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{

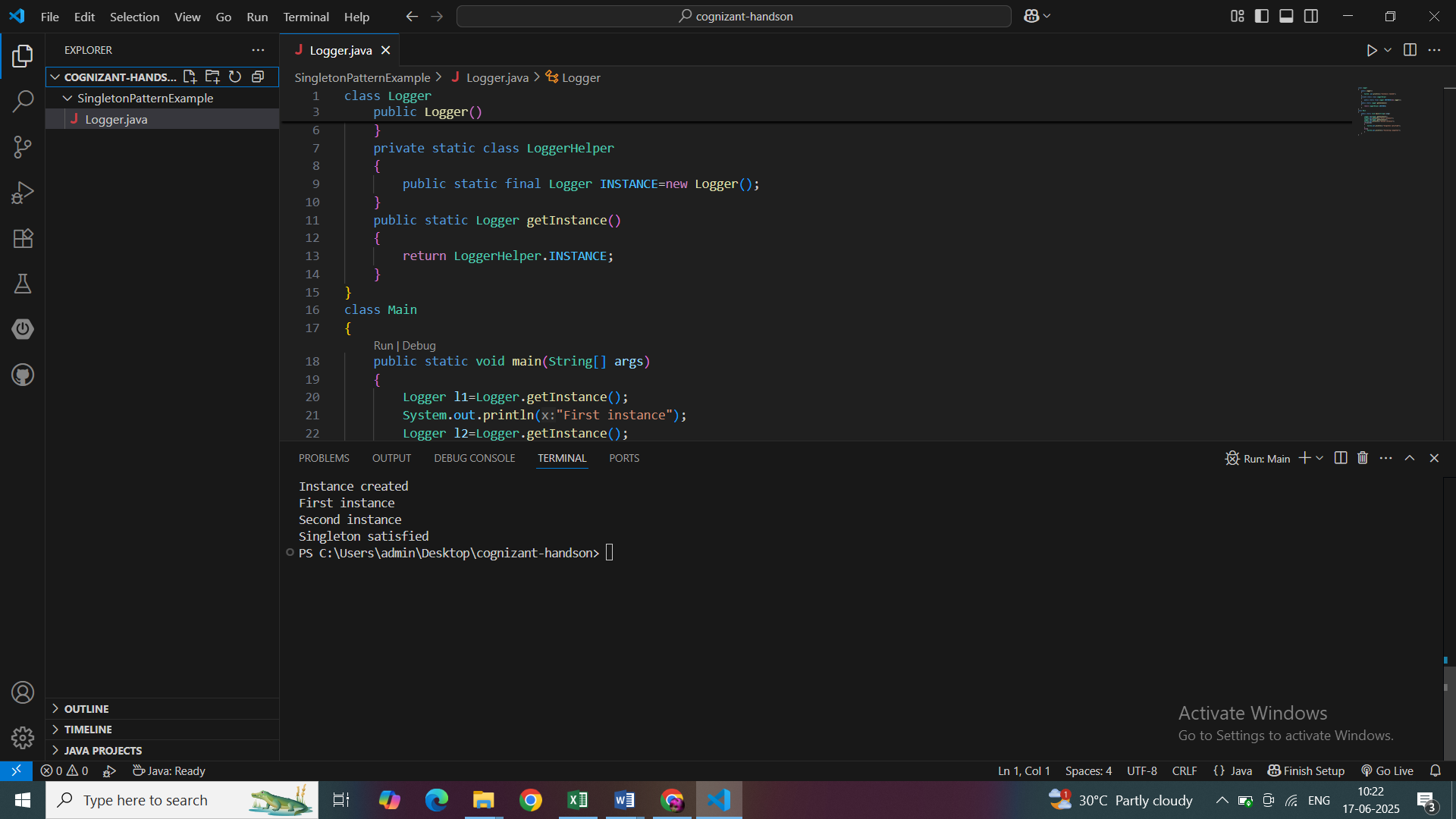
            System.out.println("Violating singleton");

        }

    }

}

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()

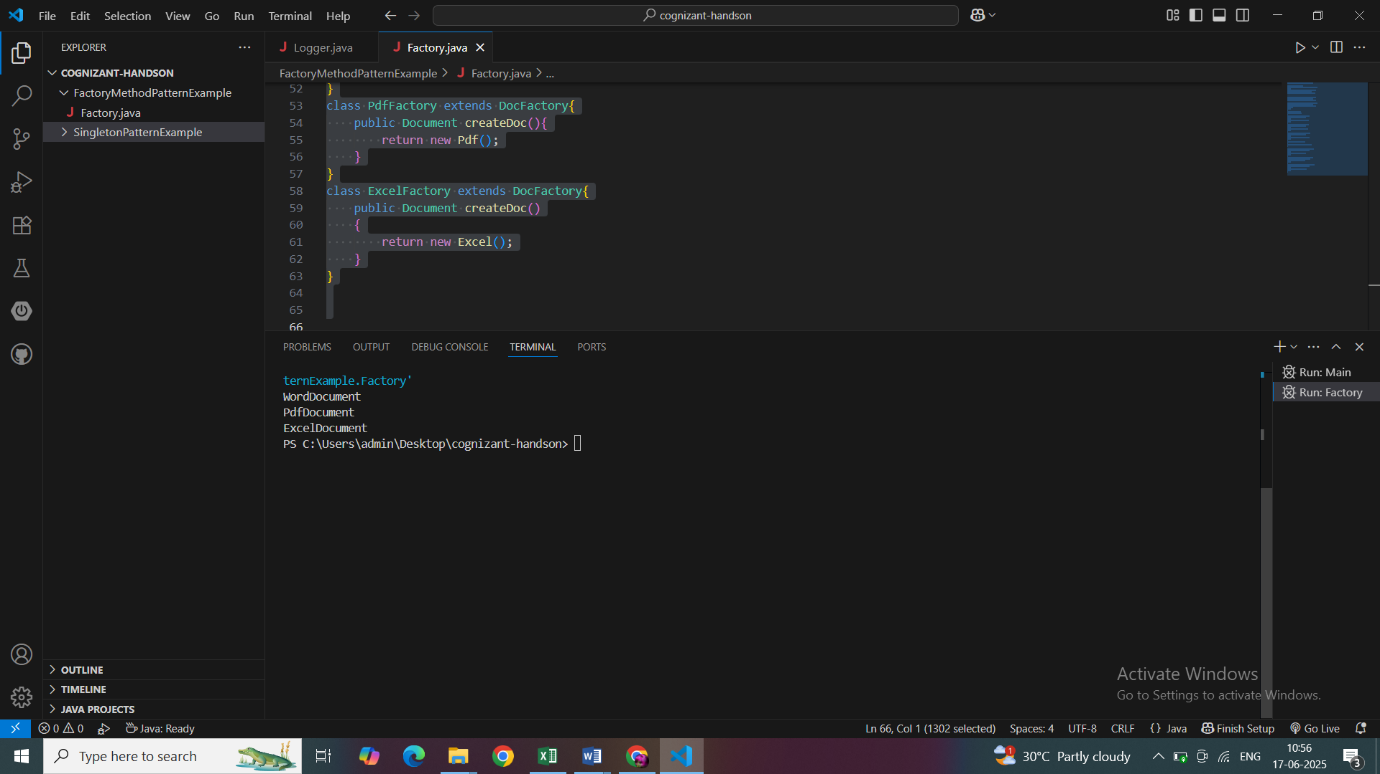
    {

        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);

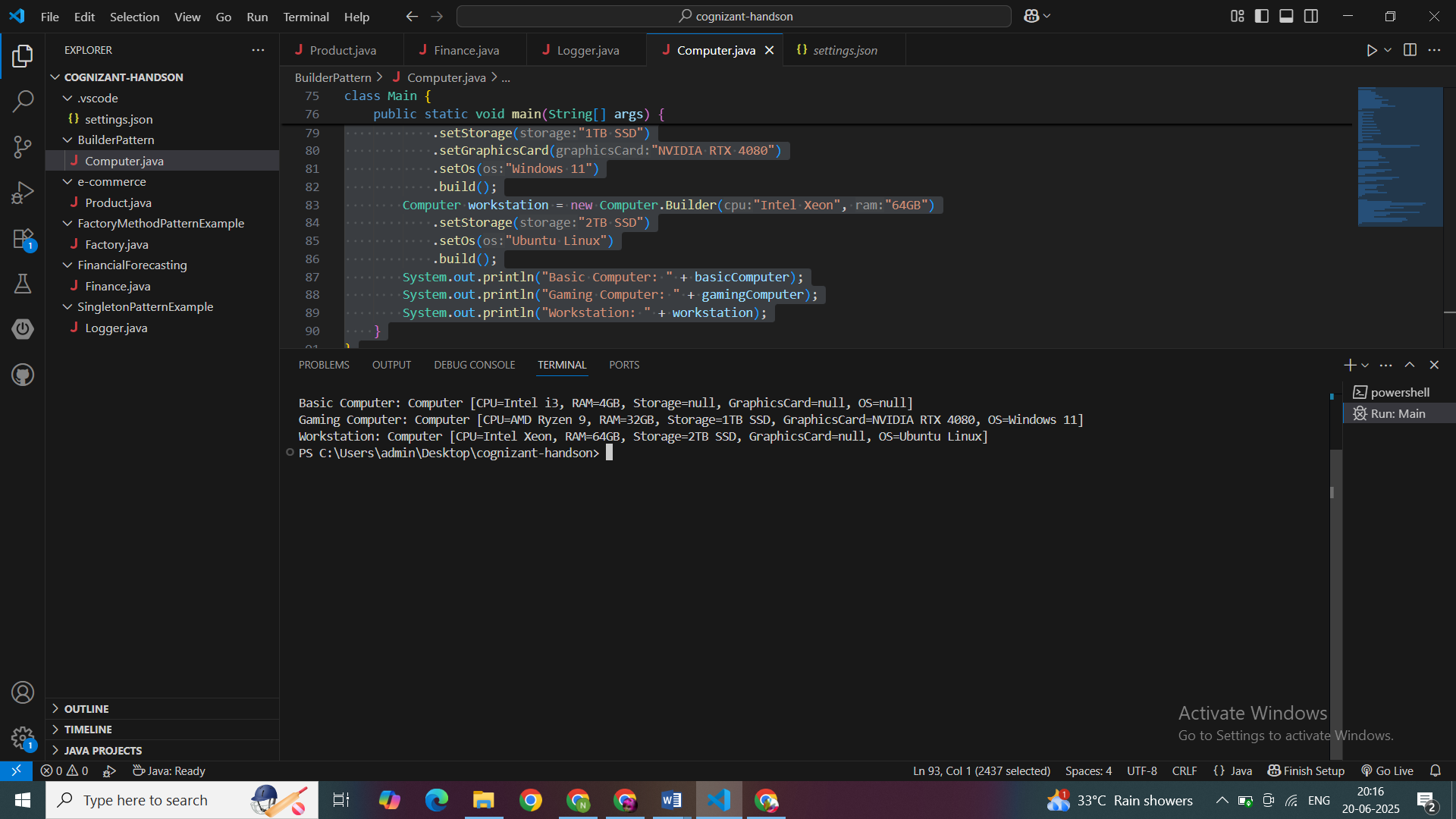
        System.out.println("Gaming Computer: " + gamingComputer);

        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();

    }

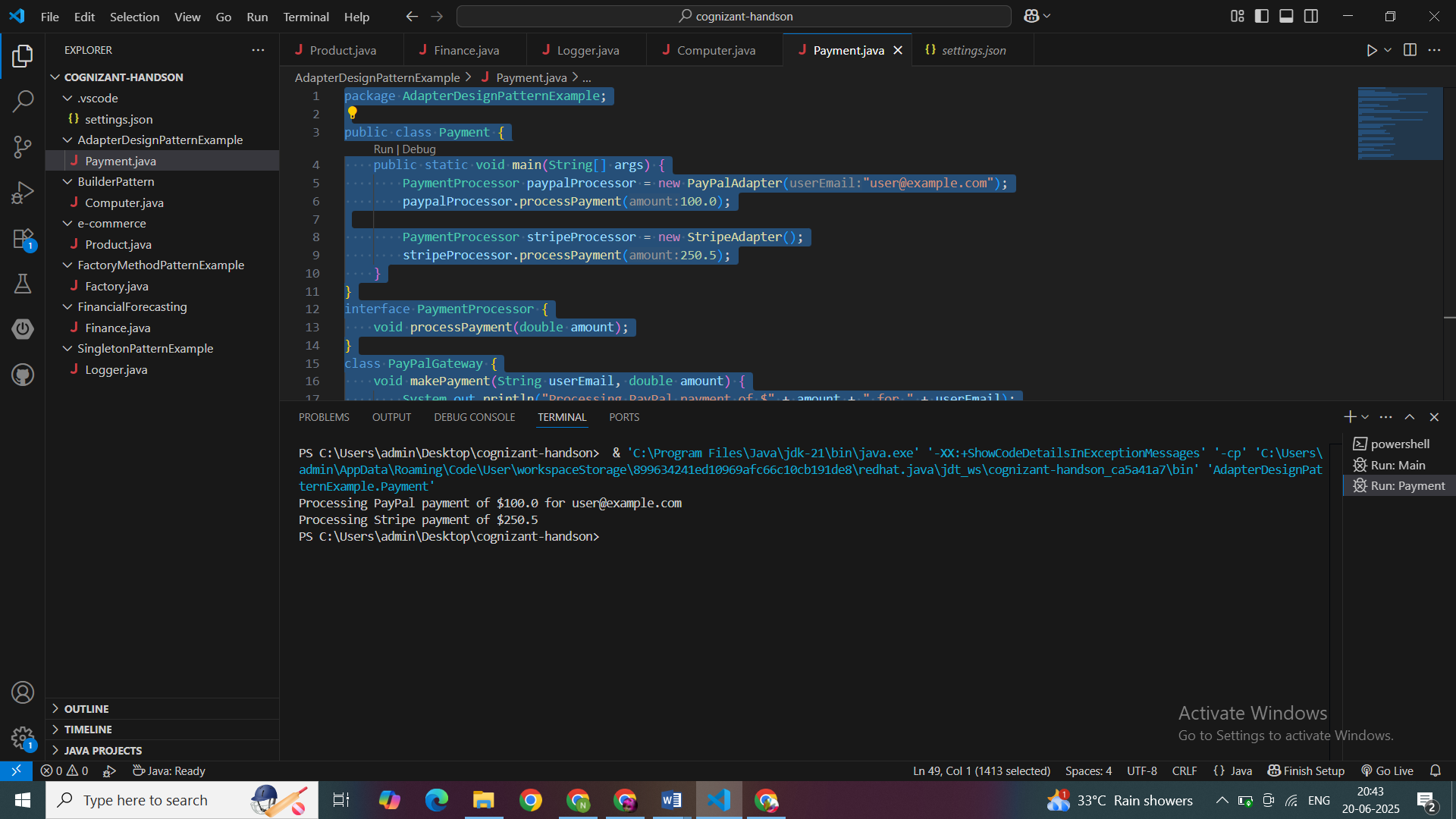
    public void processPayment(double amount) {

        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);

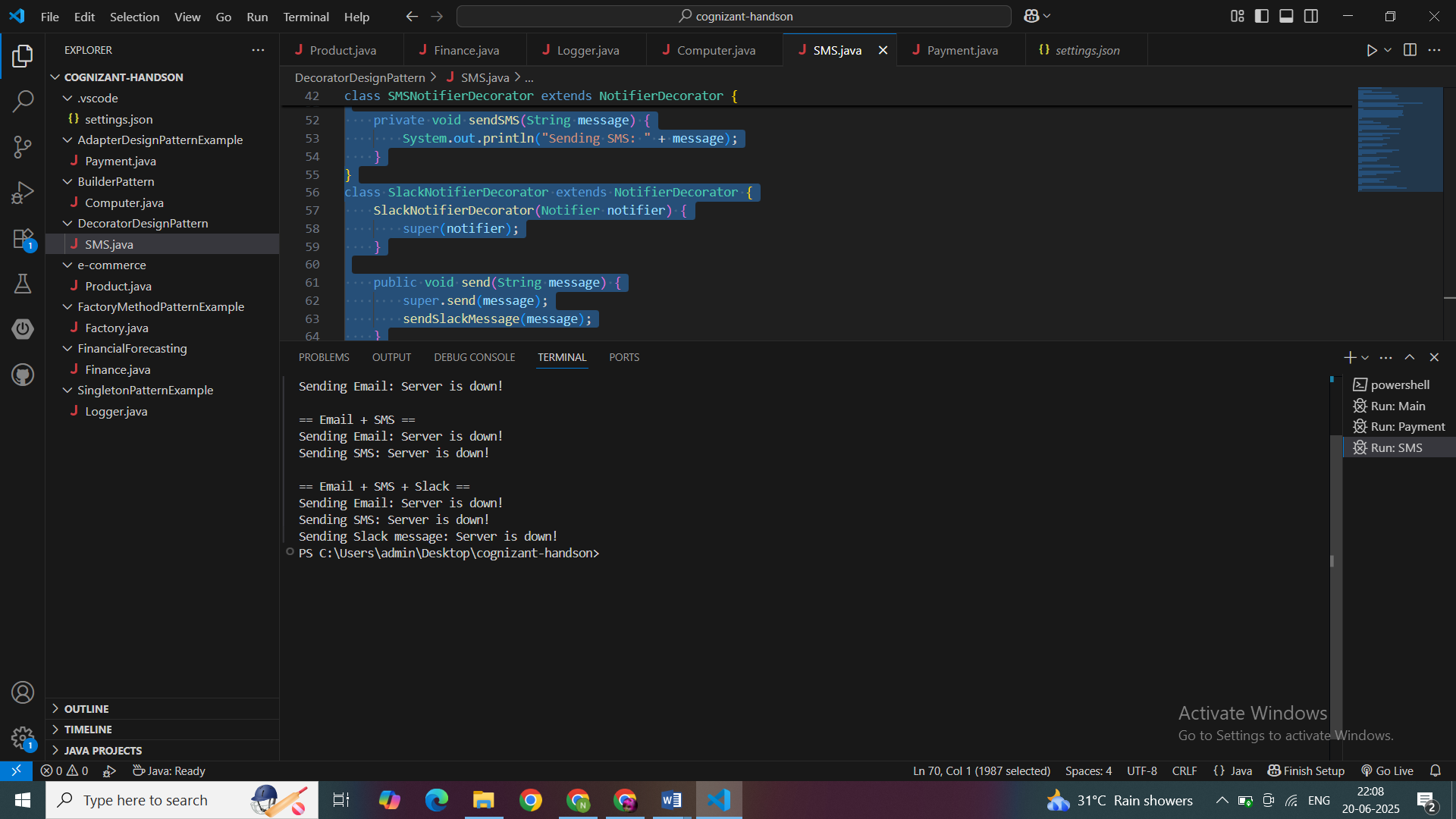
    }

    private void sendSlackMessage(String message) {

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

    }

}

**OUTPUT:  
  
**