Паттерны проектирования, структурные паттерны: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy.

13. Паттерны проектирования, порождающие 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

```
// Интерфейс MediaPlayer
interface MediaPlayer {
    void play(String audioType, String fileName);
}
// Интерфейс AdvancedMediaPlayer
interface AdvancedMediaPlayer {
    void playMp4(String fileName);
}
// Реализация AdvancedMediaPlayer
class Mp4Player implements AdvancedMediaPlayer {
    public void playMp4(String fileName) {
        System.out.println("Playing mp4 file: " + fileName);
    }
}
// Адаптер для AdvancedMediaPlayer
class MediaAdapter implements MediaPlayer {
    AdvancedMediaPlayer advancedMusicPlayer;
    public MediaAdapter(String audioType) {
        if (audioType.equalsIgnoreCase("mp4")) {
            advancedMusicPlayer = new Mp4Player();
        }
    }
    public void play(String audioType, String fileName) {
        if (audioType.equalsIgnoreCase("mp4")) {
            advancedMusicPlayer.playMp4(fileName);
        }
    }
}
// Тестирование адаптера
public class AdapterPatternDemo {
    public static void main(String[] args) {
        MediaPlayer audioPlayer = new MediaAdapter("mp4");
```

```
        audioPlayer.play("mp4", "song.mp4");
    }
}
```

2. 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

```
// Интерфейс реализации
interface DrawAPI {
    void drawCircle(int radius, int x, int y);
}

// Конкретная реализация 1
class RedCircle implements DrawAPI {
    public void drawCircle(int radius, int x, int y) {
        System.out.println("Drawing Circle[ color: red, radius: 
" + radius + ", x: " + x + ", y: " + y + "]");
    }
}

// Абстракция
abstract class Shape {
    protected DrawAPI drawAPI;

    protected Shape(DrawAPI drawAPI) {
        this.drawAPI = drawAPI;
    }

    public abstract void draw();
}

// Расширение абстракции
```

```
class Circle extends Shape {
    private int x, y, radius;

    public Circle(int x, int y, int radius, DrawAPI drawAPI) {
        super(drawAPI);
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    public void draw() {
        drawAPI.drawCircle(radius, x, y);
    }
}

// Тестирование моста
public class BridgePatternDemo {
    public static void main(String[] args) {
        Shape redCircle = new Circle(100, 100, 10, new
RedCircle());
        redCircle.draw();
    }
}
```

3. 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

```
import java.util.ArrayList;
import java.util.List;

// Класс Component
class Employee {
    private String name;
    private String position;
    private List<Employee> subordinates;

    public Employee(String name, String position) {
        this.name = name;
        this.position = position;
        subordinates = new ArrayList<>();
    }

    public void add(Employee e) {
        subordinates.add(e);
    }

    public void remove(Employee e) {
        subordinates.remove(e);
    }

    public List<Employee> getSubordinates() {
        return subordinates;
    }
```

```
    public String toString() {
        return "Employee :[ Name : " + name + ", Position : " +
position + " ]";
    }
}

// Тестирование компоновщика
public class CompositePatternDemo {
    public static void main(String[] args) {
        Employee CEO = new Employee("John", "CEO");
        Employee headSales = new Employee("Robert", "Head Sales");
        Employee salesExecutive1 = new Employee("Laura", "Sales
Executive");

        CEO.add(headSales);
        headSales.add(salesExecutive1);

        System.out.println(CEO);
        for (Employee headEmployee : CEO.getSubordinates()) {
            System.out.println(headEmployee);
            for (Employee employee :
headEmployee.getSubordinates()) {
                System.out.println(employee);
            }
        }
    }
}
```

4. 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

```
// Интерфейс Component
interface Shape {
    void draw();
}

// Реализация Component
class Circle implements Shape {
    public void draw() {
        System.out.println("Shape: Circle");
    }
}

// Базовый декоратор
abstract class ShapeDecorator implements Shape {
    protected Shape decoratedShape;

    public ShapeDecorator(Shape decoratedShape) {
        this.decoratedShape = decoratedShape;
    }

    public void draw() {
        decoratedShape.draw();
    }
}
```

```
// Конкретный декоратор
class RedShapeDecorator extends ShapeDecorator {
    public RedShapeDecorator(Shape decoratedShape) {
        super(decoratedShape);
    }

    public void draw() {
        decoratedShape.draw();
        setRedBorder(decoratedShape);
    }

    private void setRedBorder(Shape decoratedShape) {
        System.out.println("Border Color: Red");
    }
}

// Тестирование декоратора
public class DecoratorPatternDemo {
    public static void main(String[] args) {
        Shape circle = new Circle();
        Shape redCircle = new RedShapeDecorator(new Circle());

        System.out.println("Normal Circle:");
        circle.draw();

        System.out.println("\nRed Decorated Circle:");
        redCircle.draw();
    }
}
```

5. 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

```
class DVDPlayer {
    public void on() { System.out.println("DVD Player is ON"); }
    public void play(String movie) { System.out.println("Playing movie: " + movie); }
    public void off() { System.out.println("DVD Player is OFF"); }
}

class Projector {
    public void on() { System.out.println("Projector is ON"); }
    public void setInput(String input) { System.out.println("Projector input set to: " + input); }
    public void off() { System.out.println("Projector is OFF"); }
}

class SoundSystem {
    public void on() { System.out.println("Sound System is ON"); }
    public void setVolume(int level) { System.out.println("Volume set to: " + level); }
    public void off() { System.out.println("Sound System is OFF"); }
}

class HomeTheaterFacade {
    private DVDPlayer dvd;
    private Projector projector;
    private SoundSystem soundSystem;
```

```
    public HomeTheaterFacade(DVDPlayer dvd, Projector projector,
SoundSystem soundSystem) {
        this.dvd = dvd;
        this.projector = projector;
        this.soundSystem = soundSystem;
    }

    public void watchMovie(String movie) {
        dvd.on();
        projector.on();
        projector.setInput("DVD");
        soundSystem.on();
        soundSystem.setVolume(10);
        dvd.play(movie);
    }

    public void endMovie() {
        dvd.off();
        projector.off();
        soundSystem.off();
    }
}

public class FacadePatternDemo {
    public static void main(String[] args) {
        HomeTheaterFacade homeTheater = new HomeTheaterFacade(new
DVDPlayer(), new Projector(), new SoundSystem());
        homeTheater.watchMovie("Inception");
        homeTheater.endMovie();
    }
}
```

6. 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

```
import java.util.HashMap;

// Интерфейс Shape
interface Shape {
    void draw();
}

// Конкретный класс Circle
class Circle implements Shape {
    private String color;
    private int x;
    private int y;
    private int radius;

    public Circle(String color) {
        this.color = color;
    }

    public void setX(int x) {
        this.x = x;
    }

    public void setY(int y) {
        this.y = y;
    }
```

```
    public void setRadius(int radius) {
        this.radius = radius;
    }

    public void draw() {
        System.out.println("Circle: Draw() [Color : " + color + ", x : " + x + ", y :" + y + ", radius :" + radius);
    }
}

// Фабрика для создания и управления объектами
class ShapeFactory {
    private static final HashMap<String, Shape> circleMap = new HashMap<>();

    public static Shape getCircle(String color) {
        Circle circle = (Circle) circleMap.get(color);

        if (circle == null) {
            circle = new Circle(color);
            circleMap.put(color, circle);
            System.out.println("Creating circle of color: " + color);
        }
        return circle;
    }
}

// Тестирование Flyweight
public class FlyweightPatternDemo {
    private static final String colors[] = { "Red", "Green", "Blue", "White", "Black" };

    public static void main(String[] args) {
        for (int i = 0; i < 10; ++i) {
```

```
            Circle circle = (Circle)
ShapeFactory.getCircle(getRandomColor());
            circle.setX(getRandomX());
            circle.setY(getRandomY());
            circle.setRadius(100);
            circle.draw();
        }
    }

    private static String getRandomColor() {
        return colors[(int) (Math.random() * colors.length)];
    }

    private static int getRandomX() {
        return (int) (Math.random() * 100);
    }

    private static int getRandomY() {
        return (int) (Math.random() * 100);
    }
}
```

7. 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

```
// Интерфейс для изображения
interface Image {
    void display();
}

// Реальный класс для изображения
class RealImage implements Image {
    private String fileName;

    public RealImage(String fileName) {
        this.fileName = fileName;
        loadFromDisk(fileName);
    }

    private void loadFromDisk(String fileName) {
        System.out.println("Loading " + fileName);
    }

    public void display() {
        System.out.println("Displaying " + fileName);
    }
}

// Прокси-класс для изображения
class ProxyImage implements Image {
    private RealImage realImage;
```

```
    private String fileName;

    public ProxyImage(String fileName) {
        this.fileName = fileName;
    }

    public void display() {
        if (realImage == null) {
            realImage = new RealImage(fileName); // Загрузка изображения при первом обращении
        }
        realImage.display();
    }
}

// Тестирование Proxy
public class ProxyPatternDemo {
    public static void main(String[] args) {
        Image image = new ProxyImage("test_image.jpg");

        // Изображение будет загружено только при первом вызове display()
        System.out.println("First call to display:");
        image.display();

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