



Роберт Мартин

# ЧИСТЫЙ КОД

Создание,  
анализ  
и рефакторинг

- Что такое «чистый код»?
- Как улучшить плохой код?
- Почему чистый код часто «портится»?
- Почему в написании кода так важны мелочи?

*Роберт Мартин*

**Чистый код: создание, анализ и рефакторинг.  
Библиотека программиста**

*Перевел с английского Е. Матвеев*

Заведующий редакцией  
Руководитель проекта  
Ведущий редактор  
Художественный редактор  
Корректор  
Верстка

*А. Кривцов  
А. Юрченко  
Ю. Сергиенко  
Л. Адуевская  
В. Листова  
Е. Егорова*

**Мартин Р.**

M29 Чистый код: создание, анализ и рефакторинг. Библиотека программиста. — СПб.: Питер, 2018. — 464 с.: ил.

ISBN 978-5-496-00487-9

Даже плохой программный код может работать. Однако если код не является «чистым», это всегда будет мешать развитию проекта и компании-разработчика, отнимая значительные ресурсы на его поддержку и «укрощение».

Эта книга посвящена хорошему программированию. Она полна реальных примеров кода. Мы будем рассматривать код с различных направлений: сверху вниз, снизу вверх и даже изнутри. Прочитав книгу, вы узнаете много нового о коде. Более того, вы научитесь отличать хороший код от плохого. Вы узнаете, как писать хороший код и как преобразовать плохой код в хороший.

Книга состоит из трех частей. В первой части излагаются принципы, паттерны и приемы написания чистого кода; приводится большой объем примеров кода. Вторая часть состоит из практических сценариев нарастающей сложности. Каждый сценарий представляет собой упражнение по чистке кода или преобразованию проблемного кода в код с меньшим количеством проблем. Третья часть книги — концентрированное выражение ее сути. Она состоит из одной главы с перечнем эвристических правил и «запахов кода», собранных во время анализа. Эта часть представляет собой базу знаний, описывающую наш путь мышления в процессе чтения, написания и чистки кода.

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

ISBN 978-0132350884 (англ.)

© Prentice Hall, Inc.

ISBN 978-5-496-00487-9

© Перевод на русский язык ООО Издательство «Питер», 2018

© Издание на русском языке, оформление

ООО Издательство «Питер», 2018

Права на издание получены по соглашению с Prentice Hall, Inc. Upper Sadle River, New Jersey 07458.

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

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

Изготовлено в России. Изготовитель: ООО «Питер Пресс».

Место нахождения и фактический адрес: 192102, Россия, город Санкт-Петербург, улица Андреевская, дом 3, литер А, помещение 7Н. Тел.: +78127037373.

Дата изготовления: 10.2017. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Подписано в печать 29.09.17. Формат 70x100/16. Бумага писчая. Усл. п. л. 37,410. Доп. тираж 2000. Заказ 0000

Отпечатано в полном соответствии с качеством предоставленных издательством материалов в Типография «Вятка». 610033, Киров, ул. Московская, 122.

# Содержание

<b>Предисловие</b> .....	<b>14</b>
<b>Введение</b> .....	<b>20</b>
<b>Глава 1. Чистый код</b> .....	<b>23</b>
Да будет код .....	24
Плохой код .....	25
Расплата за хаос .....	26
Грандиозная переработка .....	26
Отношение .....	27
Основной парадокс .....	28
Искусство чистого кода? .....	28
Что такое «чистый код»? .....	29
Мы — авторы .....	36
Правило бойскаута .....	37
Предыстория и принципы .....	37
Заключение .....	38
Литература .....	38
<b>Глава 2. Содержательные имена (Тим Оттингер)</b> .....	<b>39</b>
Имена должны передавать намерения программиста .....	40
Избегайте дезинформации .....	41
Используйте осмысленные различия .....	42
Используйте удобопроизносимые имена .....	44
Выбирайте имена, удобные для поиска .....	45
Избегайте схем кодирования имен .....	45
Венгерская запись .....	46
Префиксы членов классов .....	46
Интерфейсы и реализации .....	47
Избегайте мысленных преобразований .....	47
Имена классов .....	48
Имена методов .....	48
Избегайте остроумия .....	48
Выберите одно слово для каждой концепции .....	49
Воздержитесь от каламбуров .....	49
Используйте имена из пространства решения .....	50
Используйте имена из пространства задачи .....	50

Добавьте содержательный контекст .....	51
Не добавляйте избыточный контекст .....	53
Несколько слов напоследок .....	53
<b>Глава 3. Функции .....</b>	<b>55</b>
Компактность! .....	58
Блоки и отступы .....	59
Правило одной операции .....	59
Секции в функциях .....	60
Один уровень абстракции на функцию .....	61
Чтение кода сверху вниз: правило понижения .....	61
Команды switch .....	62
Используйте содержательные имена .....	64
Аргументы функций .....	64
Стандартные унарные формы .....	65
Аргументы-флаги .....	66
Бинарные функции .....	66
Тернарные функции .....	67
Объекты как аргументы .....	68
Списки аргументов .....	68
Глаголы и ключевые слова .....	68
Избавьтесь от побочных эффектов .....	69
Выходные аргументы .....	70
Разделение команд и запросов .....	70
Используйте исключения вместо возвращения кодов ошибок .....	71
Изолируйте блоки try/catch .....	72
Обработка ошибок как одна операция .....	72
Магнит зависимостей Error.java .....	73
Не повторяйтесь .....	73
Структурное программирование .....	74
Как научиться писать такие функции? .....	74
Завершение .....	75
Литература .....	78
<b>Глава 4. Комментарии .....</b>	<b>79</b>
Комментарии не компенсируют плохого кода .....	81
Объясните свои намерения в коде .....	81
Хорошие комментарии .....	81
Юридические комментарии .....	82
Информативные комментарии .....	82
Представление намерений .....	82
Прояснение .....	83
Предупреждения о последствиях .....	84
Комментарии TODO .....	85
Усиление .....	85
Комментарии Javadoc в общедоступных API .....	86
Плохие комментарии .....	86
Бормотание .....	86
Избыточные комментарии .....	87
Недостоверные комментарии .....	89

Обязательные комментарии .....	90
Журнальные комментарии .....	90
Шум .....	91
Опасный шум .....	93
Не используйте комментарии там, где можно использовать функцию или переменную .....	93
Позиционные маркеры .....	94
Комментарии за закрывающей фигурной скобкой .....	94
Ссылки на авторов .....	95
Закомментированный код .....	95
Комментарии HTML .....	96
Нелокальная информация .....	96
Слишком много информации .....	97
Неочевидные комментарии .....	97
Заголовки функций .....	97
Заголовки Javadoc во внутреннем коде .....	98
Пример .....	98
Литература .....	101
<b>Глава 5. Форматирование .....</b>	<b>102</b>
Цель форматирования .....	103
Вертикальное форматирование .....	103
Газетная метафора .....	104
Вертикальное разделение концепций .....	105
Вертикальное сжатие .....	106
Вертикальные расстояния .....	107
Вертикальное упорядочение .....	112
Горизонтальное форматирование .....	112
Горизонтальное разделение и сжатие .....	113
Горизонтальное выравнивание .....	114
Отступы .....	116
Вырожденные области видимости .....	117
Правила форматирования в группах .....	118
Правила форматирования от дядюшки Боба .....	118
<b>Глава 6. Объекты и структуры данных .....</b>	<b>121</b>
Абстракция данных .....	121
Антисимметрия данных/объектов .....	123
Закон Деметры .....	126
Крушение поезда .....	126
Гибриды .....	127
Скрытие структуры .....	127
Объекты передачи данных .....	128
Активные записи .....	129
Заключение .....	130
Литература .....	130
<b>Глава 7. Обработка ошибок (Майк Физерс) .....</b>	<b>131</b>
Используйте исключения вместо кодов ошибок .....	132
Начните с написания команды try-catch-finally .....	133

Используйте непроверяемые исключения .....	135
Передавайте контекст с исключениями .....	136
Определяйте классы исключений в контексте потребностей вызывающей стороны .....	136
Определите нормальный путь выполнения .....	138
Не возвращайте null .....	139
Не передавайте null .....	140
Заклучение .....	141
Литература .....	141
<b>Глава 8. Границы (Джеймс Гренинг) .....</b>	<b>142</b>
Использование стороннего кода .....	143
Исследование и анализ границ .....	145
Изучение log4j .....	145
Учебные тесты: выгоднее, чем бесплатно .....	147
Использование несуществующего кода .....	148
Чистые границы .....	149
Литература .....	149
<b>Глава 9. Модульные тесты .....</b>	<b>150</b>
Три закона TTD .....	151
О чистоте тестов .....	152
Тесты как средство обеспечения изменений .....	153
Чистые тесты .....	154
Предметно-ориентированный язык тестирования .....	157
Двойной стандарт .....	157
Одна проверка на тест .....	159
Одна концепция на тест .....	161
F.I.R.S.T. ....	162
Заклучение .....	163
Литература .....	163
<b>Глава 10. Классы (совместно с Джеффом Лангром) .....</b>	<b>164</b>
Строение класса .....	164
Инкапсуляция .....	165
Классы должны быть компактными! .....	165
Принцип единой ответственности (SRP) .....	167
Связность .....	169
Поддержание связности приводит к уменьшению классов .....	170
Структурирование с учетом изменений .....	176
Изоляция изменений .....	179
Литература .....	180
<b>Глава 11. Системы (Кевин Дин Уомплер) .....</b>	<b>181</b>
Как бы вы строили город? .....	182
Отделение конструирования системы от ее использования .....	182
Отделение main .....	184
Фабрики .....	184
Внедрение зависимостей .....	185

Масштабирование .....	186
Поперечные области ответственности .....	189
Посредники .....	190
АОП-инфраструктуры на «чистом» Java .....	192
Аспекты AspectJ .....	195
Испытание системной архитектуры .....	196
Оптимизация принятия решений .....	197
Применяйте стандарты разумно, когда они приносят очевидную пользу .....	197
Системам необходимы предметно-ориентированные языки .....	198
Заключение .....	199
Литература .....	199
<b>Глава 12. Формирование архитектуры .....</b>	<b>200</b>
Четыре правила .....	200
Правило № 1: выполнение всех тестов .....	201
Правила № 2–4: переработка кода .....	201
Отсутствие дублирования .....	202
Выразительность .....	204
Минимум классов и методов .....	206
Заклучение .....	206
Литература .....	206
<b>Глава 13. Многопоточность (Бретт Л. Шухерт) .....</b>	<b>207</b>
Зачем нужна многопоточность? .....	208
Мифы и неверные представления .....	209
Трудности .....	210
Защита от ошибок многопоточности .....	211
Принцип единой ответственности .....	211
Следствие: ограничивайте область видимости данных .....	211
Следствие: используйте копии данных .....	212
Следствие: потоки должны быть как можно более независимы .....	212
Знайте свою библиотеку .....	213
Потоково-безопасные коллекции .....	213
Знайте модели выполнения .....	214
Модель «производители-потребители» .....	214
Модель «читатели-писатели» .....	215
Модель «обедающих философов» .....	215
Остерегайтесь зависимостей между синхронизированными методами .....	216
Синхронизированные секции должны иметь минимальный размер .....	216
О трудности корректного завершения .....	217
Тестирование многопоточного кода .....	218
Рассматривайте неперiodические сбои как признаки возможных проблем многопоточности .....	218
Начните с отладки основного кода, не связанного с многопоточностью .....	219
Реализуйте переключение конфигураций многопоточного кода .....	219
Обеспечьте логическую изоляцию конфигураций многопоточного кода .....	219
Протестируйте программу с количеством потоков, превышающим количество процессоров .....	220
Протестируйте программу на разных платформах .....	220

Применяйте инструментовку кода для повышения вероятности сбоев .....	220
Ручная инструментовка .....	221
Автоматизированная инструментовка .....	222
Заключение .....	223
Литература .....	224
<b>Глава 14. Последовательное очищение .....</b>	<b>225</b>
Реализация Args .....	226
Как я это сделал? .....	233
Args: черновик .....	233
На этом я остановился .....	245
О постепенном усовершенствовании .....	246
Аргументы String .....	248
Заключение .....	286
<b>Глава 15. Внутреннее строение JUnit .....</b>	<b>287</b>
Инфраструктура JUnit .....	288
Заключение .....	302
<b>Глава 16. Переработка SerialDate .....</b>	<b>303</b>
Прежде всего — заставить работать .....	304
...Потом очистить код .....	306
Заключение .....	320
Литература .....	321
<b>Глава 17. Запахи и эвристические правила .....</b>	<b>322</b>
Комментарии .....	323
C1: Неуместная информация .....	323
C2: Устаревший комментарий .....	323
C3: Избыточный комментарий .....	323
C4: Плохо написанный комментарий .....	323
C5: Закомментированный код .....	324
Рабочая среда .....	324
E1: Построение состоит из нескольких этапов .....	324
E2: Тестирование состоит из нескольких этапов .....	324
Функции .....	325
F1: Слишком много аргументов .....	325
F2: Выходные аргументы .....	325
F3: Флаги в аргументах .....	325
F4: Мертвые функции .....	325
Разное .....	325
G1: Несколько языков в одном исходном файле .....	325
G2: Очевидное поведение не реализовано .....	326
G3: Некорректное граничное поведение .....	326
G4: Отключенные средства безопасности .....	326
G5: Дублирование .....	327
G6: Код на неверном уровне абстракции .....	328
G7: Базовые классы, зависящие от производных .....	329



---

G8: Слишком много информации	329
G9: Мертвый код	330
G10: Вертикальное разделение	330
G11: Непоследовательность	330
G12: Балласт	331
G13: Искусственные привязки	331
G14: Функциональная зависть	331
G15: Аргументы-селекторы	332
G16: Непонятные намерения	333
G17: Неверное размещение	334
G18: Неуместные статические методы	334
G19: Используйте пояснительные переменные	335
G20: Имена функций должны описывать выполняемую операцию	335
G21: Понимание алгоритма	336
G22: Преобразование логических зависимостей в физические	336
G23: Используйте полиморфизм вместо if/Else или switch/Case	338
G24: Соблюдайте стандартные конвенции	338
G25: Заменяйте «волшебные числа» именованными константами	339
G26: Будьте точны	340
G27: Структура важнее конвенций	340
G28: Инкапсулируйте условные конструкции	341
G29: Избегайте отрицательных условий	341
G30: Функции должны выполнять одну операцию	341
G31: Скрытые временные привязки	342
G32: Структура кода должна быть обоснована	343
G33: Инкапсулируйте граничные условия	343
G34: Функции должны быть написаны на одном уровне абстракции	344
G35: Храните конфигурационные данные на высоких уровнях	345
G36: Избегайте транзитивных обращений	346
Java	347
J1: Используйте обобщенные директивы импорта	347
J2: Не наследуйте от констант	347
J3: Константы против перечислений	348
Имена	349
N1: Используйте содержательные имена	349
N2: Выбирайте имена на подходящем уровне абстракции	351
N3: По возможности используйте стандартную номенклатуру	352
N4: Недвусмысленные имена	352
N5: Используйте длинные имена для длинных областей видимости	353
N6: Избегайте кодирования	353
N7: Имена должны описывать побочные эффекты	354
Тесты	354
T1: Нехватка тестов	354
T2: Используйте средства анализа покрытия кода	354
T3: Не пропускайте тривиальные тесты	354
T4: Отключенный тест как вопрос	355
T5: Тестируйте граничные условия	355
T6: Тщательно тестируйте код рядом с ошибками	355

T7: Закономерности сбоя часто несут полезную информацию .....	355
T8: Закономерности покрытия кода часто несут полезную информацию .....	355
T9: Тесты должны работать быстро .....	356
Заключение .....	356
Литература .....	356
<b>Приложение А. Многопоточность II .....</b>	<b>357</b>
Пример приложения «клиент/сервер» .....	357
Найдите свои библиотеки .....	367
Зависимости между методами могут нарушить работу многопоточного кода .....	370
Повышение производительности .....	375
Взаимная блокировка .....	377
Тестирование многопоточного кода .....	381
Средства тестирования многопоточного кода .....	384
Полные примеры кода .....	385
<b>Приложение Б. org.jfree.date.SerialDate .....</b>	<b>390</b>
<b>Приложение В. Перекрестные ссылки .....</b>	<b>455</b>
<b>Эпилог .....</b>	<b>458</b>
<b>Алфавитный указатель .....</b>	<b>459</b>

# 1

## ЧИСТЫЙ КОД



Вы читаете эту книгу по двум причинам. Во-первых, вы программист. Во-вторых, вы хотите повысить свою квалификацию как программиста. Отлично. Хороших программистов не хватает.

Эта книга посвящена хорошему программированию. Она полна реальных примеров кода. Мы будем рассматривать код с направлений: сверху вниз, снизу

вверх, и даже изнутри. К последней странице книги вы узнаете много нового о коде. Более того, вы научитесь отличать хороший код от плохого. Вы узнаете, как писать хороший код и как преобразовать плохой код в хороший.

## Да будет код

Возможно, кто-то скажет, что книга о коде отстала от времени — код сейчас уже не так актуален; вместо него внимание следует направить на модели и требования. Нам даже доводилось слышать мнение, что код как таковой скоро перестанет существовать. Что скоро весь код будет генерироваться, а не писаться вручную. Что программисты станут попросту не нужны, потому что бизнесмены будут генерировать программы по спецификациям.

Ерунда! Код никогда не исчезнет, потому что код представляет подробности требований. На определенном уровне эти подробности невозможно игнорировать или абстрагировать; их приходится определять. А когда требования определяются настолько подробно, чтобы они могли быть выполнены компьютером, это и есть программирование. А их определение есть код.

Вероятно, уровень абстракции наших языков продолжит расти. Я также ожидаю, что количество предметно-ориентированных языков продолжит расти. И это хорошо. Но код от этого существовать не перестанет. В самом деле, все определения, написанные на этих высокоуровневых, предметно-ориентированных языках, станут кодом! И этот код должен быть достаточно компактным, точным, формальным и подробным, чтобы компьютер мог понять и выполнить его.

Люди, полагающие, что код когда-нибудь исчезнет, напоминают математиков, которые надеются когда-нибудь обнаружить неформальную математическую дисциплину. Они надеются, что когда-нибудь будут построены машины, которые будут делать то, что мы хотим, а не то, что мы приказываем сделать. Такие машины должны понимать нас настолько хорошо, чтобы преобразовать набор нечетких потребностей в идеально выполняемые программы, точно отвечающие этим потребностям.

Но этого никогда не произойдет. Даже люди, со всей их интуицией и изобретательностью, не способны создавать успешные системы на основе туманных представлений своих клиентов. Если дисциплина определения требований нас чему-то научила, так это тому, что четко определенные требования так же формальны, как сам код, и могут использоваться как исполняемые тесты этого кода!

В сущности, код представляет собой язык, на котором в конечном итоге выражаются потребности. Мы можем создавать языки, близкие к потребностям. Мы можем создавать инструменты, помогающие нам обрабатывать и собирать эти потребности в формальные структуры. Но необходимая точность никогда не исчезнет — а следовательно, код останется всегда.

## Плохой код

Недавно я читал предисловие к книге Кента Бека «Implementation Patterns» [Beck07]. Автор говорит: «...эта книга базируется на довольно непрочной предпосылке: что хороший код важен...» Непрочная предпосылка? Не согласен! На мой взгляд, эта предпосылка является одной из самых мощных, основополагающих и многогранных положений нашего ремесла (и я думаю, что Кенту это известно). Мы знаем, что хороший код важен, потому что нам приходилось так долго мириться с его отсутствием.

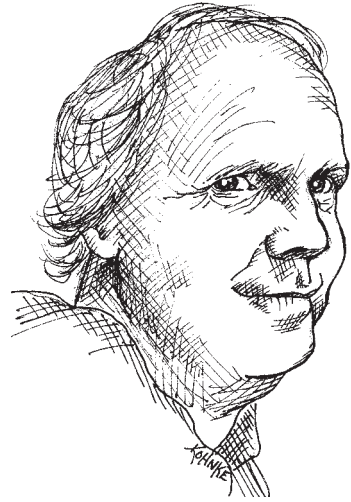
Одна компания в конце 80-х годов написала приложение-бестселлер. Приложение стало чрезвычайно популярным, многие профессионалы покупали и использовали его. Но потом циклы выпуска новых версий стали затягиваться. Ошибки не исправлялись между версиями. Время загрузки росло, а сбои происходили все чаще. Помню тот день, когда я в раздражении закрыл этот продукт и никогда не запускал его. Вскоре эта компания разорилась.

Два десятилетия спустя я встретил одного из работников той компании и спросил его, что же произошло. Ответ подтвердил мои опасения. Они торопились с выпуском продукта на рынок и не обращали внимания на качество кода. С добавлением новых возможностей код становился все хуже и хуже, пока в какой-то момент не вышел из-под контроля. *Плохой код привел к краху компании.*

Плохой код когда-нибудь мешал вашей работе? Любой сколько-нибудь опытный программист неоднократно попадал в подобную ситуацию. Мы продираемся через плохой код. Мы вязнем в хитросплетении ветвей, попадаем в скрытые ловушки. Мы с трудом прокладываем путь, надеясь получить хоть какую-нибудь подсказку, что же происходит в коде; но не видим вокруг себя ничего, кроме новых залежей невразумительного кода.

Конечно, плохой код мешал вашей работе. Почему же вы писали его? Пытались поскорее решить задачу? Торопились? Возможно. А может быть, вам казалось, что у вас нет времени качественно выполнить свою работу; что ваше начальство будет недоволено, если вы потратите время на чистку своего кода. А может, вы устали работать над программой и вам хотелось поскорее избавиться от нее. А может, вы посмотрели на список запланированных изменений и поняли, что вам необходимо поскорее «прикрутить» этот модуль, чтобы перейти к следующему. Такое бывало с каждым.

Каждый из нас смотрел на тот хаос, который он только что сотворил, и решал оставить его на завтра. Каждый с облегчением видел, что бестолковая программа работает, и решал, что рабочая мешанина — лучше, чем ничего. Каждый обещал



себе вернуться и почистить код... потом. Конечно, в те дни мы еще не знали закон Леблана: *потом равносильно никогда*.

## Расплата за хаос

Если вы занимались программированием более двух-трех лет, вам наверняка доводилось вязнуть в чужом — или в своем собственном — беспорядочном ходе. Замедление может быть весьма значительным. За какие-нибудь год-два группы, очень быстро двигавшиеся вперед в самом начале проекта, начинают ползти со скоростью улитки. Каждое изменение, вносимое в код, нарушает работу кода в двух-трех местах. Ни одно изменение не проходит тривиально. Для каждого дополнения или модификации системы необходимо «понимать» все хитросплетения кода — чтобы в программе их стало еще больше. Со временем неразбериха разрастается настолько, что справиться с ней уже не удастся. Выхода просто нет.

По мере накопления хаоса в коде производительность группы начинает снижаться, асимптотически приближаясь к нулю. В ходе снижения производительности начальство делает единственное, что оно может сделать: подключает к проекту новых работников в надежде повысить производительность. Но новички ничего не понимают в архитектуре системы. Они не знают, какие изменения соответствуют намерениям проектировщика, а какие им противоречат. Более того, они — и все остальные участники группы — находятся под страшным давлением со стороны начальства. В спешке они работают все небрежнее, отчего производительность только продолжает падать (рис. 1.1).

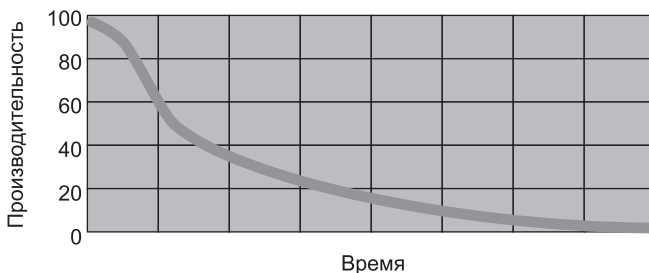


Рис. 1.1. Зависимость производительности от времени

## Грандиозная переработка

В конечном итоге группа устраивает бунт. Она сообщает начальству, что не может продолжать разработку отвратительной кодовой базы, и требует переработки архитектуры. Начальство не хочет тратить ресурсы на полную переработку проекта, но не может отрицать, что производительность просто ужасна. Со временем

начальство поддается на требования разработчиков и дает разрешение на проведение грандиозной переработки.

Набирается новая «ударная группа». Все хотят в ней участвовать, потому что проект начинается «с нуля». Разработчики будут строить «на пустом месте», и создадут нечто воистину прекрасное. Но в «ударную группу» отбирают только самых лучших и умных. Всем остальным приходится сопровождать текущую систему.

Между двумя группами начинается гонка. «Ударная группа» должна построить новую систему, которая делает то же самое, что делала старая. Более того, она должна своевременно учитывать изменения, непрерывно вносимые в старую систему. Начальство не заменяет старую систему до тех пор, пока новая система не будет полностью повторять ее функциональность.

Такая гонка может продолжаться очень долго. Мне известны случаи, как она продолжалась по 10 лет. И к моменту ее завершения оказывалось, что исходный состав давно покинул «ударную группу», а текущие участники требовали переработать новую систему, потому что в ней творился сущий хаос.

Если вы сталкивались хотя бы с некоторыми частями истории, которую я сейчас поведал, то вы уже знаете, что поддержание чистоты кода не только окупает затраченное время; оно является делом профессионального выживания.

## Отношение

Вам доводилось продирааться через код настолько запутанный, что у вас уходило недели на то, что должно было занять несколько часов? Вы видели, как изменение, которое вроде бы должно вноситься в одной строке, приходится вносить в сотнях разных модулей? Эти симптомы стали слишком привычными.

Почему это происходит с кодом? Почему хороший код так быстро загнивает и превращается в плохой код? У нас обычно находится масса объяснений. Мы жалуемся на изменения в требованиях, противоречащие исходной архитектуре. Мы стенаем о графиках, слишком жестких для того, чтобы делать все, как положено. Мы сплетничаем о глупом начальстве, нетерпимых клиентах и бестолковых типах из отдела маркетинга. Однако вина лежит вовсе не на них, а на нас самих. Дело в нашем непрофессионализме.

Возможно, проглотить эту горькую пилюлю будет непросто. Разве мы виноваты в этом хаосе? А как же требования? График? Глупое начальство и бестолковые типы из отдела маркетинга? Разве по крайней мере часть вины не лежит на них?

Нет. Начальство и маркетологи обращаются к нам за информацией, на основании которой они выдвигают свои обещания и обязательства; но даже если они к нам не обращаются, мы не должны бояться говорить им то, что мы думаем. Пользователи обращаются к нам, чтобы мы высказали свое мнение относительно того, насколько уместны требования в системе. Руководители проектов обращаются к нам за помощью в составлении графика. Мы принимаем самое деятельное

участие в планировании проекта и несем значительную долю ответственности за любые провалы; особенно если эти провалы обусловлены плохим кодом!

«Но постойте! — скажете вы. — Если я не сделаю то, что говорит мой начальник, меня уволят». Скорее всего, нет. Обычно начальники хотят знать правду, даже если по их поведению этого не скажешь. Начальники хотят видеть хороший код, даже если они помешаны на рабочем графике. Они могут страстно защищать график и требования; но это их работа. А ваша работа — так же страстно защищать код.

Чтобы стало понятнее, представьте, что вы — врач, а ваш пациент требует прекратить дурацкое мытье рук при подготовке к операции, потому что это занимает слишком много времени<sup>1</sup>! Естественно, пациент — это ваш начальник; и все же врач должен наотрез отказаться подчиниться его требованиям. Почему? Потому что врач знает об опасности заражения больше, чем пациент. Было бы слишком непрофессионально (а то и преступно) подчиниться воле пациента.

Таким образом, программист, который подчиняется воле начальника, не понимающего опасность некачественного кода, проявляет непрофессионализм.

## Основной парадокс

Программисты сталкиваются с основным парадоксом базовых ценностей. Каждый разработчик, имеющий сколько-нибудь значительный опыт работы, знает, что предыдущий беспорядок замедляет его работу. Но при этом все разработчики под давлением творят беспорядок в своем коде для соблюдения графика. Короче, у них нет времени, чтобы работать быстро!

Настоящие профессионалы знают, что вторая половина этого парадокса неверна. Невозможно выдержать график, устроив беспорядок. На самом деле этот беспорядок сразу же замедлит вашу работу, и график будет сорван. Единственный способ выдержать график — и единственный способ работать быстро — заключается в том, чтобы постоянно поддерживать чистоту в коде.

## Искусство чистого кода?

Допустим, вы согласились с тем, что беспорядок в коде замедляет вашу работу. Допустим, вы согласились, что для быстрой работы необходимо соблюдать чистоту. Тогда вы должны спросить себя: «А как мне написать чистый код?» Бесплезно пытаться написать чистый код, если вы не знаете, что это такое!

К сожалению, написание чистого кода имеет много общего с живописью. Как правило, мы способны отличить хорошую картину от плохой, но это еще не значит,

---

<sup>1</sup> Когда Игнац Земмельвейс в 1847 году впервые порекомендовал врачам мыть руки перед осмотром пациентов, его советы были отвергнуты на том основании, что у врачей слишком много работы и на мытье рук у них нет времени.



что мы умеем рисовать. Таким образом, умение отличать чистый код от грязного еще не означает, что вы умеете писать чистый код!

Чтобы написать чистый код, необходимо сознательно применять множество приемов, руководствуясь приобретенным усердным трудом чувством «чистоты». Ключевую роль здесь играет «чувство кода». Одни с этим чувством рождаются. Другие работают, чтобы развить его. Это чувство не только позволяет отличить хороший код от плохого, но и демонстрирует стратегию применения наших навыков для преобразования плохого кода в чистый код.

Программист без «чувства кода» посмотрит на грязный модуль и распознает беспорядок, но понятия не имеет, что с ним делать. Программист с «чувством кода» смотрит на грязный модуль и видит различные варианты и возможности. «Чувство кода» поможет ему выбрать лучший вариант и спланировать последовательность преобразований, сохраняющих поведение программы и приводящих к нужному результату.

Короче говоря, программист, пишущий чистый код, — это художник, который проводит пустой экран через серию преобразований, пока он не превратится в элегантно запрограммированную систему.

## Что такое «чистый код»?

Вероятно, сколько существует программистов, столько найдется и определений. Поэтому я спросил у некоторых известных, чрезвычайно опытных программистов, что они думают по этому поводу.

### **БЬЁРН СТРАУСТРУП, СОЗДАТЕЛЬ C++ И АВТОР КНИГИ «THE C++ PROGRAMMING LANGUAGE»**

Я люблю, чтобы мой код был элегантным и эффективным. Логика должна быть достаточно прямолинейной, чтобы ошибкам было трудно спрятаться; зависимости — минимальными, чтобы упростить сопровождение; обработка ошибок — полной в соответствии с выработанной стратегией; а производительность — близкой к оптимальной, чтобы не искушать людей загрязнять код беспринципными оптимизациями. Чистый код хорошо решает одну задачу.



Бьёрн использует слово «элегантный». Хорошее слово! Словарь в моем Mac-Book® выдает следующие определения: доставляющий удовольствие своим изяществом и стилем; сочетающий простоту с изобретательностью. Обратите внимание на оборот «доставляющий удовольствие». Очевидно, Бьёрн считает,

что чистый код приятно читать. При чтении чистого кода вы улыбаетесь, как при виде искусно сделанной музыкальной шкатулки или хорошо сконструированной машины.

Бьёрн также упоминает об эффективности — притом дважды. Наверное, никого не удивят эти слова, произнесенные изобретателем C++, но я думаю, что здесь кроется нечто большее, чем простое стремление к скорости. Напрасные траты процессорного времени неэлегантны, они не радуют глаз. Также обратите внимание на слово «искушение», которым Бьёрн описывает последствия неэлегантности. В этом кроется глубокая истина. Плохой код *искушает*, способствуя увеличению беспорядка! Когда другие программисты изменяют плохой код, они обычно делают его еще хуже.

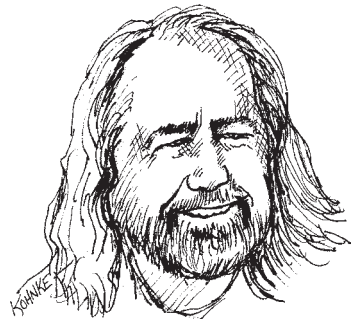
Прагматичные Дэйв Томас (Dave Thomas) и Энди Хант (Andy Hunt) высказали ту же мысль несколько иначе. Они сравнили плохой код с разбитыми окнами<sup>1</sup>. Здание с разбитыми окнами выглядит так, словно никому до него нет дела. Поэтому люди тоже перестают обращать на него внимание. Они равнодушно смотрят, как на доме появляются новые разбитые окна, а со временем начинают сами бить их. Они уродуют фасад дома надписями и устраивают мусорную свалку. Одно разбитое окно стало началом процесса разложения.

Бьёрн также упоминает о необходимости полной обработки ошибок. Это одно из проявлений внимания к мелочам. Упрощенная обработка ошибок — всего лишь одна из областей, в которых программисты пренебрегают мелочами. Утечка — другая область, состояния гонки — третья, непоследовательный выбор имен — четвертая... Суть в том, что чистый код уделяет пристальное внимание мелочам.

В завершение Бьёрн говорит о том, что чистый код хорошо решает одну задачу. Не случайно многие принципы проектирования программного обеспечения сводятся к этому простому наставлению. Писатели один за другим пытаются донести эту мысль. Плохой код пытается сделать слишком много всего, для него характерны неясные намерения и неоднозначность целей. Для чистого кода характерна целенаправленность. Каждая функция, каждый класс, каждый модуль фокусируются на конкретной цели, не отвлекаются от нее и не загрязняются окружающими подробностями.

**ГРЭДИ БУЧ, АВТОР КНИГИ  
«OBJECT ORIENTED ANALYSIS  
AND DESIGN WITH APPLICATIONS»**

Чистый код прост и прямолинеен. Чистый код читается, как хорошо написанная проза. Чистый код никогда не затемняет намерения проектировщика; он полон четких абстракций и простых линий передачи управления.



<sup>1</sup> <http://www.pragmaticprogrammer.com/booksellers/2004-12.html>.

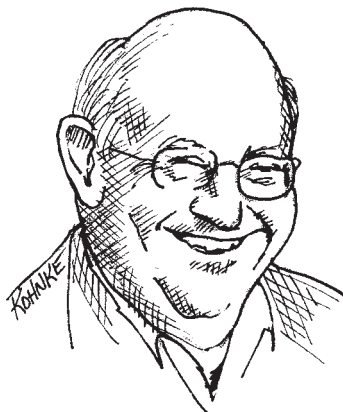
Грэди частично говорит о том же, о чем говорил Бьёрн, но с точки зрения *удобочитаемости*. Мне особенно нравится его замечание о том, что чистый код должен читаться, как хорошо написанная проза. Вспомните какую-нибудь хорошую книгу, которую вы читали. Вспомните, как слова словно исчезали, заменяясь зрительными образами! Как кино, верно? Лучше! Вы словно видели персонажей, слышали звуки, испытывали душевное волнение и сопереживали героям.

Конечно, чтение чистого кода никогда не сравнится с чтением «Властелина колец». И все же литературная метафора в данном случае вполне уместна. Чистый код, как и хорошая повесть, должен наглядно раскрыть интригу решаемой задачи. Он должен довести эту интригу до высшей точки, чтобы потом читатель воскликнул: «Ага! Ну конечно!», когда все вопросы и противоречия благополучно разрешатся в откровении очевидного решения.

На мой взгляд, использованный Грэди оборот «четкая абстракция» представляет собой очаровательный оксюморон! В конце концов, слово «четкий» почти всегда является синонимом для слова «конкретный». В словаре моего MacBook приведено следующее определение слова «четкий»: *краткий, решительный, фактический, без колебаний или лишних подробностей*. Несмотря на кажущееся смысловое противоречие, эти слова несут мощный информационный посыл. Наш код должен быть фактическим, а не умозрительным. Он должен содержать только то, что необходимо. Читатель должен видеть за кодом нашу решительность.

### «БОЛЬШОЙ» ДЭЙВ ТОМАС, ОСНОВАТЕЛЬ ОТГ, КРЕСТНЫЙ ОТЕЦ СТРАТЕГИИ ECLIPSE

Чистый код может читаться и совершенствоваться другими разработчиками, кроме его исходного автора. Для него написаны модульные и приемочные тесты. В чистом коде используются содержательные имена. Для выполнения одной операции в нем используется один путь (вместо нескольких разных). Чистый код обладает минимальными зависимостями, которые явно определены, и четким, минимальным API. Код должен быть грамотным, потому что в зависимости от языка не вся необходимая информация может быть четко выражена в самом коде.



Большой Дэйв разделяет стремление Грэди к удобочитаемости, но с одной важной особенностью. Дэйв утверждает, что чистота кода упрощает его доработку другими людьми. На первый взгляд это утверждение кажется очевидным, но его важность трудно переоценить. В конце концов, код, который легко читается, и код, который легко изменяется, — не одно и то же.

Дэйв связывает чистоту с тестами! Десять лет назад это вызвало бы множество недоуменных взглядов. Однако методология разработки через тестирование оказала огромное влияние на нашу отрасль и стала одной из самых фундамен-

тальных дисциплин. Дэйв прав. Код без тестов не может быть назван чистым, каким бы элегантным он ни был и как бы хорошо он ни читался.

Дэйв использует слово «минимальный» дважды. Очевидно, он отдает предпочтение компактному коду перед объемистым кодом. В самом деле, это положение постоянно повторяется в литературе по программированию от начала ее существования. Чем меньше, тем лучше.

Дэйв также говорил, что код должен быть *грамотным*. Это ненавязчивая ссылка на концепцию «грамотного программирования» Дональда Кнута [Knuth92]. Итак, код должен быть написан в такой форме, чтобы он хорошо читался людьми.

### МАЙКЛ ФИЗЕРС, АВТОР КНИГИ «WORKING EFFECTIVELY WITH LEGACY CODE»

Я мог бы перечислить все признаки, присущие чистому коду, но существует один важнейший признак, из которого следуют все остальные. Чистый код всегда выглядит так, словно его автор над ним тщательно потрудился. Вы не найдете никаких очевидных возможностей для его улучшения. Все они уже были продуманы автором кода. Попытавшись представить возможные усовершенствования, вы снова придете к тому, с чего все началось: вы рассматриваете код, тщательно продуманный и написанный настоящим мастером, безразличным к своему ремеслу.



Всего одно слово: тщательность. На самом деле оно составляет тему этой книги. Возможно, ее название стоило снабдить подзаголовком: «Как тщательно работать над кодом».

Майкл попал в самую точку. Чистый код — это код, над которым тщательно поработали. Кто-то не пожалел своего времени, чтобы сделать его простым и стройным. Кто-то уделил должное внимание всем мелочам и относился к коду с душой.

### РОН ДЖЕФФРИС, АВТОР КНИГ «EXTREME PROGRAMMING INSTALLED» И «EXTREME PROGRAMMING ADVENTURES IN C#»

Карьера Рона началась с программирования на языке Fortran. С тех пор он писал код практически на всех языках и на всех компьютерах. К его словам стоит прислушаться.



---

За последние коды я постоянно руководствуюсь «правилами простого кода», сформулированными Беком. В порядке важности, простой код:

- проходит все тесты;
- не содержит дубликатов;
- выражает все концепции проектирования, заложенные в систему;
- содержит минимальное количество сущностей: классов, методов, функций и т. д.

Из всех правил я уделяю основное внимание дублированию. Если что-то делается в программе снова и снова, это свидетельствует о том, что какая-то мысленная концепция не нашла представления в коде. Я пытаюсь понять, что это такое, а затем пытаюсь выразить идею более четко.

Выразительность для меня прежде всего означает содержательность имен. Обычно я провожу переименования по несколько раз, пока не остановлюсь на окончательном варианте. В современных средах программирования — таких, как Eclipse — переименование выполняется легко, поэтому изменения меня не беспокоят. Впрочем, выразительность не ограничивается одними лишь именами. Я также смотрю, не выполняет ли объект или метод более одной операции. Если это объект, то его, вероятно, стоит разбить на два и более объекта. Если это метод, я всегда применяю к нему прием «извлечения метода»; в итоге у меня остается основной метод, который более четко объясняет, что он делает, и несколько подметодов, объясняющих, как он это делает.

Отсутствие дублирования и выразительности являются важнейшими составляющими чистого кода в моем понимании. Даже если при улучшении грязного кода вы будете руководствоваться только этими двумя целями, разница в качестве кода может быть огромной. Однако существует еще одна цель, о которой я также постоянно помню, хотя объяснить ее будет несколько сложнее.

После многолетней работы мне кажется, что все программы состоят из очень похожих элементов. Для примера возьмем операцию «найти элемент в коллекции». Независимо от того, работаем ли мы с базой данных, содержащий информацию о работниках, или хеш-таблицей с парами «ключ-значение», или массивом с однотипными объектами, на практике часто возникает задача извлечь конкретный элемент из этой коллекции. В подобных ситуациях я часто инкапсулирую конкретную реализацию в более абстрактном методе или классе. Это открывает пару интересных возможностей.

Я могу определить для нужной функциональности какую-нибудь простую реализацию (например, хеш-таблицу), но поскольку все ссылки прикрыты моей маленькой абстракцией, реализацию можно в любой момент изменить. Я могу быстро двигаться вперед, сохраняя возможность внести изменения позднее.

Кроме того, абстракция часто привлекает мое внимание к тому, что же «действительно» происходит в программе, и удерживает меня от реализации поведения коллекций там, где в действительности достаточно более простых способов получения нужной информации. Сокращение дублирования, высокая выразительность и раннее построение простых абстракций. Все это составляет чистый код в моем понимании.

---

В нескольких коротких абзацах Рон представил сводку содержимого этой книги. Устранение дублирования, выполнение одной операции, выразительность, просты абстракции. Все на месте.

**УОРД КАННИНГЕМ, СОЗДАТЕЛЬ WIKI, СОЗДАТЕЛЬ FIT, ОДИН ИЗ СОЗДАТЕЛЕЙ ЭКСТРЕМАЛЬНОГО ПРОГРАММИРОВАНИЯ. ВДОХНОВИТЕЛЬ НАПИСАНИЯ КНИГИ «DESIGN PATTERNS». ДУХОВНЫЙ ЛИДЕР SMALLTALK И ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПОДХОДА. КРЕСТНЫЙ ОТЕЦ ВСЕХ, КТО ТЩАТЕЛЬНО ОТНОСИТСЯ К НАПИСАНИЮ КОДА.**

Вы работаете с чистым кодом, если каждая функция делает примерно то, что вы ожидали. Код можно назвать красивым, если у вас также создается впечатление, что язык был создан специально для этой задачи.



Подобные заявления — отличительная способность Уорда. Вы читаете их, киваете головой и переходите к следующей теме. Это звучит настолько разумно, настолько очевидно, что не выглядит чем-то глубоким и мудрым. Вроде бы все само собой разумеется. Но давайте присмотримся повнимательнее.

«...примерно то, что вы ожидали». Когда вы в последний раз видели модуль, который делал примерно то, что вы ожидали? Почему попадающиеся нам модули выглядят сложными, запутанными, приводят в замешательство? Разве они не нарушают это правило? Как часто вы безуспешно пытались понять логику всей системы и проследить ее в том модуле, который вы сейчас читаете? Когда в последний раз при чтении кода вы кивали головой так, как при очевидном заявлении Уорда?

Уорд считает, что чтение чистого кода вас совершенно не удивит. В самом деле, оно даже не потребует от вас особых усилий. Вы читаете код, и он делает примерно то, что вы ожидали. Чистый код очевиден, прост и привлекателен. Каждый модуль создает условия для следующего. Каждый модуль показывает, как будет написан следующий модуль. Чистые программы написаны настолько хорошо, что вы этого даже не замечаете. Благодаря автору код выглядит до смешного простым, как и все действительно выдающиеся творения.

А как насчет представления Уорда о красоте? Все мы жаловались на языки, не предназначенные для решения наших задач. Однако утверждение Уорда возлагает ответственность на нас. Он говорит, что при чтении красивого кода язык кажется созданным для решения конкретной задачи! Следовательно, мы сами должны позаботиться о том, чтобы язык казался простым! Языковые фанатики, задумайтесь! Не язык делает программы простыми. Программа выглядит простой благодаря работе программиста!

## Школы мысли

А как насчет меня (Дядюшка Боб)? Что я думаю по поводу чистого кода? Эта книга расскажет вам во всех подробностях, что я и мои соратники думаем о чистом коде. Вы узнаете, как, по нашему мнению, должно выглядеть чистое имя переменной, чистая функция, чистый класс и т. д. Мы излагаем свои мнения в виде беспрекословных истин и не извиняемся за свою категоричность. Для нас, на данном моменте наших карьер, они *являются* беспрекословными истинами. Они составляют нашу *школу мысли* в области чистого кода.

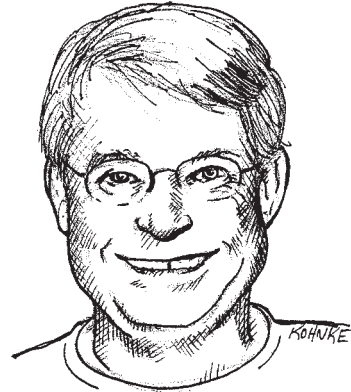
Мастера боевых искусств не достигли единого мнения по поводу того, какой из видов единоборств является лучшим, а какие приемы — самыми эффективными. Часто ведущие мастера создают собственную школу и набирают учеников. Так появилась школа дзю-дзюцу Грейси, основанная семьей Грейси в Бразилии. Так появилась школа дзю-дзюцу Хаккорю, основанная Окуямой Рюхо в Токио. Так появилась школа Джит Кун-до, основанная Брюсом Ли в Соединенных Штатах.

Ученики этих разных школ погружаются в учение основателя школы. Они посвящают себя изучению того, чему учит конкретный мастер, часто отказываясь от учений других мастеров. Позднее, когда уровень их мастерства возрастет, они могут стать учениками другого мастера, чтобы расширить свои познания и проверить их на практике. Некоторые переходят к совершенствованию своих навыков, открывают новые приемы и открывают собственные школы.

Ни одна из этих разных школ не обладает *абсолютной истиной*. Тем не менее в рамках конкретной школы мы действуем так, будто ее учение и арсенал приемов верны. Именно так и следует тренироваться в школе Хаккорю или Джит Кун-до. Но правильность принципов в пределах одной школы не делает ошибочными учения других школ.

Считайте, что эта книга является описанием Школы учителей Чистого кода. В ней представлены те методы и приемы, которыми мы сами пользуемся в своем искусстве. Мы утверждаем, что если вы последуете нашему учению, то это принесет вам такую же пользу, как и нам, и вы научитесь писать чистый и профессиональный код. Но не стоит думать, что наше учение «истинно» в каком-то абсолютном смысле. Существуют другие школы и мастера, которые имеют ничуть не меньше оснований претендовать на профессионализм. Не упускайте возможности учиться у них.

В самом деле, многие рекомендации в этой книге противоречивы. Вероятно, вы согласитесь не со всеми из них. Возможно, против некоторых вы будете яростно протестовать. Это нормально. Мы не претендуем на абсолютную истину. С дру-



гой стороны, приведенные в книге рекомендации являются плодами долгих, непростых размышлений. Мы пришли к ним после десятилетий практической работы, непрерывных проб и ошибок. Независимо от того, согласитесь вы с нами или нет, нашу точку зрения стоит по крайней мере узнать и уважать.

## Мы — авторы

Поле @author комментария javadoc говорит о том, кто мы такие. Мы — авторы. А как известно, у каждого автора имеются свои читатели. Автор несет ответственность за то, чтобы хорошо изложить свои мысли читателям. Когда вы в следующий раз напишете строку кода, вспомните, что вы — автор, и пишете для читателей, которые будут оценивать плоды вашей работы.

Кто-то спросит: так ли уж часто читается наш код? Разве большая часть времени не уходит на его написание?

Вам когда-нибудь доводилось воспроизводить запись сеанса редактирования? В 80-х и 90-х годах существовали редакторы, записывавшие все нажатия клавиш (например, Emacs). Вы могли проработать целый час, а потом воспроизвести весь сеанс, словно ускоренное кино. Когда я это делал, результаты оказывались просто потрясающими. Большинство операций относилось к прокрутке и переходу к другим модулям!

*Боб открывает модуль.*

*Он находит функцию, которую необходимо изменить.*

*Задумывается о последствиях.*

*Ой, теперь он переходит в начало модуля, чтобы проверить инициализацию переменной.*

*Снова возвращается вниз и начинает вводить код.*

*Стирает то, что только что ввел.*

*Вводит заново.*

*Еще раз стирает!*

*Вводит половину чего-то другого, но стирает и это!*

*Прокручивает модуль к другой функции, которая вызывает изменяемую функцию, чтобы посмотреть, как она вызывается.*

*Возвращается обратно и восстанавливает только что стертый код.*

*Задумывается.*

*Снова стирает!*

*Открывает другое окно и просматривает код subclasses. Переопределяется ли в нем эта функция?*

...

В общем, вы поняли. На самом деле соотношение времени чтения и написания кода превышает 10:1. Мы постоянно читаем свой старый код, поскольку это необходимо для написания нового кода.



Из-за столь высокого соотношения наш код должен легко читаться, даже если это затрудняет его написание. Конечно, написать код, не прочитав его, невозможно, так что упрощение чтения в действительности упрощает и написание кода. Уйти от этой логики невозможно. Невозможно написать код без предварительного чтения окружающего кода. Код, который вы собираетесь написать сегодня, будет легко или тяжело читаться в зависимости от того, насколько легко или тяжело читается окружающий код. Если вы хотите быстро справиться со своей задачей, если вы хотите, чтобы ваш код было легко писать — позаботьтесь о том, чтобы он легко читался.

## Правило бойскаута

Хорошо написать код недостаточно. Необходимо поддерживать чистоту кода с течением времени. Все мы видели, как код загнивает и деградирует с течением времени. Значит, мы должны активно поработать над тем, чтобы этого не произошло.

У бойскаутов существует простое правило, которое применимо и к нашей профессии:

*Оставь место стоянки чище, чем оно было до твоего прихода<sup>1</sup>.*

Если мы все будем оставлять свой код чище, чем он был до нашего прихода, то код попросту не будет загнивать. Чистка не обязана быть глобальной. Присвойте более понятное имя переменной, разбейте слишком большую функцию, уберите одно незначительное повторение, почистите сложную цепочку `if`.

Представляете себе работу над проектом, код которого *улучшается* с течением времени? Но может ли профессионал позволить себе нечто иное? Разве постоянное совершенствование не является неотъемлемой частью профессионализма?

## Предыстория и принципы

Эта книга во многих отношениях является «предысторией» для книги, написанной мной в 2002 году: «Agile Software Development: Principles, Patterns, and Practices» (сокращенно PPP). Книга PPP посвящена принципам объектно-ориентированного проектирования и практическим приемам, используемым профессиональными разработчиками. Если вы еще не читали PPP, скажу, что там развивается тема, начатая в этой книге. Прочитавшие убедятся, что многие идеи перекликаются с идеями, изложенными в этой книге на уровне кода.

---

<sup>1</sup> Из прощального послания Роберта Стивенсона Смита Баден-Пауэлла скаутам: «Постарайтесь оставить этот мир чуть лучшим, чем он был до вашего прихода...»