Практика 2: Объектно-ориентированное проектирование

Юрий Литвинов
y.litvinov@spbu.ru

13.09.2022

Абстрактные типы данных

- currentFont.size = 16 — плохо
- currentFont.size = PointsToPixels(12) — чуть лучше
- currentFont.sizeInPixels = PointsToPixels(12) — ещё чуть лучше
- currentFont.setSizeInPoints(sizeInPoints)
  currentFont.setSizeInPixels(sizeInPixels) — совсем хорошо

Пример плохой абстракции

```
public class Program {
    public void initializeCommandStack() { ... }
    public void pushCommand(Command command) { ... }
    public Command popCommand() { ... }
    public void shutdownCommandStack() { ... }
    public void initializeReportFormatting() { ... }
    public void formatReport(Report report) { ... }
    public void printReport(Report report) { ... }
    public void initializeGlobalData() { ... }
    public void shutdownGlobalData() { ... }
}
```

Пример хорошей абстракции

```
public class Employee {
    public Employee(
            FullName name,
            String address,
            String workPhone,
            String homePhone,
            TaxId taxIdNumber,
            JobClassification jobClass
    ) { ... }

    public FullName getName() { ... }
    public String getAddress() { ... }
    public String getWorkPhone() { ... }
    public String getHomePhone() { ... }
    public TaxId getTaxIdNumber() { ... }
    public JobClassification getJobClassification() { ... }
}
```

Ещё один пример абстракции

```
public class Point {
    public double x;
    public double y;
}
```

vs

```
public interface Point {
    double getX();
    double getY();
    void setCartesian(double x, double y);
    double getR();
    double getTheta();
    void setPolar(double r, double theta);
}
```

Структуры против объектов

- Объекты скрывают свои данные за абстракциями и предоставляют функции для работы с ними
- Структуры раскрывают данные и не имеют осмысленных функций

- Процедурный код позволяет легко добавлять новые функции без изменения существующих структур данных
- Объектно-ориентированный код, напротив, упрощает добавление новых классов без изменения существующих функций

Пример — списки в F#. Тип list имеет 7 методов, тип Collections.List — больше 60

Уровень абстракции (плохо)

```
public class EmployeeRoster implements MyList<Employee> {
    public void addEmployee(Employee employee) { ... }
    public void removeEmployee(Employee employee) { ... }
    public Employee nextItemInList() { ... }
    public Employee firstItem() { ... }
    public Employee lastItem() { ... }
}
```

Уровень абстракции (хорошо)

```
public class EmployeeRoster {
    public void addEmployee(Employee employee) { ... }
    public void removeEmployee(Employee employee) { ... }
    public Employee nextEmployee() { ... }
    public Employee firstEmployee() { ... }
    public Employee lastEmployee() { ... }
}
```

Общие рекомендации

- Учитывайте противоположные методы (add/remove, on/off, ...)
- Разделяйте команды и запросы, избегайте побочных эффектов
- Не возвращайте null
- Соблюдайте принцип единственности ответственности
  - Может потребоваться разделить класс на несколько разных классов просто потому, что методы по смыслу слабо связаны
- По возможности делайте некорректные состояния невыразимыми в системе типов
- Пользуйтесь семантикой языка
  - Комментарии в духе “не пользуйтесь объектом, не вызвав init()” можно заменить конструктором
- При рефакторинге надо следить, чтобы интерфейсы не деградировали

Инкапсуляция

- Принцип минимизации доступности методов
- Паблик-полей не бывает:

```
class Point {
    public float x;
    public float y;
    public float z;
}
```

vs

```
class Point {
    private float x;
    private float y;
    private float z;
    public float getX() { ... }
    public float getY() { ... }
    public float getZ() { ... }
    public void setX(float x) { ... }
    public void setY(float y) { ... }
    public void setZ(float z) { ... }
}
```

Ещё рекомендации

- Класс не должен ничего знать о своих клиентах
- Лёгкость чтения кода важнее, чем удобство его написания
- Опасайтесь семантических нарушений инкапсуляции
  - “Не будем вызывать ConnectToDB(), потому что GetRow() сам его вызовет, если соединение не установлено” — это программирование *сквозь* интерфейс
- Protected- и package- полей тоже не бывает
  - На самом деле, у класса два интерфейса — для внешних объектов и для потомков (может быть отдельно третий, для классов внутри пакета, но это может быть плохо)

Инкапсуляция чужого кода

- Инкапсуляция сторонних API
- “Учебные тесты”
- Паттерн “Адаптер”

Наследование

- Включение лучше
  - Переконфигурируемо во время выполнения
  - Более гибко
  - Иногда более естественно
- Наследование — отношение “является”, закрытого наследования не бывает
  - Наследование — это наследование интерфейса (полиморфизм подтипов, subtyping)
- Хороший тон — явно запрещать наследование (final- или sealed-классы)
  - Хотя спорно, есть и другая школа мысли
- Не вводите новых методов с такими же именами, как у родителя
- Code smells:
  - Базовый класс, у которого только один потомок
  - Пустые переопределения
  - Очень много уровней в иерархии наследования

Пример

```
class Operation {
    private char sign = '+';
    private int left;
    private int right;
    public int eval()
    {
        switch (sign) {
            case '+': return left + right;
        }
        throw new RuntimeException();
    }
}
```

vs

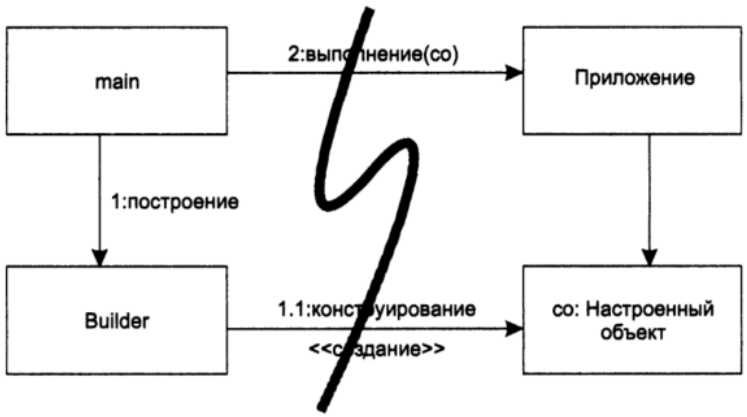
```
abstract class Operation {
    private int left;
    private int right;
    protected int getLeft() { return left; }
    protected int getRight() { return right; }
    abstract public int eval();
}

class Plus extends Operation {
    @Override public int eval() {
        return getLeft() + getRight();
    }
}
```

Конструкторы

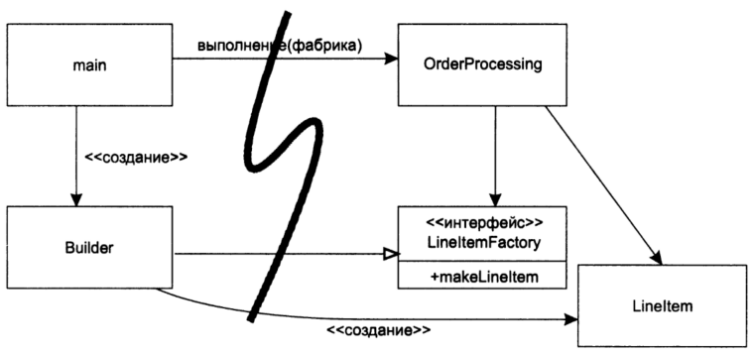
- Инициализируйте все поля, которые надо инициализировать
  - После конструктора должны выполняться все инварианты
- НЕ вызывайте виртуальные методы из конструктора
- private-конструкторы для объектов, которые не должны быть созданы (или одиночек), protected-конструкторы для абстрактных классов
- Одиночек надо использовать с большой осторожностью
- Deep copy предпочтительнее Shallow copy
  - Хотя второе может быть эффективнее

Отделение инициализации от использования

- Паттерн “Builder”:



- Паттерн “Factory”:



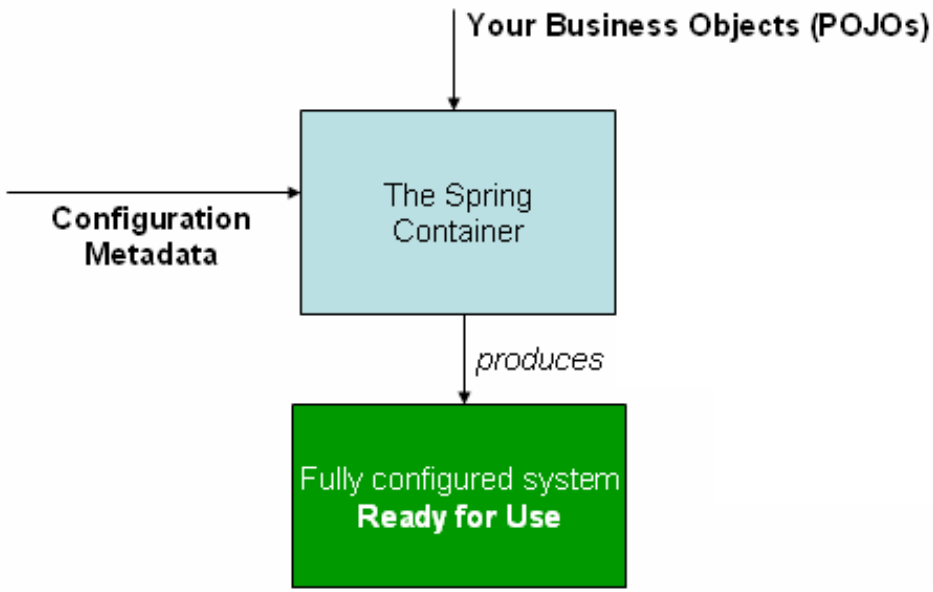
© Р. Мартин, Чистый код

Dependency Injection

- Объект не должен создавать другие объекты, если только он не предназначен специально для этого
- Инициализация системы — ответственность main или специального *контейнера*
- Бюджетный способ — реестр объектов

```
MyService myService =
 (MyService)(jndiContext.lookup("NameOfMyService"));
```

Пример IoC-контейнера: Spring



Пример

Стековый калькулятор

```
package ru.compscicenter.ioc;

public class Calculator {

    private Stack stack;

    public Calculator(Stack stack) {
        this.stack = stack;
    }

    public double calculate(String expression) {
        ...
        return stack.pop();
    }
}
```

Пример

Стек

```
public interface Stack {
    void push(double value);
    double pop();
    boolean isEmpty();
}

public class LinkedStack implements Stack {
    ...
    ...
}
```

Пример

Конфигурация контейнера

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ....>

   <bean id="stack"
         class="ru.compscicenter.ioc.LinkedStack"
         />

   <bean id="calculator"
         class="ru.compscicenter.ioc.Calculator">
      <constructor-arg ref="stack"/>
   </bean>

</beans>
```

Пример

main

```
public class Main {
    public static void main(String[] args) {
        ApplicationContext context =
            new FileSystemXmlApplicationContext("config.xml");

        Calculator calculator =
            context.getBean("calculator", Calculator.class);

        double result = calculator.calculate("1 2 + 3 *");
        System.out.println(result);
    }
}
```

Мутабельность

Мутабельность — способность изменяться

- Запутывает поток данных
- Гонки

Чтобы сделать класс немутабельным, надо:

- Не предоставлять методы, модифицирующие состояние
  - Заменить их на методы, возвращающие копию
- Не разрешать наследоваться от класса
- Сделать все поля константными
- Не давать никому ссылок на поля мутабельных типов

Всё должно быть немутабельно по умолчанию!

Про оптимизацию

Во имя эффективности (без обязательности ее достижения) делается больше вычислительных ошибок, чем по каким-либо иным причинам, включая непроходимую тупость.
– William A. Wulf

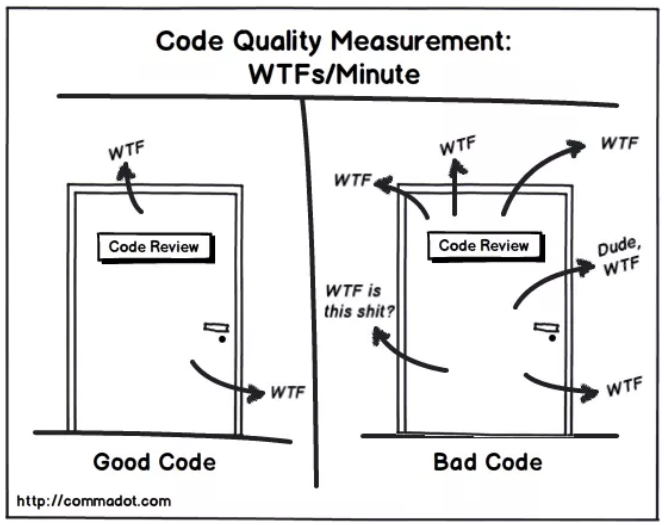
Мы обязаны забывать о мелких усовершенствованиях, скажем, на 97% рабочего времени: опрометчивая оптимизация — корень всех зол.
– Donald E. Knuth

Что касается оптимизации, то мы следуем двум правилам:
Правило 1. Не делайте этого.
Правило 2 (только для экспертов). Пока не делайте этого – т.е. пока у вас нет абсолютно четкого, но неоптимизированного решения.
– M. A. Jackson

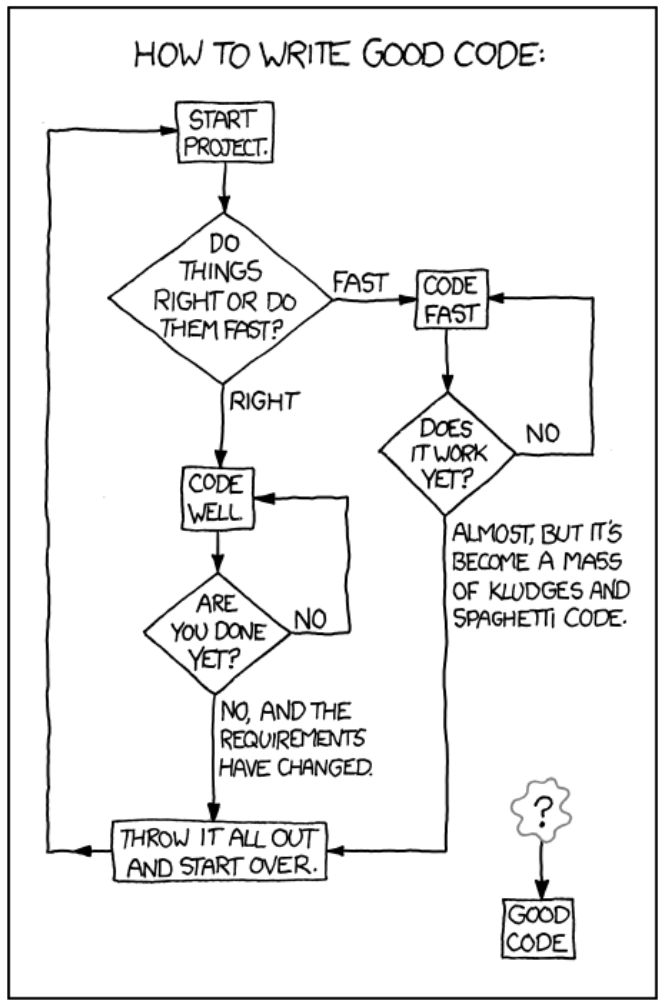
Общие рекомендации

- Fail Fast
  - Не доверяйте параметрам, переданным извне
  - assert-ы – чем больше, тем лучше
- Документируйте все открытые элементы API
  - И заодно всё остальное, для тех, кто будет это сопровождать
  - Предусловия и постусловия, исключения, потокобезопасность
- Статические проверки и статический анализ лучше, чем проверки в рантайме
  - Используйте систему типов по максимуму
- Юнит-тесты
- Continious Integration
- Не надо бояться всё переписать

Заключение

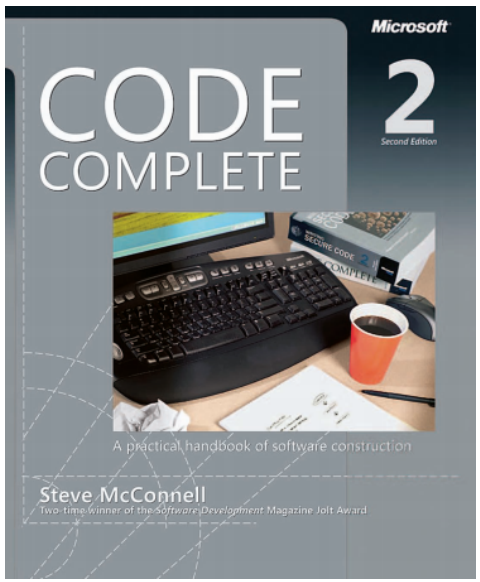


© http://commadot.com, Thom Holwerda

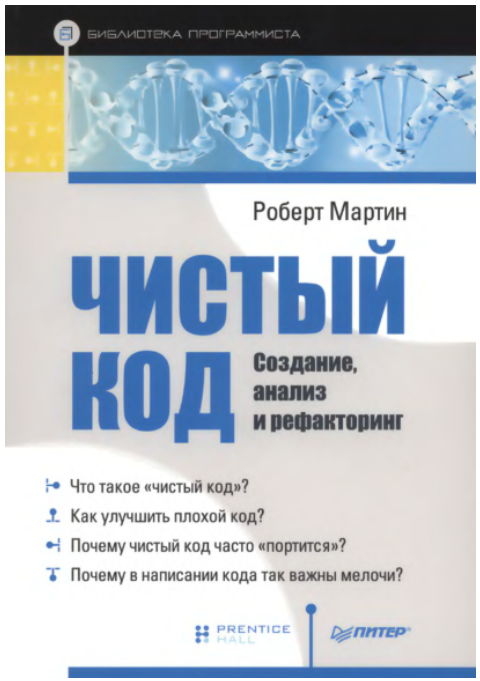


© https://xkcd.com

Книжки



Steve McConnell, Code Complete



Роберт Мартин, Чистый Код