**Name:** Paarthivi D

**Superset ID:** 6410961

**DN 4.0 - Java FSE Additional Hands-on**

**WEEK – 1**

**Design Principles and Patterns**

**Exercise 3: Implementing the Builder Pattern**

**Code:**

package builderPatternExample;

public class BuilderPatternExample {

// Product Class

static class Computer {

// Required attributes

private String CPU;

private String RAM;

// Optional attributes

private String storage;

private String graphicsCard;

private String operatingSystem;

// Private constructor

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

this.operatingSystem = builder.operatingSystem;

}

// Static Nested Builder Class

public static class Builder {

// Required attributes

private String CPU;

private String RAM;

// Optional attributes

private String storage;

private String graphicsCard;

private String operatingSystem;

// Builder constructor for required attributes

public Builder(String CPU, String RAM) {

this.CPU = CPU;

this.RAM = RAM;

}

// Setter-style methods for optional attributes

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;

}

// Final build method

public Computer build() {

return new Computer(this);

}

}

public String toString() {

return "Computer Config:" +

"\nCPU: " + CPU +

"\nRAM: " + RAM +

"\nStorage: " + (storage != null ? storage : "Not Included") +

"\nGraphics Card: " + (graphicsCard != null ? graphicsCard : "Not Included") +

"\nOperating System: " + (operatingSystem != null ? operatingSystem : "Not Installed");

}

}

// Test Builder Implementation

public static void main(String[] args) {

// Basic Computer (only required parts)

Computer basicPC = new Computer.Builder("Intel i3", "4GB").build();

// Gaming Computer (all options)

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

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 4090")

.setOperatingSystem("Windows 11")

.build();

// Mid-range Computer

Computer midPC = new Computer.Builder("AMD Ryzen 5", "16GB")

.setStorage("512GB SSD")

.setOperatingSystem("Ubuntu")

.build();

// Print all configs

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

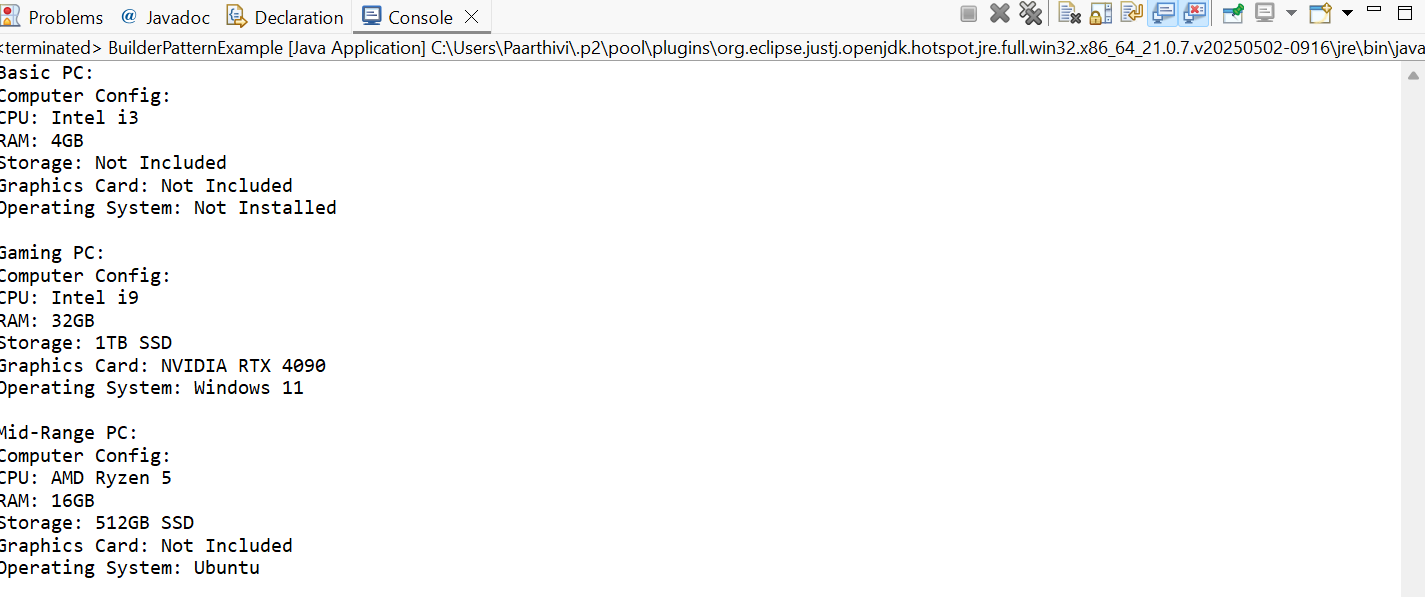
System.*out*.println("Gaming PC:\n" + gamingPC + "\n");

System.*out*.println("Mid-Range PC:\n" + midPC);

}

}

**Output Screenshot:**

****

**Exercise 4: Implementing the Adapter Pattern**

**Code:**

package adapterPatternExample;

public class AdapterPatternExample {

// Target Interface

interface PaymentProcessor {

void processPayment(double amount);

}

// Adaptee Class 1 - PayPal

static class PayPalGateway {

public void sendPayment(double amountInUSD) {

System.*out*.println("PayPal processed payment of $" + amountInUSD);

}

}

// Adaptee Class 2 - Stripe

static class StripeGateway {

public void makeStripePayment(double amount) {

System.*out*.println("Stripe processed payment of $" + amount);

}

}

// Adaptee Class 3 - Razorpay

static class RazorpayGateway {

public void payViaRazor(double amount) {

System.*out*.println("Razorpay processed payment of ₹" + amount);

}

}

// Adapter for PayPal

static class PayPalAdapter implements PaymentProcessor {

private PayPalGateway paypal = new PayPalGateway();

public void processPayment(double amount) {

paypal.sendPayment(amount);

}

}

// Adapter for Stripe

static class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe = new StripeGateway();

public void processPayment(double amount) {

stripe.makeStripePayment(amount);

}

}

// Adapter for Razorpay

static class RazorpayAdapter implements PaymentProcessor {

private RazorpayGateway razorpay = new RazorpayGateway();

public void processPayment(double amount) {

razorpay.payViaRazor(amount);

}

}

// Test Class

public static void main(String[] args) {

System.*out*.println("Unified Payment Processing via Adapters:\n");

PaymentProcessor paypalProcessor = new PayPalAdapter();

paypalProcessor.processPayment(150.00);

PaymentProcessor stripeProcessor = new StripeAdapter();

stripeProcessor.processPayment(220.75);

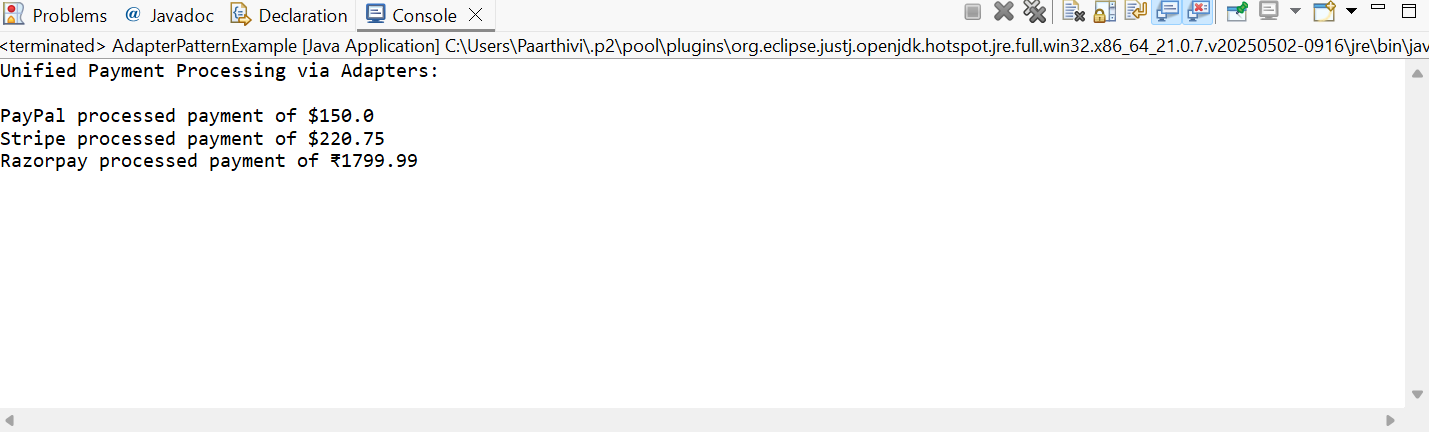
PaymentProcessor razorpayProcessor = new RazorpayAdapter();

razorpayProcessor.processPayment(1799.99);

}

}

**Output Screenshot:**



**Exercise 5: Implementing the Decorator Pattern**

**Code:**

package decoratorPatternExample;

public class DecoratorPatternExample {

// Component Interface

interface Notifier {

void send(String message);

}

// Concrete Component

static class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

// Abstract Decorator

static abstract class NotifierDecorator implements Notifier {

protected Notifier wrappedNotifier;

public NotifierDecorator(Notifier notifier) {

this.wrappedNotifier = notifier;

}

public void send(String message) {

wrappedNotifier.send(message); // delegate to wrapped component

}

}

// Concrete Decorator - SMS

static class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message); // send via wrapped notifier

sendSMS(message); // add SMS functionality

}

private void sendSMS(String message) {

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

}

}

// Concrete Decorator - Slack

static class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message); // send via wrapped notifier

sendSlack(message); // add Slack functionality

}

private void sendSlack(String message) {

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

}

}

// Test Class

public static void main(String[] args) {

// Base notifier: Email

Notifier notifier = new EmailNotifier();

// Decorate with SMS

notifier = new SMSNotifierDecorator(notifier);

// Further decorate with Slack

notifier = new SlackNotifierDecorator(notifier);

// Send message via all channels

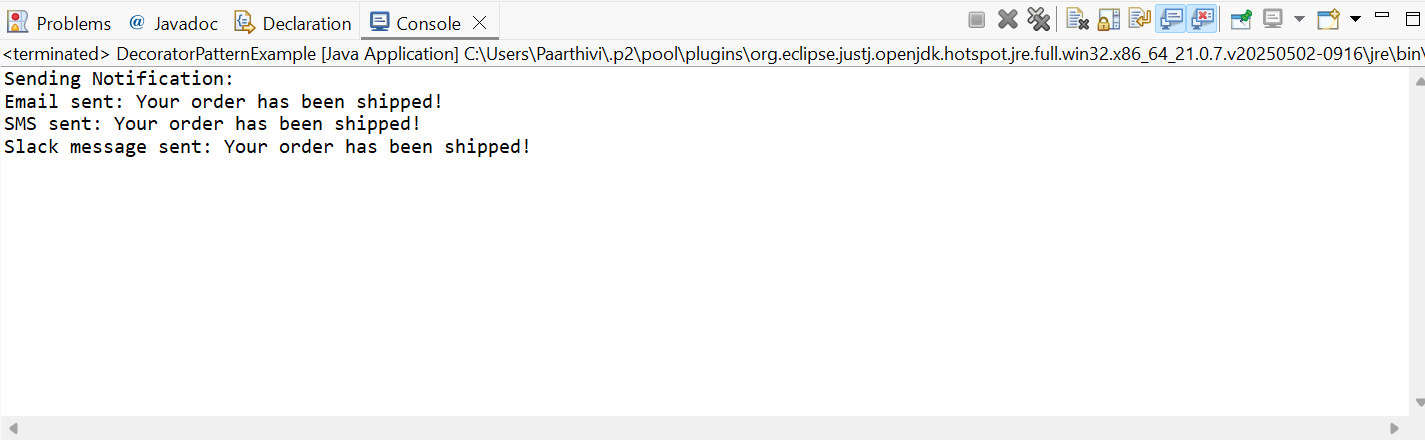
System.*out*.println("Sending Notification:");

notifier.send("Your order has been shipped!");

}

}

**Output Screenshot:**



**Exercise 6: Implementing the Proxy Pattern**

**Code:**

package proxyPatternExample;

public class ProxyPatternExample {

// Subject Interface

interface Image {

void display();

}

// Real Subject

static class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromServer(); // simulate heavy operation

}

private void loadFromServer() {

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

// Simulated delay

try {

Thread.*sleep*(1000); // Simulating network delay

} catch (InterruptedException e) {

e.printStackTrace();

}

}

public void display() {

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

}

}

// Proxy Class

static class ProxyImage implements Image {

private String filename;

private RealImage realImage;

public ProxyImage(String filename) {

this.filename = filename;

}

public void display() {

if (realImage == null) {

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

}

realImage.display(); // use cached object

}

}

// Test Class

public static void main(String[] args) {

System.*out*.println("Image Viewer Simulation using Proxy Pattern:\n");

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

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

// First time load - real loading happens

image1.display();

System.*out*.println();

// Second time - uses cache

image1.display();

System.*out*.println();

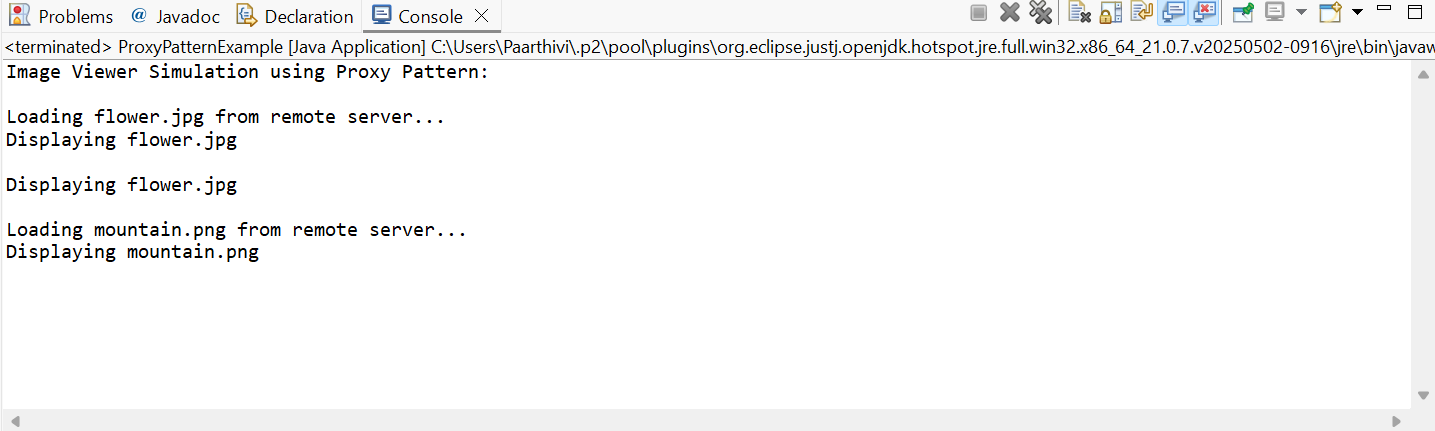
// First time load for image2

image2.display();

}

}

**Output Screenshot:**



**Exercise 7: Implementing the Observer Pattern**

**Code:**

package observerPatternExample;

import java.util.\*;

//Subject Interface

interface Stock {

void registerObserver(Observer observer);

void removeObserver(Observer observer);

void notifyObservers();

}

//Observer Interface

interface Observer {

void update(String stockName, double stockPrice);

}

//Concrete Subject

class StockMarket implements Stock {

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

private String stockName;

private double stockPrice;

public void setStock(String stockName, double stockPrice) {

this.stockName = stockName;

this.stockPrice = stockPrice;

System.*out*.println("\nStock Updated: " + stockName + " = ₹" + stockPrice);

notifyObservers();

}

public void registerObserver(Observer observer) {

observers.add(observer);

System.*out*.println("Observer registered: " + observer.getClass().getSimpleName());

}

public void removeObserver(Observer observer) {

observers.remove(observer);

System.*out*.println("Observer removed: " + observer.getClass().getSimpleName());

}

public void notifyObservers() {

for (Observer observer : observers) {

observer.update(stockName, stockPrice);

}

}

}

//Concrete Observer - Mobile App

class MobileApp implements Observer {

public void update(String stockName, double stockPrice) {

System.*out*.println("Mobile App → " + stockName + ": ₹" + stockPrice);

}

}

//Concrete Observer - Web App

class WebApp implements Observer {

public void update(String stockName, double stockPrice) {

System.*out*.println("Web App → " + stockName + ": ₹" + stockPrice);

}

}

//Test Class

public class ObserverPatternExample {

public static void main(String[] args) {

// Create stock market (subject)

StockMarket stockMarket = new StockMarket();

// Create observers

Observer mobileApp = new MobileApp();

Observer webApp = new WebApp();

// Register observers

stockMarket.registerObserver(mobileApp);

stockMarket.registerObserver(webApp);

// Simulate stock price changes

stockMarket.setStock("CTS 1", 3450.75);

stockMarket.setStock("CTS 2", 1610.20);

// Remove one observer

stockMarket.removeObserver(webApp);

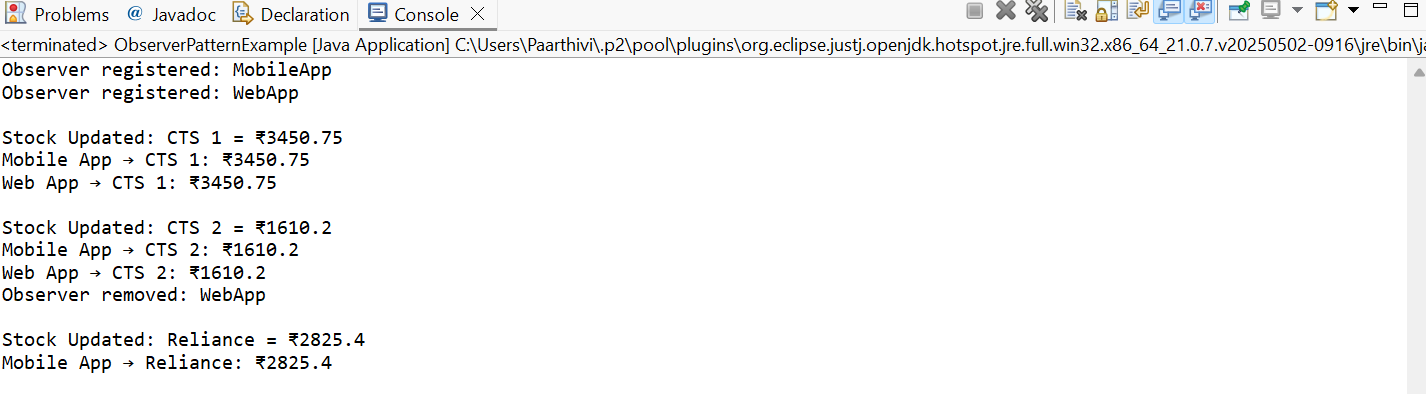
// Update again

stockMarket.setStock("Reliance", 2825.40);

}

}

**Output Screenshot:**



**Exercise 8: Implementing the Strategy Pattern**

**Code:**

package strategyPatternExample;

public class StrategyPatternExample {

// Strategy Interface

interface PaymentStrategy {

void pay(double amount);

}

// Concrete Strategy - Credit Card

static class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

public CreditCardPayment(String cardNumber) {

this.cardNumber = cardNumber;

}

@Override

public void pay(double amount) {

System.out.println("Paid ₹" + amount + " using Credit Card ending in " +

cardNumber.substring(cardNumber.length() - 4));

}

}

// Concrete Strategy - PayPal

static 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);

}

}

// Context Class

static class PaymentContext {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.paymentStrategy = strategy;

}

public void executePayment(double amount) {

if (paymentStrategy == null) {

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

} else {

paymentStrategy.pay(amount);

}

}

}

// Test Class

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

System.out.println("Selecting Credit Card Payment:");

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

context.executePayment(2500.00);

System.out.println("\nSwitching to PayPal Payment:");

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

context.executePayment(4500.75);

System.out.println("\nAttempting payment without setting a strategy:");

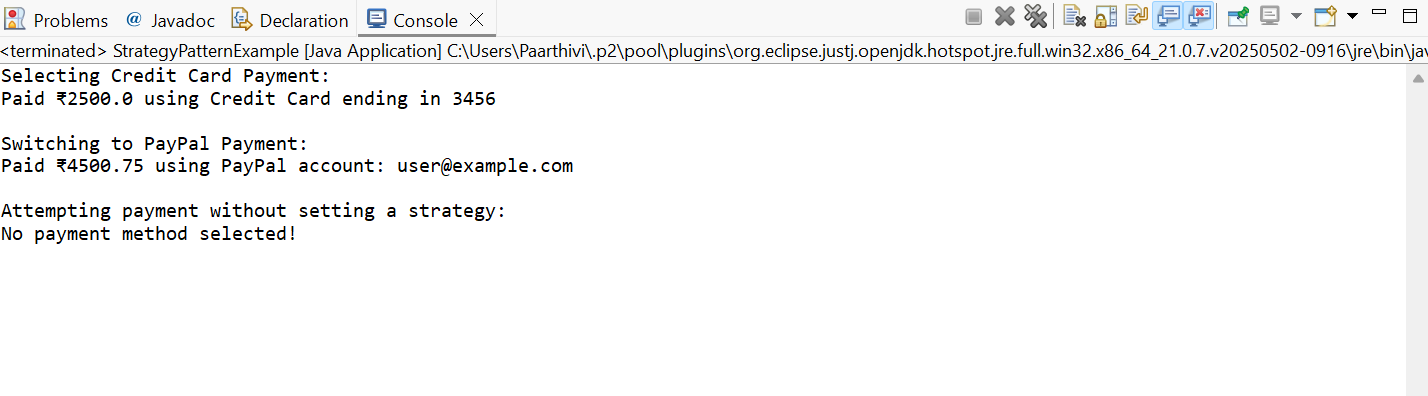
context.setPaymentStrategy(null);

context.executePayment(1000.00);

}

}

**Output Screenshot:**



**Exercise 9: Implementing the Command Pattern**

**Code:**

package commandPatternExample;

public class CommandPatternExample {

// Command Interface

interface Command {

void execute();

}

// Receiver Class

static class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

// Concrete Command - Turn Light On

static class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

// Concrete Command - Turn Light Off

static class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

// Invoker Class

static 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!");

}

}

}

// Test the Command Pattern

public static void main(String[] args) {

// Create receiver

Light livingRoomLight = new Light();

// Create commands

Command lightsOn = new LightOnCommand(livingRoomLight);

Command lightsOff = new LightOffCommand(livingRoomLight);

// Create invoker

RemoteControl remote = new RemoteControl();

// Use remote to execute commands

System.*out*.println("Turning ON the light:");

remote.setCommand(lightsOn);

remote.pressButton();

System.*out*.println("\nTurning OFF the light:");

remote.setCommand(lightsOff);

remote.pressButton();

System.*out*.println("\nPressing button with no command:");

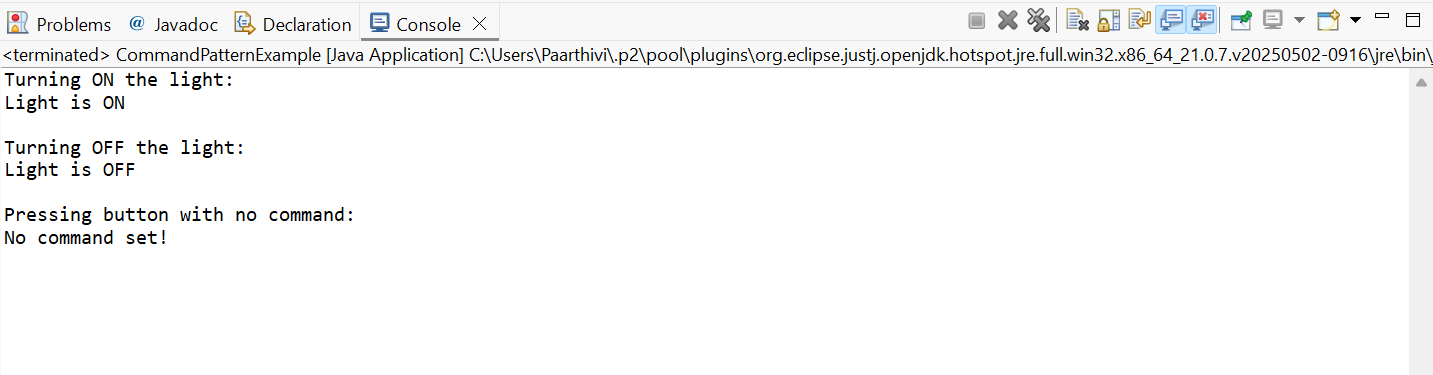
remote.setCommand(null);

remote.pressButton();

}

}

**Output Screenshot:**



**Exercise 10: Implementing the MVC Pattern**

**Code:**

package mvcPatternExample;

public class MVCPatternExample {

// Model Class

static class Student {

private String id;

private String name;

private String 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;

}

}

// View Class

static class StudentView {

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

System.*out*.println("\nStudent Details:");

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

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

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

}

}

// Controller Class

static class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

// Controller methods to update model

public void setStudentName(String name) {

model.setName(name);

}

public void setStudentId(String id) {

model.setId(id);

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

// Getters (optional)

public String getStudentName() {

return model.getName();

}

public String getStudentId() {

return model.getId();

}

public String getStudentGrade() {

return model.getGrade();

}

// Update the view

public void updateView() {

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

}

}

// Step 5: Test Class

public static void main(String[] args) {

// Create model, view, controller

Student model = new Student();

model.setName("Aarav");

model.setId("S123");

model.setGrade("A");

StudentView view = new StudentView();

StudentController controller = new StudentController(model, view);

// Initial view

controller.updateView();

// Update model via controller

System.*out*.println("\nUpdating student details...\n");

controller.setStudentName("Paarthivi");

controller.setStudentGrade("A+");

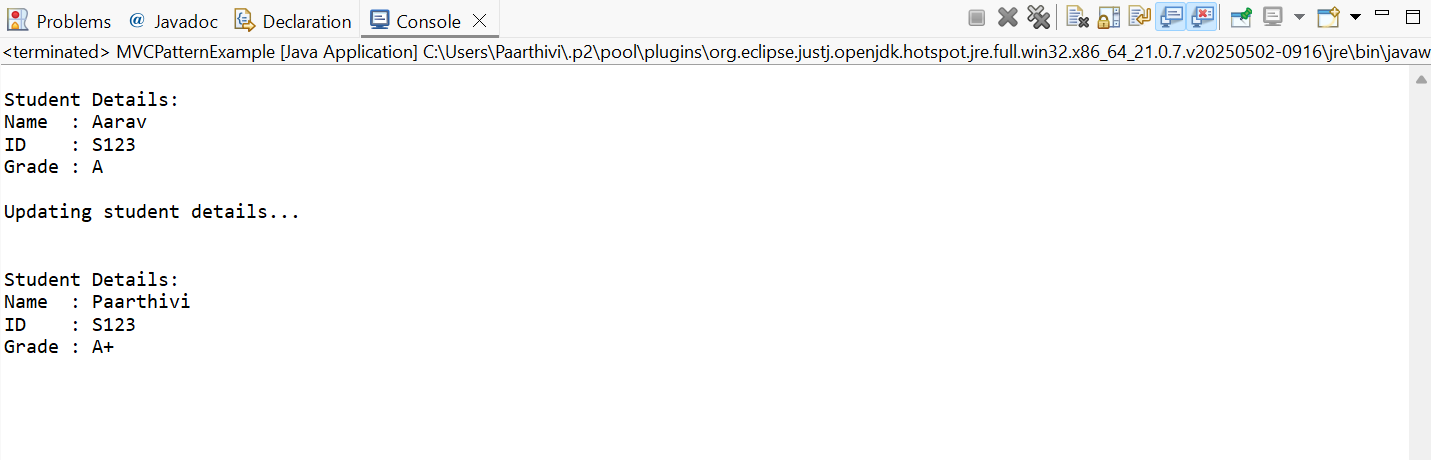
// Updated view

controller.updateView();

}

}

**Output Screenshot:**

****

**Exercise 11: Implementing Dependency Injection**

**Code:**

package dependencyInjectionExample;

public class DependencyInjectionExample {

//Repository Interface

interface CustomerRepository {

Customer findCustomerById(String customerId);

}

// Concrete Repository Implementation

static class CustomerRepositoryImpl implements CustomerRepository {

public Customer findCustomerById(String customerId) {

// Simulate fetching customer (in real-world, from DB)

return new Customer(customerId, "Paarthivi", "paarthivi@example.com");

}

}

// Supporting class: Customer Model

static class Customer {

private String id;

private String name;

private String email;

public Customer(String id, String name, String email) {

this.id = id;

this.name = name;

this.email = email;

}

public String getId() { return id; }

public String getName() { return name; }

public String getEmail() { return email; }

public void display() {

System.*out*.println("Customer Details:");

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

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

System.*out*.println("Email : " + email);

}

}

// Service Class that uses Repository

static class CustomerService {

private CustomerRepository repository;

// Constructor Injection

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void displayCustomer(String customerId) {

Customer customer = repository.findCustomerById(customerId);

if (customer != null) {

customer.display();

} else {

System.*out*.println("Customer not found.");

}

}

}

// Main method to demonstrate DI

public static void main(String[] args) {

// Inject the repository into the service

CustomerRepository repo = new CustomerRepositoryImpl(); // Dependency

CustomerService service = new CustomerService(repo); // Injected

// Use the service to fetch and display a customer

System.*out*.println("Fetching customer using injected repository...");

service.displayCustomer("C101");

}

}

**Output Screenshot:**

