

# Design Patterns

Евгений Борисов

bsevgeny@gmail.com

# Who are you?

## Big Data & Java Technical Leader

Mentoring

Consulting

Lecturing

Writing courses

Writing code

Databases

Big Data

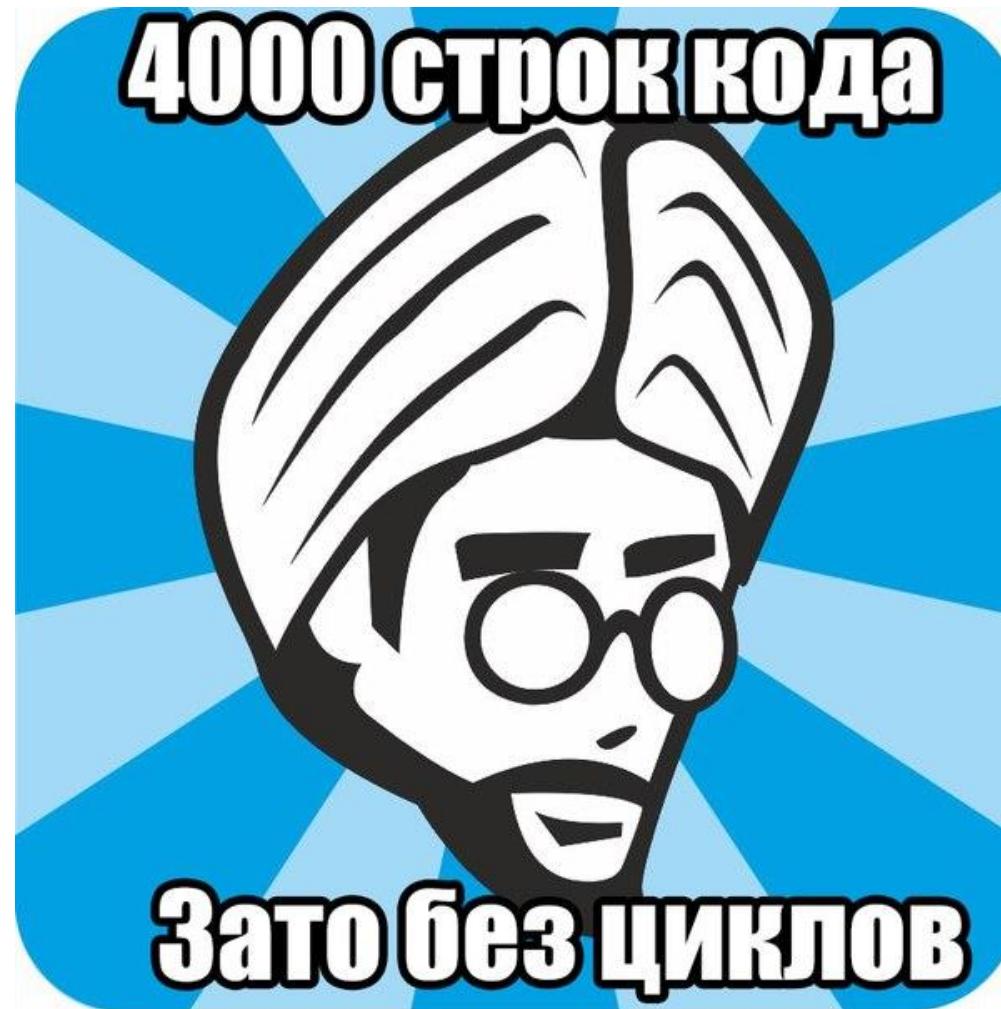
Intelligen



# Терминология

- Апликация = приложение
- Айбернет = хибернет
- Штрудель =Собака
- Компонент – использую с любым ударением
- Параметр = Параметр
- Иnam = Еnam
- Список пополняется...

# Кризис программного обеспечения



XAP – ~~х~~тестовая  
Архитектура Платформы



**BLACK FRIDAY**

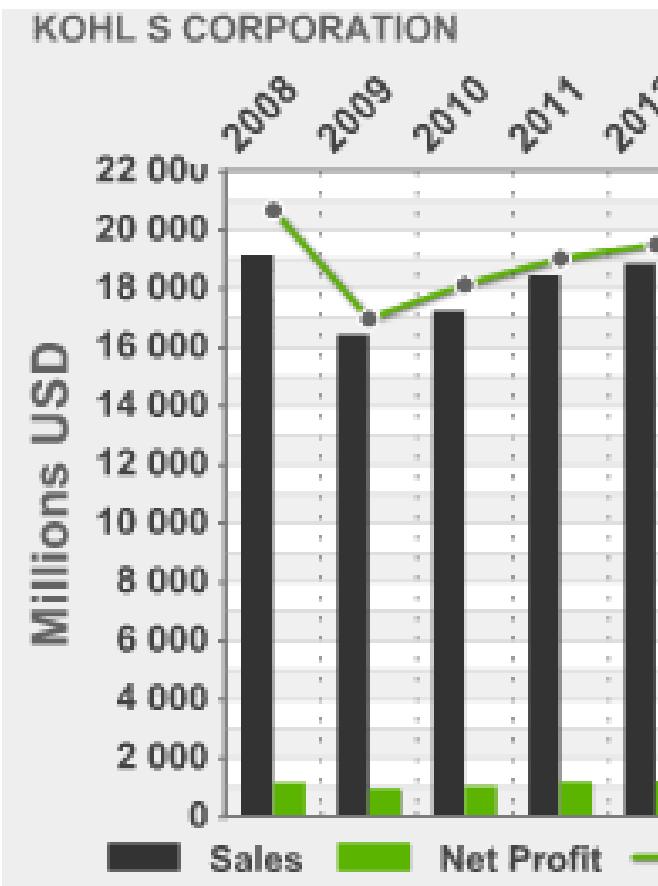
# BLACK FRIDAY – РУЛИТ!!!



# Компания кол – уневиремаг



# KOHL'S



# А что особенного случилось в 2009 году?



# Real-Life Market Situations



A Service of CNN, Fortune & Money

## The Dow's technical glitch, dissected

Publisher expects normal operations on back-up system, but still looking for root cause of Tuesday's glitch.

The publisher of the Dow industrials said that a system problem starting at 1:50 p.m. ET on Tuesday, amid unusually heavy trading volume, caused a 70-minute lag during which the value of the market measure lagged the declines in the underlying stocks.



Search PC World

Sea

Home News Hardware Reviews Software Reviews How-To Videos Downloads



Magazine  
Subscribe & Get  
a Bonus CD  
Customer Service

For all your IT

### FIND A REVIEW

Select Category

- Audio & Video
- Business Center
- Cameras
- Cell Phones & PDAs
- Communications
- Components & Hardware

Read More About: iPhone

### iPhone Activation Disasters

Three hours after getting my hands on one, I am ready to drop the thing from the 44th floor of the New York Hilton.

Jim Dalrymple, Macworld

Friday, June 29, 2007 8:00 PM PDT

## Black Friday Shoppers Crash Kohl's Web Site

By INYOUNG HWANG Bloomberg NEWS

Kohl's Corp.'s Web site crashed because of online traffic on Black Friday, a day when retailers offer bargains and the traditional start of the Christmas shopping season

This article was published November 28, 2009 at 4:37 a.m. Business, p.34

Срочно поднимайте еще сервера!!!



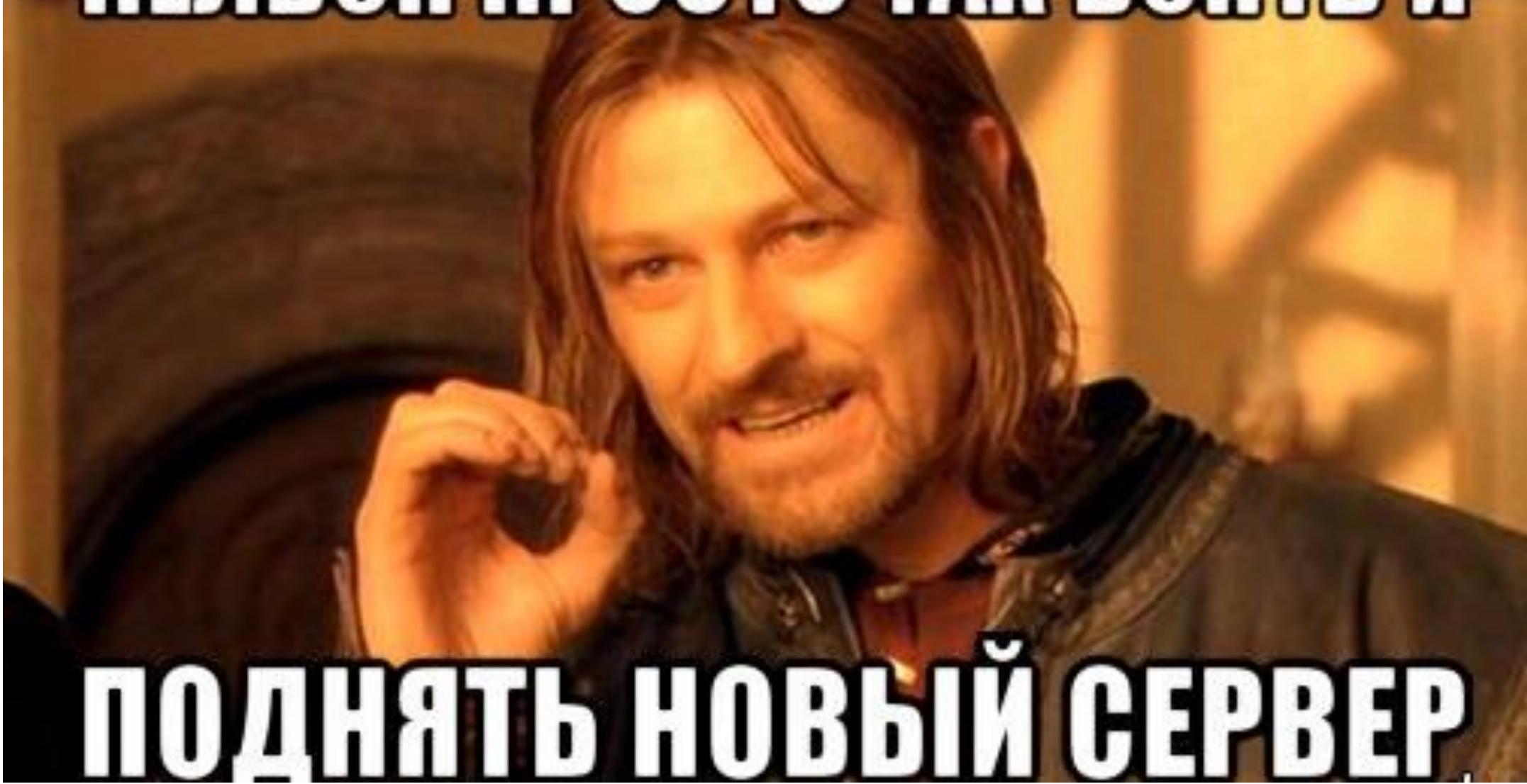
...

Слишком высокая нагрузка на сервер.



**НЕЛЬЗЯ ПРОСТО ТАК ВЗЯТЬ И**

**ПОДНЯТЬ НОВЫЙ СЕРВЕР.**



Потому что традиционная архитектура это...



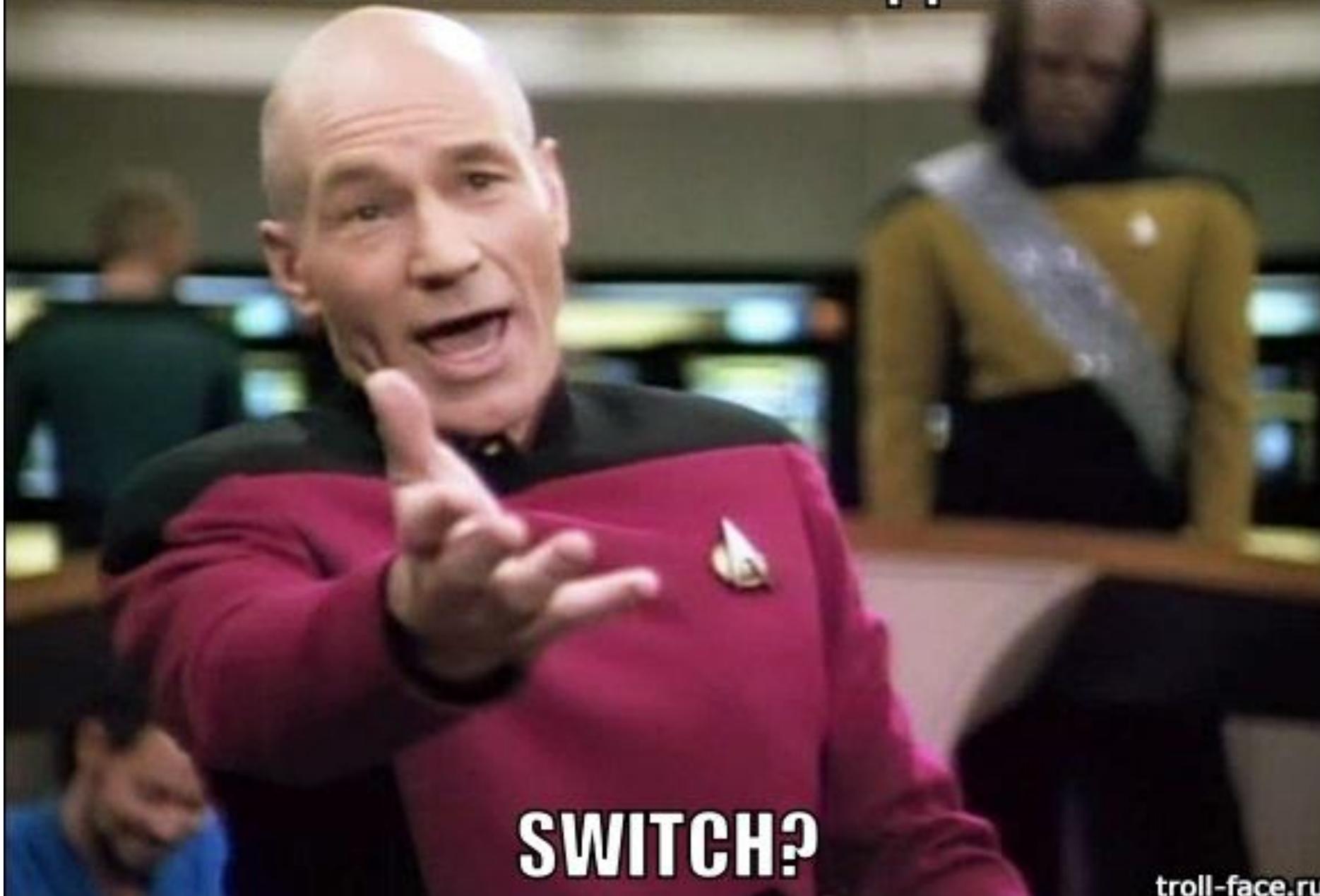
Пришла пора сменить старых героев



# Итак первое задание

- Как правильно использовать Switch
- Зачем нужен enum

ПОЧЕМУ ПРОСТО НЕ СДЕЛАТЬ



SWITCH?

Помни товарищ ты ведешь за собой народ



# Мы любим тебя, Switch ...

```
public DistribHandler resolveHandler(Integer.valueOf(documentObject.getDocumen
switch (documentObject.getDocumentType()) {
    case PDF_STORAGE:
        fileContainer = new PdfRecordFileContainer();
        getPdfFromStorage(fileContainer);
        break;
    case PDF_SRC:
        fileContainer = new PdfRecordFileContainer();
        fillObjectsForPdf(fileContainer, documentObject);
        break;
    case PDF_WS:
        fileContainer = new WsPdfRecordFileContainer();
        getPdfFromPdfWs(fileContainer, documentObject);
        break;
    case LIS:
        fileContainer = new PdfRecordFileContainer();
        getLisDocument(j, fileContainer);
        break;
    case IMAGE:
        fileContainer = new PdfRecordFileContainer();
        getPdfFromImageDocument(j, fileContainer);
        break;
    case FORM:
        fileContainer = new PdfRecordFileContainer();
        getFormDocument(fileContainer);
        break;
}
switch (value.getNumericValue()) {
    case 1:
        text = MessageFormat.format("{0} {1}", "הצעה לפולשת", emailRequest.getSubject());
        break;
    case 2:
        text = MessageFormat.format("{0} {1}", "רביישת לפולשת", emailRequest.getSubject());
        break;
    case 3:
        text = MessageFormat.format("{0} {1}", "חידוש לפולשת", emailRequest.getSubject());
        break;
    case 4:
        text = MessageFormat.format("{0} {1}", "שינויים בפולשת", emailRequest.getSubject());
        break;
    case 5:
    case 8:
        text = MessageFormat.format("{0} {1}", brandHebName);
        break;
    case 6:
    case 7:
        text = MessageFormat.format("{0} {1}", "כתב חשב ערך");
        break;
    case 9:
    case 17:
        text = MessageFormat.format("{0} - {1}", brandHebName);
        break;
    case 10:
    case 13:
        text = MessageFormat.format("{0} - {1}", emailRequest.getSubject(), brandHebName);
        break;
    case 11:
        text = emailRequest.getSubject();
        break;
    case 12:
        if (!resources.getBrandKey().isBituhYashir()) {
            text = format("5% מהתו", brandHebName);
        } else {
            text = format("5% מהתו", brandHebName);
        }
        break;
    case 14:
        text = MessageFormat.format("{0} - {1}", "ביטוח תואנות אושנות - השיקט הנפש שאל", brandHebName);
        break;
    case 15:
        text = MessageFormat.format("{0} {1}", "בקשה לאיירת קשר לחידוש בולטים", emailRequest.getSubject());
        break;
    case 16:
        text = MessageFormat.format("{0} {1}", icyDatFileConsumer.class);
        II && item.getAddresseeCode().length() >= 1
        de))) ? item.getAddresseeCode().trim() : "0";
        dressee(addressee);
        urrentPath, basket, dataConsumer);
}
icyDetailsConfConsumer.class);
urrentPath, basket, dataConsumer);

icyLetterConsumer.class);
lr(basket.getMetaDBasket().getPrtNr());
urrentPath, basket, dataConsumer);

icyCompulsoryConsumer.class);
etSeqNr(item.getCompSeqNr());
urrentPath, basket, dataConsumer);
```

# Подытожим недостатки Switch

- Куча нечитабельного кода
- Причина возникновение класса Бога
- Тяжело поддерживать
- RuntimeException аля NullPointerException, если ни один из case-ов не сработал
- Все комятят в один класс
- Когда-нибудь, кто-нибудь забудет break и тогда...

Надоело копаться в грязном коде?



# А что может заменить switch?

- Map
- Enum
- Аннотации

Сегодняшняя тема Enum



# Вопрос на 10 рублей: Сколько конструкторов может быть у Инама?

- А – 1
- Б – 57
- В – Сколько угодно
- Ни одного

# Вопрос на 10 рублей: Сколько конструкторов может быть у Инама?

- А – 1
- Б – 57
- В – СКОЛЬКО УГОДНО
- Ни одного

# Вопрос на 50 рублей: как лучше сравнивать идемы:

- А – ==
- Б – equals
- В – при помощи рефлексон
- Г – при помощи штангенциркуля

# Вопрос на 50 рублей : как лучше сравнивать идемы:

- А – ==
- Б – equals
- В – при помощи рефлексон
- Г – при помощи штангенциркуля

# Вопрос на 100 рублей:

## Конструктор Enum-а:

- А – public
- Б – protected
- В – private
- Ни один из ответов не верен

# Вопрос на 100 рублей:

## Конструктор Enum-а:

- А – public
- Б – protected
- В – private
- **НИ ОДИН ИЗ ОТВЕТОВ НЕ ВЕРЕН**

Вопрос на 250 рублей:

Выберите правильный ответ:

- А – В инаме могут быть только статические методы
- Б – В инаме не могут быть статические методы
- В – В инаме могут быть как статические так и не статические методы
- Г – В инаме вообще не может быть никаких методов, он же константа

Вопрос на 250 рублей:

Выберите правильный ответ:

- А – В инаме могут быть только статические методы
- Б – В инаме не могут быть статические методы
- В – В ИНАМЕ МОГУТ БЫТЬ КАК СТАТИЧЕСКИЕ ТАК И НЕ СТАТИЧЕСКИЕ МЕТОДЫ
- Г – В инаме вообще не может быть никаких методов, он же константа

Вопрос на 500 рублей:

Сколько методов инама можно переопределить:

- А – 1
- Б – 2
- В – 0
- Г – 3

Вопрос на 500 рублей:

Сколько методов инама можно переопределить:

- А – 1
- Б – 2
- В – 0
- Г – 3

# Вопрос на 1000 рублей:

## Кто может создавать объекты Enum-а:

- Класслоадер
- Чак Норрис
- Сборщик мусора
- Его можно создать через рефлексон

# Вопрос на 1000 рублей:

## Кто может создавать объекты Enum-а:

- КЛАССЛОАДЕР
- Чак Норрис
- Сборщик мусора
- Его можно создать через рефлексон

Вопрос на 2000 рублей:

Выберите неправильный ответ:

- В инаме можно прописать абстрактный метод, даже 2
- Методы прописанные в инаме автоматически final
- Инам может создать system classloader
- Дифолтный `toString` инама возвращает название объекта инама

Вопрос на 2000 рублей:

Выберите не правильный ответ:

- В инаме можно прописать абстрактный метод, даже 2
- **МЕТОДЫ ПРОПИСАННЫЕ В ИНАМЕ АВТОМАТИЧЕСКИ FINAL**
- Инам может создать system classloader
- Дифолтный `toString` инама возвращает название объекта инама

# Подитожим то что мы знаем

- У инама может быть много разных конструкторов, хотя обычно будет один, и пользоваться им может только класслоадер.
- Инам определяет не только метадату, но и все объекты данного типа.
- Все они статические и конечные
- Но имеют такую же функциональность как любой объект.
  - Методы
  - Проперти

# Что мы знаем про Enum

- Переопределить у него можно только `toString()`
- А значит `equals` (который делегирует в `==`) никто не переопределит
- Значит мы можем пользоваться `==` сами, чтобы не думать про `NullPointerException`-ы

В Инаме часто делаю статический метод,  
который вернёт нужный инам взависимости от  
входящих данных

# А вот мы добрались до типичного вопроса

- Что должен вернуть EntityManager если запрашиваемый объект не существует?
  1. Null
  2. Пустой объект
  3. Кинуть исключение
- А какой исключение лучше?

Checked



против  
Round 1

Runtime



# Checked Exceptions



Заставляет думать про обработку  
ексепшона

Нельзя забыть, потому что не будет  
компилироваться

А в случае RuntimeException можно не  
только это проморгать, но даже не знать  
о том, что эксепшон может случиться

# Runtime Exceptions

Зато чистый код, в котором нет ужасных try & catch

Что пишут в 99% внутри catch?

Правильно log.error(e)

А это и так будет, если его не ловить



Checked



против  
Round 2

Runtime



# Checked Exceptions



Checked exception провоцируют полезные мысли. Раз уж все равно надо писать catch, то почему бы не подумать какую еще информацию передать в логгер

А в случае Runtime Exception часто можно встретить NullpointerException без всякой дополнительной информации

# Runtime Exceptions

Пустые логи, это как раз проблема не того, кто пользуется `RuntimeException`, а того, кто возвращает `null`, вместо того, чтобы его кинуть

Но если кто то забыл поставить `catch` то `exception` все равно попадет в лог.



# Runtime Exceptions

А теперь скажите честно, вам доводилось видеть пустые catch-и, с каким-нибудь todo внутри?

**Вот это – действительно страшно**



Checked



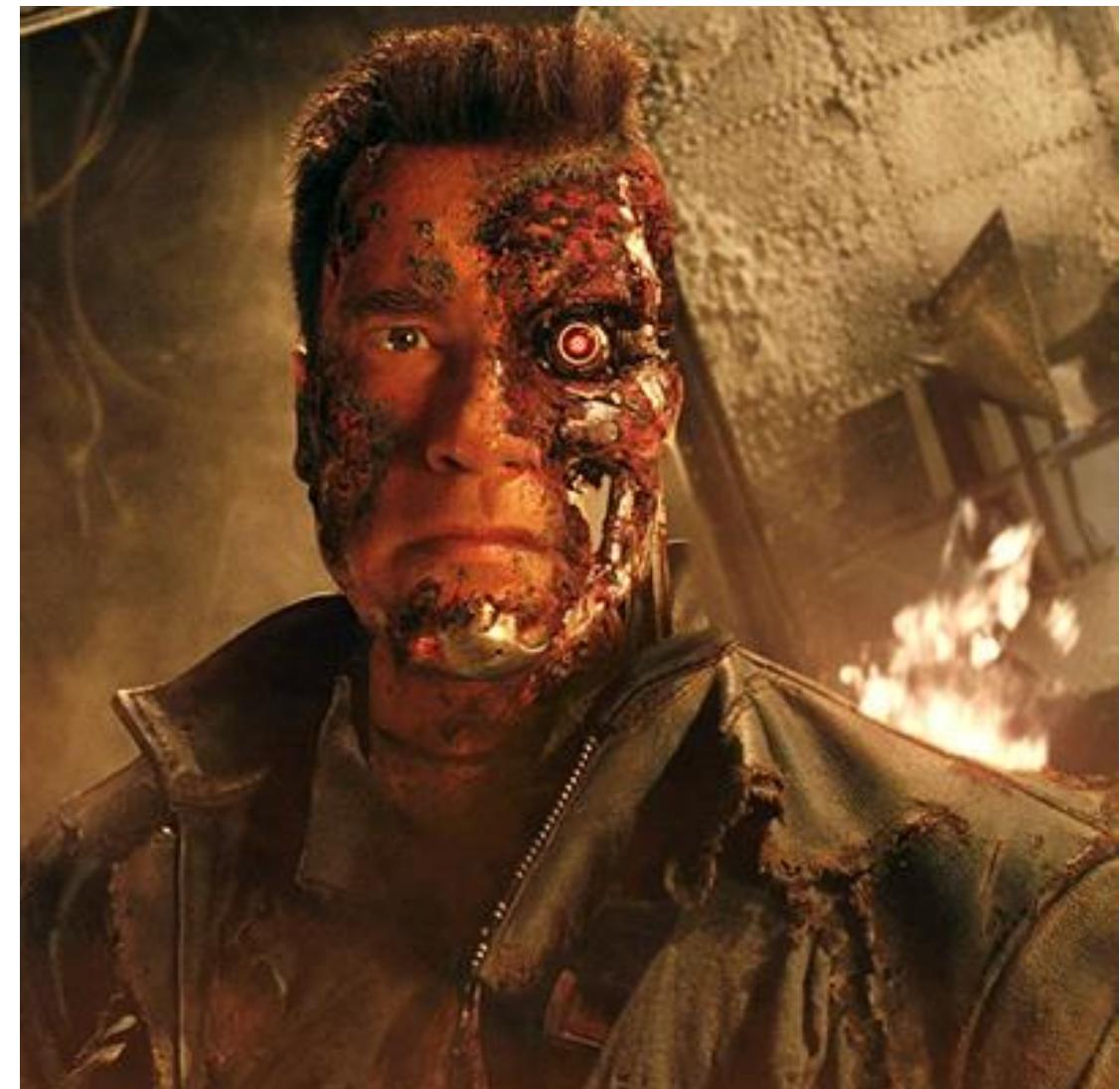
против  
Round 3

Runtime



# Checked Exceptions

Но ведь всегда можно добавить throws к декларации метода и таким образом превратить checked в runtime



# Runtime Exceptions

А вот хрен!

Экспеншон, который является часть подписи метода, ограничивает тех, кто его переопределяет, и часто нет возможности добавить новый экспеншон к подписи метода



Победа Uncheked далась не лёгко





Checked  
vs  
Unchecked

Бог не устроил бы потопа  
И не был нужен бы ковчег,  
Когда б exception-ы кидались  
unchecked

# Lombok – composition / delegate and friends

- Annotation Processor
- Работает на этапе компиляции
- Генерит код, чтобы мы не писали
- Делает джаву более похожим на нормальный язык

# Что полезно знать про lombok для написания модели и не только

- @Data – POJO (@Getter, @Setter, @ToString, @EqualsAndHashCode)
- @Value – immutable POJO
- @AllArgumentConstruct

# Что полезно знать про lombok для написания модели и не только

- @Data – POJO (@Getter, @Setter, @ToString, @EqualsAndHashCode)
- @Value – immutable POJO
- @AllArgsConstructor(onConstructor = @\_(@Autowired))
- @RequireArgumentConstructor /@NoArgumentConstructor
- @Builder / @Singular
- @Delegate – примерно как в груви
- @SneakyThrows
- @Slf4j / @Log4j / ...

Ладно, решение через  
Enum понятно, а через  
Аннотации то как?

Стоп! А насколько хорошо вы знаете Reflection?



# Опрос: ваш уровень знания Reflection



1. Я никогда не слышал, что такое reflection
2. Я знаю зачем он нужен
3. Я его использую
4. Я писал свои аннотации и считывал их в runtime
5. Я знаю разницу между RetentionPolicy.RUNTIME, RetentionPolicy.SOURCE и RetentionPolicy.CLASS

# Reflections

Evgeny Borisov

# Task – Zoo Game

- Write tree of animal classes (Tiger, Dog, Cow...)
- All classes extends from Animal and override method makeSound.
- Write Factory class which will have method createZoo.
- This method will get an integer and will return list of random animals equal to this number
- Test your application.

# Animal

```
public abstract void makeSound();
```

## Dog

```
@Override  
public void makeSound() {  
    out.println("Muuuuuu");  
}
```

## Tiger

```
@Override  
public void makeSound() {  
    out.println("au au au");  
}
```

## Cow

```
@Override  
public void makeSound() {  
    out.println("Rrrrrrr");  
}
```

```
public interface AnimalFactory {  
    List<Animal>createZoo(int number);  
}  
  
public class SimpleAnimalFactory implements AnimalFactory {  
    public List<Animal> createZoo(int number) {  
        ArrayList<Animal> animals = new ArrayList<Animal>();  
        for (int i = 0; i < number; i++) {  
            animals.add(createRandomAnimal());  
        }  
        return animals;  
    }  
  
    public Animal createRandomAnimal() { ... }  
}
```



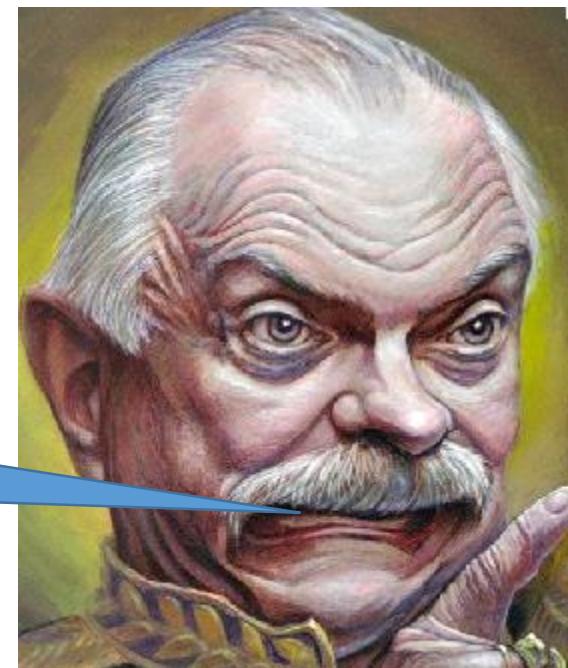
And what code  
is there?

# CreateRandomAnimal Method

```
public Animal createRandomAnimal() {  
    int randomNumber = (int) (Math.random() * 3);  
    switch (randomNumber) {  
        case 0:  
            return new Cow();  
        case 1:  
            return new Tiger();  
        case 2:  
            return new Dog();  
    }  
    return null;  
}
```

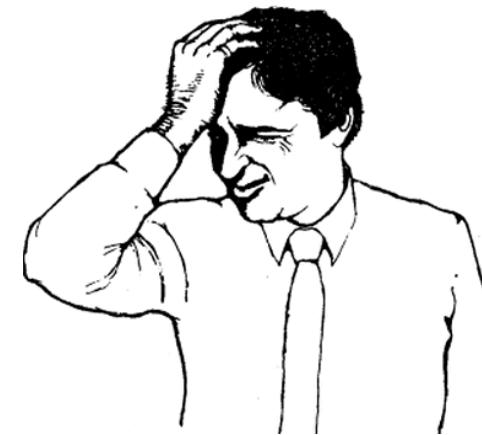
Add a cat please

שינוי איפיון



```
public Animal createRandomAnimal() {  
    int randomNumber = (int) (Math.random() * 3);  
    switch (randomNumber) {  
        case 0:  
            return new Cow();  
        case 1:  
            return new Tiger();  
        case 2:  
            return new Dog();  
        case 3:  
            return new Cat();  
    }  
    return null;  
}
```

Yes... Some day it  
will happen



# Why this solution is bad?

- Hard to maintain (someday you will forget)
- Don't use switch! Remember?
- New animal type require change in factory class

# Good Solution

- Animal factory should read all animal class names in bootstrap and add them to its cache
- createRandomAnimal method should fetch random animal class type from cache and create new instance from it

HOW???





# Introduction

- With the reflection API you can examine or manipulate:
  - Classes
    - Get modifiers, fields, methods, constructors, and superclasses.
    - Create an instance of a class whose name is not known until runtime.
  - Interfaces
    - Get constants and method
  - Objects
    - Get class of an object
    - Get and set the value of field, even if the field name is unknown to your program until runtime
    - Invoke a method on an object, even if the method is not known until runtime
  - Arrays
    - Create a new array, whose size and component type are not known until runtime, and then modify the array's components.

# Uses

- Software development tools
  - Frameworks, services, factories
  - Debuggers
  - GUI builders
  - implementing drag and drop of components
  - etc.

# The Reflection API

- `java.lang` package
  - `Class` – Represents, or reflects, classes and interfaces.
  - `Object` – Provides the `getClass` method.

# The Reflection API – `java.lang.reflect`

- Reflection package provides API for:
  - Examining Classes
    - construct new class instances and new arrays
    - information about class's modifiers
  - Manipulating Objects
    - access and modify fields of objects and classes
    - invoke methods on objects and classes
  - Arrays
    - access and modify elements of arrays

# class Class

- For each class, the Java Runtime Environment (JRE) maintains an immutable `Class` object that contains information about the class. (metadata)
- A `Class` object represents, or reflects, the class
- `class Class` include methods which return `Constructor`, `Method`, and `Field` objects

```
class Class
```

- **Class** objects also represent *interfaces*.
  - `isInterface` method
    - Determines if the specified `Class` object represents an interface type
    - You invoke `Class` methods to find out about an interface's *modifiers*, *methods*, and public *constants*

# Retrieving Class Objects

- Retrieving Class Objects
  - invoke Object.getClass
    - Class c = unknownObject.getClass()
  - getSuperclass method
    - List t = new ArrayList();  
Class c = t.getClass();  
Class s = c.getSuperclass();

# Retrieving Class Objects

- **forName** method - If the class name is unknown at compile time

- String className = "java.io.File";

- ...

- Class c = Class.forName(className);

- The method `forName` creates (instantiates) a Class object

# Examining Classes

- **Constructor** – Provides information about, and access to, a constructor for a class.  
Allows you to instantiate a class dynamically.
- **Modifier** – Provides static methods and constants that allow you to get information about the access modifiers of a class and its members.

# Class Constructors

- Class's `getConstructors` method
  - returns an array of `Constructor` objects
  - Constructor class includes methods to
    - retrieve constructor's name, set of modifiers, parameter types, and set of throwable exceptions.
    - create a new instance of the `Constructor` object's class with the `Constructor.newInstance` method

# Class Constructors

```
// retrieve constructors' information
static void showConstructors(Object o) {
    Class c = o.getClass();
    Constructor[] theConstructors =
        c.getConstructors();
    for (int i = 0; i < theConstructors.length; i++) {
        System.out.print("(" );
        Class[] parameterTypes =
            theConstructors[i].getParameterTypes();
        for (int k=0; k<parameterTypes.length; k++) {
            String parameterString =
                parameterTypes[k].getName();
            System.out.print(parameterString + " ");
        }
        System.out.println(")");
    }
}
```

# Constructors - Creating an Object

- Creating Objects
  - Create an instance (new object) of class which is unknown until runtime
    - Class's `newInstance()` method
      - Creates a new instance of the class represented by this `Class` object
      - used for parameterless constructors
    - Constructor's `newInstance(Object[] args)` method
      - Uses the constructor represented by this `Constructor` object to create and initialize a new instance of the constructor's declaring class, with the specified initialization parameters.
      - used for constructors with parameters

# Constructors - Creating an Object

```
// Constructor without parameters
static Object createObject(String className) {
    Object object = null;
    try {
        Class classDefinition = Class.forName(className);
        object = classDefinition.newInstance();
    } catch (InstantiationException e) {
        // if this Class represents an abstract class,
        // an interface, an array class, a primitive
        // type, or void; or if the instantiation
        // fails for some other reason.
        System.out.println(e);
    } catch (IllegalAccessException e) {
        //if the class or initializer is not accessible
        System.out.println(e);
    } catch (ClassNotFoundException e) {      System.out.println(e);
    }
    return object;
}
```

# Constructors - Creating an Object

```
// Constructor with parameters
public static Object createObject(Constructor constructor,
                                  Object[] arguments) {

    System.out.println ("Constructor: " + constructor.toString());
    Object object = null;
    try {
        object = constructor.newInstance(arguments);
        System.out.println ("Object: " + object.toString());
        return object;
    } catch (InstantiationException e) {
        System.out.println(e);
    } catch (IllegalAccessException e) {
        System.out.println(e);
    } catch (IllegalArgumentException e) {
        System.out.println(e);
    } catch (InvocationTargetException e) {
        System.out.println(e);
    }
    return object;
}
```

# class Class – Modifiers

- Class modifiers: public, abstract, final, ...
  - Method `getModifiers` - retrieve a set of modifiers.
  - `class Modifier` includes static methods: `isPublic`, `isAbstract`, `isFinal`, ...

```
public static void printModifiers(Object o) {  
    Class c = o.getClass();  
    int m = c.getModifiers();  
    if (Modifier.isPublic(m))  
        System.out.println("public");  
    if (Modifier.isAbstract(m))  
        System.out.println("abstract");  
    if (Modifier.isFinal(m))  
        System.out.println("final");  
}
```

# class Class – interfaces

- Identifying the Interfaces Implemented by a Class
  - getInterfaces method

```
static void printInterfaceNames(Object o) {  
    Class c = o.getClass();  
    Class[] theInterfaces = c.getInterfaces();  
    for (int i=0; i<theInterfaces.length; i++) {  
        String interfaceName =  
            theInterfaces[i].getName();  
        System.out.println(interfaceName);  
    }  
}
```

# Manipulating Objects

- **Field** – Provides information about, and dynamic access to, a field of a class or an interface.
- **Method** – Provides information about, and access to, a single method on a class or interface. Allows you to invoke the method dynamically.

# Class Fields

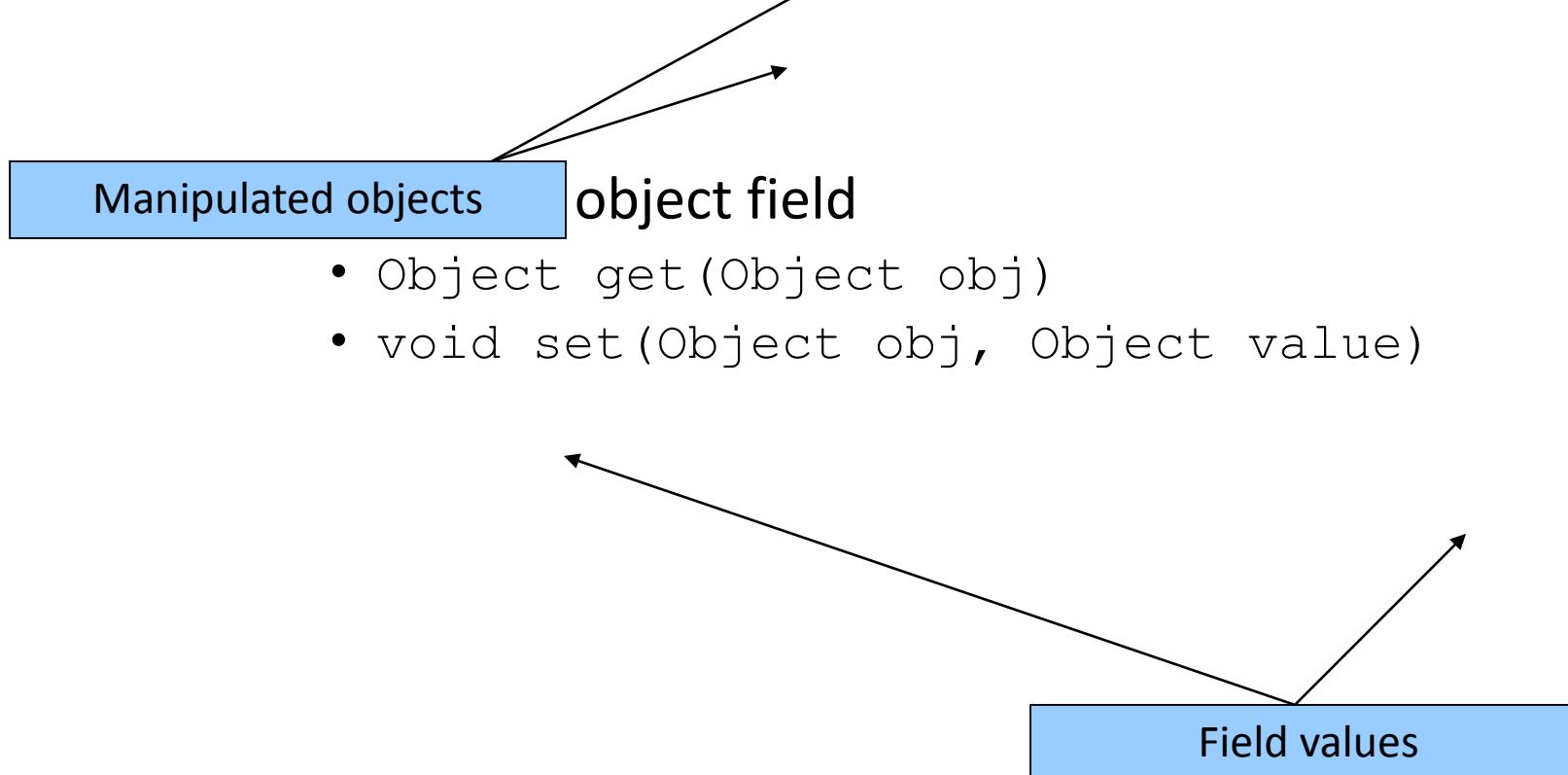
- Class's `getFields` method
  - returns an array of `Field` objects
    - includes the fields of all superclass of the class and all the interfaces it implements.
  - `Field` class includes methods to
    - retrieve the field's name, type, set of modifiers
    - get and set the value of a field

# Class Fields

```
// retrieving field information
static void printFieldNames(Object o) {
    Class c = o.getClass();
    Field[] publicFields = c.getFields();
    for (int i = 0; i < publicFields.length; i++) {
        String fieldName = publicFields[i].getName();
        Class typeClass = publicFields[i].getType();
        String fieldType = typeClass.getName();
        System.out.println("Name: " + fieldName + ",           Type: "
+ fieldType);
    }
}
```

# Class Fields – Field Manipulation

- get and set the value of a field
  - values of primitive types
    - `getInt`, `getFloat`, ...
    - `setInt`, ~~SetFloat~~, ...



# Class Fields – Field Manipulation

```
// getting field value
static void printHeight(Rectangle r) {
    Field heightField;
    Integer heightValue;
    Class c = r.getClass();
    try { heightField = c.getField("height");
        heightValue = (Integer) heightField.get(r);
        System.out.println("Height: " +
                           heightValue.toString());
    } catch (NoSuchFieldException e) {
        System.out.println(e);
    } catch (SecurityException e) {
        System.out.println(e);
    } catch (IllegalAccessException e) {
        System.out.println(e);
    }
}
```

# Class Methods

- Class's `getMethods` method
  - returns an array of `Method` objects
  - `Method` class includes methods to
    - retrieve method's name, return type, parameter types, set of modifiers, and set of throwable exceptions
    - `Method.invoke` - call the method itself

# Class Methods

```
// retrieve method's information

static void showMethods(Object o) {
    Class c = o.getClass();
    Method[] theMethods = c.getMethods();
    for (int i = 0; i < theMethods.length; i++) {
        String methodString = theMethods[i].getName();
        System.out.println("Name: " + methodString);
        String returnType =
            theMethods[i].getReturnType().getName();
        System.out.println(" Return Type: " +
                           returnType);
        Class[] parameterTypes =
            theMethods[i].getParameterTypes();
        System.out.print(" Parameter Types:");
        for (int k=0; k<parameterTypes.length; k++) {
            String parameterString =
                parameterTypes[k].getName();
            System.out.print(" " + parameterString);
        }
        System.out.println();
    }
}
```

# Class Methods

- **Output:**

Name: equals

Return Type: boolean

Parameter Types: java.lang.Object

Name: getClass

Return Type: java.lang.Class

Parameter Types:

Name: hashCode

Return Type: int

Parameter Types:

... .

# Class Methods – Invoking Methods

```
//invoke a method dynamically
Object invokeMethod(Object o, String methodName,
                    Class[] parameterTypes, Object[] arguments){

    Object result;
    Class c = o.getClass();
    try {
        Method method = c.getMethod(methodName,
                                      parameterTypes);

        Object result = method.invoke(o, arguments);
    } catch (NoSuchMethodException e) {
        System.out.println(e);
    } catch (IllegalAccessException e) {
        System.out.println(e);
    } catch (InvocationTargetException e) {
        System.out.println(e);
    }
    return result;
}
```

# Lets go back to the Task

```
public class ReflectionalAnimalFactory implements AnimalFactory {  
  
    private List<Class<? extends Animal>> animalsMDList = new ArrayList<Class<? extends Animal>>();  
  
    public ReflectionalAnimalFactory() throws ClassNotFoundException {...}  
  
    public List<Animal> createZoo(int number) throws IllegalAccessException, InstantiationException  
  
    private Animal createRandomAnimal() throws IllegalAccessException, InstantiationException {...}  
}
```

# AnimalFactory constructor

```
public ReflectionalAnimalFactory() throws ClassNotFoundException {

    File file = new File("../ZooWebApplication/src/main/java/com/idi");
    String[] list = file.list();
    for (String s : list) {
        if (Character.isLowerCase(s.charAt(0))) {
            continue; —→ Not a class (class starts with capital letter
        }
        File tempFile = new File("../src/zoo/animals/" + s);
        if (tempFile.isDirectory()) {
            continue;
        }
        String className = s.split("\\.")[0];
        Class clazz = Class.forName("com.idi." + className);
        if (com.idi.Animal.class.isAssignableFrom(clazz) &&
            !Modifier.isAbstract(clazz.getModifiers())) {
            animalsMDList.add(clazz);
        }
    }
}
```

# Creating a zoo

```
public List<Animal> createZoo(int number) throws IllegalAccessException  
    ArrayList<Animal> animals = new ArrayList<Animal>();  
    for (int i = 0; i < number; i++) {  
        animals.add(createRandomAnimal());  
    }  
    return animals;  
}
```

```
private Animal createRandomAnimal() throws IllegalAccessException
```



And what code  
is there?

# Creating a random animal

```
private Animal createRandomAnimal() throws IllegalAccessException,  
                                         InstantiationException {  
  
    int randomNumber = (int) (Math.random() * animalsMDList.size());  
    Class<? extends Animal> animalMD = animalsMDList.get(randomNumber);  
    return animalMD.newInstance();  
}
```

# Annotations

Аннотации это не магия, а просто метадата.  
Их можно ставить над:

package declarations,  
class,  
constructors,  
methods,  
fields,  
Variables and etc.

# Виды аннотаций

- **Marker**
- **Single-Element**
- **Full-value or multi-value**

# Marker

Аннотация без параметров.

```
public @interface MyTransactionAnnotation {  
}  
  
@MyTransactionAnnotation  
public void persistPerson2DB(Person p){  
    //all business logic here  
}
```

# Single-Element

Аннотация с один параметром, и если его имя: “value” то указывать его при использовании не обязательно

```
public @interface SQL {  
    String value();  
}  
  
@SQL("update PERSON blabla")  
public Person updatePersonsAge(Person p, int age){  
}
```

# Full-value or multi-value

## Аннотация с несколькими параметрами

```
public @interface Cached {  
    int maxElementsInMemory();  
    long secondsToLive();  
}
```

```
@Cached(maxElementsInMemory = 30, secondsToLive = 24*60*60)  
public Data getSomeData(String whatToFind){
```

# The Target of annotation

- @Target(ElementType.TYPE)—can be applied to any element of a class
- @Target(ElementType.FIELD)—can be applied to a field or property
- @Target(ElementType.METHOD)—can be applied to a method level annotation
- @Target(ElementType.PARAMETER)—can be applied to the parameters of a method
- @Target(ElementType.CONSTRUCTOR)—can be applied to constructors
- @Target(ElementType.LOCAL\_VARIABLE)—can be applied to local variables
- @Target(ElementType.ANNOTATION\_TYPE)—indicates that the declared type itself is an

# Annotation Retention Policy

- Source
  - Annotations are to be discarded by the compiler
- Class
  - Annotations are to be recorded in the class file by the compiler but need not be retained by the VM at run time
- Runtime
  - Annotations are to be recorded in the class file by the compiler and retained by the VM at run time, so they may be read reflectively

# Annotation Retention Policy

- Runtime

← *Какой самый крутой?*

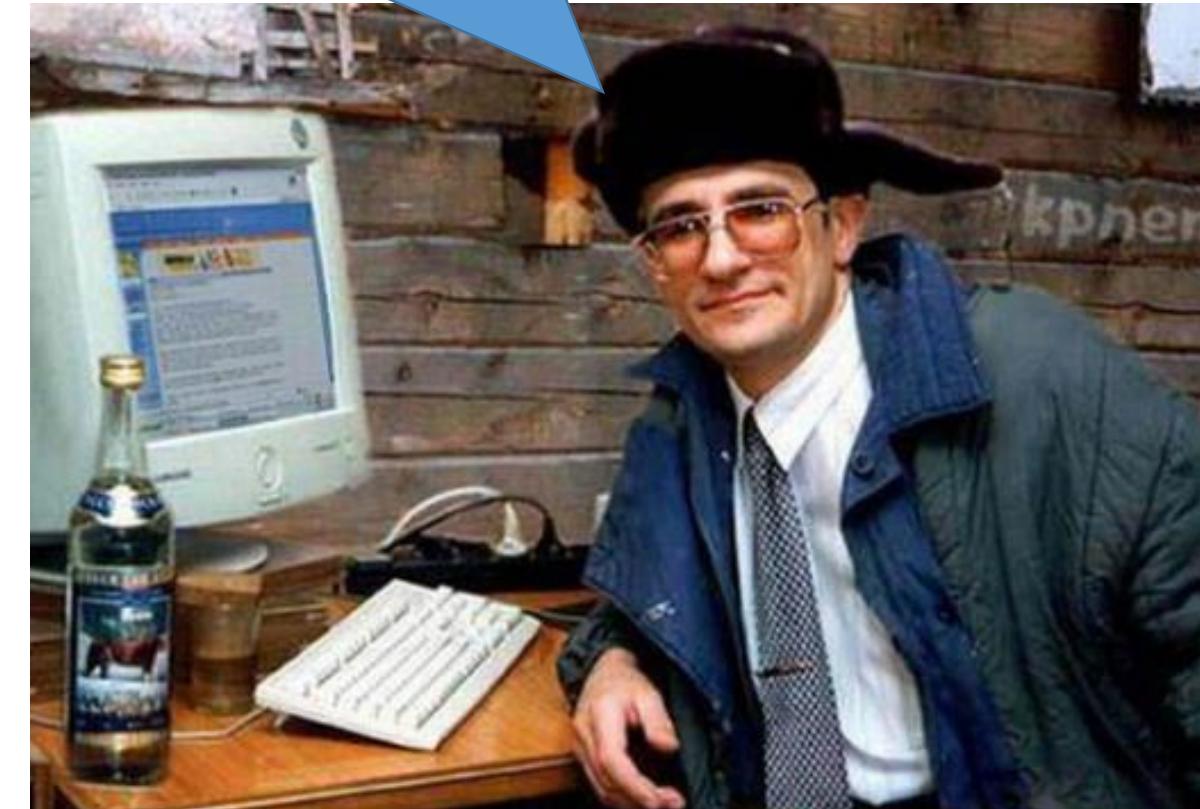
- Class

← *Какой самый бесполезный?*

- Source

↑ *Что по умолчанию?*

Retention: Source... А кому это надо?  
Чем это от комментариев отличается?



# Загадка...

```
public class BetterThanBestService extends BestService {  
  
    public void best() {  
        System.out.println("Android the best");  
    }  
  
}  
  
  
public static void main(String[] args) {  
    new BetterThanBestService();  
}
```



```
public class BestService {  
    public BestService(){  
        best();  
    }  
  
    private void best() {  
        out.println("iphone the best");  
    }  
}
```

1. Will not compile.
2. Runtime exception.
3. iPhone the best.
4. Android the best.

# Riddle

```
public class BetterThanBestService extends BestService {  
    @Override  
    Method does not override method from its superclass  
    System.out.println("Android the best");  
}
```

```
public static void main(String[] args) {  
    new BetterThanBestService();  
}
```

```
public class BestService {  
    public BestService(){  
        best();  
    }  
  
    private void best() {  
        out.println("iphone the best");  
    }  
}
```

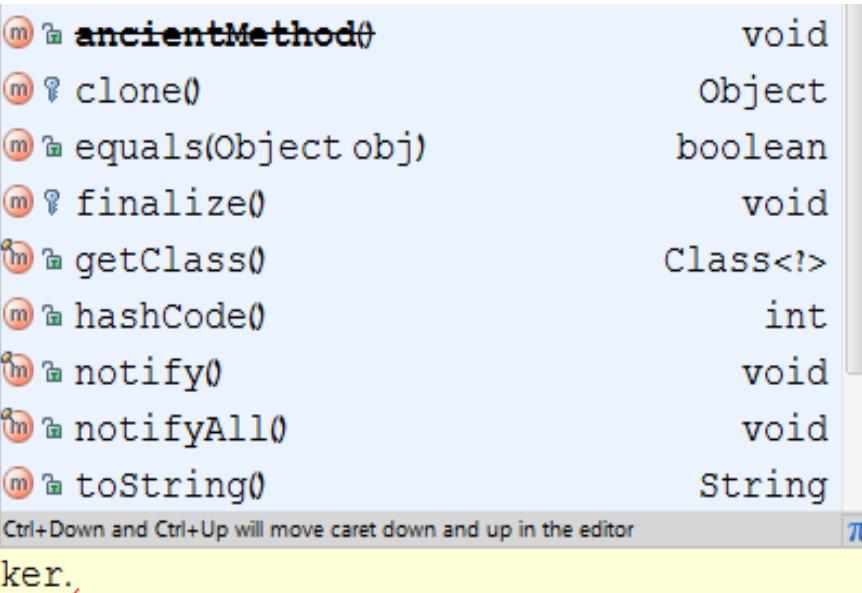
# RetentionPolicy = Source

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {

}

@Deprecated
public void ancientMethod() {
    //any code
}

public static
RussianSpeaker
russianSpeaker.
```



The screenshot shows an IntelliJ IDEA code editor with a completion dropdown open over the line 'russianSpeaker.'. The dropdown lists various methods from the Object class, each with its name, return type, and parameter types. The methods listed are: ancientMethod(), clone(), equals(Object obj), finalize(), getClass(), hashCode(), notify(), notifyAll(), and toString(). The 'ancientMethod()' method is highlighted in the list. A tooltip at the bottom of the dropdown says 'Ctrl+Down and Ctrl+Up will move caret down and up in the editor'.

| Method             | Return Type |
|--------------------|-------------|
| ancientMethod()    | void        |
| clone()            | Object      |
| equals(Object obj) | boolean     |
| finalize()         | void        |
| getClass()         | Class<?>    |
| hashCode()         | int         |
| notify()           | void        |
| notifyAll()        | void        |
| toString()         | String      |

# RetentionPolicy = RUNTIME

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyTransactionAnnotation {

}

@MyTransactionAnnotation
public void persistPerson2DB(Person p){
    //all business logic here
}
```

# Итак вот вам первый анти паттерн God Class



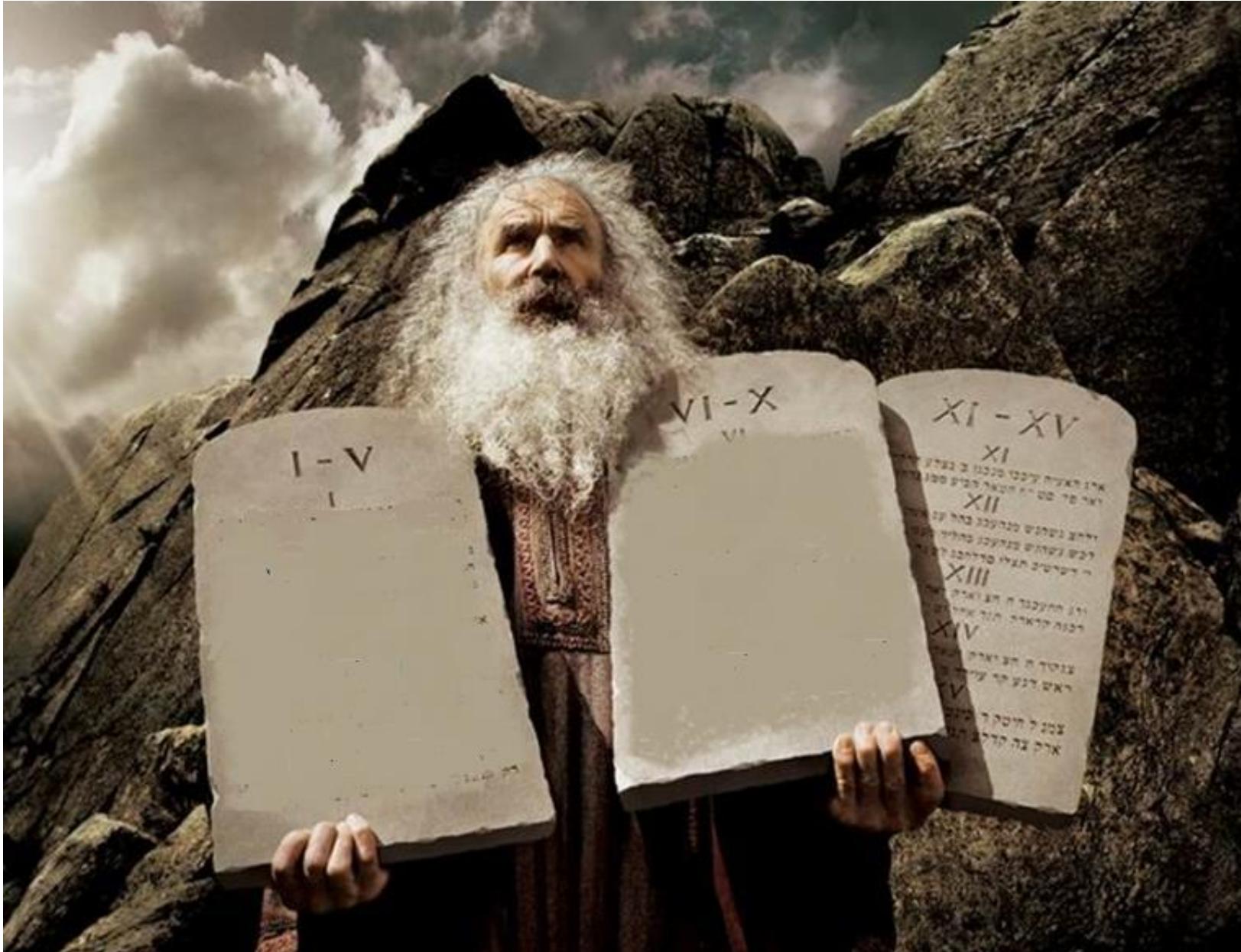
Один из причин его возникновения это спагетти код, в результате:

1. If / else
2. While
3. Switch / case
4. goto

# Признаки плохого дизайна

- **Rigid** – Сложно делать рефактор, так как одно изменение провоцирует кучу других и из этого часто не выйти
- **Fragile** – изменение в одном месте системы ломает что-то в другом, совершенно непредсказуемым образом
- **Immobile** – невозможно реюзить уже существующие компоненты, так как они не могут работать вне контекста конкретной апликации, что часто приводит к копи пасту.

# Две главные заповеди программиста



# Две главные заповеди программиста



# Признаки плохого кода



# Коментарии

**“Comments often are used as a deodorant”**

— *Refactoring*, Martin Fowler and Kent Beck

# Когда комменты уместны?

- Объяснить как работает алгоритм
- Объяснить что делает алгоритм
- Для API
- Объяснить смысл переменных
- Чтобы описать класс, показать дополнительную полезную информацию

# Когда комменты уместны?

- **Объяснить как работает алгоритм**
- ~~Объяснить что делает алгоритм~~
- **Для API**
- ~~Объяснить смысл переменных~~
- **Чтобы описать класс, показать дополнительную полезную информацию**

# Избавляемся от комментариев

- Rename
  - Fields
  - Constants
  - Methods
  - Input parameters
  - Classes
- Extract method
  - Заменять блоки комментариев вытаскивая блоки кода в методы и давая им понятные названия

# Java Конвенции

- Класс начинается с большой буквы!!!
- Константы пишутся большими буквами и чтобы разделять слова используется НИЖНЕЕ\_ПОДЧЕРКИВАНИЕ
- НИЖНЕЕ\_ПОДЧЕРКИВАНИЕ больше нигде не используется
- Забудьте всякие \_methodVariable=3
- Кроме класса(Type) (это например еще и интерфейс) Ничего не начинается с большой буквы.
- НИЧЕГО НЕ НАЧИНАЕТСЯ С БОЛЬШОЙ БУКВЫ, кроме вышесказанного!!!
- Все слова разделяются при помощи Заглавной буквы следующего слова

# Java Конвенции - примеры

- Название интерфейса: PersonService
- Да да, никаких I в начале – это вам не C#
- Название класса: PersonServiceImpl
- Название метода: printPersonDetails
- Декларирование и имена переменных:  
String name, int age, Person person
- Константа: final int КОЛИЧЕСТВО\_РЁБЕР\_У\_ЖЕНЩИНЫ = 24
- Константа: final int КОЛИЧЕСТВО\_РЁБЕР\_У\_МУЖЧИНЫ = 23

Зачем соблюдать  
конвенции?



Профессионал видит тут 5 ошибок, а сколько видите вы?





**new balance()**

935 6549  
999 23040

0.16  
0.0744

NEW

NEW

NEW

# Как давать имена

- Удовлетворить компайлер
- Удовлетворить джава конвенции
- Не делать грамматических ошибок
- Чтобы можно было выговорить (dlgFnEvalItr)
- При названии класса объяснить его функциональность (Service, Listener, Model...)
- Будь последовательным (calcPrice, calculateTime, getCalculatedResult)
- Никаких Ifc, а Impl
- Не экономить на длинне, думать о том, как будут потом искать (alt+cntrl+shift+n или с 13 интеледжей shift shift)

# Давать такие имена, чтобы потом не забыть



-А ведь зачем-то на кухню ходил...

Чего-то в лапах нёс...

Проклятый склероз!

# Еще признаки плохого кода

- Длинные методы



# Почему люди пишут длинные методы?

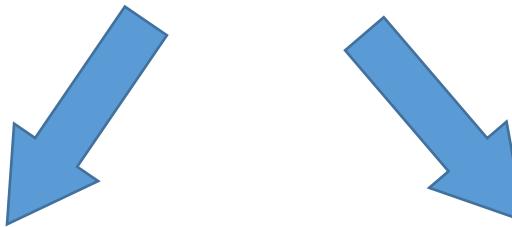
- Меняются требования к программе и уже написанный метод удлиняется
- Боязнь отойти от описания
- Боязнь сбиться с мысли
- Паранойя на почве перформенса

# Кстати о перформенсе



## Одна из причин багов...

Находятся умельцы, которые пишут вручную оптимизации



Получаются баги   Код становится не читабельный



А это уже платформа для багов

баг      баг      баг      баг      баг

```
graph TD; C[А это уже платформа для багов] --> D[баг]; C --> E[баг]; C --> F[баг]; C --> G[баг]; C --> H[баг];
```

<http://habrahabr.ru/post/165729/>

14 января в 14:03

## Предельная производительность: C#

из песочницы

 Программирование\*, Параллельное программирование\*, Высокая производительность\*

Я поделюсь 30 практиками для достижения максимальной производительности приложений, которые этого требуют. Затем, я расскажу, как применил их для коммерческого продукта и добился небывалых результатов!

Приложение было написано на C# для платформы Windows, работающее с Microsoft SQL Server. Никаких профайлеров – содержание основывается на понимании работы различных технологий, поэтому многие топики пригодятся для других платформ и языков программирования.



Предисловие

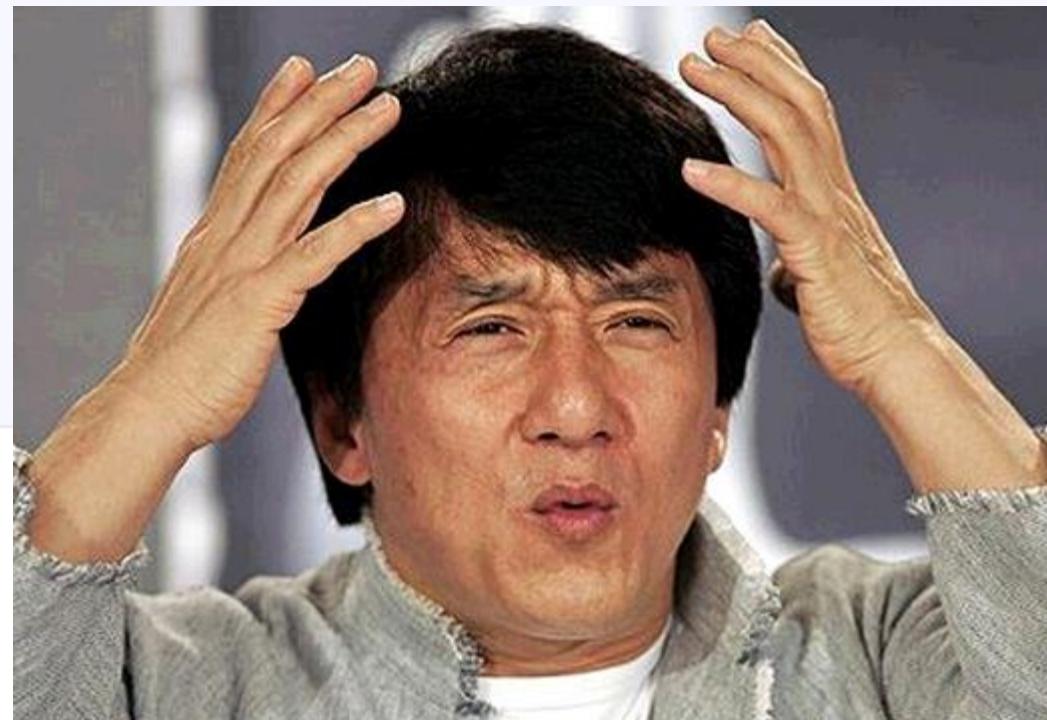
# Don't do that in Java

## 7. Разматывайте циклы.

Любой цикл – это та же конструкция «if». Если количество итераций небольшое, и их количество заранее известно, то иногда цикл лучше заменить на его тело, которое повторяется необходимое кол-во итераций (да, один раз Copy и N раз Paste).

### ▼ Пример – возведение числа в 4-ю степень

```
int power4(int v)
{
    int r = 1;
    r *= v;
    r *= v;
    r *= v;
    r *= v;
    return r;
}
```



Кстати про оптимизации HotSpot

A black and white photograph of two men from the chest up. Both men are wearing dark top hats and have mustaches. The man on the left is wearing a light-colored shirt and a dark, textured jacket. The man on the right is wearing a light-colored shirt and a dark, solid-colored jacket. They are both looking upwards and slightly to the left. A large, semi-transparent blue speech bubble is positioned in the lower center of the frame, containing the text.

Послушайте, а  
что это вообще  
значит?  
Hot Spot?

# HotSpot



# Just-In-Time Compilation

- Что мы знаем про JIT!
- Код который считается горячим кэшируется и оптимизируется
- А что значит горячий?
  - Для Сервер VM – 10000 инвокаций
  - Для Клиент VM – 1500 инвокаций
  - Можно поменять: `-XX:CompileThreshold=#`
    - Чем больше инвокаций – тем круче оптимизации
    - Чем меньше инвокаций – тем короче время разогрева

# Оптимизации HotSpot-а

- JIT Compilation
  - Оптимизации компайлера
  - Генерация более высококачественного кода, чем вы могли бы написать на нативном языке
- Adaptive Optimization
- Split Time Verification

# Адаптивные оптимизации

- Позволяют декомпилировать код
- Благодаря этому возможны более агрессивные и даже спекулятивные оптимизации
- А если что то пошло не так то всегда можно откатить к оригинальному байт код.

# Два вида оптимизаций

- У джавы есть два вида компайлера:
  - javac bytecode compiler
  - HotSpot VM JIT compiler
- Оба имплементируют похожие оптимизации
- Но bytecode compiler ограничен
  - Dynamic linking
  - Может создавать только статические оптимизации

Не засоряйте код не нужными проверками

БРОСАЙТЕ МУСОР



ПРЯМО

ЗДЕСЬ!



ТУТ ВСЁ РАВНО  
ГРЯЗНО

Акция проходит при поддержке значительной части населения страны.  
Партнеры: Общество с ограниченной моральной ответственностью  
«После нас - хоть потоп», Творческое объединение тяжело больных  
«До урны не донесу», Ассоциация «Брехня, пластик за месяц согниет».

**ЗАСРЁМ** ГОРОД  
ВМЕСТЕ  
ОБЩЕСТВЕННАЯ ОРГАНИЗАЦИЯ

АГИТАЦИОННЫЙ ПЛАКАТ СПЕЦИАЛЬНО ДЛЯ ТЕХ, КТО ДУМАЕТ, ЧТО ЕСЛИ НЕ СОРИТЬ, МИЛЛИОНЫ ДВОРНИКОВ ОСТАНУТСЯ БЕЗ РАБОТЫ

# Элеминация налчеков

- Джава - null-safe
- Ссылки не могут указывать в никуда
- И все нужные проверки добавятся сами
- Не надо писать что то типа
- If(яФрикКонтроль==null) throw NullPointerException();

# Bounds Check Elimination

- Джава гарантирует проверки границ массивов, и при обращении к несуществующему элементу кинется эксепшон

# Example – Bounds Check Elimination

```
1 public class GameTest {
2     @Test
3     public void testScore() {
4         Game game = new Game();
5         GameMove[] gameMoves = {new GameMove("bob", 2), new GameMove("robert", 3)};
6         int length = gameMoves.length;
7         for (int i = 0; i < length; i++) {
8             if (i < 0 || length <= i) throw new ArrayIndexOutOfBoundsException();
9             game.score(gameMoves[i].getPlayer(), gameMoves[i].getPoints());
10        }
11    }
12 }
```

# Развертывание циклов

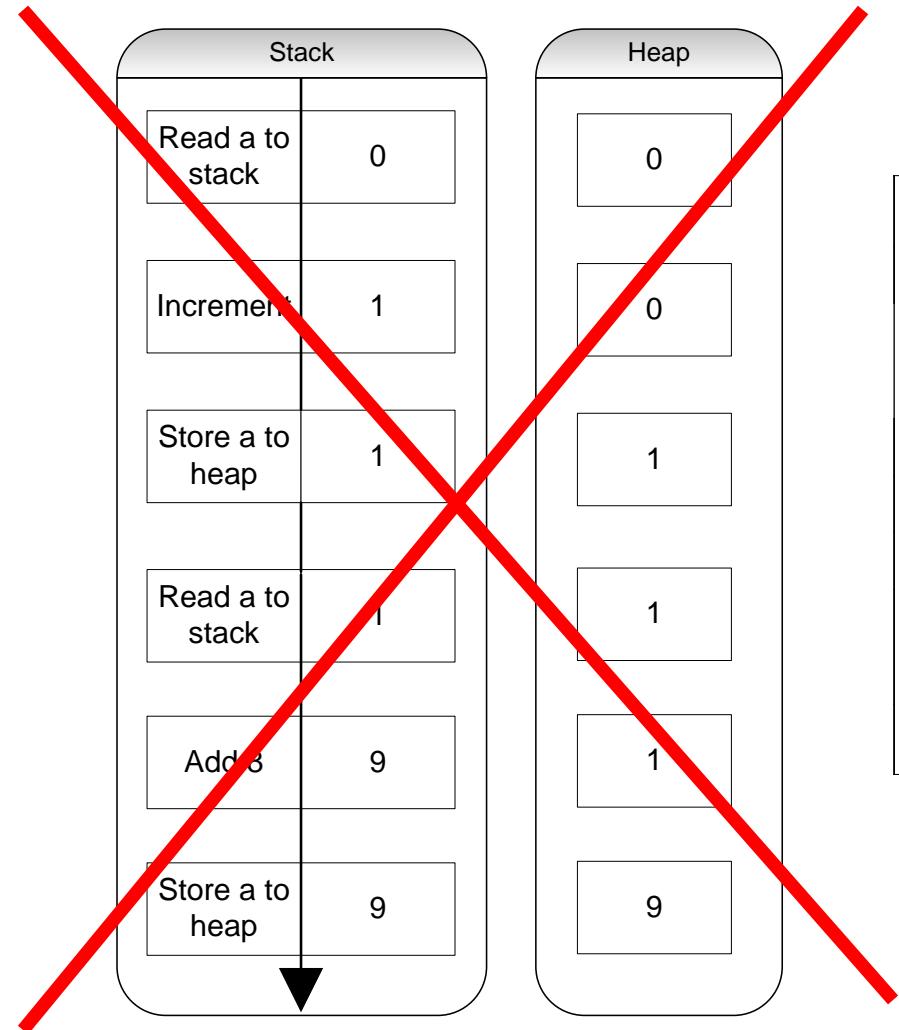
```
1 public class GameTest {  
2     @Test  
3     public void testScore() {  
4         Game game = new Game();  
5         GameMove[] gameMoves = {new GameMove("bob", 2), new GameMove("robert", 3)};  
6         for (int i = 0; i < gameMoves.length; i++) {  
7             GameMove move = gameMoves[i];  
8             game.score(move.getPlayer(), move.getPoints());  
9         }  
10    }  
11 }
```

```
1 public class GameTest {  
2     @Test  
3     public void testScore() {  
4         Game game = new Game();  
5         GameMove[] gameMoves = {new GameMove("bob", 2), new GameMove("robert", 3)};  
6         GameMove move = gameMoves[0];  
7         game.score(move.getPlayer(), move.getPoints());  
8         move = gameMoves[1];  
9         game.score(move.getPlayer(), move.getPoints());  
10    }  
11 }
```

# OSR - On Stack Replacement

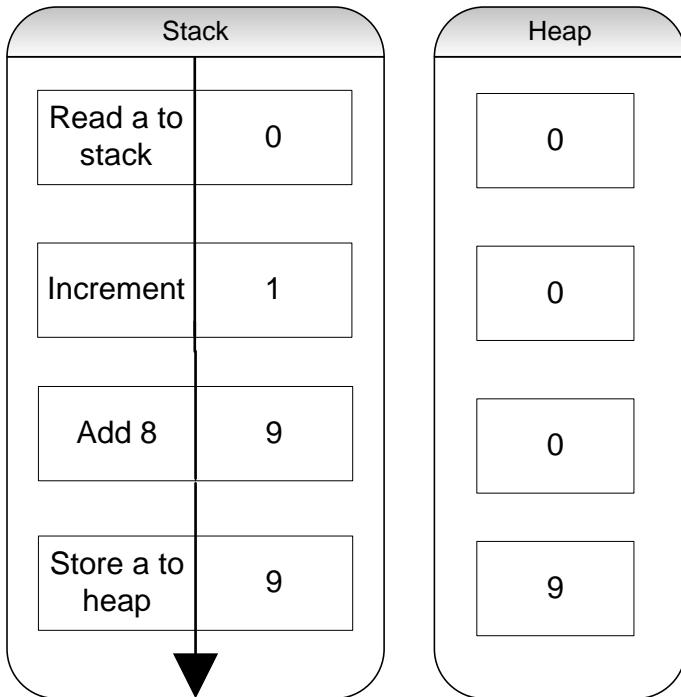
- Обычно код переключается на нативную имплементацию перед входом в метод
- А что если у нас бесконечный цикл?
- OSR - позволяет заменить этот цикл на нативную имплементацию на лету
  - По середине работы метода
- Но оптимизаций будет меньше

# Out-Of-Order Execution



```
1 public class OutOfOrder {  
2  
3     private int a;  
4  
5     public void foo() {  
6         a++;  
7         a+=8;  
8     }  
9  
10 }
```

# Out-Of-Order Execution



```
1 public class OutOfOrder {  
2  
3     private int a;  
4  
5     public void foo() {  
6         a++;  
7         a+=8;  
8     }  
9  
10 }
```

# Inlining

- Мы же любим инкапсуляцию, да? Сетторы, гетторы, всё такое, да?
- Мы же хотим короткие методы?
- Мы любим всякие тулы, которые делают static code analys?
  - А для этого опять таки нужны короткие методы
- И мы не хотим за всё это платить перформенсом!
- Вот для этого существует оптимизация `Inline`!

# Inlining

- До оптимизации

```
Point point = new Point(3, 4);  
PointService pointService = new PointService();  
pointService.printPoint(point);
```

- 2 лишних объекта будут созданы на куче

# Inlining

- После оптимизации

```
Point point = new Point(3, 4);  
System.out.println(point.getX() + " " + point.getY());
```



Но ведь этим  
объектом, которые  
зайнлинили, могут  
пользоваться в  
другом месте...



Для этого есть  
Escape Analysis



# Escape Analysis

- Это не оптимизация
- Это проверка не покидает ли объект local scope
  - Например: не возвращает ли приватный метод объект, ссылку на объект созданный внутри него.
- Это открывает огромное количество возможностей для других оптимизаций

# Пример - Lock Elision

```
1 public class GameTest {
2     @Test
3     public void testScore() {
4         Game game = new Game();
5         lock(game);
6         game.logger.info("Bob" + " scores " + 5);
7         game.totalScore += 5;
8         game.logger.info("Jane" + " scores " + 7);
9         game.totalScore += 7;
10        game.logger.info("Dwane" + " scores " + 1);
11        game.totalScore += 1;
12        unlock(game);
13    }
14 }
```

# Inlining

- Но ведь все паблик не финал методы в джаве виртуальны!
  - Dynamic binding
- И то, что заинлайнилось сейчас, при другой инвокации может потребовать иную имплементацию!



# Inlining

- А вот в таких случаях HotSpot отменяет Inlining
- Это так называемый спекулятивный инлайнинг
- По умолчанию инлайнинг ограничен до 35 байтов байткода
  - Но можно поменять -XX:MaxInlineSize=#



# Scalar Replacement

```
Point point = new Point(3, 4);  
System.out.println(point.getX() + " " + point.getY());  
  
int x = 3;  
int y = 4;  
System.out.println(x + " " + y);
```

# Constants Folding

- Literals folding
  - Before: `int foo = 9*10;`
  - After: `int foo = 90;`
- String folding or StringBuilder-ing
  - Before: `String foo = "hi Joe " + (9*10);`
  - After: `String foo = new StringBuilder().append("hi Joe ").append(9 * 10).toString();`
  - After: `String foo = "hi Joe 90";`

# Constants Folding

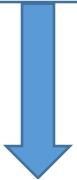
```
int x = 3;  
int y = 4;  
System.out.println(x+" "+y);
```



```
System.out.println("3 4");
```

# Итак в итоге

```
public class PointService {  
    public void printPoint(Point p){  
        System.out.println(p.getX()+" "+p.getY());  
    }  
}  
  
Point point = new Point(3, 4);  
PointService pointService = new PointService();  
pointService.printPoint(point);
```



```
System.out.println("3 4");
```

Оптимизации это круто, а что же остается нам?

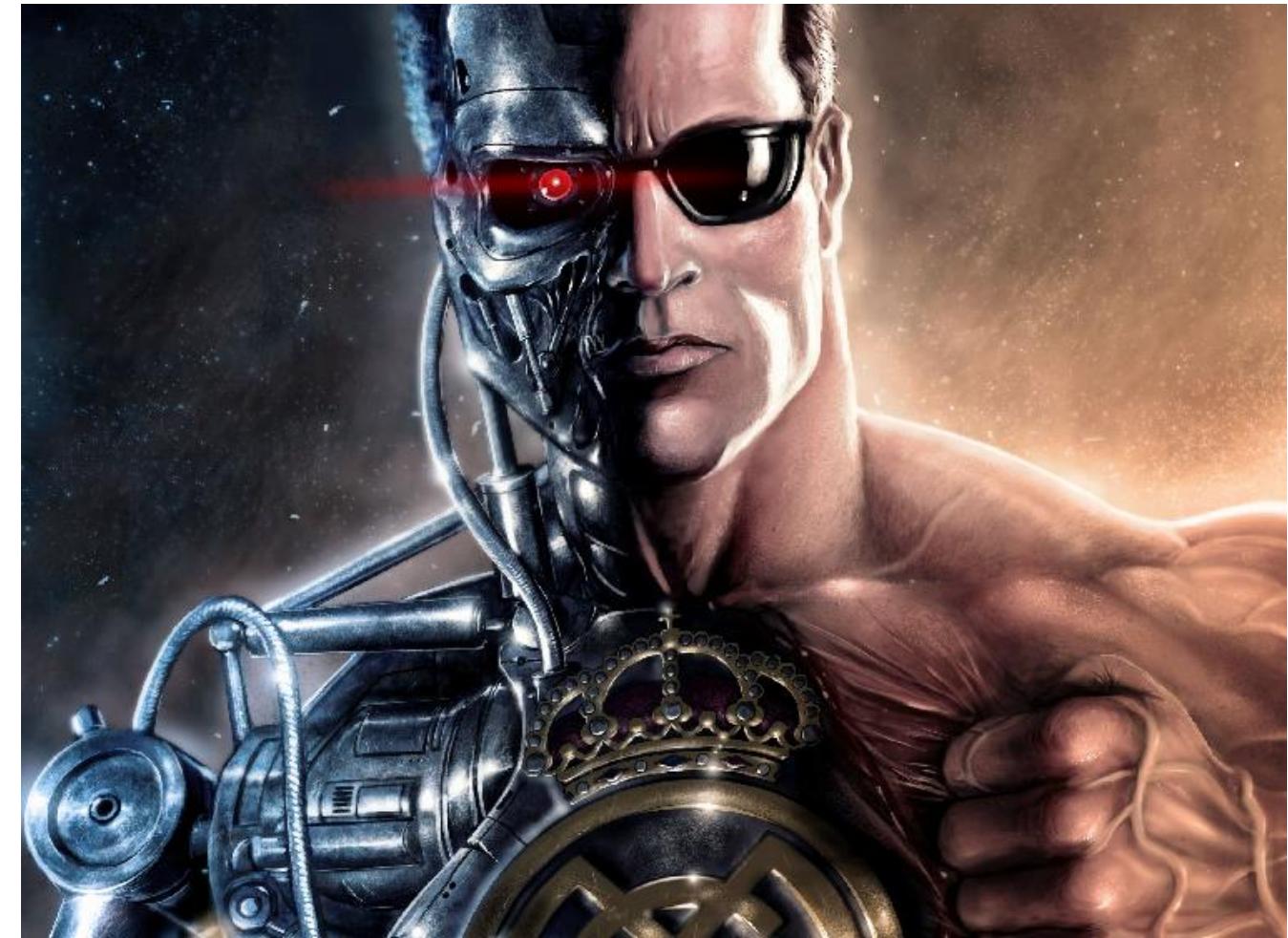


# Мы должны

- Писать качественный объектно-ориентированный код
  - Polymorphism
  - Abstraction
  - Encapsulation
  - DRY
  - KISS
  - Короткие методы
- Не мешать существующим оптимизациям

# Итак длинные методы это плохо, что делаем?

- Делаем рефактор
- Только не смешивайте  
разные уровни абстракции
- Сейчас приведу пример



# Метод: женщинаСобираетсяВГости

- Раздеться
- Зайти в душ
- Помыться
- Выйти из душа
- Одеться
- Открыть помаду
- Провести 10 раз по губам
- Закрыть помаду
- Открыть туш для глаз
- Покрасить левый глаз
- Покрасить правый глаз
- Посмотреть в зеркало
- Смыть туш
- Выбрать другой цвет
- ...

Метод: женщинаСобираетсяВГости

Как это должно быть

- Помыться
- Одеться
- Накраситься
- ...

# SOLID

- Single Responsibility Principle
- Open/Close Principle
- Liskov's Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

# Single Responsibility

- Класс должен иметь только одну ответственность
- Не должна быть более чем одна причина которая приведет к изменением в классе

# Хреновый класс

```
Class Customer {  
    + getGivenName  
    + getFamilyName  
    + getEmail  
    + writeToDB  
    + writeToFile  
}
```

Данные кастомера, могут находиться в этом классе

Функциональность какого-нибудь сервиса, обслуживающего кастомеров

А вот это получше

```
Class Customer {  
    + getGivenName  
    + getFamilyName  
    + getEmail  
}
```

```
Class CustomerService {  
    + writeCustomerToDb(Customer c)  
    + writeCustomerToFile(Customer c)  
}
```

А вот так ещё лучше

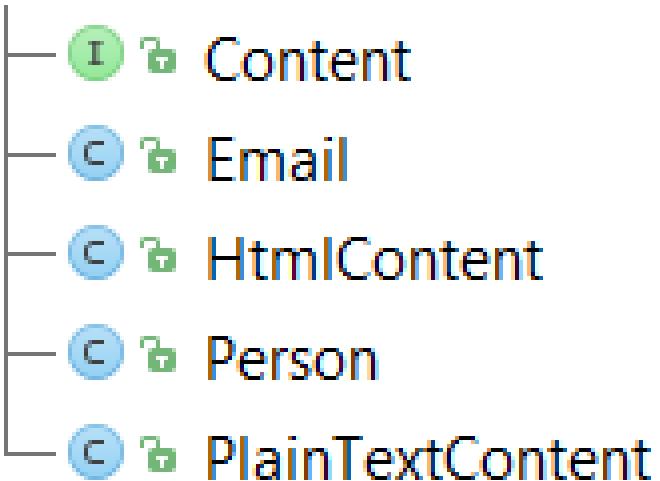
```
Class Customer {          Class CustomerDBService {  
    + getGivenName        + persistCustomer(Customer c)  
    + getFamilyName       }  
    + getEmail  
}  
  
Class CustomerFileService {  
    + saveCustomer(Customer c)  
}
```

# Задание – улучшить имеющийся модуль

```
* / public class Email {  
    private String sender;  
    private String receiver;  
    private String content;  
  
    public String getSender() { return sender; }  
    public void setSender(String sender) { this.sender = sender; }  
    public String getReceiver() { return receiver; }  
    public void setReceiver(String receiver) { this.receiver = receiver; }  
    public String getContent() { return content; }  
    public void setContent(String content) { this.content = content; }  
}
```

# БОТ КАК-ТО ТАК

```
public class Email {  
    private Person sender;  
    private Person receiver;  
    private Content content;  
    public Person getSender() { return sender; }  
    public void setSender(Person sender) { this.sender = sender; }  
    public Person getReceiver() { return receiver; }  
    public void setReceiver(Person receiver) { this.receiver = receiver; }  
    public Content getContent() { return content; }  
    public void setContent(Content content) { this.content = content; }  
}
```



# Open Closed Principle

- Каждый компонент должен быть открыт для расширения но закрыт для изменений

| Open  | Closed   |
|---|--|
| Методы открытые для наследования  | Приватные поля   |
| Методы принимающие интерфэйсы, допускающие разную имплементацию и делегирование этим имплементациям | Избегайте изменений в уже написанной бизнес логике, при расширении системы |

# Попробуйте улучшить...

```
public class Panel{  
  
    private Graphics g;  
  
    public Panel(Graphics g) {  
        this.g = g;  
    }  
    public void drawPoint(Point p){  
        // draw point using by using g  
    }  
    public void drawCircle(Circle circle){  
        // draw circle using by using g  
    }  
}
```

# ВОТ КАК ТО ТАК...

```
public class Panel{  
  
    private Graphics g;  
  
    public Panel(Graphics g) {  
        this.g = g;  
    }  
    public void drawShape(Shape shape) {  
        shape.draw(g);  
    }  
}
```

# Принцип подстановки Барбары Лисков

- *Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа не зная об этом.*
- если  $S$  является подтипом  $T$ , тогда объекты типа  $T$  в программе могут быть замещены объектами типа  $S$  без каких-либо изменений желательных свойств этой программы



```
public interface PersistedResource {  
    void load();  
    void persist();  
}  
  
public class ApplicationSettings implements PersistedResource {  
    @Override  
    public void load() {  
        //some logic here  
    }  
  
    @Override  
    public void persist() {  
        //some logic here  
    }  
}  
  
public class UserSettings implements PersistedResource {  
    @Override  
    public void load() {  
        //some logic here  
    }  
  
    @Override  
    public void persist() {  
        //some logic here  
    }  
}
```

```
public class SettingsServiceImpl implements SettingsService {
    @Override
    public List<PersistedResource> loadAll() {
        ArrayList<PersistedResource> resources = new ArrayList<PersistedResource>();
        //some logic here
        return resources;
    }

    @Override
    public void saveAll(List<PersistedResource> resources) {
        for (PersistedResource resource : resources) {
            resource.persist();
        }
    }
}

public class Main {
    public static void main(String[] args) {
        SettingsServiceImpl service = new SettingsServiceImpl();
        List<PersistedResource> resources = service.loadAll();
        //some operations with resources
        service.saveAll(resources);
    }
}
```

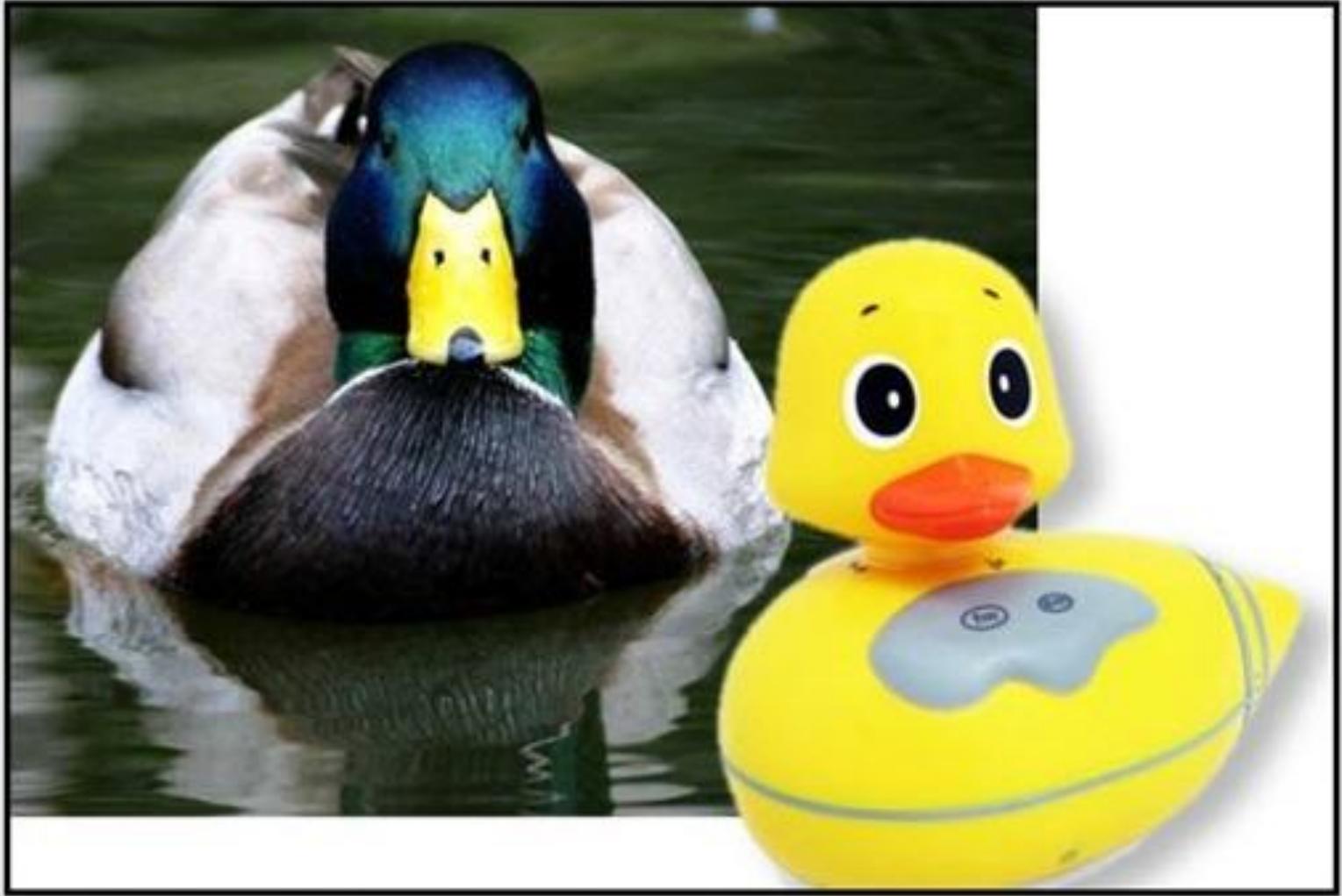
И поначалу все  
будет хорошо...



```
public class SpecialResources implements SettingsService {  
    @Override  
    public List<PersistedResource> loadAll() {  
        ArrayList<PersistedResource> resources = new ArrayList<PersistedResource>();  
        //some logic here  
        return resources;  
    }  
  
    @Override  
    public void saveAll(List<PersistedResource> resources) {  
        throw new UnsupportedOperationException();  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        SettingsServiceImpl service = new SettingsServiceImpl();  
        List<PersistedResource> resources = service.loadAll();  
        //some operations with resources  
        service.saveAll(resources);  
    }  
}
```

 **Runtime Exception!**  
**Зашибись!**



## LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You  
Probably Have The Wrong Abstraction

# Принцип разделения интерфейса

- Принцип разделения интерфейсов говорит о том, что слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе.
- Следование этому принципу помогает системе оставаться гибкой при внесении изменений в логику работы и пригодной для рефакторинга

# Вот пример нарушения этого принципа

```
public class MyMouseListener implements MouseListener {  
    public void mouseClicked(MouseEvent e) {  
        //To change body of implemented methods use File  
    }  
  
    public void mousePressed(MouseEvent e) {  
        //To change body of implemented methods use File  
    }  
  
    public void mouseReleased(MouseEvent e) {  
        //To change body of implemented methods use File  
    }  
  
    public void mouseEntered(MouseEvent e) {  
        //To change body of implemented methods use File  
    }  
  
    public void mouseExited(MouseEvent e) {  
        //To change body of implemented methods use File  
    }  
}
```

```
public class MyMouseListener implements MouseListener {  
    public void mouseClicked(MouseEvent e) {  
        //some logic  
    }  
  
    public void mousePressed(MouseEvent e) {  
    }  
  
    public void mouseReleased(MouseEvent e) {  
    }  
  
    public void mouseEntered(MouseEvent e) {  
    }  
  
    public void mouseExited(MouseEvent e) {  
    }  
}
```

```
public interface Resource {  
    void load();  
    void persist();  
}
```

```
public interface  
LoadableResource {  
    void load();  
}
```

```
public interface  
PersistableResource {  
    void persist();  
}
```

```
public class ApplicationSettings implements PersistableResource , LoadableResource {  
    @Override  
    public void load() {  
        //some logic here  
    }  
    @Override  
    public void persist() {  
        //some logic here  
    }  
}
```

```
public class SpecialSettings implements LoadableResource {  
    @Override  
    public void load() {  
        //some logic here  
    }  
}
```

```
public class SettingsServiceImpl implements SettingsService {  
    @Override  
    public List<LoadableResource> loadAll() {  
        ArrayList<LoadableResource> resources = new ArrayList<LoadableResource>();  
        //some logic here  
        return resources;  
    }  
    @Override  
    public void saveAll(List<PersistableResource> resources) {  
        for (PersistableResource resource : resources) {  
            resource.persist();  
        }  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        SettingsServiceImpl service = new SettingsServiceImpl();  
        List<LoadableResource> resources = service.loadAll();  
        //some operations with resources  
        service.saveAll(resources);  
    }  
}
```

# Принцип инверсии зависимостей

- Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
- Важный принцип, используемый для уменьшения связанности
- Чтобы снизить связанность используется впрыскивание зависимостей



# А если поменяли базу на MongoDB?

```
public class OracleDao {  
    public void savePerson(Person person) {  
        //some logic here  
    }  
}  
  
public class DBService {  
    private OracleDao dao;  
  
    public DBService(OracleDao dao) {  
        this.dao = dao;  
    }  
  
    public void doWork() {  
        Person person = readPerson();  
        dao.savePerson(person);  
    }  
  
    private Person readPerson() {...}  
}
```

```
public interface Dao {  
    void savePerson(Person person);  
}
```



```
public class OracleDao implements Dao {  
    @Override  
    public void savePerson(Person person) {  
        //some logic here  
    }  
}
```

```
public class DBService {  
    private Dao dao;  
  
    public DBService(Dao dao) {  
        this.dao = dao;  
    }  
  
    public void doWork() {  
        Person person = readPerson();  
        dao.savePerson(person);  
    }  
  
    private Person readPerson() {...}  
}
```

# Зачем нам Design Patterns?

- Не нужно изобретать велосипед



- Но и не нужно применять их через силу

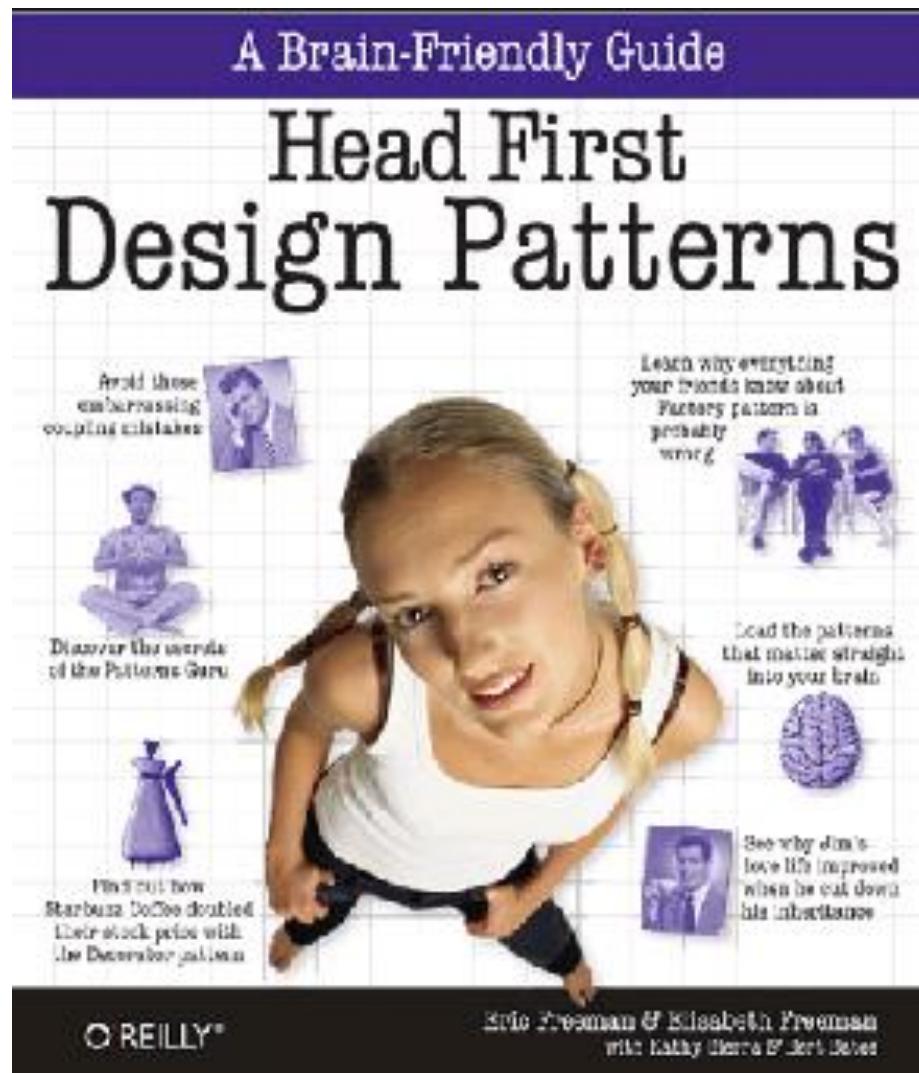


# Design patterns

- 1970 год - Christopher Alexander



# Наиболее рекомендованная книга



# Дизайн патерн по написанию дизайн патернов

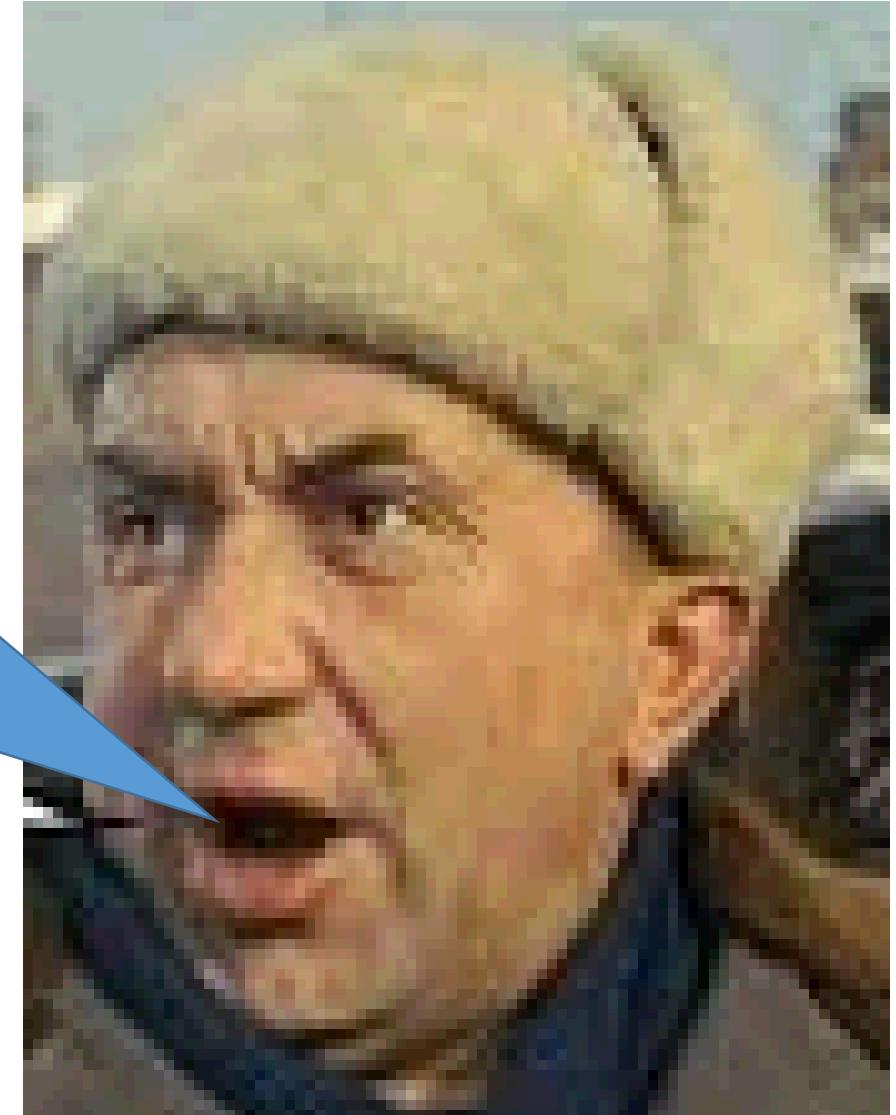
- **Pattern Name and Classification:** A descriptive and unique name that helps in identifying and referring to the pattern.
- **Intent:** A description of the goal behind the pattern and the reason for using it.
- **Also Known As:** Other names for the pattern.
- **Motivation (Forces):** A scenario consisting of a problem and a context in which this pattern can be used.
- **Applicability:** Situations in which this pattern is usable; the context for the pattern.
- **Structure:** A graphical representation of the pattern. Class diagrams and Interaction diagrams may be used for this purpose.
- **Participants:** A listing of the classes and objects used in the pattern and their roles in the design.
- **Collaboration:** A description of how classes and objects used in the pattern interact with each other.
- **Consequences:** A description of the results, side effects, and trade offs caused by using the pattern.
- **Implementation:** A description of an implementation of the pattern; the solution part of the pattern.
- **Sample Code:** An illustration of how the pattern can be used in a programming language.
- **Known Uses:** Examples of real usages of the pattern.
- **Related Patterns:** Other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.

Поехали

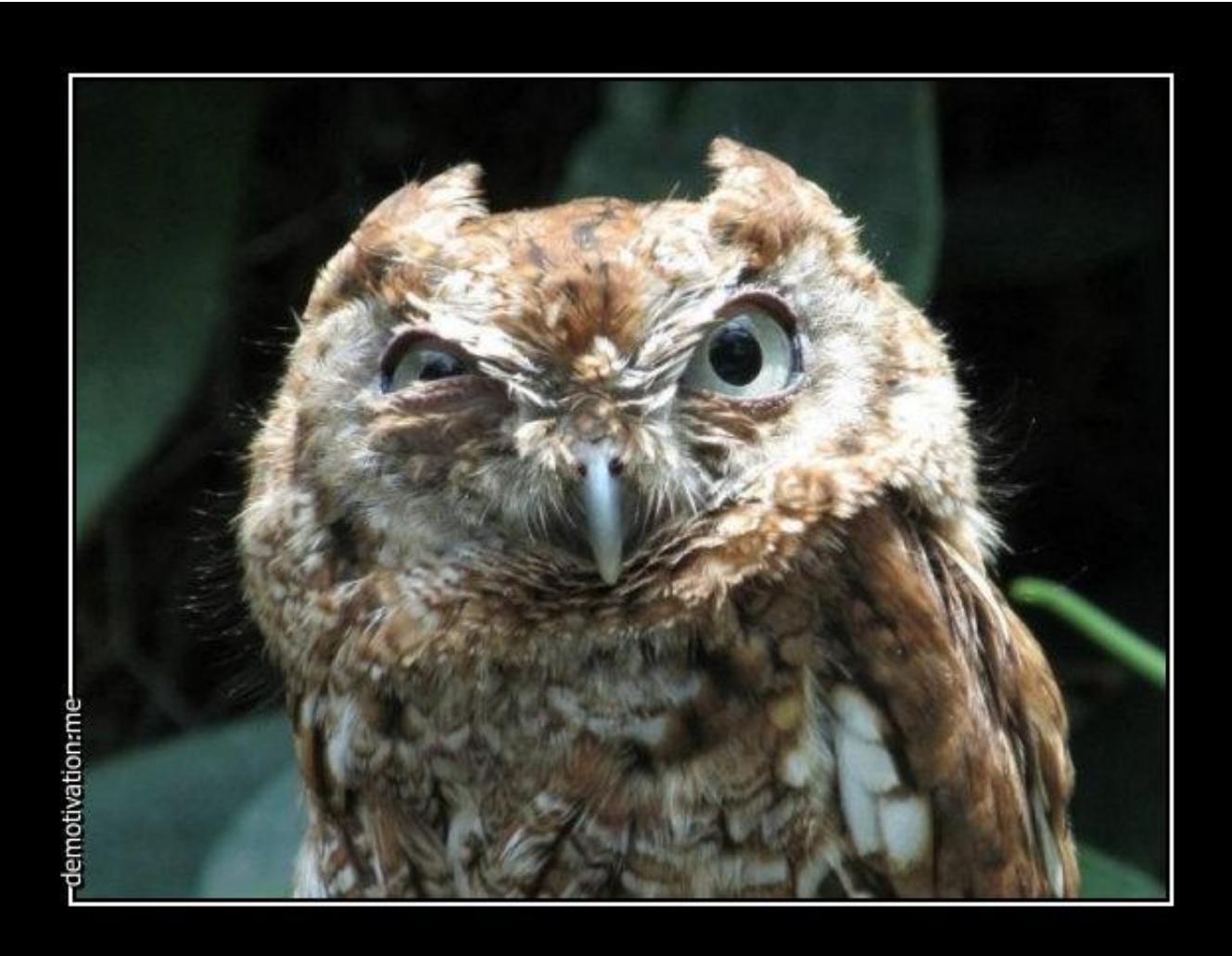
# Задание

- Напишите класс РадиоБудильник.
- У него есть вся функциональность Радио (setChannel, setVolume...)
- И вся функциональность Будильника (setAlarmTime, stopAlarm...)

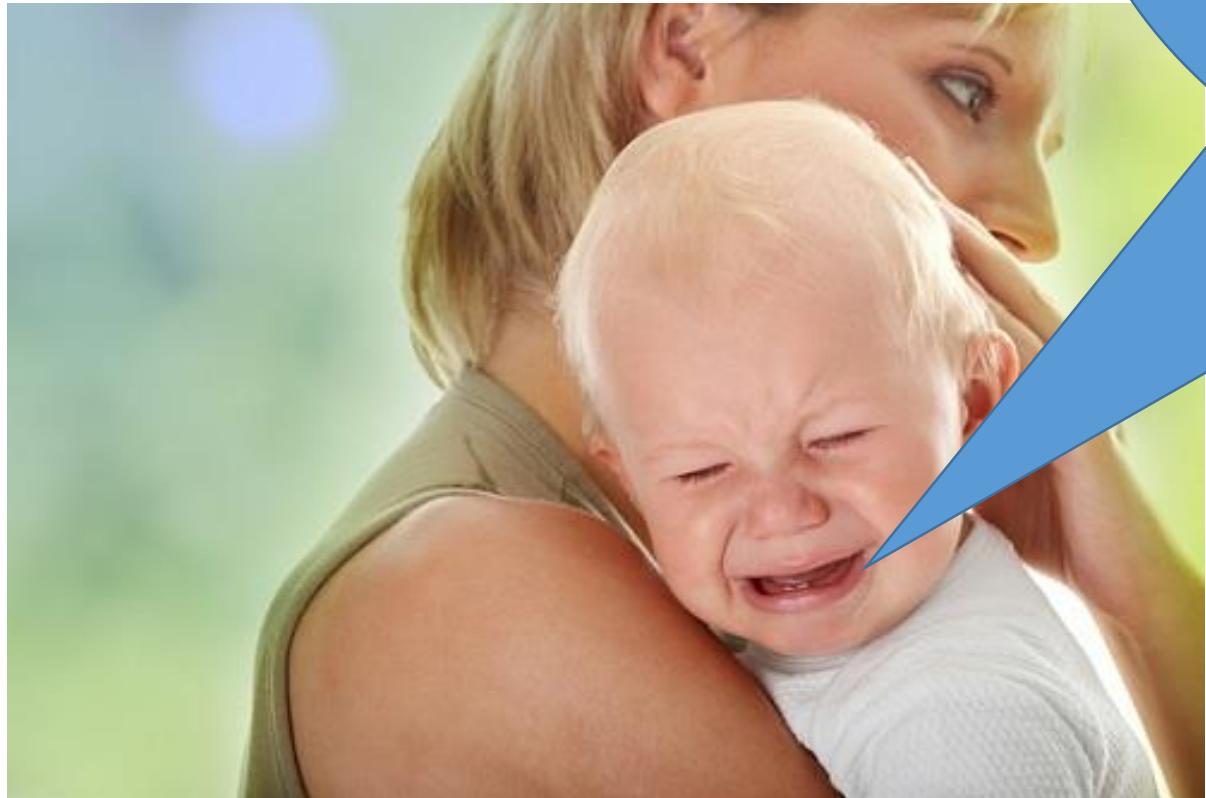
Так ведь в джаве  
нет  
множественного  
наследования!



# А в Java 8 точно нету?



Я всегда  
говорил, что  
наследование  
это плохо, что  
же делать?...



Пользуйся  
композицией  
малыш!



# Strategy



# Strategy – Проблема

- Иногда части наших объектов должны часто меняться
- И наследование не может быть решением, потому что оно очень ограничивает.
- Более того, изменения могут даже привести к нарушению принципов наследования
- Как можно поменять только часть объекта не трогая всё остальное?

# Strategy – Пример проблемы

- Разработка арпиджишной игры:
  - У интерфейса Character есть метод fight()
    - Который по разному имплементируют классы Knight, Wizard и Troll
  - Добавляется класс King
    - Алгоритм его метода fight() идентичен методу fight() у рыцаря
    - Что будем делать?
  - Добавляется класс Princess
    - А она вообще не умеет драться
    - Что делаем?

# Strategy – Решение

- Определяем интерфейс для меняющегося поведения
- Делаем ему разные имплементации
- И делегируем
- Смотрим...

# Strategy – Пример решения

```
1 public interface Character {  
2  
3     void fight();  
4 }
```

```
1 public interface FightBehavior {  
2  
3     void fight();  
4 }
```

```
1 public class SwordFightBehavior implements FightBehavior {  
2  
3     public void fight() {  
4         ...  
5     }  
6 }
```

```
1 public class NoFightBehavior implements FightBe  
2  
3     public void fight() {  
4  
5     }  
6 }
```

```
1 public class Knight implements Character {  
2     private FightBehavior swordFightBehavior;  
3  
4     public Knight() {  
5         swordFightBehavior = new SwordFightBehavior();  
6     }  
7  
8     public void fight() {  
9         swordFightBehavior.fight();  
10    }  
11 }  
12 }
```

```
1 public class Princess implements Character {  
2  
3     FightBehavior noFightBehavior;  
4  
5     public Princess() {  
6         noFightBehavior = new NoFightBehavior();  
7     }  
8  
9     public void fight() {  
10        noFightBehavior.fight();  
11    }  
12 }  
13 }
```

# А теперь давайте вы попишите код...

- Помните правила SOLID, композицию и Strategy
- Нужна апликация которая генерирует случайные математические упражнения для учеников первого класса.
- Начнем с примеров исключительно на сложение
- Хорошо пошла... А теперь добавьте вычитание. Причем апликация в случайном порядке решает какой пример сгенерировать, а потом генирит его
- А теперь еще и умножение с делением

# Созидательные дизайн паттерны





THERE CAN BE ONLY ONE

**THERE CAN BE**



**ONLY ONE!**

# Может быть только один...

- Есть много объектов которые нужны нам в одном экземпляре
  - Thread pools
  - Cache
  - Load Balancers
  - И всякие там разные сервисы

А ты можешь написать сингальтон?

- Шесть фаз понимая сингальтона:



Фаза первая:  
«Студент»



## Фаза вторая: «Стажёр»

**А ЧТО С  
МУЛЬТИРЕДИНГОМ?**

```
public class Singleton {  
    private static Singleton singleton;  
  
    private Singleton() {  
    }  
  
    public static Singleton getInstance() {  
        if (singleton==null) {  
            singleton = new Singleton();  
        }  
        return singleton;  
    }  
  
    // all business methods  
}
```

# Фаза третья: «Junior Software Engineer»

**А что с  
перформенсом?**

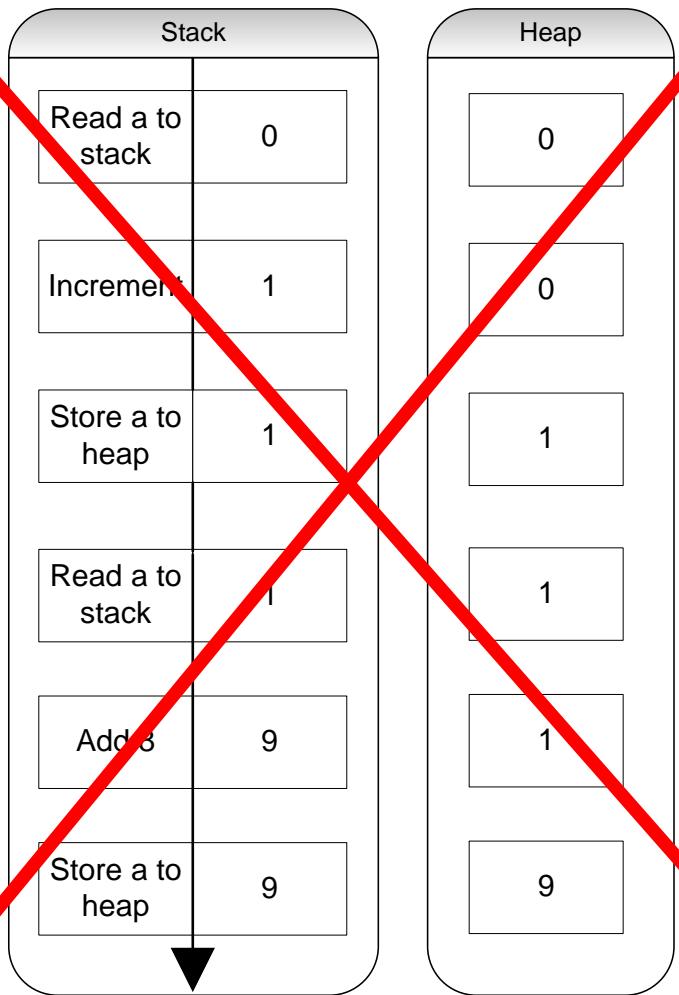
```
public class Singleton {  
    private static Singleton singleton;  
  
    private Singleton() {}  
  
    public synchronized static Singleton getInstance() {  
        if (singleton==null) {  
            singleton = new Singleton();  
        }  
        return singleton;  
    }  
  
    // all business methods  
}
```

# Фаза четвертая: «Senior Software Engineer»

**А что на счёт  
джава  
оптимизаций?**

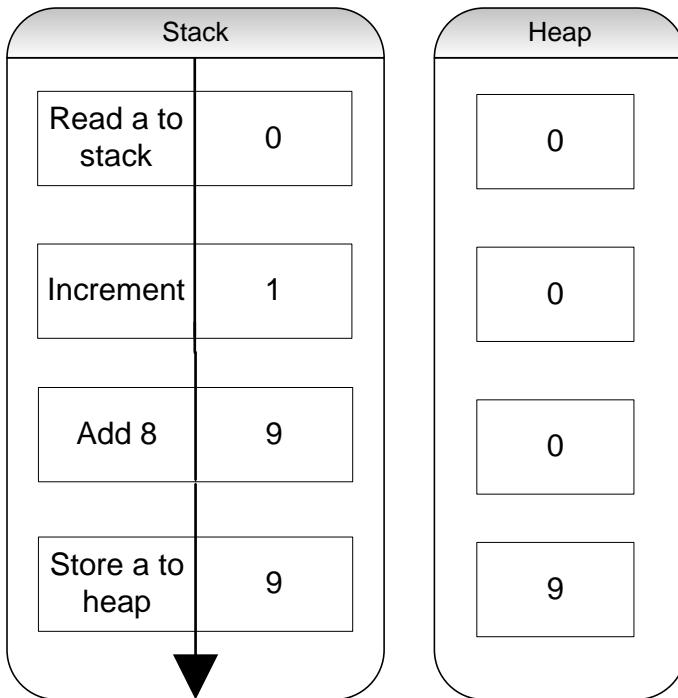
```
public class Singleton {  
    private static Singleton singleton;  
  
    private Singleton() {  
    }  
  
    public static Singleton getInstance() {  
        if (singleton == null) {  
            synchronized (Singleton.class) {  
                if (singleton == null) {  
                    singleton = new Singleton();  
                }  
            }  
        }  
        return singleton;  
    }  
  
    // all business methods  
}
```

# Out-Of-Order Execution



```
1 public class OutOfOrder {  
2  
3     private int a;  
4  
5     public void foo() {  
6         a++;  
7         a+=8;  
8     }  
9  
10 }
```

# Out-Of-Order Execution



```
1 public class OutOfOrder {  
2  
3     private int a;  
4  
5     public void foo() {  
6         a++;  
7         a+=8;  
8     }  
9  
10 }
```

# Фаза четвертая: «Senior Software Engineer»

```
public class Singleton {  
    private static Singleton singleton;  
  
    private Singleton() {  
    }  
  
    public static Singleton getInstance() {  
        if (singleton == null) {  
            synchronized (Singleton.class) {  
                if (singleton == null) {  
                    singleton = new Singleton();  
                }  
            }  
        }  
        return singleton;  
    }  
  
    // all business methods  
}
```

# Фаза пятая: «Lead Software Engineer»

```
public class Singleton {  
    private static volatile Singleton singleton;  
  
    private Singleton() {  
    }  
  
    public static Singleton getInstance() {  
        if (singleton == null) {  
            synchronized (Singleton.class) {  
                if (singleton == null) {  
                    singleton = new Singleton();  
                }  
            }  
        }  
        return singleton;  
    }  
  
    // all business methods  
}
```

А еще можно, сделать не ленивый Singleton



я не ленивый,  
я энергосберегающий

# Синглтон интелиджей

```
public class Singleton {  
    private static Singleton ourInstance = new Singleton();  
  
    public static Singleton getInstance() {  
        return ourInstance;  
    }  
  
    private Singleton() {  
    }  
}
```

# Синглтон через enum

```
public enum Singleton {  
    INSTANCE;  
  
    public void doWork() {  
        System.out.println("singleton is working...");  
    }  
}  
  
public static void main(String[] args) {  
    Singleton.INSTANCE.doWork();  
}
```

# Почему не надо писать синглтон

- Уверенность в завтрашнем дне
- Пишите вашу бизнес логику, а не изобретайте колесо
- А как вы будете тестировать?

# Шестая Фаза «Архитектор»

- Не надо писать сингальтоны, для этого есть спринг.



# Задание

- Напишите класс IRobot с единственным public методом: cleanRoom
- Робот должен сообщать о том, что он начал работать, затем выполнять логику чистки комнаты, а затем сообщать, о завершении работы
- Для начала пусть все сообщения робота выводятся в консоль.

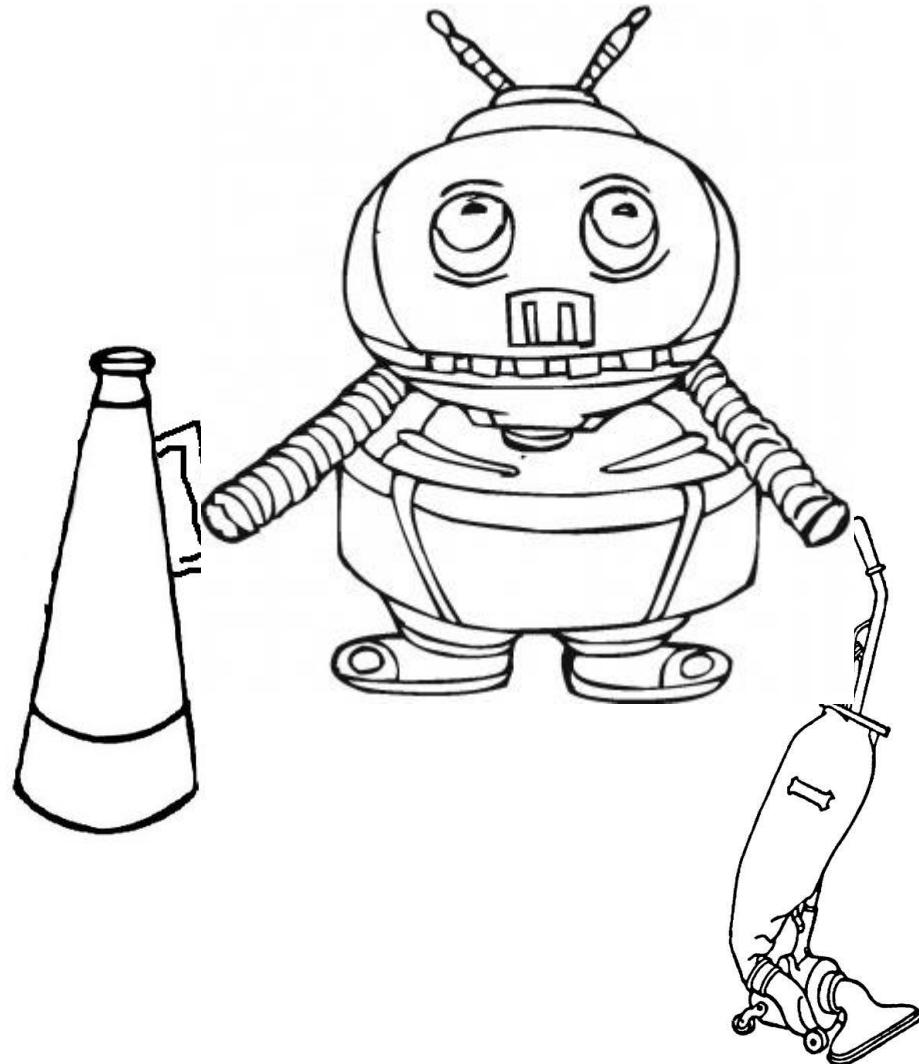


# А как ты создаешь объекты?



- Я пользуюсь: **new**
- Я нереально крут, и использую только `reflections`
- Мне их создают
- Зачем объекты? Есть статические методы!

А чем плохо пользоваться **new?**



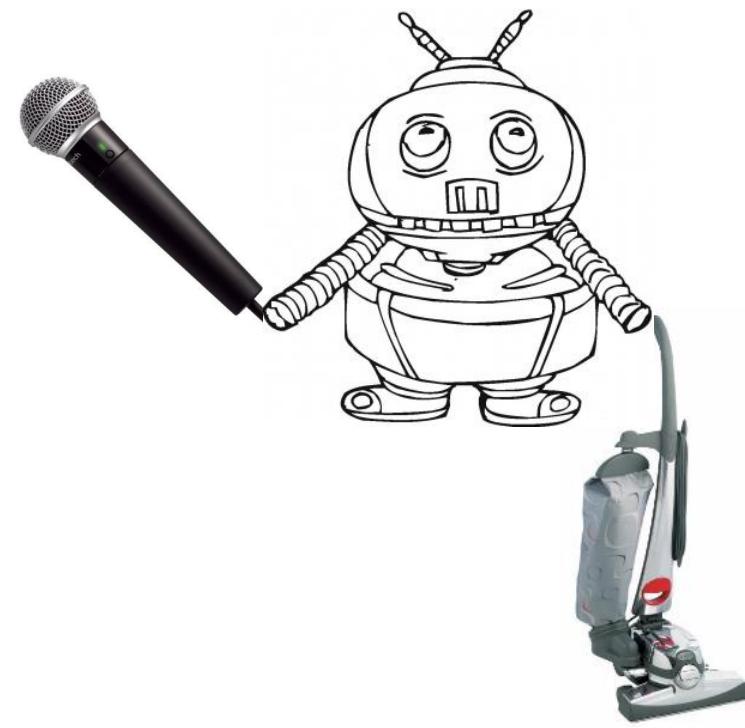
```
public class IRobot {
    private Cleaner cleaner = new Cleaner();
    private Speaker speaker = new Speaker();

    public void cleanRoom() {
        speaker.say("Я начал работать");
        cleaner.clean();
        speaker.say("Я закончил работать");
    }
}
```

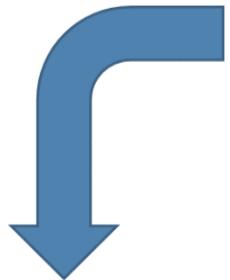
# В чём тут проблема?

```
private Cleaner cleaner = new Cleaner();  
private Speaker speaker = new Speaker();
```

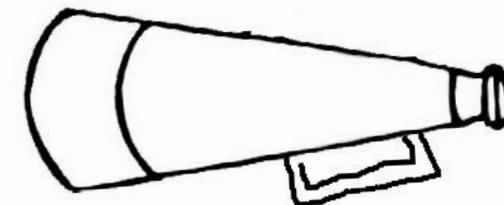
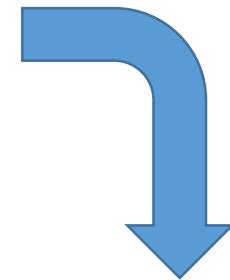
- А если надо поменять имплементацию,  
надо код вскрывать, да?



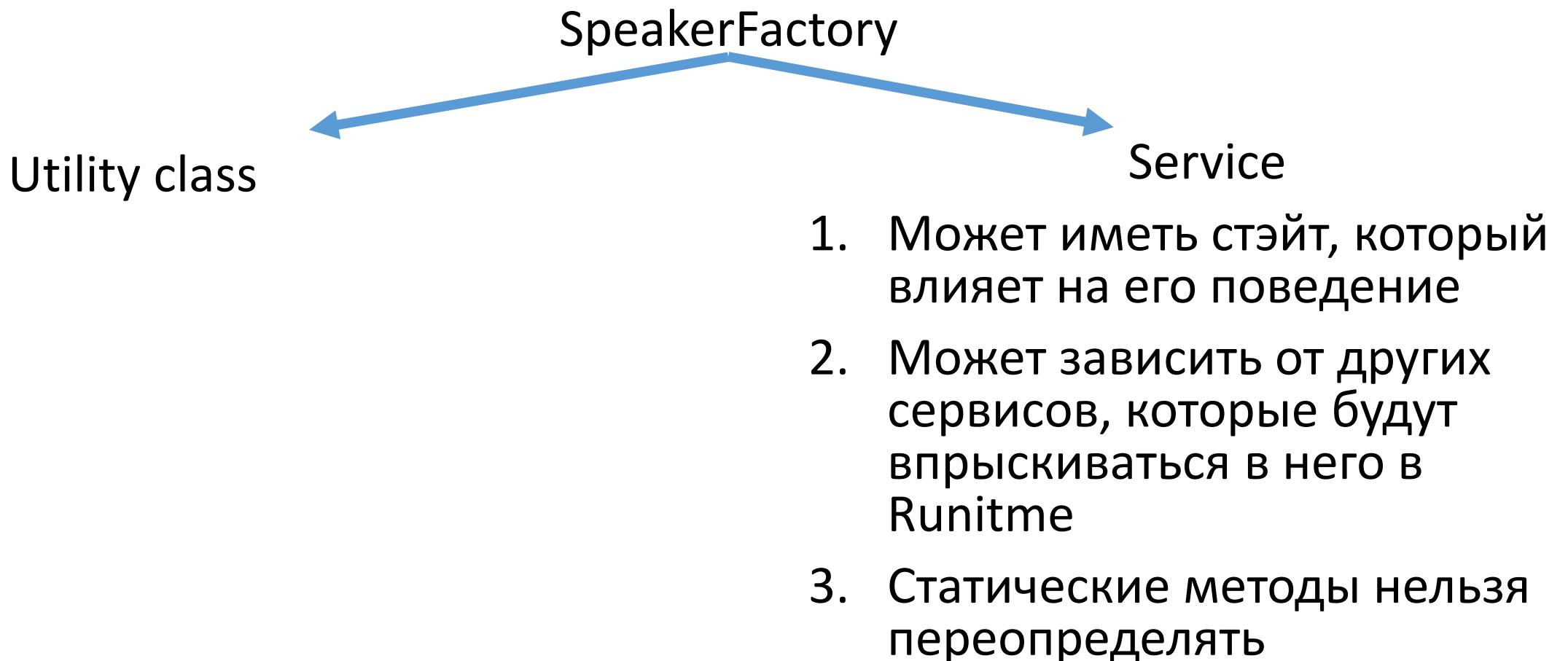
Интерфэйс лучше, правда же?



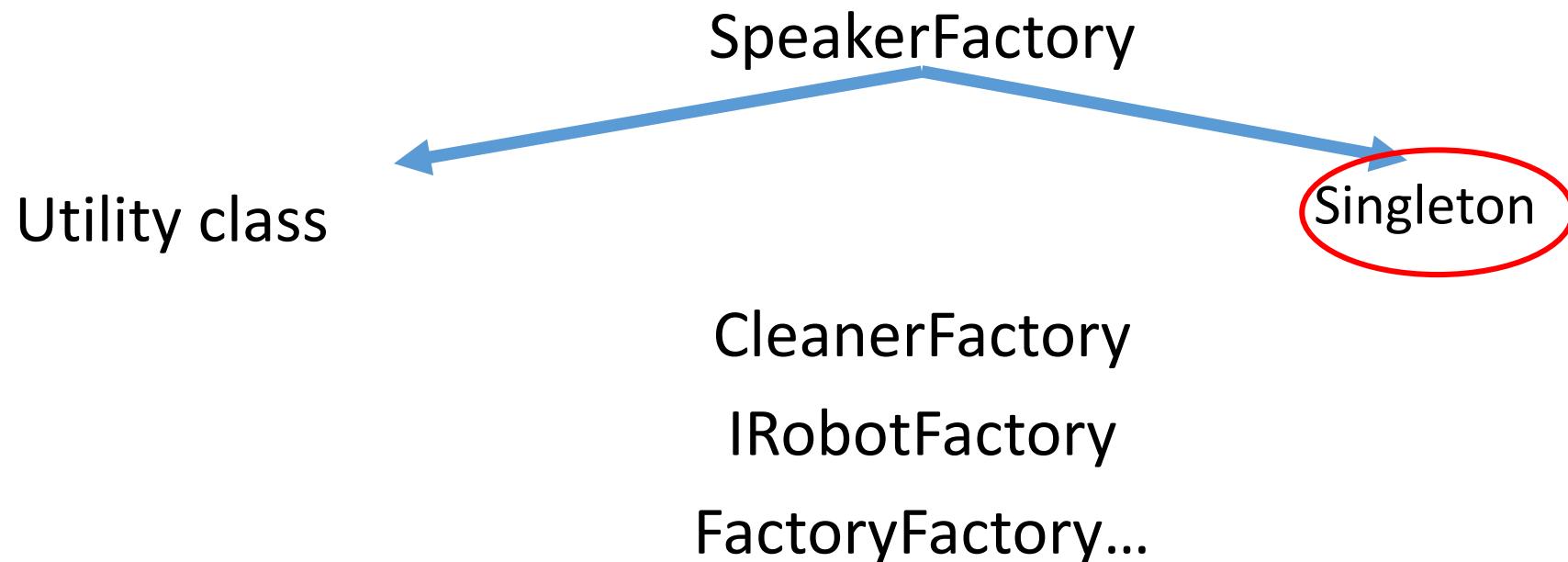
Speaker

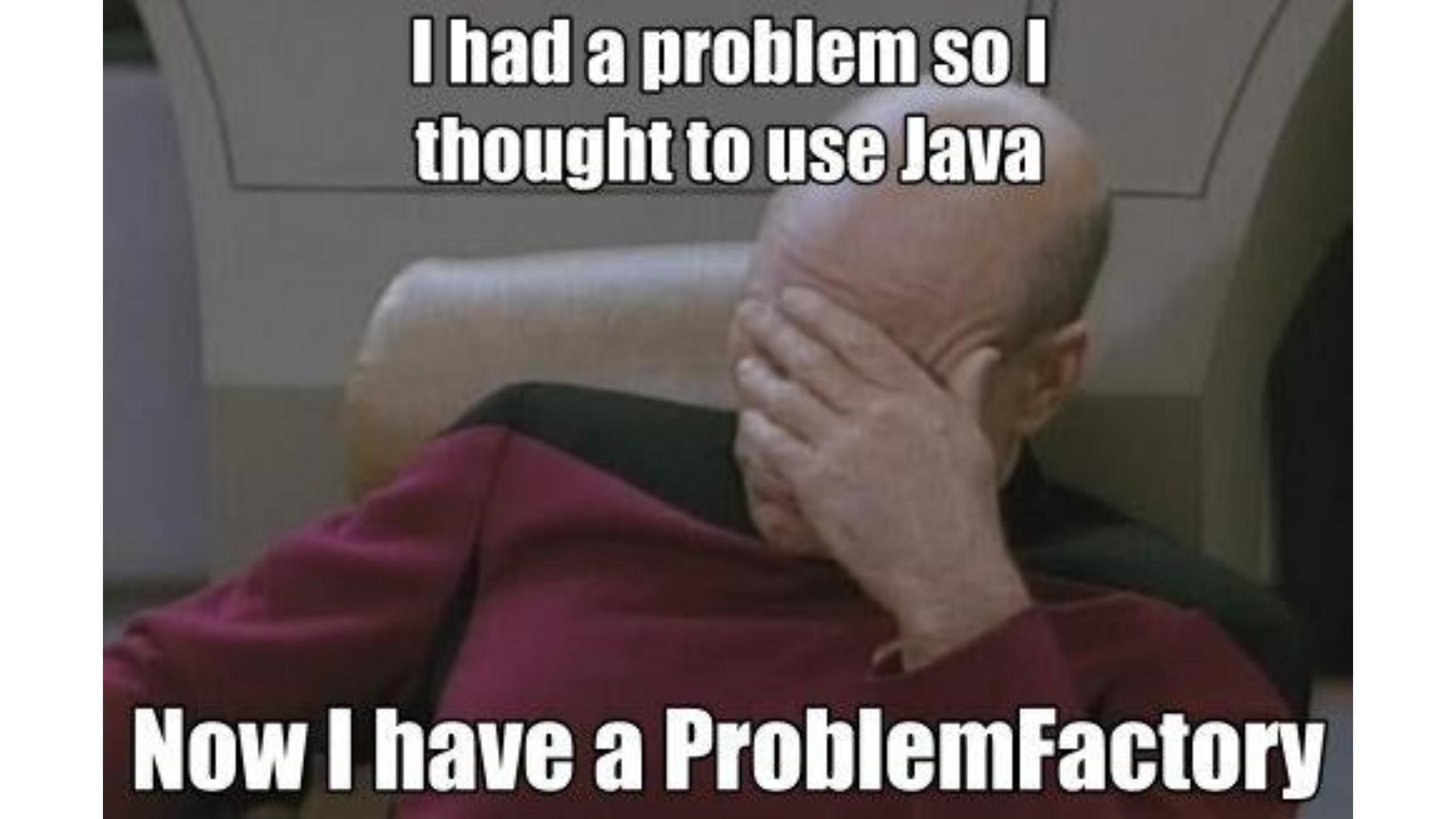


# А кто будет решать, какая имплементация?



А кто будет решать, какая имплементация?



A photograph of a man with light brown hair, wearing a maroon hoodie over a green t-shirt. He is sitting at a light-colored wooden desk, looking down with his hands clasped near his face in a gesture of distress or despair. The background is a plain, light-colored wall.

**I had a problem so I  
thought to use Java**

**Now I have a ProblemFactory**

# А почему собственно не сделать круче?

- `objectFactory.createObject(Speaker.class)`
- Причем Speaker может быть интерфейсом, а вот какая имплементация придет будет решать фактори в зависимости от того, как мы ее запограммируем
- Создание всех объектов будет ее задачей
- Что нам это даёт?
  - Централизованное место для создание всех объектов. (если надо поменять какую-то имплементацию, то менять будем в одном месте)
  - При создании объекта с ним можно делать разные фокусы
  - Например учитывать аннотации, которые есть в классе или интерфейсе

# Теперь давайте обучать наш ObjectFactory

- Напишите свою аннотацию `@InjectRandomInt`
- С параметрами `max` и `min`
- Ставить ее можно только над филдами, и в случае если этот филд интеджер, то в него будет впрыснуто случайное число между `max` и `min`

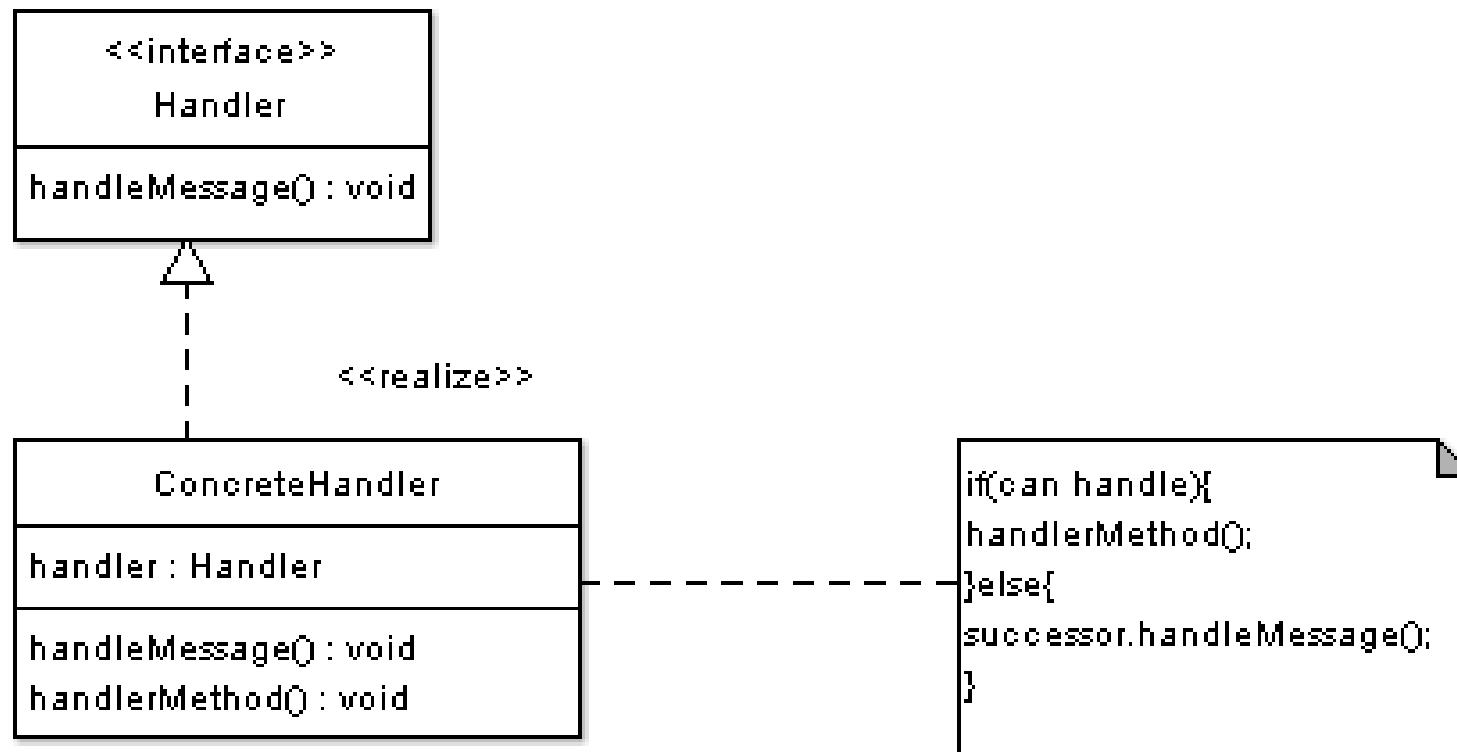
# И еще чуть чуть подкрутим

- Напишите аннотацию @Inject
- ObjectFactory будет искать подходящий тип имплементации и впрыскивать в поле над которым стоит аннотация объект этой имплементации
- Поставьте @Inject над полем speaker у вашего робота и проверьте, что всё работает

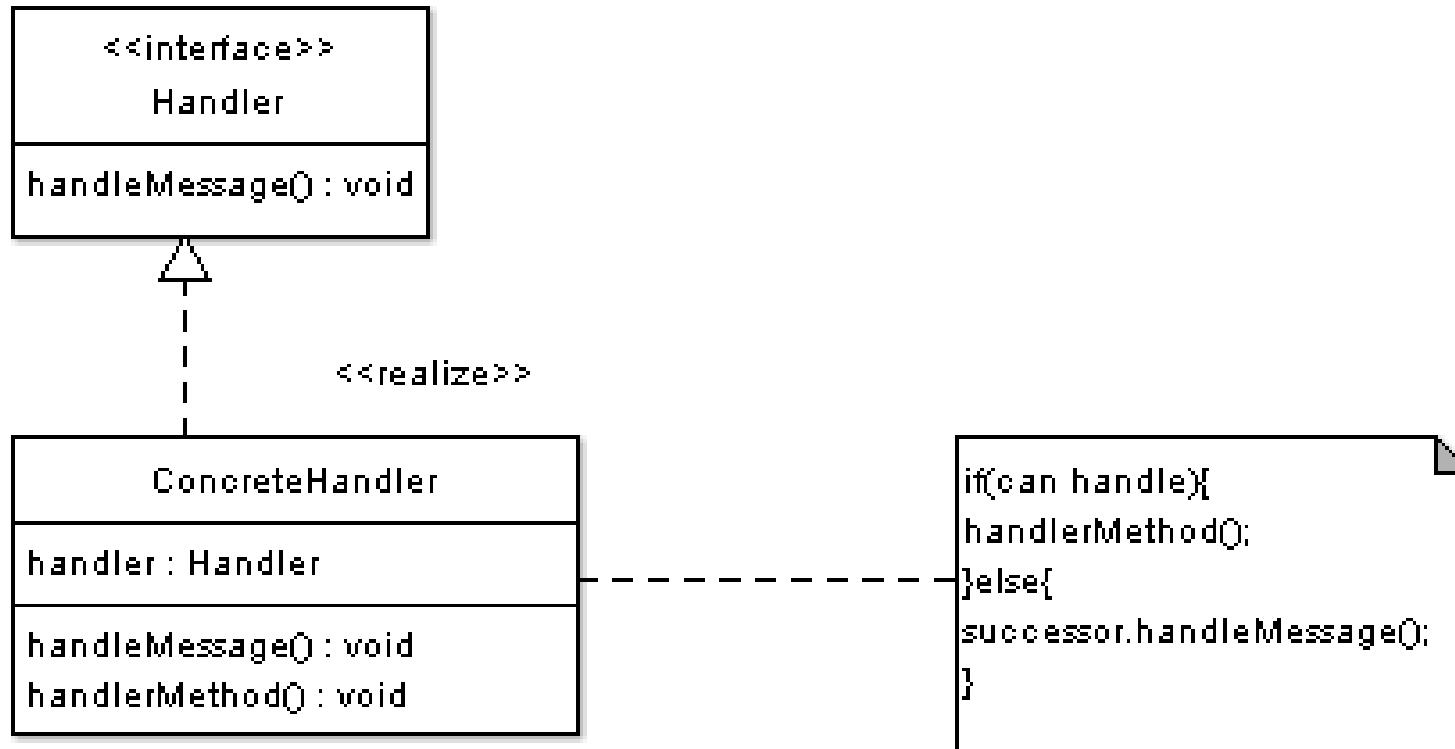
# А что у нас с архитектурой?



# (Ланцюжок відповіальностей)



# Chain-of-responsibility pattern (Ланцюжок відповіальностей)



# Что за хрень с конструктором???

- Попробуйте в конструкторе обратится к переменной в которую ObjectFactory должно впрыскивать значение
- Все переменные не инициализированы?
- Значит конструктор идет лесом...
- Что делать будем?



# Давайте разбираться

```
public class Parent {  
    public Parent() {  
        printPi();  
    }  
  
    public void printPi(){  
        System.out.println("Pi");  
    }  
  
    public static void main(String[] args) {  
        Son son = new Son();  
    }  
}
```

```
public class Son extends Parent {  
    private double pi = Math.PI;  
  
    public Son() {  
        printPi();  
    }  
  
    @Override  
    public void printPi() {  
        System.out.println(pi);  
    }  
}
```

?

Answer:

0.0  
3.141592653589793

# Расположите в правильной последовательности

- @PostConstruct
- Spring setter (annotation) injection
- Son Initializer
- Parent Initializer
- Son inline
- Parent Inline
- Son Constructor
- Parent Constructor

# Правильная последовательность

- Parent Inline / Parent Initializer (depends on order)
- Parent constructor
- Son Inline / Son Initializer (depends on order)
- Son constructor
- Spring setter (annotation) injection
- @PostConstruct Или init methods

Зачем это нужно знать???  
Разве только для интервью!!!



# Практическое применение этих знаний

```
public class BestService {  
    public BestService(){  
        chuckNorrisMethod();  
    }  
  
    public void chuckNorrisMethod() {  
        out.println("Save the world");  
    }  
}
```

```
public static void main(String[] args) {  
    new BetterThanBestService();  
}
```

```
public class BetterThanBestService extends BestService {  
    private List cache = new ArrayList();  
  
    @Override  
    public void chuckNorrisMethod() {  
        cache.add(1);  
    }  
}
```



1. Will not compile.
2. Runtime exception.
3. You can't override Chuck Norris methods.
4. Everything is ok.

# Заменяем конструктор

- Напишите поддержку для аннотации `@PostConstruct`
- Метод на которым она будет стоять будет запускаться ObjectFactory после того, как объект будет создан и настроен
- Теперь вместо того, чтобы писать конструктор вся `init` логика будет в методе над которым будет `@PostConstruct`

Устали от рефлекшонов?  
Мы к ним вернёмся позже...



I'LL BE BACK

# Описание проблемы

- Иногда некоторые части апликации должны знать, о событиях, которые происходят с другими частями апликации.
- Как мы можем их уведомлять о происходящих событиях, без того, что они всё время будут о них спрашивать.
- Это как подписка на блог.

# Пример проблемы

- Mikalai Alimenkou, время от времени, публикует посты о политической ситуации в Киеве, в своём блоге
- Как я могу быть уверен в том, что не пропущу ни одного его поста, и прочту его сразу после публикации
- Проще говоря, как мне стать подписчиком его блога

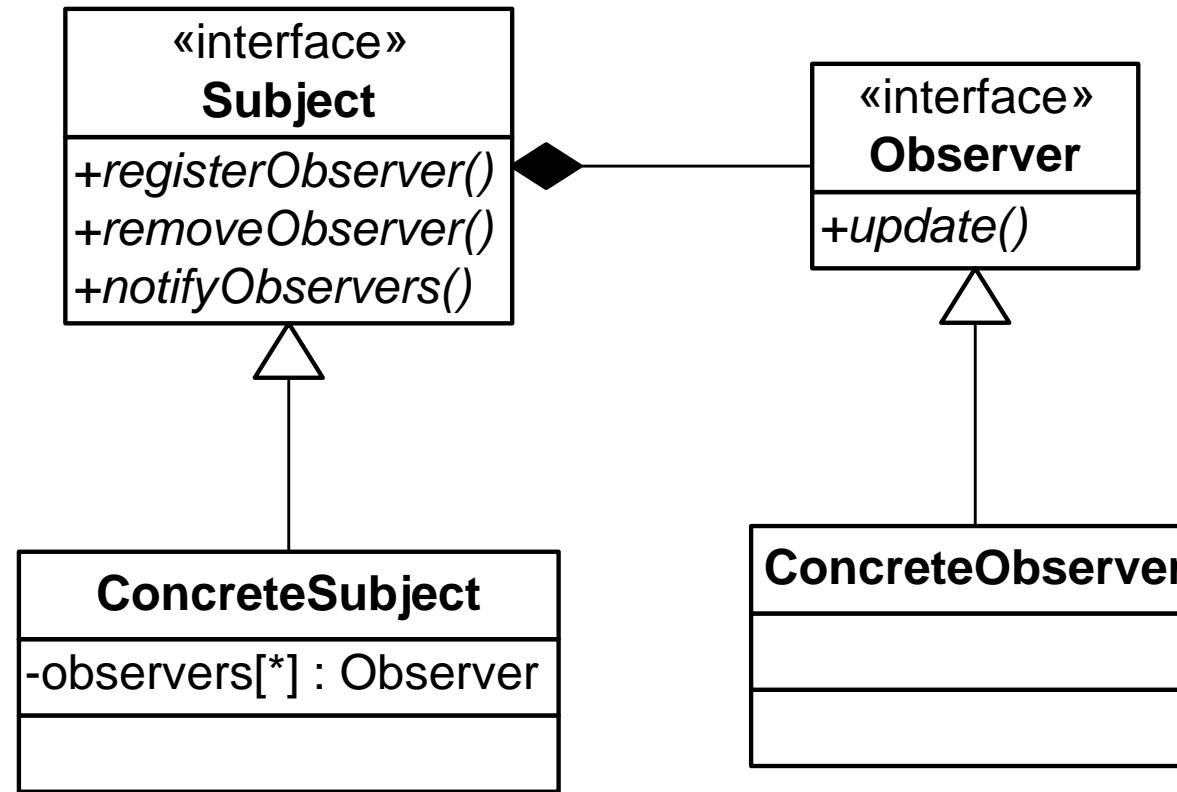
# Задание

- Напишите класс BlogEngine у которого будет метод publishPost
- Каждый подписчик должен получать сообщение о каждом посте

# Observer



# Observer – Решение



# Давайте посмотрим код

```
public class BlogEngine {  
    private Set<Subscriber> subscribers = new HashSet<Subscriber>();  
    @Inject  
    private BroadCaster broadCaster;  
  
    public void addSubscriber(Subscriber subscriber) {  
        subscribers.add(subscriber);  
    }  
  
    public void removeSubscriber(Subscriber subscriber) {  
        subscribers.remove(subscriber);  
        System.out.println("you were successfully removed");  
    }  
  
    public void savePost(BlogPost blogPost) {  
        broadCaster.broadcastPost(blogPost);  
        for (Subscriber subscriber : subscribers) {  
            subscriber.notify(new BlogPostEvent(blogPost));  
        }  
    }  
}
```

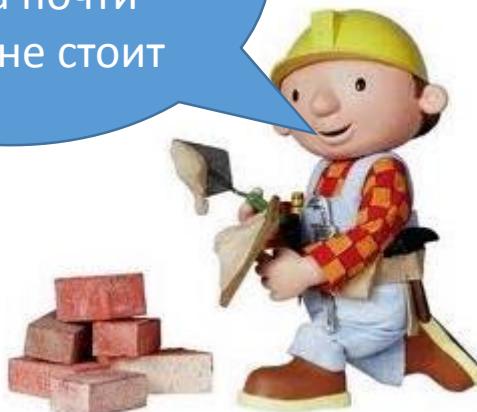
# Вы знаете что такое Immutable?

- Объект `Immutable` после того, как создан не может менять свой стэйт
- Это хорошо с многих точек зрения
- Во первых это хорошо для оптимизаций, так как стэйт неизменный.
  - например всякие работы с коллекциями, которые пользуются хашкод (ведь коли объект не может измениться, то и хашкод не может)
- А главное не надо заморачиваться по синхронизации и многопоточности



Но ведь `Immutable objects` создают кучу мусора, да и постоянное создание объектов бьёт по перформесу.

В джаве создание объекта почти ничего не стоит



Я люблю собирать молодые объекты



# Поговорим немного про виды и специфику различных сборщиков мусора



# Garbage Collector

Кто тут бессмертный?

```
public class Immortal {  
    //I don't care about encapsulation, I'm immortal!!  
    public Immortal immortal;  
  
    public Immortal() {  
    }  
}  
  
public class ImmortalTest {  
    public static void main(String[] args) {  
        Immortal immortal = new Immortal();  
        immortal.immortal = immortal;  
    }  
}
```



# Garbage Collector

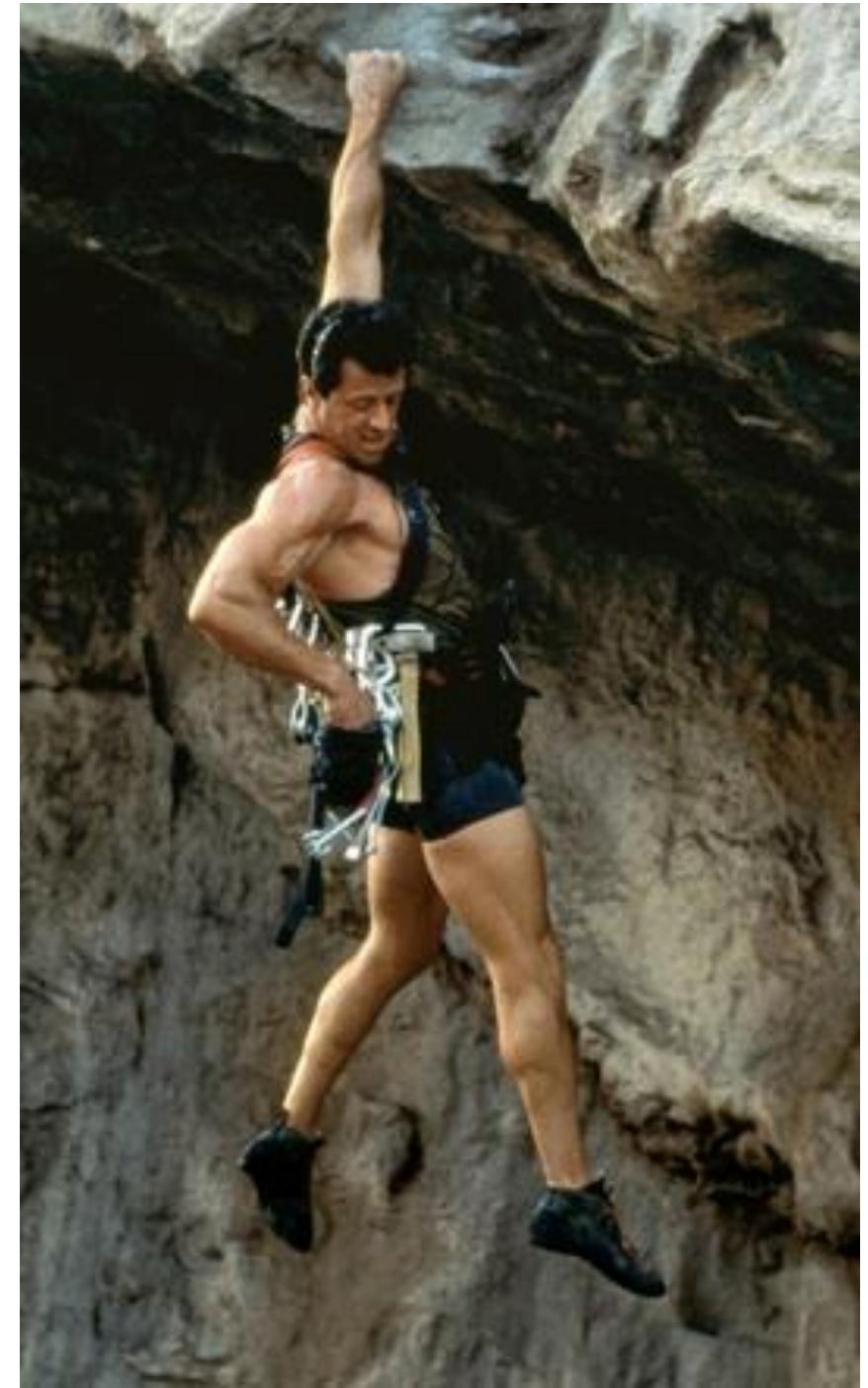
Мы бессмертны,  
пока мы нужны  
друг другу

```
Immortal a = new Immortal();  
Immortal b = new Immortal();  
a.immortal = b;  
b.immortal = a;
```

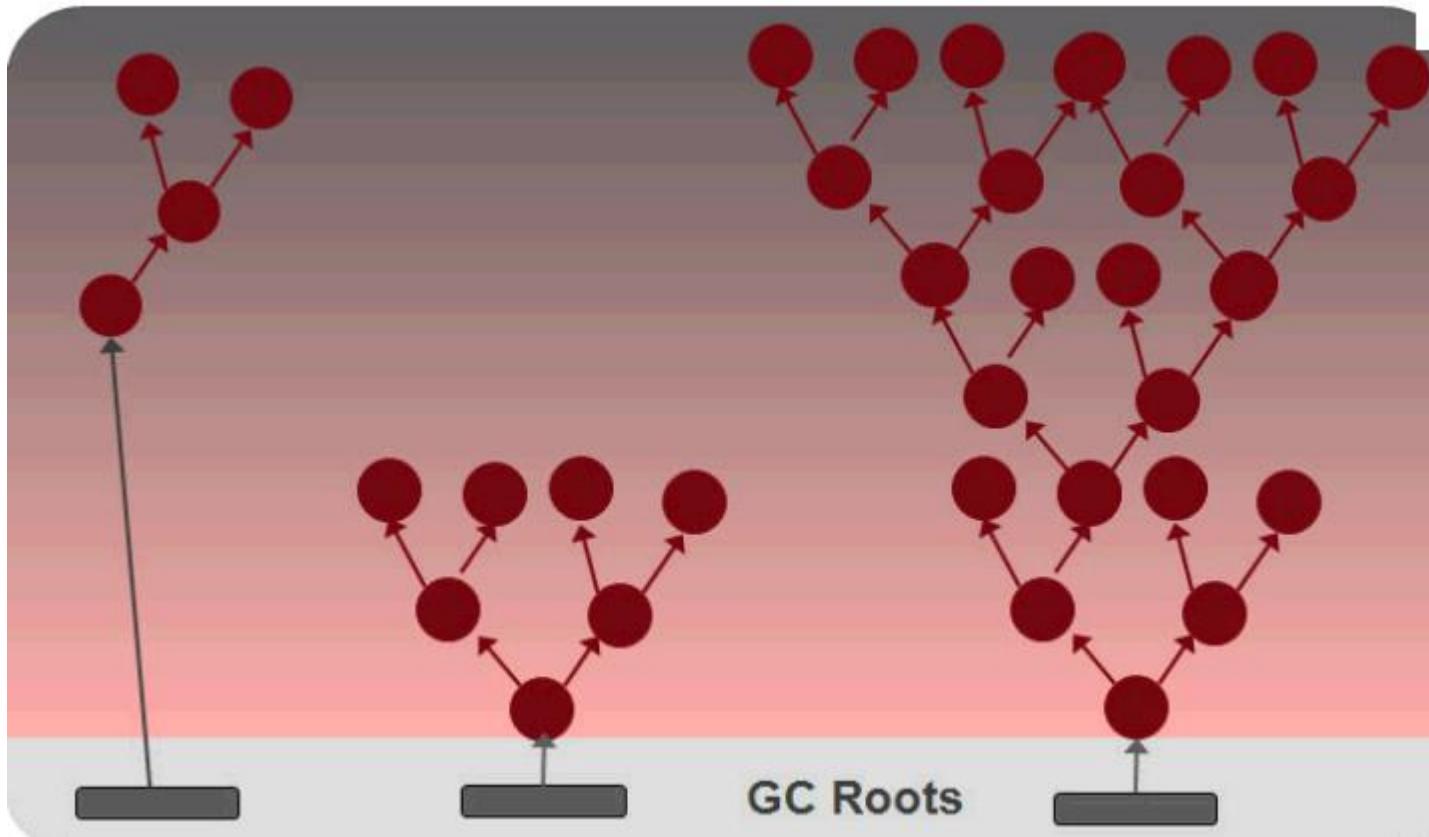


# Strong Reference

- Обычные референсы, которыми мы пользуемся в джаве называются Strong reference
- Если есть путь до рута из strong референсов до объекта, этот объект не может быть убран сборщиком мусора
- В других случаях будет собран.
- Когда-нибудь

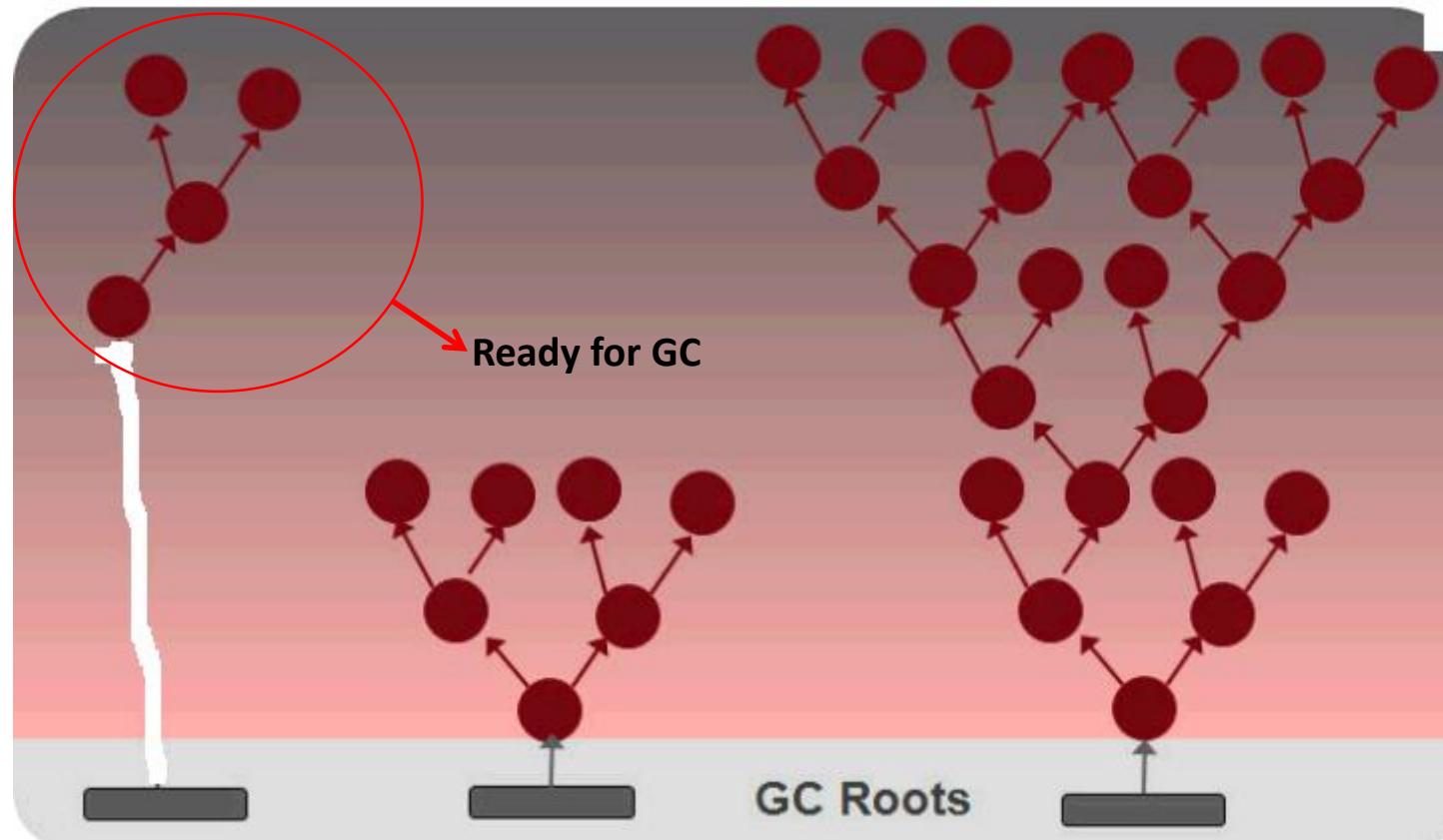


# The heap and GC



The garbage collector starts from “root” references, and walks through the object graph marking all objects that it reaches.

# The heap and GC



# Так почему всётки сборщик мусора любит молодые объекты



Покажи им Visual  
GC, ну покажи!

**ОЧЕНЬ ВАЖНЫЙ ВОРОПОС**

# Не у кого нет аллергии на пингвинов?



# Immutable task

- Напишите класс IDIClient
- State:

name : String

bonusPoints : int

debt : int



# Вопрос

- Это вообще будет компилироваться?

```
public final class IDIClient {  
    private final String name;  
    private final int bonusPoints;  
    private final int debt;
```

- Да, если будет соответствующий конструктор

```
public IDIClient(String name, int debt, int bonusPoints) {  
    this.name = name;  
    this.debt = debt;  
    this.bonusPoints = bonusPoints;  
}
```

# Immutable task

```
public class IDIClient {  
    private final String name;  
    private final int debt;  
    private final int bonusPoints;  
  
    public IDIClient(String name, int debt, int bonusPoints) {  
        this.name = name;  
        this.debt = debt;  
        this.bonusPoints = bonusPoints;  
    }  
    public String getName() {  
        return name;  
    }  
    public int getDebt() {  
        return debt;  
    }  
    public int getBonusPoints() {  
        return bonusPoints;  
    }  
}
```

А пользоваться будем как то так?

```
IDIClient orit = new IDIClient(100, 50, "Orit");
```

Вам это нравится?

А если будет больше мемберов?  
Сделаем конструктор на все?

# Builder Pattern

```
public class IDIClient {  
    private int bonusPoints;  
    private int debt;  
    private String name;  
  
    private IDIClient() {}  
  
    public static class Builder {...}  
        public int getBonusPoints() {  
            return bonusPoints;  
        }  
  
        public int getDebt() {  
            return debt;  
        }  
  
        public String getName() {  
            return name;  
        }  
    }  
  
    public static class Builder {  
        private IDIClient client;  
  
        public Builder() {  
            client = new IDIClient();  
        }  
  
        public void setBonusPoints(int bonusPoints) {  
            client.bonusPoints = bonusPoints;  
        }  
  
        public void setDebt(int debt) {  
            client.debt = debt;  
        }  
  
        public void setName(String name) {  
            client.name = name;  
        }  
  
        public IDIClient newInstance() {  
            IDIClient immutableClient = client;  
            client = new IDIClient();  
            return immutableClient;  
        }  
    }
```

# Builder Pattern

```
public static void main(String[] args) {
    IDIClient.Builder builder = new IDIClient.Builder();
    builder.setBonusPoints(100);
    builder.setDebt(50);
    builder.setName("Orit");
    IDIClient idiClient = builder.newInstance();
    System.out.println(idiClient);
}
```

```
IDIClient orit = new IDIClient.Builder().bonusPoints(100).debt(50).name("Orit").newInstance();
```

```
public static class Builder {
    private IDIClient client;

    public Builder() {
        client = new IDIClient();
    }

    public Builder bonusPoints(int bonusPoints) {
        client.bonusPoints = bonusPoints;
        return this;
    }

    public Builder debt(int debt) {
        client.debt = debt;
        return this;
    }

    public Builder name(String name) {
        client.name = name;
        return this;
    }

    public IDIClient newInstance() {
        IDIClient immutableClient = client;
        client = new IDIClient();
        return immutableClient;
    }
}
```

# Builder Pattern

Я не могу их  
сделать final,  
ведь билдер их  
вносит после  
того, как объект  
создан



```
public class IDIClient {  
    private int bonusPoints;  
    private int debt;  
    private String name;  
  
    private IDIClient() {}
```

```
public static class Builder {...}
```

Посмотри на свой  
“immutable class”.  
Все поля теперь  
не final.



# Builder Pattern

- Кроме того, данный подход подразумевает, возможность создать объект, заполненный отчасти
- А если есть обязательные поля?

Думай ещё!



# Builder Pattern

```
public class IDIClient {  
    private final String name;  
    private final int debt;  
    private final int bonusPoints;  
  
    private IDIClient(String name, int debt, int bonusPoints) {  
        this.name = name;  
        this.debt = debt;  
        this.bonusPoints = bonusPoints;  
    }  
    public static class Builder {  
        public String getName() {  
            return name;  
        }  
        public int getDebt() {  
            return debt;  
        }  
        public int getBonusPoints() {  
            return bonusPoints;  
        }  
    }  
}
```

Следующий слайд

# Builder Pattern

```
public static class Builder {
    private String name;
    private int debt;
    private int bonusPoints;
    public Builder() {}
    public Builder name(String name) {
        this.name = name;
        return this;
    }
    private void cleanBuilderFields() {...}
    private boolean clientIsValid() {...}

    public Builder debt(int debt) {
        this.debt = debt;
        return this;
    }
    public Builder bonusPoints(int bonusPoints) {
        this.bonusPoints = bonusPoints;
        return this;
    }
    public IDIClient createIDIClient() {
        if (clientIsValid()) {
            IDIClient client = new IDIClient(name, debt, bonusPoints);
            cleanBuilderFields();
            return client;
        }
        throw new IDIClientCanBeBuildException("name is null");
    }
}

private void cleanBuilderFields() {
    name = null;
    debt = 0;
    bonusPoints = 0;
}

private boolean clientIsValid() {
    if (name == null) return false;
    return true;
}
```

# Builder Pattern



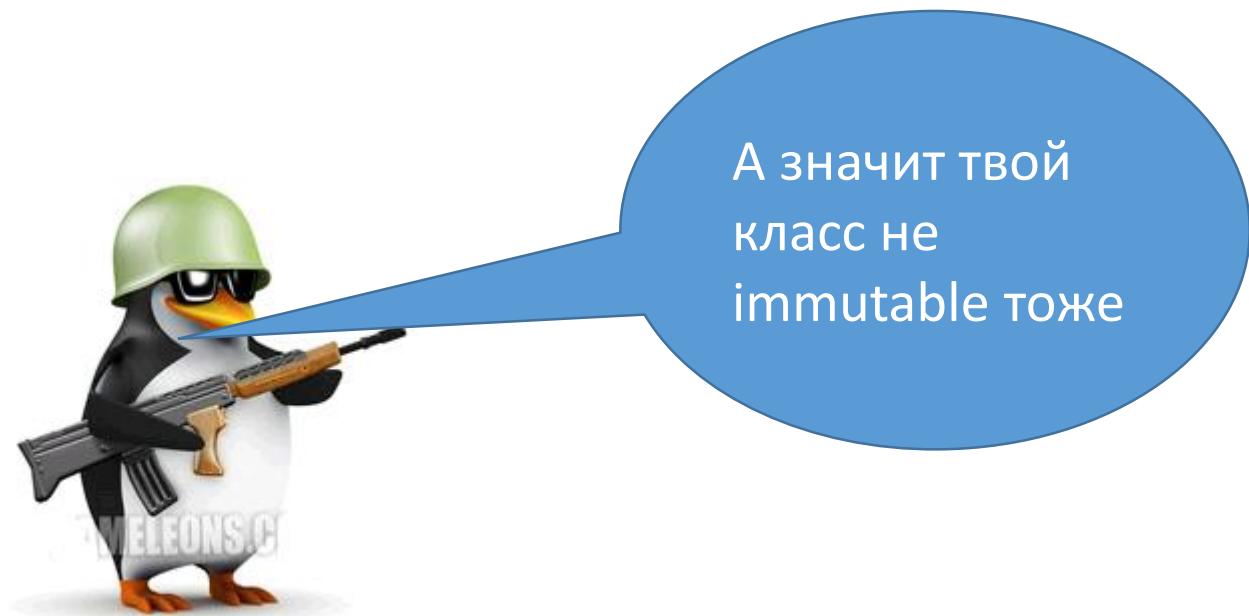
# Builder Pattern

```
public class IDIClient {
    private final String name;
    private final int debt;
    private final int bonusPoints;
    private IDIClient(String name, int debt, int bonusPoints) {
        this.name = name;
        this.debt = debt;
        this.bonusPoints = bonusPoints;
    }
    public static class Builder {...}

    public IDIClient setName(String name) {
        return new IDIClient(name, bonusPoints, debt);
    }
    public IDIClient setBonusPoints(int bonusPoints) {
        return new IDIClient(name, bonusPoints, debt);
    }
    public IDIClient setDebt(int debt) {
        return new IDIClient(name, bonusPoints, debt);
    }
    public String getName() {...}
    public int getDebt() {...}
    public int getBonusPoints() {...}
}
```

# Builder Pattern

- А что если один из полей вашего класса не является `immutable`?
- То его можно получить через `getter` и поменять



# Builder Pattern

```
public class IDIClient {  
    private final String name;  
    private final int debt;  
    private final int bonusPoints;  
    private final Date date;  
    private IDIClient(String name, int debt, int bonusPoints, Date date) {  
        this.name = name;  
        this.debt = debt;  
        this.bonusPoints = bonusPoints;  
        this.date = date;  
    }  
    public Date getDate() {  
        return date;  
    }  
    public Builder date(Date date) {  
        this.date = date;  
        return this;  
    }  
}  
  
IDIClient client = new IDIClient.Builder().name("Vadim").date(date).createIDIClient();  
  
client.getDate().setTime(1000)
```



Пингвины forever!



# Defensive copy

- Никогда не возвращай оригиналъный объект своего не immutable поля, возвращай копию.

```
public Date getDate() {           → private Date defensiveDateCopy(Date date) {  
    return defensiveDateCopy(date);  
}                                return new Date(date.getTime());  
}
```

- Но еще правильней...
- Строить immutable object из других immutable  
Например вместо Date использовать JodaTime

# Defensive copy



Кстати тут  
проблем  
нету?

```
private Date defensiveDateCopy(Date date) {  
    return date == null ? null : new Date(date.getTime());  
}  
}  
} else {  
    return new Date(date.getTime());  
}  
}
```

# Builder Pattern



# Builder Pattern

```
Date date = new Date();
IDIClient masha = new IDIClient.Builder().name("Masha").date(date).createIDIClient();

date.setTime(1000000);
```



# Use defensive copy in constructor

```
public Builder date(Date date) {  
    this.date = new Date(date.getTime());  
    return this;  
}
```

Вот теперь у нас реально Immutable object



# Task

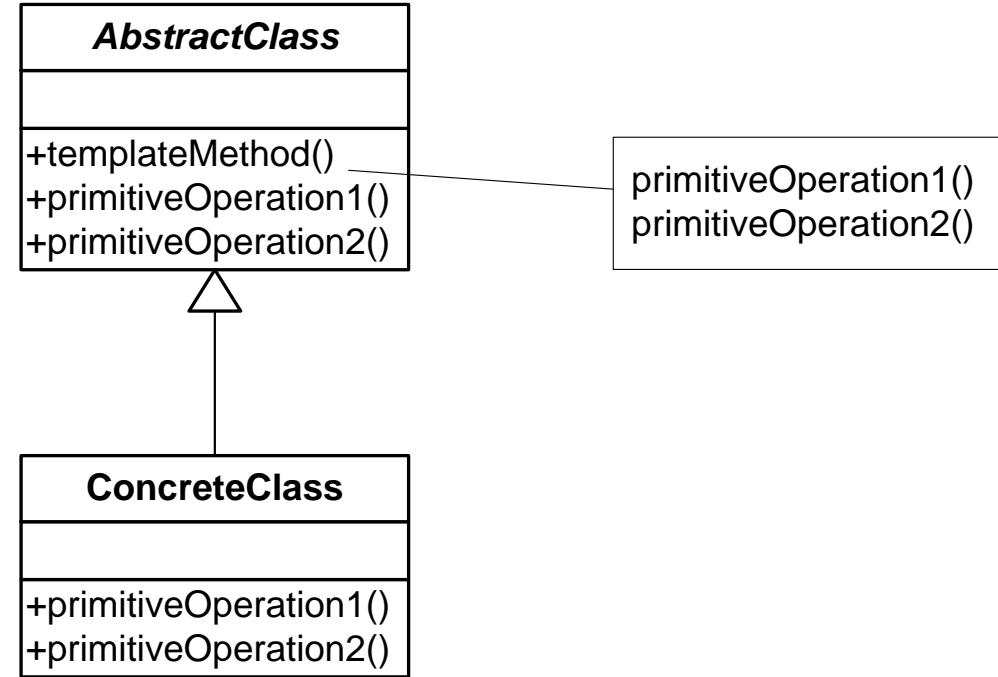
- Write structure for game module.
- There are can be different implementation of this module
- You know the order of steps, but algorithm of each step may change
- Steps are: prepare board-> player 1 move -> player 2 move ->
- Until some condition
- Calculate score
- Update best scores

# Решение – Template method

- Метод определяющий основу алгоритма и позволяющий наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.

# Template Method – Решение

- Опционально суперкласс может предоставить defaultную имплементацию



# Следующее задание

- Предположим мы разрабатываем игру аля Heroes.
- Надо написать утилиту, которая сможет подсчитать сколько денег есть у каждого игрока.
- Все чем владеет игрок (солдаты, замки, леса) может быть переведено в деньги. Для каждого типа объекта есть уникальная формула для вычисления его стоимости.
- Наша утилита должна получить лист всех объектов и вернуть его стоимость.

# Iterator

```
public class WealthCalculator {
    public double calculateWealth(List<Costable> costables) {
        double sum=0;
        for (Costable costable : costables) {
            sum += costable.calculateCost();
        }
        return sum;
    }
}

public interface Costable {
    double calculateCost();
}
```

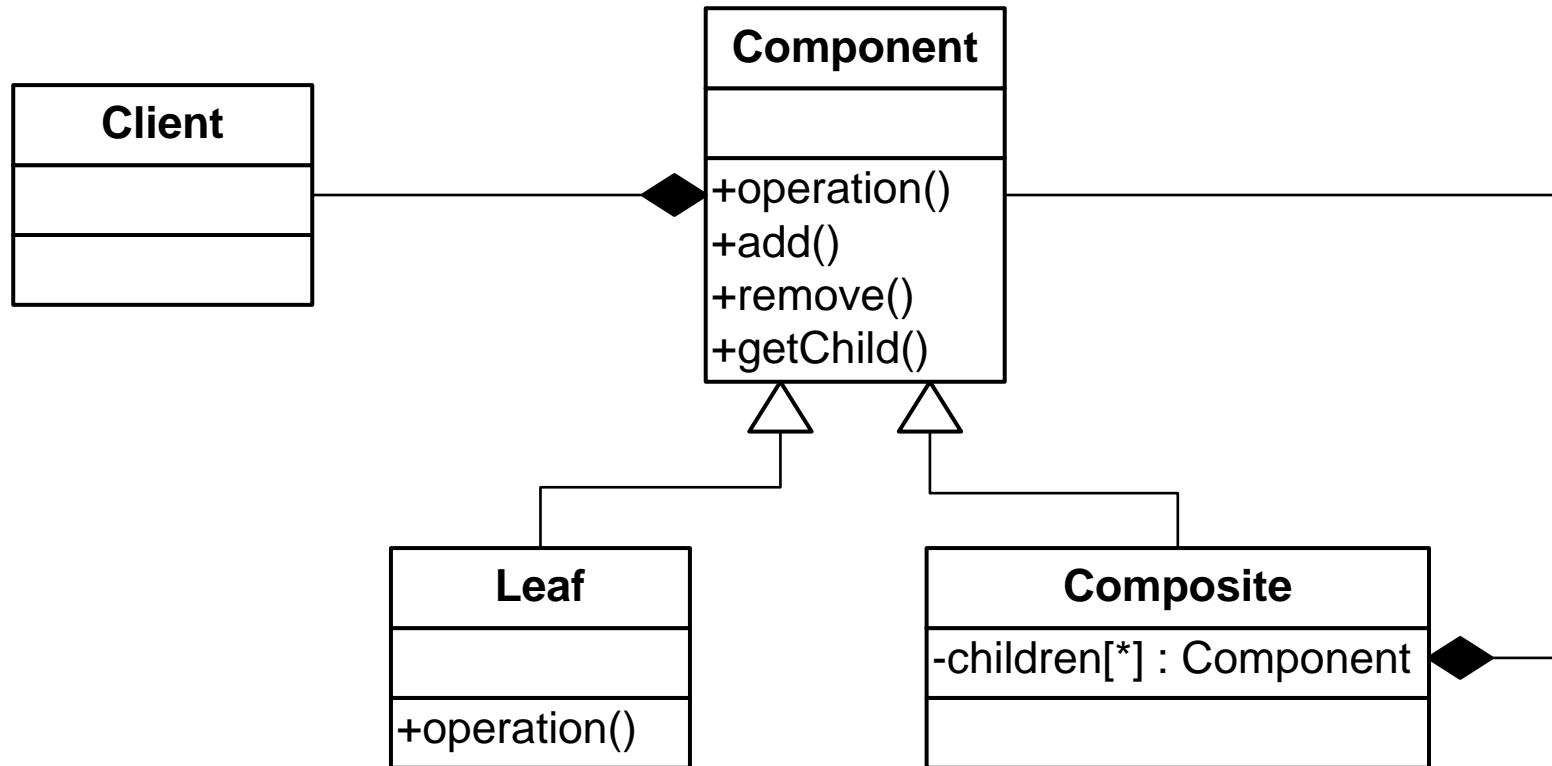
# Еще задание

- Постройте следующую структуру:
- Нужны следующие классы: комната, шкаф, полка, книга
- Как можно догадаться по смыслу часть объектов может являться контейнером для других (например в комнате может быть несколько шкафов, с полками на которых книги, или книги могут валяться в шкафу или даже в комнате)
- Могут добавится еще новые классы (типа дом)
- Важно, чтобы каждый объект являющийся контейнером умел добавлять в себя контейнеры или компоненты
- И самое главное, каждый из объектов должен уметь подсчитать свою стоимость, которая состоит из стоимости его + всех объектов, которые он вмещает
- Нет никаких ограничений по контейнерам. Например в шкафу могут помещаться ещё 2 шкафа, комнаты и три полки с книгами

# Composite

- **Компоновщик** - объединяет объекты в древовидную структуру для представления иерархии от частного к целому. Компоновщик позволяет клиентам обращаться к отдельным объектам и к группам объектов одинаково
- Паттерн определяет иерархию классов, которые одновременно могут состоять из примитивных и сложных объектов, упрощает архитектуру клиента, делает процесс добавления новых видов объекта более простым

# Composite – The Solution



# Composite – Пример

Menú

|           |                |
|-----------|----------------|
| Búsqueda  | Ing. Nuevo     |
| Modificar | Nvo Formulario |
| Guardar   | Salir          |

Paquete

Orden:

Cliente:

Marca:

Creado:  Modificado:

**Nvo Paquete** **Nvo Artículo**

**Descripción de Artículo**

Frente

Estilo:

Hilo:

Tela: tipo  color  código

Arte:

Ref:

Espalda

Cuerpo:

Impresión:

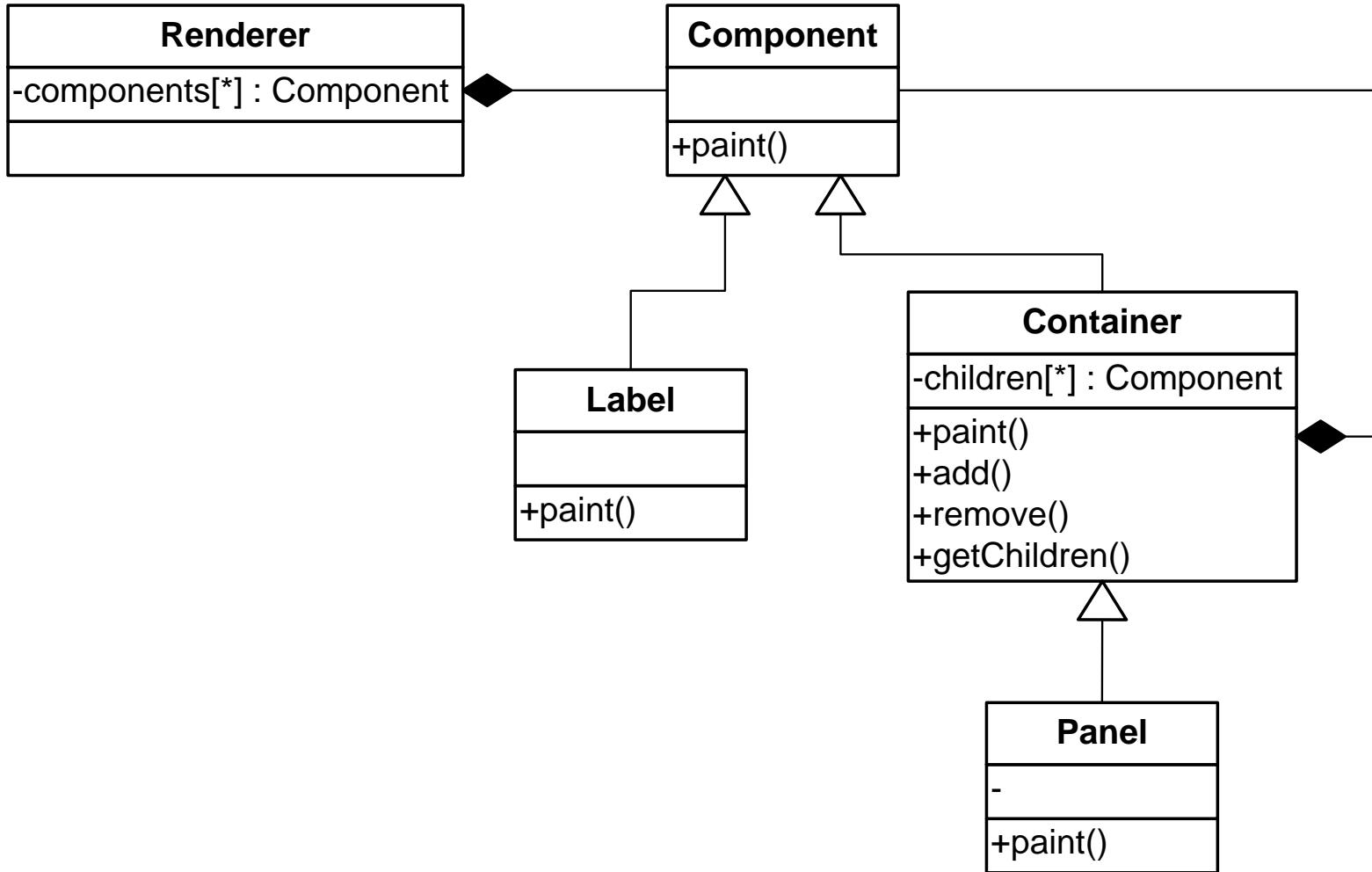
Ref:

Manga Izquierda  Manga Derecha



frame.repaint()

# Composite – Solution Example



# Composite – Solution Example

```
1 public interface Component {  
2  
3     void paint();  
4 }
```

```
1 public abstract class Container implements Component {  
2  
3     public List<Component> children;  
4  
5     public boolean add(Component component) {  
6         return children.add(component);  
7     }  
8  
9     public Component remove(int index) {  
10        return children.remove(index);  
11    }  
12  
13    public void paint() {  
14        for (Component component : children) {  
15            component.paint();  
16        }  
17    }  
18}  
19 }
```

```
1 public class Label implements Component {  
2     public void paint() {  
3         //paint label  
4     }  
5 }  
6 }
```

```
1 public class Panel extends Container {  
2  
3     @Override  
4     public void paint() {  
5         //paint itself - borders etc.  
6         super.paint();  
7     }  
8 }
```

---

```
1 public class UiRenderer {  
2  
3     List<Component> components;  
4  
5     public void paintUi() {  
6         for (Component component : components) {  
7             component.paint();  
8         }  
9     }  
10 }
```

# Более сложное задание

- Напишите утилиту, у которой будет метод подсчитывающий сколько раз определённый объект встречается в коллекшоне
- Метод должен получать лист и объект, после чего пройтись по данному лист в поисках данного ответа, и вернуть число повторений данного объекта в этом листе.

Так надо всего лишь implements equals()



## А вот и нет...

- Во первых никто не сказал, что все объекты создаются из моих классов, следовательно не везде я могу контролировать имплементацию метода equals()
- А кроме того, я хочу иметь возможность решать отдельно, для каждой инвокации данного метода, какой критерий равенства между объектами.

# Closures

- Что мы можем передавать в метод?
  - primitive, references to objects
  - А алгоритм передать мы можем?
- Только в JAVA 8, а её пока нет, у нас на проекте
- И что же делать сейчас? Ждать?
- Колбэк метод

# Callback method

```
public class DuplicateCounterUtil {  
    public static <T> int calcDuplicates(List<T> objects, T obj, MyEqualator<T> equalator) {  
        int counter = 0;  
        for (T object : objects) {  
            if (equalator.isEqual(object, obj)) {  
                counter++;  
            }  
        }  
        return counter;  
    }  
}
```

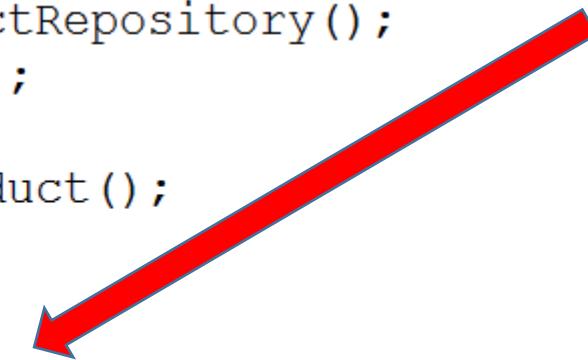
# Callback method

- Еще примером этого подхода является comparator
- Или всякие jdbcTemplate-ы
- Ну а с восьмой джавы можно делать это лямбдами

# Задание - миграция

```
ProductRepository productRepository = new ProductRepository();
ArrayList<Costable> products = new ArrayList<>();
for (int i = 0; i < 5; i++) {
    Costable product = productRepository.getProduct();
    products.add(product);
}
int total = Calculator.calculateTotal(products);
System.out.println("total = " + total);
```

```
public class Calculator {
    public static int calculateTotal(List<Priceble> items) {
        int total = 0;
        for (Priceble item : items) {
            total += item.getPrice();
        }
        return total;
    }
}
```



# Adapter

# Adapters

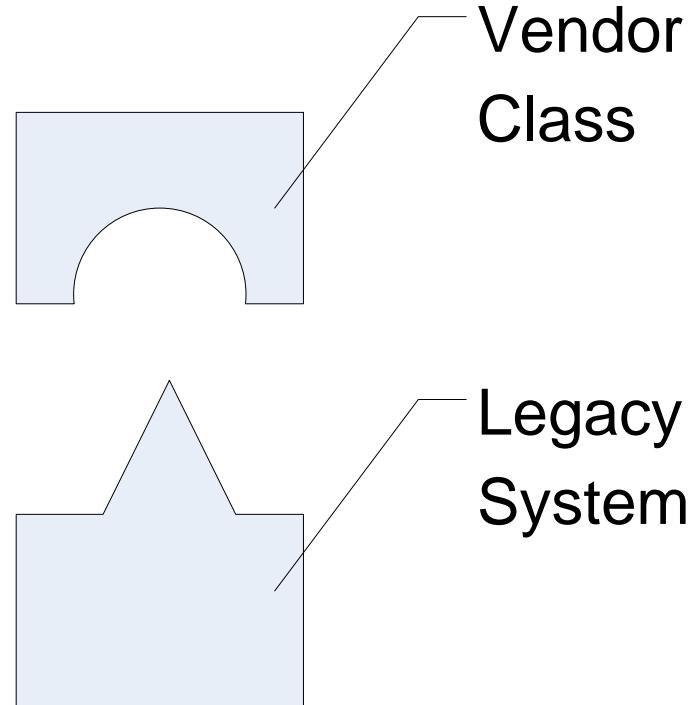


- Простой
  - Только меняет интерфейс
- Более сложный
  - Имеет некую логику внутри

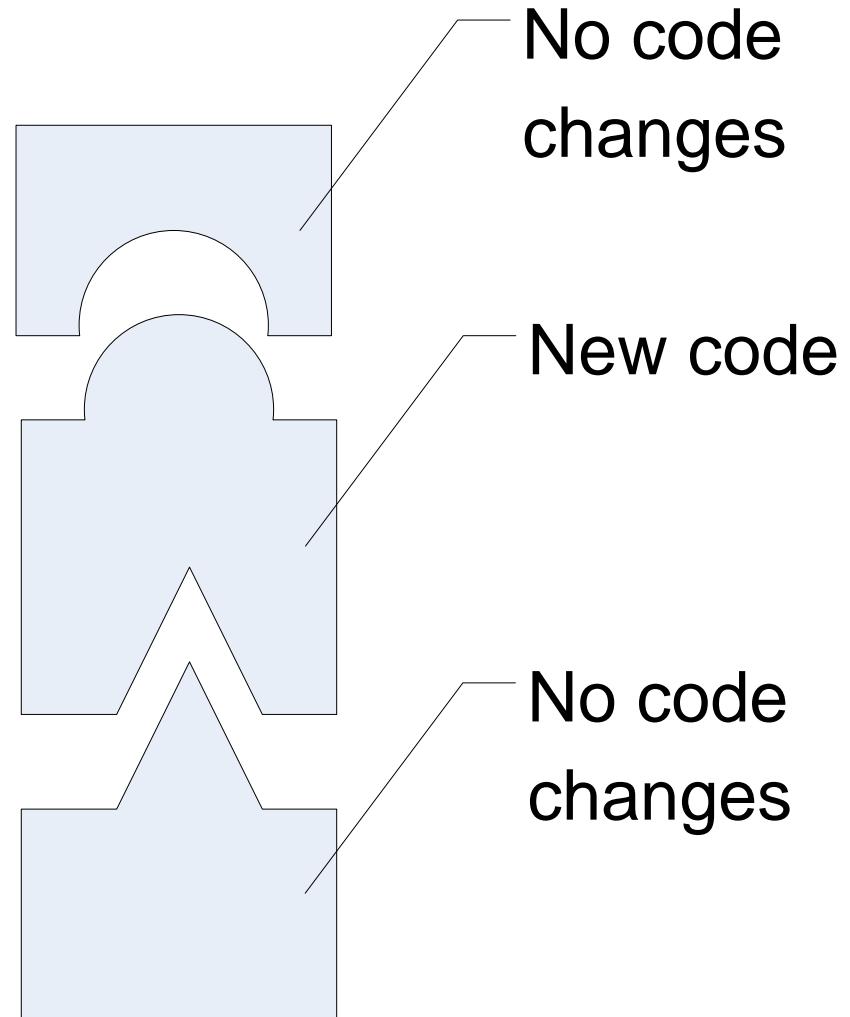


# Adapter - Проблема

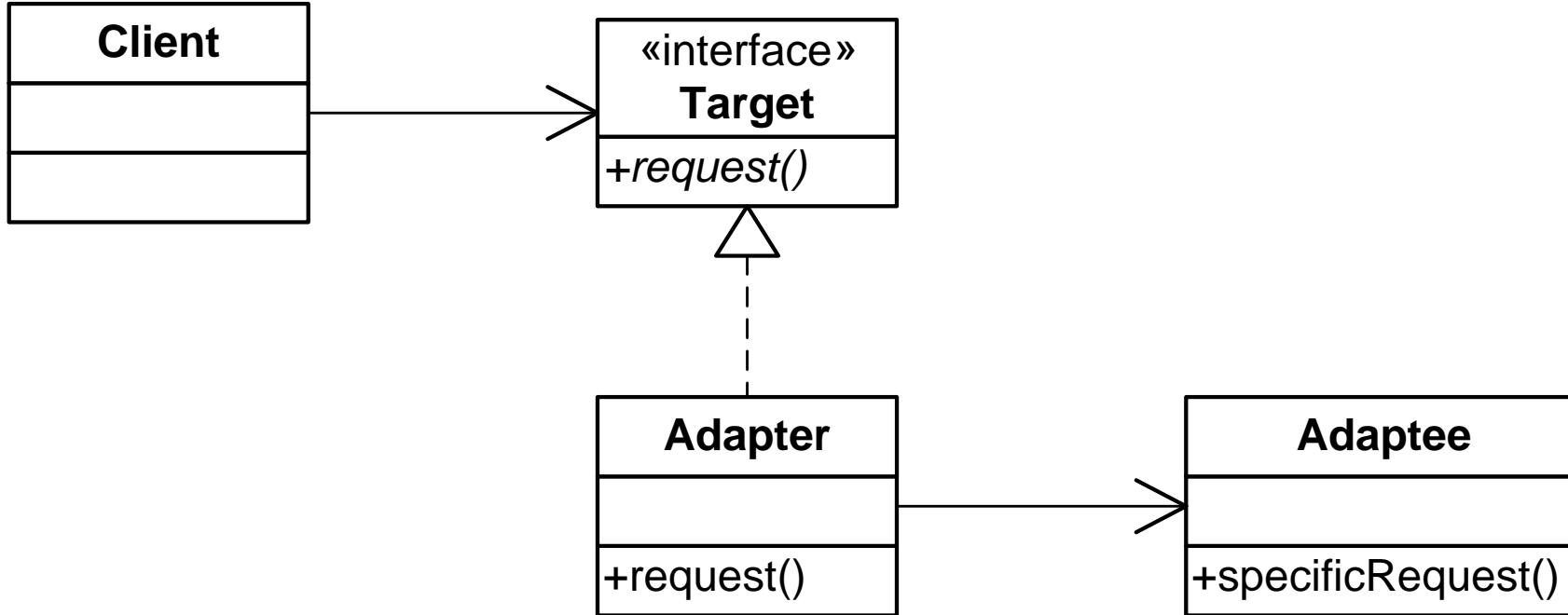
- Иногда нужны провести интеграция между двумя мирами, который работают против разных интерфейсов, но при этом имеют похожую функциональность
- И нет возможности менять уже существующий API



# Adapter – Решение



# Adapter – The Solution



- Адаптер пользуется композицией, чтобы делегировать в адаптируемый объект

# А вы знаете что такое Benchmark?

- 4 уровня понимания бенчмарка
  - 1. Студент
  - 2. Junior Software Engineer
  - 3. Senior Software Engineer
  - 4. Архитектор

# Уровень первый - Студент



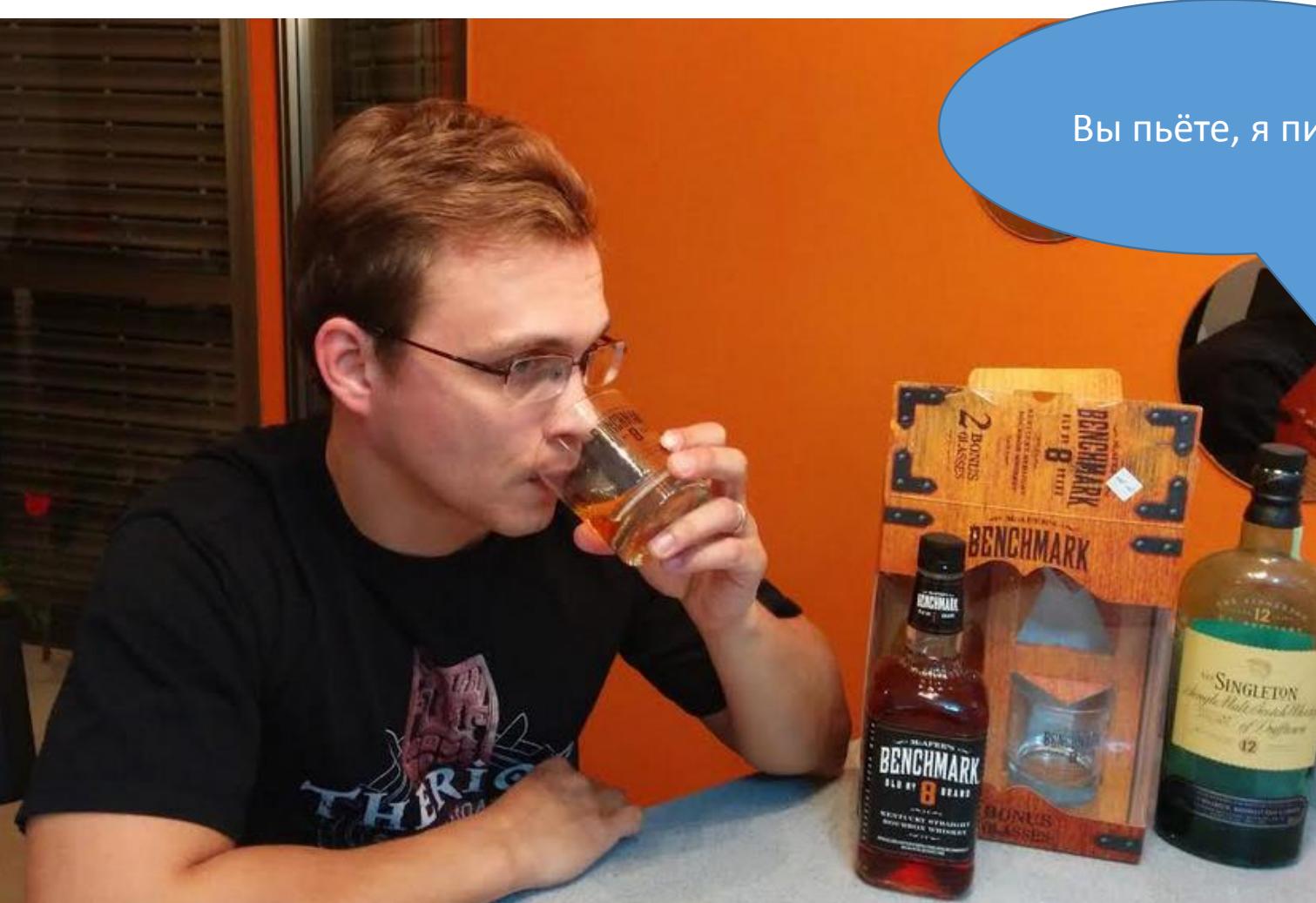
# Уровень второй - Junior Software Engineer

```
public static void main(String[] args) {
    Random random = new Random();
    long before = System.nanoTime();
    for (int i = 0; i < 1000000; i++) {
        random.nextInt(100);
    }
    long after = System.nanoTime();
    System.out.println((after-before)/1000000);
}
```

# Уровень третий - Senior Software Engineer

```
public static void main(String[] args) {  
    Random random = new Random();  
    int unhappyNumber= 0;  
    long before = System.nanoTime();  
    for (int i = 0; i < 1000000; i++) {  
        unhappyNumber = random.nextInt(100);  
    }  
    long after = System.nanoTime();  
    System.out.println((after-before)/1000000);  
    System.out.println(unhappyNumber);  
}
```

# Уровень четвертый - Архитектор



Вы пьёте, я пишу



# Задание

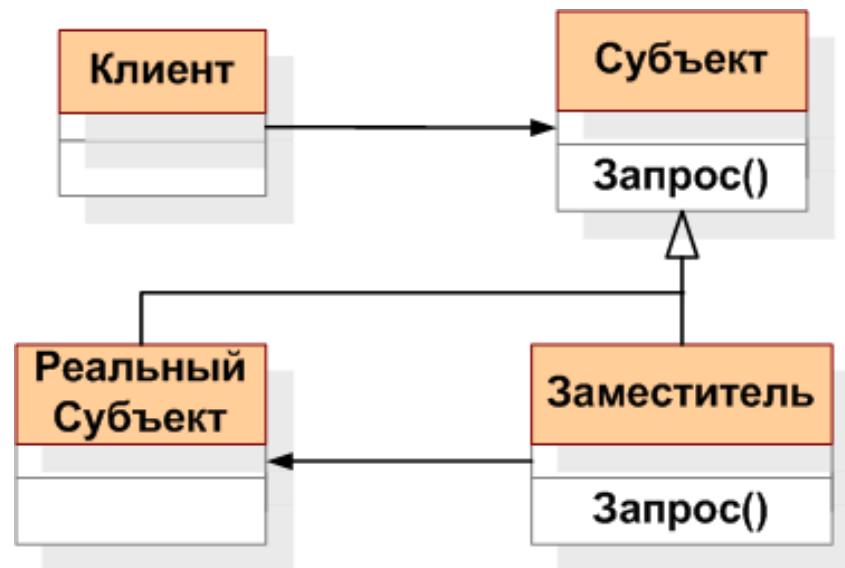
- Надо сделать бенчмарк методу clean у CleanerImpl
- Нельзя менять уже написанные классы в которых есть бизнес логике.

# Proxy

# Proxy

- **Заместитель:** объект, который контролирует доступ к другому объекту, перехватывая все вызовы (выполняет функцию контейнера).

# Proxy – «Заместитель»



**Есть два способа создать объект прокси:**

1. Имплементировать те же интерфейсы + делегация
2. Наследовать от оригинального класса

# Еще пример прокси

- А давайте сделаем Cache...

# Dynamic Proxy

- А вот теперь давайте все тоже самое, только на уровне инфраструктуры.
- Пишем свою аннотацию @Benchmark
- И обучаем наш ObjectFactory создавать прокси для объектов, классы которых аннотированы соответствующими аннотациями

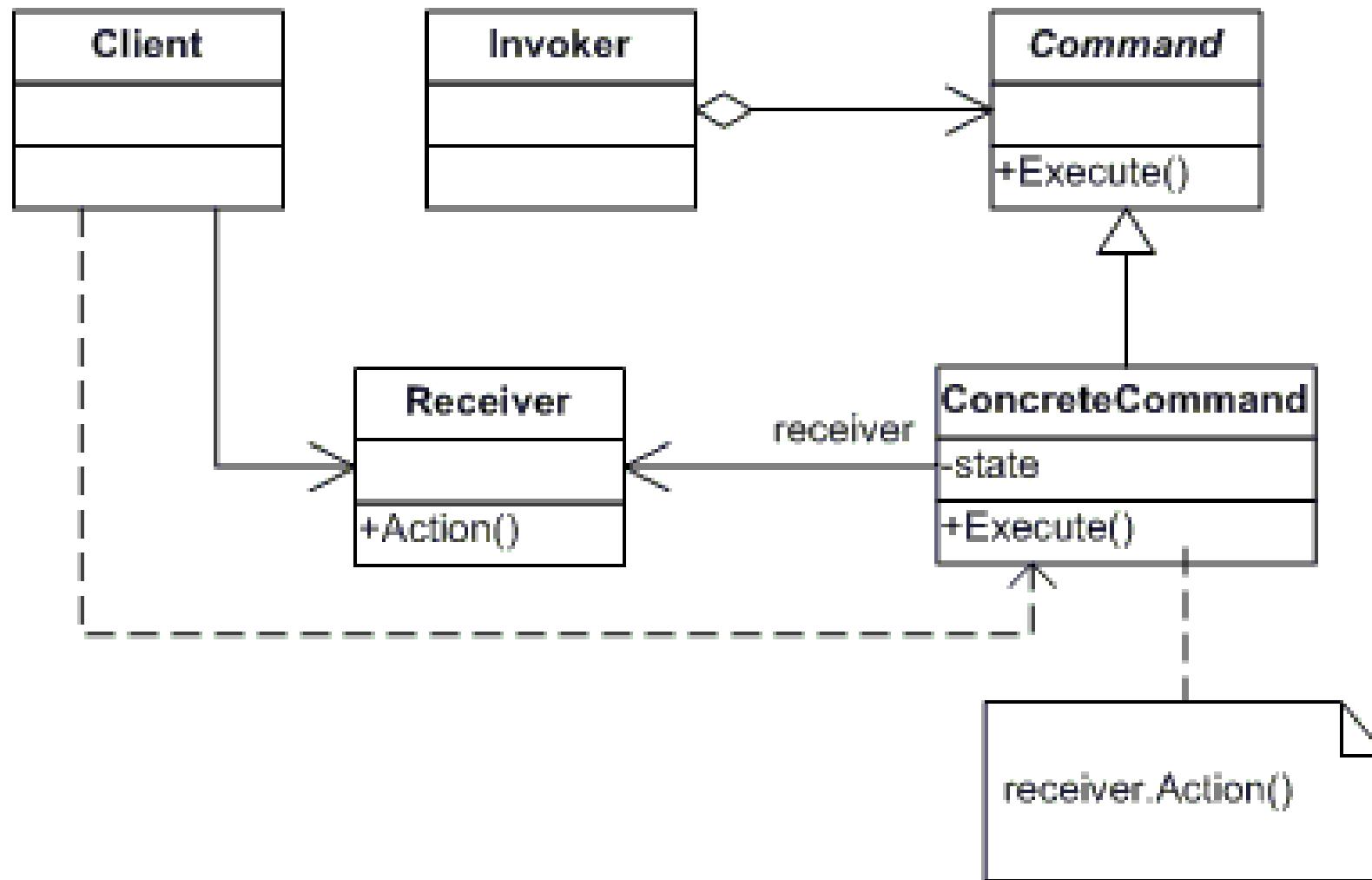
# Задание

- Напишите класс BankAccount у которого будет id, balance и методы увеличить и уменьшить баланс
- Напишите AccountManager у которого будет методы transferMoney(BankAccount a1, BankAccount a2, int amount)
  - depositeMoney(BankAccount account, int amount)
  - withDrawMoney(BankAccount account, int amount)
  - undo() – этот метод отменяет любую последнюю операцию, причём, если его вызвать несколько раз подряд он отменит несколько последних операций в обратном порядке.

# Command (Команда)

- Паттерн поведения объектов, известен также под именем Action (действие).
- Обеспечивает обработку команды в виде объекта, что позволяет сохранять её, передавать в качестве параметра методам, а также возвращать её в виде результата, как и любой другой объект.
- Объект команды заключает в себе само действие и его параметры.

# Command (Команда)



# А теперь напишем Gradle (сборщик проекта)

- Нужна платформа для написания тасков.
- Каждый таск может зависеть от других (одного или нескольких)
- У каждого таска есть action
- Пользователь нашей инфраструктурой может целый Lifecycle для своего билда, а потом запускать любой из тасков в цепочке
- Возьмем пример

# Task Deploy

- Deploy зависит от build-a
- Build зависит от assemble и check
- Check зависит от pack
- Pack зависит от tests
- Tests зависит от compileTests
- CompileTests зависит от compileSources

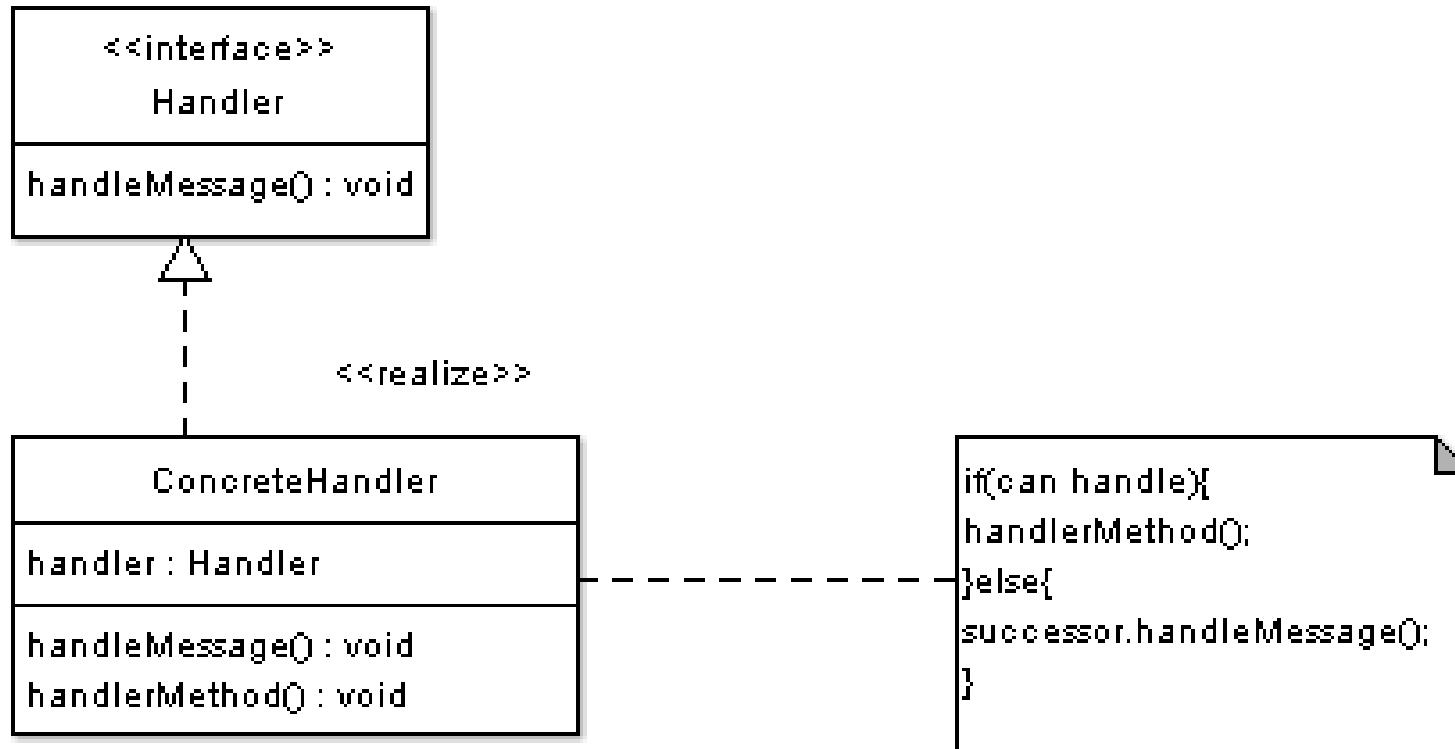
Если запустить action таска deploy? Будет примерно так:

- Compiling sources
- compiling tests...
- tests are running
- packing classes...
- Assembling...
- checking...
- Build finished!!!
- deploying...

# Задание

- Пишем класс Product у которого есть цена
- Есть ProductHandler, который должен уметь делать обработку продукту, НО
- Если продукт до 100 долларов, то будет одна обработка, если от 100 до 1000 – то другая
- 10000-100000 - третья

# Chain-of-responsibility pattern (Ланцюжок відповіальностей)



# Façade

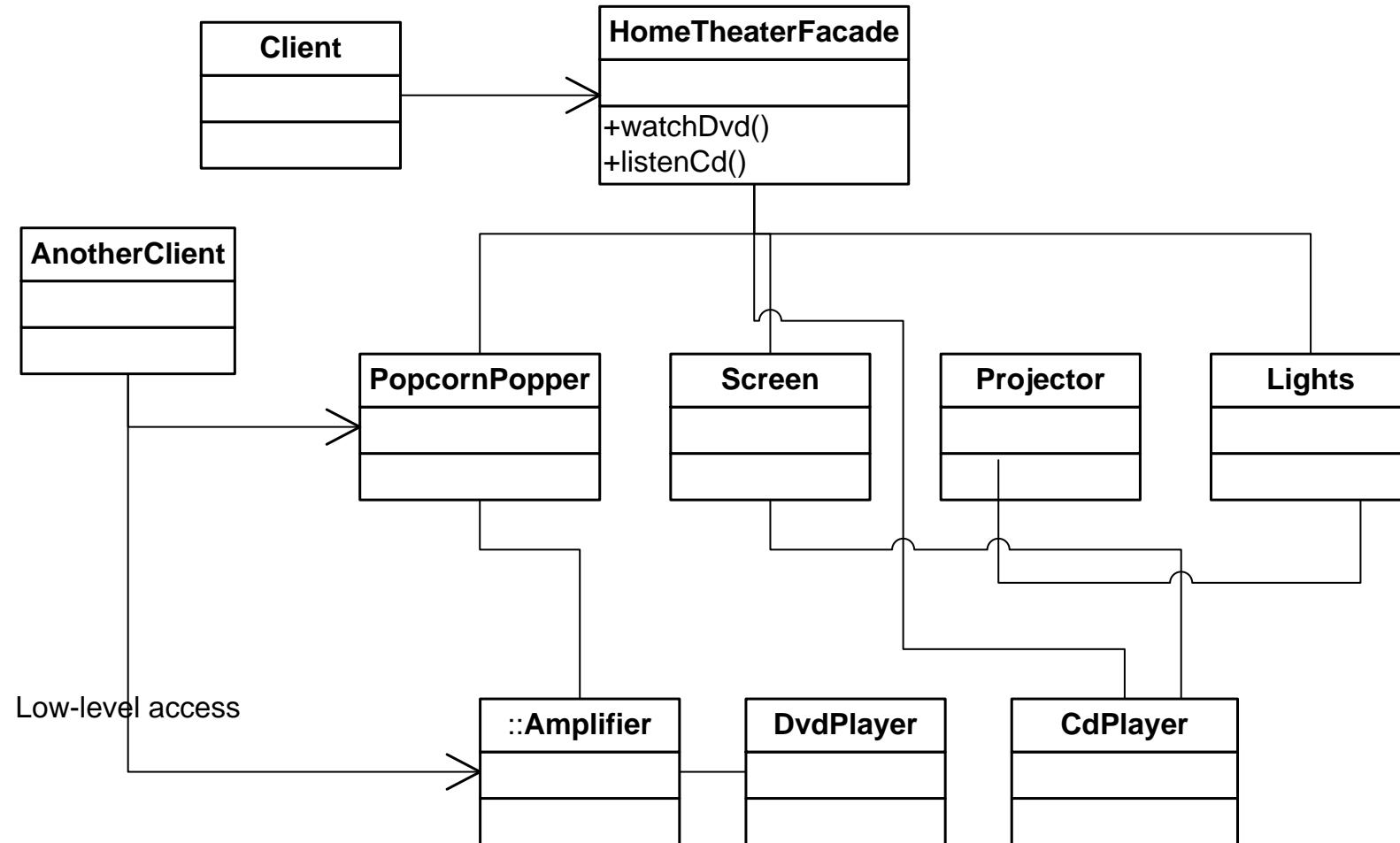
# Façade – The Problem

- Иногда хочется сделать кучу вещей не вставая со стула
- Example – Барух смотрит домашний кинотеатр:
  - PopcornPopper.pop()
  - Lights.dim()
  - Screen.down()
  - Projector.on()
  - Projector.setInput(DVD)
  - Projector.setMode(WIDE\_SCREEN)
  - Amplifier.on()
  - Amplifier.setInput(DVD)
  - Amplifier.setMode(DOLBY)
  - Amplifier.setVolume(5)
  - Player.on()
  - Player.play()

# Façade – The Solution

- Предоставляет один объединённый интерфейс к подсистемам (в данном случае слово интерфейс не имеет отношения к джаве)
- Для того, чтобы было удобнее пользоваться системой, не входя в детали
- А кроме того, что так проще системой пользоваться но еще и происходит декапалинг, между клиентом и под системами
- the subsystem, minimizing the impact of change in the subsystem
- Чем то похоже на адаптер, но
  - Адаптер конвертирует интерфейс
  - Фасад упрощает его

# Façade – The Solution



# State

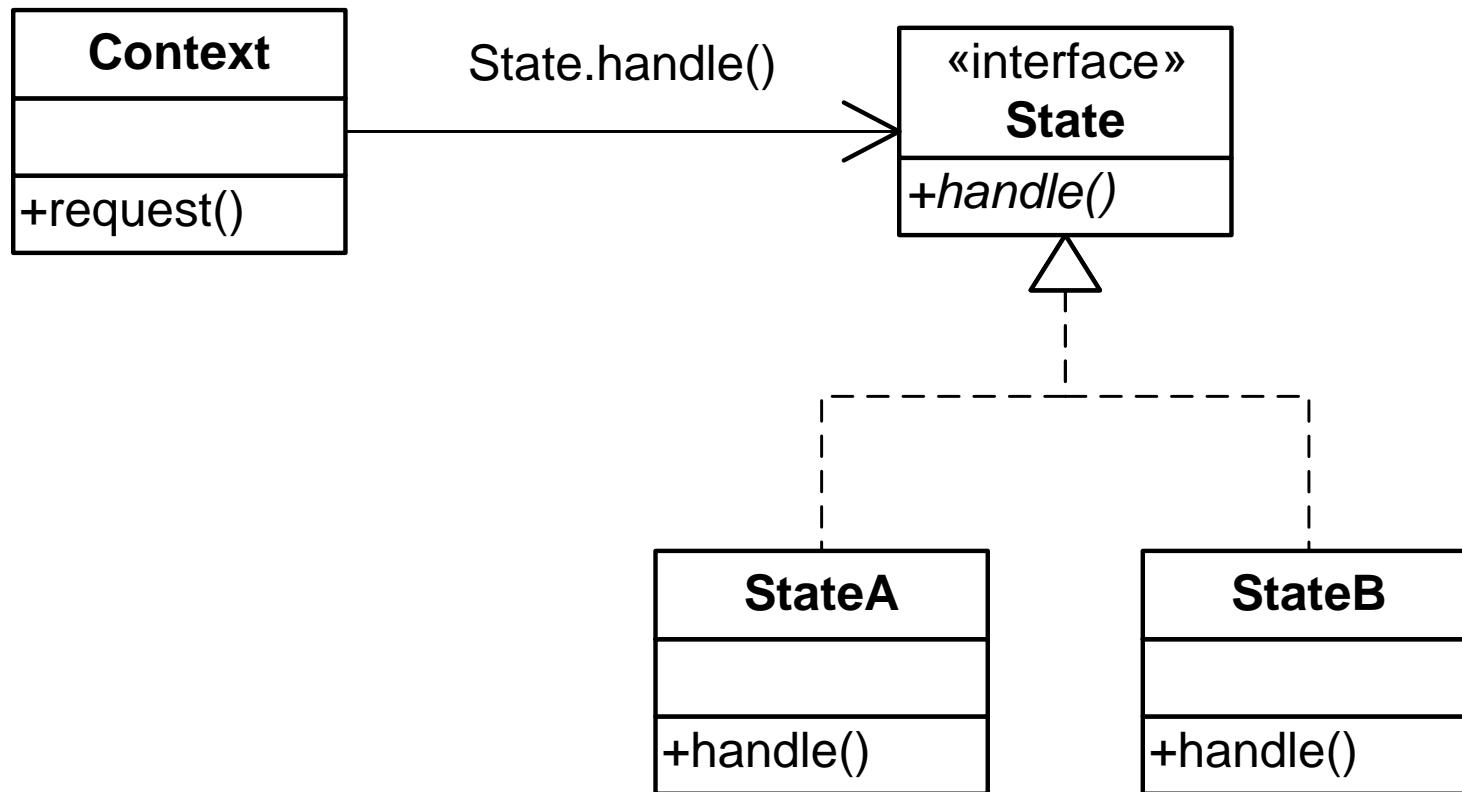
# State – The Problem

- A monolithic object's behavior is a function of its state, and it must change its behavior at run-time depending on that state
- An application is characterized by many case statements that dictate the flow of control, based on the state of the application
- How can the behavior depend on state?

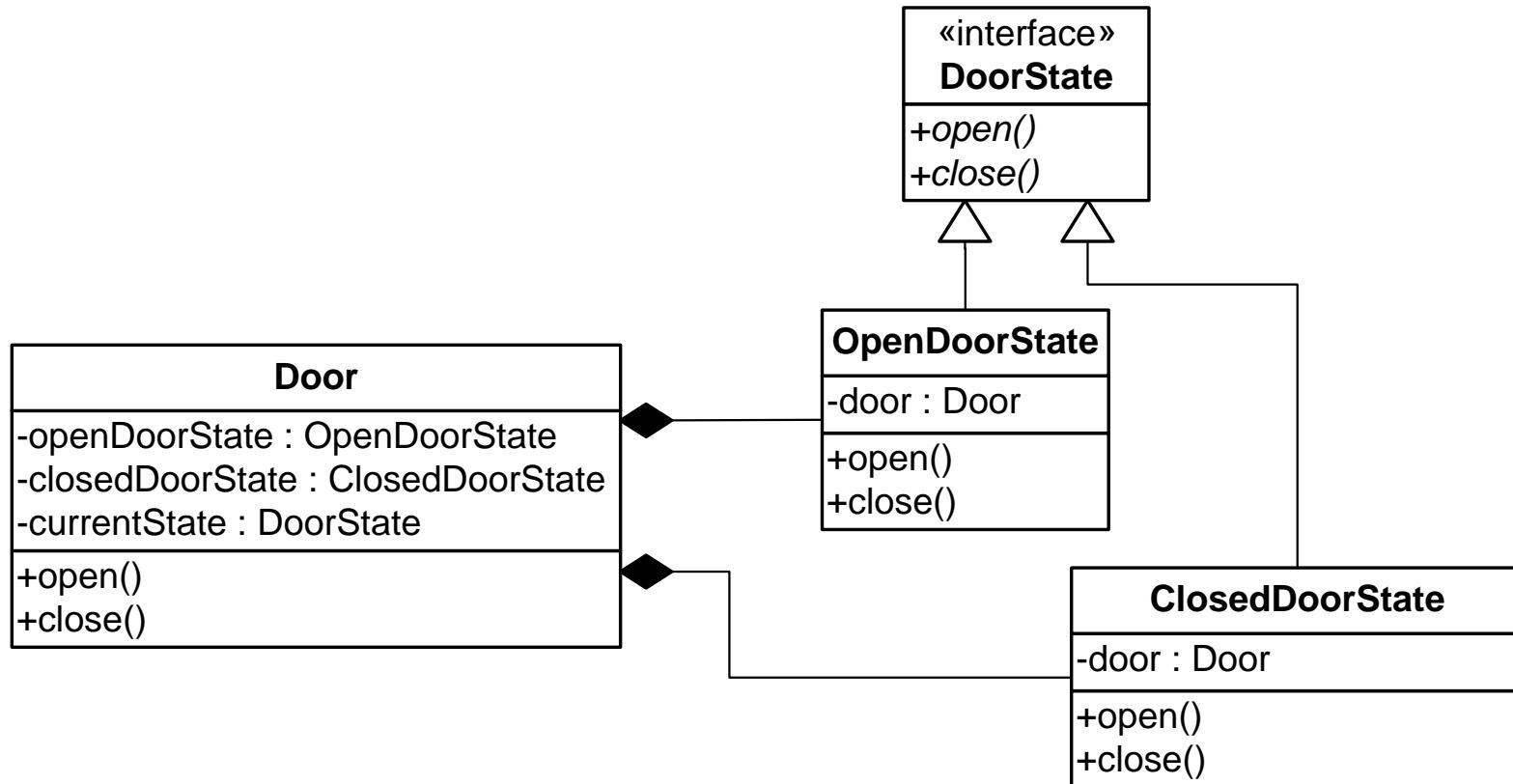
# State – The Solution

- Define a "context" class to present a single interface to the outside world
- Define a State abstract base class
- Represent the different "states" of the state machine as derived classes of the State base class
- Define state-specific behavior in the appropriate State derived classes
- Maintain a reference to the current "state" in the "context" class
- Change the state of the state machine (and the behavior) by changing the current "state" reference

# State – The Solution



# State – Solution Example



# State – Solution Example

---

```
1 public class Door {  
2     private DoorState currentState;  
3     private DoorState closedDoorState;  
4     private DoorState openDoorState;  
5  
6     ...  
7  
8     public void open() {  
9         currentState.open();  
10    }  
11  
12    public void close() {  
13        currentState.close();  
14    }  
15 }
```

---

```
1 public class ClosedDoorState implements DoorState {  
2     ...  
3  
4     public void open() {  
5         System.out.println("Opening closed door now...");  
6         door.setCurrentState(door.getOpenDoorState());  
7     }  
8  
9  
10    public void close() {  
11        System.out.println("Can't close door - it is closed already");  
12    }  
13 }
```

---

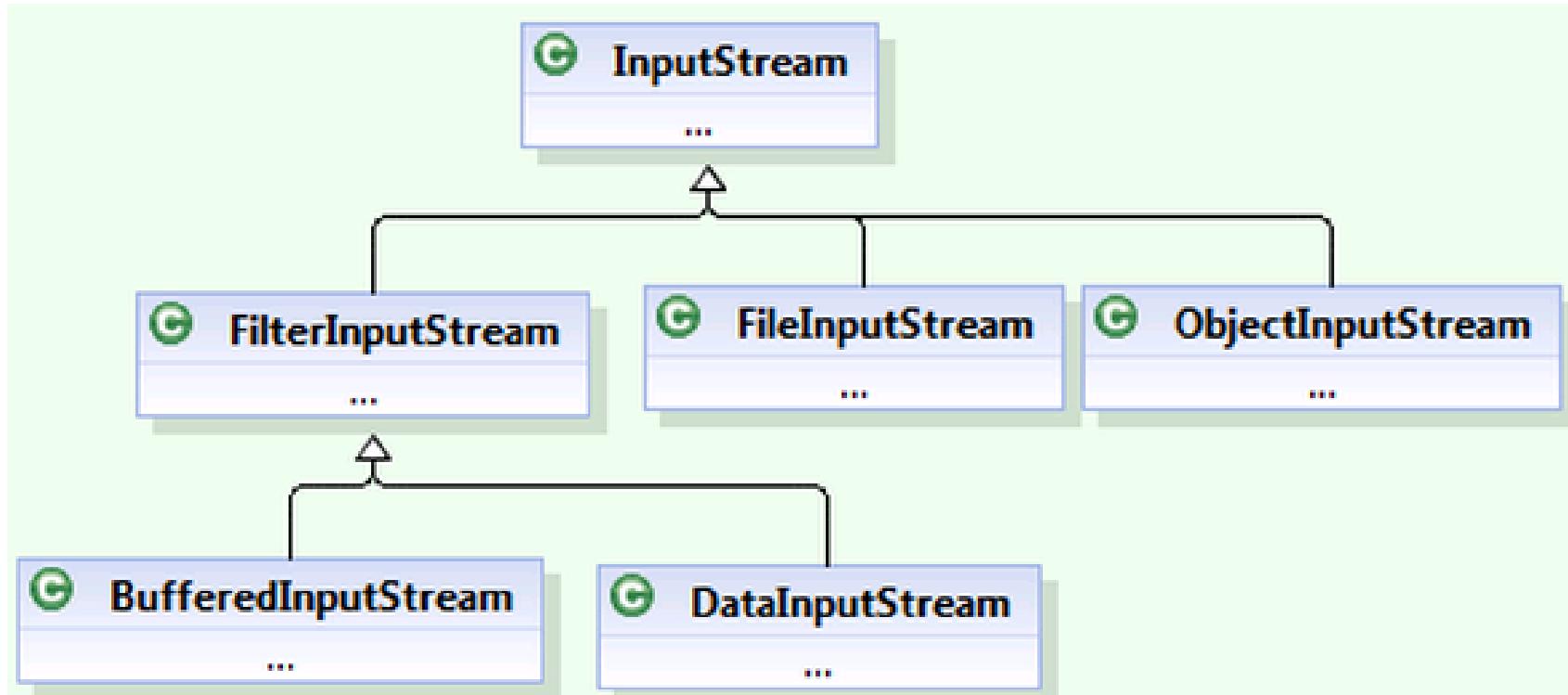
# Задание

- Напишите несколько классов имплементирующих интерфэйс EmailValidator
- AmpersandValidator, BlackListValidator, DotValidator и.т.п.
- Напишите класс BaseValidator который будет проверять, что мэйл не пустой
- Придумайте инфраструктуру, которая даст возможность легко комбинировать между различными валидациями, но BaseValidator будет при любом раскладе
- One who sends mail can mix how much he wants validators (the BaseValidator is mandatory)
- In order to check email validity only one validate method can be called

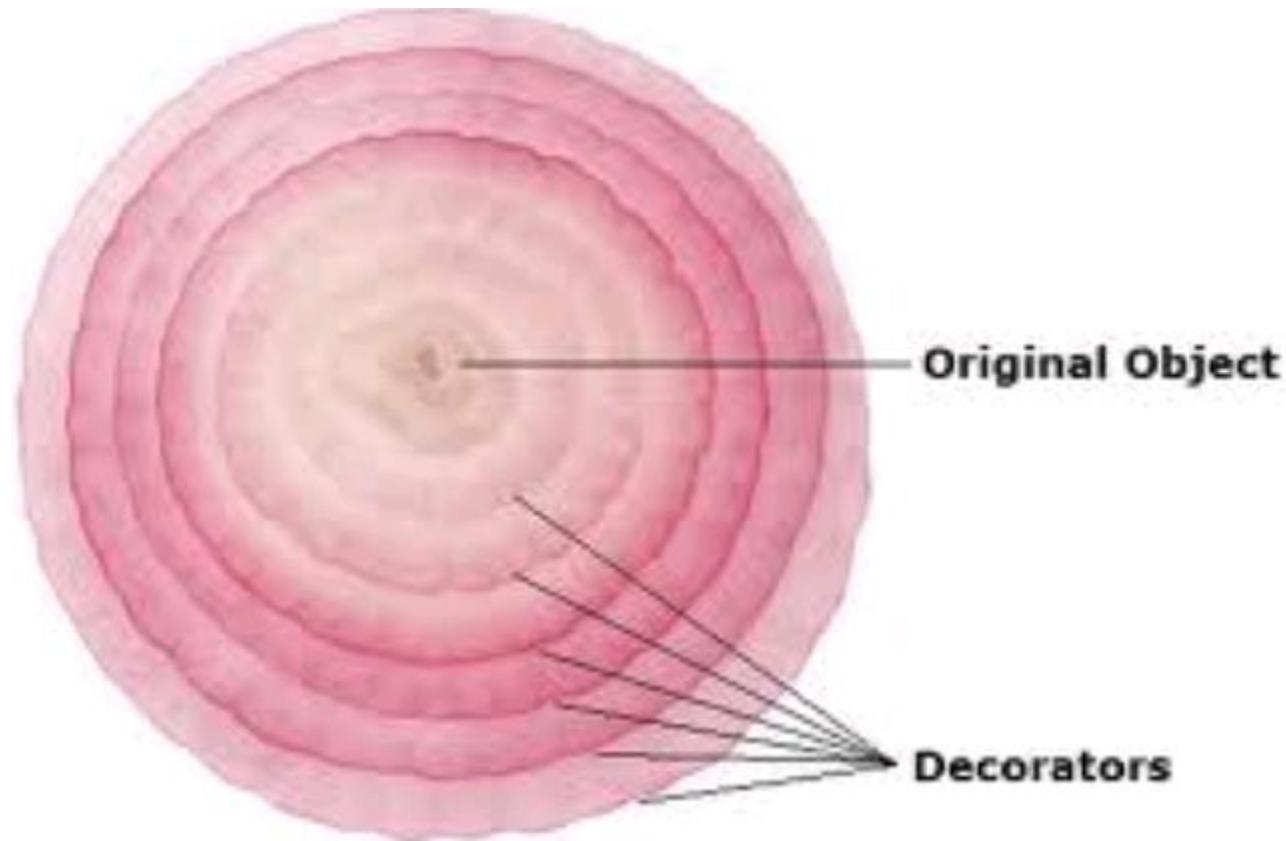
# Задание

- Напишите несколько классов имплементирующих интерфэйс EmailValidator
- AmpersandValidator, BlackListValidator, DotValidator и.т.п.
- Напишите класс BaseValidator который будет проверять, что мэйл не пустой
- Придумайте инфраструктуру, которая даст возможность легко комбинировать между различными валидациями, но BaseValidator будет при любом раскладе
- One who sends mail can mix how much he wants validators (the BaseValidator is mandatory)
- In order to check email validity only one validate method can be called

# Decorator Pattern



# Decorator Pattern



# Model View Controller

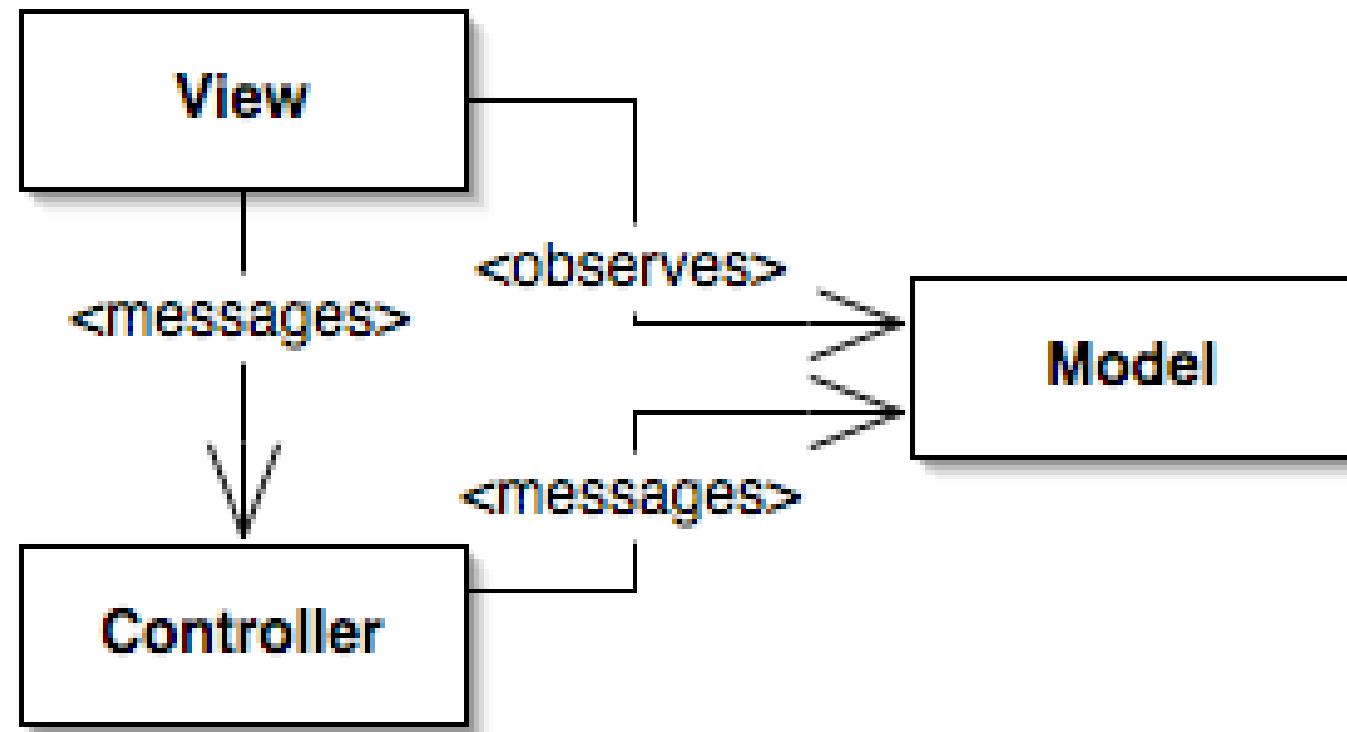
# MVC architecture

- Придуман в компании ксерокс в 1978 году.
- Идея как всегда в разделении компонентов
- M – модель отвечает за данные (Работа с Dao это тоже часть Модели)
- V – view отвечает за выдачу данных на UI (включает фильтры, рендеринг...)
- С – Контроллер – связывает модель и view, отвечает за бизнес логику, решает какую модель будет показывать какой view

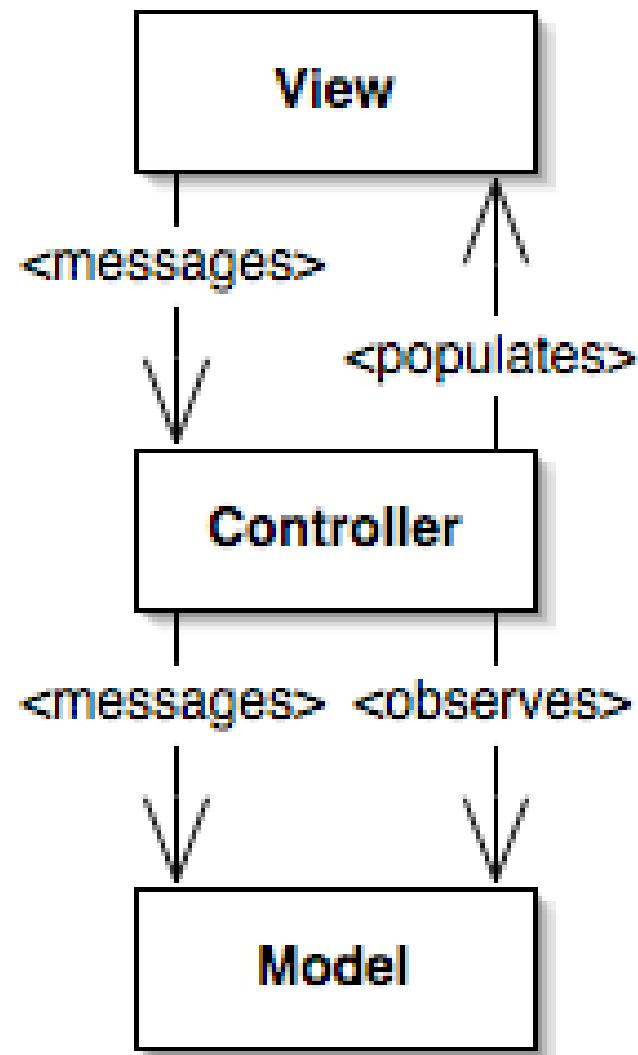
# MVC – Пример: Файловая система винды

- Различные view – Explorer, command line, Total Commander
- Model – Информация о файлах на диске
- Controller – NTFS (NT File System)

# MVC – The Smalltalk Way



# MVC – The Web Event-driven Way



# MVC and Design Patterns

- View и Контроллер имплементируют дизайн паттерн Strategy
  - Когда что-то происходит во view, то он делегирует контроллеру
  - Контроллер решает что делать
- View имплементирует композит
  - Потому что держит в себе кучу всяких компонентов, кнопок и окон
  - И когда контроллер говорит, что надо апдейтиться, то view может заставить это сделать всех
- View имплементирует Observer
  - Он может обзэрвировать модель и отображать ее изменения
  - Несколько разных view могут обзэрвировать одну модель

# MVC – Example – Java Swing

- Давайте вспомним старый недобрый Swing и implementируем MVC
- Сделайте игру «угадай число»
- Компьютер загадывает, человек отгадывает
- По окончанию выдаётся сообщение сколько попыток было сделано,
- И предлагается сыграть еще раз

# Task

- You have a repository with huge list of Employee objects
- Repository knows to retrieve employees by blocks.
- Each employee has next attributes:
  - Name, salary, companyName
- Write Map Reduce in order to calculate how much each company spend for salaries

# Map Reduce

- MapReduce is a design pattern for processing parallelizable problems across huge datasets using a large number of computers (nodes), collectively referred to as a cluster

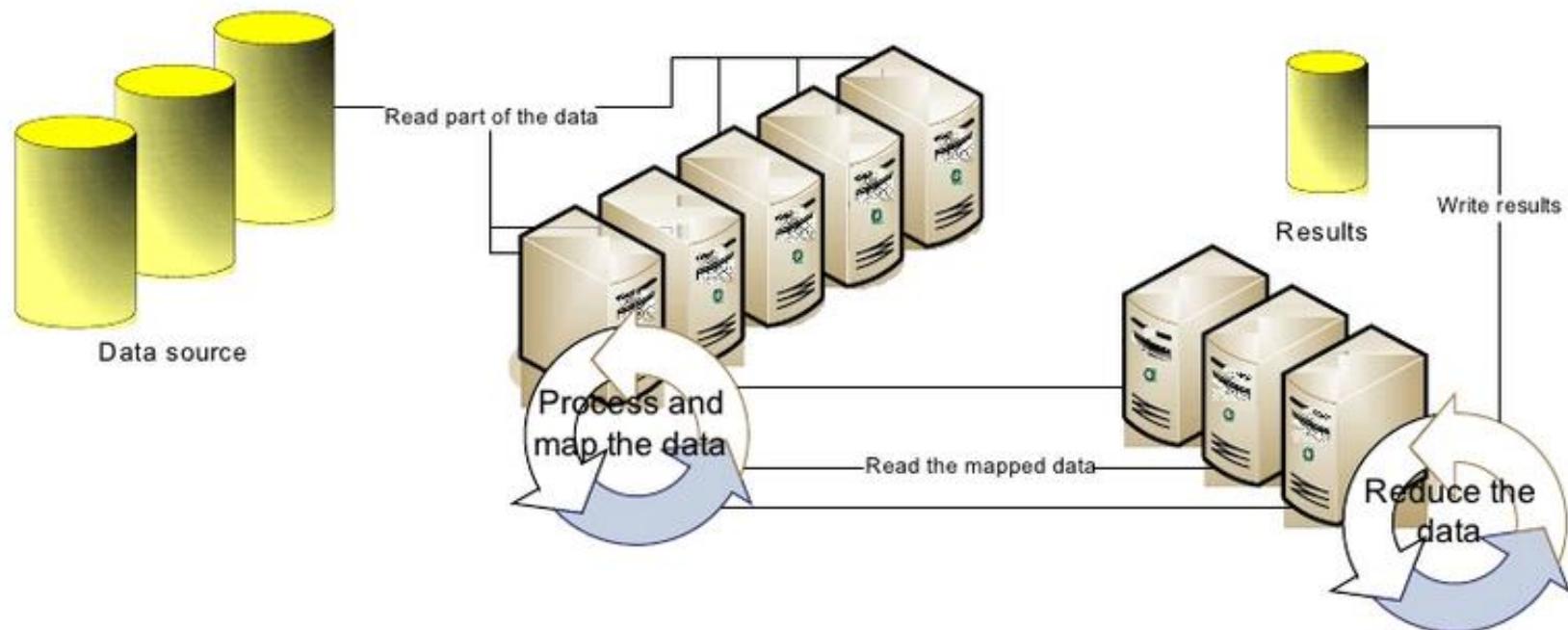
# Map Reduce

- "**Map**" step: The master node takes the input, divides it into smaller sub-problems, and distributes them to worker nodes.
- (or each node takes the input from some repository)
- The worker node processes the smaller problem, and passes the answer back to its master node.

# Map Reduce

- "**Reduce**" step: The master node then collects the answers to all the sub-problems and combines them in some way to form the output – the answer to the problem it was originally trying to solve.
- Another option that each reducer will take the answer from mapper by himself

# Map Reduce



# Map reduce

- Google uses map reduce implementation to analyze the internet
- Facebook uses map reduce
- NoSql DB built on map reduce pattern

# Пример – считаем ругательства



Вася выбирает язык программирования на основании количества ругательств используемых против этого языка в различных статьях и блогах

# Пример

- Каждый мэпер получает «страницу» и ищет в ней предложения в которых содержаться названия языков программирования
- И для каждого языка он смэпирует все ругательные слова против этого языка
- Редьюсер получает от мэпера мэп, подсчитывает количество ругательств против каждого языка
- Мастер мерджит все результаты от редьюсеров

# Repository

```
public class EmployeeRepository<T> implements Repository<Employee> {
    public final static EmployeeRepository EMPLOYEE_REPOSITORY = new EmployeeRepository();

    private final Map<Integer, List<Employee>> employeesBlocks = new HashMap<Integer, List<Employee>>();

    private EmployeeRepository() {
        employeesBlocks.put(0, Arrays.asList(new Employee("Vasya", "Google", new BigDecimal(10000)),
            new Employee("Alik", "Epam", new BigDecimal(30000))));
        employeesBlocks.put(1, Arrays.asList(new Employee("Petya", "Google", new BigDecimal(10000)),
            new Employee("Lena", "Epam", new BigDecimal(30000)),
            new Employee("Bill", "Microsoft", new BigDecimal(10000))));
        employeesBlocks.put(2, Arrays.asList(new Employee("Masha", "Google", new BigDecimal(10000))));
        employeesBlocks.put(3, Arrays.asList(new Employee("Grisha", "Microsoft", new BigDecimal(30000))));
    }

    public List<? extends Employee> getData(int segment) {
        return employeesBlocks.get(segment);
    }
}
```

# Mapper Implementation

```
public class EmployeeMapper implements Mapper<String, Employee> {
    private final int id;
    private HashMap<String, List<Employee>> mappedData;

    public EmployeeMapper(int id) {
        this.id = id;
        mappedData = new HashMap<String, List<Employee>>();
    }
    public void map() {

        List<? extends Employee> data = EmployeeRepository.EMPLOYEE_REPOSITORY.getData(id);
        for (Employee employee : data) {
            List<Employee> employees = mappedData.get(employee.getCompanyName());
            if (employees == null) {
                employees = new ArrayList<Employee>();
                mappedData.put(employee.getCompanyName(), employees);
            }
            employees.add(employee);
        }
    }

    public Map<String, List<Employee>> getMapperData() {
        return mappedData;
    }
}
```

# Reducer

```
public class CompanyReducer<K, V, M> implements Reducer<String, BigDecimal, Employee> {
    private Map<String, BigDecimal> result = new HashMap<String, BigDecimal>();

    public void reduce(Map<String, List<Employee>> map) {
        Set<String> companies = map.keySet();
        for (String companyName : companies) {
            List<Employee> employees = map.get(companyName);
            BigDecimal total = new BigDecimal(0);
            for (Employee employee : employees) {
                total = total.add(employee.getSalary());
            }
            result.put(companyName, total);
        }
    }

    public Map<String, BigDecimal> getFinalResults() {
        return result; //To change body of implemented methods use File | Settings | Fi.
    }
}
```

# Master implementation

```
final CountDownLatch latch = new CountDownLatch(4);
ExecutorService service = Executors.newFixedThreadPool(4);
final HashMap<String, BigDecimal> finalResults = new HashMap<String, BigDecimal>();
for (int i = 0; i < 4; i++) {
    final EmployeeMapper mapper = new EmployeeMapper(i);
    final CompanyReducer reducer = new CompanyReducer();
    service.submit(new Runnable() {
        public void run() {
            mapper.map();
            reducer.reduce(mapper.getMapperData());
            Map<String, BigDecimal> results = reducer.getFinalResults();
            Set<String> companies = results.keySet();
            for (String company : companies) {
                BigDecimal total = finalResults.get(company);
                if (total == null) {
                    total = results.get(company);
                    finalResults.put(company, total);
                } else {
                    finalResults.put(company, finalResults.get(company).add(total));
                }
            }
            latch.countDown();
        }
    });
}
```