ОГЛАВЛЕНИЕ

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.

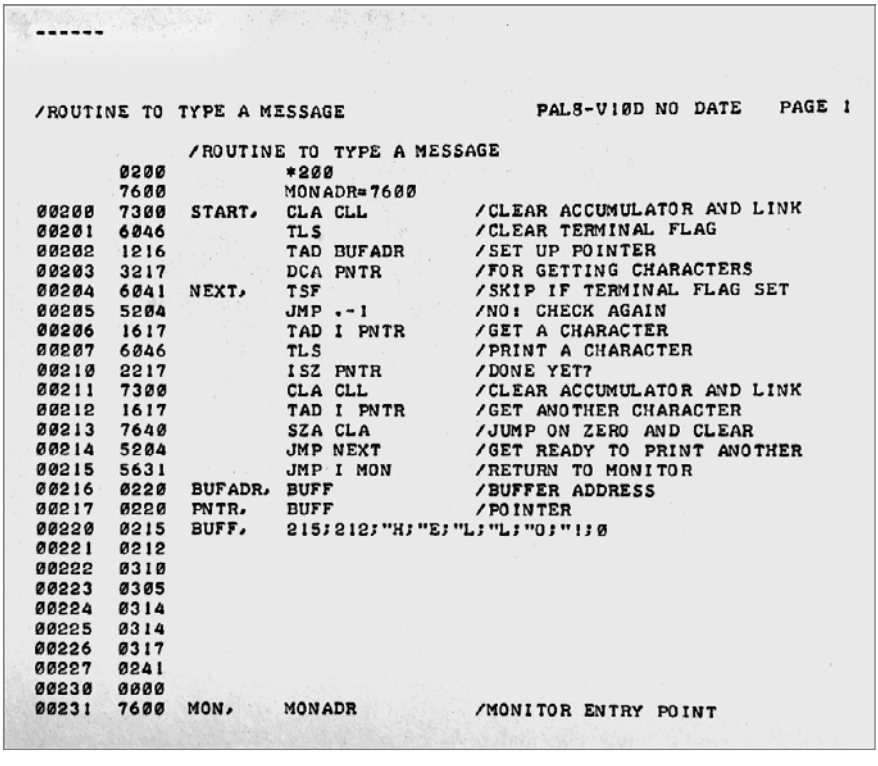
МАКСИМАЛЬНОЕ РАСКРЫТИЕ ПРЕДНАЗНАЧЕНИЯ

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»[1].

[1] *Мартин Р.* Чистый код. — С. 30.



```
------


/ROUTINE TO TYPE A MESSAGE                PAL8-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 табелей 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)[1].

Непреднамеренное дублирование

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

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

но изменяться по разным причинам[1]. 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.

---

[1] 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[1], как и другие авторы. И очень рекомендую вам прочитать эти книги и изучать эти принципы для дальнейшего совершенствования вашего мастерства.

---

[1] См. мои книги «Чистый код»; «Чистая архитектура»; «Быстрая разработка программ: Принципы, примеры, практика».