EI CODING ASSIGMENT

Suhas S Bhat

Roll No: BL.EN.U4CSE21200

[suhasb259@gmail.com](mailto:suhasb259@gmail.com)

1) **Behavioral Design Patterns**

**Observer Pattern**

**Use Case:** Weather Monitoring System

**Description:** The Observer Pattern is used to define a one-to-many dependency between objects, so that when one object (the subject) changes its state, all its dependents (the observers) are notified and updated automatically.

In the context of a weather monitoring system, we have a WeatherData class that serves as the subject. This class maintains a list of observers (such as different display devices) that need to be updated whenever the weather data changes (temperature, humidity, etc.). Each observer implements an update method to receive notifications.

import java.util.ArrayList;

import java.util.List;

// Observer Interface

interface Observer {

void update(float temperature, float humidity);

}

// Subject Interface

interface Subject {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

// WeatherData class implementing Subject

class WeatherData implements Subject {

private List<Observer> observers;

private float temperature;

private float humidity;

public WeatherData() {

observers = new ArrayList<>();

}

public void registerObserver(Observer o) {

observers.add(o);

}

public void removeObserver(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer observer : observers) {

observer.update(temperature, humidity);

}

}

public void setMeasurements(float temperature, float humidity) {

this.temperature = temperature;

this.humidity = humidity;

notifyObservers();

}

}

// Concrete Observer

class TemperatureDisplay implements Observer {

@Override

public void update(float temperature, float humidity) {

System.out.println("Temperature Display: " + temperature + "°C");

}

}

// Main Class

public class WeatherStation {

public static void main(String[] args) {

WeatherData weatherData = new WeatherData();

TemperatureDisplay tempDisplay = new TemperatureDisplay();

weatherData.registerObserver(tempDisplay);

weatherData.setMeasurements(25.0f, 65.0f);

}

}

2)**Strategy Pattern**

**Use Case:** Payment Processing System

**Description:** The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. This pattern lets the algorithm vary independently from the clients that use it.

In a payment processing system, different payment methods (like Credit Card and PayPal) can be represented as strategies. The ShoppingCart class can use any payment strategy at runtime, allowing flexibility and easy addition of new payment methods without changing the core logic of the shopping cart.

// PaymentStrategy Interface

interface PaymentStrategy {

void pay(int amount);

}

// Concrete Strategies

class CreditCardPayment implements PaymentStrategy {

@Override

public void pay(int amount) {

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

}

}

class PayPalPayment implements PaymentStrategy {

@Override

public void pay(int amount) {

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

}

}

// Context Class

class ShoppingCart {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void checkout(int amount) {

paymentStrategy.pay(amount);

}

}

// Main Class

public class PaymentSystem {

public static void main(String[] args) {

ShoppingCart cart = new ShoppingCart();

// Use Credit Card

cart.setPaymentStrategy(new CreditCardPayment());

cart.checkout(100);

// Use PayPal

cart.setPaymentStrategy(new PayPalPayment());

cart.checkout(150);

}

}

3) **Creational Design Patterns**

**Singleton Pattern**

**Use Case:** Configuration Manager

**Description:** The Singleton Pattern restricts a class to a single instance and provides a global point of access to it. This is useful when exactly one object is needed to coordinate actions across the system.

In the configuration manager example, the ConfigurationManager class ensures that there is only one instance of the configuration manager throughout the application. This instance can be accessed globally to manage application settings, ensuring that all parts of the application are using the same configuration data.

class ConfigurationManager {

private static ConfigurationManager instance;

private ConfigurationManager() { }

public static ConfigurationManager getInstance() {

if (instance == null) {

instance = new ConfigurationManager();

}

return instance;

}

public void loadConfiguration() {

System.out.println("Configuration loaded.");

}

}

// Main Class

public class SingletonExample {

public static void main(String[] args) {

ConfigurationManager config1 = ConfigurationManager.getInstance();

config1.loadConfiguration();

ConfigurationManager config2 = ConfigurationManager.getInstance();

System.out.println(config1 == config2); // true

}

}

4)**Factory Method Pattern**

**Use Case:** Shape Factory

**Description:** The Factory Method Pattern defines an interface for creating an object but lets subclasses alter the type of objects that will be created. This pattern allows the class to defer instantiation to subclasses.

In the shape factory example, the ShapeFactory interface defines a method for creating shapes. The ConcreteShapeFactory class implements this method to create different shapes (Circle, Rectangle) based on the input string, encapsulating the object creation logic.

// Product Interface

interface Shape {

void draw();

}

// Concrete Products

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.");

}

}

// Factory

abstract class ShapeFactory {

abstract Shape createShape(String type);

}

class ConcreteShapeFactory extends ShapeFactory {

@Override

Shape createShape(String type) {

switch (type.toLowerCase()) {

case "circle":

return new Circle();

case "rectangle":

return new Rectangle();

default:

return null;

}

}

}

// Main Class

public class FactoryMethodExample {

public static void main(String[] args) {

ShapeFactory factory = new ConcreteShapeFactory();

Shape shape1 = factory.createShape("circle");

shape1.draw();

Shape shape2 = factory.createShape("rectangle");

shape2.draw();

}

}

5) **Structural Design Patterns**

**Adapter Pattern**

**Use Case:** Logging Service Adapter

**Description:** The Adapter Pattern allows objects with incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces by converting the interface of a class into another interface that the client expects.

In the logging service example, the Logger interface is designed to work with the application, but we have a third-party logging library with a different interface (ThirdPartyLogger). The LoggerAdapter class implements the Logger interface and adapts calls to the ThirdPartyLogger, allowing the application to use it seamlessly.

// Target Interface

interface Logger {

void log(String message);

}

// Adaptee Class

class ThirdPartyLogger {

void logMessage(String message) {

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

}

}

// Adapter Class

class LoggerAdapter implements Logger {

private ThirdPartyLogger thirdPartyLogger;

public LoggerAdapter(ThirdPartyLogger logger) {

this.thirdPartyLogger = logger;

}

@Override

public void log(String message) {

thirdPartyLogger.logMessage(message);

}

}

// Main Class

public class AdapterExample {

public static void main(String[] args) {

ThirdPartyLogger thirdPartyLogger = new ThirdPartyLogger();

Logger logger = new LoggerAdapter(thirdPartyLogger);

logger.log("This is a log message.");

}

}

6)**Composite Pattern**

**Use Case:** File System Structure

**Description:** The Composite Pattern allows you to compose objects into tree structures to represent part-whole hierarchies. It lets clients treat individual objects and compositions of objects uniformly.

In the file system structure example, we have File and Directory classes that implement the same interface (FileSystemComponent). A Directory can contain both files and other directories, allowing for a recursive structure that can be treated uniformly. This enables us to display the entire file system hierarchy with a single call.