

# Архитектура ПО

## Лекция 2: Декомпозиция, объектно-ориентированное проектирование

Юрий Литвинов  
yurii.litvinov@gmail.com

28.09.2017г

# Подходы к декомпозиции

- ▶ Восходящее проектирование
  - ▶ Сначала создаём “кирпичики”, потом собираем из них всё более сложные системы
- ▶ Нисходящее проектирование
  - ▶ Постепенная реализация модулей
  - ▶ Строгое задание интерфейсов
  - ▶ Активное использование “заглушек”
  - ▶ Модули
    - ▶ Четкая декомпозиция
    - ▶ Минимизация
    - ▶ Один модуль — одна функциональность
    - ▶ Отсутствие побочных эффектов
    - ▶ Независимость от других модулей
    - ▶ Принцип сокрытия данных

# 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



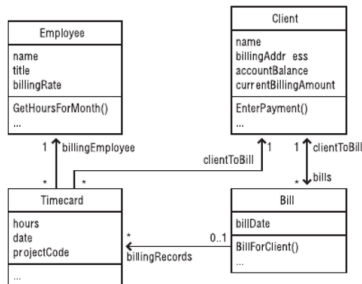
# Объекты

- ▶ 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

Архитектура часто начинается с объектной модели предметной области

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



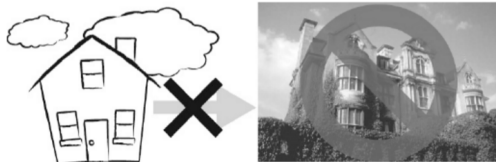
## Согласованные абстракции

- ▶ 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



## Инкапсуляция деталей реализации

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



# Соккрытие “лишней” информации

- ▶ Изоляция “личной” информации
  - ▶ секреты, которые скрывают сложность
  - ▶ секреты, которые скрывают источники изменений
- ▶ Барьеры, препятствующие сокрытию
  - ▶ избыточное распространение информации
  - ▶ поля класса как глобальные данные
  - ▶ снижение производительности





## Изоляция возможных изменений

- ▶ Определите элементы, изменение которых кажется вероятным
- ▶ Отделите элементы, изменение которых кажется вероятным
- ▶ Изолируйте элементы, изменение которых кажется вероятным
- ▶ Источники изменений
  - ▶ Бизнес-правила
  - ▶ Зависимости от оборудования
  - ▶ Ввод-вывод
  - ▶ Нестандартные возможности языка
  - ▶ Сложные аспекты проектирования и конструирования
  - ▶ Переменные статуса
  - ▶ Размеры структур данных
  - ▶ ...

## 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

## Дополнительные принципы

- ▶ Формализуйте контракты классов
- ▶ Проектируйте систему для тестирования
- ▶ Рисуйте диаграммы

# Принципы SOLID

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

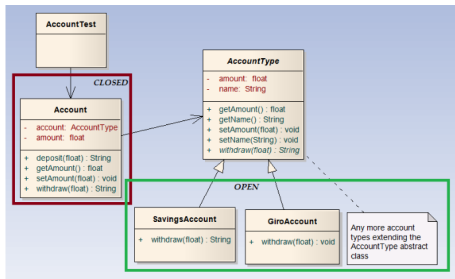
# Single responsibility principle

- ▶ 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



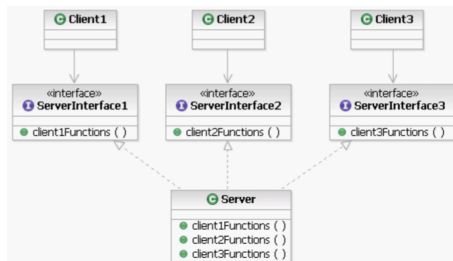
# Liskov substitution principle

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



# Interface segregation principle

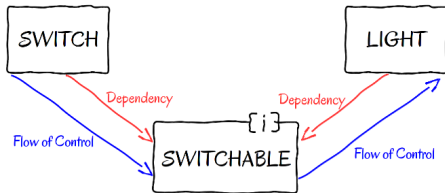
- ▶ Клиенты не должны зависеть от методов, которые они не используют
  - ▶ слишком “толстые” интерфейсы необходимо разделять на более мелкие и специфические





# 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 А не должен иметь возможность получить непосредственный доступ к объекту С, если у объекта А есть доступ к объекту В, и у объекта В есть доступ к объекту С
  - ▶ `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> {  
    ^ ^ Ipublic void addEmployee(Employee employee) { ... }  
    ^ ^ Ipublic void removeEmployee(Employee employee) { ... }  
    ^ ^ Ipublic Employee nextItemInList() { ... }  
    ^ ^ Ipublic Employee firstItem() { ... }  
    ^ ^ Ipublic 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
- ▶ Соккрытие деталей реализации
- ▶ Изоляция изменчивости
- ▶ Упаковка родственных операций
  - ▶ Статические классы вполне ок