**Student Name :- Dharan Gowda H E Track:-Java FSE**

**Exercise 1: Implementing the Singleton Pattern**

public class SingletonPatternExample{

    public static void main(String[] args) {

        Logger obj1 = Logger.getInstance();

        Logger obj2 = Logger.getInstance();

        System.out.println(obj1==obj2);

    } }

class Logger{

        private static Logger instance;

        private Logger(){

            System.out.println("I am the only instance of the logger class");

        }

        public static Logger getInstance(){

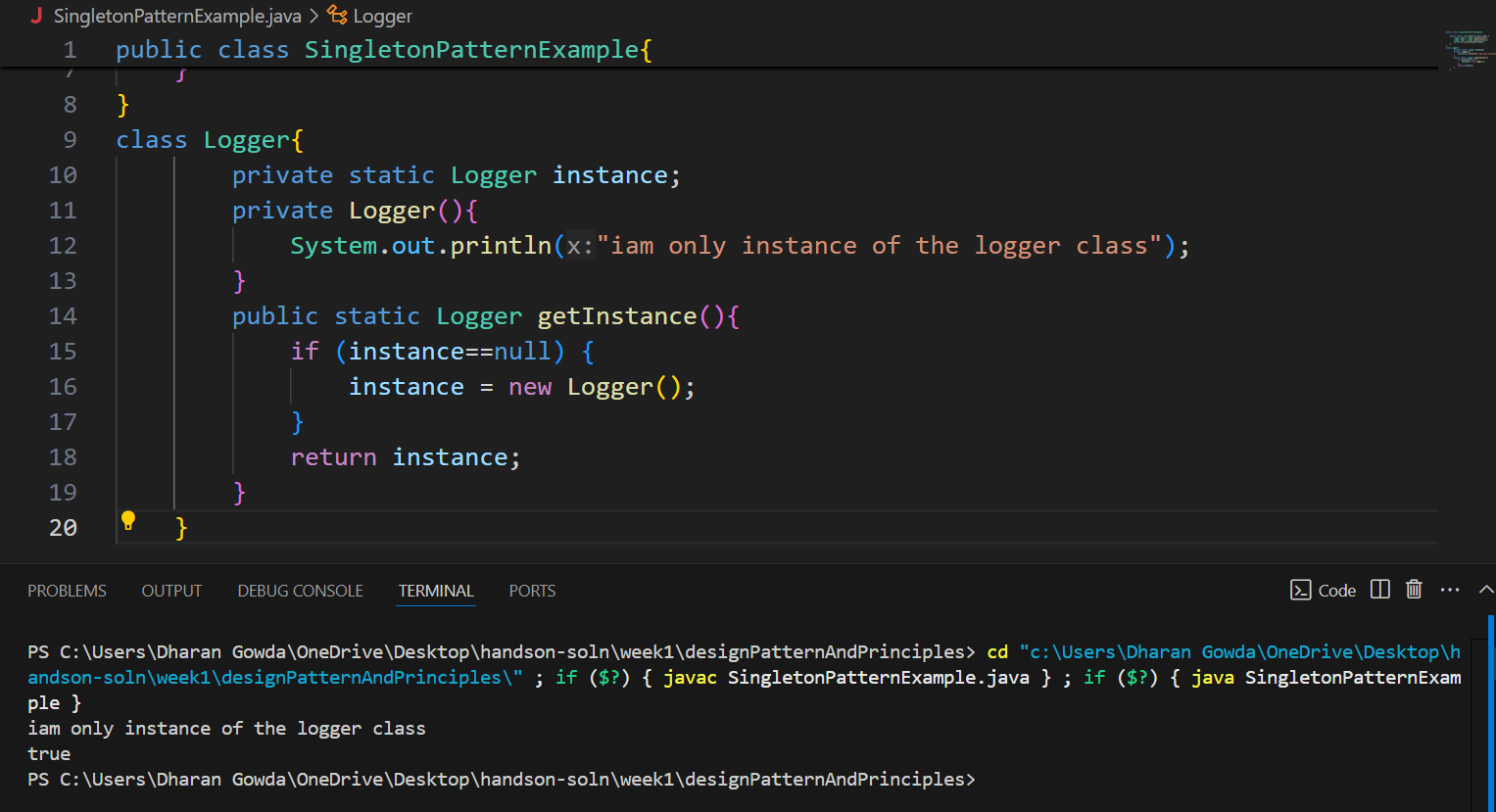
            if (instance==null) {

                instance = new Logger();

            }

            return instance}

}     **OUTPUT:-**



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

public class FactoryMethodPatternExample {

  public static void main(String[] args) {

    DocumentFactory factory = new DocumentFactory();

    Document doc1  = factory.createDocument("pdf");

    if (doc1 != null) doc1.use();

    Document doc2 = factory.createDocument("word");

    if (doc2 != null) doc2.use();

    Document doc3 = factory.createDocument("excel");

    if (doc3 != null) doc3.use();

  }

}

interface Document {

  void use();

}

class WordDocument implements Document {

  public void use() {

    System.out.println("Use me as a Word document");

  }

}

class PdfDocument implements Document {

  public void use() {

    System.out.println("Use me as a PDF document");

  }

}

class ExcelDocument implements Document {

  public void use() {

    System.out.println("Use me as an Excel document");

  }

}

class DocumentFactory {

  public Document createDocument(String str) {

    if ("excel".equalsIgnoreCase(str)) {

      return new ExcelDocument();

    } else if ("word".equalsIgnoreCase(str)) {

      return new WordDocument();

    } else if ("pdf".equalsIgnoreCase(str)) {

      return new PdfDocument();

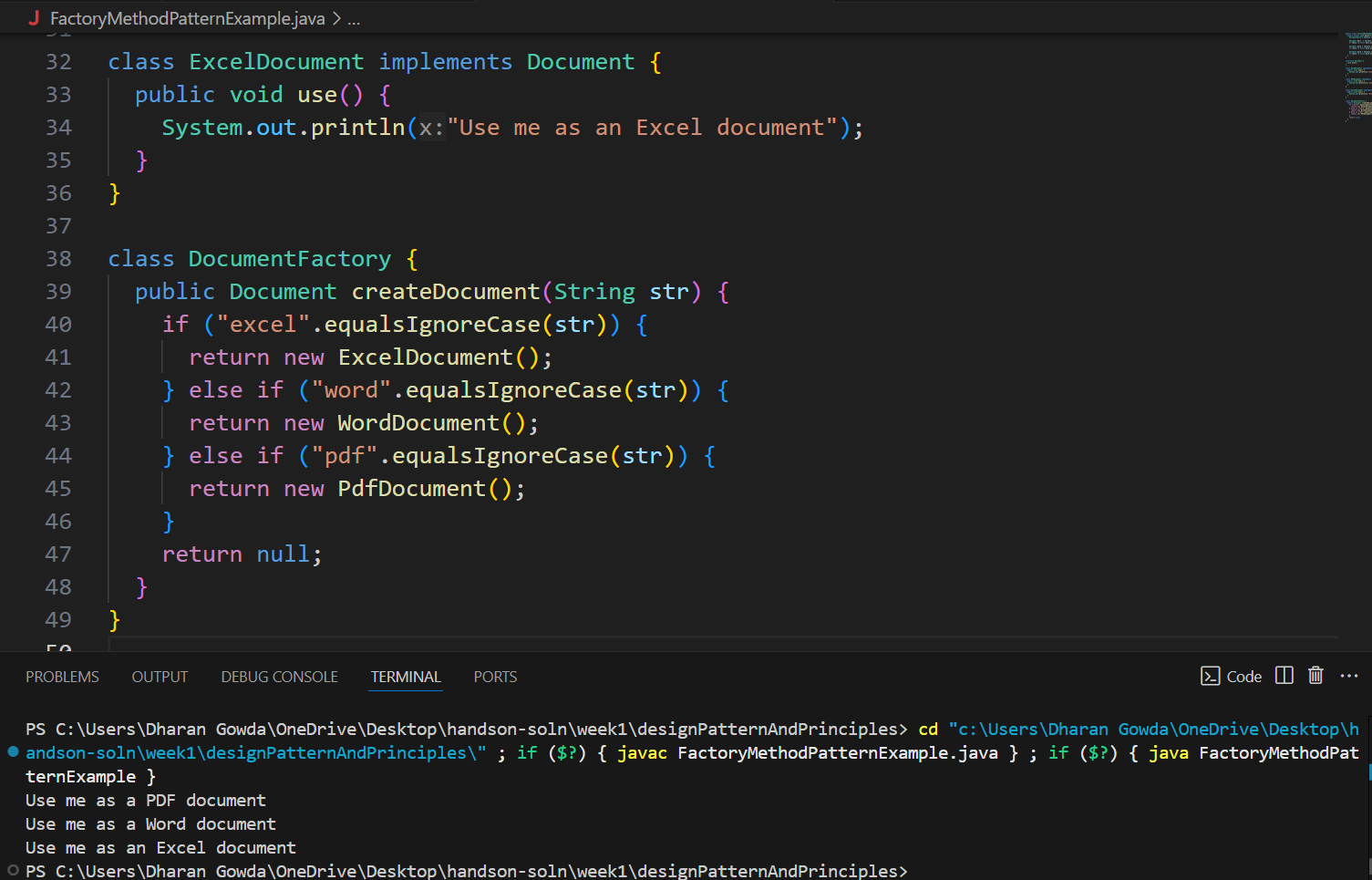
    }

    return null;

  }

}

**OUTPUT:-**

****

**Exercise 3: Implementing the Builder Pattern**

public class BuilderPattern {

    public static void main(String[] args) {

        Computer basicComputer = new Computer.Builder("Intel i3", "4GB")

                                    .storage("256GB SSD")

                                    .build();

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

                                     .storage("1TB SSD")

                                     .graphicsCard("NVIDIA RTX 4090")

                                     .wifiCard("Intel AX210")

                                     .bluetooth("v5.3")

                                     .build();

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

                                    .storage("512GB SSD")

                                    .wifiCard("Intel AX200")

                                    .build();

        System.out.println("Basic: " + basicComputer);

        System.out.println("Gaming: " + gamingComputer);

        System.out.println("Office: " + officeComputer);

    }

}

class Computer {

    private final String cpu;

    private final String ram;

    private final String storage;

    private final String graphicsCard;

    private final String wifiCard;

    private final String bluetooth;

    private Computer(Builder builder) {

        this.cpu = builder.cpu;

        this.ram = builder.ram;

        this.storage = builder.storage;

        this.graphicsCard = builder.graphicsCard;

        this.wifiCard = builder.wifiCard;

        this.bluetooth = builder.bluetooth;

    }

    public static class Builder {

        private final String cpu;

        private final String ram;

        private String storage;

        private String graphicsCard;

        private String wifiCard;

        private String bluetooth;

        public Builder(String cpu, String ram) {

            this.cpu = cpu;

            this.ram = ram;

        }

        public Builder storage(String storage) {

            this.storage = storage;

            return this;

        }

        public Builder graphicsCard(String graphicsCard) {

            this.graphicsCard = graphicsCard;

            return this;

        }

        public Builder wifiCard(String wifiCard) {

            this.wifiCard = wifiCard;

            return this;

        }

        public Builder bluetooth(String bluetooth) {

            this.bluetooth = bluetooth;

            return this;

        }

        public Computer build() {

            return new Computer(this);

        }

    }

    @Override

    public String toString() {

        return "Computer [CPU=" + cpu + ", RAM=" + ram + ", Storage=" + storage +

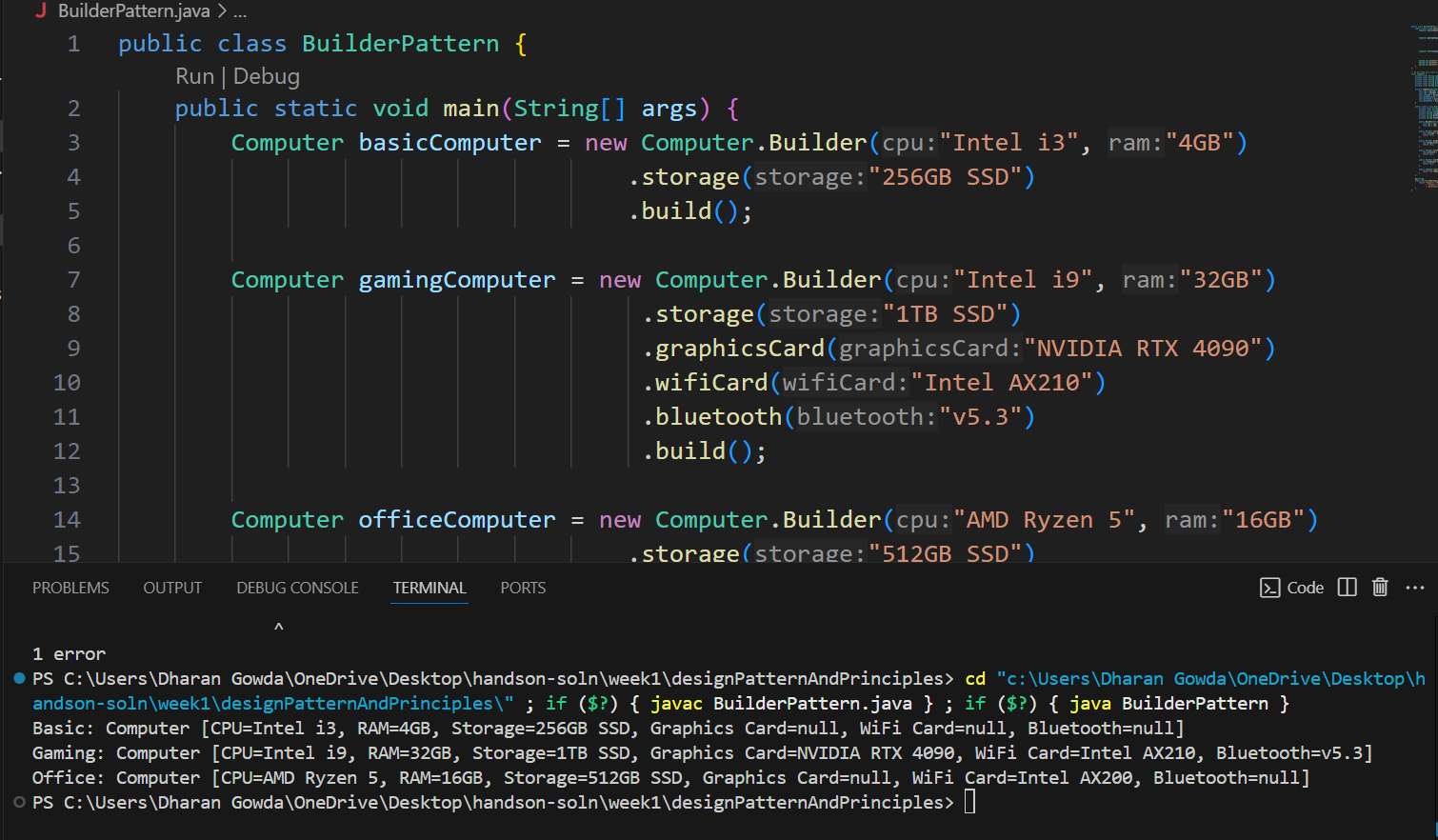
               ", Graphics Card=" + graphicsCard + ", WiFi Card=" + wifiCard +

               ", Bluetooth=" + bluetooth + "]";

    }

}

**OUTPUT:-**

****

**Exercise 4: Implementing the Adapter Pattern**

interface PaymentProcessor {

    void processPayment(double amount);

}

class PaypalGateway {

    public void makePaypalPayment(String userEmail, double amount) {

        System.out.println("Paid ₹" + amount + " using PayPal for user: " + userEmail);

    }

}

class StripeGateway {

    public void payViaStripe(double amountInRupees) {

        System.out.println("Stripe processed ₹" + amountInRupees);

    }

}

class PaypalAdapter implements PaymentProcessor {

    private PaypalGateway paypal;

    private String userEmail;

    public PaypalAdapter(String userEmail) {

        this.paypal = new PaypalGateway();

        this.userEmail = userEmail;

    }

    @Override

    public void processPayment(double amount) {

        paypal.makePaypalPayment(userEmail, amount);

    }

}

class StripeAdapter implements PaymentProcessor {

    private StripeGateway stripe;

    public StripeAdapter() {

        this.stripe = new StripeGateway();

    }

    @Override

    public void processPayment(double amount) {

        stripe.payViaStripe(amount);

}

}

// Main Class

public class AdapterPattern {

    public static void main(String[] args) {

        PaymentProcessor paypalPayment = new PaypalAdapter("user@example.com");

        PaymentProcessor stripePayment = new StripeAdapter();

        System.out.println("=== Making Payments ===");

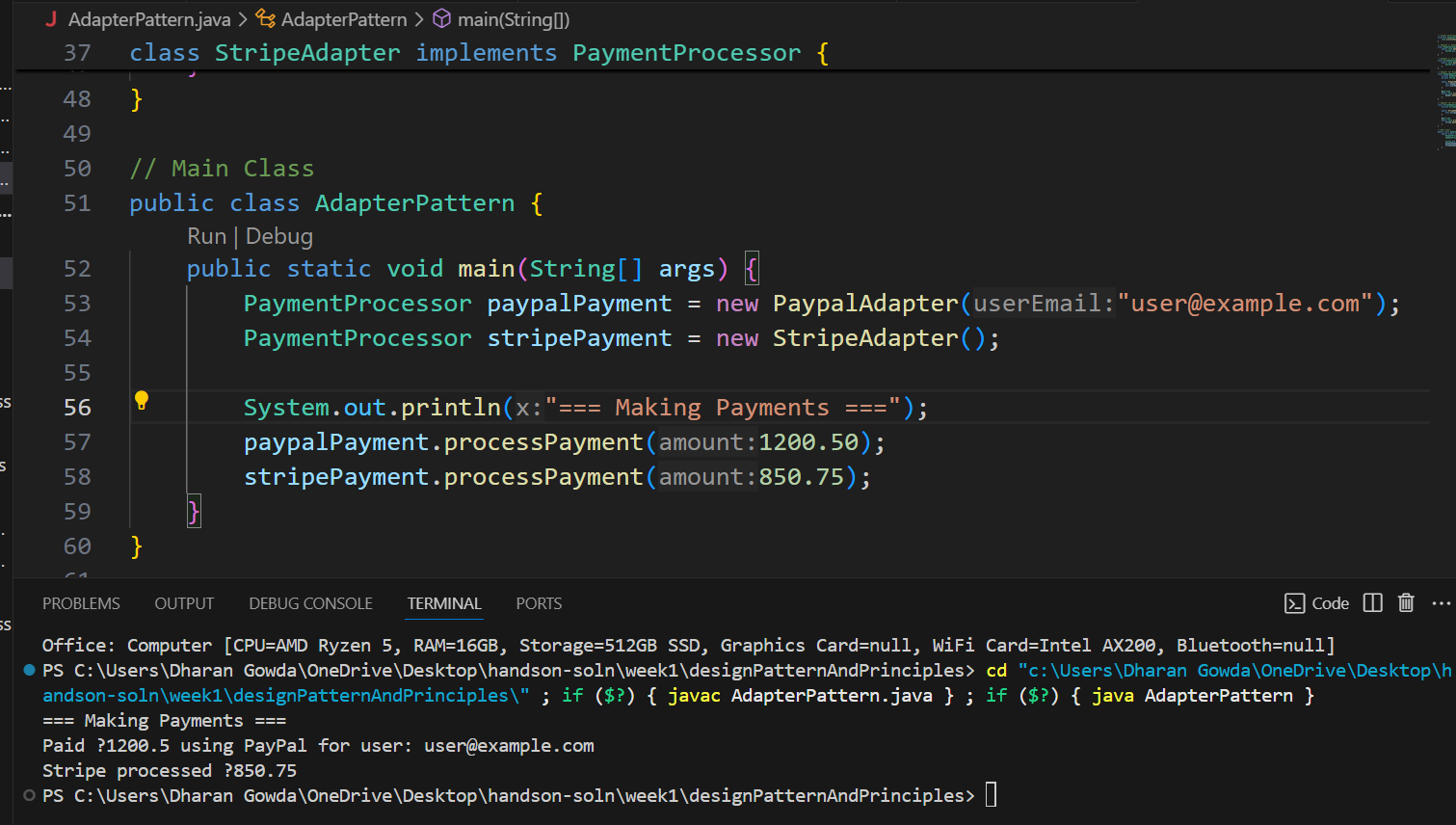
        paypalPayment.processPayment(1200.50);

        stripePayment.processPayment(850.75);

    }

}

**OUTPUT:-**

****

**Exercise 5: Implementing the Decorator Pattern**

interface Notifier {

    void send(String message);

}

class EmailNotifier implements Notifier {

    @Override

    public void send(String message) {

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

    }

}

abstract class NotifierDecorator implements Notifier {

    protected Notifier wrappedNotifier;

    public NotifierDecorator(Notifier notifier) {

        this.wrappedNotifier = notifier;

    }

    @Override

    public void send(String message) {

        wrappedNotifier.send(message);

    }

}

class SMSNotifierDecorator extends NotifierDecorator {

    public SMSNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String message) {

        super.send(message);

        sendSMS(message);

    }

    private void sendSMS(String message) {

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

    }

}

class SlackNotifierDecorator extends NotifierDecorator {

    public SlackNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String message) {

        super.send(message);

        sendSlack(message);

    }

    private void sendSlack(String message) {

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

    }

}

public class DecoratorPattern {

    public static void main(String[] args) {

        Notifier emailNotifier = new EmailNotifier();

        Notifier smsDecorator = new SMSNotifierDecorator(emailNotifier);

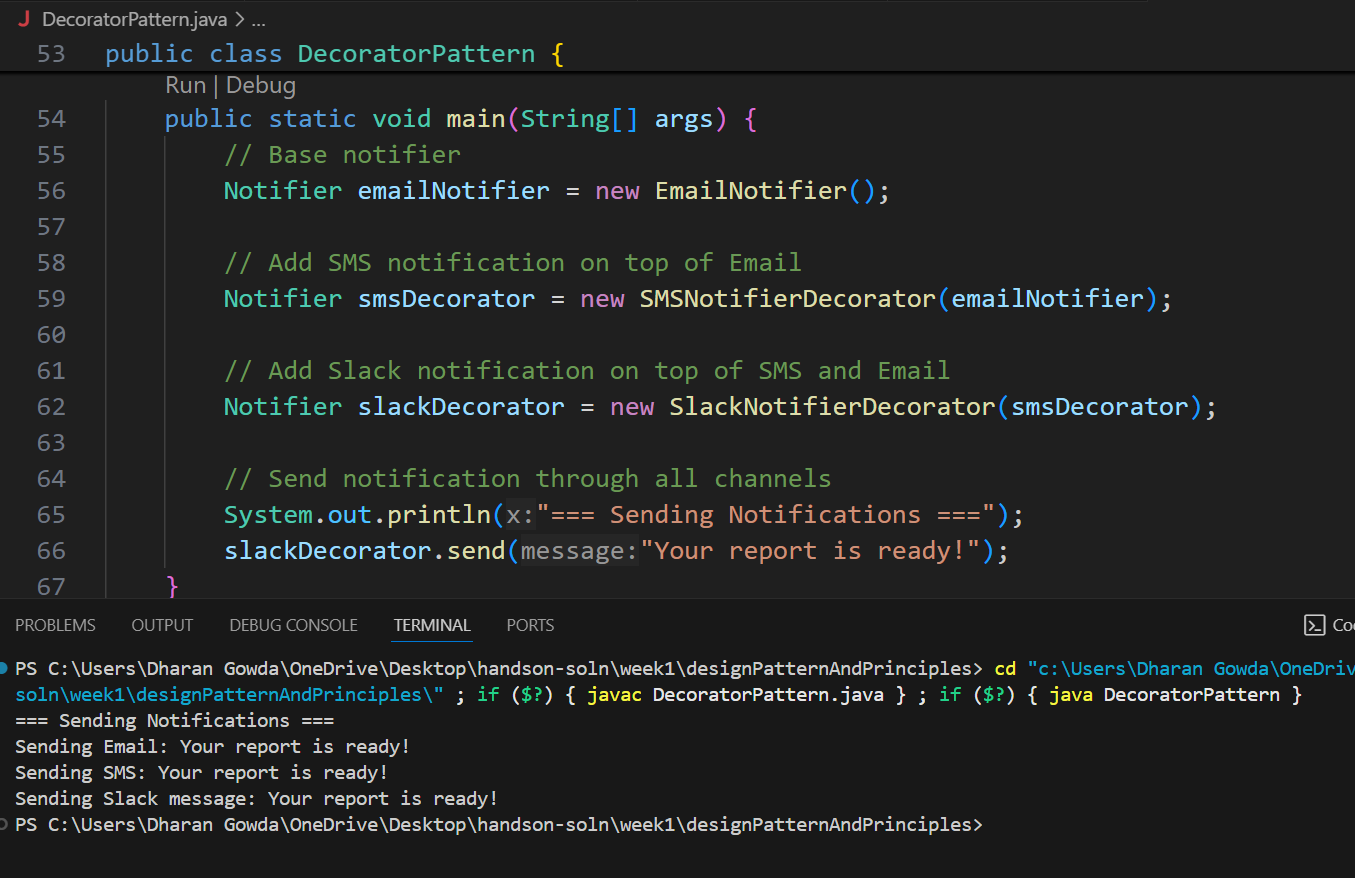
        Notifier slackDecorator = new SlackNotifierDecorator(smsDecorator);

        System.out.println("=== Sending Notifications ===");

        slackDecorator.send("Your report is ready!");

    }

} **OUTPUT:-**

****

**Exercise 6: Implementing the Proxy Pattern**

interface Image {

    void display();

}

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);

    }

    @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);

        } else {

            System.out.println("Loaded from cache: " + filename);

        }

        realImage.display();

    }

}

public class ProxyPattern {

    public static void main(String[] args) {

        Image img1 = new ProxyImage("photo1.jpg");

        Image img2 = new ProxyImage("photo2.jpg");

        img1.display();

        img1.display();

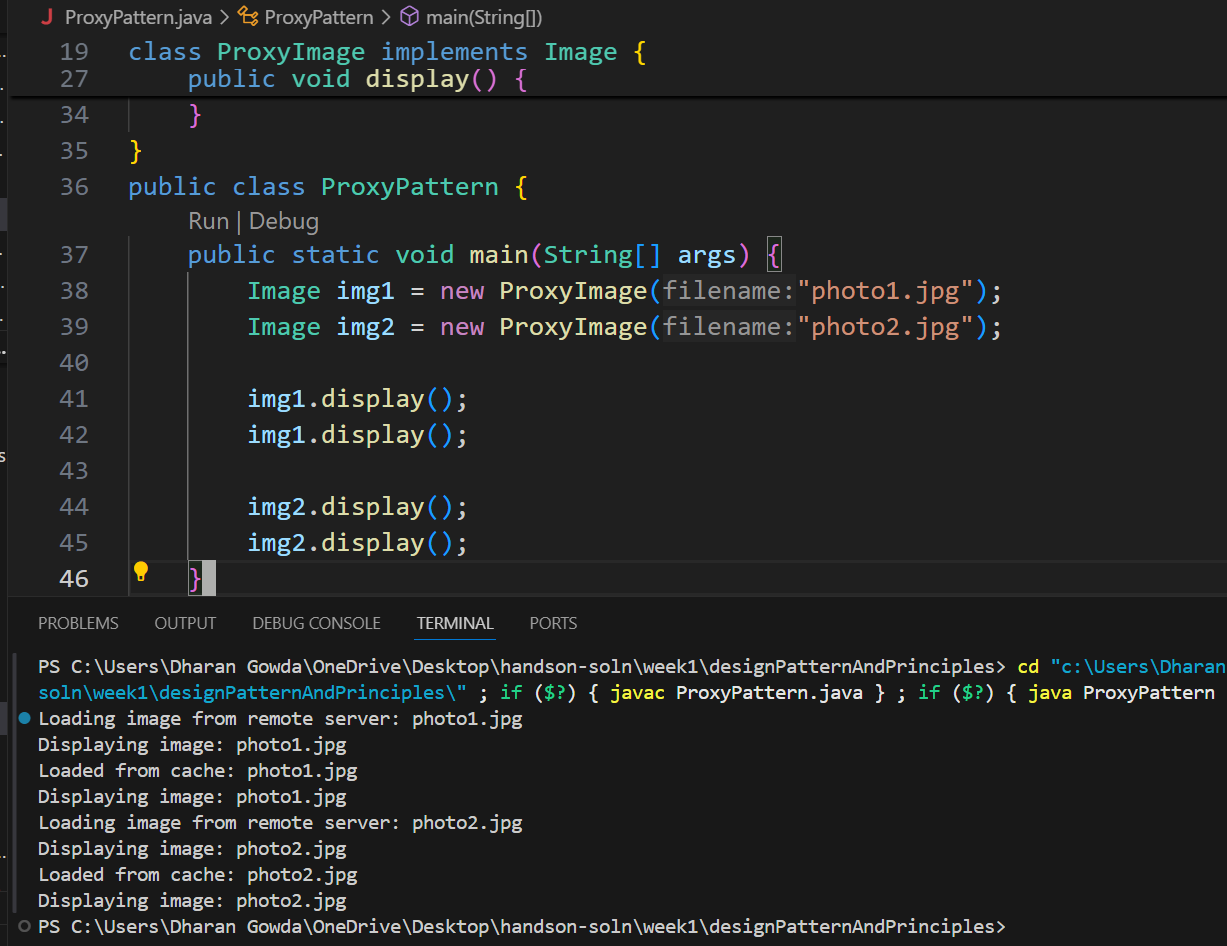
        img2.display();

        img2.display();

    }

}

**OUTPUT:-**

****

**Exercise 7: Implementing the Observer Pattern**

import java.util.\*;

interface Observer {

void update(String stock, double price);

}

interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers(String stock, double price);

}

class StockMarket implements Stock {

private List<Observer> 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(String stock, double price) {

for (Observer o : observers) {

o.update(stock, price);

}

}

public void setStockPrice(String stock, double price) {

System.out.println("\nStock Updated: " + stock + " - ₹" + price);

notifyObservers(stock, price);

}

}

class MobileApp implements Observer {

private String name;

public MobileApp(String name) {

this.name = name;

}

@Override

public void update(String stock, double price) {

System.out.println(name + " [MobileApp] - Stock Update: " + stock + " ₹" + price);

}

}

class WebApp implements Observer {

private String name;

public WebApp(String name) {

this.name = name;

}

@Override

public void update(String stock, double price) {

System.out.println(name + " [WebApp] - Stock Update: " + stock + " ₹" + price);

}

}

public class ObserverPattern {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobileClient = new MobileApp("Client A");

Observer webClient = new WebApp("Client B");

market.registerObserver(mobileClient);

market.registerObserver(webClient);

market.setStockPrice("TCS", 3500.75);

market.setStockPrice("Infosys", 1620.50);

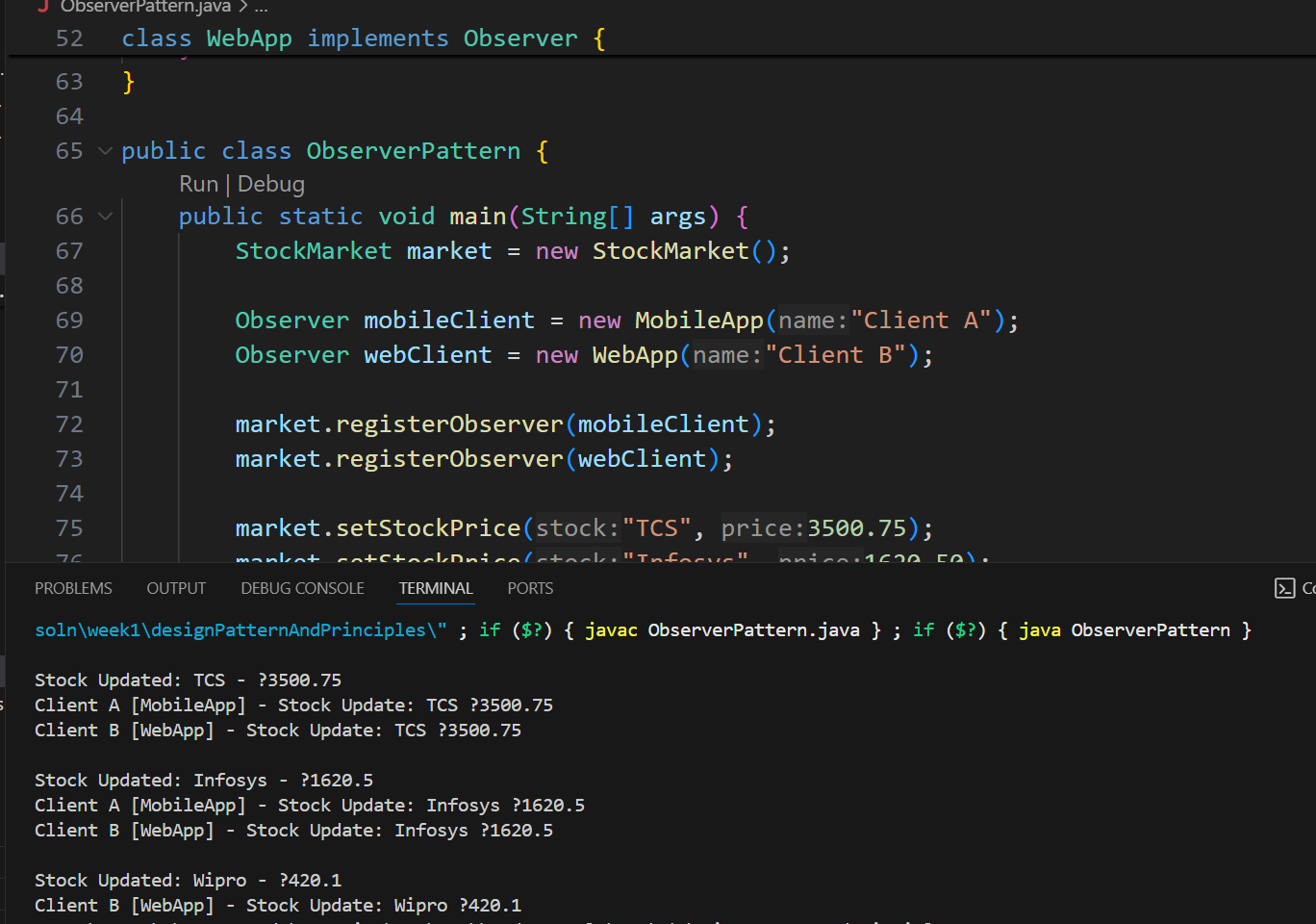
market.removeObserver(mobileClient);

market.setStockPrice("Wipro", 420.10);

}

}

**OUTPUT:-**

****

**Exercise 8: Implementing the Strategy Pattern**

interface PaymentStrategy {

void pay(double amount);

}

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: " + cardNumber);

}

}

class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

@Override

public void pay(double amount) {

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

}

}

class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void processPayment(double amount) {

if (strategy == null) {

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

} else {

strategy.pay(amount);

}

}

}

public class StrategyPattern {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

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

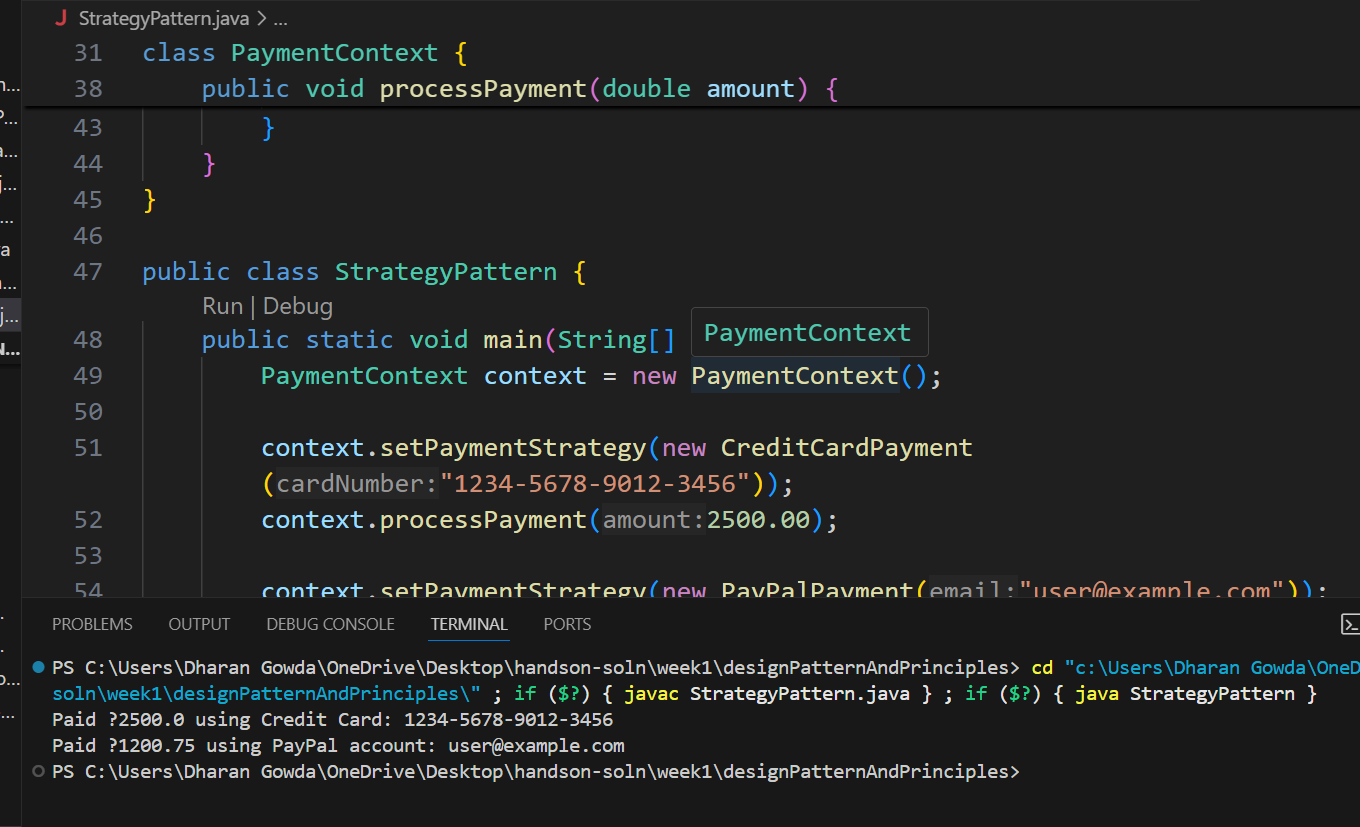
context.processPayment(2500.00);

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

context.processPayment(1200.75);

} **OUTPUT:-**

}

****

**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;

    }

    @Override

    public void execute() {

        light.turnOn();

    }

}

class LightOffCommand implements Command {

    private Light light;

    public LightOffCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOff();

    }

}

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");

        }

    }

}

public class CommandPattern {

    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();

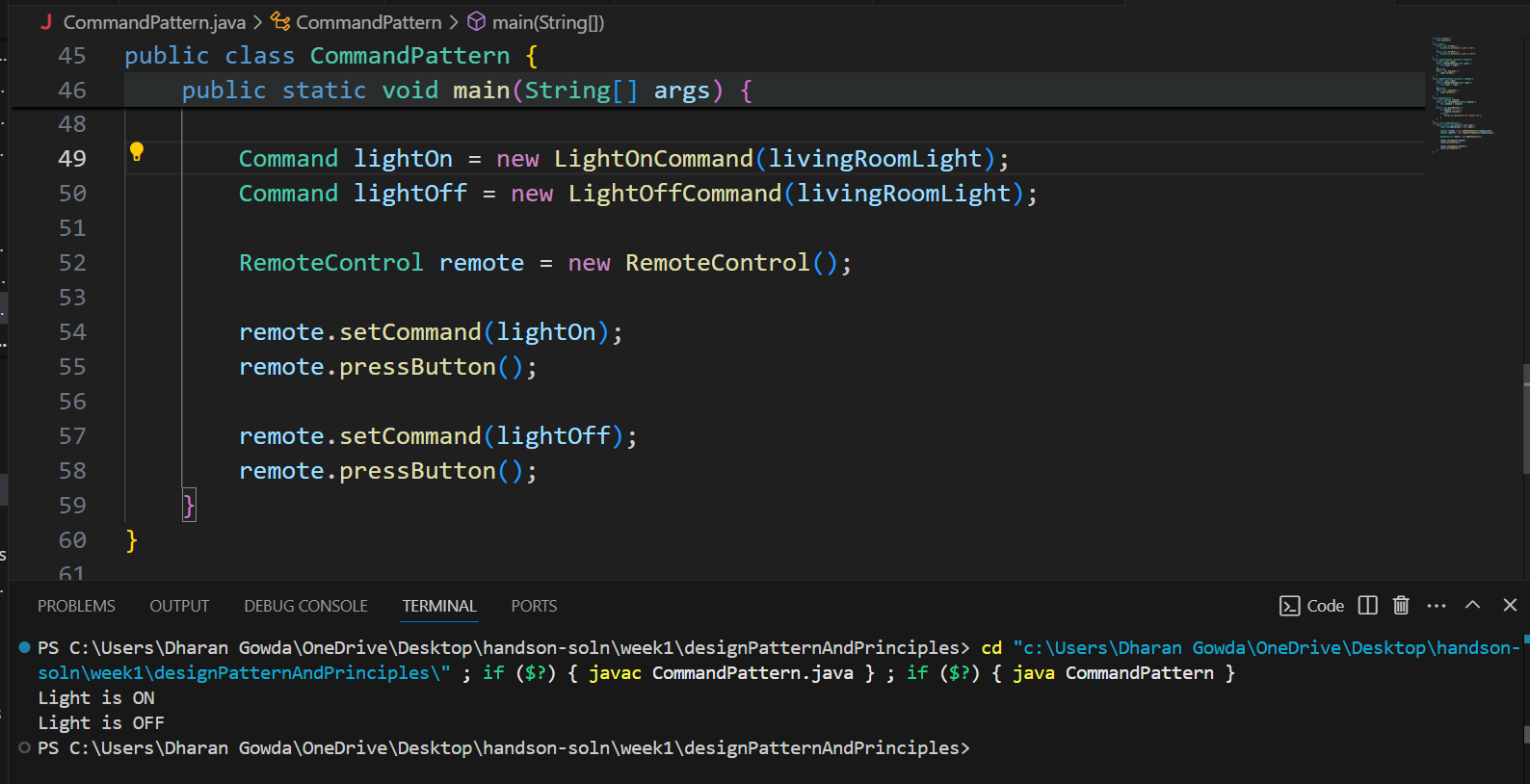
        remote.setCommand(lightOff);

        remote.pressButton();

    }

}

**OUTPUT:-**

****

**Exercise 10: Implementing the MVC Pattern**

class Student {

private String name;

private String id;

private String grade;

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

this.name = name;

this.id = id;

this.grade = 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;

}

}

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);

}

}

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 void updateView() {

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

}

}

public class MvcPattern {

public static void main(String[] args) {

Student model = new Student("Ravi Kumar", "ST101", "A");

StudentView view = new StudentView();

StudentController controller = new StudentController(model, view);

controller.updateView();

controller.setStudentName("Ravi K.");

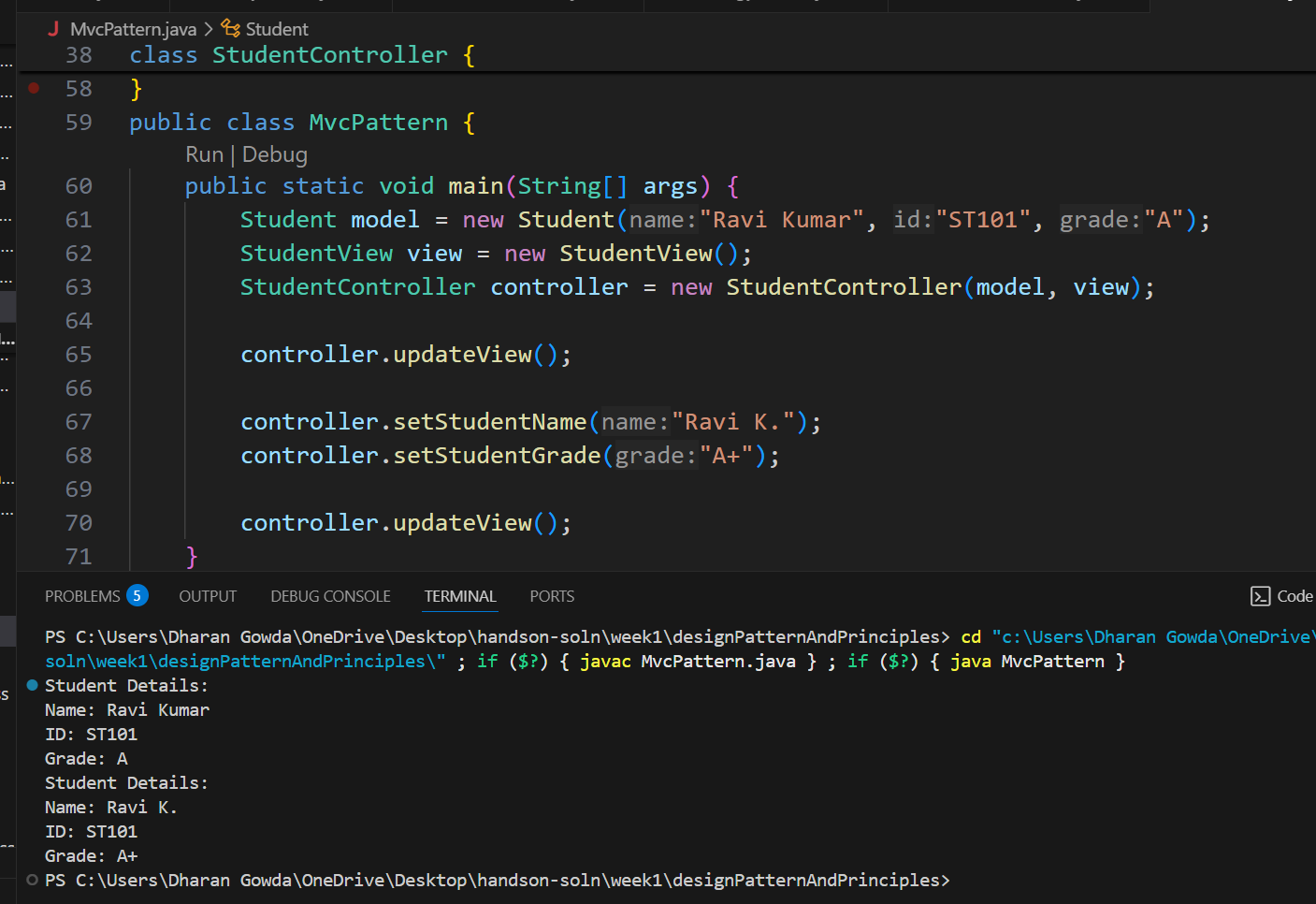
controller.setStudentGrade("A+");

controller.updateView();

}

}

**OUTPUT:-**

****

**Exercise 11: Implementing Dependency Injection**

interface CustomerRepository {

    String findCustomerById(String customerId);

}

class CustomerRepositoryImpl implements CustomerRepository {

    @Override

    public String findCustomerById(String customerId) {

        return "Customer[ID=" + customerId + ", Name=Dharan Gowda ]";

    }

}

class CustomerService {

    private CustomerRepository repository;

    public CustomerService(CustomerRepository repository) {

        this.repository = repository;

    }

    public void getCustomerDetails(String customerId) {

        String customer = repository.findCustomerById(customerId);

        System.out.println("Retrieved: " + customer);

    }

}

public class DependencyInjection {

    public static void main(String[] args) {

        CustomerRepository repository = new CustomerRepositoryImpl();

        CustomerService service = new CustomerService(repository);

        service.getCustomerDetails("C102");

    }

}

**OUTPUT:-**

****