WEEK1\_DesignPrinciplesAndPatterns\_Hands0n  
  
**Exercise 1: Implementing the Singleton Pattern**

CODE:  
  
class Logger

{

    public Logger()

    {

        System .out.println("Instance created");

    }

    private static class LoggerHelper

    {

        public static final Logger INSTANCE=new Logger();

    }

    public static Logger getInstance()

    {

        return LoggerHelper.INSTANCE;

    }

}

class Main

{

    public static void main(String[] args)

    {

        Logger l1=Logger.getInstance();

        System.out.println("First instance");

        Logger l2=Logger.getInstance();

        System.out.println("Second instance");

        if(l1==l2)

        {

            System.out.println("Singleton satisfied");

        }

        else{

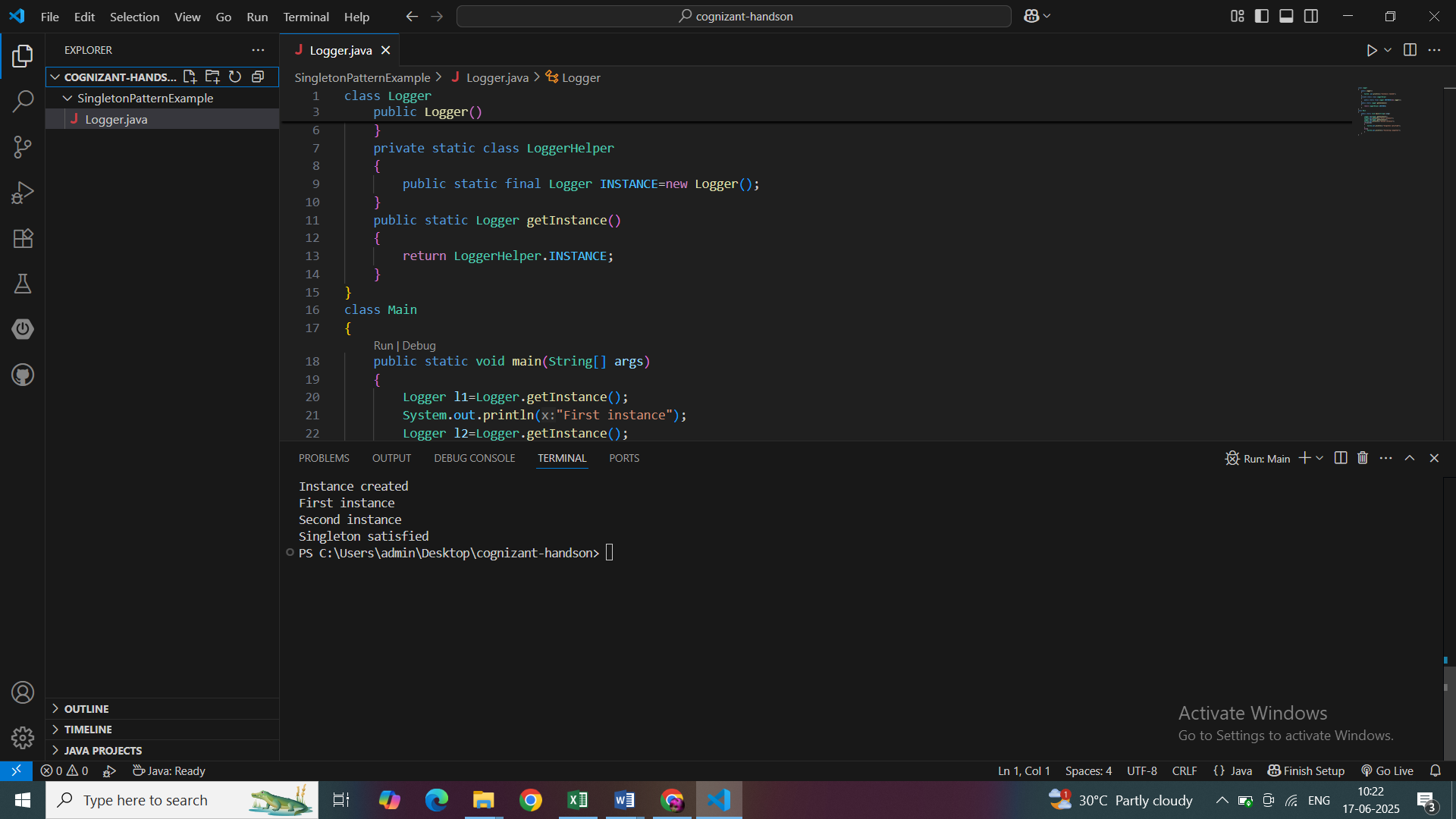
            System.out.println("Violating singleton");

        }

    }

}

OUTPUT:



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

**CODE:**package FactoryMethodPatternExample;

public class Factory {

    public static void main(String[] args)

    {

        DocFactory df=new WordFactory();

        Document d=df.createDoc();

        System.out.println(d.create());

        DocFactory df1=new PdfFactory();

        Document d1=df1.createDoc();

        System.out.println(d1.create());

        DocFactory df2=new ExcelFactory();

        Document d2=df2.createDoc();

        System.out.println(d2.create());

    }

}

interface Document{

    String create();

}

class Word implements Document{

    public String create()

    {

        return "WordDocument";

    }

}

class Pdf implements Document{

    public String create()

    {

        return "PdfDocument";

    }

}

class Excel implements Document{

    public String create()

    {

        return "ExcelDocument";

    }

}

abstract class DocFactory

{

    abstract Document createDoc();

}

class WordFactory extends DocFactory{

    public Document createDoc()

    {

        return new Word();

    }

}

class PdfFactory extends DocFactory{

    public Document createDoc(){

        return new Pdf();

    }

}

class ExcelFactory extends DocFactory{

    public Document createDoc()

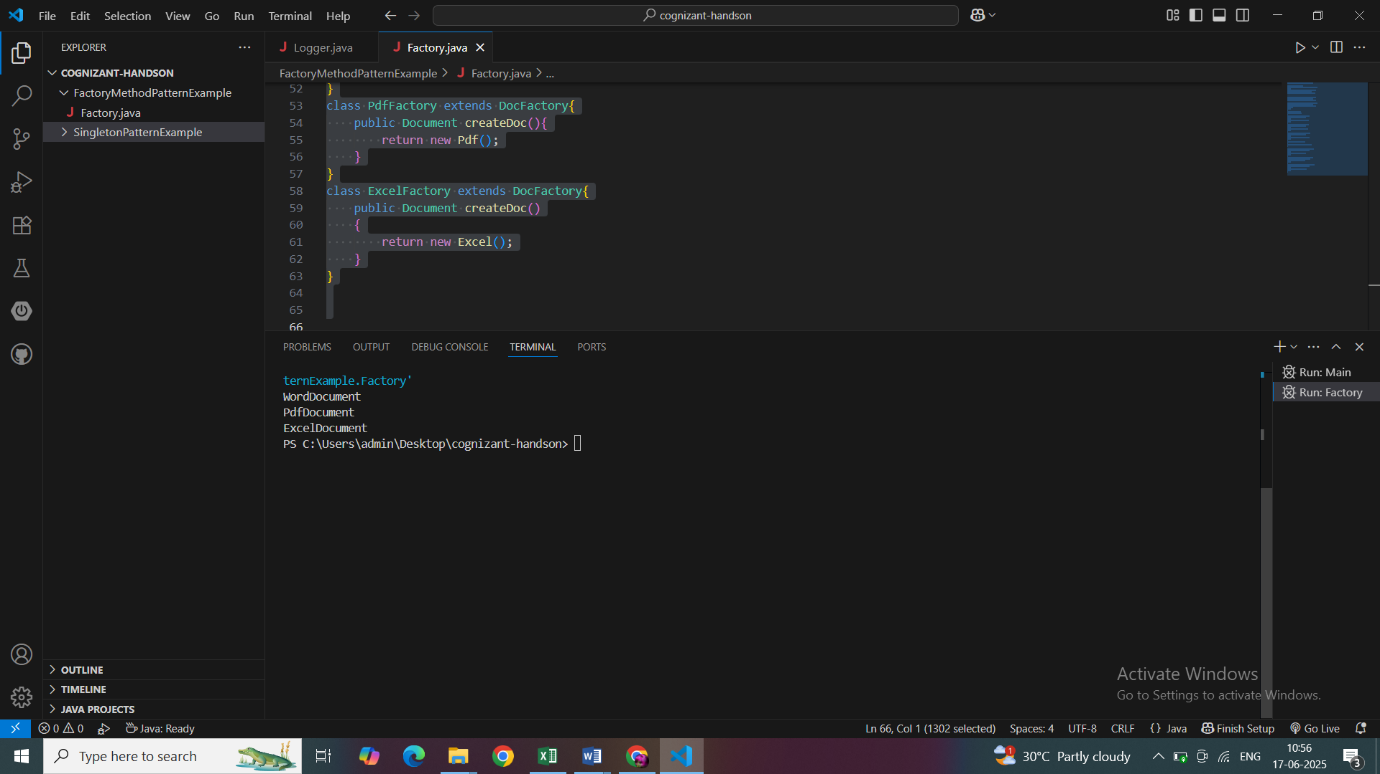
    {

        return new Excel();

    }

}

**OUTPUT:**

****

**Exercise 3: Implementing the Builder Pattern**

package BuilderPattern;

 class Computer {

    private String cpu;

    private String ram;

    private String storage;

    private String graphicsCard;

    private String os;

     private Computer(Builder builder) {

        this.cpu = builder.cpu;

        this.ram = builder.ram;

        this.storage = builder.storage;

        this.graphicsCard = builder.graphicsCard;

        this.os = builder.os;

    }

     String getCpu()

     {

        return cpu;

    }

    String getRam()

     {

         return ram;

     }

    String getStorage()

    {

         return storage;

    }

    String getGraphicsCard()

    {

         return graphicsCard;

    }

    String getOs()

     {

         return os;

     }

    public String toString() {

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

               ", GraphicsCard=" + graphicsCard + ", OS=" + os + "]";

    }

    static class Builder {

        private String cpu;

        private String ram;

        private String storage;

        private String graphicsCard;

        private String os;

        Builder(String cpu, String ram) {

            this.cpu = cpu;

            this.ram = ram;

        }

        Builder setStorage(String storage) {

            this.storage = storage;

            return this;

        }

        Builder setGraphicsCard(String graphicsCard) {

            this.graphicsCard = graphicsCard;

            return this;

        }

        Builder setOs(String os) {

            this.os = os;

            return this;

        }

        Computer build() {

            return new Computer(this);

        }

    }

}

class Main {

    public static void main(String[] args) {

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

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

            .setStorage("1TB SSD")

            .setGraphicsCard("NVIDIA RTX 4080")

            .setOs("Windows 11")

            .build();

        Computer workstation = new Computer.Builder("Intel Xeon", "64GB")

            .setStorage("2TB SSD")

            .setOs("Ubuntu Linux")

            .build();

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

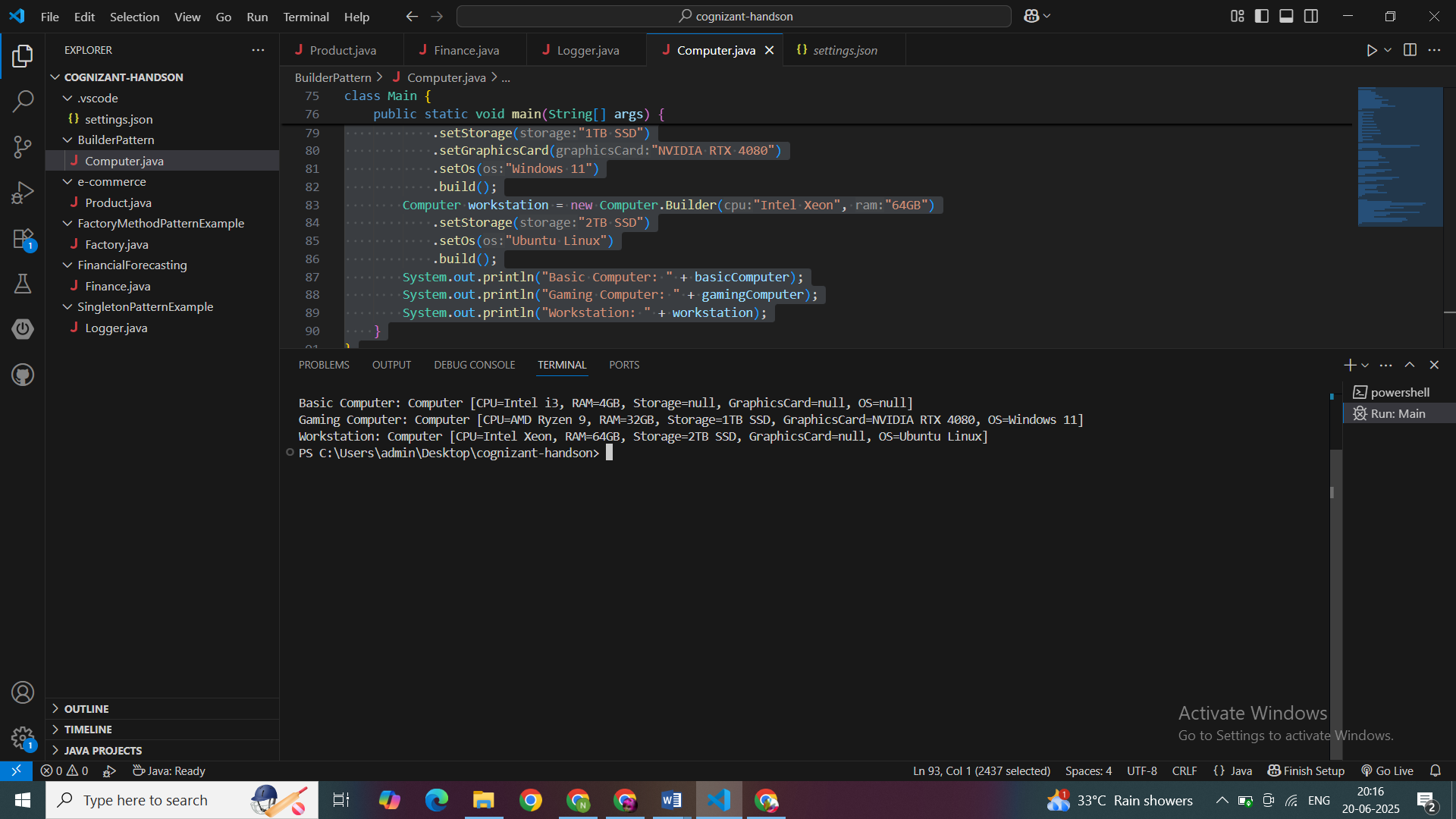
        System.out.println("Gaming Computer: " + gamingComputer);

        System.out.println("Workstation: " + workstation);

    }

}

**OUTPUT:**

****

**Exercise 4: Implementing the Adapter Pattern**

package AdapterDesignPatternExample;

public class Payment {

    public static void main(String[] args) {

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

        paypalProcessor.processPayment(100.0);

        PaymentProcessor stripeProcessor = new StripeAdapter();

        stripeProcessor.processPayment(250.5);

    }

}

interface PaymentProcessor {

    void processPayment(double amount);

}

class PayPalGateway {

    void makePayment(String userEmail, double amount) {

        System.out.println("Processing PayPal payment of $" + amount + " for " + userEmail);

    }

}

class StripeGateway {

    void sendPayment(double amountInCents) {

        System.out.println("Processing Stripe payment of $" + (amountInCents / 100));

    }

}

class PayPalAdapter implements PaymentProcessor {

    private PayPalGateway paypal;

    private String userEmail;

    PayPalAdapter(String userEmail) {

        this.userEmail = userEmail;

        this.paypal = new PayPalGateway();

    }

    public void processPayment(double amount) {

        paypal.makePayment(userEmail, amount);

    }

}

class StripeAdapter implements PaymentProcessor {

    private StripeGateway stripe;

    StripeAdapter() {

        this.stripe = new StripeGateway();

    }

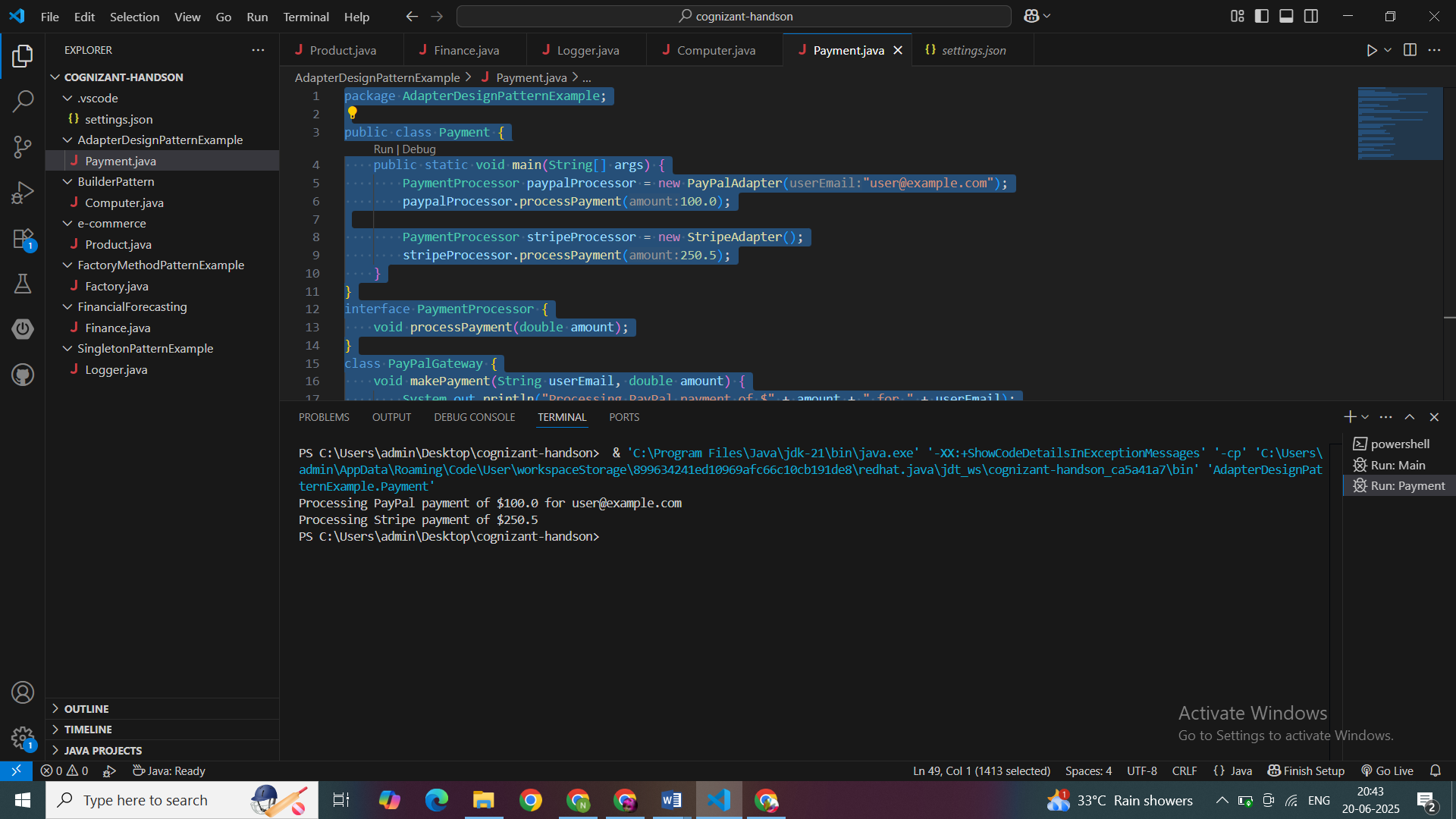
    public void processPayment(double amount) {

        stripe.sendPayment(amount \* 100);

    }

}

**OUTPUT:**

****

**Exercise 5: Implementing the Decorator Pattern**

package DecoratorDesignPattern;

public class SMS {

    public static void main(String[] args) {

        Notifier emailNotifier = new EmailNotifier();

        System.out.println("== Email Only ==");

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

        Notifier smsAndEmailNotifier = new SMSNotifierDecorator(new EmailNotifier());

        System.out.println("\n== Email + SMS ==");

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

        Notifier fullNotifier = new SlackNotifierDecorator(

                                  new SMSNotifierDecorator(

                                    new EmailNotifier()));

        System.out.println("\n== Email + SMS + Slack ==");

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

    }

}

interface Notifier {

    void send(String message);

}

class EmailNotifier implements Notifier {

    public void send(String message) {

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

    }

}

abstract class NotifierDecorator implements Notifier {

    protected Notifier notifier;

    NotifierDecorator(Notifier notifier) {

        this.notifier = notifier;

    }

    public void send(String message) {

        notifier.send(message);

    }

}

class SMSNotifierDecorator extends NotifierDecorator {

    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);

    }

}

class SlackNotifierDecorator extends NotifierDecorator {

    SlackNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    public void send(String message) {

        super.send(message);

        sendSlackMessage(message);

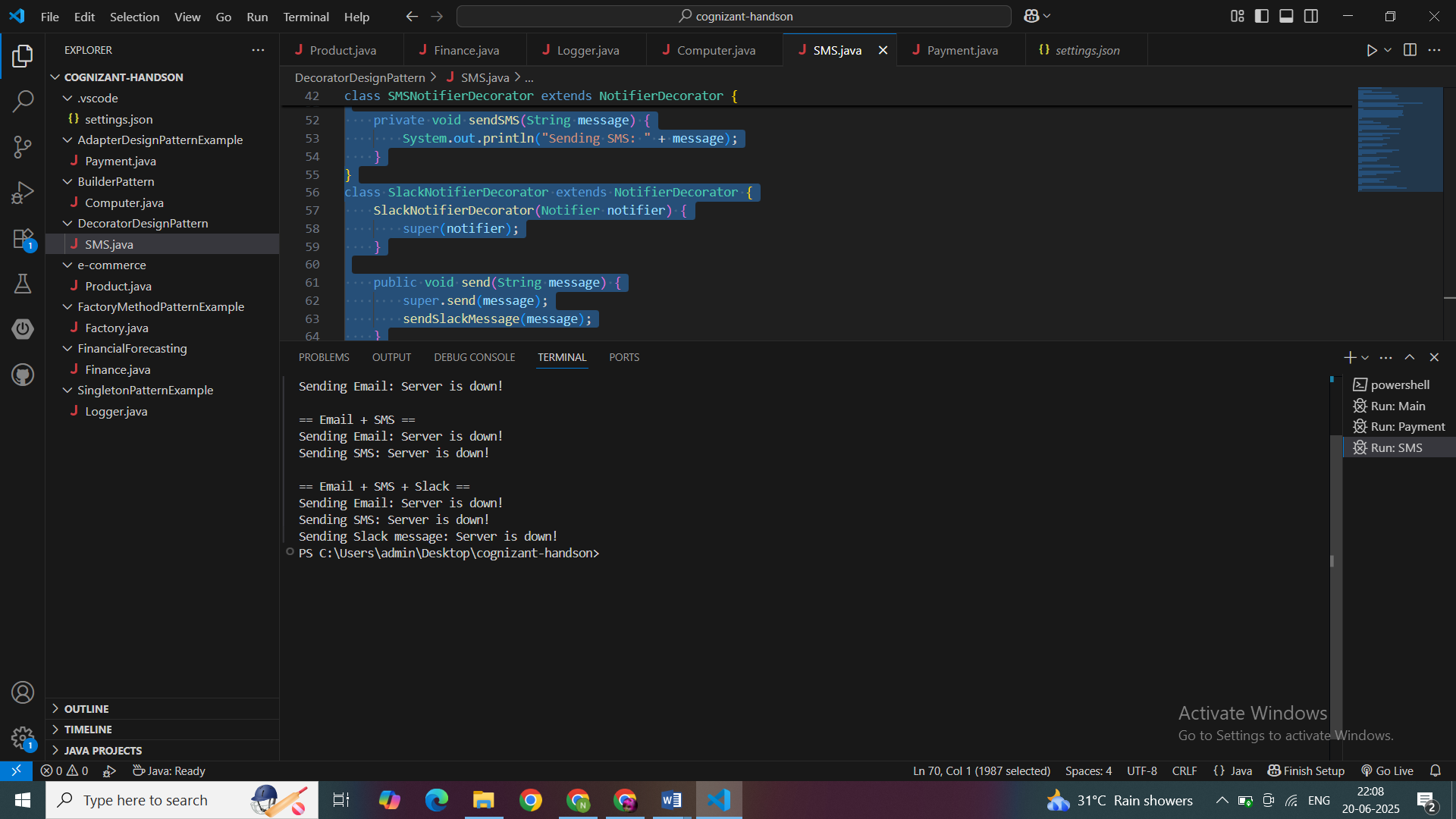
    }

    private void sendSlackMessage(String message) {

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

    }

}

**OUTPUT:  
  
**

**Exercise 6: Implementing the Proxy Pattern**

package ProxyPattern;

public class ImageNew {

    public static void main(String[] args) {

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

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

        img1.display();

        img1.display();

        img2.display();

        img2.display();

    }

}

    class ProxyImage implements Image {

    private String filename;

    private RealImage realImage;

    ProxyImage(String filename) {

        this.filename = filename;

    }

    public void display() {

        if (realImage == null) {

            realImage = new RealImage(filename);

        } else {

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

        }

        realImage.display();

    }

}

interface Image {

    void display();

}

class RealImage implements Image {

    private String filename;

    RealImage(String filename) {

        this.filename = filename;

        loadFromRemoteServer();

    }

    private void loadFromRemoteServer() {

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

        try {

            Thread.sleep(1000);

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

    }

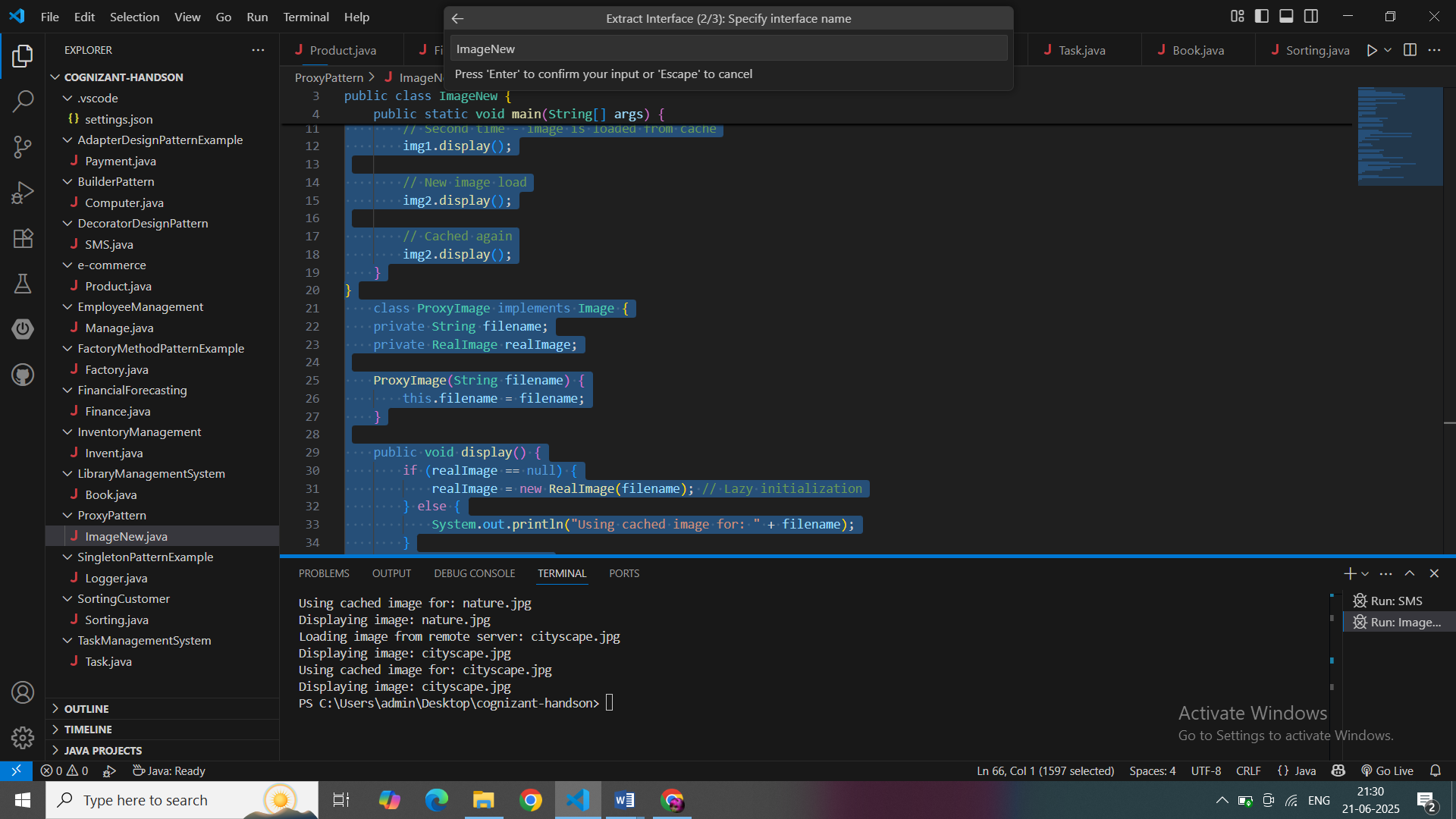
    public void display() {

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

    }

}

**OUTPUT:**

****

**Exercise 7: Implementing the Observer Pattern**

package ObserverPattern;

import java.util.\*;

public class StockNew{

     public static void main(String[] args) {

        StockMarket market = new StockMarket();

        Observer mobileUser1 = new MobileApp("Alice");

        Observer mobileUser2 = new MobileApp("Bob");

        Observer webUser1 = new WebApp("Charlie");

        market.registerObserver(mobileUser1);

        market.registerObserver(mobileUser2);

        market.registerObserver(webUser1);

        market.setStockPrice("TCS", 3500.50);

        market.setStockPrice("INFY", 1520.75);

        market.deregisterObserver(mobileUser2);

        market.setStockPrice("RELIANCE", 2789.10);

    }

}

interface Stock {

    void registerObserver(Observer o);

    void deregisterObserver(Observer o);

    void notifyObservers();

}

class StockMarket implements Stock {

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

    private String stockName;

    private double stockPrice;

    public void setStockPrice(String stockName, double stockPrice) {

        this.stockName = stockName;

        this.stockPrice = stockPrice;

        System.out.println("\nStock updated: " + stockName + " at Rs." + stockPrice);

        notifyObservers();

    }

    public void registerObserver(Observer o) {

        observers.add(o);

        System.out.println(o + " registered.");

    }

    public void deregisterObserver(Observer o) {

        observers.remove(o);

        System.out.println(o + " deregistered.");

    }

    public void notifyObservers() {

        for (Observer o : observers) {

            o.update(stockName, stockPrice);

        }

    }

}

interface Observer {

    void update(String stockName, double stockPrice);

}

class MobileApp implements Observer {

    private String user;

    MobileApp(String user) {

        this.user = user;

    }

    public void update(String stockName, double stockPrice) {

        System.out.println("[MobileApp - " + user + "] " + stockName + " is now Rs." + stockPrice);

    }

    public String toString() {

        return "MobileApp(" + user + ")";

    }

}

class WebApp implements Observer {

    private String user;

    WebApp(String user) {

        this.user = user;

    }

    public void update(String stockName, double stockPrice) {

        System.out.println("[WebApp - " + user + "] " + stockName + " is now Rs." + stockPrice);

    }

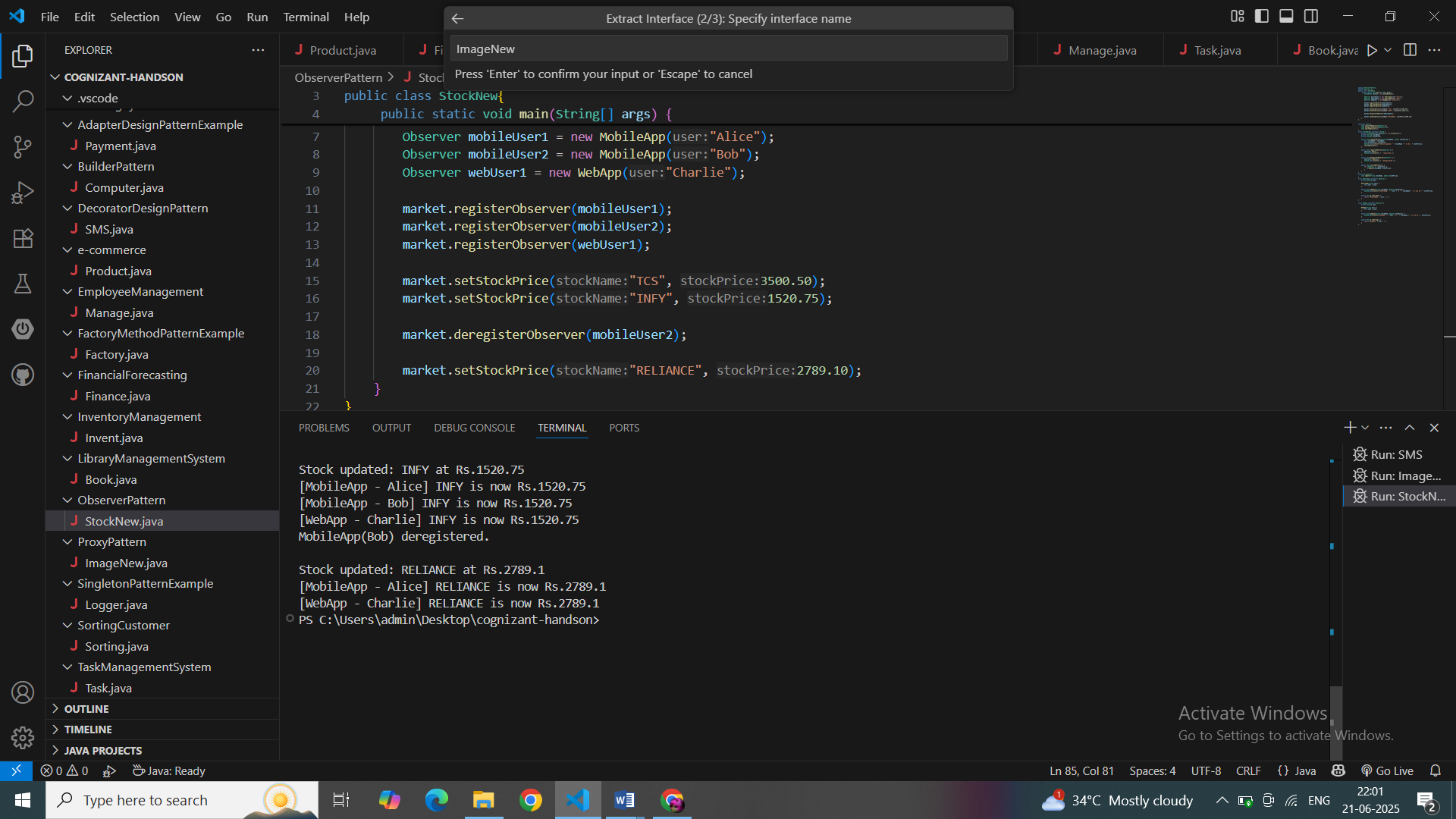
    public String toString() {

        return "WebApp(" + user + ")";

    }

}

**OUTPUT:**

****

**Exercise 8: Implementing the Strategy Pattern**

package Strategy;

public class Payment {

     public static void main(String[] args) {

        PaymentContext context = new PaymentContext();

        // Pay with Credit Card

        PaymentStrategy creditCard = new CreditCardPayment("1234567812345678", "Hema N");

        context.setPaymentStrategy(creditCard);

        context.pay(2500.00);

        System.out.println();

        // Pay with PayPal

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

        context.setPaymentStrategy(paypal);

        context.pay(1500.50);

    }

}

interface PaymentStrategy {

    void pay(double amount);

}

class CreditCardPayment implements PaymentStrategy {

    private String cardNumber;

    private String cardHolder;

    CreditCardPayment(String cardNumber, String cardHolder) {

        this.cardNumber = cardNumber;

        this.cardHolder = cardHolder;

    }

    public void pay(double amount) {

        System.out.println("Paid Rs." + amount + " using Credit Card.");

        System.out.println("Card Holder: " + cardHolder);

        System.out.println("Card Number: " + maskCardNumber(cardNumber));

    }

    private String maskCardNumber(String number) {

        return "XXXX-XXXX-XXXX-" + number.substring(number.length() - 4);

    }

}

class PayPalPayment implements PaymentStrategy {

    private String email;

    PayPalPayment(String email) {

        this.email = email;

    }

    public void pay(double amount) {

        System.out.println("Paid Rs." + amount + " using PayPal.");

        System.out.println("PayPal Account: " + email);

    }

}

class PaymentContext {

    private PaymentStrategy strategy;

    public void setPaymentStrategy(PaymentStrategy strategy) {

        this.strategy = strategy;

    }

    public void pay(double amount) {

        if (strategy == null) {

            System.out.println("Please select a payment method.");

        } else {

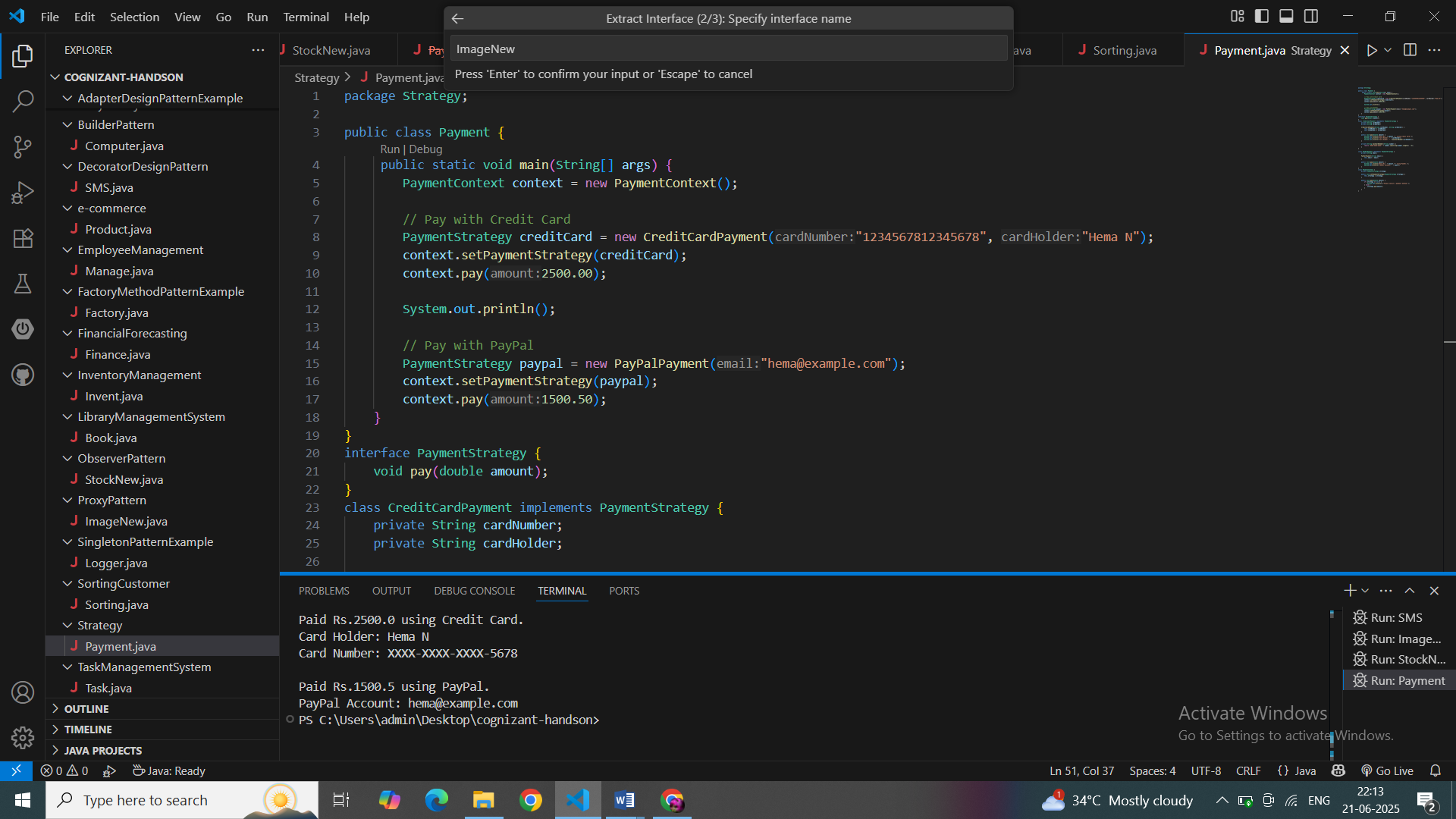
            strategy.pay(amount);

        }

    }

}

**OUTPUT:**

****

**Exercise 9: Implementing the Command Pattern**

package CommandPattern;

public class LightNew {

     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();

    }

}

interface Command {

    void execute();

}

class Light {

    void turnOn() {

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

    }

    void turnOff() {

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

    }

}

class LightOnCommand implements Command {

    Light light;

    LightOnCommand(Light light) {

        this.light = light;

    }

    public void execute() {

        light.turnOn();

    }

}

class LightOffCommand implements Command {

    Light light;

    LightOffCommand(Light light) {

        this.light = light;

    }

    public void execute() {

        light.turnOff();

    }

}

class RemoteControl {

    Command command;

    void setCommand(Command command) {

        this.command = command;

    }

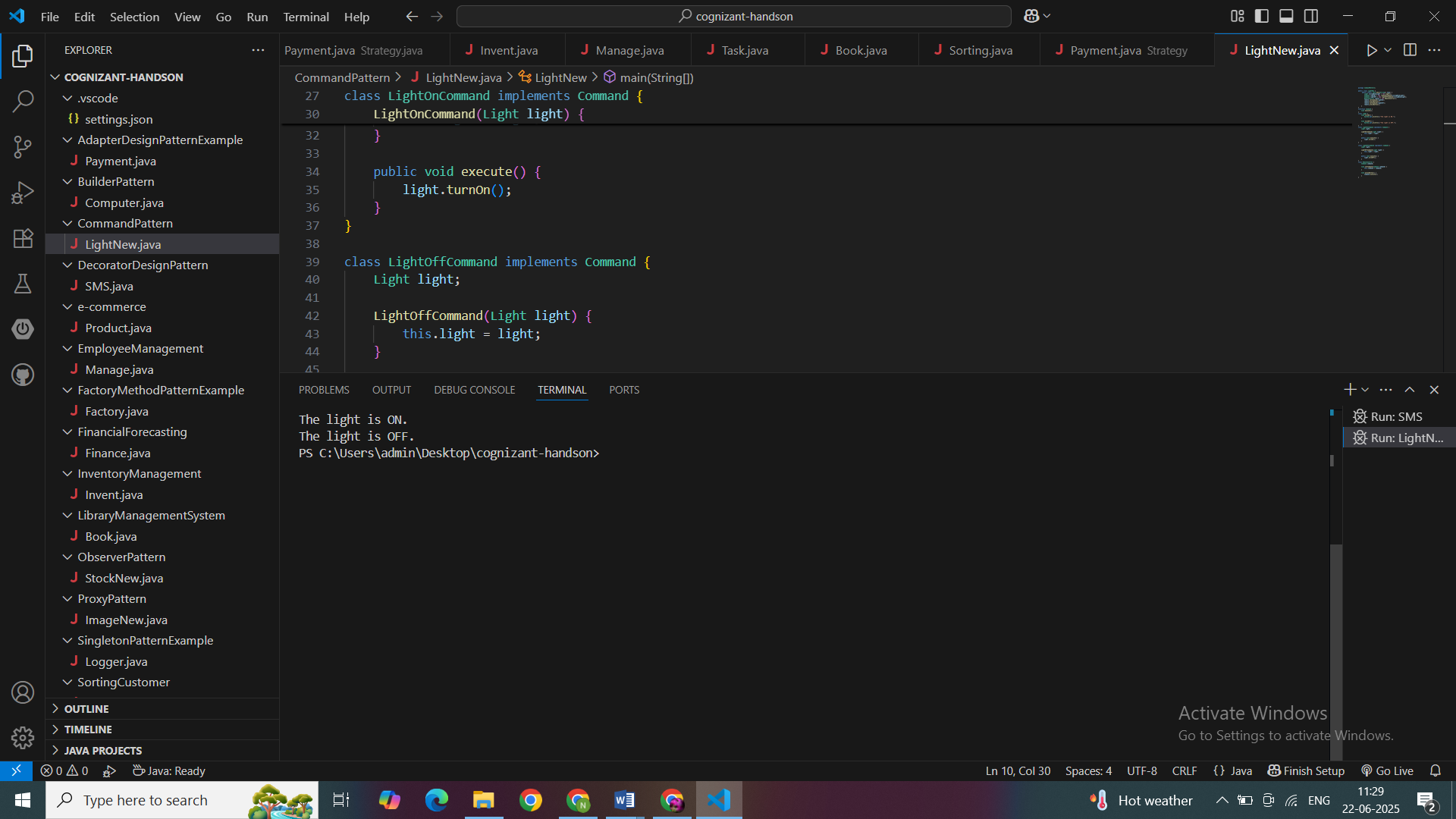
    void pressButton() {

        command.execute();

    }

}

**OUTPUT:**

****

**Exercise 10: Implementing the MVC Pattern**

package MVCPattern;

class Student {

    private String name;

    private String id;

    private String grade;

    // Constructor

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

        this.name = name;

        this.id = id;

        this.grade = grade;

    }

    // Getters and Setters

    String getName() {

        return name;

    }

    void setName(String name) {

        this.name = name;

    }

    String getId() {

        return id;

    }

    void setId(String id) {

        this.id = id;

    }

    String getGrade() {

        return grade;

    }

    void setGrade(String grade) {

        this.grade = grade;

    }

}

class StudentView {

    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;

    StudentController(Student model, StudentView view) {

        this.model = model;

        this.view = view;

    }

    // Update model data

    void setStudentName(String name) {

        model.setName(name);

    }

    void setStudentId(String id) {

        model.setId(id);

    }

    void setStudentGrade(String grade) {

        model.setGrade(grade);

    }

    // Retrieve model data

    String getStudentName() {

        return model.getName();

    }

    String getStudentId() {

        return model.getId();

    }

    String getStudentGrade() {

        return model.getGrade();

    }

    // Display updated view

    void updateView() {

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

    }

}

public class MVC {

    public static void main(String[] args) {

        // Create model and view

        Student student = new Student("John Doe", "S101", "A");

        StudentView view = new StudentView();

        // Create controller

        StudentController controller = new StudentController(student, view);

        controller.updateView();

        // Update student data

        controller.setStudentName("Jane Smith");

        controller.setStudentGrade("A+");

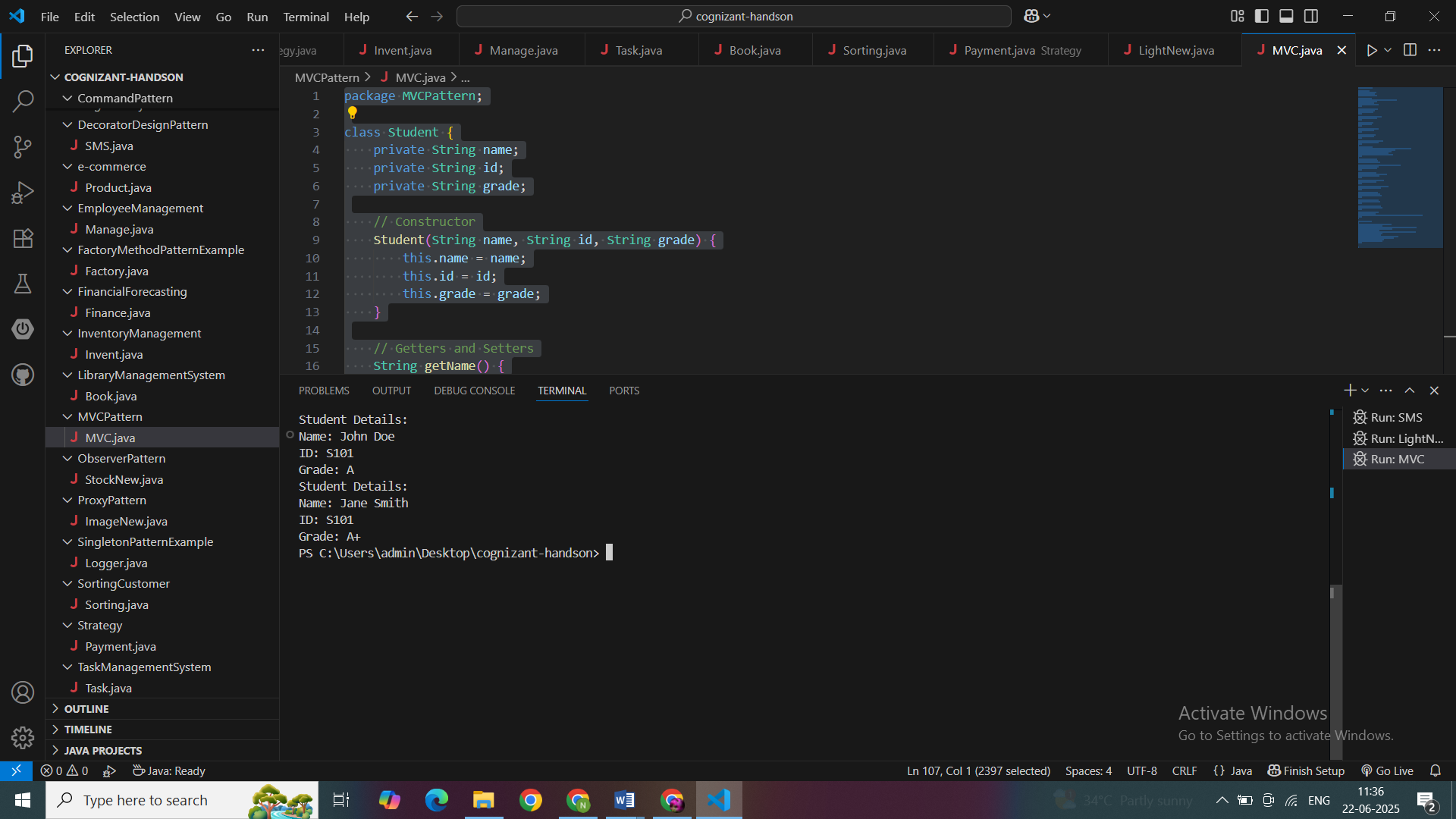
        // Display updated details

        controller.updateView();

    }

}

**OUTPUT:**

****

**Exercise 11: Implementing Dependency Injection**

interface CustomerRepository {

    String findCustomerById(int id);

}

class CustomerRepositoryImpl implements CustomerRepository {

    public String findCustomerById(int id) {

        if (id == 1) return "Customer: Alice";

        else if (id == 2) return "Customer: Bob";

        else return "Customer not found";

    }

}

class CustomerService {

    CustomerRepository customerRepository;

    CustomerService(CustomerRepository customerRepository) {

        this.customerRepository = customerRepository;

    }

    void displayCustomer(int id) {

        String customer = customerRepository.findCustomerById(id);

        System.out.println(customer);

    }

}

public class Customer {

    public static void main(String[] args) {

        CustomerRepository repository = new CustomerRepositoryImpl();

        CustomerService service = new CustomerService(repository);

        service.displayCustomer(1); // Output: Customer: Alice

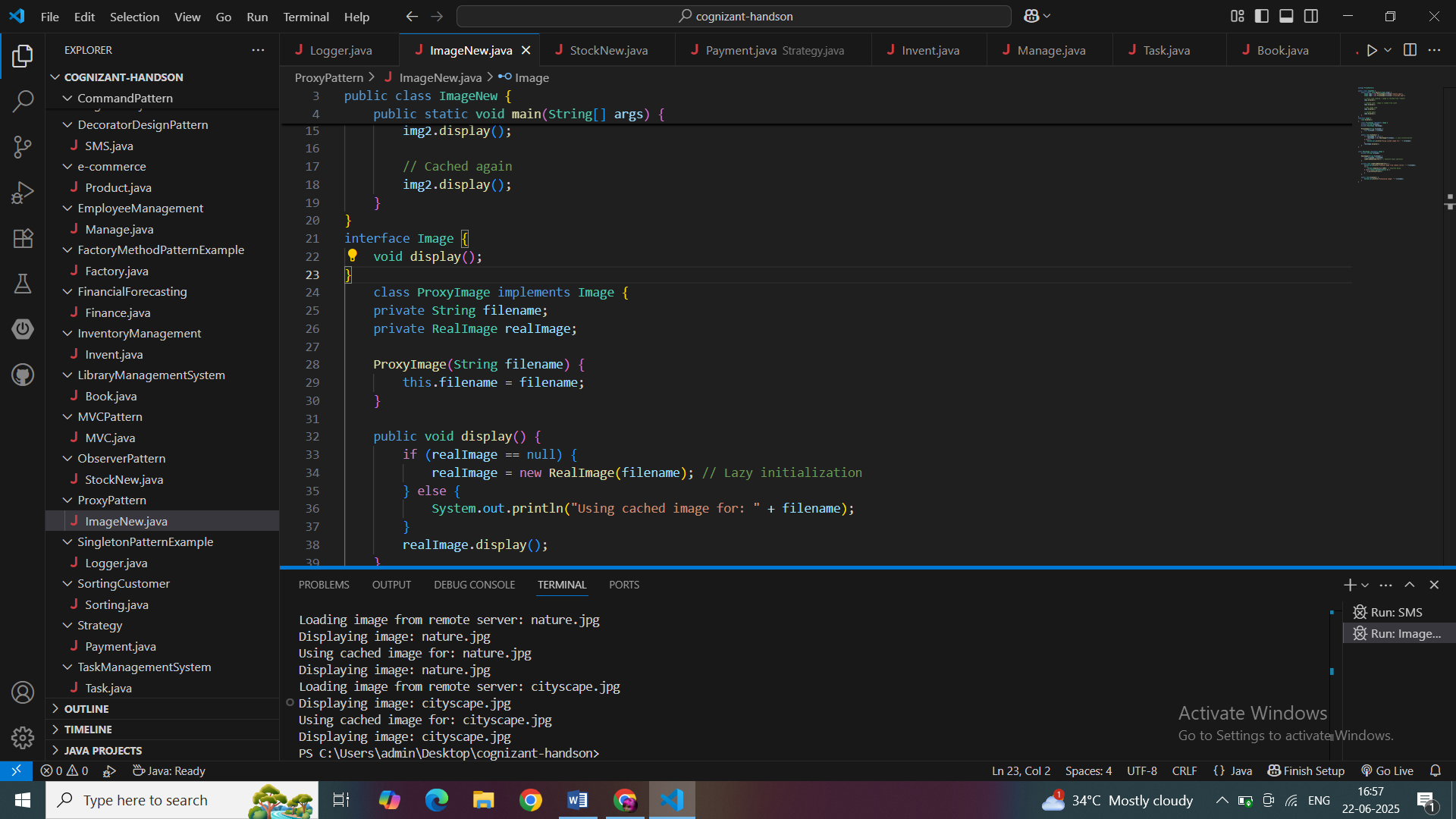
        service.displayCustomer(2); // Output: Customer: Bob

        service.displayCustomer(3); // Output: Customer not found

    }

}

**OUTPUT:**

****