**Design Patterns and Principles**

**Exercise 1: Implementing the Singleton Pattern**

(i). Define the Singleton Class

public class Logger {

private static Logger instance;

// Private constructor to prevent instantiation

private Logger() {

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

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger(); // lazy initialization

}

return instance;

}

// Example method

public void log(String message) {

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

}

}

(ii). Test the Singleton Implementation

public class LoggerTest {

public static void main(String[] args) {

// Get Logger instances

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

// Log messages

logger1.log("This is the first log message.");

logger2.log("This is the second log message.");

if (logger1 == logger2) {

System.out.println("Only one instance of Logger exists.");

} else {

System.out.println("Multiple instances of Logger exist.");

}

}

}

**Output:**

Logger initialized

[LOG]: This is the first log message.

[LOG]: This is the second log message.

Only one instance of Logger exists.

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

(i). Document Interface

public interface Document {

void open();

}

(ii). Concrete Document Classes

(a).WordDocument.java

public class WordDocument implements Document {

public void open() {

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

}

(b). PdfDocument.java

public class PdfDocument implements Document {

public void open() {

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

}

}

(c). ExcelDocument.java

public class ExcelDocument implements Document {

public void open() {

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

}

}

(iii).Implement the Factory Method

(a).Abstract Factory — DocumentFactory.java

public abstract class DocumentFactory {

public abstract Document createDocument();

}

(b).Concrete Factories

**(1).WordDocumentFactory.java**

public class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

**(2).PdfDocumentFactory.java**

public class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

**(3). ExcelDocumentFactory.java**

public class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

(iii).Test the Factory Method

public class FactoryTest {

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:**

Opening Word document.

Opening PDF document.

Opening Excel document.

**Exercise 3: Implementing the Builder Pattern**

(i).Product Class with Nested Builder

public class Computer {

// Required attributes

private String CPU;

private String RAM;

// Optional attributes

private String storage;

private String graphicsCard;

private String operatingSystem;

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

this.operatingSystem = builder.operatingSystem;

}

public static class Builder {

// Required

private String CPU;

private String RAM;

// Optional

private String storage;

private String graphicsCard;

private String operatingSystem;

public Builder(String CPU, String RAM) {

this.CPU = CPU;

this.RAM = RAM;

}

public Builder setStorage(String storage) {

this.storage = storage;

return this;

}

public Builder setGraphicsCard(String graphicsCard) {

this.graphicsCard = graphicsCard;

return this;

}

public Builder setOperatingSystem(String operatingSystem) {

this.operatingSystem = operatingSystem;

return this;

}

public Computer build() {

return new Computer(this);

}

}

// Display method

public void showSpecs() {

System.out.println("CPU: " + CPU);

System.out.println("RAM: " + RAM);

System.out.println("Storage: " + (storage != null ? storage : "N/A"));

System.out.println("Graphics Card: " + (graphicsCard != null ? graphicsCard : "N/A"));

System.out.println("Operating System: " + (operatingSystem != null ? operatingSystem : "N/A"));

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

}

}

(ii).Test Class

public class BuilderTest {

public static void main(String[] args) {

Computer basicComputer = new Computer.Builder("Intel i5", "8GB").build();

basicComputer.showSpecs();

Computer gamingComputer = new Computer.Builder("AMD Ryzen 9", "32GB")

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 4080")

.setOperatingSystem("Windows 11")

.build();

gamingComputer.showSpecs();

Computer officeComputer = new Computer.Builder("Intel i7", "16GB")

.setStorage("512GB SSD")

.setOperatingSystem("Ubuntu Linux")

.build();

officeComputer.showSpecs();

}

}

**Output:**

CPU: Intel i5

RAM: 8GB

Storage: N/A

Graphics Card: N/A

Operating System: N/A

-----

CPU: AMD Ryzen 9

RAM: 32GB

Storage: 1TB SSD

Graphics Card: NVIDIA RTX 4080

Operating System: Windows 11

-----

CPU: Intel i7

RAM: 16GB

Storage: 512GB SSD

Graphics Card: N/A

Operating System: Ubuntu Linux

**Exercise 4: Implementing the Adapter Pattern**

(i). Target Interface

public interface PaymentProcessor {

void processPayment(double amount);

}

(ii). Adaptee Classes

**(a). PayPalGateway.java**

public class PayPalGateway {

public void makePayment(String accountEmail, double amount) {

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

}

}

**(b).StripeGateway.java**

public class StripeGateway {

public void chargeCard(String cardNumber, double amount) {

System.out.println("Charged " + amount + " to card: " + cardNumber + " via Stripe");

}

}

(iii). Adapter Classes

(a). PayPalAdapter.java

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPalGateway;

private String accountEmail;

public PayPalAdapter(String accountEmail) {

this.accountEmail = accountEmail;

this.payPalGateway = new PayPalGateway();

}

public void processPayment(double amount) {

payPalGateway.makePayment(accountEmail, amount);

}

}

(b).StripeAdapter.java

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripeGateway;

private String cardNumber;

public StripeAdapter(String cardNumber) {

this.cardNumber = cardNumber;

this.stripeGateway = new StripeGateway();

}

public void processPayment(double amount) {

stripeGateway.chargeCard(cardNumber, amount);

}

}

(iii).Test the Adapter

public class AdapterTest {

public static void main(String[] args) {

PaymentProcessor paypalProcessor = new PayPalAdapter("user@example.com");

PaymentProcessor stripeProcessor = new StripeAdapter("1234-5678-9012-3456");

// Process payments through a unified interface

paypalProcessor.processPayment(250.0);

stripeProcessor.processPayment(500.0);

}

}

**Output:**

Paid 250.0 using PayPal account: user@example.com

Charged 500.0 to card: 1234-5678-9012-3456 via Stripe

**Exercise 5: Implementing the Decorator Pattern**

(i). Component Interface

public interface Notifier {

void send(String message);

}

(ii). Concrete Component

public class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

(iii). Decorator Classes

(a). Abstract Decorator — NotifierDecorator.java

public abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send(String message) {

notifier.send(message); // delegate to wrapped notifier

}

}

(b). Concrete Decorator: SMSNotifierDecorator.java

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

sendSMS(message);

}

private void sendSMS(String message) {

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

}

}

(c). Concrete Decorator: SlackNotifierDecorator.java

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

sendSlack(message);

}

private void sendSlack(String message) {

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

}

}

(iv). Test the Decorator

public class DecoratorTest {

public static void main(String[] args) {

Notifier baseNotifier = new EmailNotifier();

Notifier smsNotifier = new SMSNotifierDecorator(baseNotifier);

Notifier fullNotifier = new SlackNotifierDecorator(smsNotifier);

fullNotifier.send("Server is down!");

}

}

**Output:**

Sending Email: Server is down!

Sending SMS: Server is down!

Sending Slack message: Server is down!

**Exercise 6: Implementing the Proxy Pattern**

(i). Subject Interface

public interface Image {

void display();

}

(ii). Implement Real Subject Class

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);

}

public void display() {

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

}

}

(iii). Implement Proxy Class:

public class ProxyImage implements Image {

private RealImage realImage;

private String fileName;

public ProxyImage(String fileName) {

this.fileName = fileName;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(fileName); // lazy loading

} else {

System.out.println("Image loaded from cache: " + fileName);

}

realImage.display();

}

}

(iv). Test Class

public class ProxyTest {

public static void main(String[] args) {

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

image1.display();

image1.display();

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

image2.display();

}

}

**Exercise 7: Implementing the Observer Pattern**

(i). Subject Interface

public interface Stock {

void registerObserver(Observer observer);

void removeObserver(Observer observer);

void notifyObservers();

}

(ii). Implement Concrete Subject

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

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

private String stockName;

private double price;

public void setStockPrice(String stockName, double price) {

this.stockName = stockName;

this.price = price;

notifyObservers();

}

public void registerObserver(Observer observer) {

observers.add(observer);

}

public void removeObserver(Observer observer) {

observers.remove(observer);

}

public void notifyObservers() {

for (Observer obs : observers) {

obs.update(stockName, price);

}

}

}

(iii). Observer Interface

public interface Observer {

void update(String stockName, double price);

}

(iv). Implement Concrete Observers

(a).MobileApp.java

public class MobileApp implements Observer {

private String name;

public MobileApp(String name) {

this.name = name;

}

public void update(String stockName, double price) {

System.out.println("[" + name + " - Mobile] " + stockName + " price updated to $" + price);

}

}

(b). WebApp.java

public class WebApp implements Observer {

private String name;

public WebApp(String name) {

this.name = name;

}

public void update(String stockName, double price) {

System.out.println("[" + name + " - Web] " + stockName + " price updated to $" + price);

}

}

(v). Test the Observer

public class ObserverTest {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobileUser1 = new MobileApp("Alice");

Observer webUser1 = new WebApp("Bob");

//register

stockMarket.registerObserver(mobileUser1);

stockMarket.registerObserver(webUser1);

//stockprice

stockMarket.setStockPrice("AAPL", 190.50);

stockMarket.setStockPrice("GOOGL", 2743.00);

//remove observer

stockMarket.removeObserver(webUser1);

stockMarket.setStockPrice("TSLA", 775.25);

}

}

**Output:**

[Alice - Mobile] AAPL price updated to $190.5

[Bob - Web] AAPL price updated to $190.5

[Alice - Mobile] GOOGL price updated to $2743.0

[Bob - Web] GOOGL price updated to $2743.0

[Alice - Mobile] TSLA price updated to $775.25

**Exercise 8: Implementing the Strategy Pattern**

(i). Strategy Interface

public interface PaymentStrategy {

void pay(double amount);

}

(ii). Implement Concrete Strategies

(a). 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;

}

public void pay(double amount) {

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

}

}

(b). PayPalPayment.java

public class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

public void pay(double amount) {

System.out.println("Paid $" + amount + " using PayPal account: " + email);

}

}

(iii).Implement Context Class

public class PaymentContext {

private PaymentStrategy paymentStrategy;

//strategy at runtime

public void setPaymentStrategy(PaymentStrategy strategy) {

this.paymentStrategy = strategy;

}

public void payAmount(double amount) {

if (paymentStrategy == null) {

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

} else {

paymentStrategy.pay(amount);

}

}

}

(iv). Test the Strategy

public class StrategyTest {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

PaymentStrategy creditCard = new CreditCardPayment("1234-5678-9012-3456", "Alice");

context.setPaymentStrategy(creditCard);

context.payAmount(150.75);

// Switch to PayPal strategy

PaymentStrategy paypal = new PayPalPayment("alice@example.com");

context.setPaymentStrategy(paypal);

context.payAmount(89.99);

}

}

**Output:**

Paid $150.75 using Credit Card: 1234-5678-9012-3456 (Card Holder: Alice)

Paid $89.99 using PayPal account: [alice@example.com](mailto:alice@example.com)

**Exercise 9: Implementing the Command Pattern**

(i). Command Interface

public interface Command {

void execute();

}

(ii). Implement Concrete Commands

(a).LightOnCommand.java

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

(b). LightOffCommand.java

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

(iii).Implement Invoker Class

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 assigned.");

}

}

}

(iv).Implement Receiver Class

public class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

(v). Test Class

public class CommandTest {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

// Turn light ON

remote.setCommand(lightOn);

remote.pressButton();

// Turn light OFF

remote.setCommand(lightOff);

remote.pressButton();

}

}

**Output:**

The light is ON.

The light is OFF.

**Exercise 10: Implementing the MVC Pattern**

(i). Model Class

public class Student {

private String name;

private String id;

private String grade;

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;

}

}

(ii). View Class

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("--------------------");

}

}

(iii). Controller Class

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());

}

}

(iv).Test the MVC:

public class MVCTest {

public static void main(String[] args) {

// Create model

Student student = new Student();

student.setName("Alice");

student.setId("S101");

student.setGrade("A");

// Create view

StudentView view = new StudentView();

// Create controller

StudentController controller = new StudentController(student, view);

// Display initial student info

controller.updateView();

// Update student info

controller.setStudentName("Alice Smith");

controller.setStudentGrade("A+");

// Display updated info

controller.updateView();

}

}

**Output:**

Student Details:

Name : Alice

ID : S101

Grade : A

Student Details:

Name : Alice Smith

ID : S101

Grade : A+

**Exercise 11: Implementing Dependency Injection**

(i). Repository Interface

public interface CustomerRepository {

Customer findCustomerById(String id);

}

(ii). Implement Concrete Repository

public class CustomerRepositoryImpl implements CustomerRepository {

public Customer findCustomerById(String id) {

return new Customer(id, "John Doe");

}

}

(iii). Service Class

public class CustomerService {

private CustomerRepository customerRepository;

// Constructor Injection

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void showCustomer(String id) {

Customer customer = customerRepository.findCustomerById(id);

System.out.println("Customer ID: " + customer.getId());

System.out.println("Customer Name: " + customer.getName());

}

}

(iv).Main class

public class DIAppTest {

public static void main(String[] args) {

// Create repository implementation

CustomerRepository repository = new CustomerRepositoryImpl();

// Inject repository into service using constructor

CustomerService service = new CustomerService(repository);

// Use the service

service.showCustomer("C001");

}

}

**Output:**

Customer ID: C001

Customer Name: John Doe