

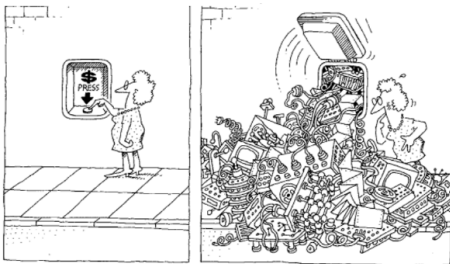
Правила написания хорошего кода в ООП

Юрий Литвинов
yurii.litvinov@gmail.com

03.04.2020г

Сложность

- ▶ **Существенная сложность** (essential complexity) — сложность, присущая решаемой проблеме; ею можно управлять, но от неё нельзя избавиться
- ▶ **Случайная сложность** (accidental complexity) — сложность, привнесённая способом решения проблемы



© G. Booch, "Object-oriented analysis and design"

Модульность

- ▶ Разделение системы на компоненты
- ▶ Потенциально позволяет создавать сколь угодно сложные системы
- ▶ Строгое определение контрактов позволяет разрабатывать независимо
- ▶ Необходим баланс между количеством и размером модулей



Сопряжение и связность

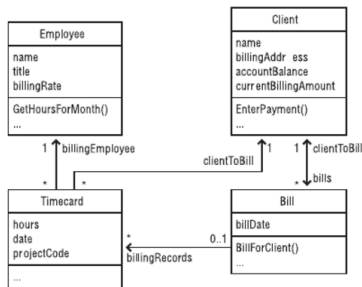
- ▶ **Сопряжение (Coupling)** — мера того, насколько взаимозависимы разные модули в программе
- ▶ **Связность (Cohesion)** — степень, в которой задачи, выполняемые одним модулем, связаны друг с другом
- ▶ Цель: слабое сопряжение и сильная связность

Объекты

- ▶ Objects may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods — **Wikipedia**
- ▶ An object stores its state in fields and exposes its behavior through methods — **Oracle**
- ▶ Each object looks quite a bit like a little computer — it has a state, and it has operations that you can ask it to perform — **Thinking in Java**
- ▶ An object is some memory that holds a value of some type — **The C++ Programming Language**
- ▶ An object is the equivalent of the quanta from which the universe is constructed — **Object Thinking**

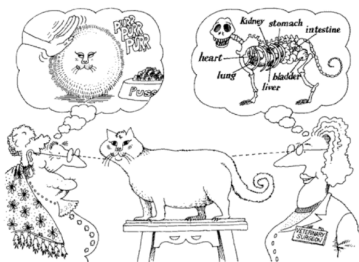
Определение объектов реального мира

- ▶ Определение объектов и их атрибутов
- ▶ Определение действий, которые могут быть выполнены над каждым объектом
- ▶ Определение связей между объектами
- ▶ Определение интерфейса каждого объекта



Согласованные абстракции

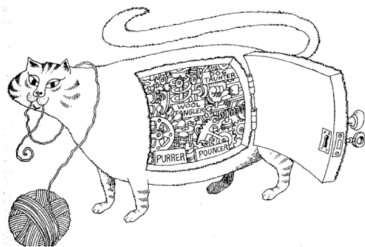
- ▶ Выделение существенных характеристик объекта и игнорирование несущественных
- ▶ Определение его концептуальных границы с точки зрения наблюдателя
 - ▶ Определение интерфейсов
- ▶ Управление сложностью через фиксацию внешнего поведения
- ▶ Необходимы разные уровни абстракции



© G. Booch, "Object-oriented analysis and design"

Инкапсуляция деталей реализации

- ▶ Отделение друг от друга внутреннего устройства и внешнего поведения
- ▶ Изолирование контрактов интерфейса от реализации
- ▶ Управление сложностью через сокрытие деталей реализации



© G. Booch, "Object-oriented analysis and design"

Изоляция сложности

- ▶ Сложные алгоритмы могут быть инкапсулированы
- ▶ Сложные структуры данных — тоже
- ▶ И даже сложные подсистемы
- ▶ Надо внимательно следить за интерфейсами



Изоляция возможных изменений

- ▶ Потенциальные изменения могут быть инкапсулированы
- ▶ Источники изменений
 - ▶ Бизнес-правила
 - ▶ Зависимости от оборудования и операционной системы
 - ▶ Ввод-вывод
 - ▶ Нестандартные возможности языка
 - ▶ Сложные аспекты проектирования и конструирования
 - ▶ Третьесторонние компоненты
 - ▶ ...

Наследование и композиция

▶ Наследование

- ▶ Отношение “Является” (is-a)
- ▶ Способ абстрагирования и классификации
- ▶ Средство обеспечения полиморфизма

▶ Композиция

- ▶ Отношение “Имеет” (has-a)
- ▶ Способ создания динамических связей
- ▶ Средство обеспечения делегирования

▶ Более-менее взаимозаменяемы

- ▶ Объект-потомок на самом деле включает в себя объект-предок
- ▶ Композиция обычно предпочтительнее

Принципы SOLID

- ▶ Single responsibility principle
- ▶ Open/closed principle
- ▶ Liskov substitution principle
- ▶ Interface segregation principle
- ▶ Dependency inversion principle

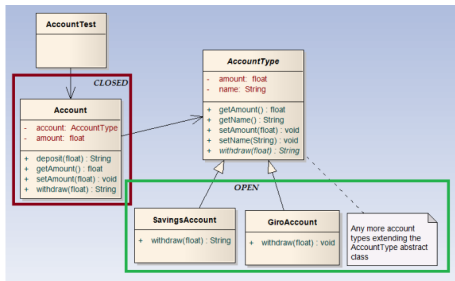
Single responsibility principle

- ▶ Каждый объект должен иметь одну обязанность
- ▶ Эта обязанность должна быть полностью инкапсулирована в класс



Open/closed principle

- ▶ программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения
 - ▶ переиспользование через наследование
 - ▶ неизменные интерфейсы



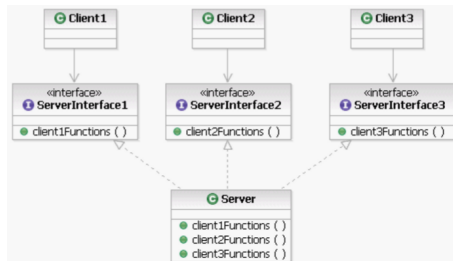
Liskov substitution principle

- ▶ Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом



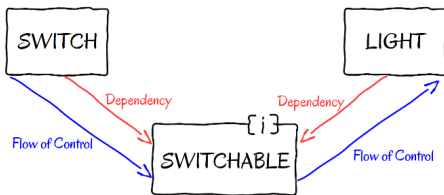
Interface segregation principle

- ▶ Клиенты не должны зависеть от методов, которые они не используют
 - ▶ слишком “толстые” интерфейсы необходимо разделять на более мелкие и специфические



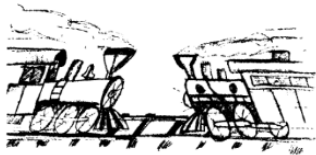
Dependency inversion principle

- ▶ Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций
- ▶ Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций



Закон Деметры

- ▶ “Не разговаривай с незнакомцами!”
- ▶ Объект A не должен иметь возможность получить непосредственный доступ к объекту C, если у объекта A есть доступ к объекту B, и у объекта B есть доступ к объекту C
 - ▶ `book.pages.last.text`
 - ▶ `book.pages().last().text()`
 - ▶ `book.lastPageText()`



© Р. Мартин, “Чистый код”

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() { ... }  
}
```

Ещё один пример абстракции

На Java

```
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);  
}
```

Уровень абстракции (плохо)

```
public class EmployeeRoster: 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

- ▶ Про каждый класс знайте, реализацией какой абстракции он является
- ▶ 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
 - ▶ 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)

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)
- ▶ 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

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.

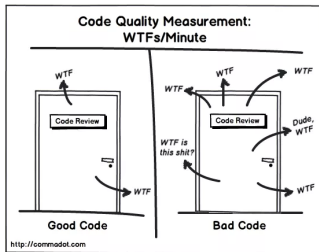
Правило 2 (только для экспертов). Пока не делайте этого – т.е. пока у вас нет абсолютно четкого, но неоптимизированного решения.

– M. A. Jackson

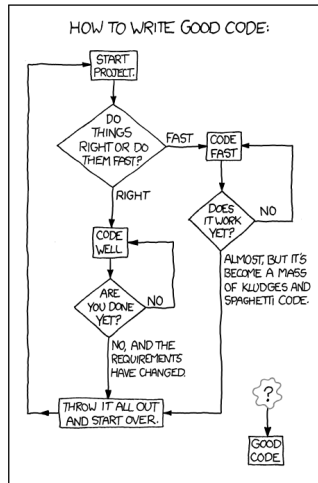
Общие рекомендации

- ▶ 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
- ▶ Continuous Integration
- ▶ Не надо бояться всё переписать

Заключение

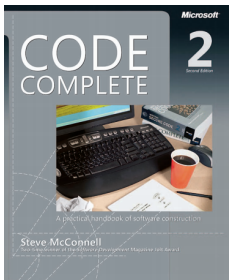


© <http://commadot.com>, Thom Holwerda

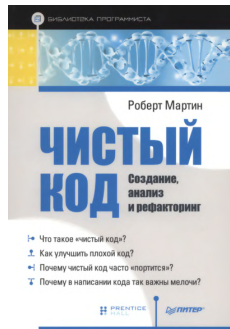


© <https://xkcd.com>

Книжки



Steve McConnell, Code Complete



Роберт Мартин, Чистый Код