**1. Singleton Pattern:**

### **Step 1:** Create a new Java project named SingletonPatternExample.

**Step 2:** Create the Logger Class

**Logger.java:**

package singleton;

public class Logger {

*// Step 2.1: Create a private static instance of Logger*

private static Logger instance;

*// Step 2.2: Make the constructor private*

private Logger() {

*// private constructor to prevent instantiation*

}

*// Step 2.3: Provide a public static method to get the instance of Logger*

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

// Logger method to add log messages

public void log(String message) {

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

}

}

**Step 3:** Create the TestLogger Class

**TestLogger.java:**

package singleton;

public class TestLogger {

public static void main(String[] args) {

*// Step 4.1: Get the instance of Logger*

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

*// Step 4.2: Log messages using both instances*

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

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

*// Step 4.3: Verify that both instances are the same*

if (logger1 == logger2) {

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

} else {

System.out.println("logger1 and logger2 are different instances.");

}

}

}

**2. Factory Method Pattern:**

**Step 1: Define Document Classes**

Create interfaces or abstract classes for different document types such as WordDocument, PdfDocument, and ExcelDocument.

**Document.java:**

package factorymethod;

public interface Document {

void open();

void close();

void save();

}

**Step 2: Create Concrete Document Classes**

Implement concrete classes for each document type that implements the Document interface.

**WordDocument.java:**

package factorymethod;

public class WordDocument implements Document {

@Override

public void open() {

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

}

@Override

public void close() {

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

}

@Override

public void save() {

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

}

}

**PdfDocument.java:**

package factorymethod;

public class PdfDocument implements Document {

@Override

public void open() {

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

}

@Override

public void close() {

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

}

@Override

public void save() {

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

}

}

**ExcelDocument.java:**

package factorymethod;

public class ExcelDocument implements Document {

@Override

public void open() {

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

}

@Override

public void close() {

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

}

@Override

public void save() {

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

}

}

**Step 3:** Implement the Factory Method

Create an abstract class DocumentFactory with a method createDocument().

**DocumentFactory.java:**

package factorymethod;

public abstract class DocumentFactory {

public abstract Document createDocument();

}

Create concrete factories for each document type.

#### **WordDocumentFactory.java:**

package factorymethod;

public class WordDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

**PdfDocumentFactory.java:**

package factorymethod;

public class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

**ExcelDocumentFactory.java:**

package factorymethod;

public class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

### **Step 4: Test the Factory Method Implementation**

Create a test class to verify the factory method implementation.

#### **TestFactoryMethod.java:**

package factorymethod;

public class TestFactoryMethod {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

wordDoc.save();

wordDoc.close();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

pdfDoc.save();

pdfDoc.close();

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

excelDoc.save();

excelDoc.close();

}

}

**3. Builder Pattern:**

### **Step 1: Define a Product Class**

Create a class Computer with attributes like CPU, RAM, Storage, etc.

#### **Computer.java:**

public class Computer {

// Define attributes

private String CPU;

private String RAM;

private String storage;

private String GPU;

private String powerSupply;

private String coolingSystem;

// Private constructor to take Builder as parameter

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.GPU = builder.GPU;

this.powerSupply = builder.powerSupply;

this.coolingSystem = builder.coolingSystem;

}

@Override

public String toString() {

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

", GPU=" + GPU + ", powerSupply=" + powerSupply +

", coolingSystem=" + coolingSystem + "]";

}

// Step 3.1: Static nested Builder class

public static class Builder {

private String CPU;

private String RAM;

private String storage;

private String GPU;

private String powerSupply;

private String coolingSystem;

// Methods to set each attribute

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 setGPU(String GPU) {

this.GPU = GPU;

return this;

}

public Builder setPowerSupply(String powerSupply) {

this.powerSupply = powerSupply;

return this;

}

public Builder setCoolingSystem(String coolingSystem) {

this.coolingSystem = coolingSystem;

return this;

}

// build() method to return Computer instance

public Computer build() {

return new Computer(this);

}

}

}

### **Step 2: Test the Builder Implementation**

Create a test class to demonstrate the creation of different configurations of Computer using the Builder pattern.

#### **TestBuilderPattern.java:**

public class TestBuilderPattern {

public static void main(String[] args) {

// Create different configurations using the Builder pattern

Computer gamingPC = new Computer.Builder()

.setCPU("Intel i9")

.setRAM("32GB")

.setStorage("1TB SSD")

.setGPU("NVIDIA RTX 3080")

.setPowerSupply("750W")

.setCoolingSystem("Liquid Cooling")

.build();

Computer officePC = new Computer.Builder()

.setCPU("Intel i5")

.setRAM("16GB")

.setStorage("512GB SSD")

.build();

// Print configurations to verify

System.out.println(gamingPC);

System.out.println(officePC);

}

}

**4. Adapter Pattern:**

**Step 1:** Define Target Interface Create an interface PaymentProcessor with a method like processPayment().

**PaymentProcessor.java:**

public interface PaymentProcessor {

void processPayment(double amount);

}

**Step 2:** Implement Adapter Classes Create classes for different payment gateways with their own methods.

**PayPalGateway.java:**

public class PayPalGateway {

public void makePayment(double amount) {

System.out.println("Processing payment of $" + amount + " through PayPal.");

}

}

**StripeGateway.java:**

public class StripeGateway {

public void pay(double amount) {

System.out.println("Processing payment of $" + amount + " through Stripe.");

}

}

### **Step 3: Implement the Adapter Class**

Create an adapter class for each payment gateway that implements PaymentProcessor and translates the calls to the gateway-specific methods.

#### **PayPalAdapter.java:**

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPalGateway;

public PayPalAdapter(PayPalGateway payPalGateway) {

this.payPalGateway = payPalGateway;

}

@Override

public void processPayment(double amount) {

payPalGateway.makePayment(amount);

}

}

**StripeAdapter.java:**

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripeGateway;

public StripeAdapter(StripeGateway stripeGateway) {

this.stripeGateway = stripeGateway;

}

@Override

public void processPayment(double amount) {

stripeGateway.pay(amount);

}

}

**Step 4:** Test the Adapter Implementation Create a test class to demonstrate the use of different payment gateways through the adapter.

**TestAdapterPattern.java:**

public class TestAdapterPattern {

public static void main(String[] args) {

// Create PayPal payment

PayPalGateway payPal = new PayPalGateway();

PaymentProcessor payPalProcessor = new PayPalAdapter(payPal);

payPalProcessor.processPayment(100.00);

// Create Stripe payment

StripeGateway stripe = new StripeGateway();

PaymentProcessor stripeProcessor = new StripeAdapter(stripe);

stripeProcessor.processPayment(200.00);

}

}

**5. Decorator Pattern:**

### **Step 1:** Define Component Interface.Create an interface Notifier with a method send().

#### **Notifier.java:**

public interface Notifier {

void send(String message);

}

**Step 2**: Implement Concrete Component Create a class EmailNotifier that implements Notifier.

**EmailNotifier.java:**

public class EmailNotifier implements Notifier {

@Override

public void send(String message) {

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

}

}

**Step 3:** Implement Decorator Classes Create an abstract decorator class NotifierDecorator that implements Notifier and holds a reference to a Notifier object.

**NotifierDecorator.java:**

public abstract class NotifierDecorator implements Notifier {

protected Notifier wrapped;

public NotifierDecorator(Notifier wrapped) {

this.wrapped = wrapped;

}

@Override

public void send(String message) {

wrapped.send(message);

}

}

Create concrete decorator classes like SMSNotifierDecorator and SlackNotifierDecorator that extend NotifierDecorator.

**SMSNotifierDecorator.java:**

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier wrapped) {

super(wrapped);

}

@Override

public void send(String message) {

super.send(message);

sendSMS(message);

}

private void sendSMS(String message) {

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

}

}

**SlackNotifierDecorator.java:**

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier wrapped) {

super(wrapped);

}

@Override

public void send(String message) {

super.send(message);

sendSlackMessage(message);

}

private void sendSlackMessage(String message) {

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

}

}

**Step 4:** Test the Decorator Implementation Create a test class to demonstrate sending notifications via multiple channels using decorators.

**TestDecoratorPattern.java:**

public class TestDecoratorPattern {

public static void main(String[] args) {

Notifier emailNotifier = new EmailNotifier();

Notifier smsNotifier = new SMSNotifierDecorator(emailNotifier);

Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);

String message = "This is a test notification.";

// Send notifications through multiple channels

slackNotifier.send(message);

}

}

**6. Proxy Pattern:**

**Step 1:** Define Subject Interface Create an interface Image with a method display().

**Image.java:**

public interface Image {

void display();

}

**Step 2**: Implement Real Subject Class Create a class RealImage that implements Image and loads an image from a remote server.

**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 " + filename + " from remote server...");

}

@Override

public void display() {

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

}

}

**Step 3:** Implement Proxy Class Create a class ProxyImage that implements Image and holds a reference to RealImage. Implement lazy initialization and caching in ProxyImage.

**ProxyImage.java:**

public class ProxyImage implements Image {

private String filename;

private RealImage realImage;

public ProxyImage(String filename) {

this.filename = filename;

}

@Override

public void display() {

if (realImage == null) {

realImage = new RealImage(filename);

}

realImage.display();

}

}

**Step 4:** Test the Proxy Implementation Create a test class to demonstrate the use of ProxyImage to load and display images.

**TestProxyPattern.java:**

public class TestProxyPattern {

public static void main(String[] args) {

Image image1 = new ProxyImage("photo1.jpg");

Image image2 = new ProxyImage("photo2.jpg");

// Images will be loaded from the remote server when display is called for the first time

image1.display();

image1.display(); // This time it should not load from the remote server

// Second image

image2.display();

image2.display(); // This time it should not load from the remote server

}

}

**7. Observer Pattern:**

**Step 1:** Define Subject Interface Create an interface Stock with methods to register, deregister, and notify observers.

**Stock.java:**

public interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

**Step 2:** Implement Concrete Subject Create a class StockMarket that implements Stock and maintains a list of observers.

**StockMarket.java:**

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

private List<Observer> observers;

private double stockPrice;

public StockMarket() {

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(stockPrice);

}

}

public void setStockPrice(double stockPrice) {

this.stockPrice = stockPrice;

notifyObservers();

}

}

**Step 3:** Define Observer Interface Create an interface Observer with a method update().

**Observer.java:**

public interface Observer {

void update(double stockPrice);

}

**Step 4:** Implement Concrete Observers Create classes MobileApp and WebApp that implement Observer.

**MobileApp.java:**

public class MobileApp implements Observer {

private String appName;

public MobileApp(String appName) {

this.appName = appName;

}

@Override

public void update(double stockPrice) {

System.out.println(appName + " received stock price update: " + stockPrice);

}

}

**WebApp.java:**

public class WebApp implements Observer {

private String appName;

public WebApp(String appName) {

this.appName = appName;

}

@Override

public void update(double stockPrice) {

System.out.println(appName + " received stock price update: " + stockPrice);

}}

**Step 5:** Test the Observer Implementation Create a test class to demonstrate the registration and notification of observers.

**TestObserverPattern.java:**

public class TestObserverPattern {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobileApp = new MobileApp("Mobile App");

Observer webApp = new WebApp("Web App");

stockMarket.registerObserver(mobileApp);

stockMarket.registerObserver(webApp);

stockMarket.setStockPrice(100.00);

stockMarket.setStockPrice(105.50);

stockMarket.setStockPrice(98.75);

stockMarket.removeObserver(webApp);

stockMarket.setStockPrice(102.00);

}

}

**8. Strategy Pattern:**

**Step 2:** Define Strategy Interface Create an interface PaymentStrategy with a method pay().

**PaymentStrategy.java:**

public interface PaymentStrategy {

void pay(double amount);

}

**Step 2**: Implement Concrete Strategies Create classes CreditCardPayment and PayPalPayment that implement PaymentStrategy.

**CreditCardPayment.java:**

public class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String cardHolderName;

private String cvv;

private String expiryDate;

public CreditCardPayment(String cardNumber, String cardHolderName, String cvv, String expiryDate) {

this.cardNumber = cardNumber;

this.cardHolderName = cardHolderName;

this.cvv = cvv;

this.expiryDate = expiryDate;

}

@Override

public void pay(double amount) {

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

// Add real payment processing logic here

}

}

**PayPalPayment.java:**

public class PayPalPayment implements PaymentStrategy {

private String email;

private String password;

public PayPalPayment(String email, String password) {

this.email = email;

this.password = password;

}

@Override

public void pay(double amount) {

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

// Add real payment processing logic here

}

}

**Step 3:** Implement Context Class Create a class PaymentContext that holds a reference to PaymentStrategy and a method to execute the strategy. **PaymentContext.java:**

public class PaymentContext {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void pay(double amount) {

if (paymentStrategy != null) {

paymentStrategy.pay(amount);

} else {

System.out.println("Payment strategy not set.");

}

}

}

**Step 4:** Test the Strategy Implementation Create a test class to demonstrate selecting and using different payment strategies.

**TestStrategyPattern.java:**

public class TestStrategyPattern {

public static void main(String[] args) {

PaymentContext paymentContext = new PaymentContext();

// Pay with Credit Card

PaymentStrategy creditCardPayment = new CreditCardPayment("1234567890123456", "John Doe", "123", "12/24");

paymentContext.setPaymentStrategy(creditCardPayment);

paymentContext.pay(250.0);

// Pay with PayPal

PaymentStrategy payPalPayment = new PayPalPayment("john.doe@example.com", "password123");

paymentContext.setPaymentStrategy(payPalPayment);

paymentContext.pay(150.0);

}

}

**9.Command Pattern:**

**Step 1:** Define Command Interface Create an interface Command with a method execute().

**Command.java:**

public interface Command {

void execute();

}

**Step 2**: Implement Concrete Commands Create classes LightOnCommand and LightOffCommand that implement Command.

**LightOnCommand.java:**

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

**LightOffCommand.java:**

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

**Step 3:** Implement Invoker Class Create a class RemoteControl that holds a reference to a Command and a method to execute the command. **RemoteControl.java:**

public class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

if (command != null) {

command.execute();

} else {

System.out.println("No command set.");

}

}

}

**Step 4:** Implement Receiver Class Create a class Light with methods to turn on and off.

**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 Create a test class to demonstrate issuing commands using the RemoteControl.

**TestCommandPattern.java:**

public class TestCommandPattern {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remoteControl = new RemoteControl();

// Turn the light on

remoteControl.setCommand(lightOn);

remoteControl.pressButton();

// Turn the light off

remoteControl.setCommand(lightOff);

remoteControl.pressButton();

}

}

**10. MVC Pattern:**

**Step 1:** Define Model Class Create a class Student with attributes like name, id, and grade.

**Student.java:**

public class Student {

private String id;

private String name;

private String grade;

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

this.id = id;

this.name = name;

this.grade = grade;

}

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getGrade() {

return grade;

}

public void setGrade(String grade) {

this.grade = grade;

}

}

**Step 2:** Define View Class Create a class StudentView with a method displayStudentDetails().

**StudentView.java:**

public class StudentView {

public void displayStudentDetails(String studentName, String studentId, String studentGrade) {

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

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

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

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

}

}

**Step 3**: Define Controller Class Create a class StudentController that handles the communication between the model and the view.

**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 String getStudentName() {

return model.getName();

}

public void setStudentId(String id) {

model.setId(id);

}

public String getStudentId() {

return model.getId();

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public String getStudentGrade() {

return model.getGrade();

}

public void updateView() {

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

}

}

**Step 4:** Test the MVC Implementation Create a main class to demonstrate creating a Student, updating its details using StudentController, and displaying them using StudentView.

**MVCPatternDemo.java:**

public class MVCPatternDemo {

public static void main(String[] args) {

// Create a student record

Student model = new Student("1", "John Doe", "A");

// Create a view to display student details

StudentView view = new StudentView();

// Create a controller

StudentController controller = new StudentController(model, view);

// Display initial student details

controller.updateView();

// Update student details

controller.setStudentName("Jane Doe");

controller.setStudentGrade("A+");

// Display updated student details

controller.updateView();

}

}

**11. Dependency Injection:**

**Step 1:** Define Repository Interface Create an interface CustomerRepository with a method findCustomerById().

**CustomerRepository.java:**

public interface CustomerRepository {

String findCustomerById(String id);

}

**Step 2**: Implement Concrete Repository Create a class CustomerRepositoryImpl that implements CustomerRepository.

**CustomerRepositoryImpl.java:**

public class CustomerRepositoryImpl implements CustomerRepository {

@Override

public String findCustomerById(String id) {

// Simulate a database call

return "Customer with ID: " + id;

}

}

**Step 3:** Define Service Class Create a class CustomerService that depends on CustomerRepository.

**CustomerService.java:**

public class CustomerService {

private CustomerRepository customerRepository;

// Constructor injection

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void displayCustomer(String id) {

String customer = customerRepository.findCustomerById(id);

System.out.println(customer);

}

}

**Step 4:** Implement Dependency Injection Use constructor injection to inject CustomerRepository into CustomerService.

**Step 5**: Test the Dependency Injection Implementation Create a main class to demonstrate creating a CustomerService with CustomerRepositoryImpl and using it to find a customer.

**DependencyInjectionDemo.java:**

public class DependencyInjectionDemo {

public static void main(String[] args) {

// Create the repository

CustomerRepository repository = new CustomerRepositoryImpl();

// Inject the repository into the service

CustomerService service = new CustomerService(repository);

// Use the service to find and display a customer

service.displayCustomer("12345");

}

}