МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ПРОМЫШЛЕННЫХ ТЕХНОЛОГИЙ И ДИЗАЙНА»

Институт информационных технологий и автоматизации

| Кафедра: | Интеллектуальных систем и защиты информации |
| --- | --- |
| Специальность: | 09.03.02 Информационные системы и технологии |

**КУРСОВАЯ РАБОТА**

по дисциплине: «Технологии и методы программирования»

на тему: «Применение паттернов проектирования GoF»

| Преподаватель: | Бусыгин К.Н. |  |  |
| --- | --- | --- | --- |
| Соколов А.П. |  | (подпись) |
| Обучающиеся: | Бирюков А.П. |  |  |
|  |  | (подпись) |

| Курс | 2 | Учебная группа | 2-МД-16 |
| --- | --- | --- | --- |

Санкт-Петербург

2022

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ](#_30j0zll) **4**

[**Теоретическая часть**](#_1fob9te) **5**

[**Порождающие паттерны**](#_t7lmgi4io7vx) **5**

[**Абстрактная фабрика (Abstract Factory)**](#_iszffyhstohb) **5**

[**Строитель (Builder)**](#_ta5frxjrg1q5) **6**

[**Фабричный метод (Factory Method)**](#_yupzvunynvbx) **7**

[**Прототип (Prototype)**](#_dpykxlrqj3pp) **8**

[**Одиночка (Singleton)**](#_3fh4lg7ulcdw) **10**

[**Структурные паттерны**](#_ctn16doeheb4) **11**

[**Адаптер (Adapter)**](#_wulgvapm3uec) **11**

[**Мост (Bridge)**](#_s1yoosdmqcn4) **12**

[**Компоновщик (Composite)**](#_8he9okaxe9tu) **16**

[**Декоратор (Decorator)**](#_gvh9exqh1nfm) **19**

[**Фасад (Facade)**](#_2asizqaahizd) **22**

[**Приспособленец (Flyweight)**](#_j71eoidvfmla) **25**

[**Заместитель (Proxy)**](#_lu605ut7chlk) **28**

[**Поведенчиские паттерны**](#_u8yby1z3mg81) **30**

[**Цепочка обязанностей (Chain of responsibility)**](#_od6qaal6km1a) **30**

[**Команда (Command)**](#_vxec9dgv8o1f) **31**

[**Интерпритатор (Interpreter)**](#_upuyjd7pzwmu) **32**

[**Итератор (Iterator)**](#_3bs8t0ojcxh2) **36**

[**Посредник (Mediator)**](#_79b1uei93zi7) **36**

[**Хранитель (Memento)**](#_kk7o2qkusael) **37**

[**Наблюдатель (Observer)**](#_x34wz6xtog9n) **40**

[**Состояние (State)**](#_1knmghw985ua) **41**

[**Стратегия (Strategy)**](#_38fic79u7cc0) **44**

[**Шаблонный метод (Template Method)**](#_ahpow62orgvs) **45**

[**Посетитель (Visitor)**](#_tvjdxpgggnrz) **50**

[**Практическая часть**](#_6c8gy7fz25cj) **52**

[ЗАКЛЮЧЕНИЕ](#_1t3h5sf) **52**

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ](#_4d34og8) **53**

### ВВЕДЕНИЕ

Курсовая работа посвящена изучению и использованию паттернов проектирования GoF. Целью работы является применение знаний и навыков паттернов проектирования и внедрение их в свои проекты.

Ставятся следующие задачи:

* Освоить теоретическую информацию, связанную с паттернами программирования;
* Разобраться в способе их применения;
* Внедрить необходимые паттерны в свои учебные проекты;

### Теоретическая часть

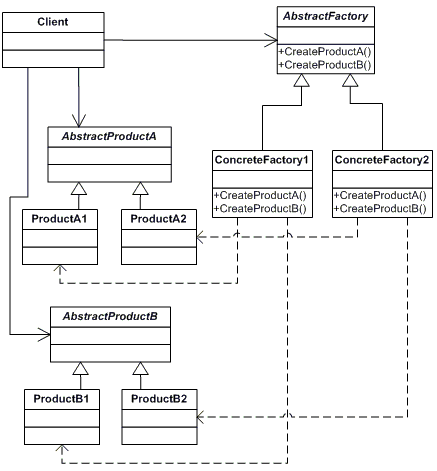
Паттернами проектирования (Design Patterns) называют решения часто встречающихся проблем в области разработки программного обеспечения. В данном случае предполагается, что есть некоторый набор общих формализованных проблем, которые довольно часто встречаются, и паттерны предоставляют ряд принципов для решения этих проблем.

### Порождающие паттерны

Порождающие паттерны — это паттерны, которые абстрагируют процесс инстанцирования или, иными словами, процесс порождения классов и объектов

### Абстрактная фабрика (Abstract Factory)

Назначение: абстрактная фабрика предоставляет интерфейс для создания семейства взаимосвязанных или родственных объектов (dependent or related objects), не специфицируя их конкретных классов.



Пример кода:

*import java.util.\*;*

*class Main {*

*public static void main(String[] args) {*

*Runnable command = () -> {*

*System.out.println("Command action");*

*};*

*Thread th = new Thread(command);*

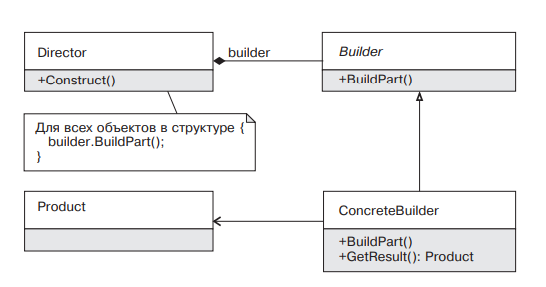
*th.start();*

*}*

*}*

### Строитель (Builder)

Назначение: Строитель отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления.



Пример кода:

*class Main {*

*public static void main(String[] args) {*

*StringBuilder builder = new StringBuilder();*

*builder.append("Hello");*

*builder.append(',');*

*builder.append("World!");*

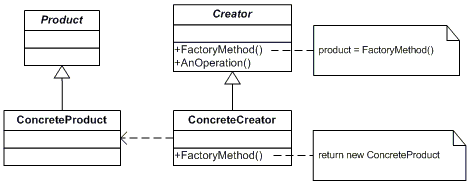
*System.out.println(builder.toString());*

*}*

*}*

### Фабричный метод (Factory Method)

Назначение: определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Фабричный метод позволяет классу делегировать инстанцирование подклассам.



Пример кода:

*import java.util.\*;*

*class Main {*

*public static void main(String[] args) {*

*Calendar calendar = Calendar.getInstance();*

*System.out.println(calendar.getTime());*

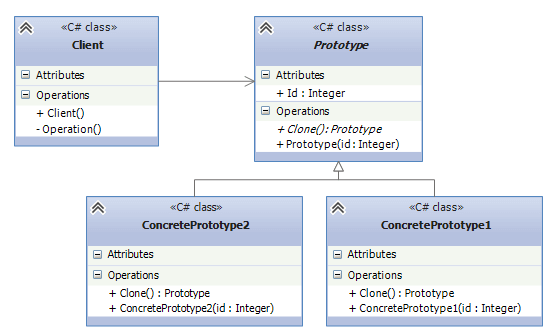
*System.out.println(calendar.getClass().getCanonicalName());*

*}*

*}*

### Прототип (Prototype)

Назначение: Прототип —​ позволяет создавать новые объекты путем клонирования уже существующих. По сути данный паттерн предлагает технику клонирования объектов.



Пример кода:

*class Main {*

*public static void main(String[] args) {*

*class CloneObject implements Cloneable {*

*@Override*

*protected Object clone() throws CloneNotSupportedException {*

*return new CloneObject();*

*}*

*}*

*CloneObject obj = new CloneObject();*

*try {*

*CloneObject pattern = (CloneObject) obj.clone();*

*} catch (CloneNotSupportedException e) {*

*//Do something*

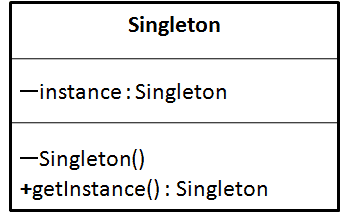
*}*

*}*

*}*

### Одиночка (Singleton)

Назначение: гарантирует, что у класса есть только один экземпляр, и предоставляет глобальную точку доступа к нему



Пример кода:

*package GeneratingPatterns.Singleton;*

*/\*Гарантирует что у класса будет только один экземпляр,*

*и предоставляет ему глобальную точку доступа\*/*

*public class SingletonMain {*

*public static void main(String[] args) {*

*Singleton singleton = Singleton.getSingleton();*

*Singleton singleton1 = Singleton.getSingleton();*

*Singleton singleton2 = Singleton.getSingleton();*

*singleton.name = "Semen";*

*System.out.println(singleton1.name);*

*singleton.name = "Ivan";*

*System.out.println(singleton2.name);*

*}*

*}*

*class Singleton {*

*String name = null;*

*static Singleton singleton = new Singleton();*

*private Singleton() {}*

*public static Singleton getSingleton() {*

*return singleton;*

*}*

*}*

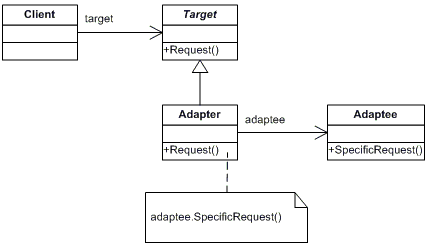
### Структурные паттерны

**Структурные паттерны** - рассматривает, как классы и объекты образуют более крупные структуры - более сложные по характеру классы и объекты.

### Адаптер (Adapter)

Назначение**:**

Преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты. Адаптер делает возможной совместную работу классов с несовместимыми интерфейсами.



Пример кода:

*import java.util.\*;*

*class Main {*

*public static void main(String[] arguments) {*

*String[] array = {"One", "Two", "Three"};*

*List<String> strings = Arrays.asList(array);*

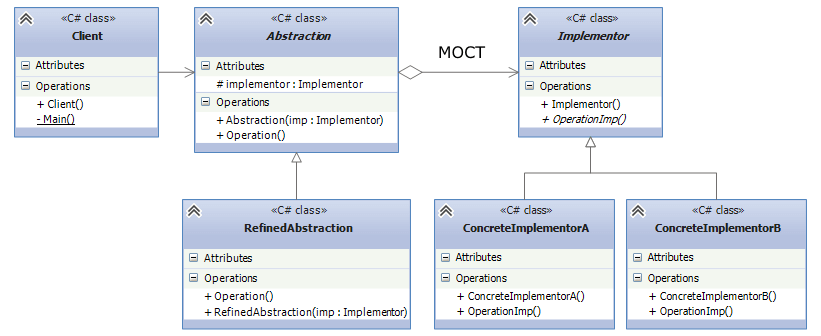
*strings.set(0, "1");*

*System.out.println(Arrays.toString(array));*

*}*

*}*

### **Мост (Bridge)**

Мост (Bridge) - структурный шаблон проектирования, который позволяет отделить абстракцию от реализации таким образом, чтобы и абстракцию, и реализацию можно было изменять независимо друг от друга.  
  
****Пример кода:  
*public class BridgeMain {*

*public static void main(String[] args) {*

*Shape[] shapes = {*

*new Circle(5, 10, 10, new LargeCircleDrawer()),*

*new Circle(20, 30, 100, new SmallCircleDrawer())};*

*for (Shape next : shapes)*

*next.draw();*

*}*

*}*

*interface Drawer {*

*public void drawCircle(int x, int y, int radius);*

*}*

*class SmallCircleDrawer implements Drawer {*

*public static final double radiusMultiplier = 0.25;*

*@Override*

*public void drawCircle(int x, int y, int radius) {*

*System.out.println("Small circle center = " + x + "," + y + " radius = " + radius \* radiusMultiplier);*

*}*

*}*

*class LargeCircleDrawer implements Drawer {*

*public static final int radiusMultiplier = 10;*

*@Override*

*public void drawCircle(int x, int y, int radius) {*

*System.out.println("Large circle center = " + x + "," + y + " radius = " + radius \* radiusMultiplier);*

*}*

*}*

*abstract class Shape {*

*protected Drawer drawer;*

*protected Shape(Drawer drawer) {*

*this.drawer = drawer;*

*}*

*public abstract void draw();*

*public abstract void enlargeRadius(int multiplier);*

*}*

*class Circle extends Shape {*

*private int x;*

*private int y;*

*private int radius;*

*public Circle(int x, int y, int radius, Drawer drawer) {*

*super(drawer);*

*setX(x);*

*setY(y);*

*setRadius(radius);*

*}*

*@Override*

*public void draw() {*

*drawer.drawCircle(x, y, radius);*

*}*

*@Override*

*public void enlargeRadius(int multiplier) {*

*radius \*= multiplier;*

*}*

*public int getX() {*

*return x;*

*}*

*public int getY() {*

*return y;*

*}*

*public int getRadius() {*

*return radius;*

*}*

*public void setX(int x) {*

*this.x = x;*

*}*

*public void setY(int y) {*

*this.y = y;*

*}*

*public void setRadius(int radius) {*

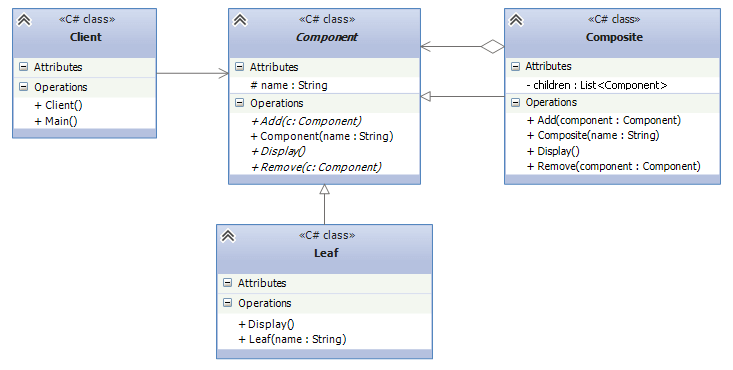
*this.radius = radius;*

*}*

*}*

### **Компоновщик (Composite)**

Компонует объекты в древовидные структуры для представления иерархий «часть — целое». Позволяет клиентам единообразно трактовать индивидуальные и составные объекты.

  
Пример кода:  
*import java.util.ArrayList;*

*import java.util.List;*

*// Компонент*

*interface Object {*

*void print();*

*}*

*// Компоновщик*

*class CompositeObject implements Object {*

*//Коллекция дочерних объектов*

*private final List<Object> ChildObjects = new ArrayList<>();*

*//Выводит объект*

*public void print() {*

*for (Object object : ChildObjects) {*

*object.print();*

*}*

*}*

*//Добавляет объект в структуру*

*public void add(Object object) {*

*ChildObjects.add(object);*

*}*

*//Удаляет объект из структуры*

*public void remove(Object object) {*

*ChildObjects.remove(object);*

*}*

*}*

*// Лист*

*class TestObject implements Object {*

*//Выводит Объект*

*public void print() {*

*System.out.println("TestObject");*

*}*

*}*

*//Клиент*

*public class CompositeMain {*

*public static void main(String[] args) {*

*//Инициализация тестовых объектов*

*TestObject TestObject1 = new TestObject();*

*TestObject TestObject2 = new TestObject();*

*TestObject TestObject3 = new TestObject();*

*TestObject TestObject4 = new TestObject();*

*//Инициализация трех комповощик-объектов*

*CompositeObject object = new CompositeObject();*

*CompositeObject object1 = new CompositeObject();*

*CompositeObject object2 = new CompositeObject();*

*//Объеденияем объекты*

*object1.add(TestObject1);*

*object1.add(TestObject2);*

*object1.add(TestObject3);*

*object2.add(TestObject4);*

*object.add(object1);*

*object.add(object2);*

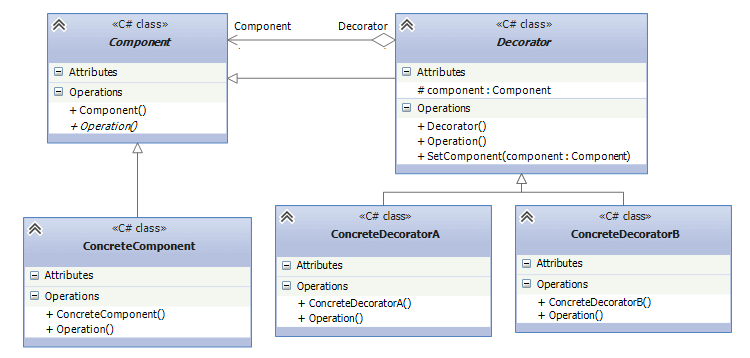
*//Выводим*

*object.print();*

*}*

*}*

### Декоратор (Decorator)

Назначение: Динамически добавляет объекту новые обязанности. Является гибкой альтернативой порождению подклассов с целью расширения функциональности. ****Пример кода:

*public class DecoratorMain {*

*public static void main(String... s) {*

*Decorator c = new DecoratorHello(new DecoratorComma(new DecoratorSpace(new MainComponent())));*

*c.doOperation();*

*c.newOperation();*

*}*

*}*

*interface InterfaceComponent {*

*void doOperation();*

*}*

*class MainComponent implements InterfaceComponent {*

*@Override*

*public void doOperation() {*

*System.out.print("World!");*

*}*

*}*

*abstract class Decorator implements InterfaceComponent {*

*protected InterfaceComponent component;*

*public Decorator (InterfaceComponent component) {*

*this.component = component;*

*}*

*@Override*

*public void doOperation() {*

*component.doOperation();*

*}*

*public void newOperation() {*

*System.out.println("Do Nothing");*

*}*

*}*

*class DecoratorSpace extends Decorator {*

*public DecoratorSpace(InterfaceComponent c) {*

*super(c);*

*}*

*@Override*

*public void doOperation() {*

*System.out.print(" ");*

*super.doOperation();*

*}*

*@Override*

*public void newOperation() {*

*System.out.println("\nNew space operation");*

*}*

*}*

*class DecoratorComma extends Decorator {*

*public DecoratorComma(InterfaceComponent c) {*

*super(c);*

*}*

*@Override*

*public void doOperation() {*

*System.out.print(",");*

*super.doOperation();*

*}*

*@Override*

*public void newOperation() {*

*System.out.println("\nNew comma operation");*

*}*

*}*

*class DecoratorHello extends Decorator {*

*public DecoratorHello(InterfaceComponent c) {*

*super(c);*

*}*

*@Override*

*public void doOperation() {*

*System.out.print("Hello");*

*super.doOperation();*

*}*

*@Override*

*public void newOperation() {*

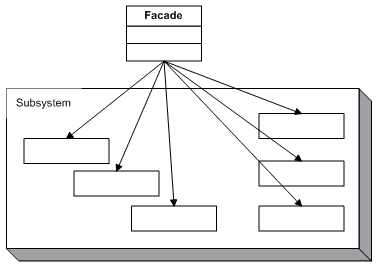
*System.out.println("\nNew hello operation");*

*}*

*}*

### Фасад (Facade)

Назначение: предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Фасад определяет интерфейс более высокого уровня, который упрощает использование подсистемы.

****Пример кода:

*import java.util.Arrays;*

*/\* Клиент \*/*

*public class FacadeMain {*

*public static void main(String[] args) {*

*Computer computer = new Computer();*

*computer.startComputer();*

*}*

*}*

*class CPU {*

*public void freeze() {*

*System.out.println("freeze");*

*}*

*public void jump(long position) {*

*System.out.println("jump position = " + position);*

*}*

*public void execute() {*

*System.out.println("execute");*

*}*

*}*

*class Memory {*

*public void load(long position, byte[] data) {*

*System.out.println("load position = " + position + ", data = " + Arrays.toString(data));*

*}*

*}*

*class HardDrive {*

*public byte[] read(long lba, int size) {*

*System.out.println("read lba = " + lba + ", size = " + size);*

*return new byte[size];*

*}*

*}*

*/\* Фасад \*/*

*class Computer {*

*private final static long BOOT\_ADDRESS = 1L;*

*private final static long BOOT\_SECTOR = 2L;*

*private final static int SECTOR\_SIZE = 3;*

*private final CPU cpu;*

*private final Memory memory;*

*private final HardDrive hardDrive;*

*public Computer() {*

*this.cpu = new CPU();*

*this.memory = new Memory();*

*this.hardDrive = new HardDrive();*

*}*

*public void startComputer() {*

*cpu.freeze();*

*memory.load(BOOT\_ADDRESS, hardDrive.read(BOOT\_SECTOR, SECTOR\_SIZE));*

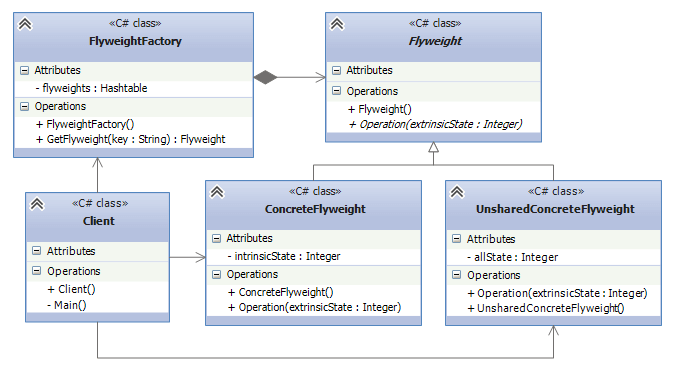
*cpu.jump(BOOT\_ADDRESS);*

*cpu.execute();*

*}*

*}*

### **Приспособленец (Flyweight)**

Назначение: Паттерн Приспособленец (Flyweight) - шаблон проектирования, который позволяет использовать разделяемые объекты сразу в нескольких контекстах. Данный паттерн используется преимущественно для оптимизации работы с памятью.  
  
  
Пример кода:  
*import java.util.HashMap;*

*abstract class CharacterStatement {*

*protected String nameCharacter;*

*protected int power;*

*protected int charisma;*

*public abstract void printStatement();*

*}*

*class ShrekStatement extends CharacterStatement {*

*public ShrekStatement(){*

*nameCharacter = "SHREK";*

*power = 94129000;*

*charisma = 20000;*

*}*

*@Override*

*public void printStatement() {*

*System.out.println("Name = " + nameCharacter + " Power = " + power + " Charisma = " + charisma);*

*}*

*}*

*class AnatolyVassermanStatement extends CharacterStatement {*

*public AnatolyVassermanStatement(){*

*nameCharacter = "Anatoly Vasserman";*

*power = 123443;*

*charisma = 901213411;*

*}*

*@Override*

*public void printStatement() {*

*System.out.println("Name = " + nameCharacter + " Power = " + power + " Charisma = " + charisma);*

*}*

*}*

*class YEStatement extends CharacterStatement {*

*public YEStatement(){*

*nameCharacter = "YE";*

*power = 1341134;*

*charisma = 15328475;*

*}*

*@Override*

*public void printStatement() {*

*System.out.println("Name = " + nameCharacter + " Power = " + power + " Charisma = " + charisma);*

*}*

*}*

*class FlyweightFactory {*

*private final HashMap<Integer, CharacterStatement> characters = new HashMap<>();*

*public CharacterStatement getCharacter(int characterCode){*

*CharacterStatement character = characters.get(characterCode);*

*if (character == null){*

*switch (characterCode) {*

*case 1 -> character = new ShrekStatement();*

*case 2 -> character = new AnatolyVassermanStatement();*

*case 3 -> character = new YEStatement();*

*}*

*characters.put(characterCode, character);*

*}*

*return character;*

*}*

*}*

*/\*Реализация шаблона "Приспособленец"\*/*

*public class FlyweightMain {*

*public static void main (String [] args){*

*FlyweightFactory factory = new FlyweightFactory();*

*int [] characterCodes = {1,2,3};*

*for (int nextCode : characterCodes){*

*CharacterStatement character = factory.getCharacter(nextCode);*

*character.printStatement();*

*}*

*}*

*}*

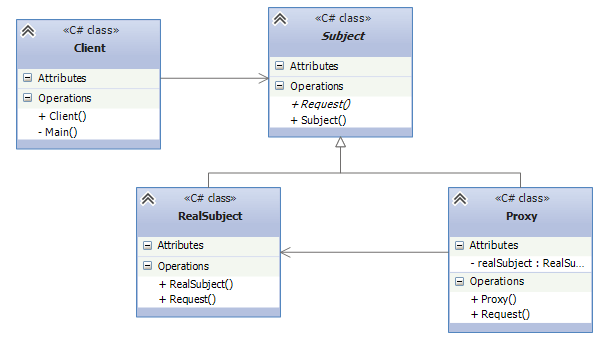
# 

# 

### Заместитель (Proxy)

Назначение:

Proxy (заместитель) — ​ является суррогатом другого объекта и контролирует доступ к нему.​



Пример кода:

*import java.util.\*;*

*import java.lang.reflect.\*;*

*class Main {*

*public static void main(String[] arguments) {*

*final Map<String, String> original = new HashMap<>();*

*InvocationHandler proxy = (obj, method, args) -> {*

*System.out.println("Invoked: " + method.getName());*

*return method.invoke(original, args);*

*};*

*Map<String, String> proxyInstance = (Map) Proxy.newProxyInstance(*

*original.getClass().getClassLoader(),*

*original.getClass().getInterfaces(),*

*proxy);*

*proxyInstance.put("key", "value");*

*System.out.println(proxyInstance.get("key"));*

*}*

*}*

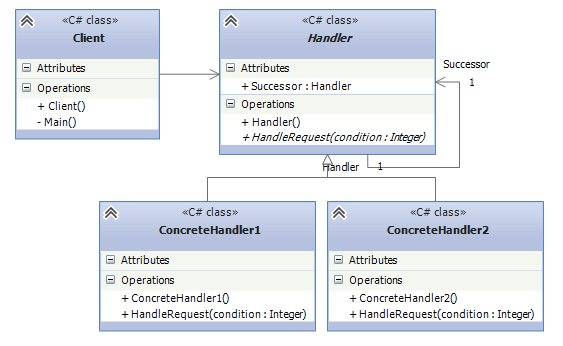
# 

### Поведенчиские паттерны

Поведенческие паттерны - они определяют алгоритмы и взаимодействие между классами и объектами, то есть их поведение.

### Цепочка обязанностей (Chain of responsibility)

Назначение: Позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким объектам. Связывает объекты-получатели в цепочку и передает запрос вдоль этой цепочки, пока его не обработают. «Цепочка обязанностей» является довольно распространенным паттерном в .NET Framework, хотя не все знают, что часто пользуются им. Цепочка обязанностей — это любое событие, аргументы которого позволяют уведомить инициатора, что событие обработано с помощью метода Handle() или путем установки свойства Handled в True.



Пример кода:

*import java.util.logging.\*;*

*class Main {*

*public static void main(String[] args) {*

*Logger logger = Logger.getLogger(Main.class.getName());*

*ConsoleHandler consoleHandler = new ConsoleHandler(){*

*@Override*

*public void publish(LogRecord record) {*

*System.out.println("LogRecord обработан");*

*}*

*};*

*logger.addHandler(consoleHandler);*

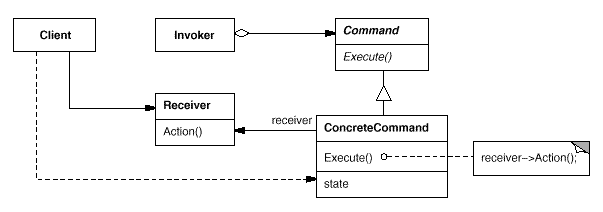
*logger.info("test");*

*}*

*}*

### Команда (Command)

Назначение: инкапсулирует запрос как объект, позволяя тем самым задавать параметры клиентов для обработки соответствующих запросов, ставить запросы в очередь или протоколировать их, а также поддерживать отмену операций.



Пример кода (из проекта по JMX):

*private void Task(String apath, String className) {*

*var path = Path.of(apath);*

*try{*

*ClassLoader loader = new URLClassLoader(new URL[] {path.toUri().toURL()});*

*var clazz = loader.loadClass(className);*

*clazz.getMethod("run").invoke(null);*

*} catch (MalformedURLException | ClassNotFoundException | NoSuchMethodException | IllegalAccessException | InvocationTargetException e) {*

*// this.status = ("error: " + Arrays.toString(e.getStackTrace()));*

*this.cancel();*

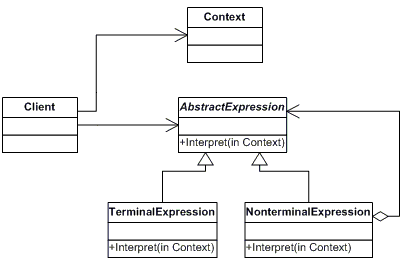
*e.printStackTrace();*

*}*

*}*

### Интерпритатор (Interpreter)

Паттерн Интерпретатор (Interpreter) определяет представление грамматики для заданного языка и интерпретатор предложений этого языка. Как правило, данный шаблон проектирования применяется для часто повторяющихся операций.



Пример кода:

*package BehavioralPatterns.Interpreter;*

*/\*Для заданного языка определяет представление его грамматики,*

*а также интерпретатор предложений этого языка\*/*

*public class InterpreterMain {*

*public static Expression getMaleExpression(){*

*Expression robert = new TerminalExpression("Robert");*

*Expression john = new TerminalExpression("John");*

*return new OrExpression(robert, john);*

*}*

*//Rule: Julie is a married women*

*public static Expression getMarriedWomanExpression(){*

*Expression julie = new TerminalExpression("Julie");*

*Expression married = new TerminalExpression("Married");*

*return new AndExpression(julie, married);*

*}*

*public static void main(String[] args) {*

*Expression isMale = getMaleExpression();*

*Expression isMarriedWoman = getMarriedWomanExpression();*

*System.out.println("John is male? " + isMale.interpret("John"));*

*System.out.println("Julie is a married women? " + isMarriedWoman.interpret("Married Julie"));*

*}*

*}*

*interface Expression {*

*boolean interpret(String context);*

*}*

*class TerminalExpression implements Expression {*

*private final String data;*

*public TerminalExpression(String data){*

*this.data = data;*

*}*

*@Override*

*public boolean interpret(String context) {*

*return context.contains(data);*

*}*

*}*

*class OrExpression implements Expression {*

*private final Expression expr1;*

*private final Expression expr2;*

*public OrExpression(Expression expr1, Expression expr2) {*

*this.expr1 = expr1;*

*this.expr2 = expr2;*

*}*

*@Override*

*public boolean interpret(String context) {*

*return expr1.interpret(context) || expr2.interpret(context);*

*}*

*}*

*class AndExpression implements Expression {*

*private final Expression expr1;*

*private final Expression expr2;*

*public AndExpression(Expression expr1, Expression expr2) {*

*this.expr1 = expr1;*

*this.expr2 = expr2;*

*}*

*@Override*

*public boolean interpret(String context) {*

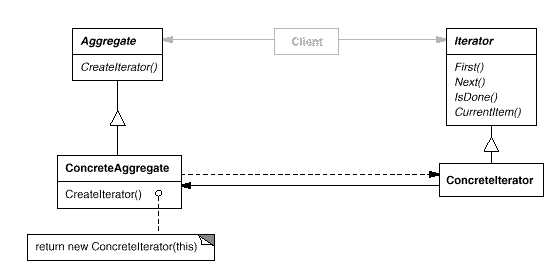
*return expr1.interpret(context) && expr2.interpret(context);*

*}*

*}*

### Итератор (Iterator)

Назначение: представляет доступ ко всем элементам составного объекта, не раскрывая его внутреннего представления



Пример кода (из проекта по JMX):

*public void stopProfiling(String name) {*

*for (Iterator<Process> iterator = processes.iterator(); iterator.hasNext(); ) {*

*Process process = iterator.next();*

*if (process.getName().equals(name)) {*

*if (process.isProfiled)*

*process.stopProfiling();*

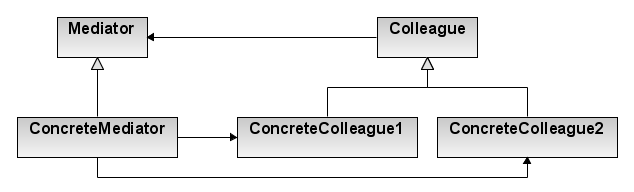
*}*

*}*

*}*

### Посредник (Mediator)

Назначение: Паттерн Посредник (Mediator) представляет такой шаблон проектирования, который обеспечивает взаимодействие множества объектов без необходимости ссылаться друг на друга. Тем самым достигается слабосвязанность взаимодействующих объектов.



Пример кода:

*import java.util.\*;*

*class Main {*

*public static void main(String[] args) {*

*Timer mediator = new Timer("Mediator");*

*TimerTask command = new TimerTask() {*

*@Override*

*public void run() {*

*System.out.println("Command pattern");*

*mediator.cancel();*

*}*

*};*

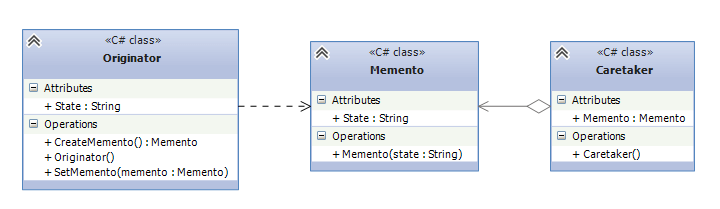
*mediator.schedule(command, 1000);*

*}*

*}*

### Хранитель (Memento)

Паттерн Хранитель (Memento) — позволяет выносить внутреннее состояние объекта за его пределы для последующего возможного восстановления объекта без нарушения принципа инкапсуляции.



Пример кода:

*package BehavioralPatterns.Memento;*

*/\*Не нарушая инкапсуляции, фиксирует и выносит за пределы объекта его внутреннее состояние,*

*так чтобы позднее можно было восстановить в нем объект\*/*

*public class MementoMain {*

*public static void main(String[] args) {*

*Originator originator = new Originator();*

*Caretaker caretaker = new Caretaker();*

*originator.setState("on");*

*System.out.printf("State is %s\n", originator.getState());*

*caretaker.setMemento(originator.saveState());*

*originator.setState("off");*

*System.out.printf("State is %s\n", originator.getState());*

*originator.restoreState(caretaker.getMemento());*

*System.out.printf("State is %s\n", originator.getState());*

*}*

*}*

*record Memento(String state) { }*

*class Caretaker {*

*private Memento memento;*

*public Memento getMemento() {*

*return memento;*

*}*

*public void setMemento(Memento memento) {*

*this.memento = memento;*

*}*

*}*

*class Originator {*

*private String state;*

*public void setState(String state) {*

*this.state = state;*

*}*

*public String getState() {*

*return state;*

*}*

*public Memento saveState() {*

*return new Memento(state);*

*}*

*public void restoreState(Memento memento) {*

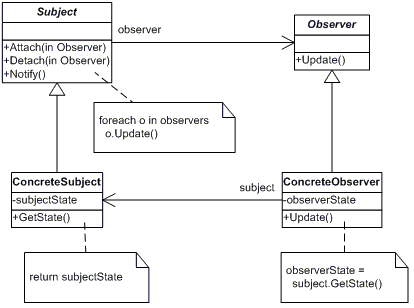
*this.state = memento.state();*

*}*

*}*

### Наблюдатель (Observer)

Назначение: определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически обновляются.



Пример кода:

*import java.util.\*;*

*class Main {*

*public static void main(String[] args) {*

*Observer observer = (obj, arg) -> {*

*System.out.println("Arg: " + arg);*

*};*

*Observable target = new Observable(){*

*@Override*

*public void notifyObservers(Object arg) {*

*setChanged();*

*super.notifyObservers(arg);*

*}*

*};*

*target.addObserver(observer);*

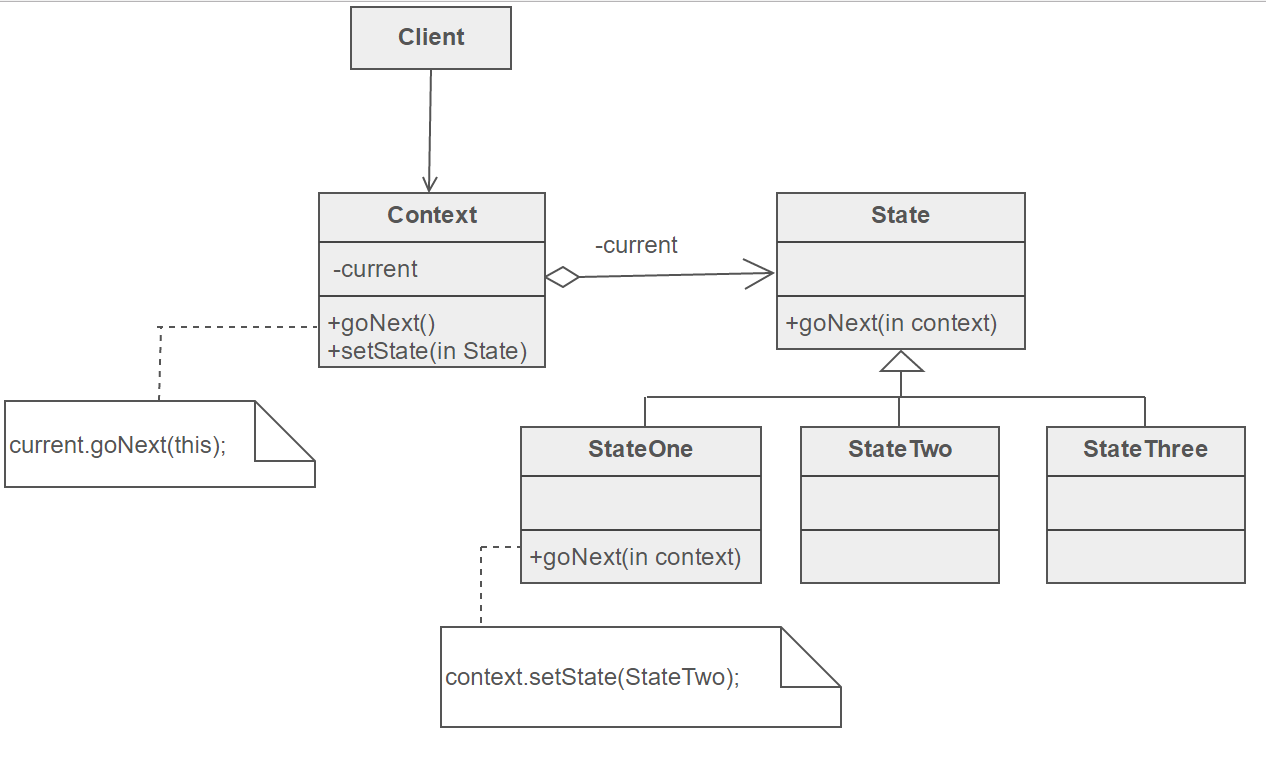
*target.notifyObservers("Hello, World!");*

*}*

*}*

### Состояние (State)

Назначение: Позволяет объекту варьировать свое поведение в зависимости от внутреннего состояния. Извне создается впечатление, что изменился класс объекта.



Пример кода:

*package BehavioralPatterns.State;*

*/\*Позволяет объекту менять свое поведение в зависимости от внутреннего состояния\*/*

*public class StateMain {*

*public static void main(String[] args) {*

*StateContext context = new StateContext();*

*context.heat();*

*context.heat();*

*context.heat();*

*context.freeze();*

*context.freeze();*

*context.freeze();*

*}*

*}*

*interface State {*

*String getName();*

*void freeze(StateContext context);*

*void heat(StateContext context);*

*}*

*class SolidState implements State {*

*private static final String NAME = "solid";*

*public String getName() {*

*return NAME;*

*}*

*public void freeze(StateContext context) {*

*System.out.println("Nothing happens.");*

*}*

*public void heat(StateContext context) {*

*context.setState(new LiquidState());*

*}*

*}*

*class LiquidState implements State {*

*private static final String NAME = "liquid";*

*public String getName() {*

*return NAME;*

*}*

*public void freeze(StateContext context) {*

*context.setState(new SolidState());*

*}*

*public void heat(StateContext context) {*

*context.setState(new GaseousState());*

*}*

*}*

*class GaseousState implements State {*

*private static final String NAME = "gaseous";*

*public String getName() {*

*return NAME;*

*}*

*public void freeze(StateContext context) {*

*context.setState(new LiquidState());*

*}*

*public void heat(StateContext context) {*

*System.out.println("Nothing happens.");*

*}*

*}*

*class StateContext {*

*private State state = new SolidState();*

*public void freeze() {*

*System.out.println("Freezing " + state.getName() + " substance...");*

*state.freeze(this);*

*}*

*public void heat() {*

*System.out.println("Heating " + state.getName() + " substance...");*

*state.heat(this);*

*}*

*public void setState(State state) {*

*System.out.println("Changing state to " + state.getName() + "...");*

*this.state = state;*

*}*

*public State getState() {*

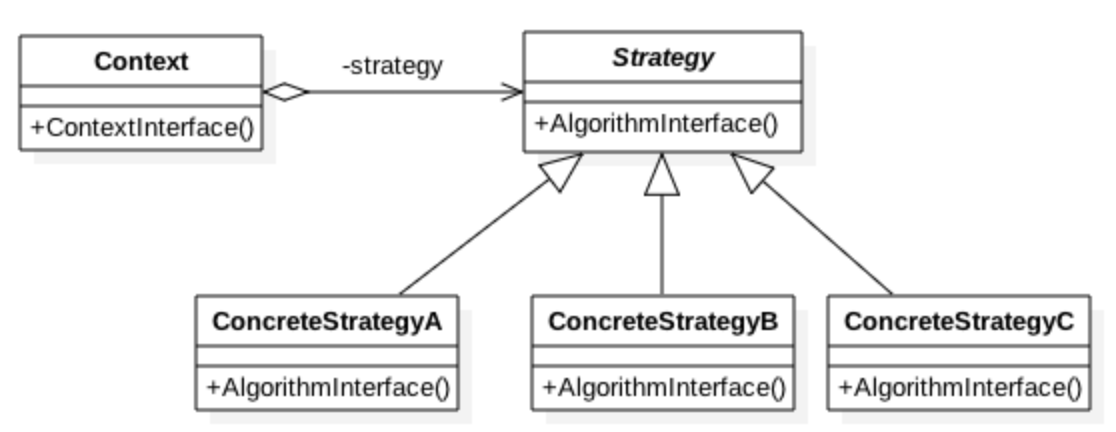
*return state;*

*}*

*}*

### Стратегия (Strategy)

Назначение: определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.



Пример кода:

*import java.util.\*;*

*class Main {*

*public static void main(String[] args) {*

*List<String> data = Arrays.asList("Moscow", "Paris", "NYC");*

*Comparator<String> comparator = Comparator.comparingInt(String::length);*

*Set dataSet = new TreeSet(comparator);*

*dataSet.addAll(data);*

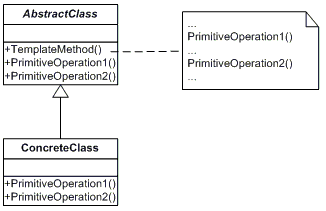
*System.out.println("Dataset : " + dataSet);*

*}*

*}*

### Шаблонный метод (Template Method)

Назначение: шаблонный метод определяет основу алгоритма и позволяет подклассам переопределять некоторые шаги алгоритма, не изменяя его структуры в целом.



Пример кода:

*package BehavioralPatterns.TemplateMethod;*

*/\*Определяет основу алгоритма и позволяет подклассам переопределить некоторые шаги алгоритма,*

*не изменяя его структуру в целом\*/*

*public class TemplateMethodMain {*

*public static void main(String[] args) {*

*final GameCode gameCode = GameCode.CHESS;*

*Game game;*

*switch (gameCode) {*

*case CHESS:*

*game = new Chess();*

*break;*

*case MONOPOLY:*

*game = new Monopoly();*

*break;*

*default:*

*throw new IllegalStateException();*

*}*

*game.playOneGame(2);*

*}*

*}*

*enum GameCode {*

*CHESS,*

*MONOPOLY*

*}*

*/\* Абстрактный класс, реализация абстрактных методов которого будет специфичной для каждого вида игры.*

*\**

*\* Файл Game.java*

*\* \*/*

*abstract class Game {*

*private int playersAmount;*

*protected abstract void initializeGame();*

*protected abstract void playGame();*

*protected abstract void endGame();*

*protected abstract void printWinner();*

*public final void playOneGame(int playersAmount) {*

*setPlayersAmount(playersAmount);*

*initializeGame();*

*playGame();*

*endGame();*

*printWinner();*

*}*

*public void setPlayersAmount(int playersAmount) {*

*this.playersAmount = playersAmount;*

*}*

*}*

*/\* Игра "Шахматы". Специфически только для шахмат реализует методы класса Game.*

*\**

*\* Файл Chess.java*

*\* \*/*

*class Chess extends Game {*

*@Override*

*protected void initializeGame() {*

*// chess specific initialization actions*

*}*

*@Override*

*protected void playGame() {*

*// chess specific play actions*

*}*

*@Override*

*protected void endGame() {*

*// chess specific actions to end a game*

*}*

*@Override*

*protected void printWinner() {*

*// chess specific actions to print winner*

*}*

*}*

*/\* Игра "Монополия". Специфически только для монополии реализует методы класса Game.*

*\**

*\* Файл Monopoly.java*

*\* \*/*

*class Monopoly extends Game {*

*@Override*

*protected void initializeGame() {*

*// monopoly specific initialization actions*

*}*

*@Override*

*protected void playGame() {*

*// monopoly specific play actions*

*}*

*@Override*

*protected void endGame() {*

*// monopoly specific actions to end a game*

*}*

*@Override*

*protected void printWinner() {*

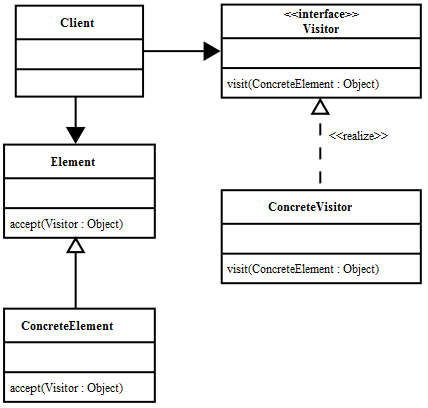
*// monopoly specific actions to print winner*

*}*

*}*

### Посетитель (Visitor)

Назначение: описывает операцию, выполняемую с каждым объектом из некоторой иерархии классов. Паттерн «Посетитель» позволяет определить новую операцию, не изменяя классов этих объектов.



Пример кода:

*import java.nio.file.\*;*

*import java.nio.file.attribute.\*;*

*import java.io.\*;*

*class Main {*

*public static void main(String[] args) {*

*SimpleFileVisitor visitor = new SimpleFileVisitor() {*

*@Override*

*public FileVisitResult visitFile(Object file, BasicFileAttributes attrs) throws IOException {*

*System.out.println("File:" + file.toString());*

*return FileVisitResult.CONTINUE;*

*}*

*};*

*Path pathSource = Paths.get(System.getProperty("java.io.tmpdir"));*

*try {*

*Files.walkFileTree(pathSource, visitor);*

*} catch (AccessDeniedException e) {*

*// skip*

*} catch (IOException e) {*

*// Do something*

*}*

*}*

*}*

### Практическая часть

Ссылка на проект программы под управлением JMX с использованием паттернов программирования: <https://github.com/Karkarych08/JMX_Project> .

### ЗАКЛЮЧЕНИЕ

В ходе данной курсовой работы были изучены и использованы паттерны программирования. Полученные знания и навыки их использования обязательно помогут нам писать более качественные, читаемые и профессиональные программы.

### СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. GoF. Паттерны объектно-ориентированного проектирования / Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес - 2020г. - 448с.

2. https://bool.dev/blog/detail/gof-design-patterns - сайт с описанием принципов работы паттернов програмирования.

3. https://javarush.ru/groups/posts/2267-patternih-proektirovanija-v-java - сайт с примерами использования паттернов програмирования.