**Exercise 1: Implementing the Singleton Pattern**

**Scenario:** You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

Program :

class LogManager {

private static LogManager uniqueInstance;

private LogManager() {

// Initialization

}

public static LogManager getUniqueInstance() {

if (uniqueInstance == null) {

uniqueInstance = new LogManager();

}

return uniqueInstance;

}

public void logMessage(String msg) {

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

}

}

class SingletonExample {

public static void main(String[] args) {

LogManager log1 = LogManager.getUniqueInstance();

log1.logMessage("This is the first log message.");

LogManager log2 = LogManager.getUniqueInstance();

log2.logMessage("This is the second log message.");

if (log1 == log2) {

System.out.println("Both log1 and log2 are the same instance.");

} else {

System.out.println("Logger instances are different.");

}

}

}

**Exercise 2: Implementing the Factory Method Pattern**

**Scenario:** You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

Program:

interface File {

void open();

void close();

}

class TextFile implements File {

public void open() {

System.out.println("Opening Word document...");

}

public void close() {

System.out.println("Closing Word document...");

}

}

class PortableDocument implements File {

public void open() {

System.out.println("Opening PDF document...");

}

public void close() {

System.out.println("Closing PDF document...");

}

}

class Spreadsheet implements File {

public void open() {

System.out.println("Opening Excel document...");

}

public void close() {

System.out.println("Closing Excel document...");

}

}

abstract class FileFactory {

public abstract File createFile();

}

class TextFileFactory extends FileFactory {

public File createFile() {

return new TextFile();

}

}

class PortableDocumentFactory extends FileFactory {

public File createFile() {

return new PortableDocument();

}

}

class SpreadsheetFactory extends FileFactory {

public File createFile() {

return new Spreadsheet();

}

}

public class FactoryMethodExample {

public static void main(String[] args) {

FileFactory textFactory = new TextFileFactory();

File textFile = textFactory.createFile();

textFile.open();

textFile.close();

FileFactory pdfFactory = new PortableDocumentFactory();

File pdfFile = pdfFactory.createFile();

pdfFile.open();

pdfFile.close();

FileFactory spreadsheetFactory = new SpreadsheetFactory();

File excelFile = spreadsheetFactory.createFile();

excelFile.open();

excelFile.close();

}

}

**Exercise 3: Implementing the Builder Pattern**

**Scenario:** You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

Program :

public class BuilderPatternDemo {

static class PC {

private String processor;

private String memory;

private String disk;

private PC(PCBuilder builder) {

this.processor = builder.processor;

this.memory = builder.memory;

this.disk = builder.disk;

}

public static class PCBuilder {

private String processor;

private String memory;

private String disk;

public PCBuilder setProcessor(String processor) {

this.processor = processor;

return this;

}

public PCBuilder setMemory(String memory) {

this.memory = memory;

return this;

}

public PCBuilder setDisk(String disk) {

this.disk = disk;

return this;

}

public PC build() {

return new PC(this);

}

}

}

public static void main(String[] args) {

PC workstation = new PC.PCBuilder()

.setProcessor("AMD Ryzen 9")

.setMemory("64GB")

.setDisk("2TB SSD")

.build();

System.out.println("Processor: " + workstation.processor);

System.out.println("Memory: " + workstation.memory);

System.out.println("Disk: " + workstation.disk);

}

}

**Exercise 4: Implementing the Adapter Pattern**

**Scenario:** You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

Program :

interface PaymentHandler {

void handlePayment(double amount);

}

class PayU {

public void executePayment(double amount) {

System.out.println("Processing payment of Rs." + amount + " through PayU.");

}

}

class RazorPay {

public void payAmount(double amount) {

System.out.println("Processing payment of Rs." + amount + " through RazorPay.");

}

}

class PhonePe {

public void initiateTransaction(double amount) {

System.out.println("Processing payment of Rs." + amount + " through PhonePe.");

}

}

class PayUAdapter implements PaymentHandler {

private PayU payU;

public PayUAdapter(PayU payU) {

this.payU = payU;

}

public void handlePayment(double amount) {

payU.executePayment(amount);

}

}

class RazorPayAdapter implements PaymentHandler {

private RazorPay razorPay;

public RazorPayAdapter(RazorPay razorPay) {

this.razorPay = razorPay;

}

public void handlePayment(double amount) {

razorPay.payAmount(amount);

}

}

class PhonePeAdapter implements PaymentHandler {

private PhonePe phonePe;

public PhonePeAdapter(PhonePe phonePe) {

this.phonePe = phonePe;

}

public void handlePayment(double amount) {

phonePe.initiateTransaction(amount);

}

}

public class AdapterPatternDemo {

public static void main(String[] args) {

PayU payU = new PayU();

RazorPay razorPay = new RazorPay();

PhonePe phonePe = new PhonePe();

PaymentHandler payUAdapter = new PayUAdapter(payU);

PaymentHandler razorPayAdapter = new RazorPayAdapter(razorPay);

PaymentHandler phonePeAdapter = new PhonePeAdapter(phonePe);

payUAdapter.handlePayment(150.00);

razorPayAdapter.handlePayment(250.00);

phonePeAdapter.handlePayment(350.00);

}

}

**Exercise 5: Implementing the Decorator Pattern**

**Scenario:** You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically

Program :

interface Messenger {

void notify(String message);

}

class MailMessenger implements Messenger {

public void notify(String message) {

System.out.println("Sending email notification: " + message);

}

}

abstract class MessengerDecorator implements Messenger {

protected Messenger messenger;

public MessengerDecorator(Messenger messenger) {

this.messenger = messenger;

}

public void notify(String message) {

messenger.notify(message);

}

}

class TextMessengerDecorator extends MessengerDecorator {

public TextMessengerDecorator(Messenger messenger) {

super(messenger);

}

public void notify(String message) {

messenger.notify(message);

sendText(message);

}

private void sendText(String message) {

System.out.println("Sending SMS notification: " + message);

}

}

class TeamsMessengerDecorator extends MessengerDecorator {

public TeamsMessengerDecorator(Messenger messenger) {

super(messenger);

}

public void notify(String message) {

messenger.notify(message);

sendTeams(message);

}

private void sendTeams(String message) {

System.out.println("Sending Teams notification: " + message);

}

}

public class DecoratorPatternDemo {

public static void main(String[] args) {

Messenger mailMessenger = new MailMessenger();

Messenger textMessenger = new TextMessengerDecorator(mailMessenger);

Messenger teamsMessenger = new TeamsMessengerDecorator(textMessenger);

teamsMessenger.notify("Hello, this is a test notification!");

}

}

**Exercise 6: Implementing the Proxy Pattern**

**Scenario:** You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching

Program :

**interface Picture {**

**void show();**

**}**

**// Implement Real Subject Class**

**class HighResolutionImage implements Picture {**

**private String fileName;**

**public HighResolutionImage(String fileName) {**

**this.fileName = fileName;**

**loadFromDisk();**

**}**

**private void loadFromDisk() {**

**System.out.println("Loading image from disk: " + fileName);**

**}**

**public void show() {**

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

**}**

**}**

**// Implement Proxy Class**

**class ImageProxy implements Picture {**

**private String fileName;**

**private HighResolutionImage highResolutionImage;**

**public ImageProxy(String fileName) {**

**this.fileName = fileName;**

**}**

**public void show() {**

**if (highResolutionImage == null) {**

**highResolutionImage = new HighResolutionImage(fileName);**

**}**

**highResolutionImage.show();**

**}**

**}**

**// Test the Proxy Implementation**

**public class ProxyPatternDemo {**

**public static void main(String[] args) {**

**Picture pic1 = new ImageProxy("photo1.jpg");**

**Picture pic2 = new ImageProxy("photo2.jpg");**

**// Image will be loaded from disk**

**pic1.show();**

**System.out.println("");**

**// Image will not be loaded from disk as it is already loaded**

**pic1.show();**

**System.out.println("");**

**// Image will be loaded from disk**

**pic2.show();**

**System.out.println("");**

**// Image will not be loaded from disk as it is already loaded**

**pic2.show();**

**}**

**}**

**Exercise 7: Implementing the Observer Pattern**

**Scenario:** You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

Program :

import java.util.ArrayList;

import java.util.List;

interface Subject {

void addObserver(Subscriber s);

void removeObserver(Subscriber s);

void notifySubscribers();

}

class Market implements Subject {

private List<Subscriber> subscribers;

private double price;

public Market() {

this.subscribers = new ArrayList<>();

}

public void addObserver(Subscriber s) {

subscribers.add(s);

}

public void removeObserver(Subscriber s) {

subscribers.remove(s);

}

public void notifySubscribers() {

for (Subscriber s : subscribers) {

s.update(price);

}

}

public void setPrice(double price) {

this.price = price;

notifySubscribers();

}

}

interface Subscriber {

void update(double price);

}

class MobileClient implements Subscriber {

private String clientName;

public MobileClient(String clientName) {

this.clientName = clientName;

}

public void update(double price) {

System.out.println(clientName + " received price update: " + price);

}

}

class WebClient implements Subscriber {

private String clientName;

WebClient(String clientName) {

this.clientName = clientName;

}

public void update(double price) {

System.out.println(clientName + " received price update: " + price);

}

}

public class ObserverPatternDemo {

public static void main(String[] args) {

Market market = new Market();

Subscriber mobileClient = new MobileClient("MobileClient");

Subscriber webClient = new WebClient("WebClient");

market.addObserver(mobileClient);

market.addObserver(webClient);

market.setPrice(100.00);

market.setPrice(101.50);

market.removeObserver(webClient);

market.setPrice(102.75);

}

}

**Exercise 8: Implementing the Strategy Pattern**

**Scenario:** You are developing a payment system where different payment methods (e.g., Credit Card, PayPal) can be selected at runtime. Use the Strategy Pattern to achieve this.

Program:

// Step 2: Define Strategy Interface

interface PaymentMethod {

void processPayment(double amount);

}

// Step 3: Implement Concrete Strategies

class CardPayment implements PaymentMethod {

private String cardHolder;

private String cardNumber;

private String securityCode;

private String expirationDate;

public CardPayment(String cardHolder, String cardNumber, String securityCode, String expirationDate) {

this.cardHolder = cardHolder;

this.cardNumber = cardNumber;

this.securityCode = securityCode;

this.expirationDate = expirationDate;

}

public void processPayment(double amount) {

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

}

}

class PayPalMethod implements PaymentMethod {

private String userEmail;

private String userPassword;

public PayPalMethod(String userEmail, String userPassword) {

this.userEmail = userEmail;

this.userPassword = userPassword;

}

public void processPayment(double amount) {

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

}

}

// Step 4: Implement Context Class

class PaymentService {

private PaymentMethod paymentMethod;

public void setPaymentMethod(PaymentMethod paymentMethod) {

this.paymentMethod = paymentMethod;

}

public void executePayment(double amount) {

paymentMethod.processPayment(amount);

}

}

// Step 5: Test the Strategy Implementation

public class StrategyPatternDemo {

public static void main(String[] args) {

PaymentService paymentService = new PaymentService();

// Pay using Credit Card

paymentService.setPaymentMethod(new CardPayment("Jane Doe", "6543210987654321", "456", "11/25"));

paymentService.executePayment(150.0);

// Pay using PayPal

paymentService.setPaymentMethod(new PayPalMethod("jane.doe@example.com", "securePassword"));

paymentService.executePayment(250.0);

}

}

**Exercise 9: Implementing the Command Pattern**

**Scenario:** You are developing a home automation system where commands can be issued to turn devices on or off. Use the Command Pattern to achieve this.

Program :

interface Action {

void perform();

}

// Implement Concrete Commands

class TurnLightOn implements Action {

private Bulb bulb;

public TurnLightOn(Bulb bulb) {

this.bulb = bulb;

}

@Override

public void perform() {

bulb.switchOn();

}

}

class TurnLightOff implements Action {

private Bulb bulb;

public TurnLightOff(Bulb bulb) {

this.bulb = bulb;

}

@Override

public void perform() {

bulb.switchOff();

}

}

// Implement Receiver Class

class Bulb {

public void switchOn() {

System.out.println("The light is on");

}

public void switchOff() {

System.out.println("The light is off");

}

}

// Implement Invoker Class

class Controller {

private Action action;

public void setAction(Action action) {

this.action = action;

}

public void pressButton() {

action.perform();

}

}

// Test the Command Implementation

public class CommandPatternDemo {

public static void main(String[] args) {

Bulb roomLight = new Bulb();

Action turnOn = new TurnLightOn(roomLight);

Action turnOff = new TurnLightOff(roomLight);

Controller controller = new Controller();

// Turn the light on

controller.setAction(turnOn);

controller.pressButton();

// Turn the light off

controller.setAction(turnOff);

controller.pressButton();

}

}

**Exercise 10: Implementing the MVC Pattern**

**Scenario:** You are developing a simple web application for managing student records using the MVC pattern.

Program :

class Pupil {

private String rollNumber;

private String fullName;

private String mark;

public Pupil(String rollNumber, String fullName, String mark) {

this.rollNumber = rollNumber;

this.fullName = fullName;

this.mark = mark;

}

public String getRollNumber() {

return rollNumber;

}

public void setRollNumber(String rollNumber) {

this.rollNumber = rollNumber;

}

public String getFullName() {

return fullName;

}

public void setFullName(String fullName) {

this.fullName = fullName;

}

public String getMark() {

return mark;

}

public void setMark(String mark) {

this.mark = mark;

}

}

// Define View Class

class PupilView {

public void showPupilDetails(String pupilName, String pupilRollNumber, String pupilMark) {

System.out.println("Pupil Details:");

System.out.println("Name: " + pupilName);

System.out.println("Roll Number: " + pupilRollNumber);

System.out.println("Mark: " + pupilMark);

}

}

// Define Controller Class

class PupilController {

private Pupil model;

private PupilView view;

public PupilController(Pupil model, PupilView view) {

this.model = model;

this.view = view;

}

public void setPupilName(String name) {

model.setFullName(name);

}

public String getPupilName() {

return model.getFullName();

}

public void setPupilRollNumber(String rollNumber) {

model.setRollNumber(rollNumber);

}

public String getPupilRollNumber() {

return model.getRollNumber();

}

public void setPupilMark(String mark) {

model.setMark(mark);

}

public String getPupilMark() {

return model.getMark();

}

public void refreshView() {

view.showPupilDetails(model.getFullName(), model.getRollNumber(), model.getMark());

}

}

// Test the MVC Implementation

public class MVCDemo {

public static void main(String[] args) {

// Create a Pupil model

Pupil model = new Pupil("101", "Alice Smith", "A+");

// Create a Pupil view

PupilView view = new PupilView();

// Create a Pupil controller

PupilController controller = new PupilController(model, view);

// Display initial pupil details

controller.refreshView();

// Update pupil details

controller.setPupilName("Bob Johnson");

controller.setPupilMark("B");

// Display updated pupil details

controller.refreshView();

}

}

**Exercise 11: Implementing Dependency Injection**

**Scenario:** You are developing a customer management application where the service class depends on a repository class. Use Dependency Injection to manage these dependencies.

Program :

interface UserRepository {

String getUserById(String userId);

}

// Implement Concrete Repository

class UserRepositoryImpl implements UserRepository {

public String getUserById(String userId) {

// Mock implementation, in a real scenario, it would interact with a database

if (userId.equals("1")) {

return "Alice Johnson";

} else {

return "User not found";

}

}

}

// Define Service Class

class UserService {

private UserRepository userRepository;

// Implement Dependency Injection

public UserService(UserRepository userRepository) {

this.userRepository = userRepository;

}

public String fetchUserDetails(String userId) {

return userRepository.getUserById(userId);

}

}

// Test the Dependency Injection Implementation

public class DependencyInjectionDemo {

public static void main(String[] args) {

// Create a UserRepository instance

UserRepository userRepository = new UserRepositoryImpl();

// Inject the repository into the service

UserService userService = new UserService(userRepository);

// Use the service to find user details

String userDetails = userService.fetchUserDetails("1");

System.out.println("User Details: " + userDetails);

} }