Практическая работа №7

Паттерны проектирования, структурные паттерны.

### Теоретическое введение

Паттерны проектирования — это повторяющиеся решения типичных задач проектирования программного обеспечения. Они помогают сделать код более гибким, поддерживаемым и повторно используемым. Порождающие паттерны проектирования играют важную роль в управлении процессом создания объектов, предоставляя гибкие способы инкапсуляции создания объектов и обеспечения их корректного использования.

Структурные паттерны (Structural Patterns) — это группа паттернов проектирования, которые фокусируются на способах организации классов и объектов в более крупные структуры. Они обеспечивают гибкость и удобство взаимодействия между компонентами системы, способствуя уменьшению связности и упрощению поддержки кода. Эти паттерны позволяют строить сложные структуры, сохраняя при этом простоту и читаемость кода.

Основные задачи структурных паттернов:

1. Упрощение архитектуры: Структурные паттерны позволяют объединять объекты и классы так, чтобы они работали как единое целое, сохраняя при этом чёткое разделение обязанностей.
2. Повышение гибкости системы: благодаря абстрагированию взаимодействия между компонентами системы можно изменять её структуру без затрагивания остального кода.
3. Улучшение повторного использования: Компоненты можно комбинировать и использовать повторно в различных частях приложения.

### Основные структурные паттерны

1. Adapter (Адаптер)

Цель: позволяет объектам с несовместимыми интерфейсами работать вместе, преобразуя интерфейс одного класса в интерфейс, ожидаемый клиентом.

Применение: используется для интеграции старого и нового кода, обеспечения совместимости библиотек или API.

Пример реализации приведен на Листинге 13.1.

#### Листинг 13.1 – Пример реализации паттерна Adapter

java

*// Новый интерфейс USB-C*

interface UsbC {

void connectUsbC();

}

*// Старый интерфейс USB*

class Usb {

void connectUsb() {

System.out.println("Connected using USB");

}

}

*// Адаптер для подключения USB через USB-C*

class UsbToUsbCAdapter implements UsbC {

private Usb usb;

public UsbToUsbCAdapter(Usb usb) {

this.usb = usb;

}

public void connectUsbC() {

System.out.println("Adapter converts USB to USB-C");

usb.connectUsb();

}

}

*// Тестирование адаптера*

public class AdapterExample {

public static void main(String[] args) {

Usb usb = new Usb();

UsbC usbC = new UsbToUsbCAdapter(usb);

usbC.connectUsbC();

}

}

1. Bridge (Мост)

Цель: разделяет абстракцию и её реализацию, позволяя изменять их независимо друг от друга.

Применение: используется, когда необходимо расширять функциональность системы независимо от её реализации, например, при создании различных типов объектов с одинаковым интерфейсом.

Пример реализации приведен на Листинге 13.2.

#### Листинг 13.2 – Пример реализации паттерна Bridge

java

*// Интерфейс реализации*

interface Moveable {

void move();

}

*// Конкретная реализация для наземного транспорта*

class GroundMove implements Moveable {

public void move() {

System.out.println("Moving on the ground");

}

}

*// Абстракция транспорта*

abstract class Vehicle {

protected Moveable moveable;

protected Vehicle(Moveable moveable) {

this.moveable = moveable;

}

public abstract void operate();

}

*// Конкретная абстракция автомобиля*

class Car extends Vehicle {

public Car(Moveable moveable) {

super(moveable);

}

public void operate() {

System.out.print("Car is ");

moveable.move();

}

}

*// Тестирование моста*

public class BridgeExample {

public static void main(String[] args) {

Vehicle car = new Car(new GroundMove());

car.operate();

}

}

1. Composite (Компоновщик)

Цель: позволяет сгруппировать объекты в древовидную структуру и обрабатывать их единообразно.

Применение: используется для представления иерархий «часть-целое», например, в файловых системах, структурах меню или графических интерфейсах.

Пример реализации приведен на Листинге 13.3.

#### Листинг 13.3 – Пример реализации паттерна Composite

java

import java.util.ArrayList;

import java.util.List;

*// Интерфейс для файловой системы*

interface FileSystemComponent {

void display();

}

*// Лист — файл*

class File implements FileSystemComponent {

private String name;

public File(String name) {

this.name = name;

}

public void display() {

System.out.println("File: " + name);

}

}

*// Контейнер — папка*

class Directory implements FileSystemComponent {

private String name;

private List<FileSystemComponent> components = new ArrayList<>();

public Directory(String name) {

this.name = name;

}

public void add(FileSystemComponent component) {

components.add(component);

}

public void display() {

System.out.println("Directory: " + name);

for (FileSystemComponent component : components) {

component.display();

}

}

}

*// Тестирование компоновщика*

public class CompositeExample {

public static void main(String[] args) {

Directory root = new Directory("Root");

root.add(new File("File1.txt"));

Directory subDir = new Directory("SubDir");

subDir.add(new File("File2.txt"));

root.add(subDir);

root.display();

}

}

1. Decorator (Декоратор)

Цель: динамически добавляет объекту новые обязанности, не изменяя его код.

Применение: используется для расширения функциональности объектов, когда невозможно изменить исходный код класса или нужно комбинировать разные функции.

Пример реализации приведен на Листинге 13.4.

#### Листинг 13.4 – Пример реализации паттерна Decorator

java

*// Интерфейс уведомлений*

interface Notification {

void send();

}

*// Базовая реализация уведомлений*

class BasicNotification implements Notification {

public void send() {

System.out.println("Sending basic notification");

}

}

*// Декоратор*

abstract class NotificationDecorator implements Notification {

protected Notification decoratedNotification;

public NotificationDecorator(Notification decoratedNotification) {

this.decoratedNotification = decoratedNotification;

}

public void send() {

decoratedNotification.send();

}

}

*// Конкретный декоратор для добавления SMS-уведомления*

class SmsNotificationDecorator extends NotificationDecorator {

public SmsNotificationDecorator(Notification decoratedNotification) {

super(decoratedNotification);

}

public void send() {

decoratedNotification.send();

System.out.println("Sending SMS notification");

}

}

*// Тестирование декоратора*

public class DecoratorExample {

public static void main(String[] args) {

Notification notification = new SmsNotificationDecorator(new BasicNotification());

notification.send();

}

}

1. Facade (Фасад)

Цель: предоставляет унифицированный интерфейс для взаимодействия с сложной подсистемой.

Применение: используется для упрощения доступа к сложным системам или библиотекам, предоставляя один интерфейс для управления несколькими компонентами.

Пример реализации приведен на Листинге 13.5.

#### Листинг 13.5 – Пример реализации паттерна Facade

java

*// Подсистемы*

class CPU {

void start() { System.out.println("CPU is starting"); }

}

class Memory {

void load() { System.out.println("Memory is loading"); }

}

class HardDrive {

void readData() { System.out.println("Reading data from Hard Drive"); }

}

*// Фасад для управления компьютером*

class ComputerFacade {

private CPU cpu;

private Memory memory;

private HardDrive hardDrive;

public ComputerFacade() {

this.cpu = new CPU();

this.memory = new Memory();

this.hardDrive = new HardDrive();

}

public void startComputer() {

cpu.start();

memory.load();

hardDrive.readData();

}

}

*// Тестирование фасада*

public class FacadeExample {

public static void main(String[] args) {

ComputerFacade computer = new ComputerFacade();

computer.startComputer();

}

}

1. Flyweight (Приспособленец)

Цель: оптимизирует использование памяти путём разделения общего состояния объектов и уменьшения количества создаваемых экземпляров.

Применение: Используется при работе с большим количеством мелких объектов, таких как символы текста или графические элементы, для уменьшения потребления ресурсов.

Пример реализации приведен на Листинге 13.6.

#### Листинг 13.6 – Пример реализации паттерна Flyweight

java

import java.util.HashMap;

*// Интерфейс символов*

interface Character {

void printCharacter();

}

*// Конкретная реализация символа*

class ConcreteCharacter implements Character {

private char symbol;

public ConcreteCharacter(char symbol) {

this.symbol = symbol;

}

public void printCharacter() {

System.out.println("Character: " + symbol);

}

}

*// Фабрика символов*

class CharacterFactory {

private HashMap<Character, ConcreteCharacter> characterMap = new HashMap<>();

public Character getCharacter(char symbol) {

if (!characterMap.containsKey(symbol)) {

characterMap.put(symbol, new ConcreteCharacter(symbol));

System.out.println("Creating character: " + symbol);

}

return characterMap.get(symbol);

}

}

*// Тестирование Flyweight*

public class FlyweightExample {

public static void main(String[] args) {

CharacterFactory factory = new CharacterFactory();

factory.getCharacter('A').printCharacter();

factory.getCharacter('B').printCharacter();

factory.getCharacter('A').printCharacter(); *// Повторное использование объекта*

}

}

1. Proxy (Заместитель)

Цель: Контролирует доступ к другому объекту, предоставляя его замену или дополнительный уровень управления.

Применение: Используется для создания отложенной инициализации объектов, управления доступом или выполнения дополнительных операций (например, логирование, кеширование).

Пример реализации приведен на Листинге 13.7.

#### Листинг 13.7 – Пример реализации паттерна Proxy

java

*// Интерфейс документа*

interface Document {

void display();

}

*// Реальный документ*

class RealDocument implements Document {

private String fileName;

public RealDocument(String fileName) {

this.fileName = fileName;

loadFromDisk();

}

private void loadFromDisk() {

System.out.println("Loading document: " + fileName);

}

public void display() {

System.out.println("Displaying document: " + fileName);

}

}

*// Прокси для защиты документа*

class DocumentProxy implements Document {

private RealDocument realDocument;

private String fileName;

public DocumentProxy(String fileName) {

this.fileName = fileName;

}

public void display() {

if (realDocument == null) {

realDocument = new RealDocument(fileName);

}

realDocument.display();

}

}

*// Тестирование Proxy*

public class ProxyExample {

public static void main(String[] args) {

Document document = new DocumentProxy("SecretDoc.pdf");

System.out.println("First call to display:");

document.display();

System.out.println("\nSecond call to display:");

document.display();

}

}

## Практическое задание

В данной практической работе представлено 7 вариантов заданий. Выбор варианта осуществляется в соответствии с порядковым номером студента в списке. Если порядковый номер студента равен 1, выполняется задание варианта №1; если порядковый номер равен 7 — выполняется вариант №7. В случае, если порядковый номер превышает количество вариантов (например, 8 или более), задание выбирается циклически, начиная с варианта №1.

### Вариант №1

Adapter – Задание

Создать систему для воспроизведения аудиофайлов разных форматов (MP3 и WAV). Разработать класс-адаптер, который позволит пользователю воспроизводить WAV-файлы через существующий интерфейс MP3-плеера.

Требования:

Интерфейс MediaPlayer должен поддерживать метод play().

Реализовать классы для воспроизведения MP3 и WAV-файлов.

Использовать адаптер для воспроизведения WAV-файлов через MP3-плеер.

### Вариант №2

Bridge – Задание

Разработать систему рисования фигур, таких как круг и квадрат, с различными цветами (красный и зелёный). Абстракция должна быть отделена от реализации цвета.

Требования:

Создать интерфейс DrawAPI для рисования.

Реализовать классы для фигур, использующие мост для выбора цвета.

Реализация цвета должна быть независимой от реализации фигур.

### Вариант №3

Composite – Задание

Создать иерархическую систему управления файлами. В системе должны быть директории и файлы, причём каждая директория может содержать как файлы, так и другие директории.

Требования:

Реализовать интерфейс FileComponent с методами для добавления и отображения содержимого.

Создать классы для файлов и директорий.

Организовать древовидную структуру иерархии файлов.

### Вариант №4

Decorator – Задание

Создать систему для расширения функционала базового текстового редактора. Например, добавьте декораторы для форматирования текста (жирный, курсив и подчёркнутый текст).

Требования:

Реализовать интерфейс Text с методом display().

Создать базовый класс для простого текста и декораторы для добавления форматирования.

Декораторы должны быть применимы в любом порядке.

### Вариант №5

Facade – Задание

Разработать систему управления домашним кинотеатром, включающую DVD-плеер, проектор и звуковую систему. Используйте фасад для упрощения управления этими компонентами.

Требования:

Создать классы для компонентов кинотеатра.

Реализовать класс-фасад HomeTheaterFacade, предоставляющий методы для запуска и завершения просмотра фильма.

Пользователь должен взаимодействовать только с фасадом.

### Вариант №6

Flyweight – Задание

Разработать приложение для создания множества объектов круга разного цвета. Оптимизируйте память, используя паттерн Flyweight для повторного использования объектов с одинаковым цветом.

Требования:

Создать класс Circle, который будет содержать общий (цвет) и уникальный (координаты) состояния.

Реализовать фабрику для управления объектами и переиспользования кругов одного цвета.

Проверить эффективность создания объектов с помощью вывода сообщений о создании.

### Вариант №7

Proxy – Задание

Создать систему для загрузки изображений с использованием прокси. Изображение должно загружаться только тогда, когда оно необходимо для отображения.

Требования:

Реализовать интерфейс Image с методом display().

Создать классы для реального изображения и прокси-изображения.

Проверить, что изображение загружается только при первом вызове метода display().

**Паттерны проектирования, поведенческие паттерны**

## Теоретическое введение

Паттерны проектирования — это повторяющиеся решения типичных задач проектирования программного обеспечения. Они помогают сделать код более гибким, поддерживаемым и повторно используемым. Порождающие паттерны проектирования играют важную роль в управлении процессом создания объектов, предоставляя гибкие способы инкапсуляции создания объектов и обеспечения их корректного использования.

Поведенческие паттерны (Creational Patterns) — Поведенческие паттерны проектирования описывают способы взаимодействия объектов и классов друг с другом, фокусируясь на алгоритмах и распределении обязанностей. Эти паттерны помогают обеспечить эффективное и гибкое взаимодействие между компонентами системы, упрощая коммуникацию и управление поведением объектов.

Основные задачи поведенческих паттернов:

1. Управление взаимодействием объектов: Обеспечение гибкости и надёжности при взаимодействии между объектами. Поведенческие паттерны определяют, как объекты сотрудничают и передают данные друг другу.
2. Инкапсуляция алгоритмов: позволяют отделить алгоритмы от классов, использующих их, что упрощает их замену или изменение без влияния на остальную систему.
3. Разделение обязанностей: чётко определяют роли объектов и классов в процессе выполнения операции, улучшая читаемость и поддержку кода.

## Основные поведенческие паттерны

1. Chain of Responsibility (Цепочка обязанностей) Цель: позволяет передавать запрос последовательно по цепочке обработчиков. Каждый обработчик может обработать запрос или передать его следующему.

Применение: Обработка событий или запросов в веб-приложениях или логгер, например с уровнями INFO, BEGUG, ERROR.

Пример реализации приведен на Листинге 14.1.

### Листинг 14.1 – Пример реализации паттерна Chain on Responbillity

java

abstract class Logger {

protected Logger nextLogger;

public void setNextLogger(Logger nextLogger) {

this.nextLogger = nextLogger;

}

public void logMessage(String message) {

if (nextLogger != null) {

nextLogger.logMessage(message);

}

}

}

class ConsoleLogger extends Logger {

public void logMessage(String message) {

System.out.println("Console Logger: " + message);

}

}

1. Command (Команда) Цель: инкапсулирует запрос как объект, позволяя параметризовать клиентов различными запросами.

Применение: Реализация отмены действий (Undo/Redo), выполнение очереди команд.

Пример реализации приведен на Листинге 14.2.

### Листинг 14.2 – Пример реализации паттерна Command

java

interface Command { void execute(); }

class Light {

public void on() { System.out.println("Light is ON"); }

}

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) { this.light = light; }

public void execute() { light.on(); }

}

1. Iterator (Итератор) Цель: обеспечивает последовательный доступ к элементам коллекции без раскрытия её внутренней структуры.

Применение: Перебор элементов коллекции (списки, массивы).

Пример реализации приведен на Листинге 14.3.

### Листинг 14.3 – Пример реализации паттерна Iterator

java

class NameRepository {

private String[] names = {"Alice", "Bob"};

public Iterator<String> getIterator() {

return new ArrayIterator();

}

private class ArrayIterator implements Iterator<String> {

int index = 0;

public boolean hasNext() { return index < names.length; }

public String next() { return names[index++]; }

}

}

1. Mediator (Посредник) Цель: Обеспечивает централизованное взаимодействие между объектами, уменьшает их связанность.

Применение: Управление элементами интерфейса или модулями системы.

Пример реализации приведен на Листинге 14.4.

### Листинг 14.4 – Пример реализации паттерна Mediator

java

class ChatRoom {

public static void showMessage(String user, String message) {

System.out.println(user + ": " + message);

}

}

1. Memento (Снимок) Цель: позволяет сохранять и восстанавливать состояние объекта, не раскрывая его внутреннего устройства.

Применение: Реализация функции сохранения и восстановления, например, текстовый редактор.

Пример реализации приведен на Листинге 14.5.

### Листинг 14.5 – Пример реализации паттерна Momento

java

class TextEditor {

private String text;

public void setText(String text) { this.text = text; }

public String save() { return text; }

}

1. Observer (Наблюдатель) Цель: описывает зависимость «один ко многим», где изменение одного объекта обновляет все зависимые.

Применение: Реализация подписки на события, например, уведомления.

Пример реализации приведен на Листинге 14.6.

### Листинг 14.6 – Пример реализации паттерна Observer

java

interface Observer { void update(); }

class User implements Observer {

public void update() { System.out.println("User notified"); }

}

1. State (Состояние) Цель: позволяет объекту изменять своё поведение при изменении состояния.

Применение: Управление состояниями объектов, например, автомат состояний заказов.

Пример реализации приведен на Листинге 14.7.

### Листинг 14.7 – Пример реализации паттерна State

java

interface State { void handle(); }

class NewOrderState implements State {

public void handle() { System.out.println("Processing new order"); }

}

1. Strategy (Стратегия) Цель: определяет семейство алгоритмов и делает их взаимозаменяемыми.

Применение: Выбор алгоритма на основе условий, например, разные способы сортировки.

Пример реализации приведен на Листинге 14.8.

### Листинг 14.8 – Пример реализации паттерна Strategy

java

interface DiscountStrategy { double applyDiscount(double price); }

class NoDiscount implements DiscountStrategy {

public double applyDiscount(double price) { return price; }

}

1. Template Method (Шаблонный метод) Цель: определяет основу алгоритма, позволяя подклассам переопределять шаги.

Применение: Создание алгоритмов с изменяемыми шагами, например, обработка данных.

Пример реализации приведен на Листинге 14.9.

### Листинг 14.9 – Пример реализации паттерна Template Method

java

abstract class Beverage {

final void prepare() {

boilWater();

brew();

}

abstract void brew();

void boilWater() { System.out.println("Boiling water"); }

}

1. Visitor (Посетитель) Цель: добавляет новые операции для объектов структуры без изменения их классов.

Применение: Выполнение операций над элементами сложной структуры, например, обработка элементов структуры документа.

Пример реализации приведен на Листинге 14.10.

### Листинг 14.10 – Пример реализации паттерна Visitor

java

interface Element { void accept(Visitor visitor); }

class Paragraph implements Element {

public void accept(Visitor visitor) { visitor.visit(this); }

}

## Практическое задание

В данной практической работе представлено 10 вариантов заданий. Выбор варианта осуществляется в соответствии с порядковым номером студента в списке. Если порядковый номер студента равен 1, выполняется задание варианта №1; если порядковый номер равен 10 — выполняется вариант №10. В случае, если порядковый номер превышает количество вариантов (например, 11 или более), задание выбирается циклически, начиная с варианта №1.

### Вариант №1

Chain of Responsibility – Задание

Реализовать систему обработки запросов разного уровня важности (например, логирование или обработка заявок). Каждый запрос должен обрабатываться соответствующим обработчиком.

Требования:

* Создать обработчики для уровней сообщений (INFO, WARNING, ERROR, CRITICAL, DEBUG).
* Реализовать возможность передачи запроса по цепочке обработчиков.
* Продемонстрировать обработку запроса разного уровня.

### Вариант №2

Command – Задание

Разработать систему управления бытовыми приборами (например, телевизор или кондиционер). Команды включения и выключения должны быть инкапсулированы в объекты.

Требования:

* Создать интерфейс Command с методом execute().
* Реализовать команды TurnOnCommand и TurnOffCommand.
* Реализовать класс RemoteControl, который вызывает команды.

### Вариант №3

Iterator – Задание

Создать систему для перебора элементов коллекции (например, список студентов). Итератор должен предоставлять доступ к элементам без раскрытия структуры.

Требования:

* Создать интерфейс Iterator с методами hasNext() и next().
* Реализовать класс коллекции StudentList.
* Реализовать итератор для перебора студентов.

### Вариант №4

Mediator – Задание

Реализовать систему чата, где несколько пользователей могут обмениваться сообщениями через посредника.

Требования:

* Создать интерфейс Mediator для координации сообщений.
* Реализовать классы User, которые будут отправлять и получать сообщения через посредника.
* Продемонстрировать обмен сообщениями между пользователями.

### Вариант №5

Memento – Задание

Создать текстовый редактор с возможностью сохранения и восстановления состояния текста.

Требования:

* Реализовать класс TextEditor для редактирования текста.
* Реализовать создание снимка состояния (Memento).
* Добавить возможность восстановления состояния редактора.

### Вариант №6

Observer – Задание

Реализовать систему уведомлений для пользователей. Когда происходит событие (например, изменение состояния объекта), все подписчики должны получать уведомление.

Требования:

* Создать интерфейс Observer и класс Subject для управления подписчиками.
* Реализовать класс User, который будет получать уведомления.
* Проверить отправку уведомлений подписчикам при изменении состояния.

### Вариант №7

State – Задание

Создать систему управления состоянием заказа в интернет-магазине (например, New, Processing, Shipped).

Требования:

* Реализовать интерфейс State с методом handle().
* Создать классы для каждого состояния заказа.
* Продемонстрировать переходы между состояниями.

### Вариант №8

Strategy – Задание

Реализовать систему расчёта стоимости доставки. Пользователь может выбрать разные стратегии доставки (например, стандартная, экспресс, курьерская).

Требования:

* Создать интерфейс DeliveryStrategy с методом calculateCost().
* Реализовать несколько стратегий доставки.
* Продемонстрировать выбор стратегии и расчёт стоимости.

### Вариант №9

Template Method – Задание

Создать систему для приготовления напитков (например, чай и кофе). Шаблонный метод должен определять последовательность шагов, а подклассы — реализовать конкретные шаги.

Требования:

* Создать абстрактный класс Beverage с шаблонным методом prepare().
* Реализовать классы Tea и Coffee.
* Продемонстрировать приготовление напитков с использованием шаблонного метода.

### Вариант №10

Visitor – Задание

Реализовать систему для обработки элементов структуры (например, элементы документа). Каждая операция обработки должна быть вынесена в отдельный класс.

Требования:

* Создать интерфейс Visitor с методами для обработки различных элементов.
* Реализовать классы элементов (Paragraph, Image).
* Продемонстрировать выполнение операций над элементами через посетителя.