

Принципы объектно-ориентированного проектирования

Юрий Литвинов
y.litvinov@spbu.ru

06.07.2023

Объекты

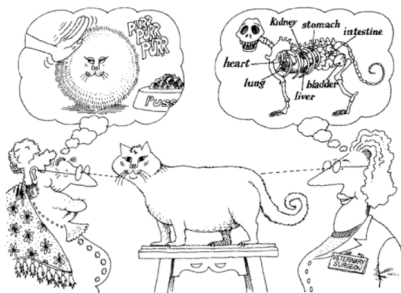
- ▶ Objects may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods — **Wikipedia**
- ▶ An object stores its state in fields and exposes its behavior through methods — **Oracle**
- ▶ Each object looks quite a bit like a little computer — 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**

Объекты

- ▶ Имеют
 - ▶ Состояние
 - ▶ Инвариант
 - ▶ Поведение
 - ▶ Идентичность
- ▶ Взаимодействуют через посылку и приём сообщений
 - ▶ Объект вправе сам решить, как обработать вызов метода (**полиморфизм**)
 - ▶ Могут существовать в разных потоках
- ▶ Как правило, являются экземплярами **классов**

Абстракция

Абстракция выделяет существенные характеристики объекта, отличающие его от остальных объектов, с точки зрения наблюдателя



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

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

Откуда брать классы

- ▶ Объектная модель предметной области
 - ▶ Основной источник “важных” объектов
 - ▶ Существительные — классы, глаголы — методы
- ▶ Изоляция сложности
- ▶ Изоляция изменений
- ▶ Изоляция служебной функциональности
- ▶ Упаковка родственных операций
 - ▶ Статические классы вполне ок

Предметно-ориентированное проектирование

- ▶ Вся архитектура строится вокруг модели предметной области
- ▶ Модель как средство анализа и проектирования
- ▶ Единый язык
- ▶ Чёткое разделение на уровни
 - ▶ Интерфейс пользователя
 - ▶ Операционный
 - ▶ Предметной области
 - ▶ Инфраструктурный
- ▶ Изоляция и минимизация модели предметной области
 - ▶ Выделение смыслового ядра

Принципы SOLID

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

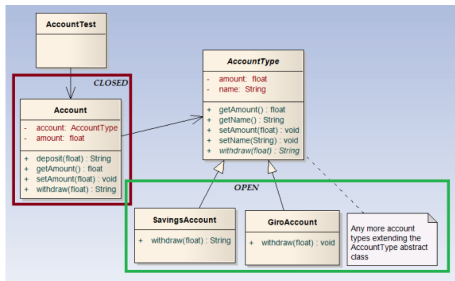
Single responsibility principle

- ▶ 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



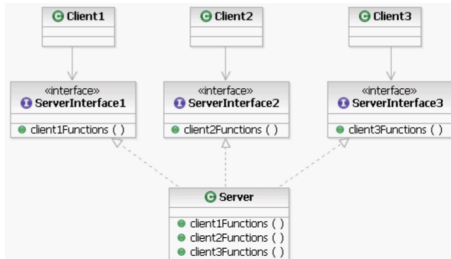
Liskov substitution principle

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



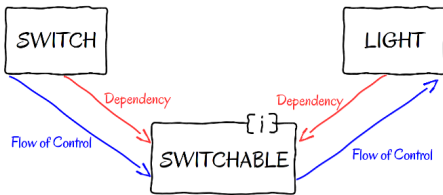
Interface segregation principle

- ▶ 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



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,  
        Address address,  
        Phone workPhone,  
        Phone homePhone,  
        TaxId taxIdNumber,  
    ) { ... }  
  
    public FullName getName() { ... }  
    public Address getAddress() { ... }  
    public Phone getWorkPhone() { ... }  
    public Phone getHomePhone() { ... }  
    public TaxId getTaxIdNumber() { ... }  
}
```


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

- ▶ Про каждый класс знайте, реализацией какой абстракции он является
 - ▶ Может потребоваться разделить класс на несколько разных классов просто потому, что методы по смыслу слабо связаны
- ▶ Учитывайте противоположные методы (add/remove, on/off, ...)
- ▶ По возможности делайте некорректные состояния невыразимыми в системе типов
- ▶ *Объясняющая способность программы важнее её работоспособности!*

Инкапсуляция

```
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);  
}
```

Ещё рекомендации

- ▶ Класс не должен ничего знать о своих клиентах
- ▶ 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)

Наследование

- ▶ Включение лучше
 - ▶ Переконфигурируемо во время выполнения
 - ▶ Более гибко
 - ▶ Иногда более естественно
- ▶ Наследование — отношение “является”
 - ▶ Наследование — это наследование интерфейса (полиморфизм подтипов, subtyping)
 - ▶ Потомок принимает на себя обязательства предка
- ▶ 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

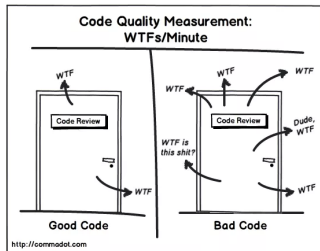
```
abstract class Operation {
    private int left;
    private int right;
    protected int getLeft() { return left; }
    protected int getRight() { return right; }
    abstract public int eval();
}

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

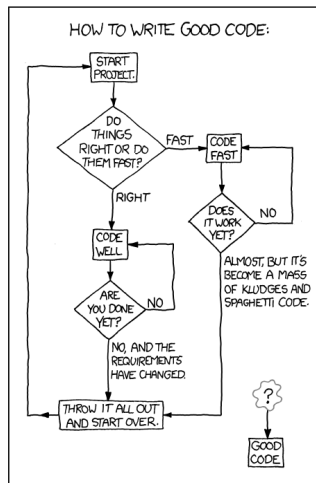
Общие рекомендации

- ▶ 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>