Принципы SOLID

Юрий Литвинов
yurii.litvinov@gmail.com

??.??.2017г

Принципы SOLID

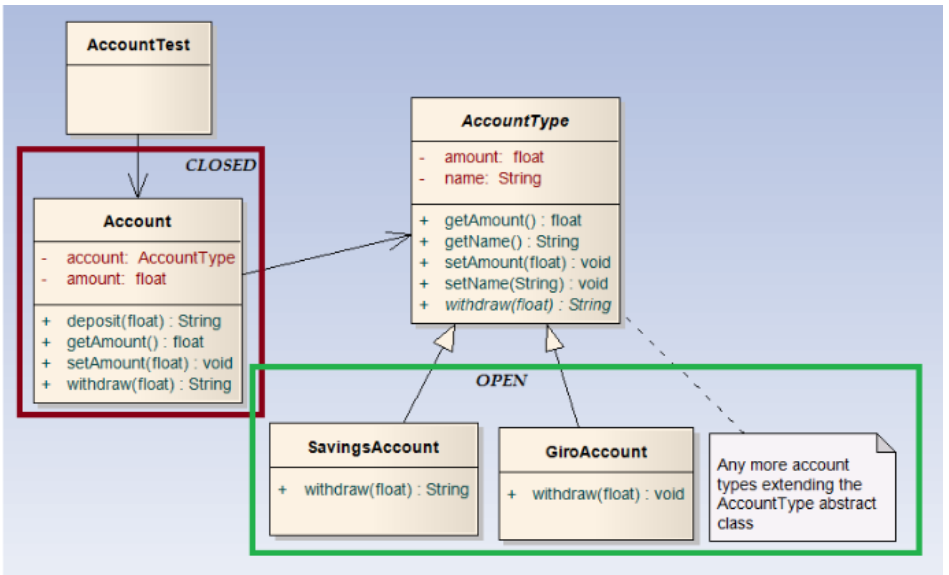
- Single responsibility principle
- Open/closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

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

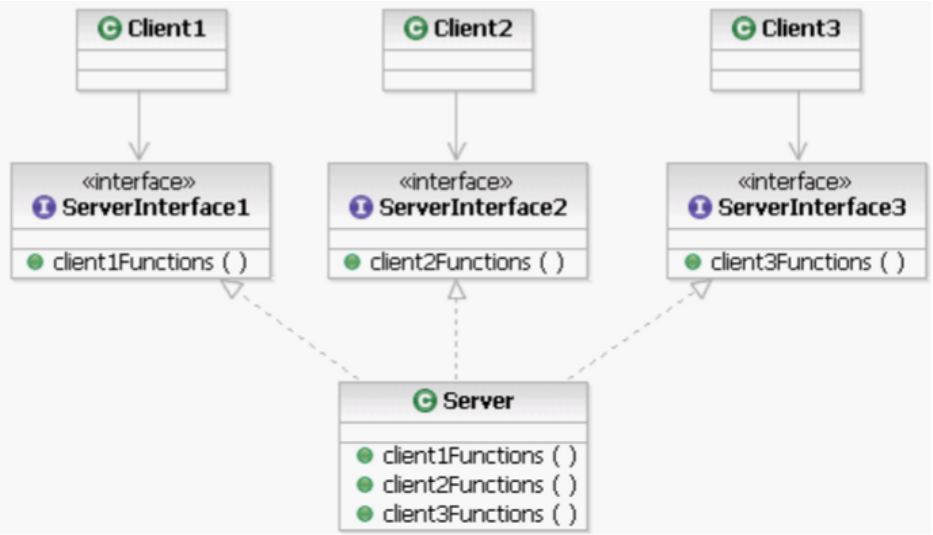


Liskov substitution principle

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

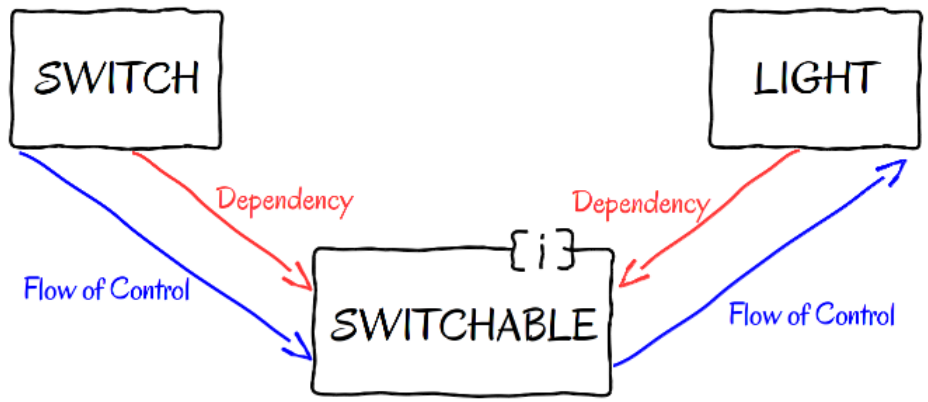
Interface segregation principle

- 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()

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() { ... }
}
```

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
- Учитывайте противоположные методы (add/remove, on/off, ...)
- 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)
- Deep copy предпочтительнее Shallow copy
  - Хотя второе может быть эффективнее

Когда создавать классы

- Объекты предметной области
- Абстрактные объекты
- Изоляция сложности
- Сокрытие деталей реализации
- Изоляция изменчивости
- Упаковка родственных операций
  - Статические классы вполне ок