**Exercise 1: Problem Statement on Design patterns**

Come up creatively with six different use cases to demonstrate your understanding of the following software design patterns by coding the same.

1. Two use cases to demonstrate two behavioural design pattern.

2. Two use cases to demonstrate two creational design pattern.

3. Two use cases to demonstrate two structural design pattern.

**1. Behavioural design pattern:**

1.1 Observer design pattern:

import java.util.ArrayList;

import java.util.List;

// Observer interface

interface Observer {

void update(float shareMarketPrice);

}

// Subject interface

interface Subject {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

class GoldRate implements Observer {

private float goldRate;

@Override

public void update(float shareMarketPrice) {

// Simple formula to calculate gold rate based on share market price

this.goldRate = shareMarketPrice \* 2; // Example conversion formula

display();

}

public void display() {

System.out.println("Updated Gold Rate: " + goldRate);

}

}

class ShareMarket implements Subject {

private List<Observer> observers;

private float shareMarketPrice;

public ShareMarket() {

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(shareMarketPrice);

}

}

public void setShareMarketPrice(float shareMarketPrice) {

this.shareMarketPrice = shareMarketPrice;

shareMarketPriceChanged();

}

private void shareMarketPriceChanged() {

notifyObservers();

}

}

public class Main {

public static void main(String[] args) {

ShareMarket shareMarket = new ShareMarket();

GoldRate goldRate = new GoldRate();

int share;

shareMarket.registerObserver(goldRate);

Scanner s=new Scanner(System.in);

System.out.println("Enter the share market price:");

share=s.nextInt();

// Simulate share market price changes

shareMarket.setShareMarketPrice(share);

}

}

1.2 Strategy design pattern:

// Strategy Interface

interface PaymentStrategy {

void pay(int amount);

}

// Concrete Strategy 1

class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

public CreditCardPayment(String cardNumber) {

this.cardNumber = cardNumber;

}

@Override

public void pay(int amount) {

System.out.println(amount + " paid using Credit Card. Card Number: " + cardNumber);

}

}

// Concrete Strategy 2

class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

@Override

public void pay(int amount) {

System.out.println(amount + " paid using PayPal. Email: " + email);

}

}

// Concrete Strategy 3

class BitcoinPayment implements PaymentStrategy {

private String walletAddress;

public BitcoinPayment(String walletAddress) {

this.walletAddress = walletAddress;

}

@Override

public void pay(int amount) {

System.out.println(amount + " paid using Bitcoin. Wallet Address: " + walletAddress);

}

}

// Context

class ShoppingCart {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void checkout(int amount) {

if (paymentStrategy == null) {

System.out.println("No payment strategy selected.");

} else {

paymentStrategy.pay(amount);

}

}

}

// Main

public class StrategyPatternExample {

public static void main(String[] args) {

ShoppingCart cart = new ShoppingCart();

cart.setPaymentStrategy(new CreditCardPayment("1234-5678-9876-5432"));

cart.checkout(100);

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

cart.checkout(200);

cart.setPaymentStrategy(new BitcoinPayment("1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa"));

cart.checkout(300);

}

}

**2. Creational design pattern:**

2.1 Singleton design pattern:

class Logger {

private static Logger instance;

// Private constructor to prevent instantiation

private Logger() {}

// Method to get the single instance of the Logger class

public static Logger getInstance() {

if (instance == null) {

synchronized (Logger.class) {

if (instance == null) {

instance = new Logger();

}

}

}

return instance;

}

// Method to log messages

public void log(String message) {

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

}

}

public class SingletonLoggerDemo {

public static void main(String[] args) {

// Get the single instance of the Logger

Logger logger = Logger.getInstance();

// Log messages from different parts of the application

logger.log("Application started.");

performTask1();

performTask2();

logger.log("Application ended.");

}

private static void performTask1() {

Logger logger = Logger.getInstance();

logger.log("Performing task 1.");

}

private static void performTask2() {

Logger logger = Logger.getInstance();

logger.log("Performing task 2.");

}

}

2.2 Factory design pattern:

interface Animal {

void displayType();

}

// Concrete class for Herbivore

class Herbivore implements Animal {

@Override

public void displayType() {

System.out.println("I am a Herbivore. I eat plants.");

}

}

// Concrete class for Carnivore

class Carnivore implements Animal {

@Override

public void displayType() {

System.out.println("I am a Carnivore. I eat meat.");

}

}

// Concrete class for Omnivore

class Omnivore implements Animal {

@Override

public void displayType() {

System.out.println("I am an Omnivore. I eat both plants and meat.");

}

}

// Factory class to create Animal objects

class AnimalFactory {

public static Animal createAnimal(String foodType) {

if (foodType.equalsIgnoreCase("plant")) {

return new Herbivore();

} else if (foodType.equalsIgnoreCase("meat")) {

return new Carnivore();

} else if (foodType.equalsIgnoreCase("both")) {

return new Omnivore();

} else {

throw new IllegalArgumentException("Unknown food type: " + foodType);

}

}

}

import java.util.Scanner;

public class AnimalFactoryDemo {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.println("Enter the type of food (plant, meat, both): ");

String foodType = scanner.nextLine();

try {

Animal animal = AnimalFactory.createAnimal(foodType);

animal.displayType();

} catch (IllegalArgumentException e) {

System.out.println(e.getMessage());

}

scanner.close();

}

}

**3. Structural design pattern:**

3.1 Adapter design pattern:

// Target Interface: The interface expected by the client (your application)

interface Shape {

void draw(int x, int y, int width, int height);

}

// Adaptee: The third-party library's interface

class LegacyRectangle {

public void drawRectangle(int x1, int y1, int x2, int y2) {

System.out.println("Drawing rectangle from (" + x1 + ", " + y1 + ") to (" + x2 + ", " + y2 + ")");

}

}

// Adapter: Implements the Target interface and translates the calls to the Adaptee

class RectangleAdapter implements Shape {

private LegacyRectangle legacyRectangle;

public RectangleAdapter(LegacyRectangle legacyRectangle) {

this.legacyRectangle = legacyRectangle;

}

@Override

public void draw(int x, int y, int width, int height) {

int x2 = x + width;

int y2 = y + height;

legacyRectangle.drawRectangle(x, y, x2, y2);

}

}

// Client: The part of your application that uses the Target interface

public class AdapterPatternExample {

public static void main(String[] args) {

Shape rectangle = new RectangleAdapter(new LegacyRectangle());

rectangle.draw(10, 20, 30, 40);

}

}

3.2 Decorator design pattern:

// Component Interface: The base interface for the text

interface Text {

String getText();

}

// Concrete Component: The base implementation of the text

class PlainText implements Text {

private String text;

public PlainText(String text) {

this.text = text;

}

@Override

public String getText() {

return text;

}

}

// Decorator: An abstract class implementing the component interface and containing a reference to a component object

abstract class TextDecorator implements Text {

protected Text decoratedText;

public TextDecorator(Text decoratedText) {

this.decoratedText = decoratedText;

}

@Override

public String getText() {

return decoratedText.getText();

}

}

// Concrete Decorators: Concrete implementations of the decorator that add specific formatting

class BoldDecorator extends TextDecorator {

public BoldDecorator(Text decoratedText) {

super(decoratedText);

}

@Override

public String getText() {

return "<b>" + decoratedText.getText() + "</b>";

}

}

class ItalicDecorator extends TextDecorator {

public ItalicDecorator(Text decoratedText) {

super(decoratedText);

}

@Override

public String getText() {

return "<i>" + decoratedText.getText() + "</i>";

}

}

class UnderlineDecorator extends TextDecorator {

public UnderlineDecorator(Text decoratedText) {

super(decoratedText);

}

@Override

public String getText() {

return "<u>" + decoratedText.getText() + "</u>";

}

}

// Client: The part of your application that uses the component interface

public class DecoratorPatternExample {

public static void main(String[] args) {

// Create base text

Text text = new PlainText("Hello, World!");

// Apply decorators

Text boldText = new BoldDecorator(text);

Text italicText = new ItalicDecorator(boldText);

Text underlinedText = new UnderlineDecorator(italicText);

// Print the formatted text

System.out.println(underlinedText.getText());

}

}