

09. OOD

*Лекции по информатике для студентов
первого курса Высшей школы ИТИС
2019 год*

МИХАИЛ АБРАМСКИЙ
старший преподаватель
Высшая школа ИТИС КФУ

Когда учимся программировать в школе

- Привет, класс, вот задача: «...». Включайте компьютер, пишите код.
- Ок.

```
Program Zadacha;  
Var x,y:integer;  
Begin  
Read(x) ;  
y:=x+1 ;  
...
```

Когда учимся в универе

- Хм... почему-то все не работает...

...стирает код...

- Начну писать заново, может поможет...



Популярнейшая иллюзия студента – начинающего программного инженера

Любой код можно переписать за 1-2 дня (одну ночь) без проблем,
без потерь, без дополнительных затрат.



А вот нет!

Промышленный проект –
огромное количество:

- Классов
- Интерфейсов
- Методов
- Строк кода вообще (Source Lines of Code, SLOC)

Время ограничено!
Деньги тем более!



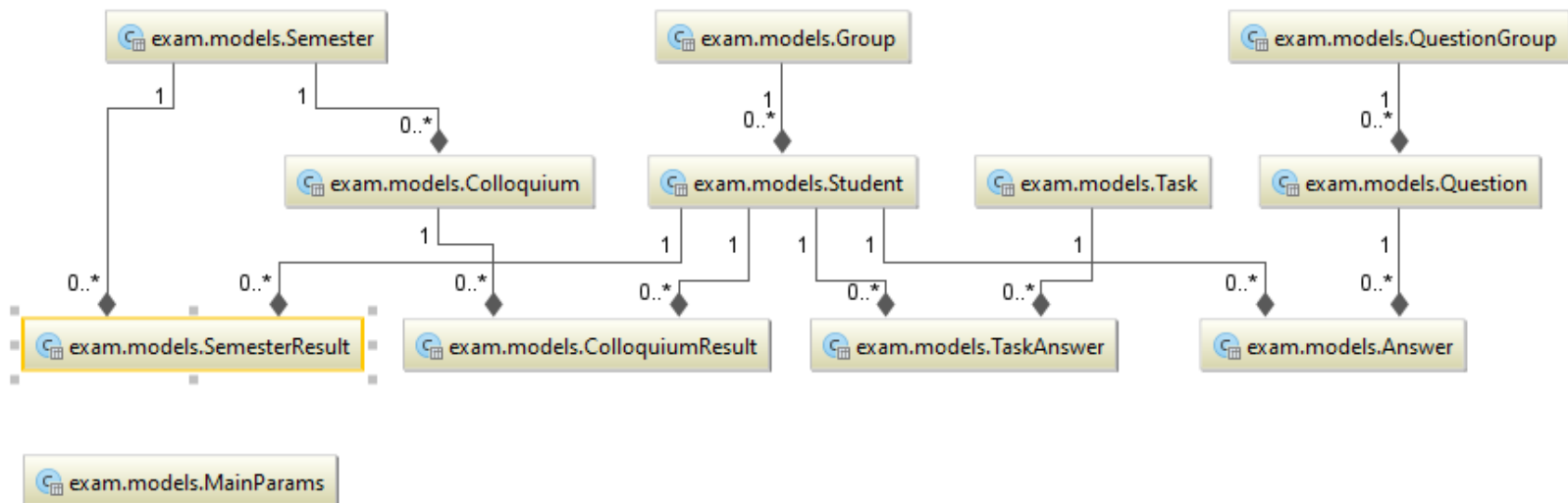
Статистика

Продукт	Строк кода (SLOC)
1991 Ядро Linux 0.1	10 239
2009 Ядро Linux 2.6.32	12 606 910
PostgreSQL	775 000
1C	3 000 000

Личный опыт

Система генерации вопросов на экзамен

- 2 дня работы по 6 часов 1 человека:
- 608 строк Python, 141 строки HTML, 12 классов базы данных



Личный опыт 2

- Один из проектов казанской IT-компании
- 3 года работы труда 6-10 разработчиков в среднем 8-9 часов в день
 - **1 087 706** строк Python, **81 373** строк HTML
 - **2155** .py-файлов
 - Всего лишь в одном файле модели данных **66** классов (а файлов таких около **20**)
 - » и это не единственные файлы, где есть классы
 - Около **12 600** коммитов
 - » А вы ведь понимаете, что коммит – это тоже набор файлов, которые тоже когда-то были написаны.

Ну что, перепишем за пару дней?

Не вопрос умений

- «Мы – крутые разработчики и пишем с первого раза великолепный код!»
- Даже тогда мир не на нашей стороне
 - Вспомните, сколько раз в том семестре по ООП к нам приходил заказчик и добавлял требования?

Этап жизненного цикла разработки

- Необходим этап, когда программную систему надо спроектировать (разработать архитектуру)
 - Оно и понятно: *см. чертежи, планы, эскизы, проекты*
- **После** сбора требований с заказчика
- **Перед** этапом конструирования ПО (кодинга, программирования)

Что такое плохая архитектура

- Изменение старых и добавление новых компонентов ведет к изменениям других старых компонентов
 - Если вводить на сайте фотогалерею – то надо менять регистрацию, учитывать в интерфейсе отображение фото профиля, в сообщениях давать возможность отправлять фото
 - Это не плохой пример. Но если эти изменения имеют лавинный эффект – дело плохо
- В результате изменений появляются ошибки в компонентах, не имеющих отношения к изменениям
 - Прикрутил на сайт фотогалерею – отвалилась регистрация
- Если проект громоздкий - сложно читать и изменять

Главное

- Проектировать надо ТАК, чтобы можно было легко дорабатывать/изменять.
- И вот это **Проблема**.

Личный опыт 2+

Тот же проект, всего лишь 1 таск заказчика, сформулированный одним предложением:

«Сделать модуль такой-то»

- новый функционал

- Срок – 2 недели, 1 разработчик

Личный опыт 2+

В итоге:

- **6 091** строк Python, **180** строк JavaScript
- **59** коммитов
- **44** класса – около 20 новых, остальные – *по шаблону других модулей*
- *Использовано в import-ах около 40 классов других модулей приложения*
- *Пришлось затронуть некоторые файлы в 4 других модулях*
 - обнаружили баги, которые не заметили ранее, поменялись важные файлы, которые должны были учесть новый функционал и т.п.

О чем говорит пример

- Новый функционал **зависит** от старого
 - Было бы глупо, если не зависел (каждый новый модуль был бы отдельным самостоятельным приложением)
- **Зависимость** нескольких типов
 - Как на уровне интерфейса (использования в своих классах)
 - Так и на уровне реализации (делаем по одним и тем же шаблонам – значит, используем те же классы, которые уже были разработаны)
- Старый функционал тоже **МОЖЕТ** **вынужденно зависеть** от нового
 - Вспомните, зачем ввели default-реализацию в интерфейсы Java 8?

Зависимость

- Если ее нет – приложение бесполезно
- Если она есть и она неуправляема – это ПЛОХО
 - Долго разбираемся в чужом коде, новый код ломает старый, появляются баги, поиск которых занимает дни и недели.

В зависимости нужно разбираться, ею нужно управлять.

S.O.L.I.D.

5 основных принципов ООП

Начало 2000-х

Принципы, при которых приложение будет легче/надежнее поддерживать и расширять в течение долгого времени.

S.O.L.I.D.

Single responsibility (SRP)

- Принцип единственной ответственности

Open-closed (OCP)

- Принцип открытости/закрытости

Liskov substitution (LSP)

- Принцип подстановки Барбары Лисков

Interface segregation (ISP)

- Принцип разделения интерфейса

Dependency inversion (DIP)

- Принцип инверсии зависимости

1. Принцип единственной ответственности

*Должна быть единственная причина
появления/изменения класса*

Tom Demarko, 1979

Meilir Page-Jones, 1988

Robert Martin, 2004



Случай в нашем примере

Представьте, что класс Contract одновременно:

- Содержит методы Contract – обновить дату, узнать заказчика и т.п.
- Содержит методы для работы с хранилищем – вернуть договора 2016 года, удалить все договоры с заказчиком «Рога и Копыта».

Это и было бы нарушением принципа SRP, но в нашем примере мы разделяли этот функционал между Contract и Storage.

В промышленной разработке аналогично

- **один** класс отвечает за поведение объектов,
- **второй** – за операции с базой данных, хранящей объекты данного класса.
 - Entity & Repository (!)

Другой пример

```
public class User {  
    // get-методы к атрибутам объекта  
    public String getUsername() {...}  
    public int getAge() {...}  
  
    // методы бизнес-логики приложения  
    public void subscribe(User u) {...}  
    public boolean isFriend(User u) {...}  
    public boolean isAdminOf(Public p) {...}  
  
    // методы работы с базой данных, где хранятся пользователи  
    public User [] getUsersByCity(String city) {...}  
    public User getUserByUsername(String username) {...}  
}  
  
User u = new User("Konstantin");  
System.out.println(u.getUsersByCity("Kazan")); //почему это мой метод???
```

Божественный объект (God Object)

«**Антипаттерн**» (так, как лучше не делать)

Класс, который берет на себя слишком много (хранит много и делает много)

- *Аналог отказа от использования функций в процедурном программировании (все в main)*

Соблюдение SRP должно защищать от появления божественного объекта.

2. Принцип открытости/закрытости

*Сущности должны быть открыты для
расширения, но закрыты для
модификации*

*Bertrand Meyer, 1988,
Robert Martin, 1996*



Суть

- Разработанный (вошедший в определенную версию) класс неприкосновенен для изменений
 - Можно только исправлять ошибки
- Новый функционал – в новые классы, которые могут переопределять существующие (и должны, вообще говоря).

Пример

```
interface UserContacts {  
    String getEmail();  
    String getPhonenumber();  
}  
  
// 2007  
class User implements UserContacts {  
    String getEmail() {...}  
    String getPhonenumber() {...}  
}  
  
//=====
```

```
// 2010  
interface UserContacts {  
    String getEmail();  
    String getPhonenumber();  
    String getVKURL();  
}
```

Скомпилированный в .class класс User (2007 год):

(((((((у меня нет метода getVKURL() и не добавить... ((((((

3. Принцип подстановки Барбары Лисков

*Замена в коде экземпляров классов на
экземпляры их подклассов (наследников)
не должна влиять на правильность
работы программы*

Barbara Liskov, 1987

Robert Martin, 2006



Простая формулировка

Функции, которые используют ссылки на базовые классы, должны иметь возможность использовать объекты подклассов, не зная об этом.

```
class A {  
    // возвращает 3 случайных элемента  
    public int [] f() {  
        Random r = new Random();  
        return new int[]{r.nextInt(), r.nextInt(), r.nextInt()};  
    }  
}  
  
class B extends A{  
    // что?  
    public int [] f() {  
        return new int[]{1, 3, 2};  
    }  
}  
  
class C extends A{  
    // не, не слышал  
    public int [] f() {  
        return new int[]{r.nextInt(), r.nextInt()};  
    }  
}
```



4. Принцип разделения интерфейса

Много интерфейсов, предназначенных для разных пользователей (других классов) лучше одного большого интерфейса, в который свален весь функционал

Robert Martin

Суть

Похоже на God Object, но не об этом.

```
interface ISmartPhone {  
    void call();  
    void sendMessage();  
    void makePhoto();  
}
```

// хочу старый домашний телефон

```
class LandPhone implements ISmartPhone {// (без вариантов)
```

// Пфффф, теперь все методы реализовывать?

// ладно, этот реализую

```
void call() {...}
```

// какой смысл? он не умеет отправлять сообщения?

```
void sendMessage() {...}
```

// а тут как?

```
void makePhoto() {...}
```

```
}
```

Как правильно

```
interface ICall {  
    void call();  
}
```

```
interface ISendMessage {  
    void sendMessage();  
}
```

```
interface IMakePhoto{  
    void makePhoto();  
}
```

```
interface ISmartPhone extends ICall, ISendMessage, IMakePhoto {  
}
```

```
// хочу старый домашний телефон  
class LandPhone implements ICall {  
    void call() {...}  
}
```

5. Принцип инверсии зависимости

*Зависимости классов должны
опираться на абстракции. Зависимости
не должны опираться на конкретную
реализацию.*

Robert Martin

Разберем на примере

- Предметная область разработка приложений
 - Команды специалистов разрабатывают различные приложения (web, мобильные) на различных технологиях.
 - В этом случае наши пример – отрывки кода CRM-системы этой компании.

Сущности

- Приложение (App)
 - его разрабатывают несколько сотрудников.
- Сотрудник (Worker)
 - участвует в работе одного приложения.

ВОТ КОД

```
// абстрактное приложение
```

```
abstract class App { ... }
```

```
// веб-приложение
```

```
class WebApp extends App { ... }
```

```
// сотрудник
```

```
class Worker {
```

```
    private App app;
```

```
    public void startWorkingOnApp() {
```

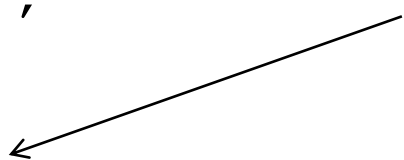
```
        app = new WebApp();
```

```
    }
```

```
    public void work() { ... }
```

```
}
```

Тут что-нибудь
вызывается у **app**.



«Зависимость от зависимости»

```
// абстрактное приложение  
abstract class App { ... }
```

```
// веб-приложение  
class WebApp extends App { ... }
```

```
// сотрудник  
class Worker {  
    private App app;  
  
    public void startWorkingOnApp() {  
        app = new WebApp();  
    }  
  
    public void work() { ... }  
}
```

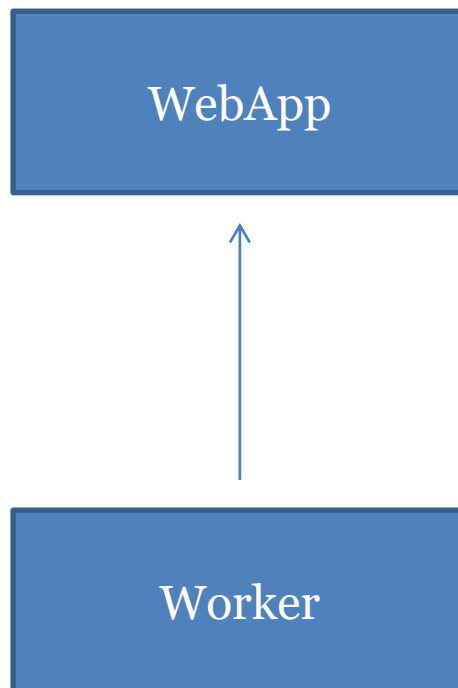
Зависимость

Зависимость
от конкретной
реализации

Зацепление (coupling)

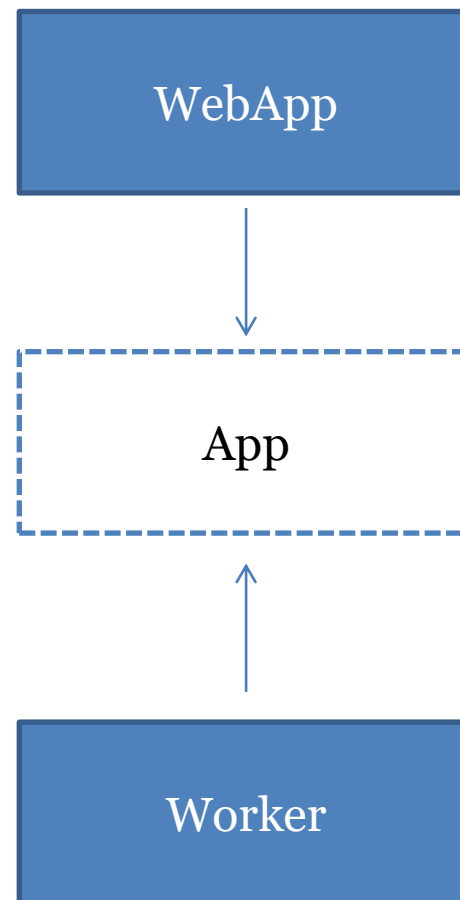
- **Сильное** (тесное – strong, tight, high) – использование в зависимостях конкретных реализаций, конструкторов.
- **Слабое** (weak, loose, low) – зависимость на уровне интерфейсов.
- В нашем примере Worker сильно связан с WebApp (т.к. мы можем вызывать метод WebApp, которого нет в App)
 - Что означает, что Worker уже заточен только на работу с WebApp

Зависимость Worker от WebApp



5-й принцип SOLID

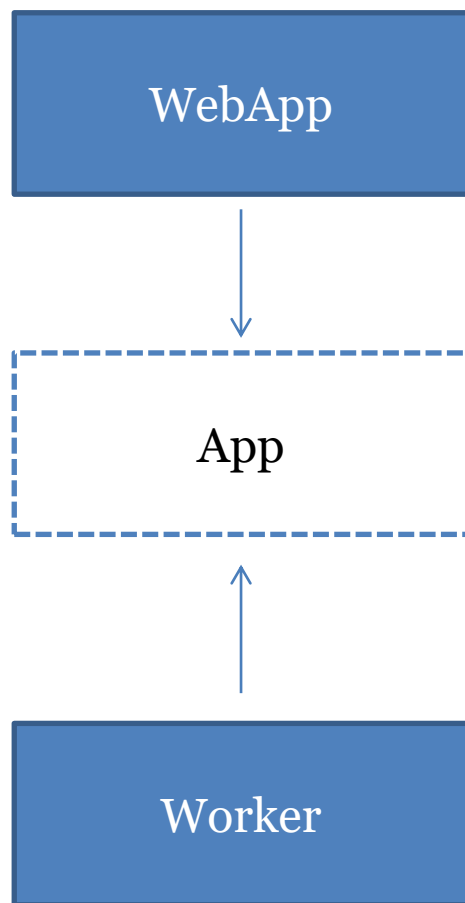
Альтернативная
формулировка:
*«Программируйте на
уровне интерфейсов»*



Worker

```
public class Worker {  
    private App app;  
  
    public Worker(App app) {  
        this.app = app;  
    }  
}
```


«Программируйте на уровне интерфейсов»



Стрелка от WebApp «повернулась»

Это и есть принцип **инверсии зависимости (Dependency Inversion)**.

Зависимость «разорвана» — объект сам не обращается за конкретными реализациями, ему их передают.

ПЕРЕЧИСЛЕНИЯ

Хардкод (Hardcode).

Случай с числами

```
public static void main(String[] args) {  
    Scanner scanner = new Scanner(System.in);  
    int[] a = new int[42];  
    for (int i = 0; i < 42; i++) {  
        a[i] = scanner.nextInt();  
    }  
    for (int i = 0; i < 42 / 2; i++) {  
        a[i] = a[42 - i - 1];  
    }  
}
```

Хардкод. Еще хуже

```
public static void main(String[] args) {  
    Scanner scanner = new Scanner(System.in);  
    int[] a = new int[42];  
    for (int i = 0; i <= 41; i++) {  
        a[i] = scanner.nextInt();  
    }  
    for (int i = 0; i < 21; i++) {  
        a[i] = a[41 - i];  
    }  
}
```

Константы как частный способ решения проблемы

```
public static void main(String[] args) {  
    Scanner scanner = new Scanner(System.in);  
    final int SIZE = 42;  
    int[] a = new int[SIZE];  
    for (int i = 0; i < SIZE; i++) {  
        a[i] = scanner.nextInt();  
    }  
    for (int i = 0; i < SIZE / 2; i++) {  
        a[i] = a[SIZE - i - 1];  
    }  
}
```

Более частый случай – строковый хардкод

```
if (person.getGender().equals("Мужской")) {  
    ...  
}  
  
...  
if (direction.getName().equals("Вверх")) {  
    ...  
}  
  
...  
if (season.getName().equals("Лето")) {  
    ...  
}
```

Одно из решений – строковые константы

```
final String SUMMER = "Summer";  
final String MALE_GENDER = "Male";
```

Проблемы:

- название и значение дублируют друг друга,
- где хранить, чтобы обращаться?
- как быть с Summer и SUMMER?
- ...

Другое решение

```
class Season {  
    final static int WINTER = 0;  
    final static int SPRING = 1;  
    final static int SUMMER = 2;  
    final static int FALL = 3;  
}
```

Проблемы:

- Откуда знать весь диапазон значений и как его перебрать?
- Если `x == 0`, то `Season.WINTER == x`, но действительно ли корректно считать левую переменную `x` хранящей значение «Зима»

Итак

Нужен тип данных:

- Чтобы у переменных этого типа явно было видно значение.
- Чтобы можно было легко перебрать все его значения,
- *Чтобы не хардкодить,*

Он есть! И это...

Перечисления (Enumerations)

Объявление:

```
enum Season {  
    WINTER, SPRING, SUMMER, FALL  
}
```

Использование:

```
Season s = Season.SPRING;
```

Решаем проблемы.

Перебираем с помощью `values()`

- `values()` возвращает массив из всех значений перечисления

```
for (Season season: Season.values()) {  
    System.out.println(season);  
}
```

Решаем проблемы. Сравнить можно только с другими значениями перечисления

```
Season season = Season.SUMMER;
```

```
...
```

```
if (season == Season.WINTER)  
    System.out.println("NEW YEAR!");
```

```
// у каждого есть свой порядковый номер  
// выведет 3
```

```
System.out.print(Season.FALL.ordinal());
```

```
// но вот такое сделать не получится
```

```
if (season == 3) {
```

```
    ...
```

```
}
```

Решаем проблемы. Ввод

- Значение можно восстановить по строке
 - Надо вводить строку с точностью до регистра!

// Прокатит

```
Season season = Season.valueOf("WINTER");
```

...

// Не прокатит

```
Season season = Season.valueOf("Winter");
```

Все гораздо интереснее

- Вы думаете, эти WINTER, SUMMER – просто константы?
- А вот и нет! Это объекты!

Другой enum. Цвет

```
enum Color {  
    RED, GREEN, BLUE, WHITE, BLACK  
}
```

У каждого цвета есть значения RGB.

Наша потребность:

- Чтобы каждый цвет знал свои значения,
- Чтобы каждый цвет мог возвращать строку-представление RGB

Для этого изменим enum.

«В НОВОМ ЦВЕТЕ»

```
enum Color {  
    RED(255, 0, 0), GREEN(0, 255, 0),  
    BLUE(0, 0, 255), WHITE(255, 255, 255),  
    BLACK(0, 0, 0);  
  
    private int r, g, b;  
    Color(int r, int g, int b) {  
        this.r = r;  
        this.g = g;  
        this.b = b;  
    }  
    public String getRGBValues() {  
        return "(" + r + "," + g + "," + b + ")";  
    }  
}
```


«В новом цвете». Использование

```
Color color = Color.BLACK;  
System.out.println(color.getRGBValues());
```