**Exercise 1: Implementing the Singleton Pattern**

**Logger.java**

package singleton;

public class Logger {

    private static Logger instance;

    private Logger() {

        System.out.println("Logger initialized.");

    }

    public static Logger getInstance() {

        if (instance == null) {

            instance = new Logger();

        }

        return instance;

    }

    public void log(String message) {

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

    }

}

**Main.java**

package singleton;

public class main {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        Logger logger2 = Logger.getInstance();

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

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

        if (logger1 == logger2) {

            System.out.println("Only one instance of Logger is used.");

        } else {

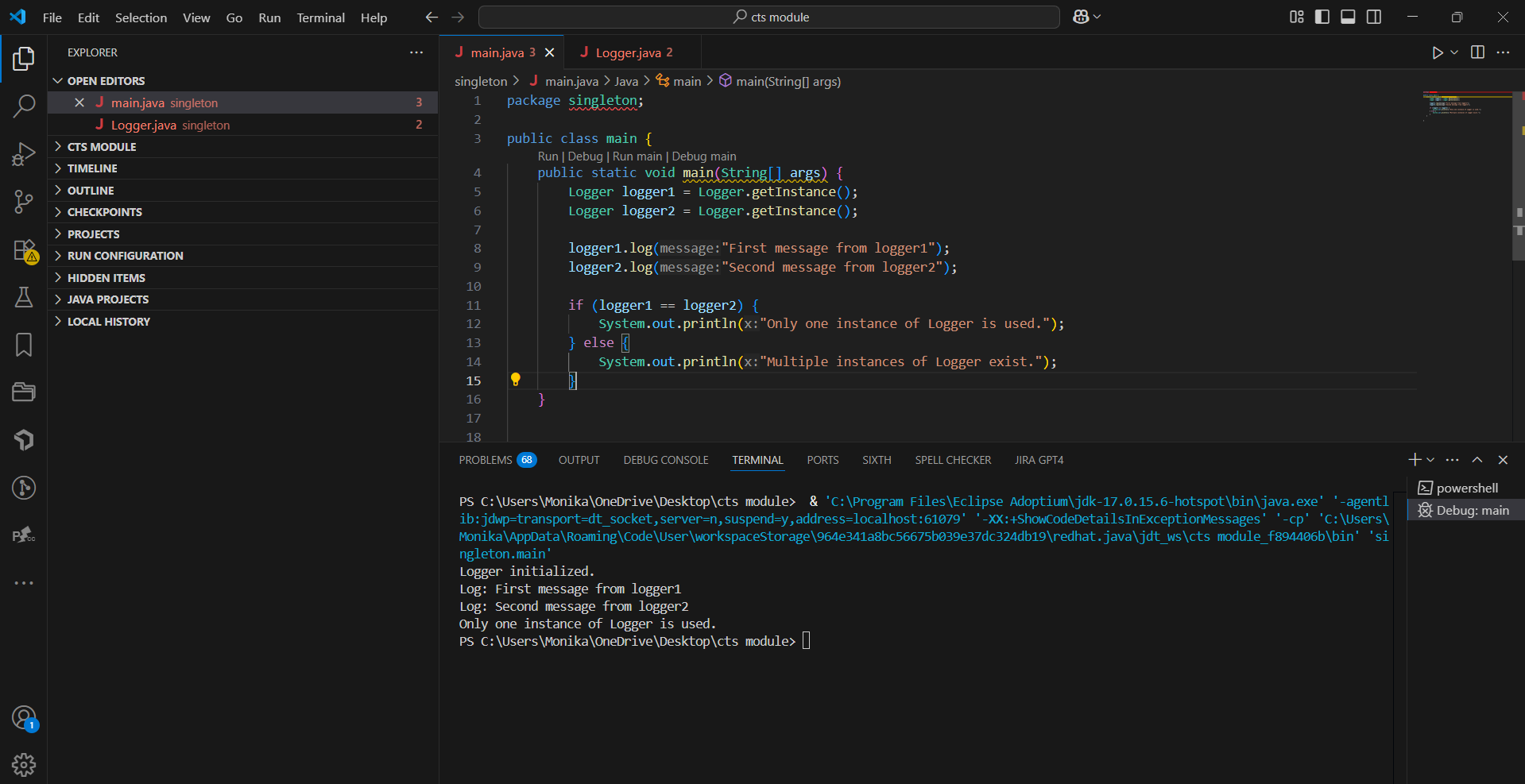
            System.out.println("Multiple instances of Logger exist.");

        }

    }

}

**Output:**



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

**Main.java**

interface Document {

void open();

}

class WordDocument implements Document {

public void open() {

System.out.println("Opening Word Document.");

}

}

class PdfDocument implements Document {

public void open() {

System.out.println("Opening PDF Document.");

}

}

class ExcelDocument implements Document {

public void open() {

System.out.println("Opening Excel Document.");

}

}

abstract class DocumentFactory {

public abstract Document createDocument();

}

class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

public class Main {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document word = wordFactory.createDocument();

word.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdf = pdfFactory.createDocument();

pdf.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

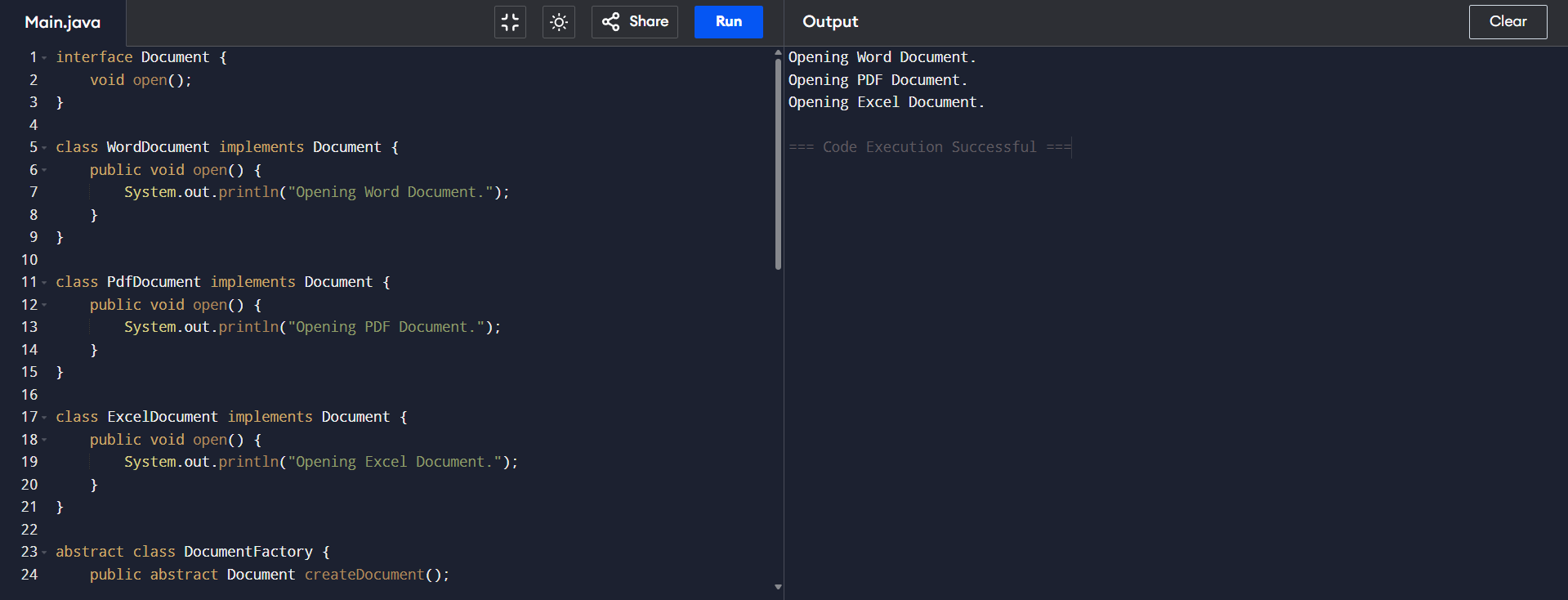
Document excel = excelFactory.createDocument();

excel.open();

}

}

**Output:**



**Exercise 3: Implementing the Builder Pattern**

**BuilderPattern.java**

class Computer {

    private final String cpu;

    private final int ram;

    private final int storage;

    private final boolean graphicsCard;

    private final String os;

    public static class Builder {

        private String cpu;

        private int ram;

        private int storage;

        private boolean graphicsCard;

        private String os;

        public Builder cpu(String c) { cpu = c; return this; }

        public Builder ram(int r) { ram = r; return this; }

        public Builder storage(int s) { storage = s; return this; }

        public Builder graphicsCard(boolean g) { graphicsCard = g; return this; }

        public Builder os(String o) { os = o; return this; }

        public Computer build() { return new Computer(this); }

    }

    private Computer(Builder b) {

        cpu = b.cpu;

        ram = b.ram;

        storage = b.storage;

        graphicsCard = b.graphicsCard;

        os = b.os;

    }

    public String toString() {

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

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

    }

}

class BuilderTest {

    public static void main(String[] args) {

        Computer pc = new Computer.Builder()

                        .cpu("i7")

                        .ram(16)

                        .storage(512)

                        .graphicsCard(true)

                        .os("Windows 11")

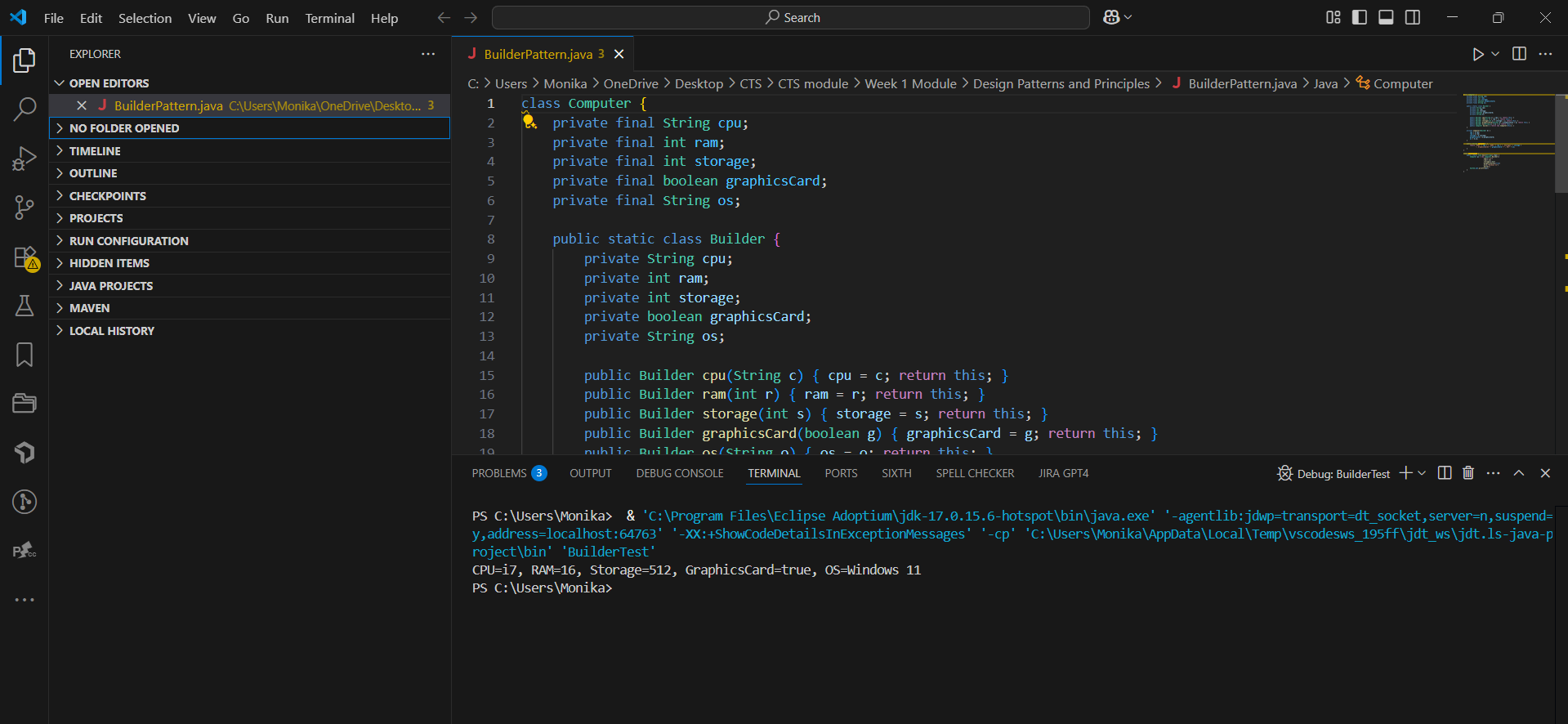
                        .build();

        System.out.println(pc);

    }

}

**Output:**

****

**Exercise 4: Implementing the Adapter Pattern**

**AdapterPattern.java**

interface PaymentProcessor {

    void pay(double amt);

}

class LegacyBank {

    public void makePayment(double a) {

        System.out.println("Paid via bank: " + a);

    } }

class BankAdapter implements PaymentProcessor {

    private LegacyBank bank = new LegacyBank();

    public void pay(double amt) {

        bank.makePayment(amt);

    } }

class AdapterPattern {

    public static void main(String[] args) {

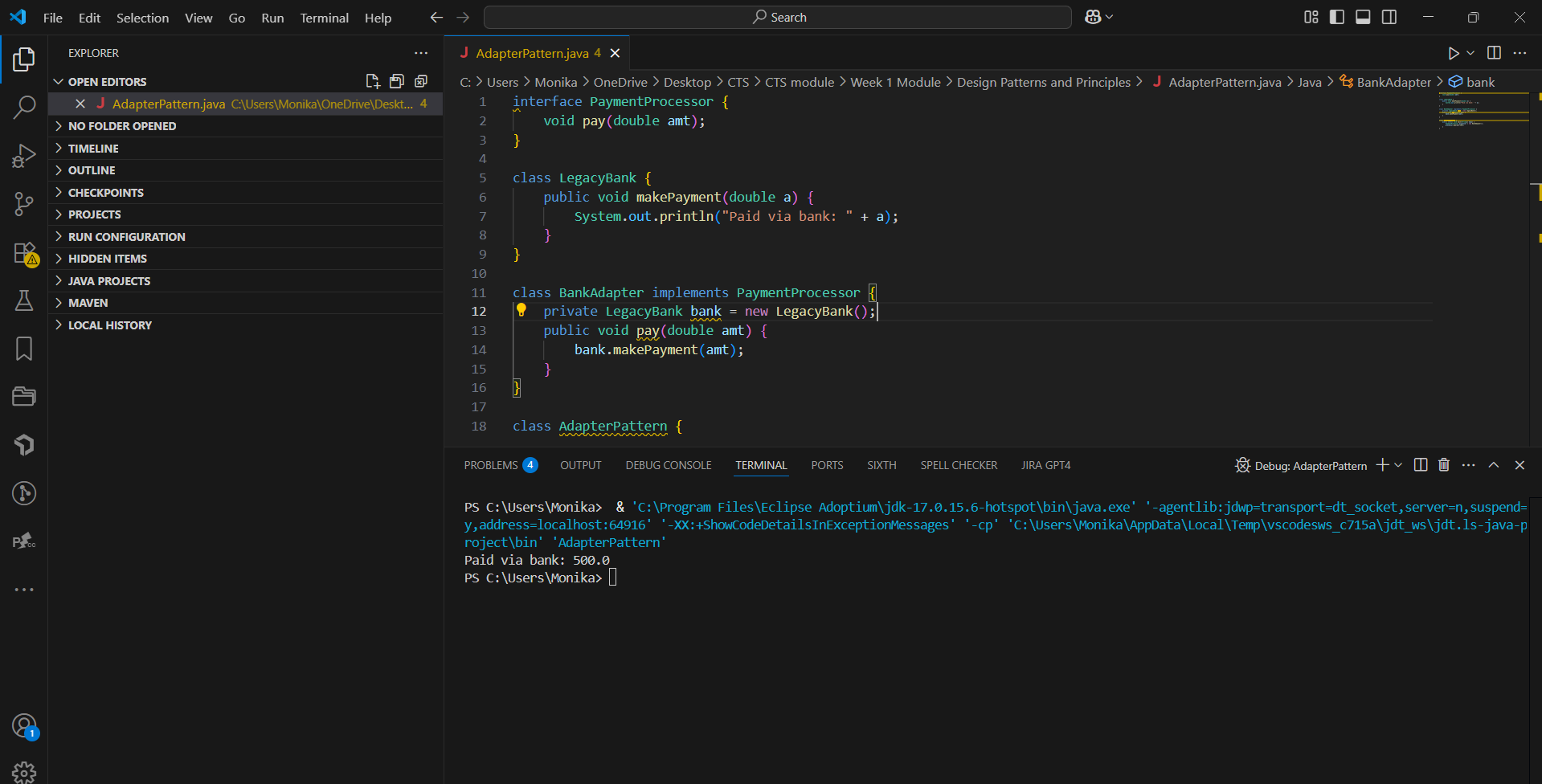
        PaymentProcessor processor = new BankAdapter();

        processor.pay(500);

    }

}

**Output:**



**Exercise 5: Implementing the Decorator Pattern**

**DecoratorPattern.java**

interface Notifier {

    void send(String msg);

}

class EmailNotifier implements Notifier {

    public void send(String msg) {

        System.out.println("Email: " + msg);

    }

}

class NotifierDecorator implements Notifier {

    protected Notifier wrap;

    public NotifierDecorator(Notifier n) {

        wrap = n;

    }

    public void send(String msg) {

        wrap.send(msg);

    }

}

class SMSDecorator extends NotifierDecorator {

    public SMSDecorator(Notifier n) {

        super(n);

    }

    public void send(String msg) {

        super.send(msg);

        System.out.println("SMS: " + msg);

    }

}

class DecoratorTest {

    public static void main(String[] args) {

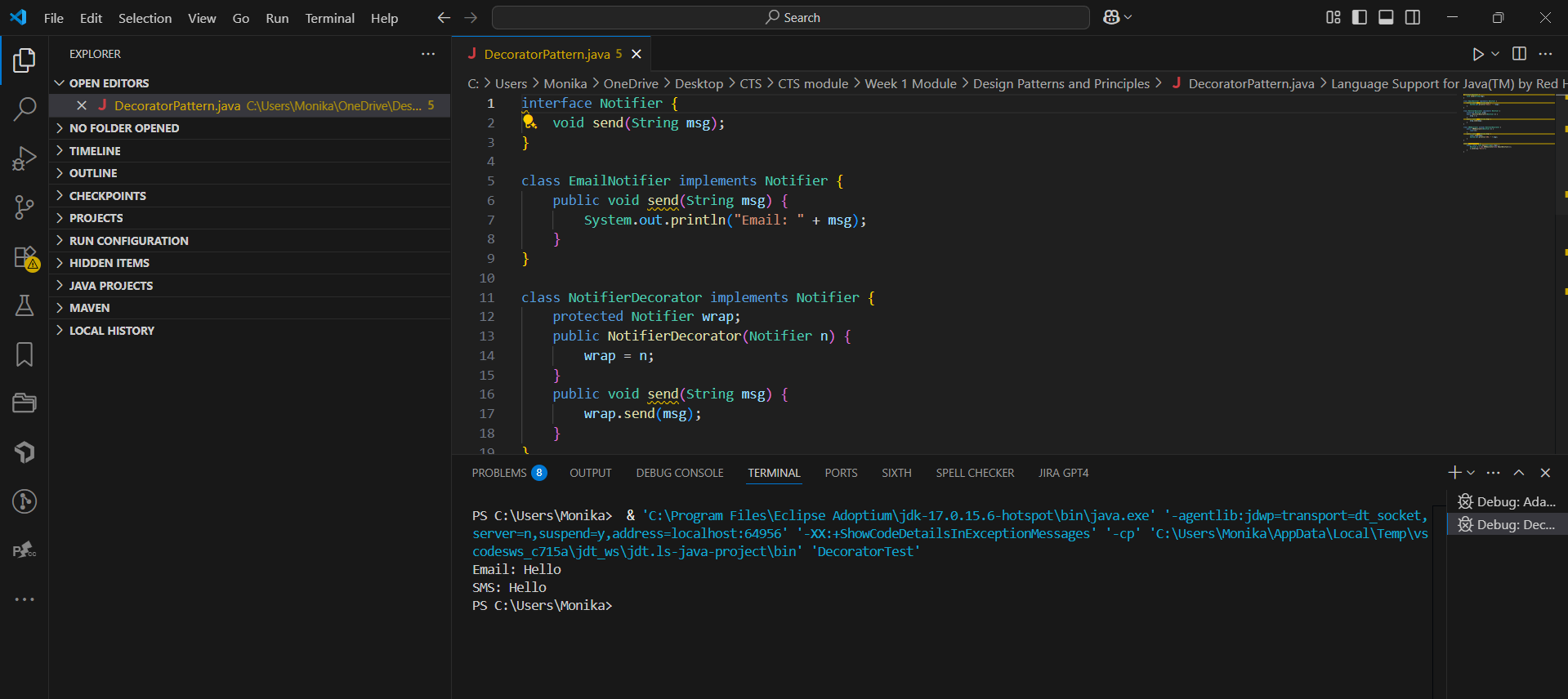
        Notifier n = new SMSDecorator(new EmailNotifier());

        n.send("Hello");

    }

}

**Output:**



**Exercise 6: Implementing the Proxy Pattern**

**ProxyPattern.java**

interface Image {

    void display();

}

class RealImage implements Image {

    private String file;

    public RealImage(String f) {

        file = f;

        load();

    }

    private void load() {

        System.out.println("Loading: " + file);

    }

    public void display() {

        System.out.println("Showing: " + file);

    }

}

class ProxyImage implements Image {

    private String file;

    private RealImage real;

    public ProxyImage(String f) {

        file = f;

    }

    public void display() {

        if (real == null) {

            real = new RealImage(file);

        }

        real.display();

    }

}

class ProxyTest {

    public static void main(String[] args) {

        Image img = new ProxyImage("photo.jpg");

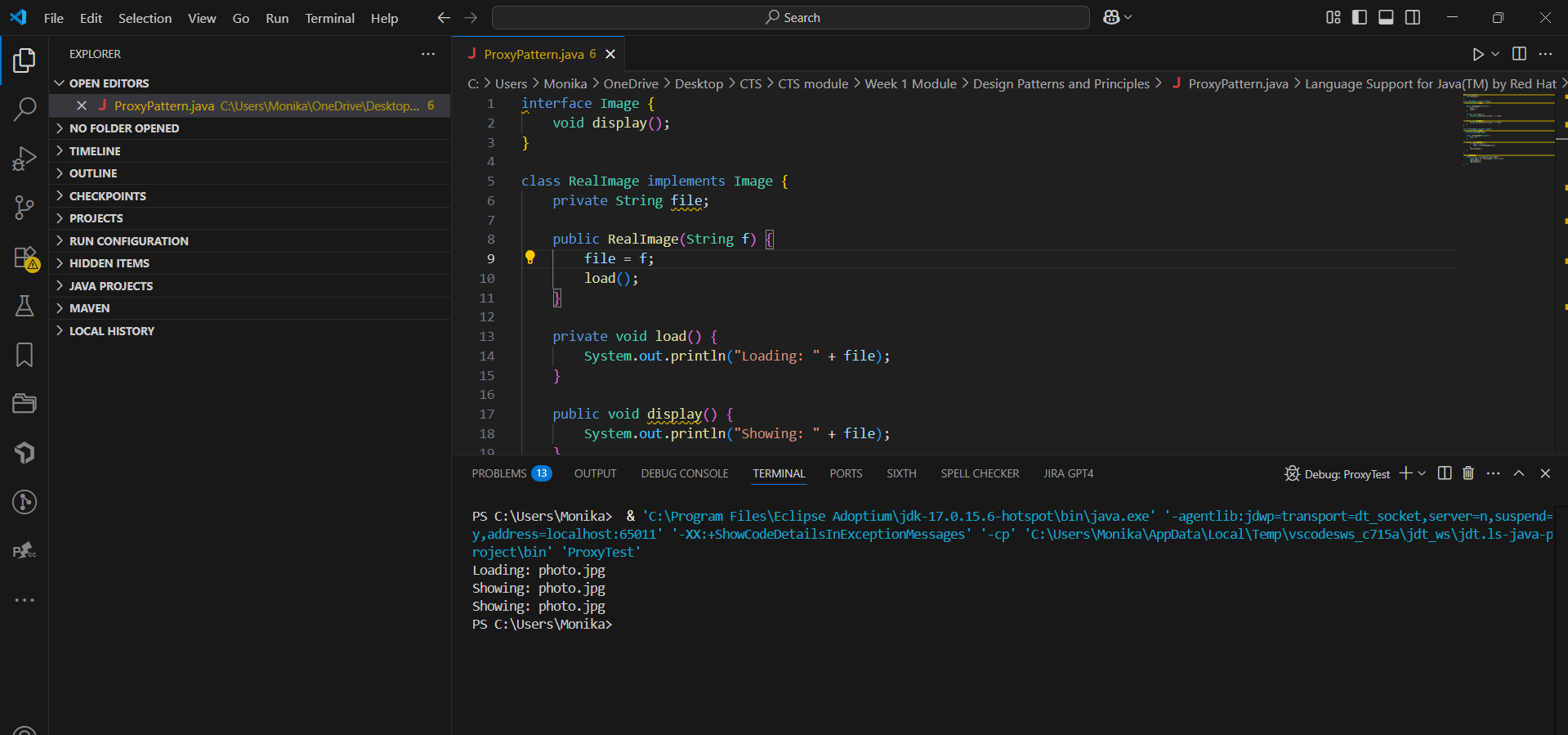
        img.display();

        img.display();

    }

}

**Output:**



**Exercise 7: Implementing the Observer Pattern**

**ObserverPattern.java**

import java.util.\*;

interface Observer {

    void update(String stock, double price);

}

interface Subject {

    void register(Observer o);

    void notifyObservers();

}

class StockMarket implements Subject {

    private String stock;

    private double price;

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

    public StockMarket(String s, double p) {

        stock = s;

        price = p;

    }

    public void register(Observer o) {

        obs.add(o);

    }

    public void changePrice(double p) {

        price = p;

        notifyObservers();

    }

    public void notifyObservers() {

        for (Observer o : obs) {

            o.update(stock, price);

        }

    }

}

class MobileApp implements Observer {

    public void update(String s, double p) {

        System.out.println("Mobile App - " + s + ": " + p);

    }

}

class ObserverTest {

    public static void main(String[] args) {

        StockMarket m = new StockMarket("AAPL", 150);

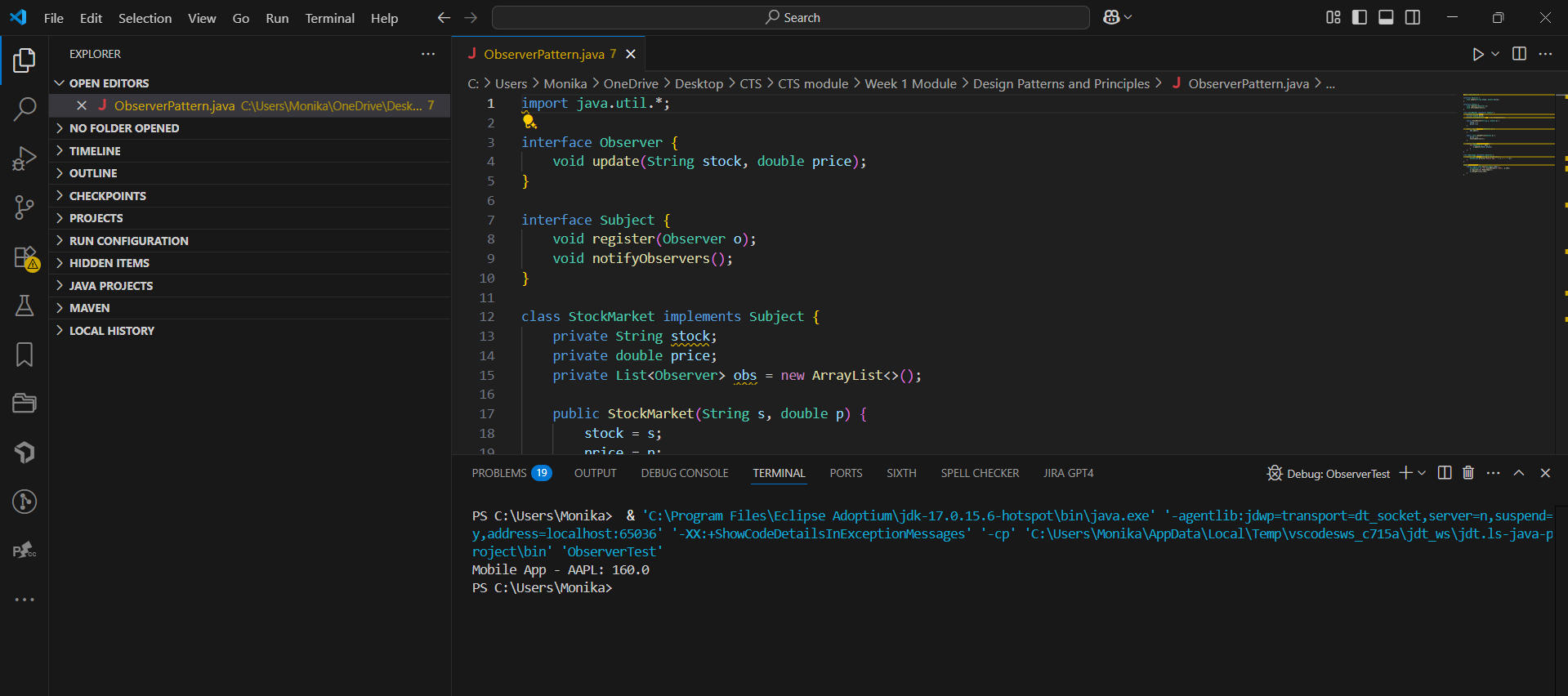
        m.register(new MobileApp());

        m.changePrice(160);

    }

}

**Output:**



**Exercise 8: Implementing the Strategy Pattern**

**StrategyPattern.java**

import java.util.Arrays;

interface Strategy {

    void sort(int[] arr);

}

class BubbleSort implements Strategy {

    public void sort(int[] arr) {

        for (int i = 0; i < arr.length - 1; i++)

            for (int j = 0; j < arr.length - i - 1; j++)

                if (arr[j] > arr[j + 1]) {

                    int tmp = arr[j];

                    arr[j] = arr[j + 1];

                    arr[j + 1] = tmp;

                }

    }

}

class SelectionSort implements Strategy {

    public void sort(int[] arr) {

        for (int i = 0; i < arr.length - 1; i++) {

            int min = i;

            for (int j = i + 1; j < arr.length; j++)

                if (arr[j] < arr[min])

                    min = j;

            int tmp = arr[min];

            arr[min] = arr[i];

            arr[i] = tmp;

        }

    }

}

class Context {

    private Strategy strat;

    public Context(Strategy s) {

        strat = s;

    }

    public void execute(int[] arr) {

        strat.sort(arr);

    }

}

class StrategyPattern {

    public static void main(String[] args) {

        int[] arr1 = {5, 2, 8, 1};

        int[] arr2 = {9, 4, 3, 7};

        Context bubbleContext = new Context(new BubbleSort());

        bubbleContext.execute(arr1);

        System.out.println("Bubble Sort: " + Arrays.toString(arr1));

        Context selectionContext = new Context(new SelectionSort());

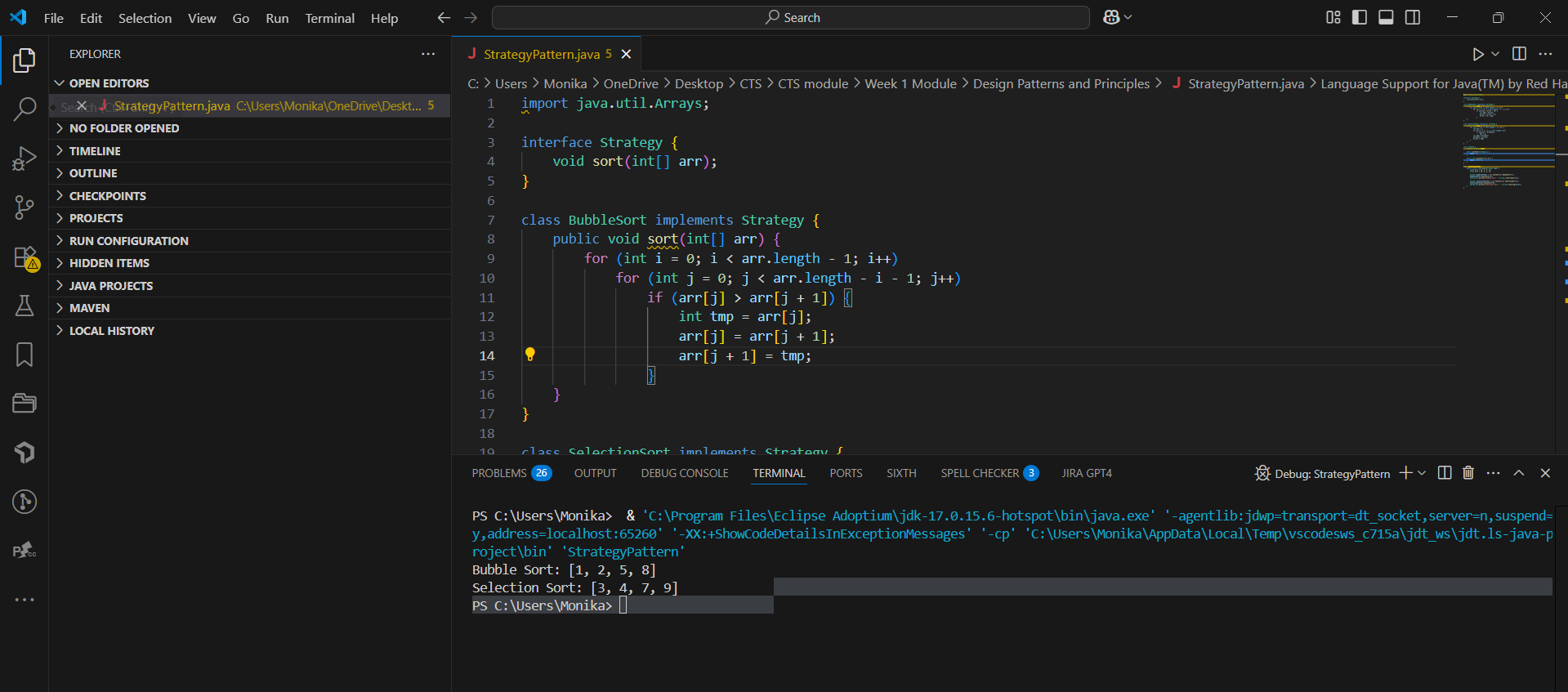
        selectionContext.execute(arr2);

        System.out.println("Selection Sort: " + Arrays.toString(arr2));

    }

}

**Output:**



**Exercise 9: Implementing the Command Pattern**

**CommandPattern.java**

interface Command {

    void execute();

}

class Light {

    public void on() {

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

    }

    public void off() {

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

    }

}

class LightOnCmd implements Command {

    private Light l;

    public LightOnCmd(Light l) {

        this.l = l;

    }

    public void execute() {

        l.on();

    }

}

class Remote {

    private Command c;

    public void set(Command c) {

        this.c = c;

    }

    public void press() {

        c.execute();

    }

}

class CommandPattern {

    public static void main(String[] args) {

        Light l = new Light();

        Remote r = new Remote();

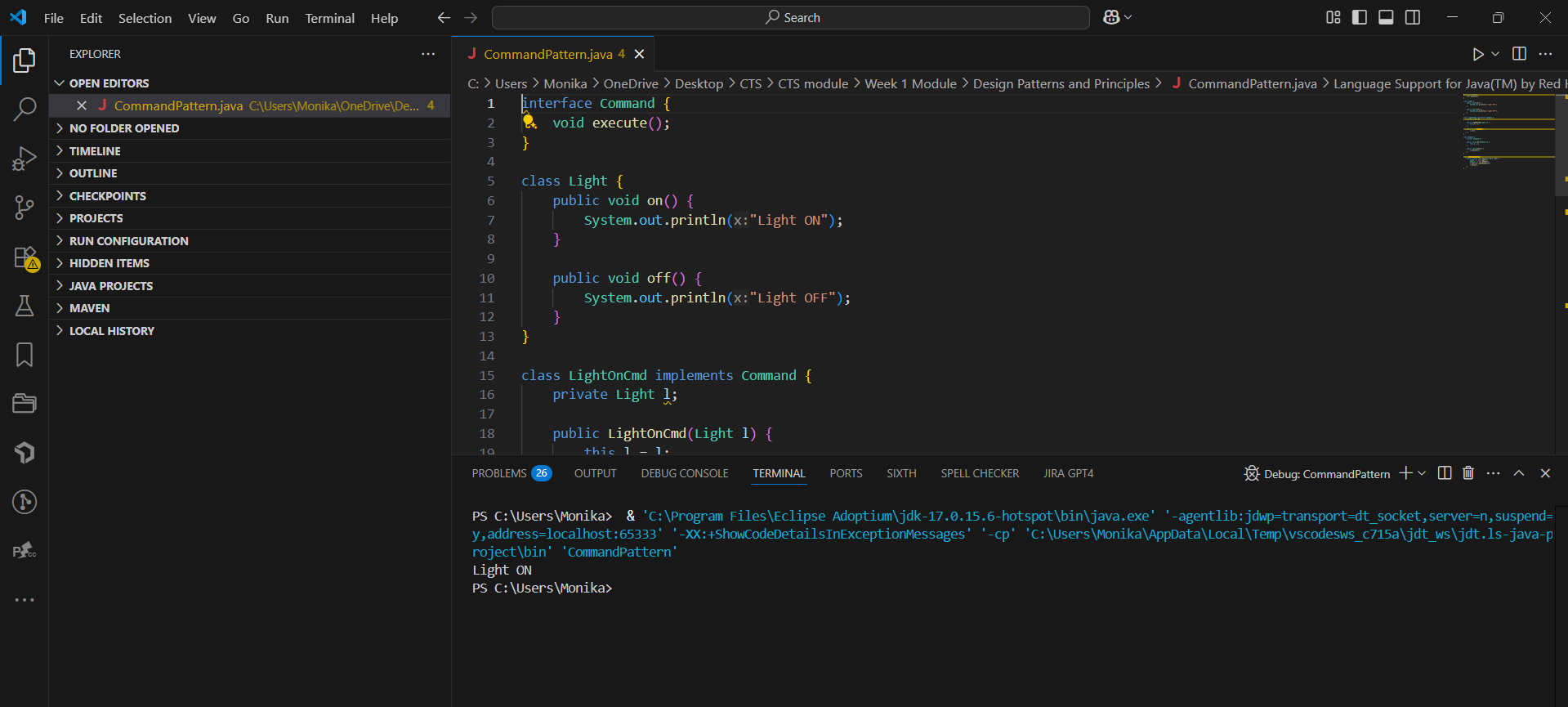
        r.set(new LightOnCmd(l));

        r.press();

    }

}

**Output:**

****

**Exercise 10: Implementing the MVC Pattern**

**MVCPattern.java**

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 String getId() { return id; }

    public String getGrade() { return grade; }

    public void setName(String name) { this.name = name; }

    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 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 student = new Student("Alice", "S001", "A");

        StudentView view = new StudentView();

        StudentController controller = new StudentController(student, view);

        controller.updateView();

        controller.setStudentName("Bob");

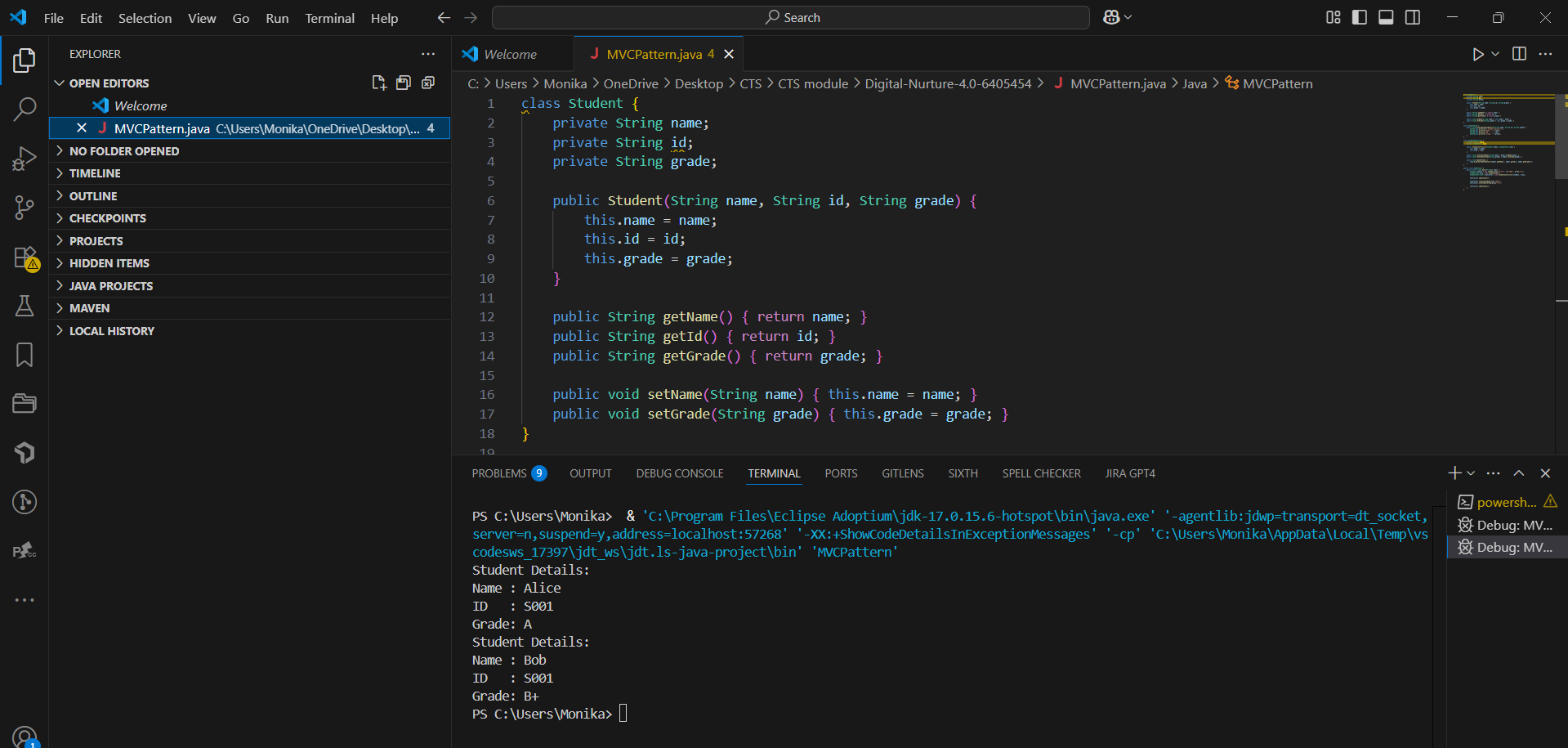
        controller.setStudentGrade("B+");

        controller.updateView();

    }

}

**Output:**

****

**Exercise 11: Implementing Dependency Injection**

**DependencyInjection.java**

interface CustomerRepository {

String findCustomerById(String id);

}

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

return "Customer{id='" + id + "', name='John Doe'}";

}

}

class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void displayCustomer(String id) {

String customer = repository.findCustomerById(id);

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

}

}

public class DependencyInjection {

public static void main(String[] args) {

CustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

service.displayCustomer("C123");

}

}

**Output:**

