**Design Principles & Patterns**

**(Codes with Output)**

**Exercise 1: Implementing the Singleton Pattern**

**Logger.java**

public class Logger {

    private static Logger instance;

    private Logger() {

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

    }

    public static Logger getInstance() {

        if (instance == null) {

            instance = new Logger();

        }

        return instance;

    }

    public void log(String message) {

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

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        Logger logger2 = Logger.getInstance();

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

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

        if (logger1 == logger2) {

            System.out.println("Both logger instances are the same.");

        } else {

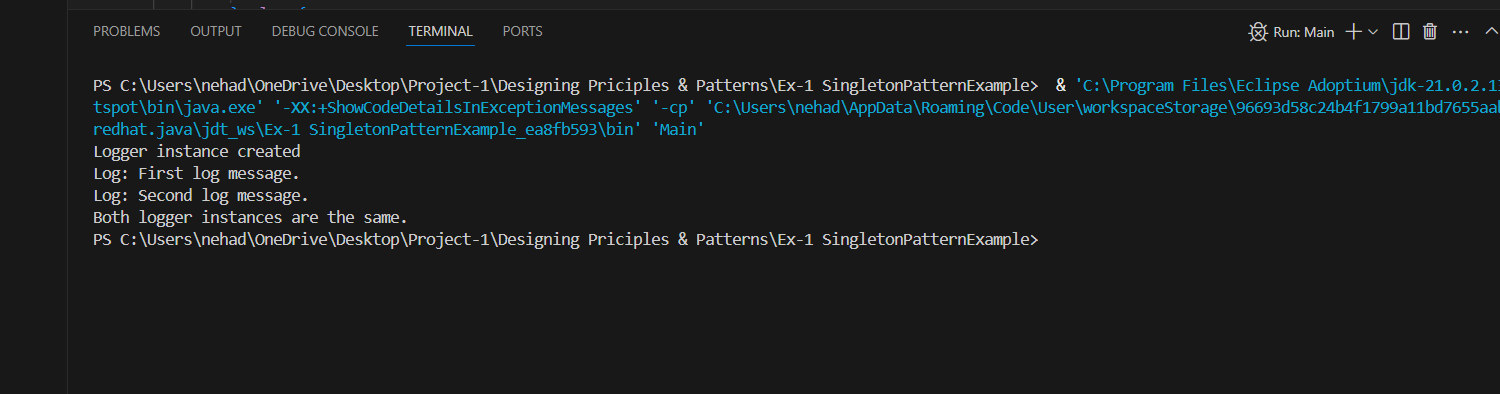
            System.out.println("Different logger instances exist!");

        }

    }

}

**OUTPUT:**



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

**Document.java**

public interface Document {

    void open();

}

**WordDocument.java**

public class WordDocument implements Document {

    public void open() {

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

    }

}

**ExcelDocument.java**

public class ExcelDocument implements Document {

    public void open() {

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

    }

}

**PdfDocument.java**

public class PdfDocument implements Document {

    public void open() {

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

    }

}

**DocumentFactory.java**

public abstract class DocumentFactory {

    public abstract Document createDocument();

}

**WordDocumentFactory.java**

public class WordDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new WordDocument();

    }

}

**ExcelDocumentFactory.java**

public class ExcelDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new ExcelDocument();

    }

}

**PdfDocumentFactory.java**

public class PdfDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new PdfDocument();

    }

}

**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();

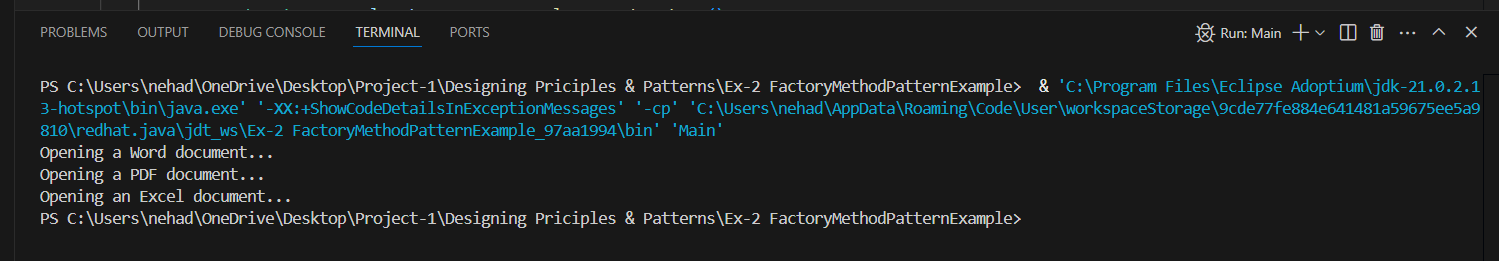
        Document excelDoc = excelFactory.createDocument();

        excelDoc.open();

    }

}

**OUTPUT:**



**Exercise 3: Implementing the Builder Pattern**

**Computer.java**

public class Computer {

    // Required attributes

    private String CPU;

    private String RAM;

    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;

    }

    @Override

    public String toString() {

        return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", Storage=" + storage + ", Graphics Card=" + graphicsCard + "]";

    }

    // Static nested Builder class

    public static class Builder {

        private String CPU;

        private String RAM;

        private String storage;

        private String graphicsCard;

        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 Computer build() {

            return new Computer(this);

        }

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        // Gaming PC

        Computer gamingPC = new Computer.Builder()

                .setCPU("Intel i9")

                .setRAM("32GB")

                .setStorage("1TB SSD")

                .setGraphicsCard("NVIDIA RTX 4080")

                .build();

        // Office PC

        Computer officePC = new Computer.Builder()

                .setCPU("Intel i5")

                .setRAM("8GB")

                .setStorage("512GB SSD")

                .build();

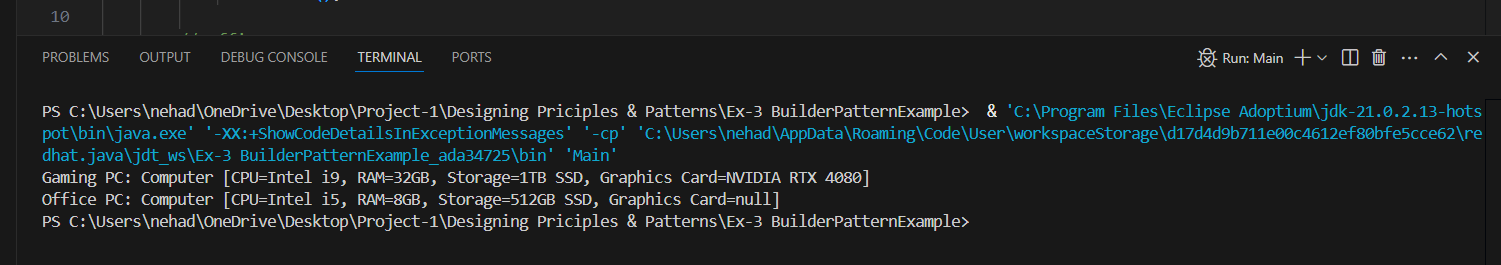
        System.out.println("Gaming PC: " + gamingPC);

        System.out.println("Office PC: " + officePC);

    }

}

**OUTPUT:**



**Exercise 4: Implementing the Adapter Pattern**

**PaymentProcessor.java**

public interface PaymentProcessor {

    void processPayment(double amount);

}

**PayPalGateway.java**

public class PayPalGateway {

    public void sendPayment(double amount) {

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

    }

}

**PayPalAdapter.java**

public class PayPalAdapter implements PaymentProcessor {

    private PayPalGateway payPalGateway;

    public PayPalAdapter(PayPalGateway gateway) {

        this.payPalGateway = gateway;

    }

    @Override

    public void processPayment(double amount) {

        payPalGateway.sendPayment(amount);

    }

}

**StripeGateway.java**

public class StripeGateway {

    public void makePayment(double amount) {

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

    }

}

**StripeAdapter.java**

public class StripeAdapter implements PaymentProcessor {

    private StripeGateway stripeGateway;

    public StripeAdapter(StripeGateway gateway) {

        this.stripeGateway = gateway;

    }

    public void processPayment(double amount) {

        stripeGateway.makePayment(amount);

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

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

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

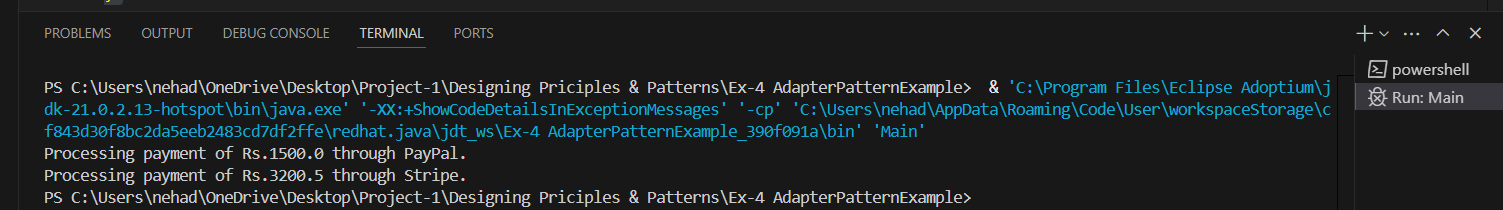
        paypalProcessor.processPayment(1500.00);

        stripeProcessor.processPayment(3200.50);

    }

}

**OUTPUT:**



**Exercise 5: Implementing the Decorator Pattern**

**EmailNotifier.java**

public class EmailNotifier implements Notifier {

    @Override

    public void send(String message) {

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

    }

}

**Notifier.java**

public interface Notifier {

    void send(String message);

}

**NotifierDecorator.java**

public abstract class NotifierDecorator implements Notifier {

    protected Notifier wrappedNotifier;

    public NotifierDecorator(Notifier notifier) {

        this.wrappedNotifier = notifier;

    }

    @Override

    public void send(String message) {

        wrappedNotifier.send(message);

    }

}

**SlackNotifierDecorator.java**

public class SlackNotifierDecorator extends NotifierDecorator {

    public SlackNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String message) {

        super.send(message);

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

    }

}

**SMSNotifierDecorator.java**

public class SMSNotifierDecorator extends NotifierDecorator {

    public SMSNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String message) {

        super.send(message);

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

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        // Base notifier

        Notifier notifier = new EmailNotifier();

        // Add SMS and Slack as decorators

        notifier = new SMSNotifierDecorator(notifier);

        notifier = new SlackNotifierDecorator(notifier);

        // Send message

        notifier.send("System alert: CPU usage is high.");

    }

}

**OUTPUT:**

A screen shot of a computer

AI-generated content may be incorrect.

**Exercise 6: Implementing the Proxy Pattern**

**Image.java**

public interface Image {

    void display();

}

**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); // Lazy initialization

        } else {

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

        }

        realImage.display();

    }

}

**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);

    }

    public void display() {

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

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        Image img1 = new ProxyImage("pic1.jpg");

        // Image will be loaded from remote server

        img1.display();

        System.out.println();

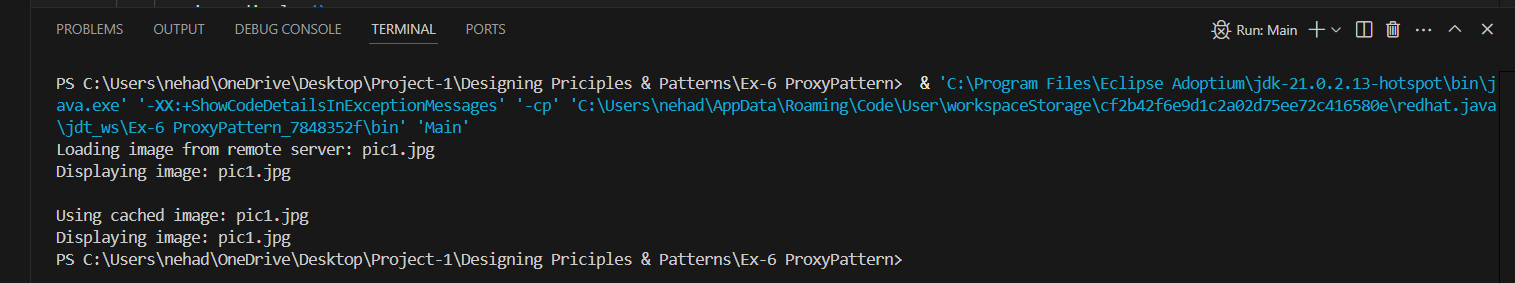
        // Image will be loaded from cache

        img1.display();

    }

}

**OUTPUT:**



**Exercise 7: Implementing the Observer Pattern**

**Observer.java**

public interface Observer {

    void update(String stockName, double price);

}

**Stock.java**

public interface Stock {

    void registerObserver(Observer observer);

    void removeObserver(Observer observer);

    void notifyObservers(String stockName, double price);

}

import java.util.ArrayList;

import java.util.List;

**StockManager.java**

public class StockMarket implements Stock {

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

    public void registerObserver(Observer observer) {

        observers.add(observer);

    }

    public void removeObserver(Observer observer) {

        observers.remove(observer);

    }

    public void notifyObservers(String stockName, double price) {

        for (Observer observer : observers) {

            observer.update(stockName, price);

        }

    }

    // Simulate stock price change

    public void setStockPrice(String stockName, double price) {

        System.out.println("Stock update: " + stockName + " -> Rs." + price);

        notifyObservers(stockName, price);

    }

}

**MobileApp.java**

public class MobileApp implements Observer {

    private String name;

    public MobileApp(String name) {

        this.name = name;

    }

    @Override

    public void update(String stockName, double price) {

        System.out.println(name + " Mobile App - Stock Updated: " + stockName + " at Rs." + price);

    }

}

**WebAPP.java**

public class WebApp implements Observer {

    private String name;

    public WebApp(String name) {

        this.name = name;

    }

    @Override

    public void update(String stockName, double price) {

        System.out.println(name + " Web App - Stock Updated: " + stockName + " at Rs." + price);

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        StockMarket stockMarket = new StockMarket();

        Observer mobileApp = new MobileApp("InvestorPlus");

        Observer webApp = new WebApp("StockWatcher");

        stockMarket.registerObserver(mobileApp);

        stockMarket.registerObserver(webApp);

        stockMarket.setStockPrice("Cognizant", 3450.75);

        stockMarket.setStockPrice("INFY", 1502.30);

    }

}

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 8: Implementing the Strategy Pattern**

**PaymentStrategy.java**

public interface PaymentStrategy {

    void pay(double amount);

}

**CreditCardPayment.java**

public class CreditCardPayment implements PaymentStrategy {

    private String cardNumber;

    public CreditCardPayment(String cardNumber) {

        this.cardNumber = cardNumber;

    }

    @Override

    public void pay(double amount) {

        System.out.println(

                "Paid Rs." + amount + " using Credit Card ending with "

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

    }

}

**PaymentContex.java**

public class PaymentContext {

    private PaymentStrategy strategy;

    public void setPaymentStrategy(PaymentStrategy strategy) {

        this.strategy = strategy;

    }

    public void payAmount(double amount) {

        if (strategy == null) {

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

            return;

        }

        strategy.pay(amount);

    }

}

**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 Rs." + amount + " using PayPal account: " + email);

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();

        // Pay using Credit Card

        context.setPaymentStrategy(new CreditCardPayment("1234567812345678"));

        context.payAmount(2500.00);

        // Pay using PayPal

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

        context.payAmount(4200.50);

    }

}

**OUTPUT:**

A screen shot of a computer

AI-generated content may be incorrect.

**Exercise 9: Implementing the Command Pattern**

**Command.java**

public interface Command {

    void execute();

}

**Light.java**

public class Light {

    public void turnOn() {

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

    }

    public void turnOff() {

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

    }

}

**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();

    }

}

**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.");

        }

    }

}

**Main.java**

public class Main {

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);

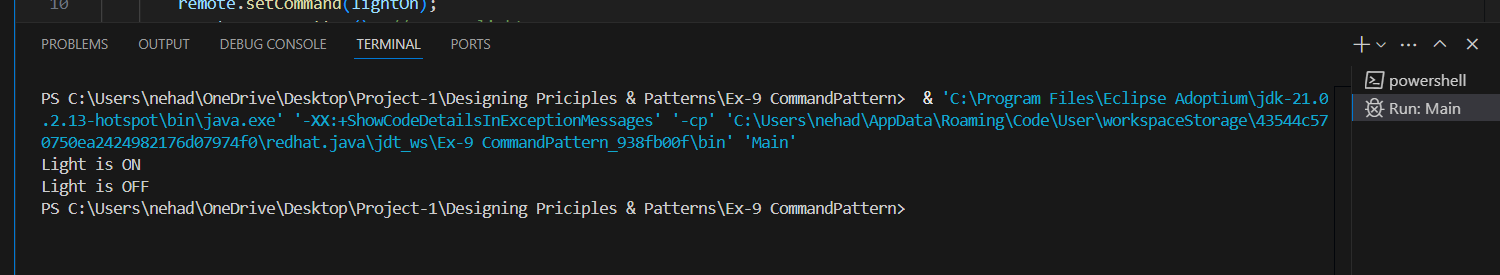
remote.pressButton(); // Turns light ON

remote.setCommand(lightOff);

remote.pressButton(); // Turns light OFF

}

}

**OUTPUT:**

**Exercise 10: Implementing the MVC Pattern**

**Student.java**

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;

}

}

**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);

}

}

**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());

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

// Create Model

Student student = new Student();

student.setName("Neha Dasari");

student.setId("S101");

student.setGrade("A");

// Create View

StudentView view = new StudentView();

// Create Controller

StudentController controller = new StudentController(student, view);

// Display original data

controller.updateView();

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

// Update model data

controller.setStudentName("Neha D.");

controller.setStudentGrade("A+");

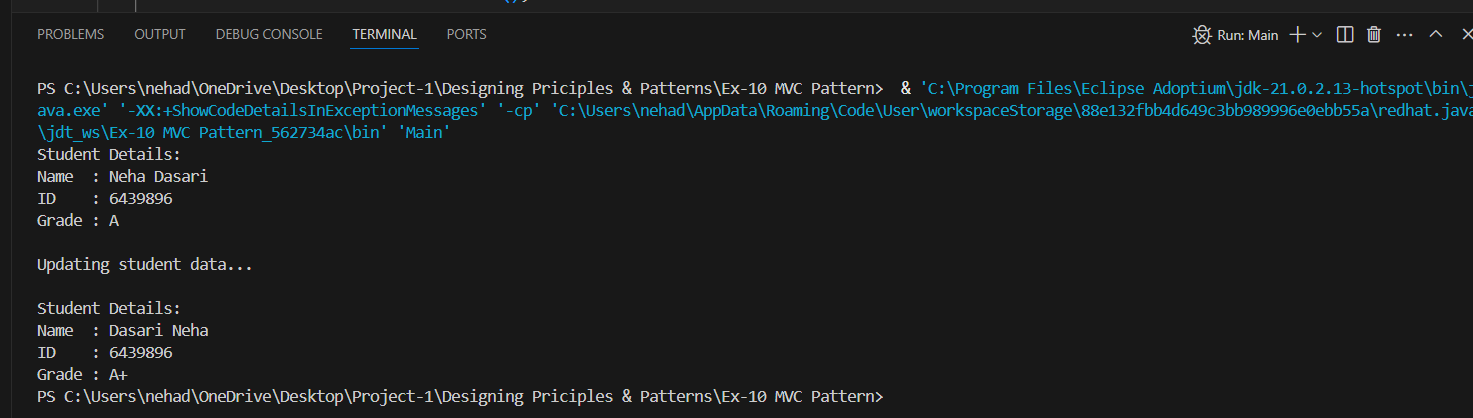
// Display updated data

controller.updateView();

}

}

**OUTPUT:**



**Exercise 11: Implementing Dependency Injection**

**Customer.java**

public class Customer {

private String id;

private String name;

public Customer(String id, String name) {

this.id = id;

this.name = name;

}

public String getId() {

return id;

}

public String getName() {

return name;

}

}

**CustomerRepository.java**

public interface CustomerRepository {

Customer findCustomerById(String id);

}

**CustomerRepositoryImpl.java**

public class CustomerRepositoryImpl implements CustomerRepository {

@Override

public Customer findCustomerById(String id) {

// Simulate fetching customer from a database

System.out.println("Fetching customer from repository...");

return new Customer(id, "Neha Dasari");

}

}

**CustomerService.java**

public class CustomerService {

private CustomerRepository customerRepository;

// Constructor injection

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void getCustomerDetails(String id) {

Customer customer = customerRepository.findCustomerById(id);

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

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

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

// Create repository object

CustomerRepository repository = new CustomerRepositoryImpl();

// Inject repository into service

CustomerService service = new CustomerService(repository);

// Use service to get customer details

service.getCustomerDetails("C101");

}

}

**OUTPUT:**

A screen shot of a computer

AI-generated content may be incorrect.