

11. OOP LEVEL UP

Паттерны проектирования *(intro)*

Информатика, ИТИС, 1 курс 2 семестр

М.М.Абрамский

2017

План

- ✓ Как могут быть связаны классы друг с другом?
 - ✓ Классификация, примеры, иллюстрация
- Как правильно проектировать приложения с точки зрения классов
 - ✓ Концепции и правила
 - **Конкретные решения**

Прописные истины

Каждый разработчик должен понимать, что за него и до него успели уже подумать другие разработчики.

- Скажем «нет!» самописному коду!
- Скажем «ура!» готовому: приемам, подходам, шаблонам, парадигмам, библиотекам и т.п.

Зачем нужно готовое?

- Сокращается время разработки.
- Разработка становится массовой.
 - разработчик быстрее понимает логику другого разработчика, опирающегося на те же стандарты, приемы, библиотеки.
- Уменьшается возможность ошибки при проектировании и конструировании

Что такое готовое

- Не только библиотеки, алгоритмы, фреймворки
- Но и приемы, рекомендации, подходы
 - Что-то такое в прошлом семестре мы видели
 - TOP-DOWN, «долой хардкод», Code Conventions и т.п.

Example #1

```
class Shotgun {  
    public void shoot() {  
        // boom  
    }  
}  
  
class Cannon {  
    public void shoot() {  
        // boooooooooooooom  
    }  
}
```

Example #1

ShotGun [] weapons =

... ЭЭЭ ...

Cannon [] weapons =

... ХММ ...

Как надо было все делать

- Пушку и ружье делать наследниками абстрактного класса `ShootingWeapon` с абстрактным методом `shoot()` (интерфейс тоже сгодился бы),
- ```
ShootingWeapon [] weapons = new ShootingWeapon[...];
```

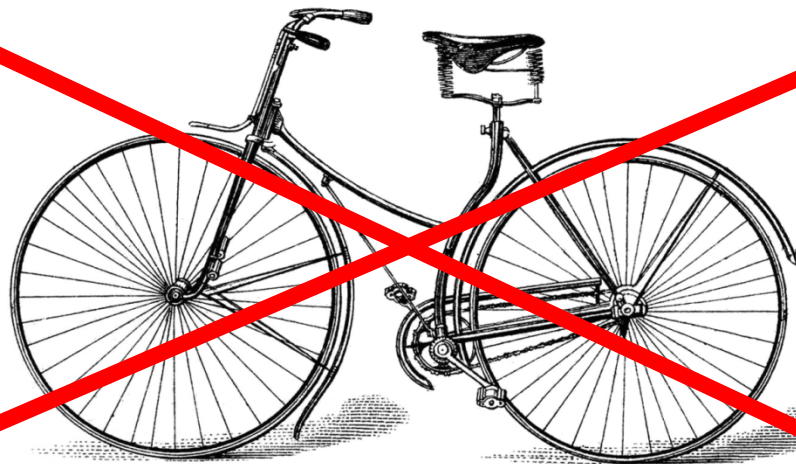


Еще на первом слайде все  
поняли, как и что нужно было  
делать.

# Потому что мы – крутые

- *У нас в голове есть заложенный способ решения данной проблемы (т.к у нас есть опыт).*
  - способ,
  - прием,
  - шаблон,
    - *по которому мы правильно проектируем наше решение.*
- *Шаблон (eng) - **Pattern***

# Цель



# Уточненная цель



Программируем, как умеем!

# Самая хорошая цель



# Вот правильный герб



# Ok, folks, serious business...

- На прошлой паре мы разбирали SOLID-принципы хорошего проектирования классов
- Когда мы придумываем классы в приложении
  - Мы можем нарушить эти принципы
  - Долго придумывать решение
- Паттерны проектирования помогут нам **быстро** применять **хорошие** способы проектирования.

# Паттерн

- Имя
- Группа (тип)
- Назначение
  - В каких случаях надо применять
- Решение
  - Как применять



# **Words with alternative meanings**

Abstract Factory, Singleton, Prototype, Builder, Factory Method, Adapter, Decorator, Proxy, Composite, Bridge, Flyweight, Facade, Interpreter, Iterator, Command, Observer, Visitor, Mediator, Состояние State, Strategy, Memento, Chain of Responsibility, Template Method

# Новый смысл слов (rus)

Абстрактная фабрика, Одиночка, Прототип,  
Строитель, Фабричный метод, Адаптер,  
Декоратор, Заместитель, Компоновщик, Мост,  
Приспособленец, Фасад, Интерпретатор,  
Итератор, Команда, Наблюдатель, Посетитель,  
Посредник, Состояние, Стратегия, Хранитель,  
Цепочка обязанностей, Шаблонный метод

# Примеры

- Представим себе, что проектируем RPG
- Вспомним старый проект с системой управления договорами
- ...

# Дисклеймер

- Примеры игрушечные
  - И искусственные
    - Не надо кричать «вообще-то можно было сделать проще»
- Можете представлять их себе в виде классов слоя бизнес-логики приложения
- Буду приводить либо реализацию, либо UML-диаграмму

# Boss



- В игре – 1 главный злодей.
- Появляется он не сразу (очевидно) –
  - Предположим, что его создание зависит от ваших результатов в течение игры
  - Ему переходят все виды побежденных вами врагов на уровне.
  - Нельзя заранее создать его в начале игры.

```
public class Game {
 private Player player;
 private EvilBoss evilBoss;

 public Game() {
 player = new Player();
 // no boss here
 }

 public void gameProcess() {
 //...
 player.attack(evilBoss);
 //...
 }
}
```

```
public class Player {
 public void attack(EvilBoss evilBoss) {
 // ки-я!... а evilBoss вообще создан?
 }
}
```

А если нет – что может случиться?

# Пытаемся исправить ситуацию

```
public class Player {
 public void attack(EvilBoss evilBoss) {
 if (evilBoss == null) {
 evilBoss = new EvilBoss(...);
 }
 // ...вот теперь точно ки-я!
 }
}
```

Все ли в порядке?



# Юк!

- Создание EvilBoss зависит от побежденных врагов
  - Получается, Player должен их знать, чтобы передать в конструктор EvilBoss.
    - Лишняя зависимость
- *Если в игре есть уровни, где несколько боссов, то есть наверное суперкласс (интерфейс) Boss (IBoss), и тогда получается*

```
public void attack(Boss boss) {
 if (evilBoss == null) {
 evilBoss = new EvilBoss(...);
 }
}
```

Что это?

# Решение

- Спрятать всю логику создания босса в evilBoss.
- Вернемся к тому, что у класса – один объект, но мы не хотим, чтобы Player задумывался о создании злодея.
  - Логично, чо.

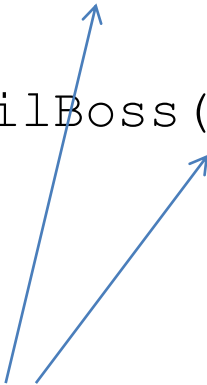
# Singleton (Одиночка)

- Класс должен иметь только один экземпляр.
- *Примеры:*
  - Столица – класс, но у конкретной страны – одна столица.
  - Если мы развернули систему управления договорами, то хранилище у нас в этой системе – одно.
  - Main Evil Guy (Boss)

# Решение

```
public class EvilBossSingleton {

 private static EvilBoss evilBoss = null;
 public static EvilBoss getEvilBoss(...) {
 if (evilBoss == null)
 evilBoss = new EvilBoss(...);
 return evilBoss;
 }
}
```



Вот здесь все, что нужно, для создания EvilBoss

# Перейдем на мелких приспешников

Канон: вы подходите к замку/к дереву/к  
крепости/в комнату/в зал/в лифт/...

...и вас атакуют...

... пехотинцы / лучники / монстры /  
животные/ злые маги /...

# Упростим

- Возьмем для примера два вида плохих парней:
  - Одни стреляют в вас из лука
    - назовем их **злыми эльфами**
  - Другие подбегают и дубасят вас молотом
    - назовем их **злыми гномами**

# Не буду загадывать загадки

- Всем очевидно, что оба типа плохих парней – наследники общего подкласса

```
abstract public class EvilGuy {
 abstract public void attack(Player p);
}
```

```
public class EvilElf extends EvilGuy {
 public void attack(Player p) { ... }
}
```

```
public class EvilDwarf extends EvilGuy {
 public void attack(Player p) { ... }
}
```

# Сражение в коде

```
public void gameProcess() {

 //...
 while (battleIsGoing) {
 String who = randomBadGuy.generate();
 if (who == "elf") {
 badGuys.add(new EvilElf());
 } else if (who == "dwarf") {
 badGuys.add(new EvilDwarf());
 }
 //...
 }
}
```



**КТО ЗАМЕТИЛ ОЧЕНЬ  
ГЛУПЫЕ ОШИБКИ?**

# Сражение в коде

```
public void gameProcess() {

 //...
 while (battleIsGoing) {
 String who = randomBadGuy.generate();
 if (who.equals("elf")) {
 badGuys.add(new EvilElf());
 } else if (who.equals("dwarf")) {
 badGuys.add(new EvilDwarf());
 }
 //...
 }
}
```

Все равно фу...

# Сражение в коде

```
public void gameProcess() {

 //...
 while (battleIsGoing) {
 String who = randomBadGuy.generate();
 if (who.equals("elf")) {
 badGuys.add(new EvilElf());
 } else if (who.equals("dwarf")) {
 badGuys.add(new EvilDwarf());
 }
 //...
 }
}
```

Если будет новый злодей – придется лезть в код Game и менять его

Чем он провинился? Его и так все время меняют!

Враги все равно для него все одинаковые.

**Что нарушается?**

# Factory Method (фабричный метод)

Метод в родительском классе.

Создает экземпляры класса, при этом реализация определяется в подклассах, будучи зависимой от того, кто создает

– А не наоборот!

# Решение

```
abstract public class EvilGuy {
 abstract public void attack(Player p);
 public static EvilGuy getNext(String who) {
 if (who.equals("elf")) {
 return new EvilElf();
 } else if (who.equals("dwarf")) {
 return new EvilDwarf();
 } else {
 return null;
 }
 }
 public static String generate() {
 // генерируем
 }
}
```

# Решение

```
public void gameProcess() {

 //...
 while (battleIsGoing) {
 String who = EvilGuy.generate();
 badGuys.add(EvilGuy.getNext(who));
 //...
 }
}
```

Если будет новый враг – идем менять EvilGuy  
(логично - сам виноват, наплодил наследников, еще и плохих парней)

# Singleton & Factory Method

- Метод у синглтона – тоже реализация паттерна «Фабричный метод»
  - Просто новые объекты не создаются.

# Что общего?







*Злодей говорит: «Я нашел способ самокопирования!»  
Вжух! И сразу появилось его 100 копий.*

Фабричный метод – слишком общее решение.

# Prototype (прототип)

- Назначение: создание новых экземпляров класса по некоторому образцу (прототипу), при этом идеал может определяться подклассом.
- Примеры:
  - `clone()` в `Cloneable`,

# Реализация

```
class EvilBoss implements Cloneable {
 public EvilBoss clone() throws CloneNotSupportedException {
 return (EvilBoss) super.clone();
 }
}

class CopyPasteMachine{ // кто ответственный за копирование
 private EvilBoss evilBoss;
 public CopyPasteMachine(EvilBoss e) { this.evilBoss = e; }
 public EvilBoss copyPaste() throws CloneNotSupportedException {
 return (EvilBoss) this.evilBoss.clone();
 }
}
```

... где-то в gameProcess ...

```
CopyPasteMachine cpm = new CopyPasteMachine(ideal);
```

```
// а-ха-ха-ха-ха-ха!
```

```
EvilBoss twin = cpm.copyPaste();
```

*Прототип – тоже... что?*

# Еще раз про генерацию плохих парней

- *Вы пришли на уровень 1 –и там боретесь с плохими эльфами и гномами*
  - *Вы перешли на уровень 2 – и там боретесь с плохими личами и пандами*
  - *Вы перешли на уровень 3 – и боретесь там с зергами и протосами*
  - *На уровне 4 – злые кролики и злые фламинго*
- ...

**И для каждого из уровней использован FactoryMethod. Что смущает?**

# Генерация генерации

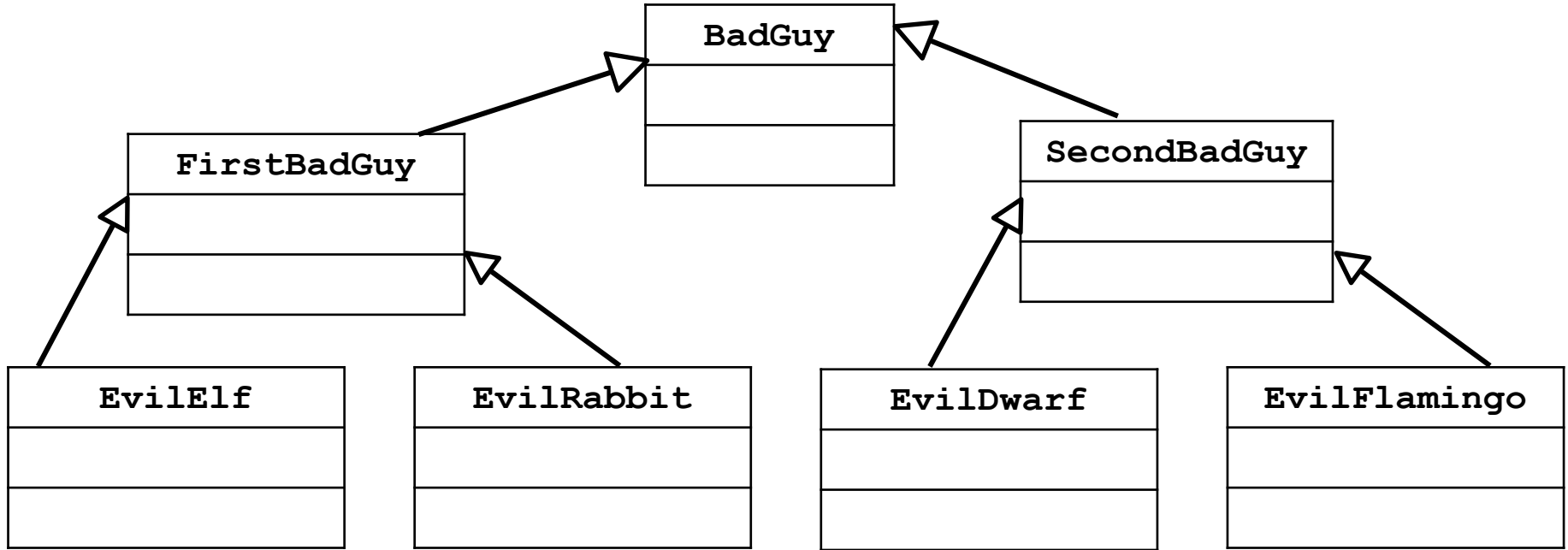
- Способ генерации во всех случаях однотипен!
  - Различаются лишь классы врагов!
- Значит сам способ генерации тоже можно сгенерировать по общему шаблону!
  - Генерация однотипных способов, которые генерируют *однотипных* врагов.

# Abstract Factory

## (Абстрактная фабрика)

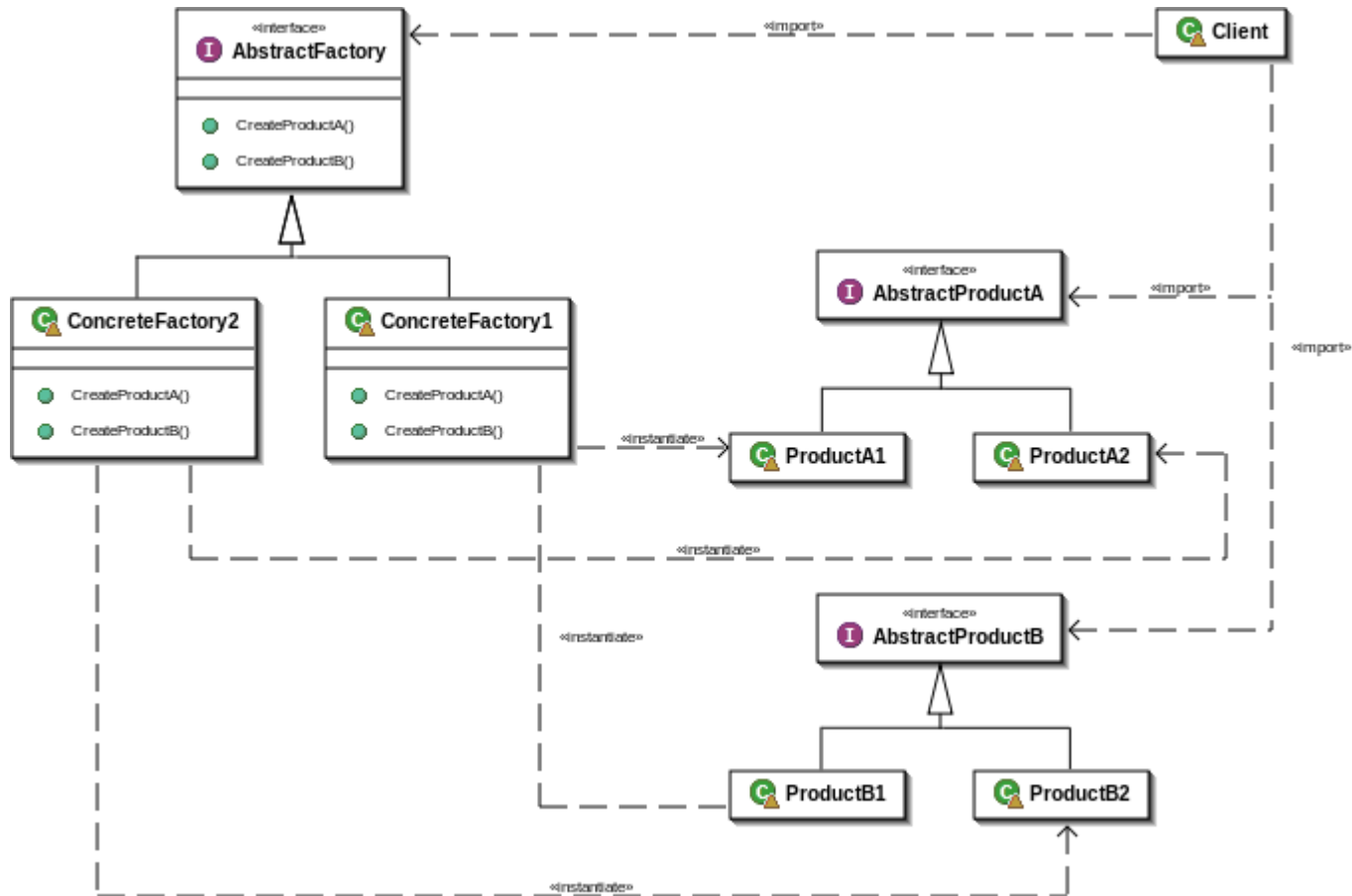
- Генерация целое семейств объектов, при этом их конечный тип неизвестен!
  - На первом уровне будут генерировать эльфы и гномы – но мы не знаем, в какой последовательности.

# Изменения



Не привязываемся к уровням,  
эти товарищи могут появиться все на одном и том же уровне  
Фишка в том, кто с кем атакует в тот или иной момент времени.

# Abstract Factory - UML





# Решение

```
public abstract class AbstractBadGuyFactory {
 public abstract FirstBadGuy createFirstBadGuy();
 public abstract SecondBadGuy createSecondBadGuy();
}

public class FantasyFactory extends AbstractBadGuyFactory {
 public FirstBadGuy createFirstBadGuy() {
 return new EvilElf();
 }

 public SecondBadGuy createSecondBadGuy() {
 return new EvilDwarf();
 }
}
```

# Решение

```
public abstract class AbstractBadGuyFactory {
 public abstract FirstBadGuy createFirstBadGuy();
 public abstract SecondBadGuy createSecondBadGuy();
}
```

```
public class FantasyFactory extends AbstractBadGuyFactory {
 public FirstBadGuy createFirstBadGuy() {
 return new EvilElf();
 }
}
```

```
 public SecondBadGuy createSecondBadGuy() {
 return new EvilDwarf();
 }
}
```

```
}
```

Что плохого в решении?

# Проблема

- Да, если вы решите, что нужно будет на разных уровнях разное количество злодеев или просто захотите добавить новых, придется много переписывать
  - Но у нас есть решение!
  - `generate()` – и мы гибко можем, меняя перечень врагов, гибко их генерировать.
    - *Не будем подробно останавливаться*

# Вспоминаем badGuys.add

- Создаваемые враги где-то хранятся – в наборе врагов
  - Мы еще ничего не знаем о коллекциях, не шумите (некоторые).
- Мы шарахнули по ним файрболлом
- Нужна проверка – по кому попало.
  - Надо пройти по набору врагов

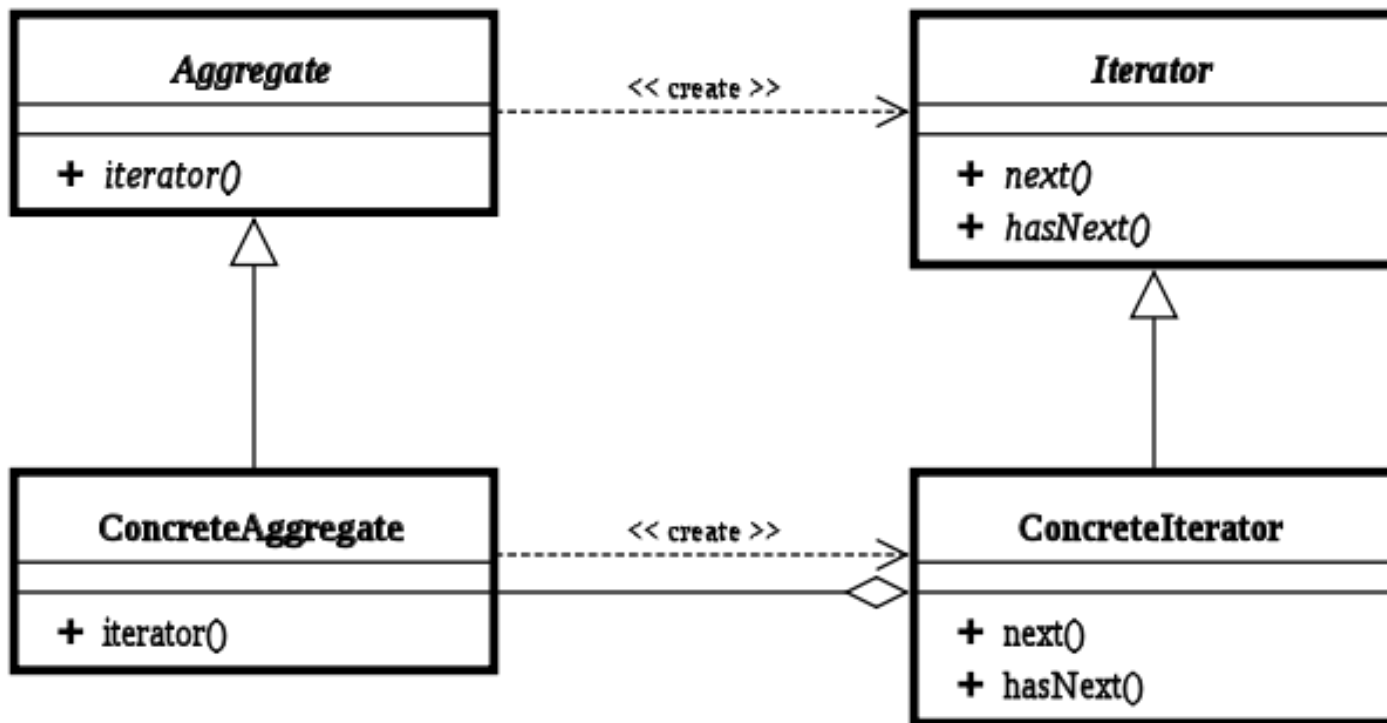
# **Единственный набор, которую мы знаем - массив**

- Но массив – слишком плоско, примитивно и про реализацию. Все на виду, никакой инкапсуляции.
- Чтобы пройти по набору элементом, нужны два метода:
  - Дай очередной элемент.
  - Есть ли еще?

# Iterator (Итератор)

- Назначение: последовательный обход всех элементов.
- В Java реализован полноценно как часть языка.

# Iterator – UML диграмма



# Решение

```
Iterator i = badGuys.iterator();
while (i.hasNext()) {
 EvilGuy item = i.next();
 // делаем что угодно с item
}
```



# Еще раз про Java

- Набор объектов – коллекция
- Все коллекции в Java имеют итератор
  - Реализуют интерфейс Iterable
  - Пройдем вместе с коллекциями
- Благодаря итератору работает цикл `for each!!`

# Ваши враги эволюционировали

- Раньше было все понятно – гномы атакуют топором, эльфы – стрелами, кролики – ушами.
  - Поведение при атаке напрямую привязывалось к объекту.
- А теперь появились враги, которые могут подбирать с земли любое оружие, и атаковать им (и луком, и мечом, например)
  - Хаха, да вы и сами так делаете, у вас целый инвентарь с собой размером с дом (вспоминайте похожие игры).

# Суть атаки

- Реализация метода `attack` теперь не может быть привязана к классам `Player` или `BadGuy`, т.к. есть необходимость гибко ее менять
  - Какое оружие подберем – так и атакуем.
  - А еще мы должны уметь пользоваться тем или иным видом оружия.
    - При этом разные персонажи могут использовать разные наборы оружия.
      - » Т.е. у них есть разный набор поведения, т.е. чего?

# Наследование уже не вариант

Нельзя реализовывать разные наборы действий (только древовидная иерархия) для схожих объектов.

## **Пример:**

Поведения: ударить мечом, выстрелить из лука, использовать фэйрболл.

Допустим, что воин-пехотинец должен уметь первые два действия, а воин-маг – первое и третье.

# Но

- Мы можем реализовать кучу разных интерфейсов
  - Действительно – каждое поведение интерфейс, множественная реализация интерфейса
  - Но что делать с реализацией – телами методов?
    - Код будет вынужденно дублироваться!
- Да здравствует композиция!
  - и еще паттерн

# Strategy (Стратегия)

Определяет семейство алгоритмов, инкапсулируя их все и позволяя подставлять один вместо другого. Можно менять алгоритм независимо от клиента, который им пользуется.

*Поведение при действии тем или иным видом оружия – отдельный класс!*

# Реализация

```
abstract class ArmoryStrategy {
 abstract void attack();
}

class SwordStrategy extends ArmoryStrategy {
 void attack() { ... }
}

class BowStrategy extends ArmoryStrategy {
 void attack() { ... }
}

public class WarriorSoldier {
 private ArmoryStrategy currentArmoryStrategy;
 public void setCurrentArmory(ArmoryStrategy a) {
 currentArmoryStrategy = a;
 }
 public void attack() {
 currentArmoryStrategy.attack();
 }
}
```

Реализация нужного поведения –  
вопрос управления интерфейсом  
Warrior

ArmoryStrategy должен знать  
о вас, о враге и о типе оружия  
– это уже частность, никаких  
проблем не должно быть.

# Тонкости Legacy

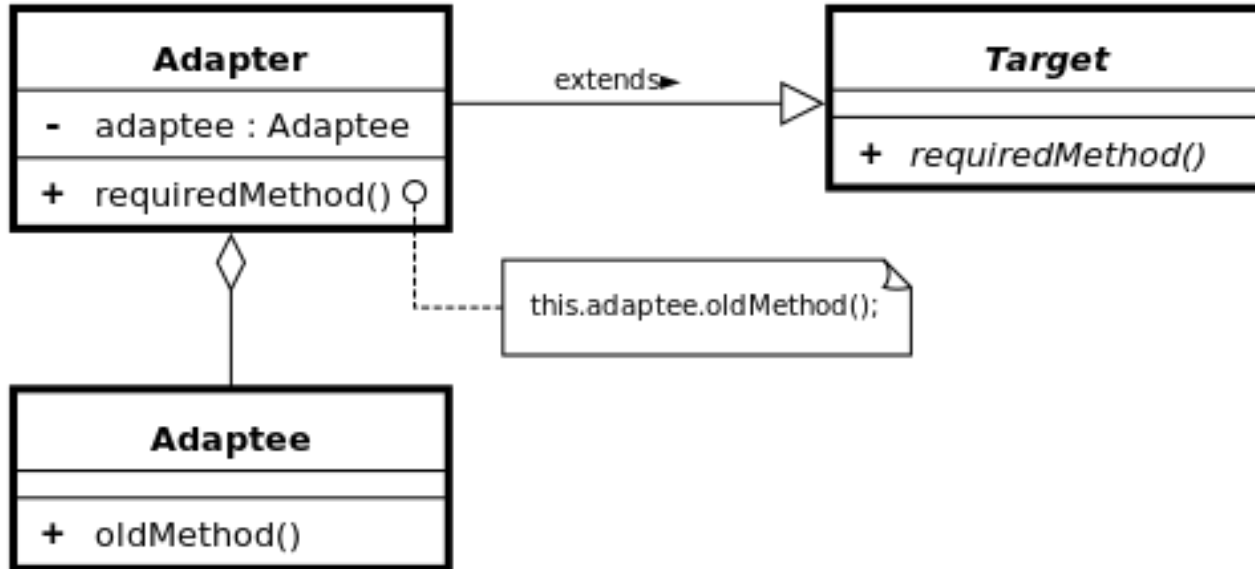
- Вы купили компанию, в которой тоже была своя RPG. Там был реализован класс оружие – который вам необходим для игры
  - Но у этого класса совершенно другой интерфейс.
    - У вас `Sword.attack()`
    - У него `MegaWeapon.kick()`
- Переписывать не вариант – отвалится та часть их игры, которую вы хотели не менять.
  - Как быть, если хочется использовать этот класс?



# Adapter (Адаптер)

- Преобразует интерфейс класса в некоторый другой необходимый интерфейс.
- Обеспечивает совместную работу классов, которая была бы невозможна без данного паттерна из-за несовместимости интерфейсов.

# Adapter – UML



# «Жизненно»

- Какой-то там волшебник взял и сделал ваш меч магическим – теперь им можно залечивать раны
  - Новое поведение
  - Новые методы
- Но наследоваться мы не можем – объект уже существует.
  - И он все еще меч.

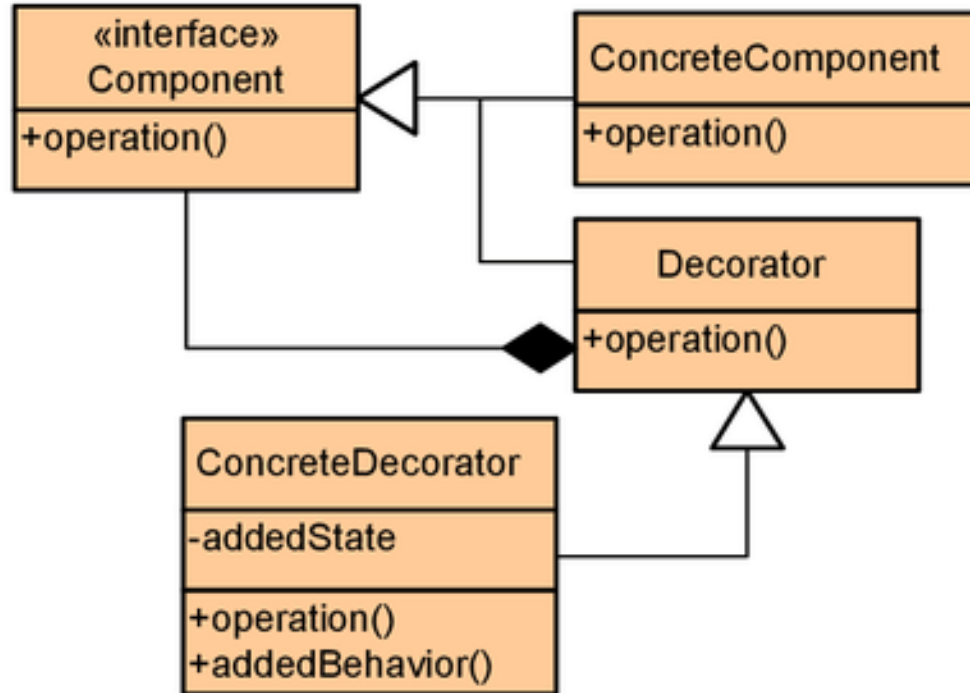
# «Жизненно»

- Крутой меч – тоже меч.
- Но не любой меч – крутой.
- Точное определение наследование, но наследовать нельзя. Как быть?

# Decorator (Декоратор)

*Динамически возлагает на объект новые функции. Декораторы применяются для расширения имеющейся функциональности и являются гибкой альтернативой порождению подклассов.*

# Decorator - UML



# Еще один «фентези» пример

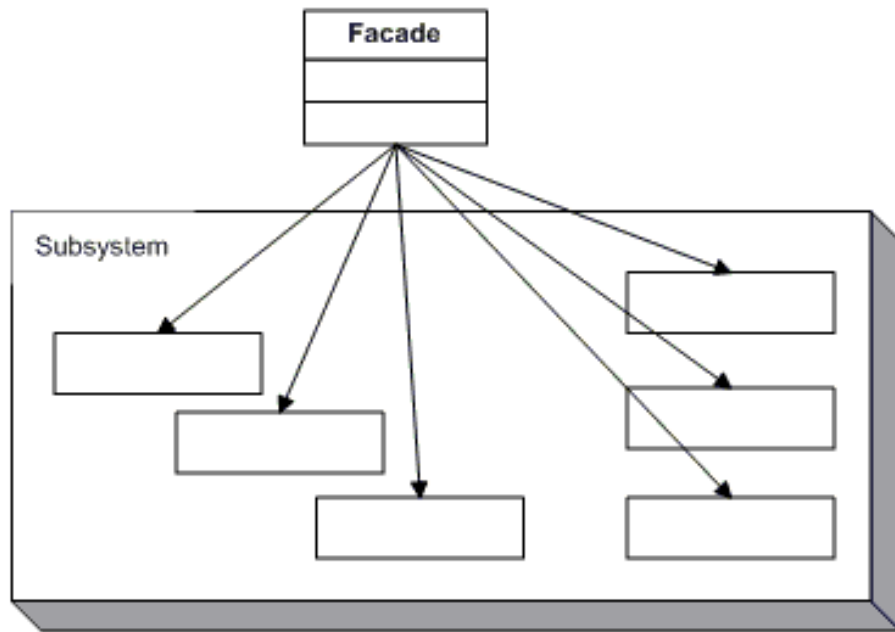
- Вы пришли в лавку (магазин), но там вы можете не только купить-продать товар, но и
  - Починить оружие.
  - Узнать новости.
  - Выпить.
  - Полечиться.
- То, что мы видим – проявление нескольких объектов (а точнее их интерфейсов) за единой вывеской

# Facade (Фасад)

Предоставляет унифицированный интерфейс к множеству интерфейсов в некоторой подсистеме. Определяет интерфейс более высокого уровня, облегчающий работу с подсистемой.



# Фасад - UML



# 3 забавных паттерна

- Адаптер, Декоратор и Фасад часто путают.
- **А вы не путайте!**
  - Декоратор – новые интерфейсы к существующему объекту без наследования
  - Адаптер – полная замена интерфейса
  - Фасад – скрывание интерфейса нескольких разных объектов за одним общим.

# Классификация паттернов

- Порождающие паттерны
  - Абстрактная фабрика (Abstract Factory), Одиночка (Singleton), Прототип (Prototype), Строитель (Builder), Фабричный метод (Factory Method)
- Структурные паттерны
  - Адаптер (Adapter), Декоратор (Decorator), Заместитель (Proxy), Компоновщик (Composite), Мост (Bridge), Приспособленец (Flyweight), Фасад (Facade)
- Паттерны поведения
  - Интерпретатор (Interpreter), Итератор (Iterator), Команда (Command), Наблюдатель (Observer), Посетитель (Visitor), Посредник (Mediator), Состояние (State), Стратегия (Strategy), Хранитель (Memento), Цепочка обязанностей (Chain of Responsibility), Шаблонный метод (Template Method)
- ...есть еще... (паттерны паттернов и др.)

# Каталог паттернов (часть 1)

## *Copy-paste из книги Э.Гаммы и др.*

- **Abstract Factory (абстрактная фабрика)**
  - Предоставляет интерфейс для создания семейств, связанных между собой, или независимых объектов, конкретные классы которых неизвестны.
- **Adapter (адаптер)**
  - Преобразует интерфейс класса в некоторый другой интерфейс, ожидаемый клиентами. Обеспечивает совместную работу классов, которая была бы невозможна без данного паттерна из-за несовместимости интерфейсов.
- **Bridge (мост)**
  - Отделяет абстракцию от реализации, благодаря чему появляется возможность независимо изменять то и другое.

# Каталог паттернов (часть 2)

## *Copy-paste из книги Э.Гаммы и др.*

- **Builder (строитель)**

- Отделяет конструирование сложного объекта от его представления, позволяя использовать один и тот же процесс конструирования для создания различных представлений.

- **Chain of Responsibility (цепочка обязанностей)**

- Можно избежать жесткой зависимости отправителя запроса от его получателя, при этом запросом начинает обрабатываться один из нескольких объектов. Объекты-получатели связываются в цепочку, и запрос передается по цепочке, пока какой-то объект его не обработает.

- **Command (команда)**

- Инкапсулирует запрос в виде объекта, позволяя тем самым параметризовывать клиентов типом запроса, устанавливать очередность запросов, протоколировать их и поддерживать отмену выполнения операций.

# Каталог паттернов (часть 3)

## *Copy-paste из книги Э.Гаммы и др.*

- **Composite (компоновщик)**
  - Группирует объекты в древовидные структуры для представления иерархий типа «часть-целое». Позволяет клиентам работать с единичными объектами так же, как с группами объектов.
- **Decorator (декоратор)**
  - Динамически возлагает на объект новые функции. Декораторы применяются для расширения имеющейся функциональности и являются гибкой альтернативой порождению подклассов.
- **Facade (фасад)**
  - Предоставляет унифицированный интерфейс к множеству интерфейсов в некоторой подсистеме. Определяет интерфейс более высокого уровня, облегчающий работу с подсистемой.

# Каталог паттернов (часть 4)

## *Copy-paste из книги Э.Гаммы и др.*

- **Factory Method (фабричный метод)**
  - Определяет интерфейс для создания объектов, при этом выбранный класс инстанцируется подклассами.
- **Flyweight (приспособленец)**
  - Использует разделение для эффективной поддержки большого числа мелких объектов.
- **Interpreter (интерпретатор)**
  - Для заданного языка определяет представление его грамматики, а также интерпретатор предложений языка, использующий это представление.
- **Iterator (итератор)**
  - Дает возможность последовательно обойти все элементы составного объекта, не раскрывая его внутреннего представления.

# Каталог паттернов (часть 5)

## *Сору-paste из книги Э.Гаммы и др.*

- **Mediator (посредник)**

- Определяет объект, в котором инкапсулировано знание о том, как взаимодействуют объекты из некоторого множества. Способствует уменьшению числа связей между объектами, позволяя им работать без явных ссылок друг на друга. Это, в свою очередь, дает возможность независимо изменять схему взаимодействия.

- **Memento (хранитель)**

- Позволяет, не нарушая инкапсуляции, получить и сохранить во внешней памяти внутреннее состояние объекта, чтобы позже объект можно было восстановить точно в таком же состоянии.

- **Observer (наблюдатель)**

- Определяет между объектами зависимость типа один-ко-многим, так что при изменении состояния одного объекта все зависящие от него получают извещение и автоматически обновляются.



# Каталог паттернов (часть 6)

## *Copy-paste из книги Э.Гаммы и др.*

- **Prototype (прототип)**
  - Описывает виды создаваемых объектов с помощью прототипа и создает новые объекты путем его копирования.
- **Proxy (заместитель)**
  - Подменяет другой объект для контроля доступа к нему.
- **Singleton (одиночка)**
  - Гарантирует, что некоторый класс может иметь только один экземпляр, и предоставляет глобальную точку доступа к нему.
- **State (состояние)**
  - Позволяет объекту варьировать свое поведение при изменении внутреннего состояния. При этом создается впечатление, что поменялся класс объекта.

# Каталог паттернов (часть 7)

## *Copy-paste из книги Э.Гаммы и др.*

- **Strategy (стратегия)**

- Определяет семейство алгоритмов, инкапсулируя их все и позволяя подставлять один вместо другого. Можно менять алгоритм независимо от клиента, который им пользуется.

- **Template Method (шаблонный метод)**

- Определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.

- **Visitor (посетитель)**

- Представляет операцию, которую надо выполнить над элементами объекта. Позволяет определить новую операцию, не меняя классы элементов, к которым он применяется.

**TO BE CONTINUED**

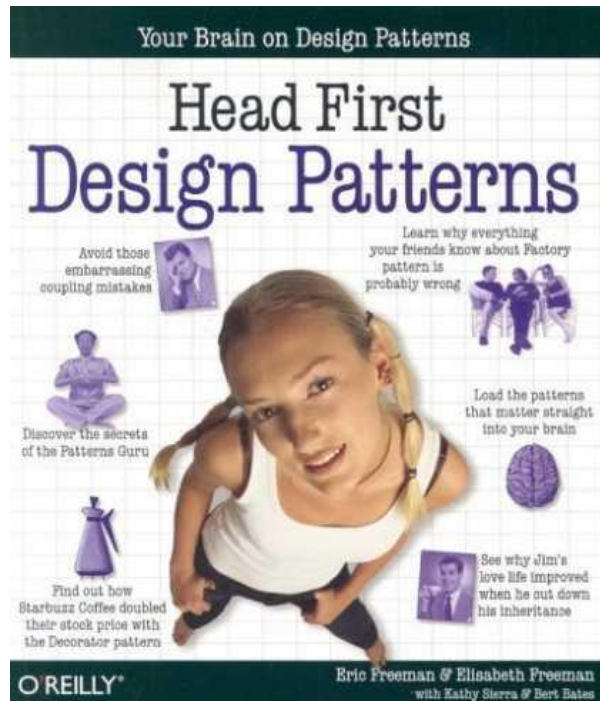
# Словарь

*Абстрактная фабрика, Адаптер, Декоратор,  
Итератор, Паттерн проктирования, Прокси,  
Прототип, Синглтон, Стратегия, Фабричный  
метод, Фасад*

# !!! Go read this

Eric Freeman,  
Elizabeth Freeman  
(with Kathy Sierra  
and Bert Bates)

**Design Patterns**  
*Head First*



# !!! Go read this

Erich Gamma  
Ralph Johnson  
Richard Helm  
John Vlissides  
(банда четырех)

**Design Patterns**  
**Elements of Reusable**  
**Object-Oriented Software**



# Прочитать

- Э.Фримен, Э.Фримен. «Паттерны проектирования» (Head First)
- Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. «Приемы ОО проектирования. Паттерны проектирования»
- <http://design-pattern.ru/> (тут больше паттернов)