**Module 1 - Design Patterns and Principles**

**Exercise 1: Implementing the Singleton Pattern**

public class Logger {

private static Logger instance;

private Logger() {

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

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

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

}

}

public class LoggerTest {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

logger1.log("First message");

Logger logger2 = Logger.getInstance();

logger2.log("Second message");

if (logger1 == logger2) {

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

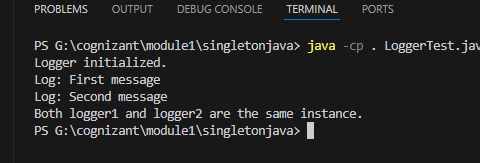
} else {

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

}

}

}



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

public class FactoryPatternTest {

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();

}

}

public class PdfDocument implements Document {

public void open() {

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

}

}

public class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

public class WordDocument implements Document {

public void open() {

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

}

}

public class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}



**Exercise 3: Implementing the Builder Pattern**

public class BuilderPatternTest {

public static void main(String[] args) {

// Create computer with all features

Computer gamingPC = new Computer.Builder("Intel i9", "32GB")

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 4080")

.build();

// Create computer with only mandatory features

Computer officePC = new Computer.Builder("Intel i5", "16GB").build();

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

gamingPC.showSpecs();

System.out.println("\nOffice PC:");

officePC.showSpecs();

}

}

public class Computer {

// Required parameters

private String cpu;

private String ram;

// Optional parameters

private String storage;

private String graphicsCard;

// Private constructor

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

}

// Static nested Builder class

public static class Builder {

private String cpu;

private String ram;

private String storage;

private String graphicsCard;

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 Computer build() {

return new Computer(this);

}

}

// Print method

public void showSpecs() {

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

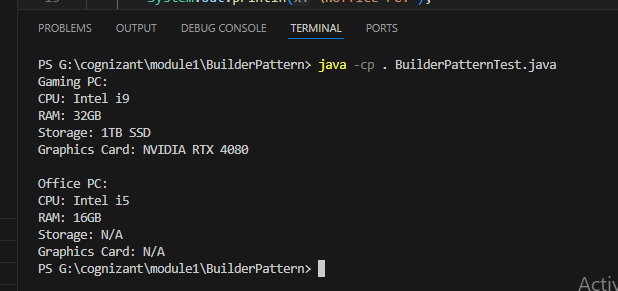
System.out.println("RAM: " + ram);

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

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

}

}



**Exercise 5: Implementing the Decorator Pattern**

interface Notifier {

void send();

}

class EmailNotifier implements Notifier {

@Override

public void send() {

System.out.println("Sending Email Notification");

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier wrappedNotifier;

public NotifierDecorator(Notifier notifier) {

this.wrappedNotifier = notifier;

}

public void send() {

wrappedNotifier.send();

}

}

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send() {

super.send();

sendSMS();

}

private void sendSMS() {

System.out.println("Sending SMS Notification");

}

}

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send() {

super.send();

sendSlack();

}

private void sendSlack() {

System.out.println("Sending Slack Notification");

}

}

public class DecoratorPatternExample {

public static void main(String[] args) {

Notifier emailNotifier = new EmailNotifier();

Notifier smsAndEmail = new SMSNotifierDecorator(emailNotifier);

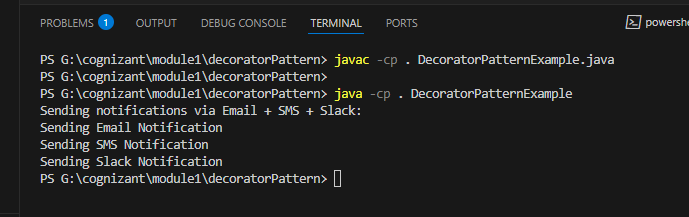
Notifier slackSmsAndEmail = new SlackNotifierDecorator(smsAndEmail);

System.out.println("Sending notifications via Email + SMS + Slack:");

slackSmsAndEmail.send();

}

}



**Exercise 6: Implementing the Proxy Pattern**

public class Main {

public static void main(String[] args) {

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

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

image1.display();

image1.display();

image2.display();

}

}

interface Image {

void display();

}

class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromServer();

}

private void loadFromServer() {

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

}

@Override

public void display() {

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

}

}

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); // Lazy initialization

} else {

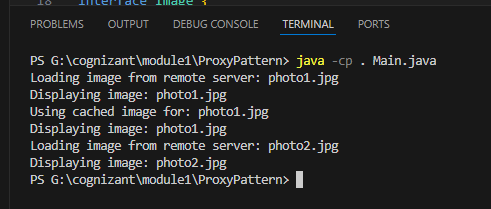
System.out.println("Using cached image for: " + filename);

}

realImage.display();

}

}



**Exercise 8: Implementing the Strategy Pattern**

interface PaymentStrategy {

void pay(double amount);

}

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 (" + cardHolder + ")");

}

}

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 (" + email + ")");

}

}

class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void pay(double amount) {

if (strategy != null) {

strategy.pay(amount);

} else {

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

}

}

}

public class StrategyPatternExample {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-3456", "John Doe"));

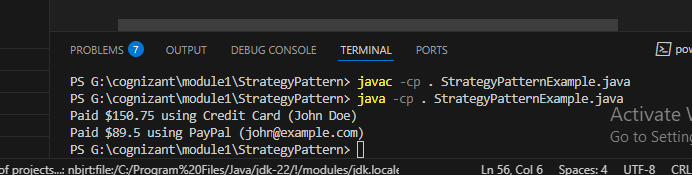
context.pay(150.75);

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

context.pay(89.50);

}

}



**Exercise 9: Implementing the Command Pattern**

interface Command {

void execute();

}

class Light {

public void turnOn() {

System.out.println("Light is ON");

}

public void turnOff() {

System.out.println("Light is OFF");

}

}

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

public class CommandPatternExample {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

remote.setCommand(lightOn);

remote.pressButton();

remote.setCommand(lightOff);

remote.pressButton();

}

}

