Перемещение кода в классы» (4.1.5) и в завершение «Пробное удаление с последующей компиляцией» (4.5.1) для избавления от избыточных методов.

Согласно третьему правилу результат должен быть хорошо заметным. Перечисления вполне заметны, хотя не все распознают их как запах кода. Данная трансформация настолько полезна для будущего рефакторинга, что на этот нюанс можно не обращать внимания.

13.5.2. Использование целых чисел и строк в качестве кода типа

Иногда бывает так, что нет возможности добавить перечисление или просто нужно привести что-то в рабочее состояние очень быстро. В таких случаях я часто использую в качестве кода типа целые числа или строки. Если мы берем строки, то получаем преимущество в том, что их текст служит той же цели, что и имя константы. Код строкового типа также очень гибок, поскольку нам не приходится объявлять все значения заранее. Так что в ситуациях ускоренного экспериментирования я использую именно этот подход.

Листинг 13.9. Строки в качестве кода типа

```
function area(width: number, shape: string)
{
    if (shape === "circle")
        return (width/2) * (width/2) * Math.PI;
    else if (shape === "square")
        return width * width;
}
```

Листинг 13.10. Целые числа в качестве кода типа

```
const CIRCLE = 0;
const SQUARE = 1;
function area(width: number, shape: number)
{
    if (shape === CIRCLE)
        return (width/2) * (width/2) * Math.PI;
    else if (shape === SQUARE)
        return width * width;
}
```

При использовании целочисленных именованных констант или строк мы можем включать любую нужную нам информацию, а значит, риск ее утраты отсутствует.

Этот метод подразумевает последующее использование предыдущего. Когда скорость экспериментирования снижается, очередным шагом идет замена строк или целых чисел перечислениями. Поскольку мы встраиваем информацию в имя константы или содержимое строки, то преобразование в перечисление проблем не вызывает. Таким образом, второе правило соблюдается.

Обычно мы проверяем код типа с помощью цепочки `else if` или `switch`. И то и другое заметить легко. Это свойство особенно верно ввиду того, что строки или константы выравниваются вертикально, поскольку мы проверяем одну переменную несколько раз.

13.5.3. Добавление в код магических чисел

Итак, пойдем еще на шаг далее и используем константы не только в качестве кода типа. Если я занят, экспериментирую или хочу подчеркнуть, что некий код требует рефакторинга, то не стесняюсь добавлять магические числа прямо в код. Чаще всего я делаю это, когда его пишу; константы я встраиваю очень редко.

Используя эту технику, мы рискуем уничтожить информацию, так что нужно быть внимательными. Если константа имеет плохое или ошибочное имя, то она не добавляет информации, и я без проблем ее встраиваю. Если я не могу определить, несет ли имя информацию или же я просто о нем чего-то не знаю, то всегда при встраивании константы добавляю комментарий, гарантируя соблюдение первого правила.

Листинг 13.11. До

```
const FOUR_THIRDS = 4/3;
class Sphere {
  volume() {
    let result = FOUR_THIRDS;
    for (let i = 0; i < 3; i++)
      result = result * this.radius;
    return result * Math.PI;
  }
}
```

Листинг 13.12. После

```
class Sphere {
  volume() {
    let result = 4/3;
    for (let i = 0; i < 3; i++)
      result = result * this.radius;
    return result * 3.141592653589793;
  }
}
```

Константы
встроены

Если оказывается, что магические числа должны быть константами, то их можно без проблем извлечь повторно. Следовательно, дальнейший рефакторинг не оказывается существенно усложненным.

Лучше всего это преобразование проявляет себя в третьем правиле. Практически все реагируют на присутствие в коде магических чисел. Если команда программистов едина в идее не просто извлекать константы, но также исправлять весь метод, то такой подход эффективно выделит нужный код.

13.5.4. Добавление в код комментариев

Как уже говорилось, для сохранения информации можно использовать комментарии. При этом они служат двум целям, кроме того, привлекают внимание, по крайней мере если мы следуем рекомендации из главы 8 и позже удаляем их. Как

мы видели в начале части I, прекрасным сигналом может стать комментарий, который должен стать в будущем именем метода.

Листинг 13.13. До

```
function subMin(arr: number[][]){
  let min = Number.POSITIVE_INFINITY;
  for (let x = 0; x < arr.length; x++) {
    for(let y = 0; y < arr[x].length; y++)
    {
      min = Math.min(min, arr[x][y]);
    }
  }
  for (let x = 0; x < arr.length; x++) {
    for(let y = 0; y < arr[x].length; y++)
    {
      arr[x][y] -= min;
    }
  }
  return min;
}
```

Листинг 13.14. После

```
function subMin(arr: number[][]){
  // Find min
  let min = Number.POSITIVE_INFINITY;
  for (let x = 0; x < arr.length; x++) {
    for(let y = 0; y < arr[x].length; y++) {
      min = Math.min(min, arr[x][y]);
    }
  }
  // Sub from each element
  for (let x = 0; x < arr.length; x++) {
    for(let y = 0; y < arr[x].length; y++) {
      arr[x][y] -= min;
    }
  }
  return min;
}
```

Комментарии, которые могут
(и должны) быть именами методов

Сложно уничтожить информацию, добавляя что-либо. И все же это возможно, если добавляемая в комментарий информация окажется намеренно сбивающей с толку. Так что если мы верим, что вносим в комментарий точную информацию, то проблем возникнуть не должно и первое правило будет соблюдено.

Добавление комментариев, которые могут стать именами методов, — это отличный способ просигнализировать о том, откуда нужно начинать будущий рефакторинг. Таким образом, одновременно обеспечивается простая точка входа и подсказываются имена методов тому, кто будет позже выполнять рефакторинг. Второе правило отлично соблюдается.

Большинство программ-редакторов выделяют комментарии отдельным цветом и иногда стилем, чтобы их было проще заметить. Но даже если не принимать это во внимание, то, следуя совету из главы 8, комментарии надо добавлять редко, тогда они будут бросаться в глаза.

13.5.5. Добавление в код пробелов

Еще один способ, которым можно выделить предполагаемое место для разделения метода, — это вставка пробела. Подобно комментариям, этот метод был также использован в части I книги. Отличается он тем, что здесь не нужно предлагать имя метода. Добавление пробела полезно, когда мы видим структуру, но недостаточно ее понимаем, чтобы назвать. Тем не менее, вдобавок к группировке инструкций, пустые строки могут служить и для группировки полей, и для предположения области для инкапсуляции данных.

Поскольку этот подход схож с комментариями, то с его помощью также можно намеренно ввести в заблуждение. В приведенном ниже примере специально помещен такой пробел в выражение, чтобы при чтении его было легко понять неправильно. Того же эффекта можно достичь через группировку инструкций или полей. Но поскольку мы, конечно, преследуем благие намерения, то проблем с соблюдением первого правила возникнуть не должно.

Листинг 13.15. До

```
let cursor = cursor+1 % arr.length;
```

Листинг 13.16. После

```
let cursor = (cursor + 1) % arr.length; ←
```

Явные скобки необходимы, потому что операция деления по модулю имеет тот же приоритет выполнения, что и умножение

Если использовать пустые строки для группировки инструкций, то становится виднее, где извлекать метод (3.2.1). Если использовать этот подход для группировки полей, то будет проще понять, где инкапсулировать данные (6.2.3). В любом случае пустые строки оказываются полезны.

Разработчики хорошо замечают паттерны, а пустые строки являются очень заметным паттерном, так как выделяются, подобно абзацам в книге.

13.5.6. Группировка элементов на основе имен

Еще один способ, которым можно указать в коде претендентов на инкапсуляцию, — это группировка элементов с общими аффиксами. Большинство разработчиков делают это автоматически, потому что так код выглядит приятнее. Но после прочтения главы 6 уже стало понятно, насколько эта техника может также быть полезна и в целях рефакторинга.

Листинг 13.17. До

```
class PopupWindow {
    private windowPosition: Point2d;
    private hasFocus: number;
    private screenWidth: number;
    private screenHeight: number;
    private windowSize: Point2d;
}
```

Листинг 13.18. После

```
class PopupWindow {
    private windowPosition: Point2d;
    private windowSize: Point2d;
    private hasFocus: number;
    private screenWidth: number;
    private screenHeight: number;
}
```

Гораздо проще заметить общий префикс window

Этот метод опасно применять в тех редких случаях, когда не работает правило «Всегда избегать общих аффиксов» (6.2.1). В любой другой ситуации с его помощью можно акцентировать информацию, делая аффиксы более заметными.

Общие аффиксы подпадают под конкретное правило, указывающее на конкретный шаблон рефакторинга «Инкапсуляция данных». Значит, когда мы видим общие аффиксы, нам просто нужно следовать шаблонам и правилам, что довольно просто. Как уже говорилось, люди склонны размещать аффиксы рядом по наитию, потому что они сильно привлекают внимание.

13.5.7. Добавление контекста в имена

Если имена методов и полей еще не имеют общих аффиксов, то можно намеренно ввести соответствующие аффиксы. Добавление аффикса может стать отчетливым сигналом само по себе, но если нам нужно усилить акцент, то можно разделить имя еще и нижним подчеркиванием.

Листинг 13.19. До

```
function avg(arr: number[]) {
    return sum(arr) / size(arr);
}
function size(arr: number[]) {
    return arr.length;
}
function sum(arr: number[]) {
    let sum = 0;
    for (let i = 0; i < arr.length; i++)
        sum += arr[i];
    return sum;
}
```

Листинг 13.20. После

```
function avg_ArrUtil(arr: number[]) {
    return sum_ArrUtil(arr)/size_ArrUtil(arr);
}
function size_ArrUtil(arr: number[]) {
    return arr.length;
}
function sum_ArrUtil(arr: number[]) {
    let sum = 0;
    for (let i = 0; i < arr.length; i++)
        sum += arr[i];
    return sum;
}
```

Добавление
контекста
к имени метода

При этом нужно действовать осторожно, чтобы добавляемый контекст оказался точным. С другой стороны, даже если в итоге придется инкапсулировать совместно методы и поля, которые вместе быть не должны, то можно будет позже разделить этот класс, дополнительно инкапсулировав те два несвязанных класса.

Как и в предыдущем правиле, здесь происходит движение напрямую к правилу об общих аффиксах и соответствующему рефакторингу. К тому же доработка имен всегда оказывается полезным приемом.

Объединение общих аффиксов делает их максимально заметными, а значит, эта техника отлично сочетается с предыдущей. Если же нет времени на поиск всех методов с одинаковым аффиксом или на их группировку, то для придания им заметности можно нарушить общепринятый стиль написания, о чем говорилось во введении к этой технике.

13.5.8. Создание длинных методов

Если выясняется, что некоторые методы извлечены недостаточно хорошо, можно встроить их, сформировав один длинный метод. Длинные методы являются предупреждением для большинства разработчиков, явно сигнализируя о том, что здесь требуется внимание.

Листинг 13.21. До

```
function animate() {
  handleChosen();
  handleDisplaying();
  handleCompleted();
  handleMoving();
}
function handleChosen() {
  if (value >= threshold
    && banner.state === State.Chosen) {
    // ...
  }
}
function handleDisplaying() {
  if (value >= target
    && banner.state === State.Displaying) {
    // ...
  }
}
function handleCompleted() {
  if (banner.state === State.Completed) {
    // ...
  }
}
function handleMoving() {
  if (banner.state === State.Moving
    && banner.target === banner.current) {
    // ...
  }
}
```

Листинг 13.22. После

```
function animate() {
  if (value >= threshold
    && banner.state === State.Chosen) {
    // ...
  }
  if (value >= target
    && banner.state === State.Displaying) {
    // ...
  }
  if (banner.state === State.Completed) {
    // ...
  }
  if (banner.state === State.Moving
    && banner.target === banner.current) {
    // ...
  }
}
```

Проще заметить, что они все относятся к banner.state

У исходных методов были имена, и если нет уверенности, что они ошибочны, то их информацию нужно сохранить. Для этого можно использовать комментарии, что также повысит видимость.

Когда методы извлечены вразрез с подобающей внутренней структурой, они могут затруднить дальнейший рефакторинг. Встраивание таких методов позволяет повторно проанализировать и определить правильную структуру.

Длинные методы обнаружить не так просто, как другие рассмотренные признаки. Тем не менее разработчики обычно их замечают и фиксируют их

местонахождение. Программисты запоминают их, чтобы либо избегать, либо учитывать как признак плохого кода. Так что недостаток заметности длинных методов вполне компенсируется их запоминаемостью.

13.5.9. Добавление в методы большого числа параметров

Добавление в метод большого числа входных параметров — это одна из моих любимых техник сигнализирования о необходимости рефакторинга. Помимо привлечения внимания к местам определения метода, становится невозможно не заметить и места его вызова.

Есть два распространенных способа, которыми люди избегают наличия большого количества параметров. Первый мы разбирали в главе 7, когда помещали параметры в нетипизированную структуру вроде `HashMap`, тем самым оговаривая компилятор. Второй способ — это создание объекта данных или структуры. Здесь значения уже именованы и типизированы. Однако такие классы обычно не соответствуют внутренней структуре, поэтому они не устраняют запах, а лишь его маскируют. Получающийся в результате обоих подходов код нужно переделывать.

Листинг 13.23. До, версия 1: карта

```
function stringConstructor(
  conf: Map<string, string>,
  parts: string[]) {
  return conf.get("prefix")
    + parts.join(conf.get("joiner"))
    + conf.get("postfix");
}
```

Листинг 13.24. До, версия 2: объект данных

```
class StringConstructorConfig {
  constructor(
    public readonly prefix: string,
    public readonly joiner: string,
    public readonly postfix: string) { }
}
function stringConstructor(
  conf: StringConstructorConfig,
  parts: string[]) {
  return conf.prefix
    + parts.join(conf.joiner)
    + conf.postfix;
}
```

Листинг 13.25. После

```
function stringConstructor(
  prefix: string,
  joiner: string,
  postfix: string,
  parts: string[]) {
  return prefix + parts.join(joiner) + postfix;
}
```

Переделав объект данных или структуру в длинный список параметров, мы сохраним и типы, и имена. Если сделать из `Map` множество параметров, то ключи станут именами переменных, и этим можно даже добавить информацию в виде явных типов. В обоих случаях риск уничтожения информации отсутствует.

Удаление длинного списка параметров зачастую требует некоторого рефакторинга в виде создания классов и перемещения в них кода, что позволяет постепенно определить, какие параметры являются связанными и должны находиться в одном классе. Но обычно преобразование объектов данных или хеш-карт в параметры не делает рефакторинг сложнее.

Самая же сильная сторона этого способа — в его заметности. Как уже говорилось, и место определения метода, и все места его вызовов открыто заявляют о необходимости рефакторинга. По сути, во всем коде оказываются разбросаны небольшие дорожные указатели, направляющие нас к проблемному методу.

13.5.10. Использование геттеров и сеттеров

Еще один подход, привносящий подобные дорожные указатели, — это использование геттеров и сеттеров вместо глобальных переменных или публичных полей. С помощью геттеров и сеттеров можно легко инкапсулировать данные и обращаться к ним. В свою очередь, эти геттеры и сеттеры должны исчезать по мере насыщения инкапсулирующего класса перемещаемым в него кодом.

Листинг 13.26. До

```
let screenWidth: number;  
let screenHeight: number;
```

Листинг 13.27. После

```
class Screen {  
  constructor(  
    private width: number,  
    private height: number) { }  
  getWidth() { return this.width; }  
  getHeight() { return this.height; }  
}  
let screen: Screen;
```

Этот метод также является аддитивным, поскольку код добавляется, а не изменяется и не удаляется. В результате риск утратить информацию в ходе преобразования отсутствует.

При рефакторинге подобных данных инкапсуляция зачастую выступает первым шагом, так как позволяет не только упростить его, но и сокращает необходимые для него усилия.

По общепринятому соглашению геттеры и сеттеры должны сопровождаться приставками `get` или `set` соответственно. Это синтаксическое соглашение упрощает их обнаружение в местах определения и точках вызова, что аналогично использованию большого числа параметров.

РЕЗЮМЕ

- С помощью плохого кода можно сигнализировать о проблемах в рабочих процессах, таких как нехватка времени или особенности приоритета.
- Базу кода следует разделять на безупречный и legacy-код. Безупречные части обычно сохраняют качество дольше.
- Для определения плохого кода нет идеального способа, но есть четыре популярных средства: правила из этой книги, запахи кода, цикломатическая и когнитивная сложность.
- Следуя трем правилам, можно безопасно увеличить разрыв между безупречным и legacy-кодом:
 - никогда не уничтожать полезную информацию;
 - содействовать дальнейшему рефакторингу;
 - делать проблемы более заметными.
- К примерам конкретных способов применения вышеназванных правил относятся:
 - использование перечислений;
 - использование целых чисел и строк в качестве кода типа;
 - добавление в код магических чисел;
 - добавление в код комментариев;
 - добавление в код пробелов;
 - группировка элементов на основе их имен;
 - добавление в имена контекста;
 - создание длинных методов;
 - добавление в методы большого числа параметров;
 - использование геттеров и сеттеров.

14

Подведение итогов

В этой главе

- ✓ Анализ пройденного путешествия.
- ✓ Раскрытие внутренних принципов книги.
- ✓ Рекомендации по продолжению путешествия.

В этой главе мы сначала еще раз кратко пересмотрим все, что изучили по ходу книги, чтобы освежить в памяти этот длинный путь. Затем я объясню главные идеи и принципы, послужившие основой для написания, и расскажу, как вы можете задействовать их для решения аналогичных задач. Завершится же глава рекомендациями по наиболее естественному продолжению движения по этому пути.

14.1. КРАТКИЙ ОБЗОР ПРОЙДЕННОГО

Когда вы начали читать эту книгу, то наверняка либо совсем не имели представления о рефакторинге, либо ваше представление о нем сильно отличалось от приводимых в ней трактовок. Я надеюсь, что мой труд сделает рефакторинг доступнее и эффективнее для многих людей. Мне хотелось понизить порог вхождения на территорию сложных принципов, таких как запахи кода, задействование возможностей компилятора, переключение функционала и др.

Чем богаче наш язык, тем тоньше и ярче мы воспринимаем окружающий мир, и я надеюсь, что мне удалось обогатить ваш словарный запас названиями правил и шаблонов рефакторинга.

14.1.1. Введение: мотивация

В первых двух главах мы изучили, что такое рефакторинг, почему он важен и когда нужно акцентировать на нем внимание. Закладывая основу, мы определили цели рефакторинга: снижение хрупкости путем локализации инвариантов, повышение гибкости за счет сокращения зацепления кода и углубление понимания предметной области программного обеспечения.

14.1.2. Часть I: конкретизирование

В части I мы прошлись по, казалось бы, неплохой базе кода и улучшили ее. Используя набор правил в качестве опоры, мы сосредотачивали свое внимание только на нужных аспектах и избегали погружения в подробности, что отняло бы неоправданно много времени. Параллельно с освоением правил мы составили небольшой каталог мощных шаблонов рефакторинга.

Сначала мы научились разбивать длинные функции. Затем узнали, как заменять код типа классами, что позволяет преобразовывать функции в методы за счет их перемещения в классы. Расширив базу кода, мы перешли к объединению `if`, функций и классов. В завершение части I мы разобрали продвинутые шаблоны рефакторинга, способствующие инкапсуляции.

14.1.3. Часть II: расширение горизонтов

Разобравшись с потоком рефакторинга и сформировав устойчивое понимание того, что и как нужно рефакторить, мы перешли на более высокий уровень абстракции. В части II вместо обсуждения конкретных правил и шаблонов рефакторинга мы изучили множество социотехнических аспектов, влияющих на рефакторинг и качество кода. При этом мы разобрали темы, касающиеся корпоративной культуры, навыков и инструментов, закрепив все это в действенных рекомендациях и выводах.

Среди изученных в части II инструментов были компиляторы, переключение функционала, канбан, теория ограничений и другие. Мы разобрали ряд изменений в культуре ведения разработок, а именно подходы к удалению, добавлению и ухудшению кода. Завершили же мы эту часть знакомством с конкретными навыками работы с кодом, такими как раскрытие структуры и безопасная оптимизация производительности.

14.2. РАСКРЫТИЕ ВНУТРЕННЕЙ ФИЛОСОФИИ

В этой книге содержится масса полезной информации — слишком много для удержания в уме одним человеком. К счастью, если вы усвоите внутренние принципы, то вам не придется запоминать все специфические детали для их эффективного применения. В этой связи я хочу рассказать вам, как сам воспринимаю и использую правила, а также все остальные принципы книги.

14.2.1. Поиск все меньших шагов

Эта книга опирается на тот же основополагающий принцип, что и разработка через тестирование и прочие подходы: совершать небольшие шаги, чтобы существенно сократить риск ошибок. Я никогда не показываю одно лишь конечное состояние, потому что именно на пути к нему и таятся все сложности. Возможность разбить большую задачу на небольшие части является важным элементом программирования. Этот же принцип можно использовать при рассмотрении глобальной трансформации. И здесь тоже мы находим небольшие трансформации, которые можно объединить для получения глобального результата.

В главе 13 и во всех правилах из части I были разобраны шаги, которые принимаются, когда конечное состояние неизвестно. Небольшие шаги, которые сосредоточены на переходе от одного рабочего состояния к другому. Это называется «из зеленого в зеленое». Как правило, это означает, что нам нужно пройти несколько промежуточных этапов, на каждом из которых добиваться минимальных улучшений.

Помимо снижения рисков, переход от зеленого к зеленому дает нам гораздо больше гибкости для попутного или вынужденного изменения направления. Так, если будет обнаружено нечто важное, то для переключения внимания на него придется лишь дойти до следующего зеленого состояния. Если мы вдруг получаем срочный запрос на исправление, то можем сделать `git reset` обратно к последнему предыдущему зеленому состоянию, пожертвовав минимальным объемом проделанной работы. Нам приходится не просто переключать ветки и позже опять возвращаться к сделанному, но и сбрасывать проделанный рефакторинг, потому что когда мы находимся в его середине, то вынуждены удерживать в уме множество тонких нитей. Если в разгаре рефакторинга переключиться на другой контекст, то потом мы эти нити уже вряд ли вспомним и риск внести ошибки сильно возрастет. Контекст следует изменять, только переходя из зеленых состояний и возвращаясь к ним.

Мы также обсудили, как разбивать трансформации, требующие изменений в культуре и коде, на небольшие шаги между стабильными состояниями. В главе 10,

изучая переключение функционала, мы рассмотрели эту технологию и разобрали соответствующие шаги адаптивирования необходимой культуры. На верхнем уровне я рекомендовал выработать рефлекс добавления и удаления инструкций `if` во всех изменениях. Только когда эта техника становится естественно присущей нам, можно начинать использовать ее преимущества в продакшен-средах. Если же сразу поспешить с этим переходом, то возникнет риск упустить в новом коде некоторые переключатели `if` и по случайности либо выдать сырой продукт, либо внести в него ошибки.

14.2.2. Поиск внутренней структуры

Мы очень много говорили о структуре. Фактически вся глава 11 была посвящена именно ей. Когда я провожу рефакторинг, то представляю себя скульптором, работающим с глиной. Он начинает с бесформенного куска материала и постепенно формирует его, раскрывая внутреннюю структуру. Я говорю «глина», потому что считаю, что код слишком эластичен и обратим и сравнение работы с ним с резкой камня не очень подходит. Если же этот нюанс «эластичности материала» в расчет не брать, то суть идеи прекрасно выразил Микеланджело.

В каждом каменном блоке заключена статуя,
раскрытие которой является задачей скульптора.

Микеланджело Буонарроти

В раскрытии внутреннего кода мне помогает один интересный прием, суть которого обширно раскрывалась в части I книги: с помощью строк я определяю, где должны находиться методы. Затем с помощью этих методов я определяю, где должны находиться классы. На практике же я иду еще дальше и позволяю классам указывать, где должны находиться пространства имен или пакеты. Суть приема в том, чтобы начать изнутри и затем надстраивать изменения все более и более абстрактными слоями. Так что пусть лучше у меня будет на один метод больше, чем одного не хватит. Один метод может определить разницу в наличии или отсутствии общих аффиксов, а значит, и еще одного класса.

14.2.3. Использование правил для совместной работы

Как и везде в реальном мире, в нашей области нет универсального средства. Нет полноценной и простой модели. Правила этой книги не исключение, они не постулаты и на самом деле являются инструментами, а не законами. Будет фатальной ошибкой применять их как жесткие инструкции или, что еще хуже,

использовать для понукания членов команды. Как уже говорилось в предыдущей главе, чувство психологической безопасности в команде разработчиков является основополагающим. Если правила помогают вам чувствовать себя во время рефакторинга безопасно и уверенно, то это хорошо. Если же они используются для того, чтобы тыкать в них друг друга носом, то это, конечно, плохо. Правила — это хорошая основа для обсуждения качества кода. Они являются отличной отправной точкой и служат прекрасным средством для понимания необходимости в рефакторинге и выработки мотивации к его изучению.

14.2.4. Интересы команды важнее личных интересов

Продолжая эту тему, я хочу подчеркнуть важность команды. Разработка программного обеспечения — это командная деятельность. Согласно принципам DevOps и гибкой разработки мы должны сосредоточиться на сотрудничестве. Легко пасть жертвой идеи о том, что эффективность могут повысить отдельные разработчики, работая параллельно. Подобная организация труда приводит к формированию «силосов знаний», то есть от распараллеливания обычно больше вреда, чем пользы. Прекрасным примером более выгодной коллективной работы являются техники парного или ансамблевого программирования. При верной реализации подобные действия помогают распределять знания, навыки и ответственность, что ведет к повышению взаимного доверия и приверженности общей цели. Вот как гласит африканская поговорка.

Если хочешь идти быстро — иди в одиночку.
Если хочешь идти далеко — идите вместе.

Африканская поговорка

Если сказать иначе, продукт доставляет команда, а не отдельные люди. Когда меня спрашивают: «Эта строка не слишком длинная?» или «А вот это плохо?» — то я отвечаю такими вопросами.

1. «А ваши разработчики понимают это?»
2. «Они этим довольны?»
3. «А есть более простая версия, которая не нарушает никаких требований производительности/безопасности?»

Вся команда должна принять на себя обязательство за всю базу кода, над которой она работает. Мы хотим изменять код быстро и уверенно, а значит, нужно разобраться со всем, что этому мешает.

14.2.5. Простота важнее универсальности

Если вы озадачитесь выработкой собственных правил, а я вам очень советую это сделать, то должны следовать важному принципу. Когда мы видим код, который кажется плохим, и хотим создать правило, его запрещающее, легко угодить в ловушку, стараясь придумать нечто универсальное. Такой подход приведет к созданию неопределенных и обобщенных правил, во многом подобных запахам кода. Они окажутся очень полезными и впечатляюще проработаны, но многие из них будут лишены самого важного свойства: простоты применения.

В когнитивной психологии описываются две системы когнитивных задач, каждая со своими особенностями. Система 1 быстра, но ей недостает точности. Для использования этой системы практически не требуется энергии, поэтому наш мозг отдает предпочтение ей. Система 2 медленна и энергозатратна, хотя при этом точна. Есть классический тест, который демонстрирует эти системы в действии. Ответьте на вопрос: «По сколько животных каждого вида взял с собой в ковчег Моисей?» Если вы ответите «два», то это ответ вашей системы 1. Если же вы верно отметили, что ковчег заполнял Ной, а не Моисей, то это уже ответ системы 2.

В любое время мы можем одновременно выполнять несколько задач системы 1, например жевать жвачку и в то же время идти или ехать. А вот задачу системы 2 можно выполнять одновременно только одну, например говорить или писать текст. Многозадачность — это не свойственная человеку черта. Некоторые действительно могут быстро переключаться между задачами, но так как мы ничего не распараллеливаем, то для описанных процессов практического смысла в этом нет.

Программирование в первую очередь относится к решению задач, а значит, попадает в область системы 2. На протяжении книги я указывал, что разработчики уже затрачивают свой умственный ресурс на решаемую задачу, иногда вплоть до полного его исчерпания. Значит, любые правила, соблюдение которых мы хотим от них потребовать, должны быть просты, чтобы их можно было применять без лишних раздумий.

Если мы хотим вызвать изменение в поведении, то по шкале от «просто, но ошибочно» до «сложно, но верно» нам следует отдавать предпочтение простоте. Упрощение тоже может создавать проблемы. Однако здесь можно опереться еще на одно человеческое свойство: здравый смысл. Представление правил, подобных приведенным в книге, с оговоркой, что они являются ориентирами, а не законами, должно отбить у людей охоту следовать им бездумно.

14.2.6. Использование объектов или функций высшего порядка

На протяжении книги мы использовали множество объектов и классов. А теперь надо отметить, что практически во всех передовых языках программирования появилась возможность, которая избавляет нас от этих конструкций. Эту возможность можно встретить под разными именами: функции высшего порядка, лямбда-функции, делегаты, замыкания и стрелки. Некоторые из них упомянуты в книге, но я, признаюсь, старался держаться от них подальше. Это решение обусловлено лишь стремлением сделать стиль повествования максимально понятным программистам с любыми языковыми предпочтениями.

С точки зрения рефакторинга объект с одним методом и функция высшего порядка — это одно и то же. Если у объекта есть поля, то это замыкание. Связанность у них одинаковая. Функции выглядят более эффектно, но у некоторых могут вызывать сложности в прочтении, а значит, имеет смысл применить уже введенное ранее правило: использовать то, чему отдаст предпочтение ваша команда. Если хотите попрактиковаться, то пройдитесь по коду из части I и от-рефакторите его так, как в листинге далее.

Листинг 14.1. Объект

```
function remove(
  shouldRemove: RemoveStrategy)
{
  for (let y = 0; y < map.length; y++)
    for (let x = 0; x < map[y].length; x++)
      if (shouldRemove.check(map[y][x]))
        map[y][x] = new Air();
}

class Key1 implements Tile {
  // ...
  moveHorizontal(dx: number) {
    remove(new RemoveLock1());
    moveToTile(playerx + dx, playery);
  }
}

interface RemoveStrategy {
  check(tile: Tile): boolean;
}

class RemoveLock1
  implements RemoveStrategy
{
  check(tile: Tile) {
    return tile.isLock1();
  }
}
```

Сигнатура типа одного метода в RemoveStrategy

.check удален, поскольку здесь только один метод

Сигнатура типа одного метода в RemoveStrategy

Тело из RemoveLock1 в виде функции высшего порядка

Листинг 14.2. Функция высшего порядка

```
function remove(
  shouldRemove: (tile: Tile) => boolean)
{
  for (let y = 0; y < map.length; y++)
    for (let x = 0; x < map[y].length; x++)
      if (shouldRemove(map[y][x]))
        map[y][x] = new Air();
}

class Key1 implements Tile {
  // ...
  moveHorizontal(dx: number) {
    remove(tile => tile.isLock1());
    moveToTile(playerx + dx, playery);
  }
}
```

14.3. КУДА ДВИГАТЬСЯ ДАЛЬШЕ

Начатое путешествие в изучение рефакторинга можно продолжить в разных направлениях. Наиболее естественными продолжениями станут макроархитектура, микроархитектура и качество ПО. Ниже я дам рекомендации по каждому из этих направлений.

14.3.1. Микроархитектура

Микро-, она же внутрикомандная, архитектура являлась основным предметом этой книги, и переход к ней станет, вероятно, наиболее естественным. Эта область работает с такими понятиями, как связанность и хрупкость, начиная с выражений и вплоть до (но не включая их) публичных интерфейсов и разработки API. В этом направлении я склонен выделять два пути.

- Можно углубиться в более сложные и подробные запахи через книгу Роберта Мартина «Чистый код».
- Либо расширить арсенал шаблонов рефакторинга с помощью книги Мартина Фаулера «Рефакторинг».

14.3.2. Макроархитектура

Вы также можете выбрать освоение макро-, или межкомандной, архитектуры. Как говорилось в главе 11, в этой сфере доминирует закон Конвея, гласящий, что наша (макро-) архитектура будет отражать структуру взаимодействия внутри организации. Это означает ни много ни мало: чтобы повлиять на код, мы должны сосредоточиться на изменении людей. В качестве прекрасного пособия по организации команд и закону Конвея я рекомендую прочесть книгу *Team Topologies* Мэтью Скелтона (IT Revolution Press, 2019).

14.3.3. Качество программного обеспечения

И последний путь, который я хочу здесь упомянуть, — это изучение качества ПО. По всей книге мы затрагивали вопрос качества в различных контекстах, и оно проявляется в различных формах, соответствуя различным нуждам.

Группам разработчиков продукта, поставляющим программное обеспечение незнакомым с кодом людям, я рекомендую изучение тестирования. Рефакторинг встроен в технику разработки через тестирование, и, хотя достичь мастерства в этой области сложно, начинать ее легко. В качестве пособия мне больше всего нравится книга «Экстремальное программирование. Разработка через

тестирование»¹ Кента Бека. Несмотря на то что тестирование не гарантирует полной безопасности, оно решает большое количество проблем, которые могут возникнуть у пользователя.

Команды разработки платформ поставляют ПО другим программистам в виде библиотек, фреймворков или расширяемых инструментов. Им я рекомендую изучить теорию типов. С помощью современных языков можно выражать в системе типов множество сложных свойств и поручать проверку их валидности компилятору. В то же время типы помогают разрабатывать документацию, выступают в качестве руководства для пользователей программного продукта, а также гарантируют соблюдение определенных свойств, предотвращая ошибки. В этом отношении я советую прочесть «Типы в языках программирования» Бенджамина К. Пирса. В ней автор тактично вводит читателя в функциональное программирование и типы, давая инструменты и понимание, которые можно перенести на другие парадигмы программирования. Система безопасности типов является гарантом, хотя включает в себя лишь то, чему мы сами ее обучим.

Самые же амбициозные читатели могут посвятить себя изучению доказуемой корректности с помощью зависимых типов систем автоматического доказательства теорем. Доказуемая корректность является эталоном в сфере качества программного обеспечения. Вот только для выработки мастерства здесь требуется приложить немало усилий. К счастью, освоенные в этой области уроки легко переносятся на любые сферы программирования. Для чтения рекомендую книгу *Type-driven Development with Irdis* Эдвина Брэди (Manning, 2017), которая также построена на основе функционального программирования. Что касается спроса на предоставляемое этой дисциплиной качество, то я должен отметить, что сейчас, на момент написания книги, которую вы держите в руках, он невелик. Но перспективы развития этого направления, безусловно, интересны: для доказуемой корректности по-прежнему создаются новые языки программирования, например Lean. Так что можно надеяться, что соответствующее ПО найдет свое место, поскольку оно безусловно и охватывает все аспекты.

РЕЗЮМЕ

- Стремясь сделать рефакторинг более доступным, мы подчеркнули его важность, после чего перешли к изучению на примерах, используя конкретные правила и шаблоны. Затем мы расширили горизонты и обсудили социотехнические стороны, влияющие на качество кода.
- Внутренняя философия этой книги состоит в парадигме выполнения больших трансформаций путем их разделения на мелкие шаги между стабильными состояниями.

¹ Бек К. Экстремальное программирование. Разработка через тестирование. — СПб.: Питер, 2022.

- Осознавая, что структура зачастую оказывается скрыта, мы используем строки, чтобы определить, где должны находиться методы, с помощью которых уже определяем, где должны находиться классы.
- Правила должны использоваться в качестве подспорья командной работе и сотрудничеству. При этом во время рефакторинга главным ориентиром должен выступать здравый смысл.
- Правила и рекомендации в этой книге составлены для людей с учетом окружающей их среды и ситуации. Если мы хотим изменить поведение, то при выборе между простотой и корректностью должны склоняться к первой.

Надеюсь, что для вас эта книга оказалась интересной и полезной. Выражаю свою благодарность за ваше внимание.

Приложение

Установка инструментов

для части I

Для установки TypeScript мы используем среду Node.js, поэтому сначала нужно установить ее.

NODE.JS

1. Перейдите по адресу <https://nodejs.org/en> и скачайте версию LTS.
2. Проследуйте процессу установки.
3. Проверьте версию программы, открыв PowerShell (или другую консоль) и выполнив команду:

```
npm --version
```

Вернуться должно что-то вроде 6.14.6.

TYPESCRIPT

1. Откройте PowerShell и выполните:

```
npm install -g typescript
```

Эта команда задействует предоставляемый Node.js пакетный менеджер (npm), чтобы сделать `install` компилятора TypeScript глобально (-g), а не в локальный каталог.

2. Проверьте версию командой:

```
tsc --version
```

Вернуться должно что-то вроде Version 4.0.3.

VISUAL STUDIO CODE

1. Перейдите по адресу <https://code.visualstudio.com> и скачайте установщик.
2. Проследуйте указаниям процесса установки. При возникновении приглашения выбрать советую установить следующие флажки:
 - Add "Open with code" to the Windows Explorer File context menu;
 - Add "Open with code" to the Windows Explorer Directory context menu.

Они позволят открывать каталог или файл в Visual Studio Code просто щелчком правой кнопкой мыши.

GIT

1. Перейдите по ссылке <https://git-scm.com/downloads> и скачайте установщик.
2. Проследуйте указаниям процесса установки.
3. Проверьте версию продукта, запустив PowerShell и выполнив:

```
git --version
```

Вернуться должно что-то вроде `git version 2.24.0.windows.2`.

НАСТРОЙКА ПРОЕКТА TYPESCRIPT

1. Откройте консоль там, где хотите сохранять создаваемую игру.
 - `git clone https://github.com/thedrlambda/five-lines` скачивает исходный код игры;
 - `cd five-lines` переходит в каталог с игрой;
 - `tsc -w` компилирует TypeScript в JavaScript при каждом изменении.
2. Откройте `index.html` в браузере.

СОЗДАНИЕ ПРОЕКТА TYPESCRIPT

1. Откройте каталог с игрой в Visual Studio Code.
2. Выберите Terminal, а затем New Terminal.
3. Выполните команду `tsc -w`.
4. Теперь TypeScript будет компилировать изменения в фоновом режиме, и терминал можно закрыть.
5. При каждом внесении изменения подождите секунду, после чего можете обновлять `index.html` в браузере.

Инструкции о том, как победить в игре, показываются в браузере при ее запуске.

КАК НАСТРОИТЬ УРОВЕНЬ

Через код также можно изменить уровень, так что можете спокойно создавать собственные карты, обновляя массив в переменной `map`. Указываемые числа соответствуют типам клеток и поясняются в следующей таблице:

0	Воздух	2	Неразрушимый
1	Флакс	8	Желтый ключ
3	Игрок	9	Желтый замок
4	Камень	10	Синий ключ
6	Ящик	11	Синий замок

Числа 5 и 7 представляют падающие версии ящиков и камней, поэтому для создания уровней не используются. Если вам нужна подсказка, то попробуйте уровень ниже. Его задача — переместить оба ящика в нижний правый угол один поверх другого.

Листинг П.1. Вариация уровня

```
let playerx = 5;
let playery = 3;
let map: Tile[][] = [
  [2, 2, 2, 2, 2, 2, 2, 2],
  [2, 0, 4, 6, 8, 6, 2, 2],
  [2, 1, 1, 1, 1, 1, 2, 2],
  [2, 0, 0, 0, 4, 3, 0, 2],
  [2, 2, 9, 2, 2, 0, 0, 2],
  [2, 2, 2, 2, 2, 2, 2, 2],
];
```

Кратчайшим решением для этого уровня будет $\leftarrow \uparrow \uparrow \downarrow \leftarrow \leftarrow \downarrow \rightarrow \rightarrow \uparrow \leftarrow \leftarrow \downarrow \rightarrow \rightarrow \rightarrow \uparrow \leftarrow \downarrow \rightarrow$.

Краткий обзор правил

- Пять строк (3.1.1) — метод не должен содержать больше строк, чем необходимо для прохождения через основную структуру данных.
- Вызов или передача (3.1.1) — функция должна либо вызывать методы для объекта, либо передавать этот объект в качестве аргумента, но не то и другое сразу.
- if только в начале (3.5.1) — когда присутствует оператор `if`, он должен идти в функции первым.
- Никогда не использовать `if` с `else` (4.1.1) — никогда не использовать `if` с `else`, если только мы не выполняем проверку в отношении типа данных, который не контролируем.
- Никогда не использовать `switch` (4.2.4) — никогда не использовать `switch`. Исключение возможно, только если у вас нет `default`, в каждом `case` есть `return`, а компилятор выполняет проверку на исчерпывающую полноту.
- Наследовать только от интерфейсов (4.3.2) — наследовать только от интерфейсов в противоположность классам или абстрактным классам.
- Использовать чистые условия (5.3.2) — условия никогда не должны присваивать значения переменным, выбрасывать исключения или взаимодействовать с вводом-выводом.
- Избегать интерфейсов с единственной реализацией (5.4.3) — у вас не должно быть интерфейсов со всего одним реализующим классом.
- Не использовать геттеры или сеттеры (6.1.1) — не использовать методы, которые напрямую присваивают или возвращают нелогическое поле.
- Всегда избегать общих аффиксов (6.2.1) — в коде не должно быть методов или переменных с общими префиксами или суффиксами.

Кристиан Клаусен

Пять строк кода.

Роберт Мартин рекомендует

Перевел с английского Д. И. Брайт

Заведующая редакцией	<i>Ю. Сергиенко</i>
Ведущий редактор	<i>Н. Гринчик</i>
Литературный редактор	<i>Н. Викторова</i>
Художественный редактор	<i>В. Мостипан</i>
Корректоры	<i>О. Андриевич, Е. Павлович</i>
Верстка	<i>Г. Блинов</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сапсони́евский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 10.2022.

Наименование: книжная продукция.

Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск,
ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 24.08.22. Формат 70×100/16. Бумага офсетная.
Усл. п. л. 29,670. Тираж 1000. Заказ 0000.