**Design Principles and patterns**

**Exercise 1: Implementing the singleton pattern**

**Code:**

class Logger{

private static Logger instance;

    private Logger(){

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

    }

    public static Logger getInstance(){

        if(instance==null){

            instance= new Logger();

        }

        return instance;

    }

    public void print(String message){

        System.out.println(message);

    }

}

 class TestSingleton{

  public static void main(String[] args){

    Logger l1= Logger.getInstance();

    l1.print("first log message");

    Logger l2=Logger.getInstance();

    l2.print("second log message");

    if(l1==l2)

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

    else System.out.println("singleton doesnt work");

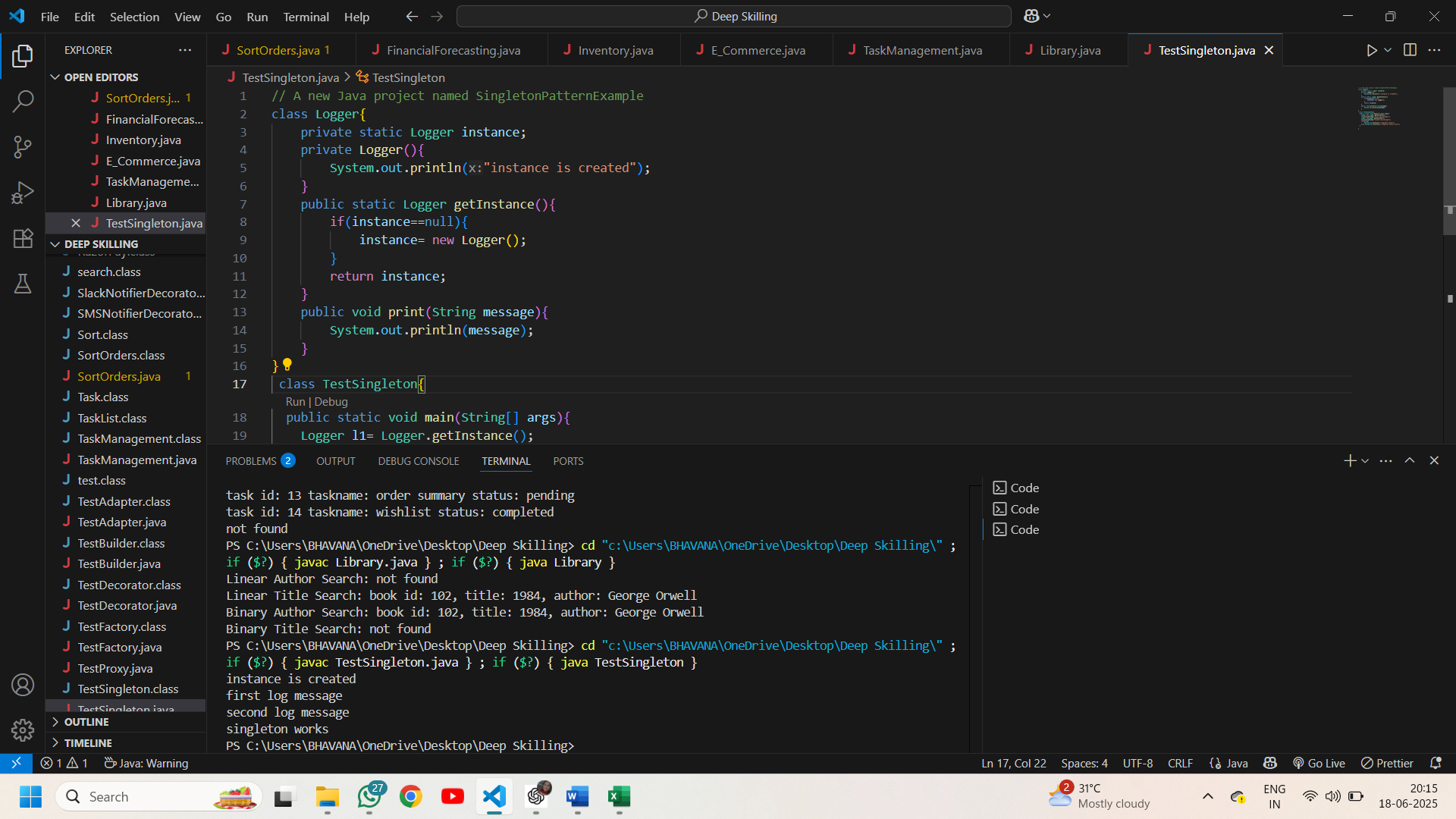
}

}

  }

}

**Output:**



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

**Code:**

interface Document{

    void open();

}

class word implements Document{

     public void open(){

         System.out.println("opening word doc");

     }

}

class Pdf implements Document{

     public void open(){

        System.out.println("opening pdf doc");

     }

}

class excel implements Document{

    public void open(){

       System.out.println("opening excel doc");

    }

}

abstract class DocumentFactory{

    abstract Document createDocument();

}

class CreateWord extends DocumentFactory{

     Document createDocument(){

        return new word();

    }

}

class createPdf extends DocumentFactory{

    Document createDocument(){

        return new Pdf();

    }

}

class createExcel extends DocumentFactory{

    Document createDocument(){

       return new excel();

    }

}

public class TestFactory {

    public static void main(String[] args){

        CreateWord ob= new CreateWord();

        Document d=ob.createDocument();

        d.open();

        createPdf ob1= new createPdf();

        Document d1=ob1.createDocument();

        d1.open();

        createExcel ob2= new createExcel();

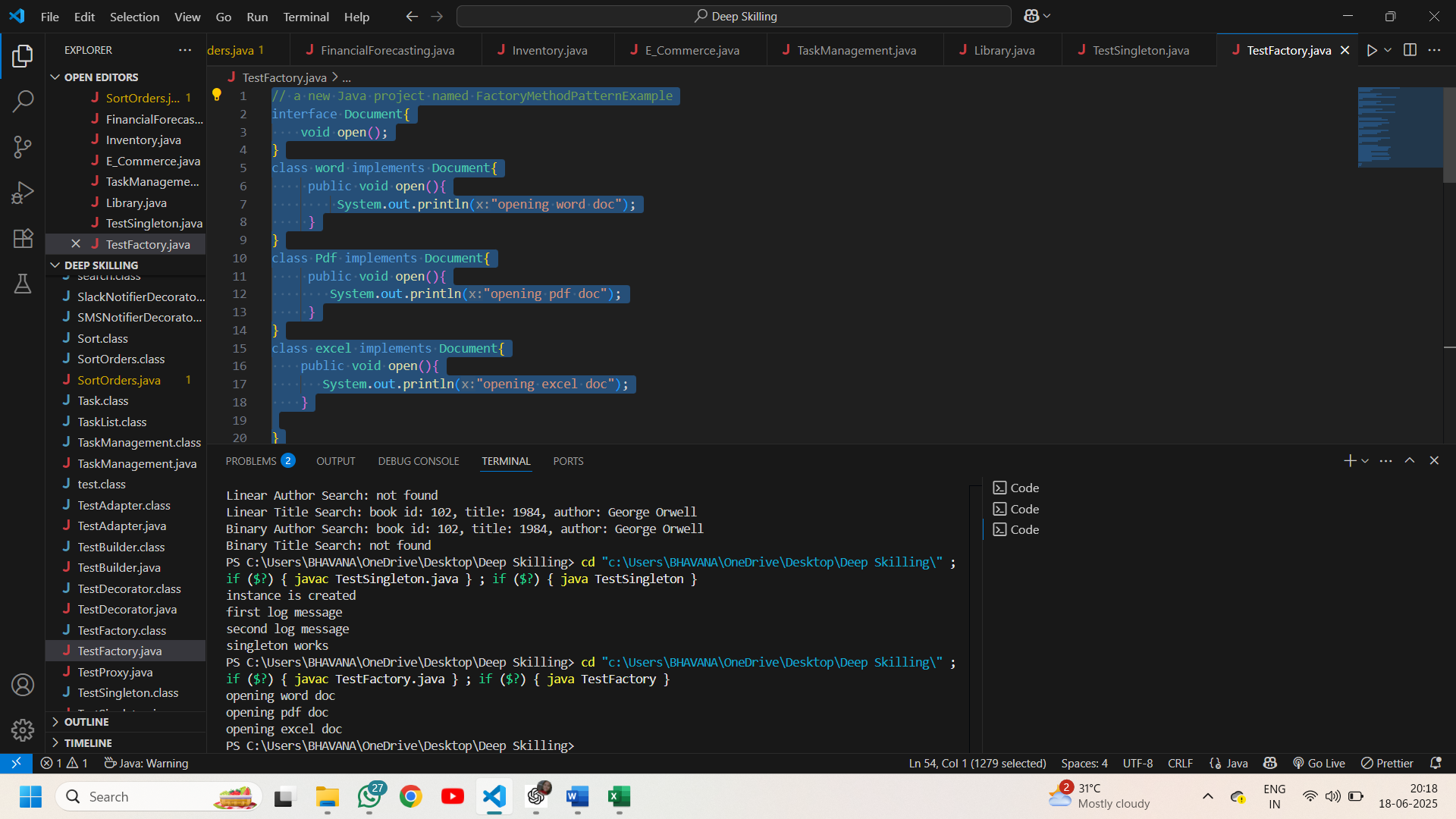
        Document d2=ob2.createDocument();

        d2.open();

    }

}

**Output:**



**Exercise 3: Implementing the Builder Pattern**

**Code:**

class Computer {

     String cpu;

     int ram, storage;

    private Computer(Builder b){

        this.cpu=b.cpu;

        this.ram=b.ram;

        this.storage=b.storage;

    }

    static class Builder{

        String cpu;

        int ram, storage;

         Builder setCPU(String c){

           this.cpu=c;

           return this;

        }

        Builder setRAM(int r){

            this.ram=r;

            return this;

        }

        Builder setStorage(int s){

            this.storage=s;

            return this;

        }

        Computer build(){

            return new Computer(this);

        }

    }

}

class TestBuilder{

    public static void main(String[] args) {

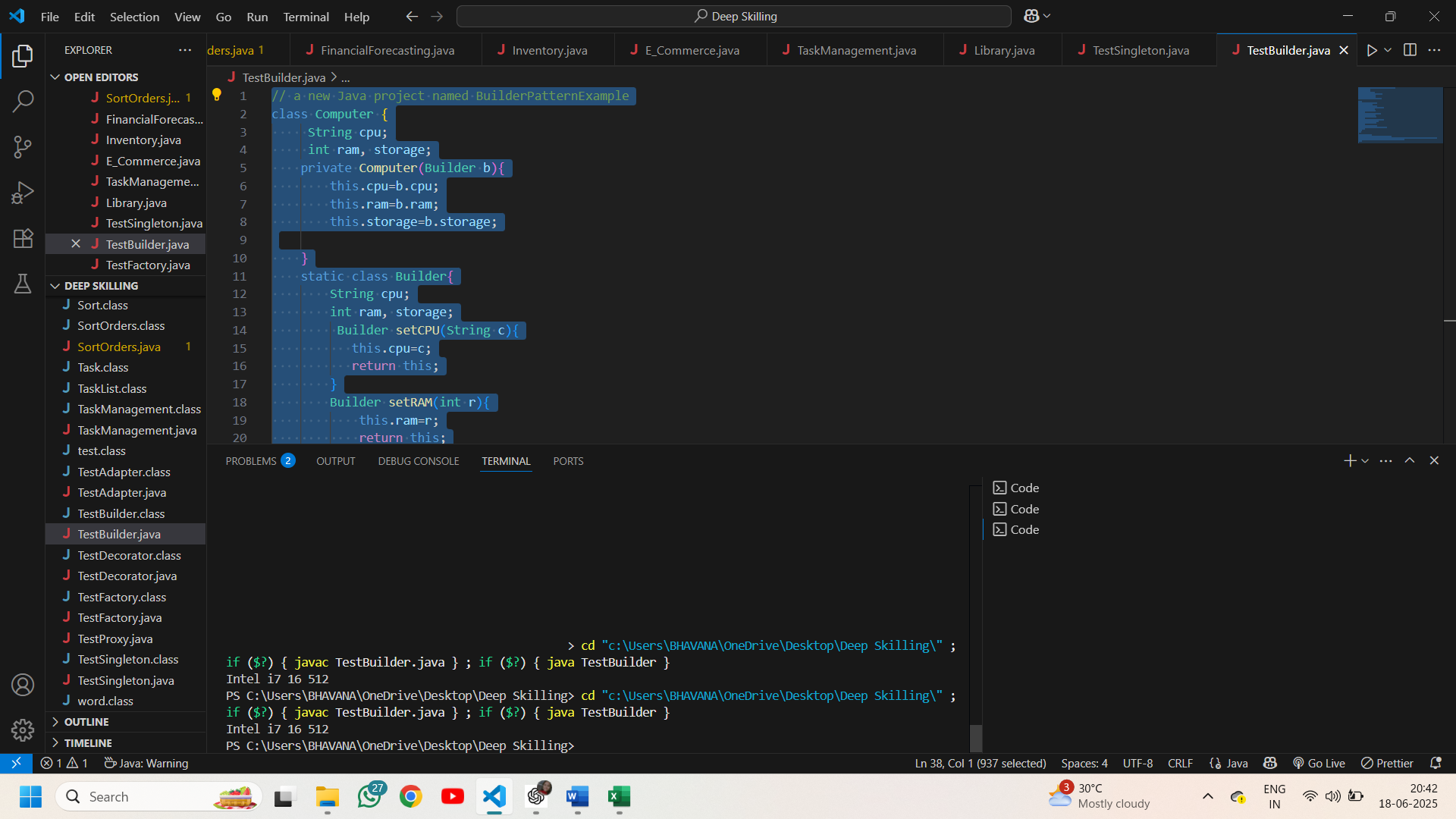
        Computer ob= new Computer.Builder().setCPU("Intel i7").setRAM(16).setStorage(512).build();

        System.out.println(ob.cpu+" "+ob.ram+" "+ob.storage);

    }

}

**Output:**



**Exercise 4: Implementing the Adapter Pattern**

**Code:**

interface PaymentProcessor{

    void processPayment();

}

class Gpay{

    void payment(){

        System.out.println("payment through gpay");

    }

}

class Phonepe{

    void payment(){

        System.out.println("payment through phonpe");

    }

}

class RazorPay{

    void payment(){

        System.out.println("payment through razorpay");

    }

}

class paymentGpay implements PaymentProcessor{

    public void processPayment(){

        Gpay ob= new Gpay();

        ob.payment();

    }

}

class paymentPhonepe implements PaymentProcessor{

    public void processPayment(){

        Phonepe ob= new Phonepe();

        ob.payment();

    }

}

class paymentRazorpay implements PaymentProcessor{

    public void processPayment(){

RazorPay ob= new RazorPay();

        ob.payment();

    }

}

public class TestAdapter {

    public static void main(String[] args) {

        PaymentProcessor ob1= new paymentGpay();

        ob1.processPayment();

        PaymentProcessor ob2= new paymentPhonepe();

        ob2.processPayment();

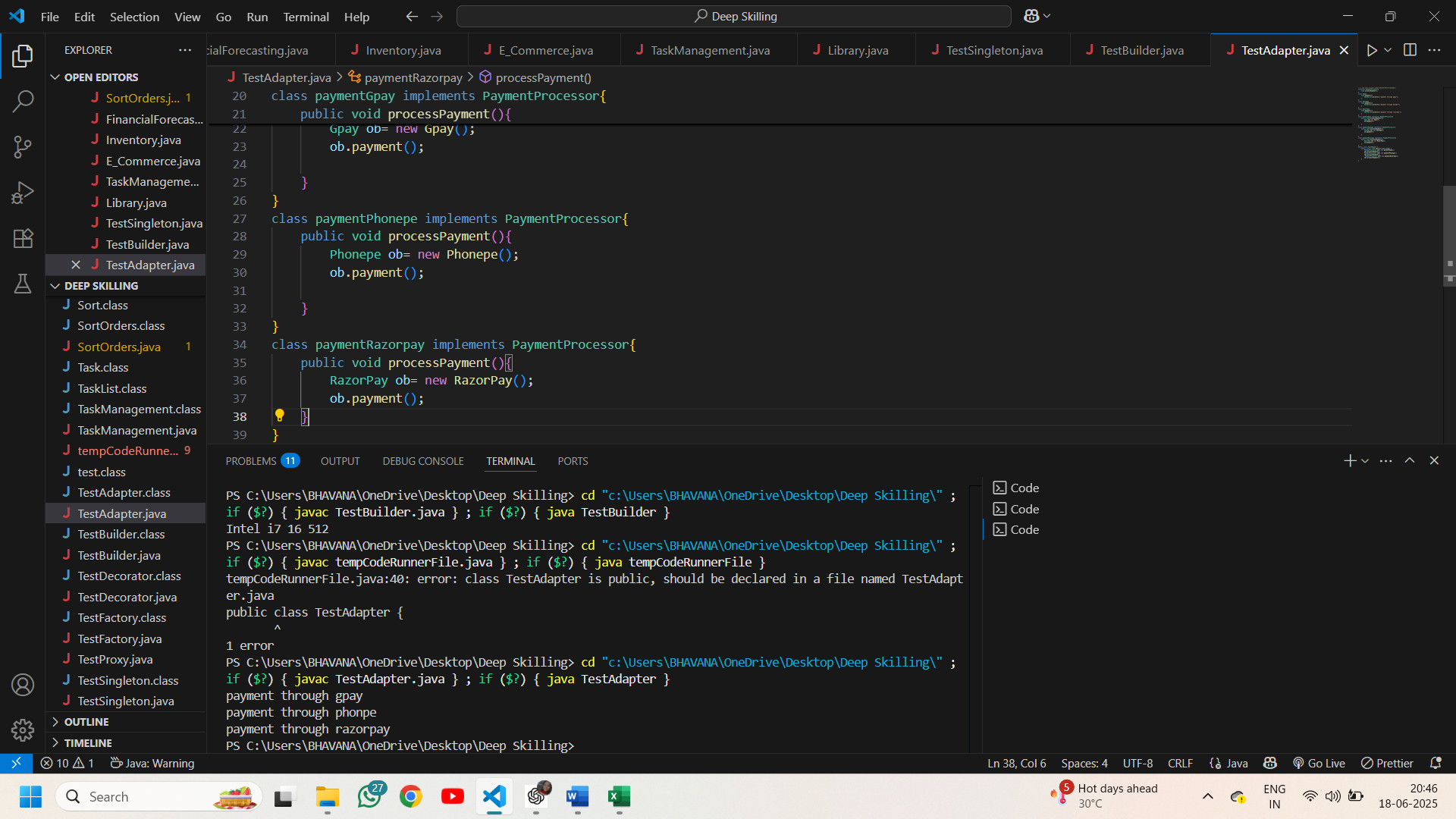
         PaymentProcessor ob3= new paymentRazorpay();

        ob3.processPayment();

    }

}

**Output:**



**Exercise 5: Implementing the Decorator Pattern**

**Code:**

interface Notifier {

    void send();

}

class EmailNotifier implements Notifier{

    public void send(){

     System.out.println("notification via email");

    }

}

abstract class NotifierDecorator implements Notifier{

    Notifier instance;

}

class SMSNotifierDecorator extends NotifierDecorator{

     public void send(){

     System.out.println("notification via sms");

     }

}

class SlackNotifierDecorator extends NotifierDecorator{

    public void send(){

        System.out.println("notification via slack");

    }

}

public class TestDecorator {

    public static void main(String args[]){

        SMSNotifierDecorator ob1=new SMSNotifierDecorator();

        ob1.send();

        SlackNotifierDecorator ob2 = new SlackNotifierDecorator();

        ob2.send();

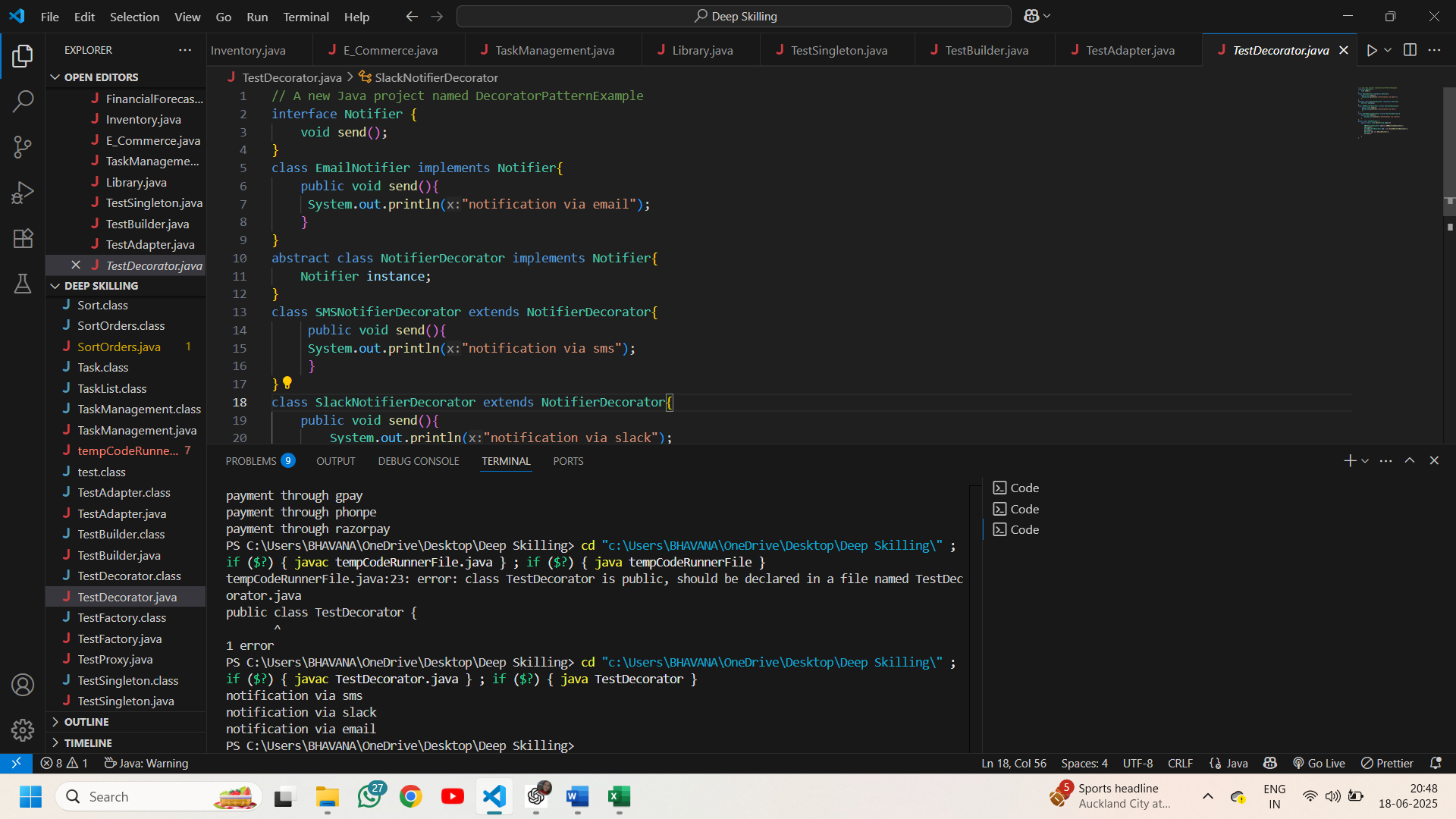
        Notifier ob= new EmailNotifier();

        ob.send();

    }

}

**Output:**



**Exercise 6: Implementing the Proxy Pattern**

**Code:**

import java.util.\*;

interface image{

    void display();

}

class RealImage implements image{

    String filename;

    RealImage(String filename){

        this.filename=filename;

        loadFromRemoteServer();

    }

    void loadFromRemoteServer(){

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

        try{

            Thread.sleep(1000);

        }catch(InterruptedException e){

            e.printStackTrace();

        }

    }

    public void display(){

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

    }

}

class ProxyImage implements image{

    String filename;

    Map<String,RealImage> mp= new HashMap<>();

    ProxyImage(String filename){

        this.filename=filename;

    }

    public void display(){

        RealImage real= mp.get(filename);

        if(real==null){

            System.out.println("Image not in cache. Loading...");

            real=new RealImage(filename);

            mp.put(filename, real);

        }else System.out.println("Loaded from cache.");

        real.display();

    }

    RealImage inst;

}

public class TestProxy {

    public static void main(String[] args) {

        image image1 = new ProxyImage("waterfalls.jpg");

        image image2 = new ProxyImage("city.jpg");

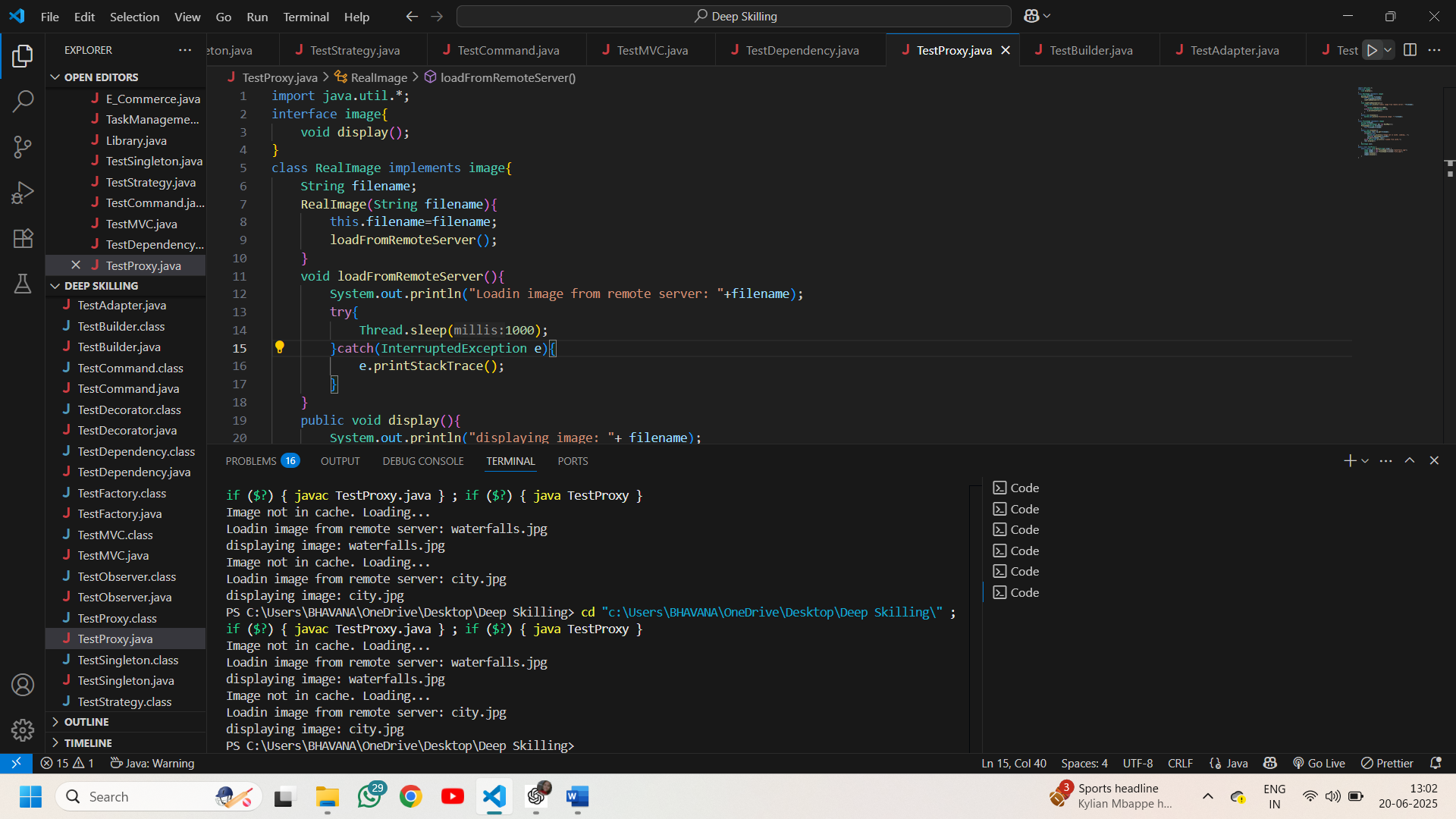
        image1.display();

        image2.display();

    }

}

**Output:**



**Exercise 7: Implementing the Observer Pattern**

**Code:**

import java.util.\*;

interface stock{

    void register(Observer o);

    void de\_register(Observer o);

    void notify\_observers();

}

class StockMarket implements stock{

    List<Observer> obs= new ArrayList<>();

    double price;

    void setPrice(double price){

        this.price=price;

        notify\_observers();

    }

    public void register(Observer o){

        obs.add(o);

    }

    public void de\_register(Observer o){

        obs.remove(o);

    }

    public void notify\_observers(){

       // MobileApp ob= new MobileApp();

        for(Observer ob:obs){

          ob.update(price);

        }

    }

}

interface Observer{

    void update(double price);

}

class MobileApp implements Observer{

    String user;

    MobileApp(String u){

        this.user=u;

    }

    public void update(double p){

        System.out.println(user+", Mobile:stock price got updated to "+ p);

    }

}

class WebApp implements Observer{

    String user;

    WebApp(String u){

        this.user=u;

    }

    public void update(double p){

       System.out.println(user+ ", Web: stock price got updated! "+ p);

    }

}

public class TestObserver {

    public static void main(String args[]){

        StockMarket sm= new StockMarket();

        Observer o1= new MobileApp("Ramesh");

        Observer o2= new WebApp("Henry");

        sm.register(o1);

        sm.register(o2);

        sm.setPrice(7653.8);

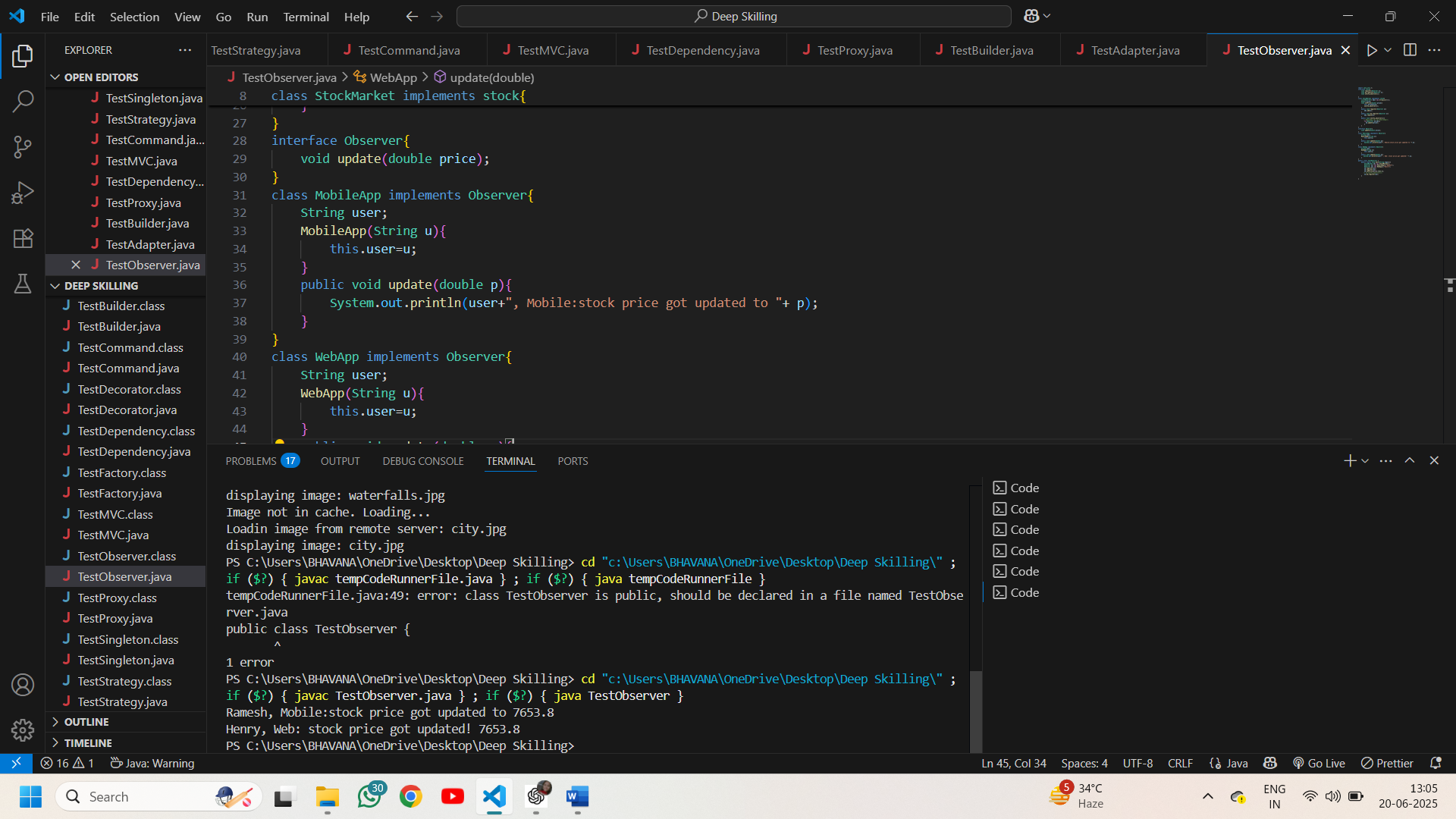
       // sm.notify\_observers();

        sm.de\_register(o2);

    }

}

**Output:**



**Exercise 8: Implementing the Strategy Pattern**

**Code:**

interface PaymentStrategy{

    void pay(double amount);

}

class CreditCardPayment implements PaymentStrategy{

    String cardnumber;

    String cardHolderName;

    CreditCardPayment(String cardnumber,String cardHolderName){

        this.cardnumber= cardnumber;

        this.cardHolderName=cardHolderName;

    }

    public void pay(double amount){

        System.out.println(amount+"/- paid using credit card whose number is: " +cardnumber);

    }

}

class PayPalPayment implements PaymentStrategy{

    String email;

    PayPalPayment(String e){

        this.email=e;

    }

    public void pay(double amount){

        System.out.println(amount+"/- paid using paypal whose email is: "+email);

    }

}

class PaymentContext{

    PaymentStrategy ps;

    void setPayment(PaymentStrategy strat){

        this.ps=strat;

    }

    void payAmount(double amount){

    if(ps==null) System.out.println("payment method is not selected");

    else{

        ps.pay(amount);

    }

    }

}

public class TestStrategy {

    public static void main(String[] args) {

    PaymentContext ob= new PaymentContext();

    ob.payAmount(35131.82);

    ob.setPayment(new CreditCardPayment("1234-5632-5642-6432","Bhavana Ravada"));

    ob.payAmount(51777);

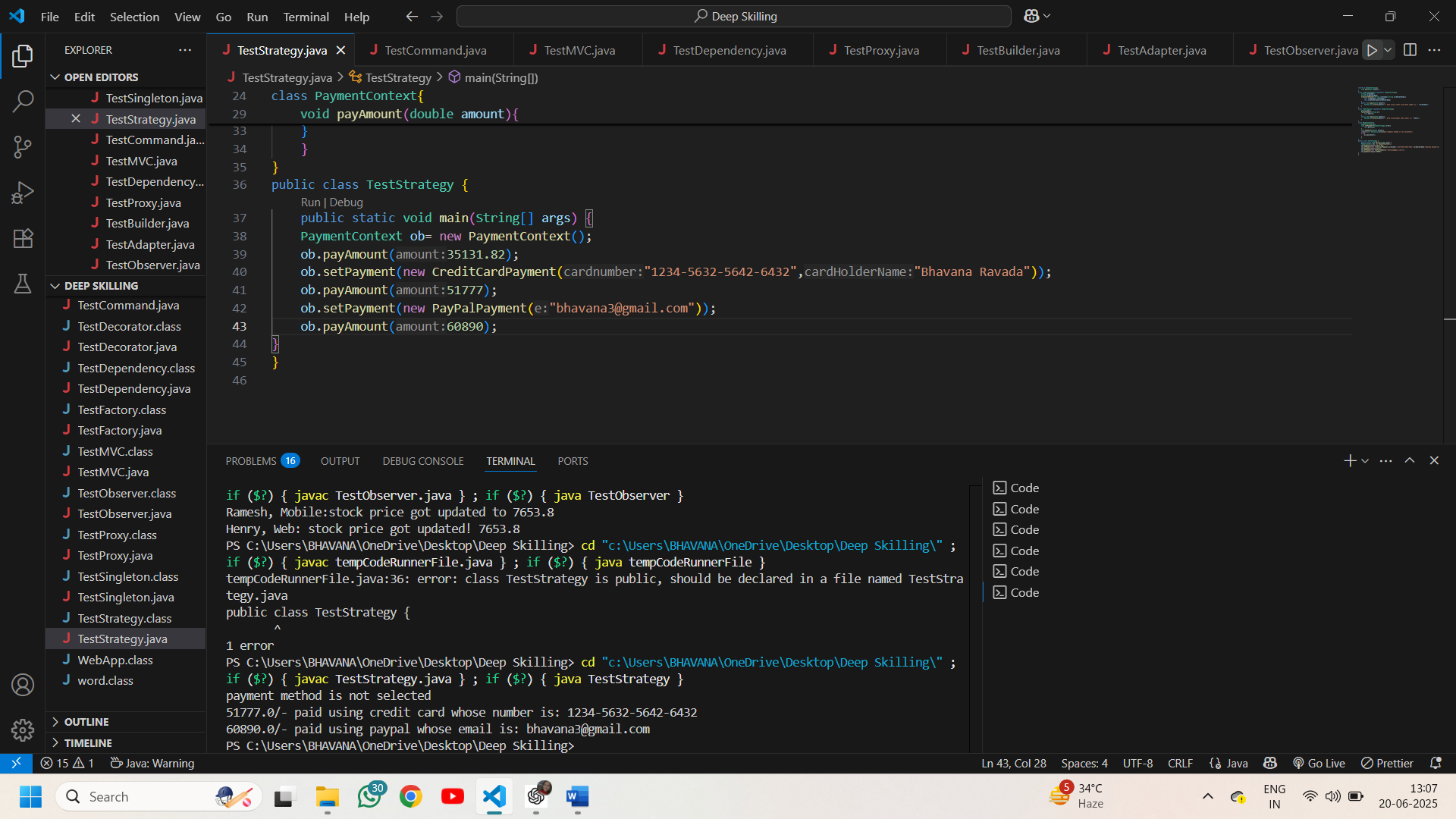
    ob.setPayment(new PayPalPayment("bhavana3@gmail.com"));

    ob.payAmount(60890);

}

}

**Output:**



**Exercise 9: Implementing the Command Pattern**

**Code:**

interface command{

    void execute();

}

class LightsOnCommand implements command{

    Light light;

    LightsOnCommand(Light light){

        this.light=light;

    }

     public void execute(){

       light.lightOn();

     }

}

class LightsOffCommand implements command{

    Light light;

    LightsOffCommand(Light light){

        this.light=light;

    }

     public void execute(){

       light.lightOff();

     }

}

class RemoteControl{

    command com;

    void setCommand(command com){

      this.com=com;

    }

    void pressButton(){

        if(com==null) System.out.println("No command set");

        else com.execute();

    }

}

class Light{

    void lightOn(){

      System.out.println("turn on the lights");

    }

    void lightOff(){

      System.out.println("turn off the lights");

    }

}

public class TestCommand {

    public static void main(String args[]){

        RemoteControl ob= new RemoteControl();

        ob.setCommand(new LightsOnCommand(new Light()));

        ob.pressButton();

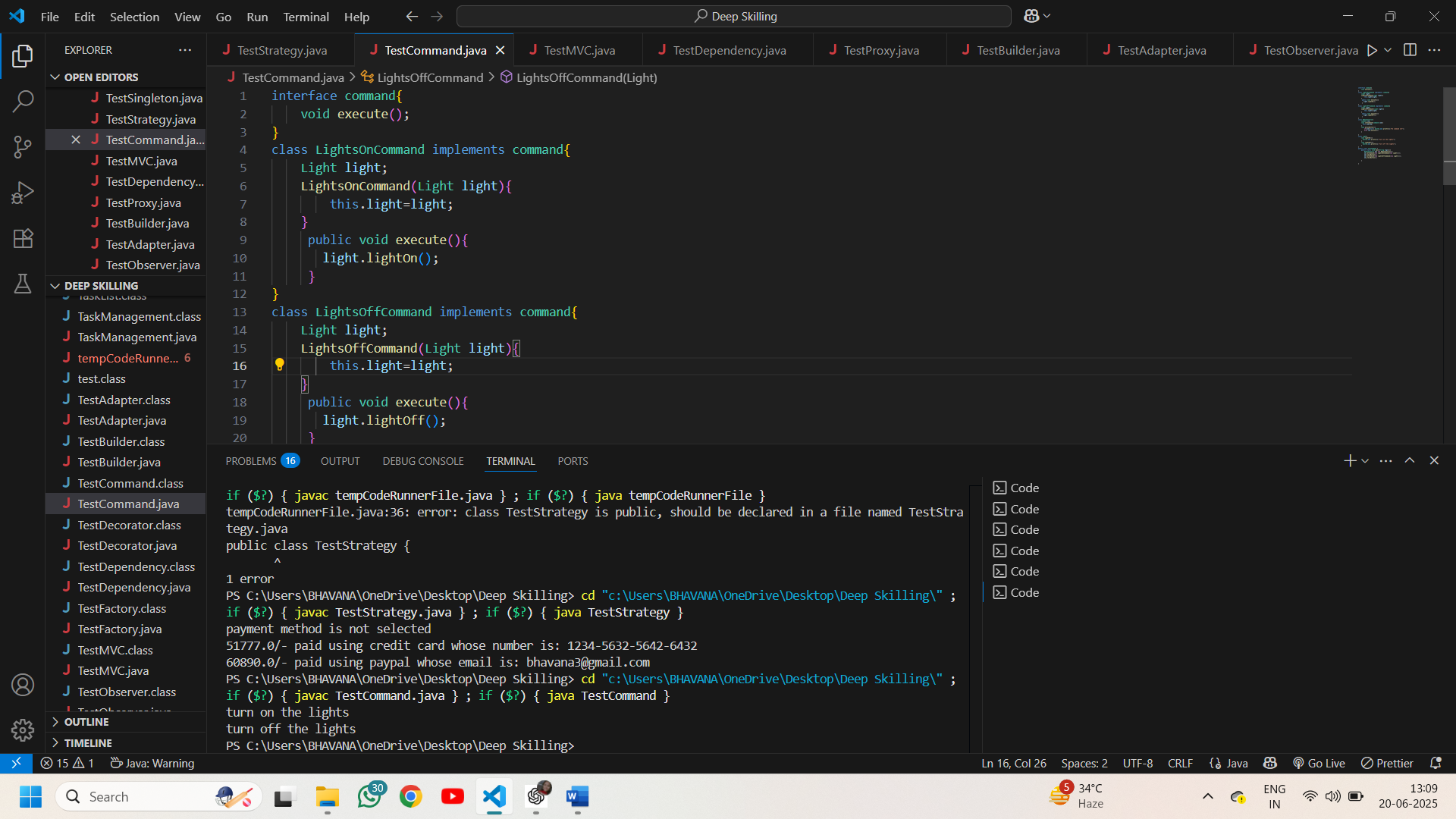
        ob.setCommand(new LightsOffCommand(new Light()));

        ob.pressButton();

    }

}

**Output:**



**Exercise 10: Implementing the MVC Pattern**

**Code:**

class Student{

    String name,id,grade;

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

        this.name= name;

        this.id=id;

        this.grade=grade;

    }

    void setName(String n){

        this.name=n;

    }

    void setId(String id){

        this.id=id;

    }

    void setGrade(String g){

        this.grade=g;

    }

    String getName(){ return name; }

    String getId(){   return id; }

    String getGrade(){ return grade;}

}

class StudentView{

    void displayStudentDetails(String name,String id,String grade){

        System.out.println("Student Details:");

        System.out.println("Name: "+name+" Id: "+id+" Grade: "+grade );

    }

}

class StudentController{

    Student stud;

    StudentView details;

    StudentController(Student stud,StudentView details){

        this.stud=stud;

        this.details=details;

    }

    void setStudentName(String name){

        stud.setName(name);

    }

    void setStudentId(String id){

        stud.setId(id);

    }

    void setStudentGrade(String grade){

        stud.setGrade(grade);

    }

    String getStudentName(){

        return stud.getName();

    }

    String getStudentId(){

        return stud.getId();

    }

    String getStudentGrade(){

        return stud.getGrade();

    }

    void updateView(){

        details.displayStudentDetails(stud.getName(), stud.getId(), stud.getGrade());

    }

}

public class TestMVC {

    public static void main(String args[]){

        StudentController ob= new StudentController(new Student("Bhavana","22NM1A05D3","A"), new StudentView());

        ob.updateView();

        ob.setStudentName("Vishnu priya");

        ob.setStudentGrade("A+");

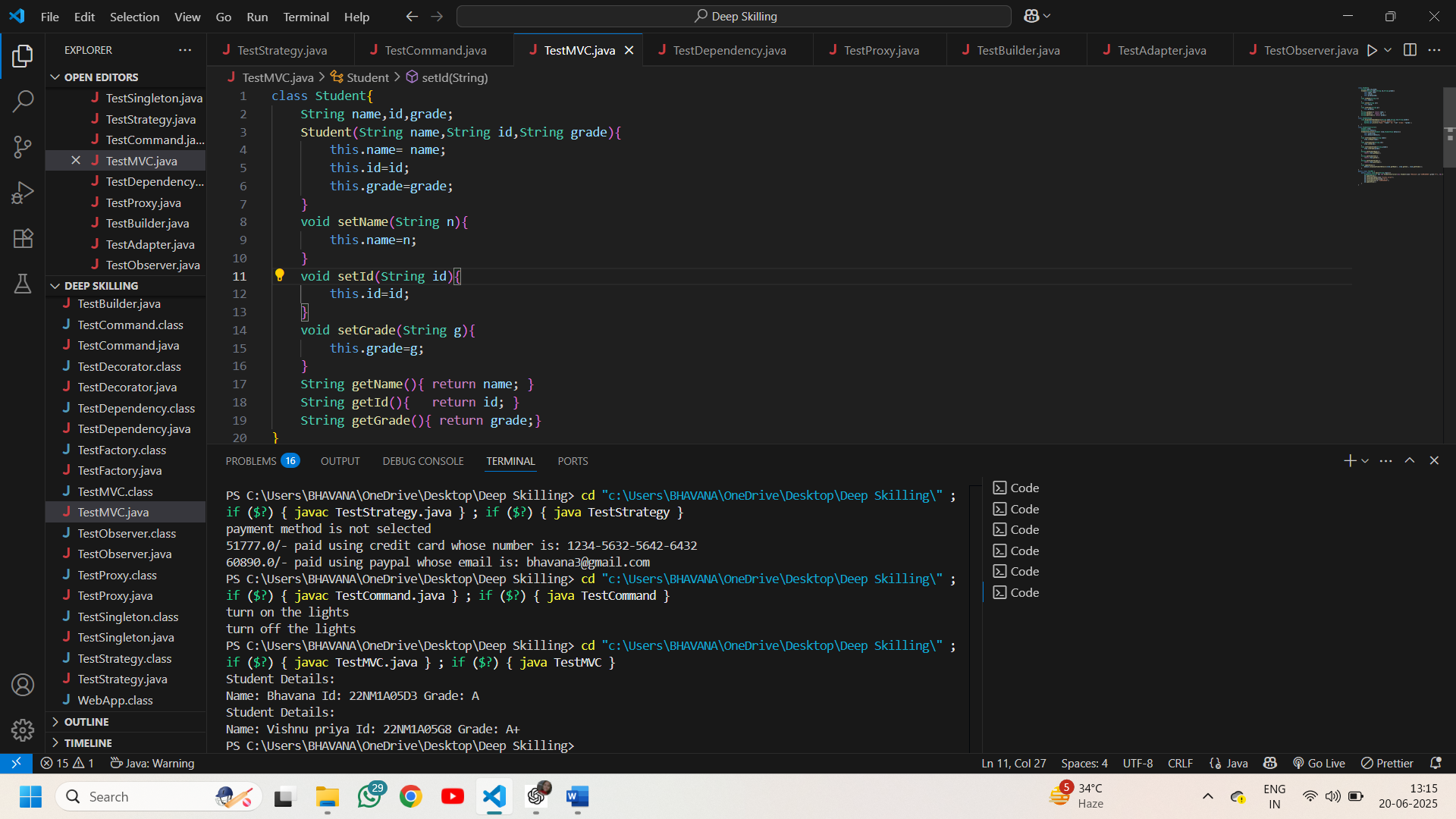
        ob.setStudentId("22NM1A05G8");

        ob.updateView();

    }

}

**Output:**



**Exercise 11: Implementing Dependency Injection**

**Code:**

import java.util.\*;

interface CustomerRepository{

    Customer findCustomerById(int id);

}

class CustomerRepositoryImpl implements CustomerRepository{

    Map<Integer,Customer> mp= new HashMap<>();

    CustomerRepositoryImpl(){

        mp.put(1,new Customer(1,"Bhavana Ravada","ravadabhavana3@gmail.com"));

        mp.put(2,new Customer(2,"Mounika Lanka","mouni23@gmail.com"));

    }

    public Customer findCustomerById(int id){

        return mp.get(id);

    }

}

class Customer{

    int id;

    String name,email;

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

        this.id=id;

        this.name=name;

        this.email=email;

    }

    public String toString(){

        return "id: "+id+" name: "+name+" email: "+email;

    }

}

class CustomerService{

    CustomerRepository repo;

    CustomerService(CustomerRepository repo){

        this.repo=repo;

    }

    void display(int id){

        Customer customer= repo.findCustomerById(id);

        if(customer==null) System.out.println("customer not found");

        else System.out.println("customer found "+customer);

    }

}

public class TestDependency {

    public static void main(String args[]){

    CustomerService ob= new CustomerService(new CustomerRepositoryImpl());

    ob.display(2);

    ob.display(4);

    }

}

**Output:**

