**Design Patterns and Principles**

**Exercise 1: Implementing the Singleton Pattern**

### **Step 1: Create a New Java Project**

**Project Name:** SingletonPatternExample  
 (Use any IDE like Eclipse, IntelliJ, or VS Code to create this.)

### **Step 2: Define a Singleton Class**

**Filename:** Logger.java

public class Logger {

// Step 2: Private static instance of the class

private static Logger instance;

// Step 2: Private constructor to prevent instantiation from outside

private Logger() {

System.out.println("Logger instance created.");

}

// Step 3: Public static method to get the single instance

public static Logger getInstance() {

if (instance == null) {

instance = new Logger(); // Lazy initialization

}

return instance;

}

// Method to simulate logging

public void log(String message) {

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

}

}

### **Step 3: Implement the Singleton Pattern**

To ensure that the Logger class follows the Singleton pattern, we need to:

* Make the constructor private.
* Hold a **private static instance** of the class.
* Provide a **public static method** (getInstance()) to access the single instance.

**Filename:** Logger.java

public class Logger {

    // Step 1: Private static instance of Logger

    private static Logger instance;

    // Step 2: Private constructor to prevent instantiation

    private Logger() {

        System.out.println("Logger Initialized");

    }

    // Step 3: Public static method to get the instance

    public static Logger getInstance() {

        if (instance == null) {

            instance = new Logger(); // Lazy initialization

        }

        return instance;

    }

    public void log(String message) {

        System.out.println("[LOG]: " + message);

    }

}

### Test Class

**Filename:** Main.java

public class Main {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        logger1.log("Starting application...");

        Logger logger2 = Logger.getInstance();

        logger2.log("Performing operation...");

        // Verify if both references are the same

        System.out.println("Are both loggers the same? " + (logger1 == logger2));

    }

}

### Output:

markdown

Logger Initialized

[LOG]: Starting application...

[LOG]: Performing operation...

Are both loggers the same? true

### **Step 4: Test the Singleton Implementation**

**Filename:** Main.java

public class Main {

public static void main(String[] args) {

// Attempt to get Logger instances

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

// Logging messages

logger1.log("First log message.");

logger2.log("Second log message.");

// Verifying singleton behavior

if (logger1 == logger2) {

System.out.println("Both logger1 and logger2 refer to the same instance.");

} else {

System.out.println("Different instances created! Singleton pattern failed.");

}

}

}

### 

### **Output (Expected):**

pgsql

Logger instance created.

Log: First log message.

Log: Second log message.

Both logger1 and logger2 refer to the same instance.

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

### **Step 1: Create a New Java Project**

**Project Name:** FactoryMethodPatternExample

### **Step 2: Define Document Interface**

**Filename:** Document.java

public interface Document {

void open();

}

**Step 3: Create Concrete Document Classes**

**Filename:** WordDocument.java

public class WordDocument implements Document {

@Override

public void open() {

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

}

}

**Filename:** PdfDocument.java

public class PdfDocument implements Document {

@Override

public void open() {

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

}

}

**Filename:** ExcelDocument.java

public class ExcelDocument implements Document {

@Override

public void open() {

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

}

}

### **Step 4: Implement the Factory Method**

**Filename:** DocumentFactory.java

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**Filename:** WordDocumentFactory.java

public class WordDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

**Filename:** PdfDocumentFactory.java

public class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

**Filename:** ExcelDocumentFactory.java

public class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

### **Step 5: Test the Factory Method Implementation**

**Filename:** Main.java

public class Main {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

### **Output (Expected):**

mathematica

Opening Word document...

Opening PDF document...

Opening Excel document...

**Exercise 3: Implementing the Builder Pattern**

### **Step 1: Create a New Java Project**

**Project Name:** BuilderPatternExample

### **Step 2: Define the Product Class**

**Filename:** Computer.java

public class Computer {

// Required and optional parts

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

private String keyboard;

private String mouse;

// Step 4: Private constructor

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

this.keyboard = builder.keyboard;

this.mouse = builder.mouse;

}

// Step 3: Static nested Builder class

public static class Builder {

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

private String keyboard;

private String mouse;

public Builder setCPU(String CPU) {

this.CPU = CPU;

return this;

}

public Builder setRAM(String RAM) {

this.RAM = RAM;

return this;

}

public Builder setStorage(String storage) {

this.storage = storage;

return this;

}

public Builder setGraphicsCard(String graphicsCard) {

this.graphicsCard = graphicsCard;

return this;

}

public Builder setKeyboard(String keyboard) {

this.keyboard = keyboard;

return this;

}

public Builder setMouse(String mouse) {

this.mouse = mouse;

return this;

}

// Final build() method to return a Computer object

public Computer build() {

return new Computer(this);

}

}

@Override

public String toString() {

return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", Storage=" + storage +

", GraphicsCard=" + graphicsCard + ", Keyboard=" + keyboard + ", Mouse=" + mouse + "]";

}

}

### **Step 5: Test the Builder Implementation**

**Filename:** Main.java

public class Main {

public static void main(String[] args) {

// Building a high-end computer

Computer gamingPC = new Computer.Builder()

.setCPU("Intel i9")

.setRAM("32GB")

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 4080")

.setKeyboard("Mechanical Keyboard")

.setMouse("Gaming Mouse")

.build();

// Building a basic computer

Computer basicPC = new Computer.Builder()

.setCPU("Intel i5")

.setRAM("8GB")

.setStorage("256GB SSD")

.build();

System.out.println("Gaming PC: " + gamingPC);

System.out.println("Basic PC: " + basicPC);

}

}

### **Output (Expected):**

arduino

Gaming PC: Computer [CPU=Intel i9, RAM=32GB, Storage=1TB SSD, GraphicsCard=NVIDIA RTX 4080, Keyboard=Mechanical Keyboard, Mouse=Gaming Mouse]

Basic PC: Computer [CPU=Intel i5, RAM=8GB, Storage=256GB SSD, GraphicsCard=null, Keyboard=null, Mouse=null]

**Exercise 4: Implementing the Adapter Pattern**

### **Step 1: Create a New Java Project**

**Project Name:** AdapterPatternExample

### **Step 2: Define Target Interface**

**Filename:** PaymentProcessor.java

public interface PaymentProcessor {

void processPayment(double amount);

}

### **Step 3: Implement Adaptee Classes**

These represent third-party payment gateways with their own interfaces.

**Filename:** PayPalGateway.java

public class PayPalGateway {

public void makePayment(double amount) {

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

}

}

**Filename:** StripeGateway.java

public class StripeGateway {

public void sendPayment(double amount) {

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

}

}

**Filename:** RazorpayGateway.java

public class RazorpayGateway {

public void executeTransaction(double amount) {

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

}

}

### **Step 4: Implement Adapter Classes**

These translate PaymentProcessor calls to the corresponding gateway methods.

**Filename:** PayPalAdapter.java

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPal;

public PayPalAdapter(PayPalGateway payPal) {

this.payPal = payPal;

}

@Override

public void processPayment(double amount) {

payPal.makePayment(amount);

}

}

**Filename:** StripeAdapter.java

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe;

}

@Override

public void processPayment(double amount) {

stripe.sendPayment(amount);

}

}

**Filename:** RazorpayAdapter.java

public class RazorpayAdapter implements PaymentProcessor {

private RazorpayGateway razorpay;

public RazorpayAdapter(RazorpayGateway razorpay) {

this.razorpay = razorpay;

}

@Override

public void processPayment(double amount) {

razorpay.executeTransaction(amount);

}

}

### **Step 5: Test the Adapter Implementation**

**Filename:** Main.java

public class Main {

public static void main(String[] args) {

// Using PayPal

PaymentProcessor paypal = new PayPalAdapter(new PayPalGateway());

paypal.processPayment(2500);

// Using Stripe

PaymentProcessor stripe = new StripeAdapter(new StripeGateway());

stripe.processPayment(3200);

// Using Razorpay

PaymentProcessor razorpay = new RazorpayAdapter(new RazorpayGateway());

razorpay.processPayment(1500);

}

}

### **Output (Expected):**

cpp

Paid ₹2500.0 using PayPal.

Paid ₹3200.0 using Stripe.

Paid ₹1500.0 using Razorpay.

**Exercise 5: Implementing the Decorator Pattern**

### **Step 1: Create a New Java Project**

**Project Name:** DecoratorPatternExample

### **Step 2: Define the Component Interface**

**Filename:** Notifier.java

public interface Notifier {

void send(String message);

}

### **Step 3: Implement the Concrete Component**

**Filename:** EmailNotifier.java

public class EmailNotifier implements Notifier {

@Override

public void send(String message) {

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

}

}

### **Step 4: Implement Decorator Classes**

#### **Abstract Decorator**

**Filename:** NotifierDecorator.java

public abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

@Override

public void send(String message) {

notifier.send(message);

}

}

#### **Concrete Decorator – SMS**

**Filename:** SMSNotifierDecorator.java

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

sendSMS(message);

}

private void sendSMS(String message) {

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

}

}

#### **Concrete Decorator – Slack**

**Filename:** SlackNotifierDecorator.java

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

sendSlack(message);

}

private void sendSlack(String message) {

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

}

}

### **Step 5: Test the Decorator Implementation**

**Filename:** Main.java

public class Main {

public static void main(String[] args) {

// Basic Email Notification

Notifier email = new EmailNotifier();

// Add SMS functionality

Notifier emailAndSMS = new SMSNotifierDecorator(email);

// Add Slack functionality on top of Email and SMS

Notifier fullNotifier = new SlackNotifierDecorator(emailAndSMS);

// Send message via all channels

fullNotifier.send("System maintenance at 11 PM tonight.");

}

}

### **Output (Expected):**

sql

Sending Email: System maintenance at 11 PM tonight.

Sending SMS: System maintenance at 11 PM tonight.

Sending Slack message: System maintenance at 11 PM tonight.

**Exercise 6: Implementing the Proxy Pattern**

### **Step 1: Create a New Java Project**

**Project Name:** ProxyPatternExample

### **Step 2: Define the Subject Interface**

**Filename:** Image.java

public interface Image {

void display();

}

### **Step 3: Implement the Real Subject Class**

**Filename:** RealImage.java

public class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromRemoteServer();

}

private void loadFromRemoteServer() {

System.out.println("Loading image from remote server: " + filename);

}

@Override

public void display() {

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

}

}

### **Step 4: Implement the Proxy Class**

**Filename:** ProxyImage.java

public class ProxyImage implements Image {

private RealImage realImage;

private String filename;

public ProxyImage(String filename) {

this.filename = filename;

}

@Override

public void display() {

if (realImage == null) {

realImage = new RealImage(filename); // Lazy loading

} else {

System.out.println("Using cached image: " + filename);

}

realImage.display();

}

}

### **Step 5: Test the Proxy Implementation**

**Filename:** Main.java

public class Main {

public static void main(String[] args) {

Image image1 = new ProxyImage("nature.jpg");

// First call: image will be loaded from remote

image1.display();

System.out.println();

// Second call: image will be loaded from cache

image1.display();

}

}

### **Output (Expected):**

arduino

Loading image from remote server: nature.jpg

Displaying image: nature.jpg

Using cached image: nature.jpg

Displaying image: nature.jpg

**Exercise 7: Implementing the Observer Pattern**

### **Step 1: Create a New Java Project**

**Project Name:** ObserverPatternExample

### **Step 2: Define Subject Interface**

**Filename:** Stock.java

public interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

### **Step 3: Implement Concrete Subject**

**Filename:** StockMarket.java

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

private List<Observer> observers = new ArrayList<>();

private double stockPrice;

@Override

public void registerObserver(Observer o) {

observers.add(o);

}

@Override

public void removeObserver(Observer o) {

observers.remove(o);

}

@Override

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockPrice);

}

}

public void setStockPrice(double price) {

this.stockPrice = price;

System.out.println("\nStock price updated to: ₹" + price);

notifyObservers();

}

}

### **Step 4: Define Observer Interface**

**Filename:** Observer.java

public interface Observer {

void update(double price);

}

### **Step 5: Implement Concrete Observers**

**Filename:** MobileApp.java

public class MobileApp implements Observer {

private String user;

public MobileApp(String user) {

this.user = user;

}

@Override

public void update(double price) {

System.out.println("MobileApp - " + user + ": Stock price changed to ₹" + price);

}

}

**Filename:** WebApp.java

public class WebApp implements Observer {

private String user;

public WebApp(String user) {

this.user = user;

}

@Override

public void update(double price) {

System.out.println("WebApp - " + user + ": Stock price changed to ₹" + price);

}

}

### **Step 6: Test the Observer Implementation**

**Filename:** Main.java

public class Main {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobileUser1 = new MobileApp("Alice");

Observer webUser1 = new WebApp("Bob");

stockMarket.registerObserver(mobileUser1);

stockMarket.registerObserver(webUser1);

stockMarket.setStockPrice(1500.50);

stockMarket.setStockPrice(1523.75);

stockMarket.removeObserver(webUser1);

stockMarket.setStockPrice(1550.00);

}

}

### **Output (Expected):**

yaml

Stock price updated to: ₹1500.5

MobileApp - Alice: Stock price changed to ₹1500.5

WebApp - Bob: Stock price changed to ₹1500.5

Stock price updated to: ₹1523.75

MobileApp - Alice: Stock price changed to ₹1523.75

WebApp - Bob: Stock price changed to ₹1523.75

Stock price updated to: ₹1550.0

MobileApp - Alice: Stock price changed to ₹1550.0

**Exercise 8: Implementing the Strategy Pattern**

### **Step 1: Create a New Java Project**

**Project Name:** StrategyPatternExample

### **Step 2: Define the Strategy Interface**

**Filename:** PaymentStrategy.java

public interface PaymentStrategy {

void pay(double amount);

}

### **Step 3: Implement Concrete Strategies**

**Filename:** CreditCardPayment.java

public class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String cardHolder;

public CreditCardPayment(String cardNumber, String cardHolder) {

this.cardNumber = cardNumber;

this.cardHolder = cardHolder;

}

@Override

public void pay(double amount) {

System.out.println("Paid ₹" + amount + " using Credit Card (Card Holder: " + cardHolder + ").");

}

}

**Filename:** PayPalPayment.java

public class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

@Override

public void pay(double amount) {

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

}

}

### **Step 4: Implement the Context Class**

**Filename:** PaymentContext.java

public class PaymentContext {

private PaymentStrategy paymentStrategy;

// Set the strategy dynamically

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

// Use the strategy to perform payment

public void executePayment(double amount) {

if (paymentStrategy == null) {

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

} else {

paymentStrategy.pay(amount);

}

}

}

### **Step 5: Test the Strategy Implementation**

**Filename:** Main.java

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

// Use Credit Card payment

context.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-3456", "Aakanksha G."));

context.executePayment(2500);

// Switch to PayPal payment

context.setPaymentStrategy(new PayPalPayment("aakanksha@example.com"));

context.executePayment(1500);

}

}

### **Output (Expected):**

less

Paid ₹2500.0 using Credit Card (Card Holder: Aakanksha G.).

Paid ₹1500.0 using PayPal (Email: aakanksha@example.com).

**Exercise 9: Implementing the Command Pattern**

### **Step 1: Create a New Java Project**

**Project Name:** CommandPatternExample

### **Step 2: Define the Command Interface**

**Filename:** Command.java

public interface Command {

void execute();

}

### **Step 3: Implement Concrete Command Classes**

**Filename:** LightOnCommand.java

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

**Filename:** LightOffCommand.java

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

### **Step 4: Implement the Invoker Class**

**Filename:** RemoteControl.java

public class RemoteControl {

private Command command;

// Set the command to be executed

public void setCommand(Command command) {

this.command = command;

}

// Execute the command

public void pressButton() {

if (command != null) {

command.execute();

} else {

System.out.println("No command assigned to the button.");

}

}

}

### **Step 5: Implement the Receiver Class**

**Filename:** Light.java

public class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

### **Step 6: Test the Command Implementation**

**Filename:** Main.java

public class Main {

public static void main(String[] args) {

// Receiver

Light livingRoomLight = new Light();

// Concrete Commands

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

// Invoker

RemoteControl remote = new RemoteControl();

// Turn the light on

remote.setCommand(lightOn);

remote.pressButton();

// Turn the light off

remote.setCommand(lightOff);

remote.pressButton();

}

}

### **Output (Expected):**

vbnet

The light is ON.

The light is OFF.

**Exercise 10: Implementing the MVC Pattern**.

### **Step 1: Create a New Java Project**

**Project Name:** MVCPatternExample

### **Step 2: Define the Model Class**

**Filename:** Student.java

public class Student {

private String name;

private String id;

private String grade;

// Constructor

public Student(String name, String id, String grade) {

this.name = name;

this.id = id;

this.grade = grade;

}

// Getters and setters

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getGrade() {

return grade;

}

public void setGrade(String grade) {

this.grade = grade;

}

}

### **Step 3: Define the View Class**

**Filename:** StudentView.java

public class StudentView {

public void displayStudentDetails(String name, String id, String grade) {

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

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

System.out.println("ID: " + id);

System.out.println("Grade: " + grade);

System.out.println();

}

}

### **Step 4: Define the Controller Class**

**Filename:** StudentController.java

public class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void setStudentName(String name) {

model.setName(name);

}

public void setStudentId(String id) {

model.setId(id);

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public String getStudentName() {

return model.getName();

}

public String getStudentId() {

return model.getId();

}

public String getStudentGrade() {

return model.getGrade();

}

public void updateView() {

view.displayStudentDetails(model.getName(), model.getId(), model.getGrade());

}

}

### **Step 5: Test the MVC Implementation**

**Filename:** Main.java

public class Main {

public static void main(String[] args) {

// Step 1: Create the model

Student student = new Student("Aakanksha", "S101", "A");

// Step 2: Create the view

StudentView view = new StudentView();

// Step 3: Create the controller

StudentController controller = new StudentController(student, view);

// Initial display

controller.updateView();

// Modify student data

controller.setStudentName("Aakanksha Ghodki");

controller.setStudentGrade("A+");

// Display updated data

controller.updateView();

}

}

### **Output (Expected):**

yaml

Student Details:

Name: Aakanksha

ID: S101

Grade: A

Student Details:

Name: Aakanksha Ghodki

ID: S101

Grade: A+

**Exercise 11: Implementing Dependency Injection**

### **Step 1: Create a New Java Project**

**Project Name:** DependencyInjectionExample

### **Step 2: Define the Repository Interface**

**Filename:** CustomerRepository.java

public interface CustomerRepository {

String findCustomerById(int id);

}

### **Step 3: Implement the Concrete Repository**

**Filename:** CustomerRepositoryImpl.java

public class CustomerRepositoryImpl implements CustomerRepository {

@Override

public String findCustomerById(int id) {

// Simulating a customer record from a database

return "Customer{id=" + id + ", name='Aakanksha Ghodki'}";

}

}

### **Step 4: Define the Service Class**

**Filename:** CustomerService.java

public class CustomerService {

private CustomerRepository customerRepository;

// Step 5: Constructor injection

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void displayCustomer(int id) {

String customer = customerRepository.findCustomerById(id);

System.out.println("Fetched: " + customer);

}

}

### **Step 6: Test the Dependency Injection Implementation**

**Filename:** Main.java

public class Main {

public static void main(String[] args) {

// Creating the repository

CustomerRepository repository = new CustomerRepositoryImpl();

// Injecting the dependency into the service

CustomerService service = new CustomerService(repository);

// Using the service

service.displayCustomer(101);

}

}

### **Output (Expected):**

bash

Fetched: Customer{id=101, name='Aakanksha Ghodki'}