



---

# ОГЛАВЛЕНИЕ

---

<b>Предисловие</b> .....	<b>15</b>
<b>Вступление</b> .....	<b>18</b>
О термине «мастерство» .....	18
Единственный правильный путь .....	19
Введение в книгу .....	19
Для себя .....	19
Для общества .....	20
Структура книги .....	22
Примечание для руководителей .....	22
<b>Благодарности</b> .....	<b>23</b>
<b>Об авторе</b> .....	<b>24</b>
<b>От издательства</b> .....	<b>26</b>
<b>Глава 1. Мастерство</b> .....	<b>27</b>

## ЧАСТЬ I ПРИНЯТЫЕ ПРАКТИКИ

Экстремальное программирование .....	39
Жизненный цикл .....	40

Разработка через тестирование .....	41
Рефакторинг .....	42
Простота проектирования .....	43
Совместное программирование .....	44
Пользовательское тестирование .....	44
<b>Глава 2. Разработка через тестирование .....</b>	<b>45</b>
Общие сведения .....	46
Программное обеспечение .....	48
Три закона TDD .....	49
Четвертый закон .....	59
Основы .....	61
Простые примеры .....	61
Стек .....	62
Простые множители .....	76
Игра в боулинг .....	85
Резюме .....	101
<b>Глава 3. Дополнительные возможности TDD .....</b>	<b>102</b>
Сортировка 1 .....	103
Сортировка 2 .....	107
Мертвая точка .....	115
Настрой, действуй, проверь .....	122
Введение в BDD .....	123
Конечные автоматы .....	124
И снова про BDD .....	126
Тестовые двойники .....	126
Пустышка .....	129
Заглушка .....	133
Шпион .....	135

---

Подставной объект .....	137
Имитация .....	140
Принцип неопределенности TDD .....	142
Лондон против Чикаго .....	154
Выбор между гибкостью и определенностью .....	155
Лондонская школа .....	155
Классическая школа, или Школа Чикаго .....	156
Синтез .....	157
Архитектура .....	158
Резюме .....	160
<b>Глава 4. Разработка тестов .....</b>	<b>161</b>
Тестирование баз данных .....	162
Тестирование графических интерфейсов .....	164
Графический ввод .....	167
Шаблоны тестирования .....	168
Связанный с тестом подкласс .....	168
Самошунтирование .....	170
Скромный объект .....	170
Проектирование тестов .....	174
Проблема хрупких тестов .....	174
Однозначное соответствие .....	175
Разрыв соответствия .....	176
Магазин видеопроката .....	178
Конкретика против общности .....	194
Определение очередности преобразований .....	196
$\{\}$ $\rightarrow$ ничто .....	198
Ничто $\rightarrow$ константа .....	198
Константа $\rightarrow$ переменная .....	199
Отсутствие условий $\rightarrow$ выбор .....	200

---

Значение → список .....	200
Оператор → рекурсия .....	201
Выбор → итерация .....	201
Значение → измененное значение .....	202
Пример: числа Фибоначчи .....	202
Определение очередности преобразований .....	206
Резюме .....	207
<b>Глава 5. Рефакторинг .....</b>	<b>208</b>
Что такое рефакторинг .....	209
Основной инструментарий .....	211
Переименование .....	211
Выделение методов .....	212
Выделение переменной .....	214
Выделение поля .....	215
Кубик Рубика .....	226
Практики .....	227
Тесты .....	227
Быстрые тесты .....	227
Устранение взаимно однозначных соответствий .....	228
Непрерывный рефакторинг .....	228
Безжалостный рефакторинг .....	229
Поддержка проходимости тестов! .....	229
Оставляйте себе выход .....	230
Резюме .....	230
<b>Глава 6. Простой дизайн .....</b>	<b>232</b>
YAGNI .....	236
Тестовое покрытие .....	238
Степень покрытия .....	239

---

Асимптотическая цель .....	241
Дизайн? .....	241
Но это еще не все .....	242
<b>Максимальное раскрытие предназначения .....</b>	<b>242</b>
Базовая абстракция .....	244
Тесты: вторая половина проблемы .....	245
<b>Минимизация дублирования .....</b>	<b>246</b>
Непреднамеренное дублирование .....	247
<b>Минимизация размера .....</b>	<b>248</b>
Простой дизайн .....	249
<b>Глава 7. Совместное программирование .....</b>	<b>250</b>
<b>Глава 8. Приемочное тестирование .....</b>	<b>254</b>
Порядок действий .....	257
Непрерывная сборка .....	258
<b>ЧАСТЬ II</b>	
<b>СТАНДАРТЫ</b>	
Ваш новый технический директор .....	260
<b>Глава 9. Производительность .....</b>	<b>261</b>
Мы никогда не будем делать дрянь .....	262
Легкая адаптивность .....	264
Постоянная готовность .....	265
Стабильная производительность .....	267
<b>Глава 10. Качество .....</b>	<b>269</b>
Постоянное улучшение .....	270
Бесстрашная компетентность .....	271

Исключительное качество .....	272
Мы не будем заваливать работой отдел контроля качества .....	273
Болезнь отдела тестирования .....	274
Отдел контроля качества ничего не найдет .....	274
Автоматизация тестирования .....	275
Автоматизированное тестирование и пользовательские интерфейсы .....	276
Тестирование пользовательского интерфейса .....	278
<b>Глава 11. Смелость .....</b>	<b>279</b>
Прикрываем друг другу спину .....	280
Честная оценка .....	281
Умение говорить «нет» .....	283
Непрерывное интенсивное обучение .....	284
Наставничество .....	285
<b>ЧАСТЬ III</b>	
<b>ЭТИКА</b>	
Самый первый программист .....	288
75 лет .....	289
Ботаники и Спасители .....	294
Образцы для подражания и злодеи .....	297
Мы правим миром .....	298
Катастрофы .....	299
Клятва .....	301
<b>Глава 12. Вред .....</b>	<b>303</b>
Прежде всего — не навреди .....	304
Не навреди обществу .....	305
Нарушение функционирования .....	307

---

Нарушение структуры .....	310
Программное обеспечение .....	311
Тесты .....	313
Лучшая работа .....	314
Делаем это правильно .....	315
Что такое хорошая структура .....	316
Матрица Эйзенхауэра .....	318
Программисты как заинтересованные лица .....	320
Делать все возможное. ....	322
Повторяемое доказательство .....	324
Дейкстра .....	324
Доказательство правильности. ....	325
Структурное программирование .....	328
Функциональная декомпозиция .....	330
Разработка через тестирование. ....	331
<b>Глава 13. Верность своим принципам .....</b>	<b>334</b>
Малые циклы .....	335
История управления исходным кодом. ....	335
Git .....	341
Короткие циклы .....	342
Непрерывная интеграция .....	343
Ветки и переключатели .....	344
Непрерывное развертывание. ....	346
Непрерывная сборка. ....	348
Неустанное улучшение .....	349
Покрытие тестами .....	349
Мутационное тестирование .....	350
Семантическая стабильность. ....	351



Очистка .....	352
Творения .....	352
Поддержание высокой продуктивности .....	353
Вязкость .....	354
Управление отвлекающими факторами .....	357
Управление временем .....	360
<b>Глава 14. Работа в команде .....</b>	<b>362</b>
Работать как одна команда .....	363
Открытый/виртуальный офис .....	363
Честная и справедливая оценка .....	365
Ложь .....	366
Честность, безошибочность, точность .....	367
История 1. Проект «Векторизация» .....	368
История 2. pCCU .....	370
Уроки .....	371
Безошибочность .....	372
Точность .....	374
Обобщение .....	375
Честность .....	376
Уважение .....	379
Никогда не переставай учиться .....	379

---

## Но это еще не все

Тесты не только побуждают создавать несвязанные и надежные проекты, но и позволяют улучшать эти проекты с течением времени. Как я уже не раз упоминал, надежный набор тестов значительно снижает страх перед изменениями. Если у вас есть такой набор, который к тому же работает быстро, ничто не мешает вам улучшать дизайн кода каждый раз, когда обнаруживается лучший подход. Или если выясняется, что новые требования невозможно реализовать при текущем дизайне, тесты позволят безбоязненно внести необходимые изменения.

И именно поэтому первое и самое важное правило простого дизайна касается именно покрытия тестами. Без набора тестов, покрывающих всю систему, остальные три правила бесполезны, поскольку их лучше всего применять *постфактум*. Они имеют отношение к рефакторингу, который практически невозможно осуществить без хорошего исчерпывающего набора тестов.

## МАКСИМАЛЬНОЕ РАСКРЫТИЕ ПРЕДНАЗНАЧЕНИЯ

На заре эры программирования код не давал представления о том, зачем он нужен. Более того, само слово «код» предполагало нечто неявное и скрытое. Пример кода, который писался в те дни, показан на рис. 6.2.

Обратите внимание на множество комментариев. Без них было не обойтись, поскольку сам код вообще ничего не говорил о предназначении программы.

Впрочем, 1970-е остались далеко позади. Языки, которые мы сейчас используем, *чрезвычайно* выразительны. При должной практике можно научиться создавать код, который читается как «хорошо написанная проза», «не затемняет намерения проектировщика»<sup>1</sup>.

---

<sup>1</sup> *Мартин Р.* Чистый код. — С. 30.

```

-----
/RROUTINE TO TYPE A MESSAGE                                PALS-V10D NO DATE    PAGE 1

                /ROUTINE TO TYPE A MESSAGE
0200             *200
7600             MONADR=7600
00200 7300 START, CLA CLL      /CLEAR ACCUMULATOR AND LINK
00201 6046             TLS      /CLEAR TERMINAL FLAG
00202 1216             TAD BUFADR /SET UP POINTER
00203 3217             DCA PNTR  /FOR GETTING CHARACTERS
00204 6041 NEXT,      TSF       /SKIP IF TERMINAL FLAG SET
00205 5204             JMP .-1   /NO: CHECK AGAIN
00206 1617             TAD I PNTR /GET A CHARACTER
00207 6046             TLS      /PRINT A CHARACTER
00210 2217             ISZ PNTR  /DONE YET?
00211 7300             CLA CLL   /CLEAR ACCUMULATOR AND LINK
00212 1617             TAD I PNTR /GET ANOTHER CHARACTER
00213 7640             SZA CLA   /JUMP ON ZERO AND CLEAR
00214 5204             JMP NEXT  /GET READY TO PRINT ANOTHER
00215 5631             JMP I MON /RETURN TO MONITOR
00216 0220             BUFADR, BUFF /BUFFER ADDRESS
00217 0220             PNTR,  BUFF /POINTER
00220 0215             BUFF,   215;212;"H";"E";"L";"L";"O";"!"0
00221 0212
00222 0310
00223 0305
00224 0314
00225 0314
00226 0317
00227 0241
00230 0000
00231 7600 MON,      MONADR      /MONITOR ENTRY POINT

```

Рис. 6.2. Пример ранней программы

В качестве примера рассмотрим фрагмент кода на языке Java из упражнения с магазином видеопроката, которое вы выполняли в главе 4:

```

public class RentalCalculator {
    private List<Rental> rentals = new ArrayList<>();

    public void addRental(String title, int days) {
        rentals.add(new Rental(title, days));
    }

    public int getRentalFee() {
        int fee = 0;
        for (Rental rental : rentals)

```

```
        fee += rental.getFee();
    return fee;
}

public int getRenterPoints() {
    int points = 0;
    for (Rental rental : rentals)
        points += rental.getPoints();
    return points;
}
}
```

Человек, который не работает над этим проектом, очевидно, не сможет понять всего, что происходит внутри данного кода. Однако даже при самом беглом взгляде несложно определить основной замысел проектировщика. Назначение переменных, функций и типов понятно по их именам. Легко увидеть структуру алгоритма. Это очень красноречивый код. И очень простой.

## Базовая абстракция

Чтобы вы не думали, что выразительность сводится исключительно к выбору описательных имен для функций и переменных, должен сказать, что существует еще одна проблема. Это разделение уровней и представление лежащей в основе абстракции.

Программную систему можно назвать выразительной, если каждая строка кода, каждая функция и каждый модуль находятся в однозначно определенном разделе, который четко отображает уровень кода и его место в общей абстракции.

Последнее предложение может оказаться несколько сложным для понимания, поэтому попробую пояснить его на практике.

Представим приложение со сложным набором требований. Здесь я люблю использовать в качестве примера систему начисления заработной платы.

- Сотрудники с почасовой оплатой получают деньги каждую пятницу на основании представленных таблиц учета рабочего времени.

При работе более 40 часов в неделю каждый дополнительный час оплачивается по полуторной ставке.

- Сотрудникам, работающим за проценты, зарплата выплачивается в первую и третью пятницу каждого месяца. Она состоит из базового оклада плюс комиссионные, рассчитываемые по предоставленным квитанциям о продажах.
- Сотрудникам с фиксированным окладом зарплата начисляется в последний день месяца.

Без труда можно представить набор функций со сложным оператором `switch` или цепочкой операторов `if/else`, которые описывают вышеперечисленные условия. Но такой набор функций, скорее всего, скроет лежащую в основе кода абстракцию. Что это за абстракция?

```
public List<Paycheck> run(Database db) {
    Calendar now = SystemTime.getCurrentDate();
    List<Paycheck> paychecks = new ArrayList<>();
    for (Employee e : db.getAllEmployees()) {
        if (e.isPayDay(now))
            paychecks.add(e.calculatePay());
    }
    return paychecks;
}
```

Обратите внимание: здесь не упоминаются многочисленные детали, которыми наполнены требования. Основная цель создания приложения заключается в том, что все сотрудники должны получать деньги в свои дни зарплаты. Фундаментальная основа создания простого и выразительного дизайна состоит в отделении высокоуровневой политики от низкоуровневой реализации деталей.

## Тесты: вторая половина проблемы

Вспомним первоначальную формулировку первого правила Бека.

*Система (код и тесты) должна сообщать все, что вы хотите сообщить.*

Он сформулировал это именно так по определенной причине, и в некотором смысле очень жаль, что формулировка была изменена.

Насколько бы говорящим вы ни сделали производственный код, он не сможет передать контекст своего использования. Это задача тестов.

Каждый написанный тест, особенно в ситуации, когда все тесты изолированы и не связаны, демонстрирует, как именно предполагается использовать производственный код. Хорошо написанные тесты — это примеры использования тех частей кода, работу которых они проверяют.

Таким образом, код *в совокупности* с тестами показывает функцию каждого элемента системы и способ его применения.

Как это связано с дизайном? Целиком и полностью. При создании каждого проекта наша основная цель состоит в том, чтобы упростить другим программистам процессы понимания, улучшения и обновления наших систем. И нет лучшего способа достичь этой цели, чем заставить систему сообщать свое предназначение и предполагаемые варианты использования.

## МИНИМИЗАЦИЯ ДУБЛИРОВАНИЯ

На заре существования программного обеспечения в принципе не существовало редакторов исходного кода. Код писали карандашом на специальных бланках. Соответственно, лучшим инструментом редактирования был ластик. Возможность скопировать и вставить фрагмент кода попросту отсутствовала.

Поэтому код не дублировался. Куда проще было создать экземпляр фрагмента кода и поместить его в подпрограмму.

Затем появились редакторы исходного кода, а вместе с ними и возможность копирования/вставки. Внезапно стало намного проще скопировать фрагмент кода, вставить его в другое место и редактировать, пока он не заработает.

В результате с годами накапливались системы, в коде которых присутствовало дублирование.

Оно обычно порождает проблемы. Необходимость отредактировать одновременно два или более одинаковых фрагмента возникает достаточно часто. Искать такие фрагменты сложно. Правильно отредактировать их еще сложнее, поскольку они существуют в разных контекстах. Фактически дублирование порождает хрупкость.

В общем, похожие фрагменты кода лучше сводить к одному экземпляру, абстрагируя этот код в новую функцию и предоставляя ей соответствующие аргументы, сообщающие о различиях в контексте.

Такая стратегия работает не всегда. Бывает, например, так, что дублирование происходит в коде, проходящем через сложную структуру данных. И разные части системы будут использовать один и тот же цикл и код обхода только для того, чтобы поработать с этой структурой.

Но так как со временем любая структура данных меняется, программистам придется искать все дубликаты кода обхода, чтобы соответствующим образом обновить их. Чем больше дублируется код, тем выше риск хрупкости.

Дублирование кода обхода можно устранить, инкапсулировав его в одном месте и воспользовавшись для передачи в него необходимых операций лямбда-выражением, объектом `Command`, паттерном «Стратегия» (Strategy) или даже паттерном «Шаблонный метод» (Template Method)<sup>1</sup>.

## Непреднамеренное дублирование

Убирать дублирующийся код нужно далеко не во всех случаях. Иногда фрагменты кода могут быть очень похожими, даже идентичными,

---

<sup>1</sup> Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Паттерны объектно-ориентированного проектирования. — СПб.: Питер.

но изменяться по разным причинам<sup>1</sup>. Я называю такую ситуацию *непреднамеренным дублированием* (accidental duplication). Здесь не требуется ничего делать. По мере изменения требований дубликаты будут развиваться по отдельности, и непреднамеренное дублирование исчезнет.

Как видите, работать с дублирующимся кодом не так-то просто. Чтобы определить, какой код продублирован сознательно, а где дублирование было непреднамеренным, а затем инкапсулировать и изолировать дубликаты, требуется взвешенный и обстоятельный подход.

Легкость определения того, какие фрагменты были продублированы сознательно, а какие непреднамеренно, сильно зависит от возможности по виду кода понять его предназначение. Непреднамеренное дублирование имеет разное предназначение, в то время как предназначение сознательно созданных дубликатов пересекается.

Инкапсуляция и изоляция последних с помощью абстрагирования, лямбда-выражений и шаблонов проектирования требует значительного рефакторинга. А рефакторинг невозможен без надежного набора тестов.

Именно поэтому устранение дублирующегося кода стоит в списке правил простого дизайна на третьем месте. После возможностей тестирования и понимания предназначения кода.

## МИНИМИЗАЦИЯ РАЗМЕРА

Простой дизайн состоит из простых элементов. А простые элементы имеют небольшой размер. Последнее правило создания простого дизайна гласит: после того, как вы прошли все тесты, максимально проявили предназначение кода и минимизировали дублирование, приходит пора поработать над уменьшением размеров кода. Это касается каждой написанной вами функции. И естественно, это нужно делать без нарушения трех других принципов.

---

<sup>1</sup> См. принцип единственной ответственности в книге: *Мартин Р.* Быстрая разработка программ: Принципы, примеры, практика.



Как этого добиться? В основном путем выделения как можно большего количества функций. В предыдущей главе мы говорили о том, что выделять функции следует до тех пор, пока остается такая возможность.

В результате мы получим прекрасный набор маленьких функций с красивыми длинными именами, позволяющими понять их предназначение.

## Простой дизайн

Много лет назад мы с Кентом Беком обсуждали принципы дизайна, и он сказал слова, которые я запомнил навсегда: если как можно точнее следовать четырем изложенным им принципам, то все остальные принципы дизайна будут соблюдены автоматически.

Не знаю, правда ли это. Не знаю, обязательно ли идеально покрытая тестами, выразительная, не имеющая дубликатов кода и имеющая минимальный размер программа будет соответствовать принципу открытости-закрытости или принципу единственной ответственности. Но я совершенно уверен в том, что знание принципов хорошего дизайна и хорошей архитектуры (например, принципов SOLID) значительно облегчает создание четко разделенных и простых проектов.

В этой книге речь не об этих принципах. О них я уже много раз писал<sup>1</sup>, как и другие авторы. И очень рекомендую вам прочитать эти книги и изучать эти принципы для дальнейшего совершенствования вашего мастерства.

---

<sup>1</sup> См. мои книги «Чистый код»; «Чистая архитектура»; «Быстрая разработка программ: Принципы, примеры, практика».

---