**Behavioral Design Patterns**

**Strategy Pattern**

interface PaymentStrategy {

void pay(int amount);

}

class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String name;

private String cvv;

public CreditCardPayment(String cardNumber, String name, String cvv) {

this.cardNumber = cardNumber;

this.name = name;

this.cvv = cvv;

}

@Override

public void pay(int amount) {

System.out.println("Paid " + amount + " using Credit Card.");

}

}

class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

@Override

public void pay(int amount) {

System.out.println("Paid " + amount + " using PayPal.");

}

}

class ShoppingCart {

private List<Item> items;

private PaymentStrategy paymentStrategy;

public ShoppingCart() {

this.items = new ArrayList<>();

}

public void addItem(Item item) {

items.add(item);

}

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void checkout() {

int total = items.stream().mapToInt(Item::getPrice).sum();

paymentStrategy.pay(total);

}

}

class Item {

private String name;

private int price;

public Item(String name, int price) {

this.name = name;

this.price = price;

}

public int getPrice() {

return price;

}

}public class Main {

public static void main(String[] args) {

ShoppingCart cart = new ShoppingCart();

cart.addItem(new Item("Laptop", 1000));

cart.addItem(new Item("Phone", 500));

cart.setPaymentStrategy(new CreditCardPayment("1234567890123456", "John Doe", "123"));

cart.checkout();

cart.setPaymentStrategy(new PayPalPayment("john.doe@example.com"));

cart.checkout();

}

}

Observer Pattern

interface Observer {

void update(float temperature, float humidity, float pressure);

}

interface Subject {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

class WeatherData implements Subject {

private List<Observer> observers;

private float temperature;

private float humidity;

private float pressure;

public WeatherData() {

observers = new ArrayList<>();

}

@Override

public void registerObserver(Observer o) {

observers.add(o);

}

@Override

public void removeObserver(Observer o) {

observers.remove(o);

}

@Override

public void notifyObservers() {

for (Observer observer : observers) {

observer.update(temperature, humidity, pressure);

}

}

public void setMeasurements(float temperature, float humidity, float pressure) {

this.temperature = temperature;

this.humidity = humidity;

this.pressure = pressure;

notifyObservers();

}

}

class CurrentConditionsDisplay implements Observer {

private float temperature;

private float humidity;

@Override

public void update(float temperature, float humidity, float pressure) {

this.temperature = temperature;

this.humidity = humidity;

display();

}

public void display() {

System.out.println("Current conditions: " + temperature + "F degrees and " + humidity + "% humidity");

}

}public class Main {

public static void main(String[] args) {

WeatherData weatherData = new WeatherData();

CurrentConditionsDisplay currentDisplay = new CurrentConditionsDisplay();

weatherData.registerObserver(currentDisplay);

weatherData.setMeasurements(80, 65, 30.4f);

weatherData.setMeasurements(82, 70, 29.2f);

}

}

**Creational Design Patterns**

**Singleton Pattern**

// Logger.java

public class Logger {

private static Logger loggerInstance;

private Logger() {

// private constructor

}

public static Logger getInstance() {

if (loggerInstance == null) {

loggerInstance = new Logger();

}

return loggerInstance;

}

public void log(String message) {

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

}

}

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

logger1.log("This is the first log message.");

logger2.log("This is the second log message.");

System.out.println(logger1 == logger2); // should print true

}

}

Use Case 4: Factory Pattern

interface Shape {

void draw();

}

class Circle implements Shape {

@Override

public void draw() {

System.out.println("Drawing a Circle");

}

}

class Rectangle implements Shape {

@Override

public void draw() {

System.out.println("Drawing a Rectangle");

}

}

class ShapeFactory {

public Shape getShape(String shapeType) {

if (shapeType == null) {

return null;

}

if (shapeType.equalsIgnoreCase("CIRCLE")) {

return new Circle();

} else if (shapeType.equalsIgnoreCase("RECTANGLE")) {

return new Rectangle();

}

return null;

}

}

public class Main {

public static void main(String[] args) {

ShapeFactory shapeFactory = new ShapeFactory();

Shape shape1 = shapeFactory.getShape("CIRCLE");

shape1.draw();

Shape shape2 = shapeFactory.getShape("RECTANGLE");

shape2.draw();

}

}

**Structural Design Patterns**

**1. Adapter Pattern**

// Target interface

interface MediaPlayer {

void play(String audioType, String fileName);

}

// Adaptee interface

interface AdvancedMediaPlayer {

void playMp4(String fileName);

void playMp3(String fileName);

}

// Concrete Adaptee

class Mp4Player implements AdvancedMediaPlayer {

public void playMp4(String fileName) {

System.out.println("Playing mp4 file. Name: " + fileName);

}

public void playMp3(String fileName) {

// do nothing

}

}

class Mp3Player implements AdvancedMediaPlayer {

public void playMp3(String fileName) {

System.out.println("Playing mp3 file. Name: " + fileName);

}

public void playMp4(String fileName) {

// do nothing

}

}

// Adapter class

class MediaAdapter implements MediaPlayer {

AdvancedMediaPlayer advancedMusicPlayer;

public MediaAdapter(String audioType) {

if (audioType.equalsIgnoreCase("mp3")) {

advancedMusicPlayer = new Mp3Player();

} else if (audioType.equalsIgnoreCase("mp4")) {

advancedMusicPlayer = new Mp4Player();

}

}

public void play(String audioType, String fileName) {

if (audioType.equalsIgnoreCase("mp3")) {

advancedMusicPlayer.playMp3(fileName);

} else if (audioType.equalsIgnoreCase("mp4")) {

advancedMusicPlayer.playMp4(fileName);

}

}

}

// Concrete Target

class AudioPlayer implements MediaPlayer {

MediaAdapter mediaAdapter;

public void play(String audioType, String fileName) {

if (audioType.equalsIgnoreCase("mp3")) {

mediaAdapter = new MediaAdapter("mp3");

mediaAdapter.play(audioType, fileName);

} else if (audioType.equalsIgnoreCase("mp4")) {

mediaAdapter = new MediaAdapter("mp4");

mediaAdapter.play(audioType, fileName);

} else {

System.out.println("Invalid media. " + audioType + " format not supported");

}

}

}

// Client

public class AdapterPatternDemo {

public static void main(String[] args) {

AudioPlayer audioPlayer = new AudioPlayer();

audioPlayer.play("mp3", "beyond\_the\_horizon.mp3");

audioPlayer.play("mp4", "alone.mp4");

}

}

2. Decorator Pattern

// Component interface

interface Coffee {

String getDescription();

double getCost();

}

// Concrete Component

class SimpleCoffee implements Coffee {

public String getDescription() {

return "Simple coffee";

}

public double getCost() {

return 5.0;

}

}

// Decorator

abstract class CoffeeDecorator implements Coffee {

protected Coffee decoratedCoffee;

public CoffeeDecorator(Coffee coffee) {

this.decoratedCoffee = coffee;

}

public String getDescription() {

return decoratedCoffee.getDescription();

}

public double getCost() {

return decoratedCoffee.getCost();

}

}

// Concrete Decorators

class MilkDecorator extends CoffeeDecorator {

public MilkDecorator(Coffee coffee) {

super(coffee);

}

public String getDescription() {

return decoratedCoffee.getDescription() + ", Milk";

}

public double getCost() {

return decoratedCoffee.getCost() + 1.0;

}

}

class SugarDecorator extends CoffeeDecorator {

public SugarDecorator(Coffee coffee) {

super(coffee);

}

public String getDescription() {

return decoratedCoffee.getDescription() + ", Sugar";

}

public double getCost() {

return decoratedCoffee.getCost() + 0.5;

}

}

// Client

public class DecoratorPatternDemo {

public static void main(String[] args) {

Coffee coffee = new SimpleCoffee();

System.out.println(coffee.getDescription() + " $" + coffee.getCost());

coffee = new MilkDecorator(coffee);

System.out.println(coffee.getDescription() + " $" + coffee.getCost());

coffee = new SugarDecorator(coffee);

System.out.println(coffee.getDescription() + " $" + coffee.getCost());

}

}