**Паттерны**

*вопросы-ответы Хилькевич Игорь 04.04.2020*

Вопросы

1. Что такое «шаблон проектирования»?

2. Назовите основные характеристики шаблонов.

3. Типы шаблонов проектирования.

4. Приведите примеры основных шаблонов проектирования.

5. Приведите примеры порождающих шаблонов проектирования.

6. Приведите примеры структурных шаблонов проектирования.

7. Приведите примеры поведенческих шаблонов проектирования.

8. Что такое «антипаттерн»? Какие антипаттерны вы знаете?

9. Что такое Dependency Injection?

Доп. вопросы

1. Паттерны в spring framework

2. Какие паттерны применяются в Hibernate?

**Ответы на вопросы**

1. Что такое «шаблон проектирования»?

**Шаблон** (паттерн) проектирования (design pattern) — это проверенное и готовое к использованию решение. Это не класс и не библиотека, которую можно подключить к проекту, это нечто большее - он не зависит от языка программирования, не является законченным образцом, который может быть прямо преобразован в код и может быть реализован по разному в разных языках программирования.

Плюсы использования шаблонов:

* снижение сложности разработки за счёт готовых абстракций для решения целого класса проблем.
* облегчение коммуникации между разработчиками, позволяя ссылаться на известные шаблоны.
* унификация деталей решений: модулей и элементов проекта.
* возможность отыскав удачное решение, пользоваться им снова и снова.
* помощь в выборе наиболее подходящего варианта проектирования.

Минусы:

* слепое следование некоторому выбранному шаблону может привести к усложнению программы.
* желание попробовать некоторый шаблон в деле без особых на то оснований.

[https://github.com/enhorse/java-interview/blob/master/patterns.md](https://github.com/enhorse/java-interview/blob/master/patterns.md#%D0%A7%D1%82%D0%BE-%D1%82%D0%B0%D0%BA%D0%BE%D0%B5-%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD-%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)

2. Назовите основные характеристики шаблонов.

* **Имя** - все шаблоны имеют уникальное имя, служащее для их идентификации;
* **Назначение** - назначение данного шаблона;
* **Задача** - задача, которую шаблон позволяет решить;
* **Способ решения** - способ, предлагаемый в шаблоне для решения задачи в том контексте, где этот шаблон был найден;
* **Участники** - сущности, принимающие участие в решении задачи;
* **Следствия** - последствия от использования шаблона как результат действий, выполняемых в шаблоне;
* **Реализация** - возможный вариант реализации шаблона.

[https://github.com/enhorse/java-interview/blob/master/patterns.md](https://github.com/enhorse/java-interview/blob/master/patterns.md#%D0%A7%D1%82%D0%BE-%D1%82%D0%B0%D0%BA%D0%BE%D0%B5-%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD-%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)

3. Типы шаблонов проектирования.

* **Основные** (Fundamental) - основные строительные блоки других шаблонов. Большинство других шаблонов использует эти шаблоны в той или иной форме.
* **Порождающие шаблоны** (Creational) — шаблоны проектирования, которые абстрагируют процесс создание экземпляра. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Шаблон, порождающий классы, использует наследование, чтобы изменять созданный объект, а шаблон, порождающий объекты, делегирует создание объектов другому объекту.
* **Структурные шаблоны** (Structural) определяют различные сложные структуры, которые изменяют интерфейс уже существующих объектов или его реализацию, позволяя облегчить разработку и оптимизировать программу.
* **Поведенческие шаблоны** (Behavioral) определяют взаимодействие между объектами, увеличивая таким образом его гибкость.

[https://github.com/enhorse/java-interview/blob/master/patterns.md](https://github.com/enhorse/java-interview/blob/master/patterns.md#%D0%A7%D1%82%D0%BE-%D1%82%D0%B0%D0%BA%D0%BE%D0%B5-%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD-%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)

4. Приведите примеры основных шаблонов проектирования.

* **Делегирование** (Delegation pattern) - Сущность внешне выражает некоторое поведение, но в реальности передает ответственность за выполнение этого поведения связанному объекту.
* **Функциональный дизайн** (Functional design) - Гарантирует, что каждая сущность имеет только одну обязанность и исполняет ее с минимумом побочных эффектов на другие.
* **Неизменяемый интерфейс** (Immutable interface) - Создание неизменяемого объекта.
* **Интерфейс** (Interface) - Общий метод структурирования сущностей облегчающий их понимание.
* **Интерфейс-маркер** (Marker interface) - В качестве атрибута (как пометки объектной сущности) применяется наличие или отсутствие реализации интерфейса-маркера. В современных языках программирования вместо этого применяются атрибуты или аннотации.
* **Контейнер свойств** (Property container) - Позволяет добавлять дополнительные свойства сущности в контейнер внутри себя, вместо расширения новыми свойствами.
* **Канал событий** (Event channel) - Создает централизованный канал для событий. Использует сущность-представитель для подписки и сущность-представитель для публикации события в канале. Представитель существует отдельно от реального издателя или подписчика. Подписчик может получать опубликованные события от более чем одной сущности, даже если он зарегистрирован только на одном канале.

[https://github.com/enhorse/java-interview/blob/master/patterns.md](https://github.com/enhorse/java-interview/blob/master/patterns.md#%D0%A7%D1%82%D0%BE-%D1%82%D0%B0%D0%BA%D0%BE%D0%B5-%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD-%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)

5. Приведите примеры порождающих шаблонов проектирования.

* **Абстрактная фабрика** (Abstract factory) - Класс, который представляет собой интерфейс для создания других классов.

*Пример в java - newInstance() из javax.xml.parsers.DocumentBuilderFactory.*

interface Lada {

long getLadaPrice();

}

interface Ferrari {

long getFerrariPrice();

}

interface Porshe {

long getPorshePrice();

}

interface InteAbsFactory {

Lada getLada();

Ferrari getFerrari();

Porshe getPorshe();

}

class UaLadaImpl implements Lada {// первая

public long getLadaPrice() {

return 1000;

}

}

class UaFerrariImpl implements Ferrari {

public long getFerrariPrice() {

return 3000;

}

}

class UaPorsheImpl implements Porshe {

public long getPorshePrice() {

return 2000;

}

}

class UaCarPriceAbsFactory implements InteAbsFactory {

public Lada getLada() {

return new UaLadaImpl();

}

public Ferrari getFerrari() {

return new UaFerrariImpl();

}

public Porshe getPorshe() {

return new UaPorsheImpl();

}

}// первая

class RuLadaImpl implements Lada {// вторая

public long getLadaPrice() {

return 10000;

}

}

class RuFerrariImpl implements Ferrari {

public long getFerrariPrice() {

return 30000;

}

}

class RuPorsheImpl implements Porshe {

public long getPorshePrice() {

return 20000;

}

}

class RuCarPriceAbsFactory implements InteAbsFactory {

public Lada getLada() {

return new RuLadaImpl();

}

public Ferrari getFerrari() {

return new RuFerrariImpl();

}

public Porshe getPorshe() {

return new RuPorsheImpl();

}

}// вторая

​

public class AbstractFactoryTest {//тест

public static void main(String[] args) {

String country = "UA";

InteAbsFactory ifactory = null;

if(country.equals("UA")) {

ifactory = new UaCarPriceAbsFactory();

} else if(country.equals("RU")) {

ifactory = new RuCarPriceAbsFactory();

}

​

Lada lada = ifactory.getLada();

System.out.println(lada.getLadaPrice());

}

}

<https://refactoring.guru/ru/design-patterns/abstract-factory>

<https://www.youtube.com/watch?v=cmyUI_ZezoU&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=5>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

<https://www.baeldung.com/java-abstract-factory-pattern>

* **Строитель** (Builder) - Класс, который представляет собой интерфейс для создания сложного объекта.

*Пример в java - StringBuilder*.

class Car {

public void buildBase() {

print("Делаем корпус");

}

public void buildWheels() {

print("Ставим колесо");

}

public void buildEngine(Engine engine) {

print("Ставим движок: " + engine.getEngineType());

}

private void print(String msg){

System.out.println(msg);

}

}

interface Engine {

String getEngineType();

}

class OneEngine implements Engine {

public String getEngineType() {

return "Первый двигатель";

}

}

class TwoEngine implements Engine {

public String getEngineType() {

return "Второй двигатель";

}

}

abstract class Builder {

protected Car car;

public abstract Car buildCar();

}

class OneBuilderImpl extends Builder {

public OneBuilderImpl(){

car = new Car();

}

public Car buildCar() {

car.buildBase();

car.buildWheels();

Engine engine = new OneEngine();

car.buildEngine(engine);

return car;

}

}

class TwoBuilderImpl extends Builder {

public TwoBuilderImpl(){

car = new Car();

}

public Car buildCar() {

car.buildBase();

car.buildWheels();

Engine engine = new OneEngine();

car.buildEngine(engine);

car.buildWheels();

engine = new TwoEngine();

car.buildEngine(engine);

return car;

}

}

class Build {

private Builder builder;

public Build(int i){

if(i == 1) {

builder = new OneBuilderImpl();

} else if(i == 2) {

builder = new TwoBuilderImpl();

}

}

public Car buildCar(){

return builder.buildCar();

}

}

​

public class BuilderTest {//тест

public static void main(String[] args) {

Build build = new Build(1);

build.buildCar();

}

}

<https://refactoring.guru/ru/design-patterns/builder>

<https://www.youtube.com/watch?v=63_ExLjusac&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=7>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

<https://stackoverflow.com/questions/2169190/example-of-builder-pattern-in-java-api>

* **Фабричный метод** (Factory method) - Делегирует создание объектов наследникам родительского класса. Это позволяет использовать в коде программы не специфические классы, а манипулировать абстрактными объектами на более высоком уровне.

*Пример в java - метод toString() у Object.*

class Factory {

public OS getCurrentOS(String inputos) {

OS os = null;

if (inputos.equals("windows")) {

os = new windowsOS();

} else if (inputos.equals("linux")) {

os = new linuxOS();

} else if (inputos.equals("mac")) {

os = new macOS();

}

return os;

}

}

interface OS {

void getOS();

}

class windowsOS implements OS {

public void getOS () {

System.out.println("применить для виндовс");

}

}

class linuxOS implements OS {

public void getOS () {

System.out.println("применить для линукс");

}

}

class macOS implements OS {

public void getOS () {

System.out.println("применить для мак");

}

}

​

public class FactoryTest {//тест

public static void main(String[] args){

String win = "linux";

Factory factory = new Factory();

OS os = factory.getCurrentOS(win);

os.getOS();

}

}

<https://refactoring.guru/ru/design-patterns/factory-method>

<https://www.youtube.com/watch?v=TwIjjTC5g7g&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=4>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

<https://stackoverflow.com/questions/12592520/example-of-factory-pattern-in-java-jdk>

* **Прототип** (Prototype) - Определяет интерфейс создания объекта через клонирование другого объекта вместо создания через конструктор.

*Пример в java - метод clone() у Object.*

interface Copyable {

Copyable copy();

}

class ComplicatedObject implements Copyable {

private Type type;

public enum Type {

ONE, TWO

}

public ComplicatedObject copy() {

ComplicatedObject complicatedobject = new ComplicatedObject();

return complicatedobject;

}

public void setType(Type type) {

this.type = type;

}

}

​

public class PrototypeTest {//тест

public static void main(String[] args) {

ComplicatedObject prototype = new ComplicatedObject();

ComplicatedObject clone = prototype.copy();

clone.setType(ComplicatedObject.Type.ONE);

}

}

<https://refactoring.guru/ru/design-patterns/prototype>

<https://www.youtube.com/watch?v=7X3eQZX5b9Y&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=8>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

<https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries>

* **Одиночка** (Singleton) - Класс, который может иметь только один экземпляр.

*Пример в java - метод getDesktop() у Desktop.*

class Singleton {

private static Singleton instance = null;

private Singleton() {}

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

public void setUp() {

System.out.println("setUp");

}

}

​

public class SingletonTest {//тест

public static void main(String[] args){

Singleton singelton = Singleton.getInstance();

singelton.setUp();

}

}

<https://refactoring.guru/ru/design-patterns/singleton>

<https://www.youtube.com/watch?v=vyr9GO7dLBQ&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=6>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

<https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries>

[https://github.com/enhorse/java-interview/blob/master/patterns.md](https://github.com/enhorse/java-interview/blob/master/patterns.md#%D0%A7%D1%82%D0%BE-%D1%82%D0%B0%D0%BA%D0%BE%D0%B5-%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD-%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)

6. Приведите примеры структурных шаблонов проектирования.

* **Адаптер** (Adapter) - Объект, обеспечивающий взаимодействие двух других объектов, один из которых использует, а другой предоставляет несовместимый с первым интерфейс.

*Пример в java - метод asList() у Arrays.*

class PBank {

private int balance;

public PBank() { balance = 100; }

public void getBalance() {

System.out.println("PBank balance = " + balance);

}

}

class ABank {

private int balance;

public ABank() { balance = 200; }

public void getBalance() {

System.out.println("ABank balance = " + balance);

}

}

class PBankAdapter extends PBank {

private ABank abank;

public PBankAdapter(ABank abank) {

this.abank = abank;

}

public void getBalance() {

abank.getBalance();

}

}

​

public class AdapterTest {//тест

public static void main(String[] args) {

PBank pbank = new PBank();

pbank.getBalance();

PBankAdapter abank = new PBankAdapter(new ABank());

abank.getBalance();

}

}

<https://refactoring.guru/ru/design-patterns/adapter>

<https://www.youtube.com/watch?v=VunUuXyJw2U&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=10>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

<https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries>

* **Мост** (Bridge) - Разделяет реализацию и абстракцию, дает возможность изменять их свободно друг от друга. Делает конкретные классы независимыми от классов реализации интерфейса.

*Пример в java - метод newSetFromMap() у Collections.*

interface Engine {

void setEngine();

}

abstract class Car {

protected Engine engine;

public Car(Engine engine){

this.engine = engine;

}

abstract public void setEngine();

}

class SportCar extends Car {

public SportCar(Engine engine) {

super(engine);

}

public void setEngine() {

System.out.print("SportCar engine: ");

engine.setEngine();

}

}

class UnknownCar extends Car {

public UnknownCar(Engine engine) {

super(engine);

}

public void setEngine() {

System.out.print("UnknownCar engine: ");

engine.setEngine();

}

}

class SportEngine implements Engine {

public void setEngine(){

System.out.println("sport");

}

}

class UnknownEngine implements Engine {

public void setEngine(){

System.out.println("unknown");

}

}

public class BridgeTest {//тест

public static void main(String[] args) {

Car sportCar = new SportCar(new SportEngine());

sportCar.setEngine();

System.out.println();

Car unknownCar = new UnknownCar(new UnknownEngine());

unknownCar.setEngine();

}

}

<https://refactoring.guru/ru/design-patterns/bridge>

<https://www.youtube.com/watch?v=1KOkb-B2qnM&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=11>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

<https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries>

* **Компоновщик** (Composite) - Объект, который объединяет в себе объекты, подобные ему самому.

*Пример в java - метод add(Component) у java.awt.Container.*

interface Car {

void draw(String color);

}

class SportCar implements Car {

public void draw(String color) {

System.out.println("SportCar color: " + color);

}

}

class UnknownCar implements Car {

public void draw(String color) {

System.out.println("UnknownCar color: " + color);

}

}

class Drawing implements Car {

private List<Car> cars = new ArrayList<Car>();

public void draw(String color) {

for(Car car : cars) {

car.draw(color);

}

}

public void add(Car s){

this.cars.add(s);

}

public void clear(){

System.out.println();

this.cars.clear();

}

}

​

public class CompositeTest {//тест

public static void main(String[] args) {

Car sportCar = new SportCar();

Car unknownCar = new UnknownCar();

Drawing drawing = new Drawing();

drawing.add(sportCar);

drawing.add(unknownCar);

drawing.draw("green");

drawing.clear();

drawing.add(sportCar);

drawing.add(unknownCar);

drawing.draw("white");

}

}

<https://refactoring.guru/ru/design-patterns/composite>

<https://www.youtube.com/watch?v=eZx8eiTntAs&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=12>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

<https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries>

* **Декоратор** (Decorator) - Класс, расширяющий функциональность другого класса без использования наследования.

*Пример в java - java.io.InputStream, OutputStream, Reader и Writer.*

interface Car {

void draw();

}

class SportCar implements Car {

public void draw() {

System.out.println("SportCar");

}

}

class UnknownCar implements Car {

public void draw() {

System.out.println("UnknownCar");

}

}

abstract class CarDecorator implements Car {

protected Car decorated;

public CarDecorator(Car decorated){

this.decorated = decorated;

}

public void draw(){

decorated.draw();

}

}

class BlueCarDecorator extends CarDecorator {

public BlueCarDecorator(Car decorated) {

super(decorated);

}

public void draw() {

decorated.draw();

setColor();

}

private void setColor(){

System.out.println("Color: red");

}

}

​

public class DecoratorTest {//тест

public static void main(String[] args) {

Car sportCar = new SportCar();

Car blueUnknownCar = new BlueCarDecorator(new UnknownCar());

sportCar.draw();

System.out.println();

blueUnknownCar.draw();

}

}

<https://refactoring.guru/ru/design-patterns/decorator>

<https://www.youtube.com/watch?v=2aB2B3b3bQA&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=13>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

<https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries>

* **Фасад** (Facade) - Объект, который абстрагирует работу с несколькими классами, объединяя их в единое целое.

*Пример в java - javax.faces.context.ExternalContext, который используется внутри ServletContext, HttpSession, HttpServletRequest, HttpServletResponseи т.д.*

interface Car {

void start();

void stop();

}

class Key implements Car {

public void start() {

System.out.println("Вставить ключи");

}

public void stop() {

System.out.println("Вытянуть ключи");

}

}

class Engine implements Car {

public void start() {

System.out.println("Запустить двигатель");

}

public void stop() {

System.out.println("Остановить двигатель");

}

}

class Facade {

private Key key;

private Engine engine;

public Facade() {

key = new Key();

engine = new Engine();

}

public void startCar() {

key.start();

engine.start();

}

public void stoptCar() {

key.stop();

engine.stop();

}

}

​

public class FacadeTest {//тест

public static void main(String[] args) {

Facade facade = new Facade();

facade.startCar();

System.out.println();

facade.stoptCar();

}

}

<https://refactoring.guru/ru/design-patterns/facade>

<https://www.youtube.com/watch?v=7thoI-_K9EU&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=14>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

<https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries>

* **Приспособленец** (Flyweight) - Вместо создания большого количества похожих объектов, объекты используются повторно. Экономит память.

*Пример в java - пул строк, а также метод valueOf(int) у java.lang.Integer (также на Boolean, Byte, Character, Short, Long и BigDecimal)*

class Flyweight {

private int row;

public Flyweight(int row) {

this.row = row;

System.out.println("ctor: " + this.row);

}

void report(int col) {

System.out.print(" " + row + col);

}

}

​

class Factory {

private Flyweight[] pool;

public Factory(int maxRows) {

pool = new Flyweight[maxRows];

}

public Flyweight getFlyweight(int row) {

if (pool[row] == null) {

pool[row] = new Flyweight(row);

}

return pool[row];

}

}

​

public class FlyweightTest {//тест

public static void main(String[] args) {

int rows = 5;

Factory theFactory = new Factory(rows);

for (int i = 0; i < rows; i++) {

for (int j = 0; j < rows; j++) {

theFactory.getFlyweight(i).report(j);

}

System.out.println();

}

}

}

<https://refactoring.guru/ru/design-patterns/flyweight>

<https://www.youtube.com/watch?v=d9Cx-2LSGNA&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=15>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

<https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries>

* **Заместитель** (Proxy) - Объект, который является посредником между двумя другими объектами, и который реализует/ограничивает доступ к объекту, к которому обращаются через него.

*Пример в java - javax.persistence.PersistenceContext.*

interface Image {

void display();

}

class RealImage implements Image {

private String file;

public RealImage(String file){

this.file = file;

load(file);

}

private void load(String file){

System.out.println("Загрузка " + file);

}

public void display() {

System.out.println("Просмотр " + file);

}

}

class ProxyImage implements Image {

private String file;

private RealImage image;

public ProxyImage(String file){

this.file = file;

}

public void display() {

if(image == null){

image = new RealImage(file);

}

image.display();

}

}

​

public class ProxyTest {//тест

public static void main(String[] args) {

Image image = new ProxyImage("test.jpg");

image.display();

image.display();

}

}

<https://refactoring.guru/ru/design-patterns/proxy>

<https://www.youtube.com/watch?v=EzDkgArMO3U&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=16>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

<https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries>

[https://github.com/enhorse/java-interview/blob/master/patterns.md](https://github.com/enhorse/java-interview/blob/master/patterns.md#%D0%A7%D1%82%D0%BE-%D1%82%D0%B0%D0%BA%D0%BE%D0%B5-%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD-%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)

7. Приведите примеры поведенческих шаблонов проектирования.

* **Цепочка обязанностей** (Chain of responsibility) - Позволяет избежать жесткой зависимости отправителя запроса от его получателя, при этом запрос может быть обработан несколькими объектами.

*Пример в java - метод log() у java.util.logging.Logger.*

interface Payment {

void setNext(Payment payment);

void pay();

}

class VisaPayment implements Payment {

private Payment payment;

public void setNext(Payment payment) {

this.payment = payment;

}

public void pay() {

System.out.println("Visa Payment");

}

}

class PayPalPayment implements Payment {

private Payment payment;

public void setNext(Payment payment) {

this.payment = payment;

}

public void pay() {

System.out.println("PayPal Payment");

}

}

​

public class ChainofResponsibilityTest {//тест

public static void main(String[] args) {

Payment visaPayment = new VisaPayment();

Payment payPalPayment = new PayPalPayment();

visaPayment.setNext(payPalPayment);

visaPayment.pay();

}

}

<https://refactoring.guru/ru/design-patterns/chain-of-responsibility>

<https://www.youtube.com/watch?v=66_0-u8P5DQ&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=18>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

<https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries>

* **Команда** (Command) - Позволяет инкапсулировать различные операции в отдельные объекты.

*Пример в java - все реализации java.lang.Runnable.*

interface Command {

void execute();

}

class Car {

public void startEngine() {

System.out.println("запустить двигатель");

}

public void stopEngine() {

System.out.println("остановить двигатель");

}

}

class StartCar implements Command {

Car car;

public StartCar(Car car) {

this.car = car;

}

public void execute() {

car.startEngine();

}

}

class StopCar implements Command {

Car car;

public StopCar(Car car) {

this.car = car;

}

public void execute() {

car.stopEngine();

}

}

class CarInvoker {

public Command command;

public CarInvoker(Command command){

this.command = command;

}

public void execute(){

this.command.execute();

}

}

​

public class CommandTest {//тест

public static void main(String[] args) {

Car car = new Car();

StartCar startCar = new StartCar(car);

StopCar stopCar = new StopCar(car);

CarInvoker carInvoker = new CarInvoker(startCar);

carInvoker.execute();

}

}

<https://refactoring.guru/ru/design-patterns/command>

<https://www.youtube.com/watch?v=T3oXyVYmkyY&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=19>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

<https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries>

* **Интерпретатор** (Interpreter) - Решает часто встречающуюся, но подверженную изменениям, задачу.

*Пример в java - все подклассы java.text.Format.*

interface Expression {

String interpret(Context context);

}

class Context {

public String getLowerCase(String s){

return s.toLowerCase();

}

public String getUpperCase(String s){

return s.toUpperCase();

}

}

class LowerExpression implements Expression {

private String s;

public LoverExpression(String s) {

this.s = s;

}

public String interpret(Context context) {

return context.getLoverCase(s);

}

}

class UpperExpression implements Expression {

private String s;

public UpperExpression(String s) {

this.s = s;

}

public String interpret(Context context) {

return context.getUpperCase(s);

}

}

​

public class InterpreterTest {//тест

public static void main(String[] args) {

String str = "TesT";

Context context = new Context();

Expression loverExpression = new LoverExpression(str);

str = loverExpression.interpret(context);

System.out.println(str);

Expression upperExpression = new UpperExpression(str);

str = upperExpression.interpret(context);

System.out.println(str);

}

}

<https://www.youtube.com/watch?v=8fRHz7_9pGI&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=20>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

<https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries>

* **Итератор** (Iterator) - Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из объектов, входящих в состав агрегации.

*Пример в java - все реализации java.util.Iterator.*

interface Iterator {

boolean hasNext();

Object next();

}

class Numbers {

public int num[] = {1 , 2, 3};

public Iterator getIterator() {

return new NumbersIterator();

}

private class NumbersIterator implements Iterator {

int ind;

public boolean hasNext() {

if(ind < num.length) return true;

return false;

}

public Object next() {

if(this.hasNext()) return num[ind++];

return null;

}

}

}

​

public class IteratorTest {//тест

public static void main(String[] args) {

Numbers numbers = new Numbers();

Iterator iterator = numbers.getIterator();

while (iterator.hasNext()) {

System.out.println(iterator.next());

}

}

}

<https://refactoring.guru/ru/design-patterns/iterator>

<https://www.youtube.com/watch?v=yY6oy8xHLT8&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=21>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

<https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries>

* **Посредник** (Mediator) - Обеспечивает взаимодействие множества объектов, формируя при этом слабую связанность и избавляя объекты от необходимости явно ссылаться друг на друга.

*Пример в java - метод execute() у java.util.concurrent.Executor.*

class Mediator {

public static void sendMessage(User user, String msg){

System.out.println(user.getName() + ": " + msg);

}

}

class User {

private String name;

public User(String name){

this.name = name;

}

public String getName() {

return name;

}

public void sendMessage(String msg){

Mediator.sendMessage(this, msg);

}

}

​

public class MediatorTest {//тест

public static void main(String[] args) {

User user1 = new User("user1");

User user2 = new User("user2");

user1.sendMessage("message1");

user2.sendMessage("message2");

}

}

<https://refactoring.guru/ru/design-patterns/mediator>

<https://www.youtube.com/watch?v=ZnyNsrcLl2I&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=22>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

<https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries>

* **Хранитель** (Memento) - Позволяет не нарушая инкапсуляцию зафиксировать и сохранить внутренние состояния объекта так, чтобы позднее восстановить его в этих состояниях.

*Пример в java - все реализации java.io.Serializable.*

class Memento {

private String name;

private int age;

public Memento(String name, int age){

this.name = name;

this.age = age;

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

}

class User {

private String name;

private int age;

public User(String name, int age) {

this.name = name;

this.age = age;

System.out.println(String.format("create: name = %s, age = %s", name, age));

}

public Memento save(){

System.out.println(String.format("save: name = %s, age = %s", name, age));

return new Memento(name, age);

}

public void restore(Memento memento){

name = memento.getName();

age = memento.getAge();

System.out.println(String.format("restore: name = %s, age = %s", name, age));

}

}

class SaveUser {

private List<Memento> list = new ArrayList<Memento>();

public void add(Memento memento){

list.add(memento);

}

public Memento get(int ind){

return list.get(ind);

}

}

​

public class MementoTest { // Test

public static void main(String[] args) {

SaveUser saveUser = new SaveUser();

User user1 = new User("Peter", 17);

User user2 = new User("Ian", 19);

saveUser.add(user1.save());

user1.restore(saveUser.get(0));

}

}

<https://refactoring.guru/ru/design-patterns/memento>

<https://www.youtube.com/watch?v=nu2CN3r4Jp0&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=23>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

* **Наблюдатель** (Observer) - Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.

*Пример в java - все реализации java.util.EventListener (практически во всем Swing).*

interface Observer {

void event(List<String> strings);

}

class University {

private List<Observer> observers = new ArrayList<Observer>();

private List<String> students = new ArrayList<String>();

public void addStudent(String name) {

students.add(name);

notifyObservers();

}

public void removeStudent(String name) {

students.remove(name);

notifyObservers();

}

public void addObserver(Observer observer){

observers.add(observer);

}

public void removeObserver(Observer observer) {

observers.remove(observer);

}

public void notifyObservers(){

for (Observer observer : observers) {

observer.event(students);

}

}

}

class Director implements Observer {

public void event(List<String> strings) {

System.out.println("The list of students has changed: " + strings);

}

}

​

public class ObserverTest {//тест

public static void main(String[] args) {

University university = new University();

Director director = new Director();

university.addStudent("Vaska");

university.addObserver(director);

university.addStudent("Anna");

university.removeStudent("Vaska");

}

}

<https://refactoring.guru/ru/design-patterns/observer>

<https://www.youtube.com/watch?v=br5201sWOHM&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=24>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

<https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries>

* **Состояние** (State) - Используется в тех случаях, когда во время выполнения программы объект должен менять свое поведение в зависимости от своего состояния.

*Пример в java - метод execute() у javax.faces.lifecycle.LifeCycle (контролируется FacesServlet, поведение зависит от текущей фазы (состояния) жизненного цикла JSF).*

interface State {

void doAction();

}

class StartPlay implements State {

public void doAction() {

System.out.println("start play");

}

}

class StopPlay implements State {

public void doAction() {

System.out.println("stop play");

}

}

class PlayContext implements State {

private State state;

public void setState(State state){

this.state = state;

}

public void doAction() {

this.state.doAction();

}

}

​

public class StateTest {//тест

public static void main(String[] args) {

PlayContext playContext = new PlayContext();

State startPlay = new StartPlay();

State stopPlay = new StopPlay();

playContext.setState(startPlay);

playContext.doAction();

playContext.setState(stopPlay);

playContext.doAction();

}

}

<https://refactoring.guru/ru/design-patterns/state>

<https://www.youtube.com/watch?v=gpY98f7A_8M&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=25>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

<https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries>

* **Стратегия** (Strategy) - Определяет ряд алгоритмов позволяя взаимодействовать между ними. Алгоритм стратегии может быть изменен во время выполнения программы.

*Пример в java - метод compare() java.util.Comparator, выполненный среди других методов sort() у Collections.*

interface Strategy {

void download(String file);

}

class DownloadWindownsStrategy implements Strategy {

public void download(String file) {

System.out.println("windows download: " + file);

}

}

class DownloadLinuxStrategy implements Strategy {

public void download(String file) {

System.out.println("linux download: " + file);

}

}

class Context {

private Strategy strategy;

public Context(Strategy strategy){

this.strategy = strategy;

}

public void download(String file){

strategy.download(file);

}

}

​

public class StrategyTest {//тест

public static void main(String[] args) {

Context context = new Context(new DownloadWindownsStrategy());

context.download("file.txt");

context = new Context(new DownloadLinuxStrategy());

context.download("file.txt");

}

}

<https://refactoring.guru/ru/design-patterns/strategy>

<https://www.youtube.com/watch?v=rsB2exGsR4I&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=26>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

<https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries>

* **Шаблонный метод** (Template method) - Определяет основу алгоритма и позволяет наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.

*Пример в java - все не абстрактные методы java.io.InputStream, java.io.OutputStream, java.io.Readerи java.io.Writer.*

abstract class Car {

abstract void startEngine();

abstract void stopEngine();

​

public final void start(){

startEngine();

stopEngine();

}

}

class OneCar extends Car {

public void startEngine() {

System.out.println("Start engine.");

}

public void stopEngine() {

System.out.println("Stop engine.");

}

}

class TwoCar extends Car {

public void startEngine() {

System.out.println("Start engine.");

}

public void stopEngine() {

System.out.println("Stop engine.");

}

}

​

public class TemplateTest {//тест

public static void main(String[] args) {

Car car1 = new OneCar();

car1.start();

System.out.println();

Car car2 = new TwoCar();

car2.start();

}

}

<https://refactoring.guru/ru/design-patterns/template-method>

<https://www.youtube.com/watch?v=ib8HrjOkT-0&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=27>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

<https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries>

* **Посетитель** (Visitor) - Описывает операцию, которая выполняется над объектами других классов. При изменении класса Visitor нет необходимости изменять обслуживаемые классы.

*Пример в java - java.nio.file.FileVisitor и SimpleFileVisitor.*

interface Visitor {

void visit(SportCar sportCar);

void visit(Engine engine);

void visit(Whell whell);

}

interface Car {

void accept(Visitor visitor);

}

class Engine implements Car {

public void accept(Visitor visitor) {

visitor.visit(this);

}

}

class Whell implements Car {

public void accept(Visitor visitor) {

visitor.visit(this);

}

}

class SportCar implements Car {

Car[] cars;

public SportCar(){

cars = new Car[]{new Engine(), new Whell()};

}

public void accept(Visitor visitor) {

for (int i = 0; i < cars.length; i++) {

cars[i].accept(visitor);

}

visitor.visit(this);

}

}

class CarVisitor implements Visitor {

public void visit(SportCar computer) {

print("car");

}

public void visit(Engine engine) {

print("engine");

}

public void visit(Whell whell) {

print("whell");

}

private void print(String string) {

System.out.println(string);

}

}

​

public class VisitorTest {//тест

public static void main(String[] args) {

Car computer = new SportCar();

computer.accept(new CarVisitor());

}

}

<https://refactoring.guru/ru/design-patterns/visitor>

<https://www.youtube.com/watch?v=IcPftlAIIXg&t=409s>

<https://www.youtube.com/watch?v=vc2Ay_IzEwM>

<https://javarush.ru/groups/posts/584-patternih-proektirovanija>

<https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries>

[https://github.com/enhorse/java-interview/blob/master/patterns.md](https://github.com/enhorse/java-interview/blob/master/patterns.md#%D0%A7%D1%82%D0%BE-%D1%82%D0%B0%D0%BA%D0%BE%D0%B5-%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD-%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)

8. Что такое «антипаттерн»? Какие антипаттерны вы знаете?

**Антипаттерн** (anti-pattern) — это распространённый подход к решению класса часто встречающихся проблем, являющийся неэффективным, рискованным или непродуктивным.

**Poltergeists** (полтергейсты) - это классы с ограниченной ответственностью и ролью в системе, чьё единственное предназначение — передавать информацию в другие классы. Их эффективный жизненный цикл непродолжителен. Полтергейсты нарушают стройность архитектуры программного обеспечения, создавая избыточные (лишние) абстракции, они чрезмерно запутаны, сложны для понимания и трудны в сопровождении. Обычно такие классы задумываются как классы-контроллеры, которые существуют только для вызова методов других классов, зачастую в предопределенной последовательности.

Признаки появления и последствия антипаттерна

* Избыточные межклассовые связи.
* Временные ассоциации.
* Классы без состояния (содержащие только методы и константы).
* Временные объекты и классы (с непродолжительным временем жизни).
* Классы с единственным методом, который предназначен только для создания или вызова других классов посредством временной ассоциации.
* Классы с именами методов в стиле «управления», такие как startProcess.

Типичные причины

* Отсутствие объектно-ориентированной архитектуры (архитектор не понимает объектно-ориентированной парадигмы).
* Неправильный выбор пути решения задачи.
* Предположения об архитектуре приложения на этапе анализа требований (до объектно-ориентированного анализа) могут также вести к проблемам на подобии этого антипаттерна.

**Внесенная сложность** (Introduced complexity): Необязательная сложность дизайна. Вместо одного простого класса выстраивается целая иерархия интерфейсов и классов. Типичный пример «Интерфейс - Абстрактный класс - Единственный класс реализующий интерфейс на основе абстрактного».

**Инверсия абстракции** (Abstraction inversion): Сокрытие части функциональности от внешнего использования, в надежде на то, что никто не будет его использовать.

**Неопределенная точка зрения** (Ambiguous viewpoint): Представление модели без спецификации её точки рассмотрения.

**Большой комок грязи** (Big ball of mud): Система с нераспознаваемой структурой.

**Божественный объект** (God object): Концентрация слишком большого количества функций в одной части системы (классе).

**Затычка на ввод данных** (Input kludge): Забывчивость в спецификации и выполнении поддержки возможного неверного ввода.

**Раздувание интерфейса** (Interface bloat): Разработка интерфейса очень мощным и очень сложным для реализации.

**Волшебная кнопка** (Magic pushbutton): Выполнение результатов действий пользователя в виде неподходящего (недостаточно абстрактного) интерфейса. Например, написание прикладной логики в обработчиках нажатий на кнопку.

**Перестыковка** (Re-Coupling): Процесс внедрения ненужной зависимости.

**Дымоход** (Stovepipe System): Редко поддерживаемая сборка плохо связанных компонентов.

**Состояние гонки** (Race hazard): непредвидение возможности наступления событий в порядке, отличном от ожидаемого.

**Членовредительство** (Mutilation): Излишнее «затачивание» объекта под определенную очень узкую задачу таким образом, что он не способен будет работать с никакими иными, пусть и очень схожими задачами.

**Сохранение или смерть** (Save or die): Сохранение изменений лишь при завершении приложения.

[https://github.com/enhorse/java-interview/blob/master/patterns.md](https://github.com/enhorse/java-interview/blob/master/patterns.md#%D0%A7%D1%82%D0%BE-%D1%82%D0%B0%D0%BA%D0%BE%D0%B5-%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD-%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)

9. Что такое Dependency Injection?

**Dependency Injection** (внедрение зависимости) - это набор паттернов и принципов разработки программного обеспечения, которые позволяют писать слабосвязный код. В полном соответствии с принципом единой обязанности объект отдаёт заботу о построении требуемых ему зависимостей внешнему, специально предназначенному для этого общему механизму.

[https://github.com/enhorse/java-interview/blob/master/patterns.md](https://github.com/enhorse/java-interview/blob/master/patterns.md#%D0%A7%D1%82%D0%BE-%D1%82%D0%B0%D0%BA%D0%BE%D0%B5-%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD-%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)

**Ответы на доп. вопросы**

1. Паттерны в spring framework

Singleton - Creating beans with default scope.

Factory - Bean Factory classes

Prototype - Bean scopes

Adapter - Spring Web and Spring MVC

Proxy - Spring Aspect Oriented Programming support

Template Method - JdbcTemplate, HibernateTemplate etc

Front Controller - Spring MVC DispatcherServlet

Data Access Object - Spring DAO support

Dependency Injection and Aspect Oriented Programming

2. Какие паттерны применяются в Hibernate?

Domain Model – объектная модель предметной области, включающая в себя как поведение так и данные.

Data Mapper – слой мапперов (Mappers), который передает данные между объектами и базой данных, сохраняя их независимыми друг от друга и себя.

Proxy — применяется для ленивой загрузки.

Factory — используется в SessionFactory