**Design Patterns And Principles**

**EXERCISE:1 Implementing the Singleton Pattern**

**In Logger.Java**

package singleton;

public class TestLogger {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

logger1.log("First message");

logger2.log("Second message");

if (logger1 == logger2) {

System.out.println("Both logger1 and logger2 are the same instance.");

} else {

System.out.println("Different instances of Logger were created.");

}}}

**In TestLogger**

package singleton;

public class TestLogger {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

logger1.log("First message");

logger2.log("Second message");

if (logger1 == logger2) {

System.out.println("Both logger1 and logger2 are the same instance.");

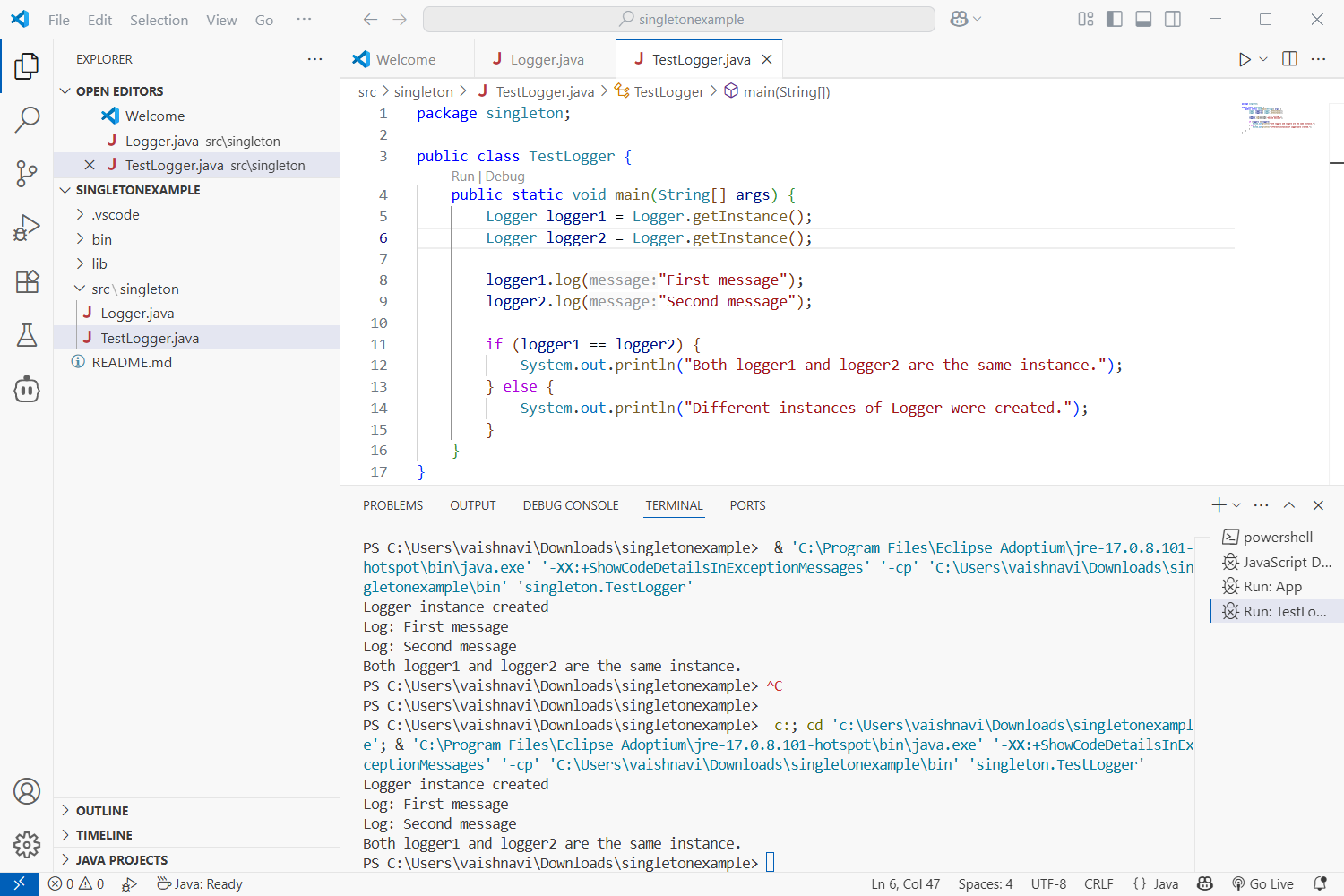
}

else

System.out.println("Different instances of Logger were created.");

}}}

**Output**

****

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

**Document Interface**

public interface Document {

    void open();

}

**Create Concrete Document Classes**

public class WordDocument implements Document {

    public void open() {

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

    }

}

public class PdfDocument implements Document {

public void open() {

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

}

}

public class ExcelDocument implements Document {

public void open() {

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

}

}

**Implement the Factory Method**

public class PdfDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new PdfDocument();

    }

}

public class excelDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new excelDocument();

    }

}

**Test the Factory Method**

public class Main {

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document wordDoc = wordFactory.createDocument();

        wordDoc.open();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdfDoc = pdfFactory.createDocument();

        pdfDoc.open();

        DocumentFactory excelFactory = new excelDocumentFactory();

        Document excelDoc = excelFactory.createDocument();

        excelDoc.open();

    }

}

public class PdfDocument implements Document {

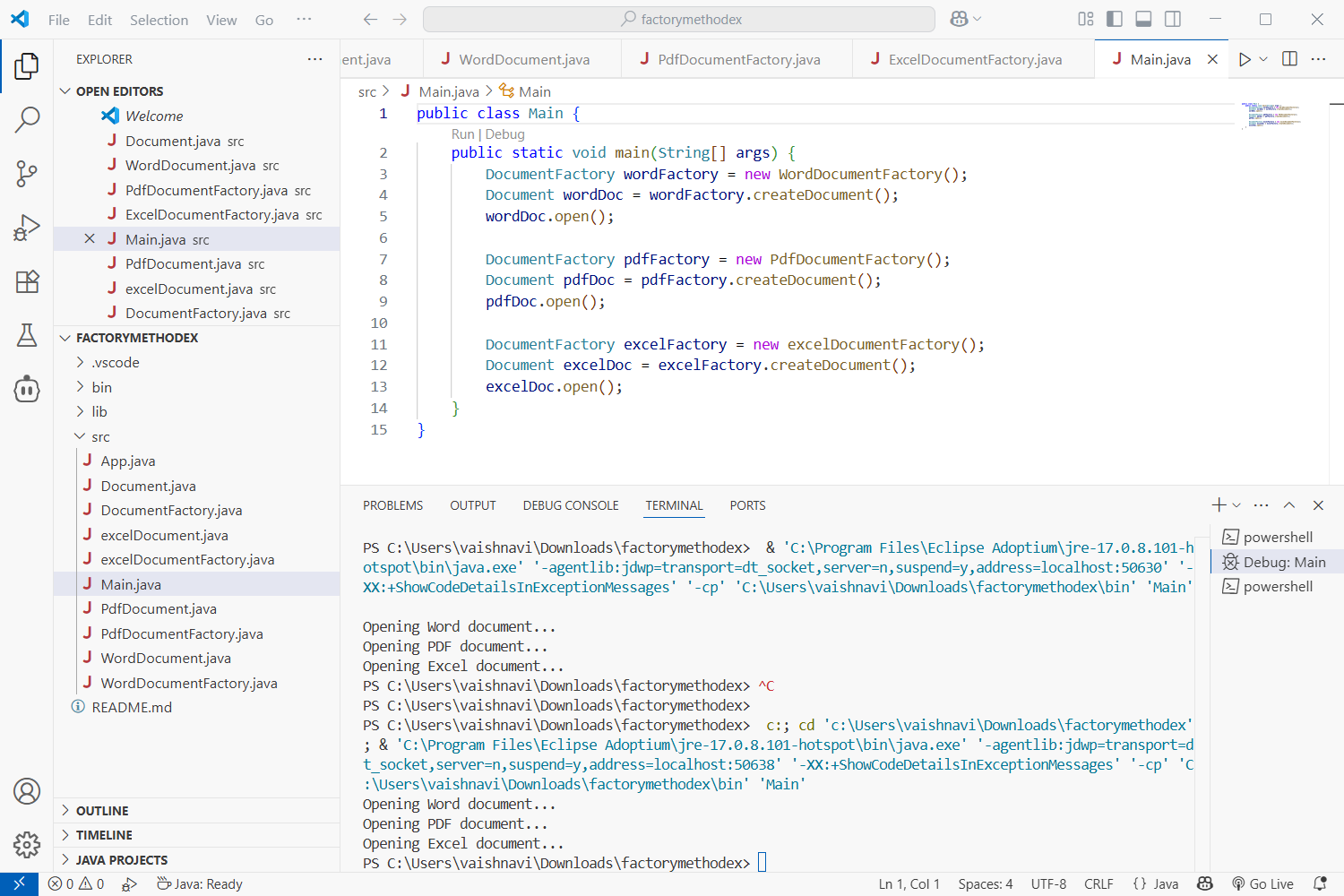
    public void open() {

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

    }

}

**Output**

****

**Exercise 3: Implementing the Builder Pattern**

**Define the Product Class**

public class Computer {

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

// Private constructor

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

}

// Static nested Builder class

public static class Builder {

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

public Builder(String CPU, String RAM) {

this.CPU = CPU;

this.RAM = RAM;

}

public Builder setStorage(String storage) {

this.storage = storage;

return this;

}

public Builder setGraphicsCard(String graphicsCard) {

this.graphicsCard = graphicsCard;

return this;

}

public Computer build() {

return new Computer(this);

}

}

// Display method

public void displayConfiguration() {

System.out.println("CPU: " + CPU);

System.out.println("RAM: " + RAM);

System.out.println("Storage: " + (storage != null ? storage : "Not included"));

System.out.println("Graphics Card: " + (graphicsCard != null ? graphicsCard : "Not included"));

System.out.println("-----------");

}

}

**Create a Test Class**

public class Main {

public static void main(String[] args) {

// Basic Computer

Computer basicComputer = new Computer.Builder("Intel i5", "8GB").build();

basicComputer.displayConfiguration();

// Gaming Computer

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

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 4080")

.build();

gamingComputer.displayConfiguration();

// Workstation

Computer workstation = new Computer.Builder("AMD Ryzen 9", "64GB")

.setStorage("2TB HDD")

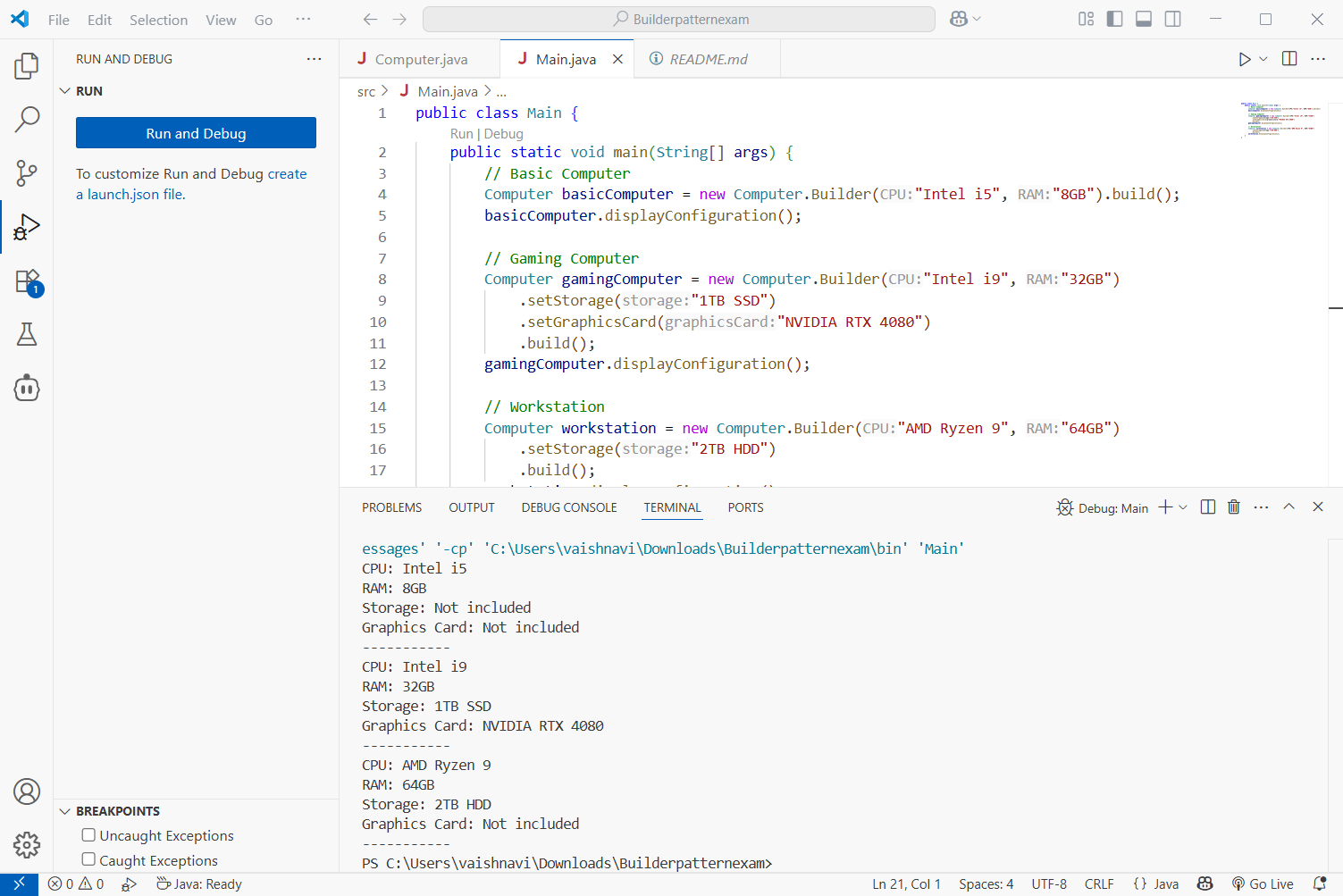
.build();

workstation.displayConfiguration();

}

}

**Output**



**Exercise 4: Implementing the Adapter Pattern**

**Target Interface**

public interface PaymentProcessor {

void processPayment(double amount);

}

**Adaptee Classes**

public class StripeGateway {

public void makeStripePayment(double amount) {

System.out.println("Processing payment with Stripe: $" + amount);

}

}

public class PayPalGateway {

public void sendPayment(double amount) {

System.out.println("Processing payment with PayPal: $" + amount);

}

}

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe;

}

public void processPayment(double amount) {

stripe.makeStripePayment(amount);

}

}

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway paypal;

public PayPalAdapter(PayPalGateway paypal) {

this.paypal = paypal;

}

public void processPayment(double amount) {

paypal.sendPayment(amount);

}

}

**Test the Adapter Implementation**

public class Main {

public static void main(String[] args) {

PaymentProcessor stripeProcessor = new StripeAdapter(new StripeGateway());

stripeProcessor.processPayment(100.0);

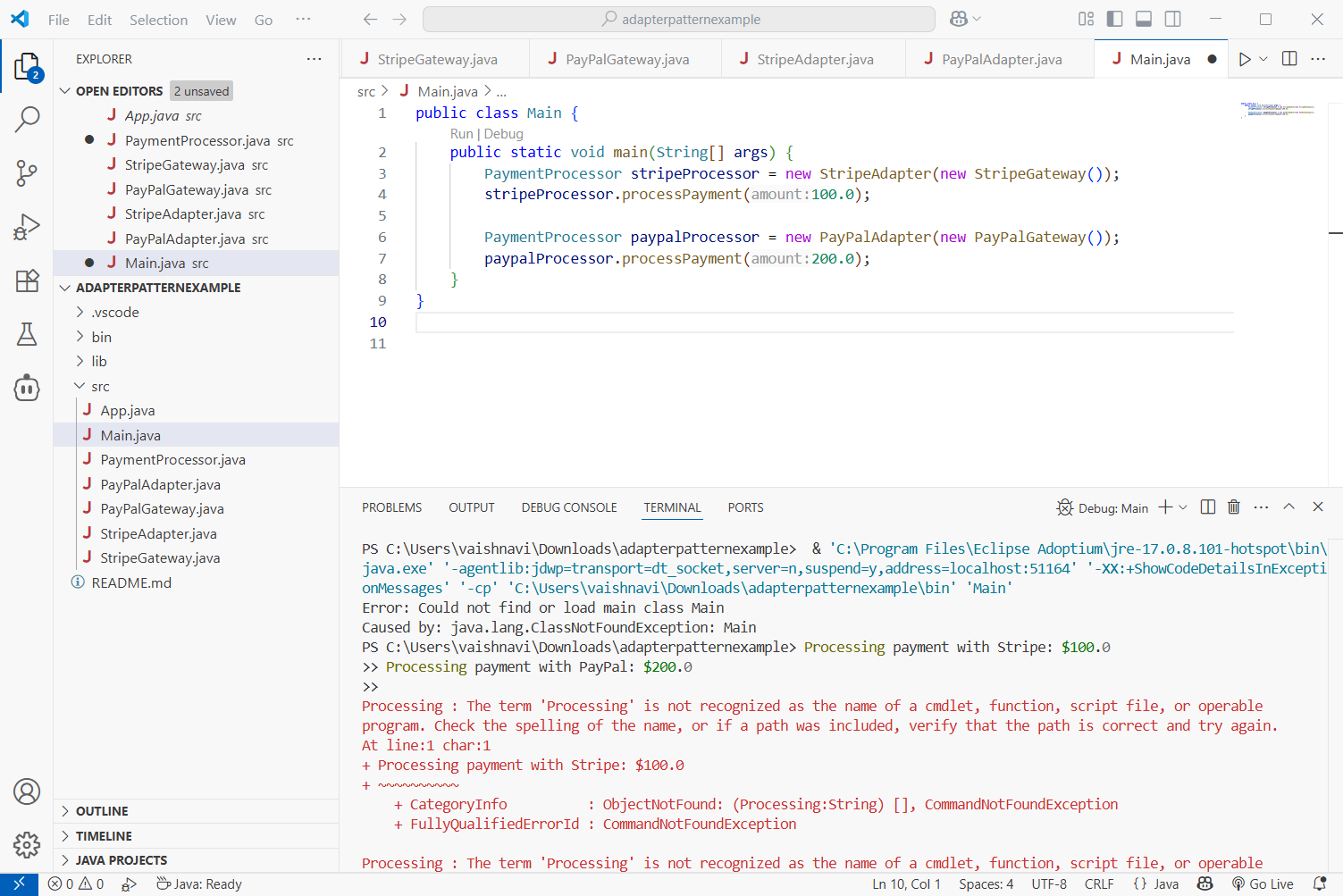
PaymentProcessor paypalProcessor = new PayPalAdapter(new PayPalGateway());

paypalProcessor.processPayment(200.0);

}

}

**Output**



**Exercise 5: Implementing the Decorator Pattern**

**Component Interface**

public interface Notifier {

void send(String message);

}

**Concrete Component**

public class EmailNotifier implements Notifier {

public void send(String message) {

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

}}

**Decorator Classes**

public abstract class NotifierDecorator implements Notifier {

protected Notifier wrappee;

public NotifierDecorator(Notifier notifier) {

this.wrappee = notifier;

}

public void send(String message) {

wrappee.send(message);

}

}

public class SMSNotifierDecorator extends NotifierDecorator {

public 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);

}

}

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

sendSlack(message);

}

private void sendSlack(String message) {

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

}

}

**Test the Decorator Implementation**

public class Main {

public static void main(String[] args) {

// Base notifier

Notifier notifier = new EmailNotifier();

// Add SMS notification

notifier = new SMSNotifierDecorator(notifier);

// Add Slack notification

notifier = new SlackNotifierDecorator(notifier);

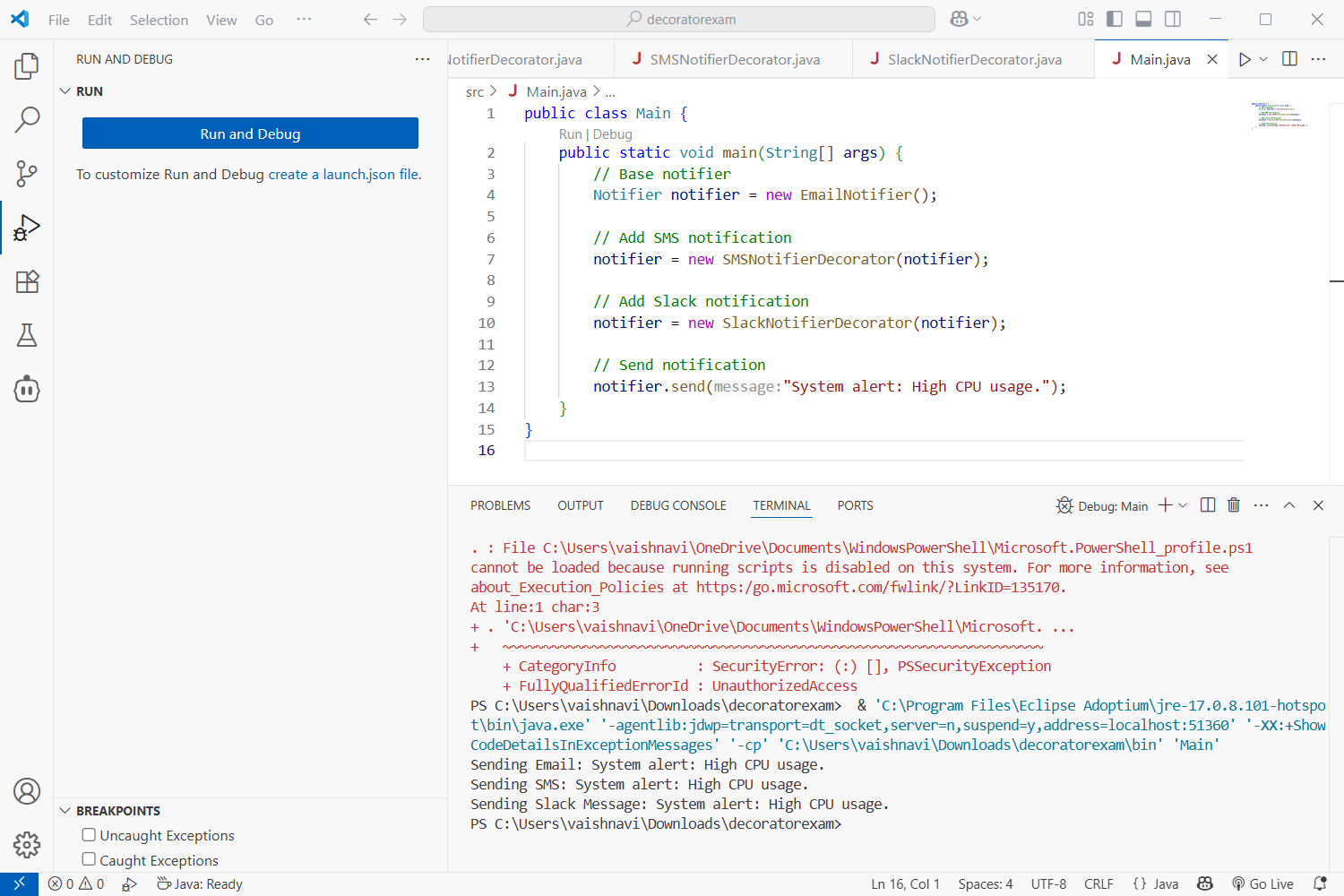
// Send notification

notifier.send("System alert: High CPU usage.");

}

}

**Output**

****

**Exercise 6: Implementing the Proxy Pattern**

**Image.java**

public interface Image {

void display();

}

**RealImage.java**

public class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromRemoteServer();

}

private void loadFromRemoteServer() {

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

}

@Override

public void display() {

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

}

}

**ProxyImage.java**

public class ProxyImage implements Image {

private RealImage realImage;

private String filename;

public ProxyImage(String filename) {

this.filename = filename;

}

@Override

public void display() {

if (realImage == null) {

realImage = new RealImage(filename);

}

realImage.display();

}

}

**ProxyPatternTest.java**

public class ProxyPatternTest {

public static void main(String[] args) {

Image image1 = new ProxyImage("image1.jpg");

Image image2 = new ProxyImage("image2.jpg");

// Image is loaded only when display is called

System.out.println("Calling display on image1 for the first time:");

image1.display();

System.out.println("\nCalling display on image1 again:");

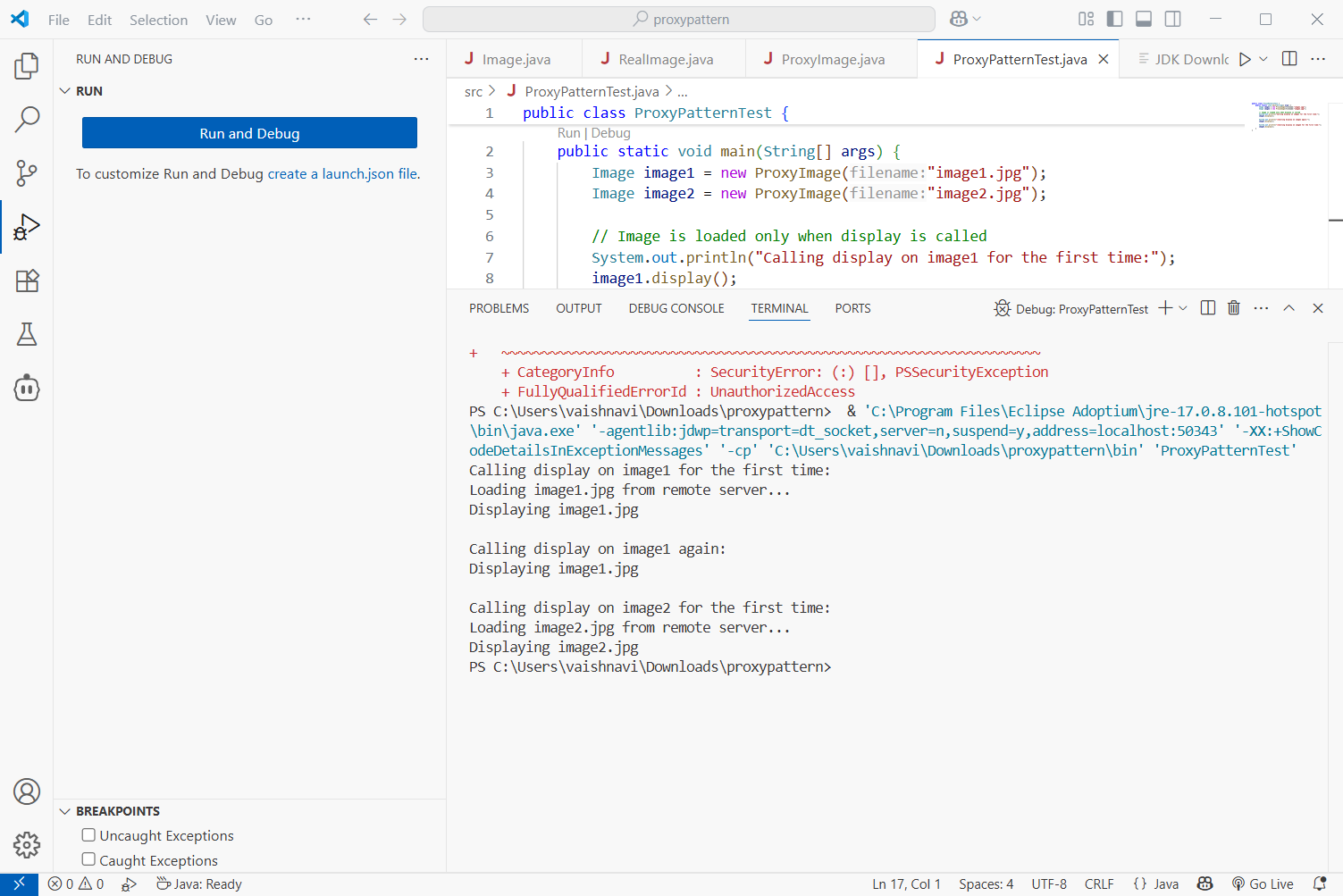
image1.display();

System.out.println("\nCalling display on image2 for the first time:");

image2.display();

}

}



**Exercise 7: Implementing the Observer Pattern**

**Stock.java**

public interface Stock {

void registerObserver(Observer observer);

void removeObserver(Observer observer);

void notifyObservers();

}

**StockMarket.java**

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

private List<Observer> observers;

private double stockPrice;

public StockMarket() {

observers = new ArrayList<>();

}

@Override

public void registerObserver(Observer observer) {

observers.add(observer);

}

@Override

public void removeObserver(Observer observer) {

observers.remove(observer);

}

@Override

public void notifyObservers() {

for (Observer observer : observers) {

observer.update(stockPrice);

}

}

public void setStockPrice(double price) {

System.out.println("Stock price updated to $" + price);

this.stockPrice = price;

notifyObservers();

}

}

**Observer.java**

public interface Observer {

void update(double stockPrice);

}

**MobileApp.java**

public class MobileApp implements Observer {

private String name;

public MobileApp(String name) {

this.name = name;

}

@Override

public void update(double stockPrice) {

System.out.println("MobileApp " + name + " received stock price update: $" + stockPrice);

}

}

**WebApp.java**

public class WebApp implements Observer {

private String name;

public WebApp(String name) {

this.name = name;

}

@Override

public void update(double stockPrice) {

System.out.println("WebApp " + name + " received stock price update: $" + stockPrice);

}

}

**ObserverPatternTest.java**

public class ObserverPatternTest {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobileApp1 = new MobileApp("Alpha");

Observer webApp1 = new WebApp("Beta");

stockMarket.registerObserver(mobileApp1);

stockMarket.registerObserver(webApp1);

stockMarket.setStockPrice(150.25);

stockMarket.setStockPrice(155.75);

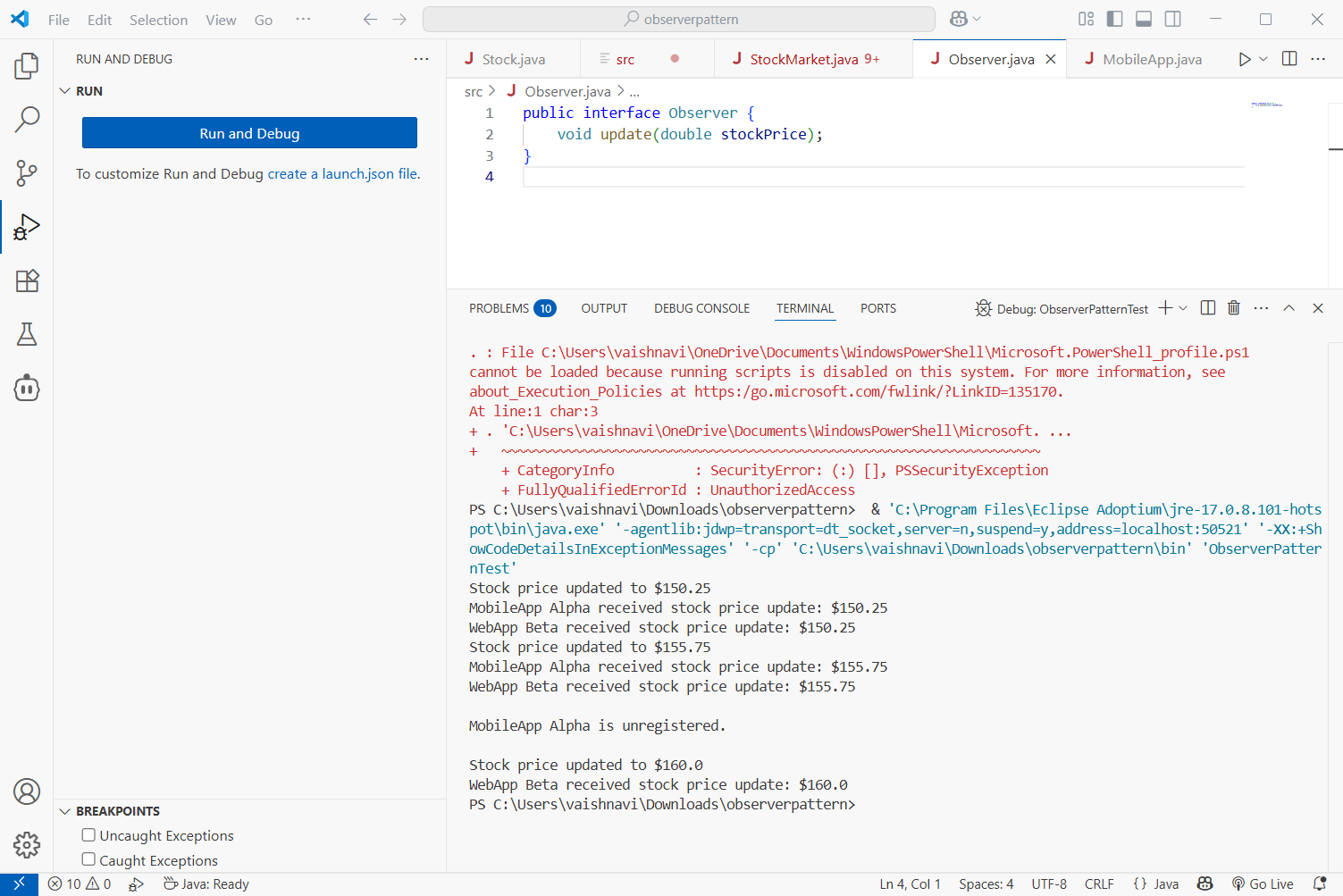
stockMarket.removeObserver(mobileApp1);

System.out.println("\nMobileApp Alpha is unregistered.\n");

stockMarket.setStockPrice(160.00);

}

}



**Exercise 8: Implementing the Strategy Pattern**

**PaymentStrategy.java**

public interface PaymentStrategy {

void pay(double amount);

}

**CreditCardPayment.java**

public 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);

}

**}**

**PayPalPayment.java**

public 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);

}

**}**

**PaymentContext.java**

public class PaymentContext {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void payAmount(double amount) {

if (paymentStrategy == null) {

System.out.println("No payment strategy selected!");

return;

}

paymentStrategy.pay(amount);

}

}

**StrategyPatternTest.java**

public class StrategyPatternTest {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

// Use Credit Card

context.setPaymentStrategy(new CreditCardPayment("1234-5678-9876-5432"));

context.payAmount(100.50);

// Switch to PayPal

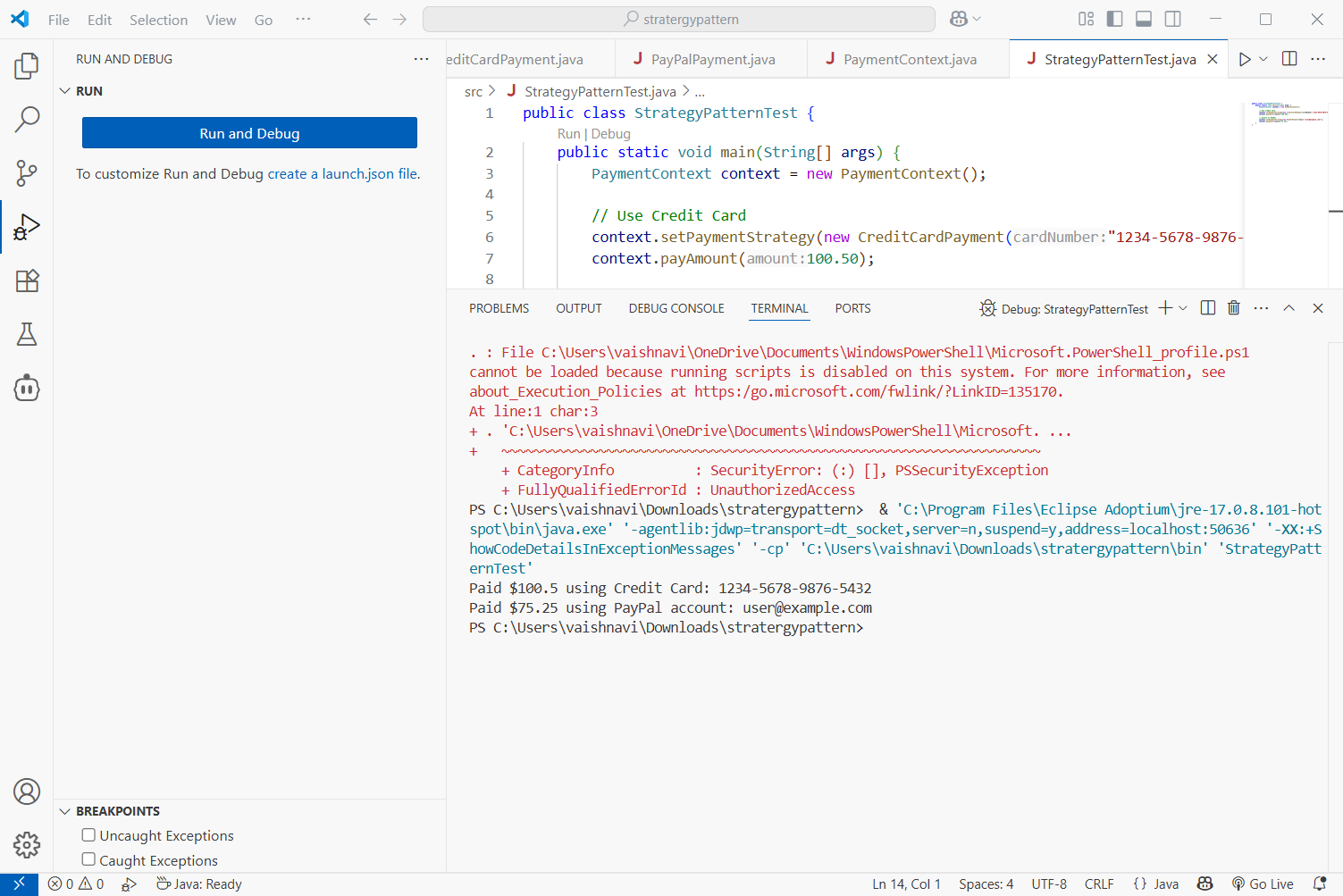
context.setPaymentStrategy(new PayPalPayment("user@example.com"));

context.payAmount(75.25);

}

}

**Output**



**Exercise 9: Implementing the Command Pattern**

**Command.java**

public interface Command {

void execute();

}

**LightOnCommand.java**

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

**LightOffCommand.java**

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

**RemoteControl.java**

public 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 assigned.");

}

}

}

**Light.java**

public class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

**CommandPatternTest.java**

public class CommandPatternTest {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

System.out.println("Turning the light ON:");

remote.setCommand(lightOn);

remote.pressButton();

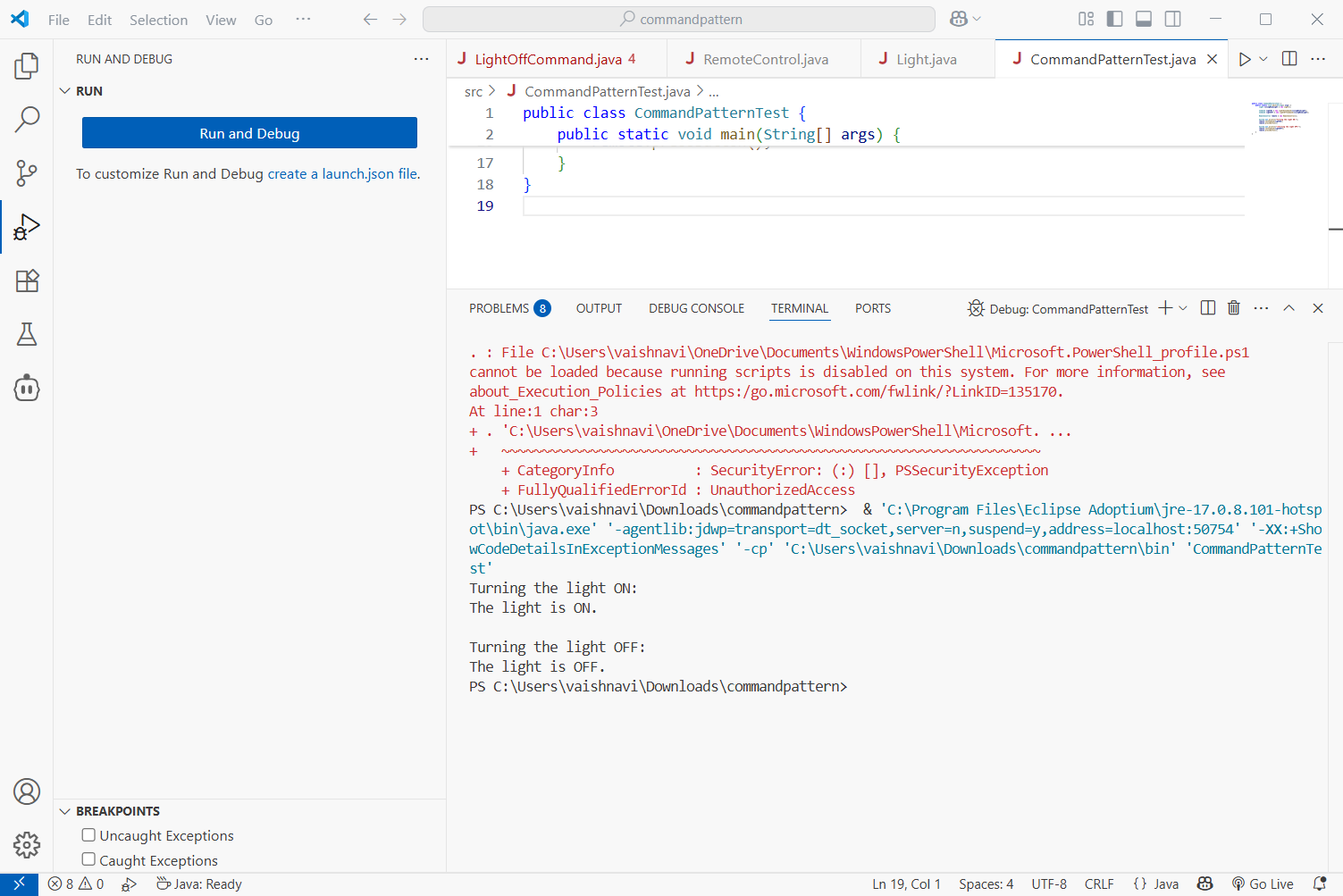
System.out.println("\nTurning the light OFF:");

remote.setCommand(lightOff);

remote.pressButton();

}

}



**Exercise 10: Implementing the MVC Pattern**

**Student.java**

public 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;

}

// Getters and setters

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;

}

}

**StudentView.java**

public 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);

}

}

**StudentController.java**

public 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 String getStudentName() {

return model.getName();

}

public void setStudentId(String id) {

model.setId(id);

}

public String getStudentId() {

return model.getId();

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public String getStudentGrade() {

return model.getGrade();

}

public void updateView() {

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

}

}

**MVCPatternTest.java**

public class MVCPatternTest {

public static void main(String[] args) {

// Model

Student student = new Student("John Doe", "ST123", "A");

// View

StudentView view = new StudentView();

// Controller

StudentController controller = new StudentController(student, view);

// Display initial data

controller.updateView();

System.out.println("\nUpdating student name and grade...\n");

// Update data using controller

controller.setStudentName("Jane Smith");

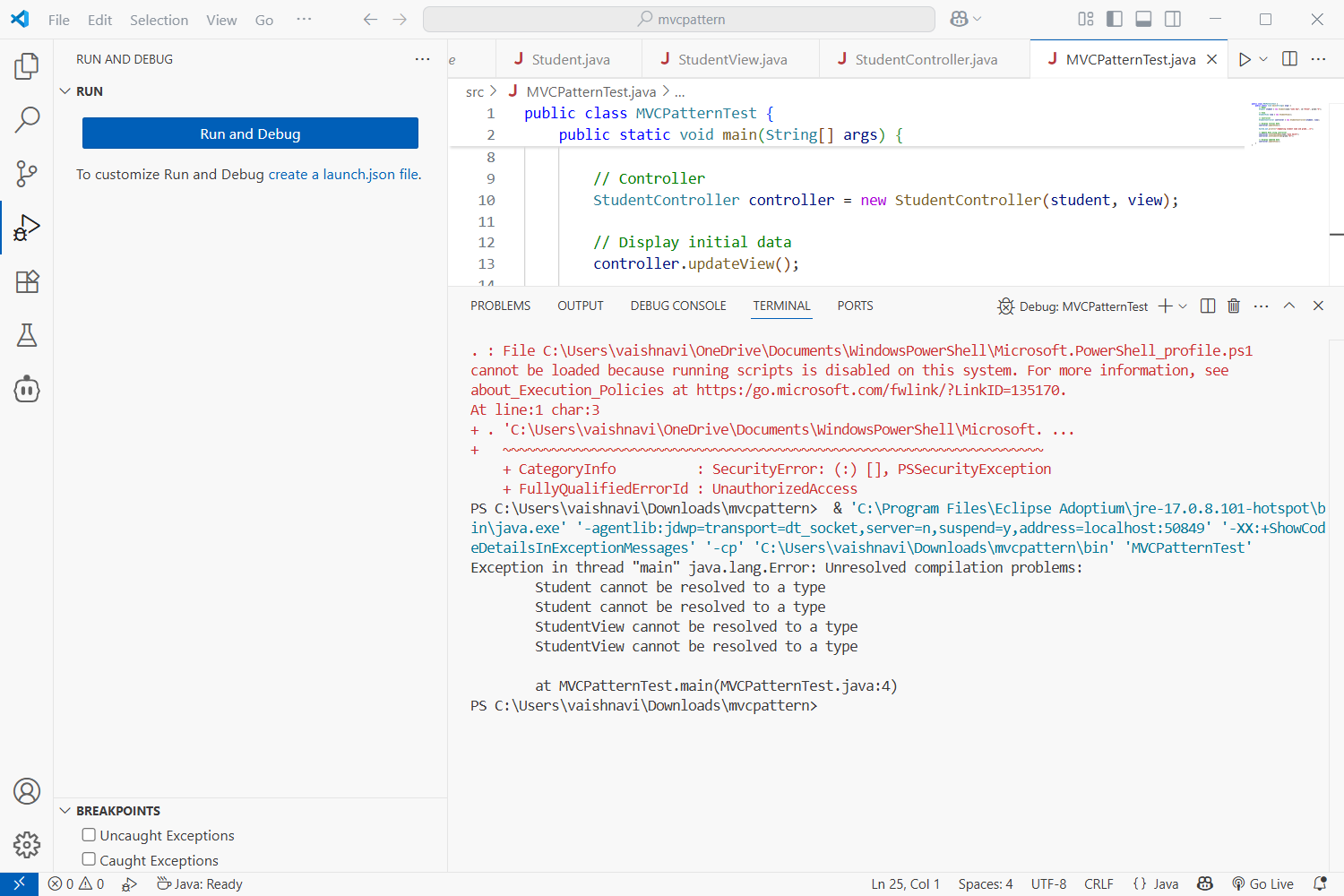
controller.setStudentGrade("A+");

// Display updated data

controller.updateView();

}

}



**Exercise 11: Implementing Dependency Injection**

**Customer.java**

public interface CustomerRepository {

Customer findCustomerById(String customerId);

}

**CustomerRepositoryImpl.java**

public class CustomerRepositoryImpl implements CustomerRepository {

@Override

public Customer findCustomerById(String customerId) {

// Simulated database lookup

return new Customer(customerId, "Alice Johnson");

}

}

**CustomerService.java**

public class CustomerService {

private CustomerRepository customerRepository;

// Constructor Injection

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void displayCustomer(String customerId) {

Customer customer = customerRepository.findCustomerById(customerId);

if (customer != null) {

System.out.println("Customer found:");

System.out.println("ID: " + customer.getId());

System.out.println("Name: " + customer.getName());

} else {

System.out.println("Customer not found.");

}

}

}

**Customer.java**

public class Customer {

private String id;

private String name;

public Customer(String id, String name) {

this.id = id;

this.name = name;

}

// Getters

public String getId() {

return id;

}

public String getName() {

return name;

}

}

**DependencyInjectionTest.java**

public class DependencyInjectionTest {

public static void main(String[] args) {

// Create repository

CustomerRepository repository = new CustomerRepositoryImpl();

// Inject repository into service

CustomerService service = new CustomerService(repository);

// Use the service

service.displayCustomer("C001");

}

}

