**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**Project name : SingletonPatternExample**

**Logger.java**

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);

}

}

**LoggerTest.java**

public class LoggerTest {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

logger1.log("Message from logger1");

Logger logger2 = Logger.getInstance();

logger2.log("Message from logger2");

if (logger1 == logger2) {

System.out.println("Both logger instances are the same (Singleton verified).");

} else {

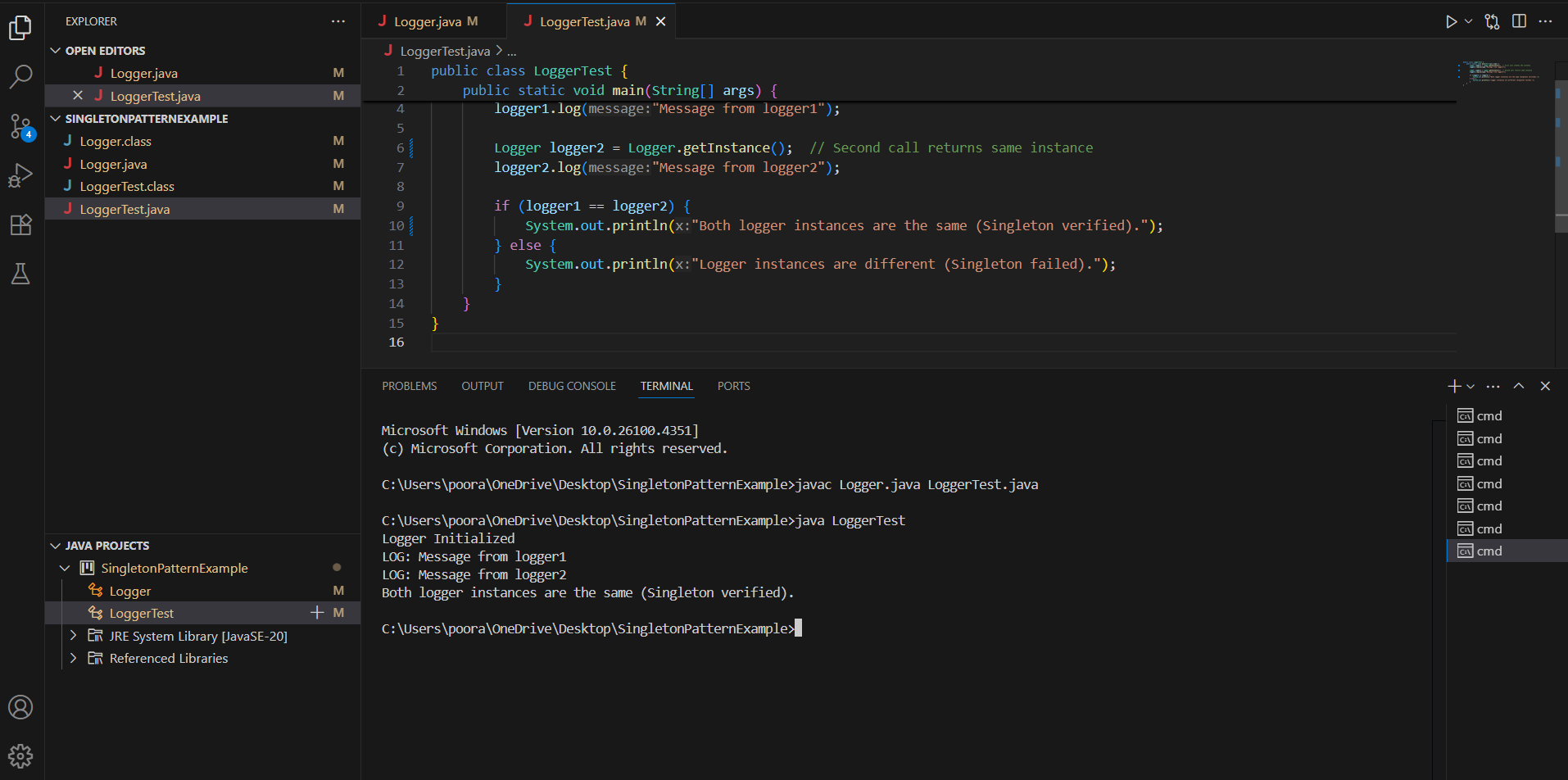
System.out.println("Logger instances are different (Singleton failed).");

}

}

}

**Output screenshot:**

****

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

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Project name: FactoryMethodPatternExample**

**Document.java**

public interface Document {

void open();

}

**WordDocument.java**

public class WordDocument implements Document {

@Override

public void open() {

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

}

}

**PdfDocument.java**

public class PdfDocument implements Document {

@Override

public void open() {

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

}

}

**ExcelDocument.java**

public class ExcelDocument implements Document {

@Override

public void open() {

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

}

}

**DocumentFactory.java**

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**WordFactory.java**

public class WordFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

**PdfFactory.java**

public class PdfFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

**ExcelFactory.java**

public class ExcelFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

**DocumentTest.java**

public class DocumentTest {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordFactory();

Document word = wordFactory.createDocument();

word.open();

DocumentFactory pdfFactory = new PdfFactory();

Document pdf = pdfFactory.createDocument();

pdf.open();

DocumentFactory excelFactory = new ExcelFactory();

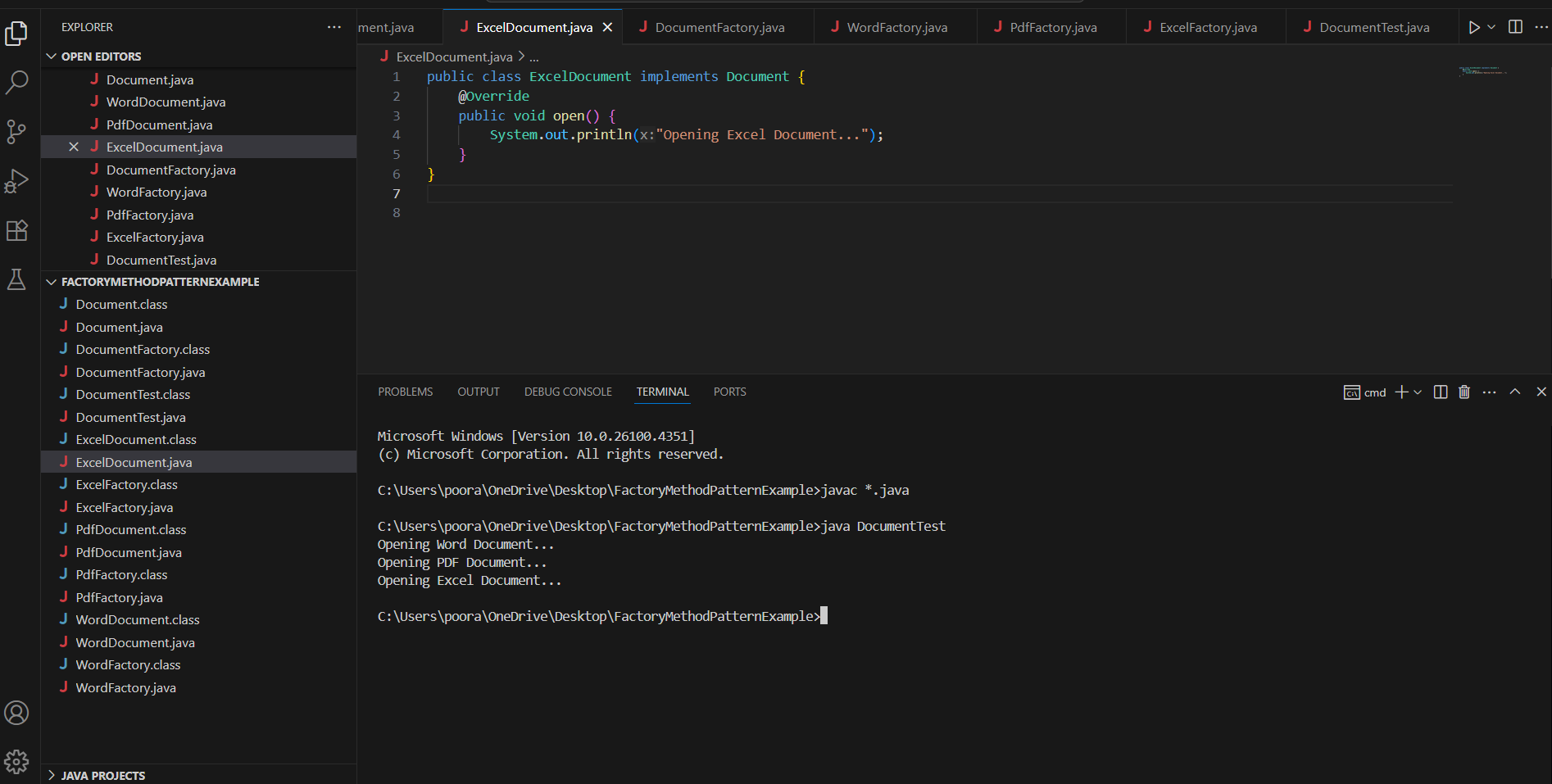
Document excel = excelFactory.createDocument();

excel.open();

}

}

**Output screenshot:**

****

**Exercise 3: Implementing the Builder Pattern**

**Scenario:**

You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

**Project name : BuilderPatternExample**

**Computer.java**

public class Computer {

private final String cpu;

private final String ram;

private final String storage;

private final String gpu;

private final boolean isBluetoothEnabled;

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

this.gpu = builder.gpu;

this.isBluetoothEnabled = builder.isBluetoothEnabled;

}

public static class Builder {

private String cpu;

private String ram;

private String storage;

private String gpu;

private boolean isBluetoothEnabled;

public Builder setCpu(String cpu) {

this.cpu = cpu;

return this;

}

public Builder setRam(String ram) {

this.ram = ram;

return this;

}

public Builder setStorage(String storage) {

this.storage = storage;

return this;

}

public Builder setGpu(String gpu) {

this.gpu = gpu;

return this;

}

public Builder enableBluetooth(boolean isBluetoothEnabled) {

this.isBluetoothEnabled = isBluetoothEnabled;

return this;

}

public Computer build() {

return new Computer(this);

}

}

@Override

public String toString() {

return "Computer Configuration:\n" +

"CPU: " + cpu + "\n" +

"RAM: " + ram + "\n" +

"Storage: " + storage + "\n" +

"GPU: " + (gpu != null ? gpu : "None") + "\n" +

"Bluetooth Enabled: " + isBluetoothEnabled;

}

}

**BuilderPatternTest.java**

public class BuilderPatternTest {

public static void main(String[] args) {

Computer basicComputer = new Computer.Builder()

.setCpu("Intel i5")

.setRam("8GB")

.setStorage("256GB SSD")

.enableBluetooth(false)

.build();

Computer gamingComputer = new Computer.Builder()

.setCpu("AMD Ryzen 9")

.setRam("32GB")

.setStorage("1TB SSD")

.setGpu("NVIDIA RTX 4080")

.enableBluetooth(true)

.build();

System.out.println(basicComputer);

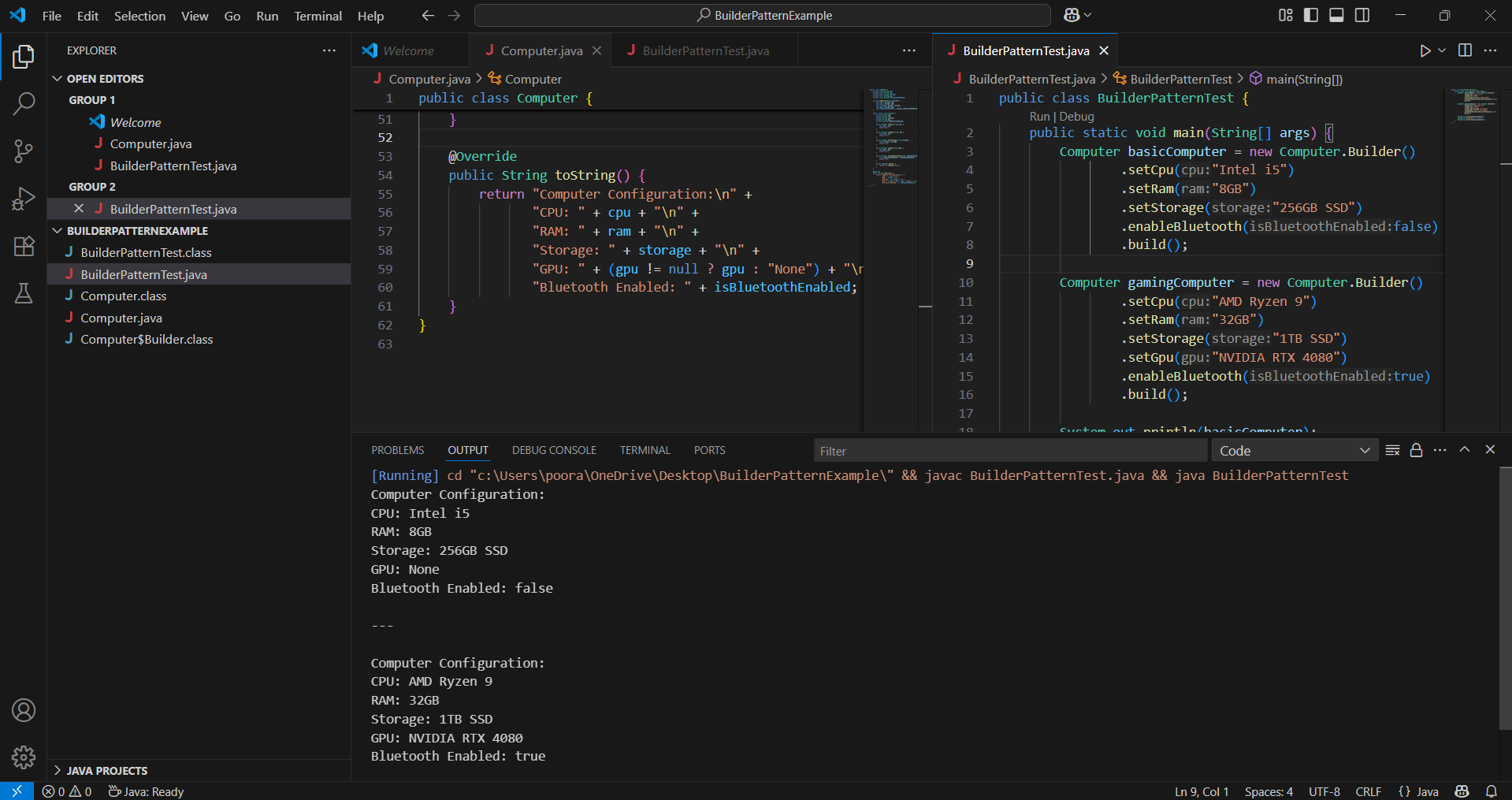
System.out.println("\n---\n");

System.out.println(gamingComputer);

}

}

**Output screenshot:**

****

**Exercise 4: Implementing the Adapter Pattern**

**Scenario:**

You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

**Project name : AdapterPatternExample**

**PaymentProcessor.java**

public interface PaymentProcessor {

void processPayment(double amount);

}

**StripeGateway.java**

public class StripeGateway {

public void makeStripePayment(double amountInDollars) {

System.out.println("Processing payment of $" + amountInDollars + " via Stripe.");

}

}

**StripeAdapter.java**

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe;

}

@Override

public void processPayment(double amount) {

stripe.makeStripePayment(amount);

}

}

**PayPalGateway.java**

public class PayPalGateway {

public void sendPayment(double amount) {

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

}

}

**PayPalAdapter.java**

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway paypal;

public PayPalAdapter(PayPalGateway paypal) {

this.paypal = paypal;

}

@Override

public void processPayment(double amount) {

paypal.sendPayment(amount);

}

}

**RazorpayGateway.java**

public class RazorpayGateway {

public void payUsingRazor(double amountInINR) {

System.out.println("Processing payment of ₹" + amountInINR + " via Razorpay.");

}

}

**RazorpayAdapter.java**

public class RazorpayAdapter implements PaymentProcessor {

private RazorpayGateway razorpay;

public RazorpayAdapter(RazorpayGateway razorpay) {

this.razorpay = razorpay;

}

@Override

public void processPayment(double amount) {

razorpay.payUsingRazor(amount);

}

}

**AdapterPatternTest.java**

public class AdapterPatternTest {

public static void main(String[] args) {

// Stripe

StripeGateway stripe = new StripeGateway();

PaymentProcessor stripeProcessor = new StripeAdapter(stripe);

stripeProcessor.processPayment(250.75);

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

// PayPal

PayPalGateway paypal = new PayPalGateway();

PaymentProcessor paypalProcessor = new PayPalAdapter(paypal);

paypalProcessor.processPayment(150.00);

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

// Razorpay

RazorpayGateway razorpay = new RazorpayGateway();

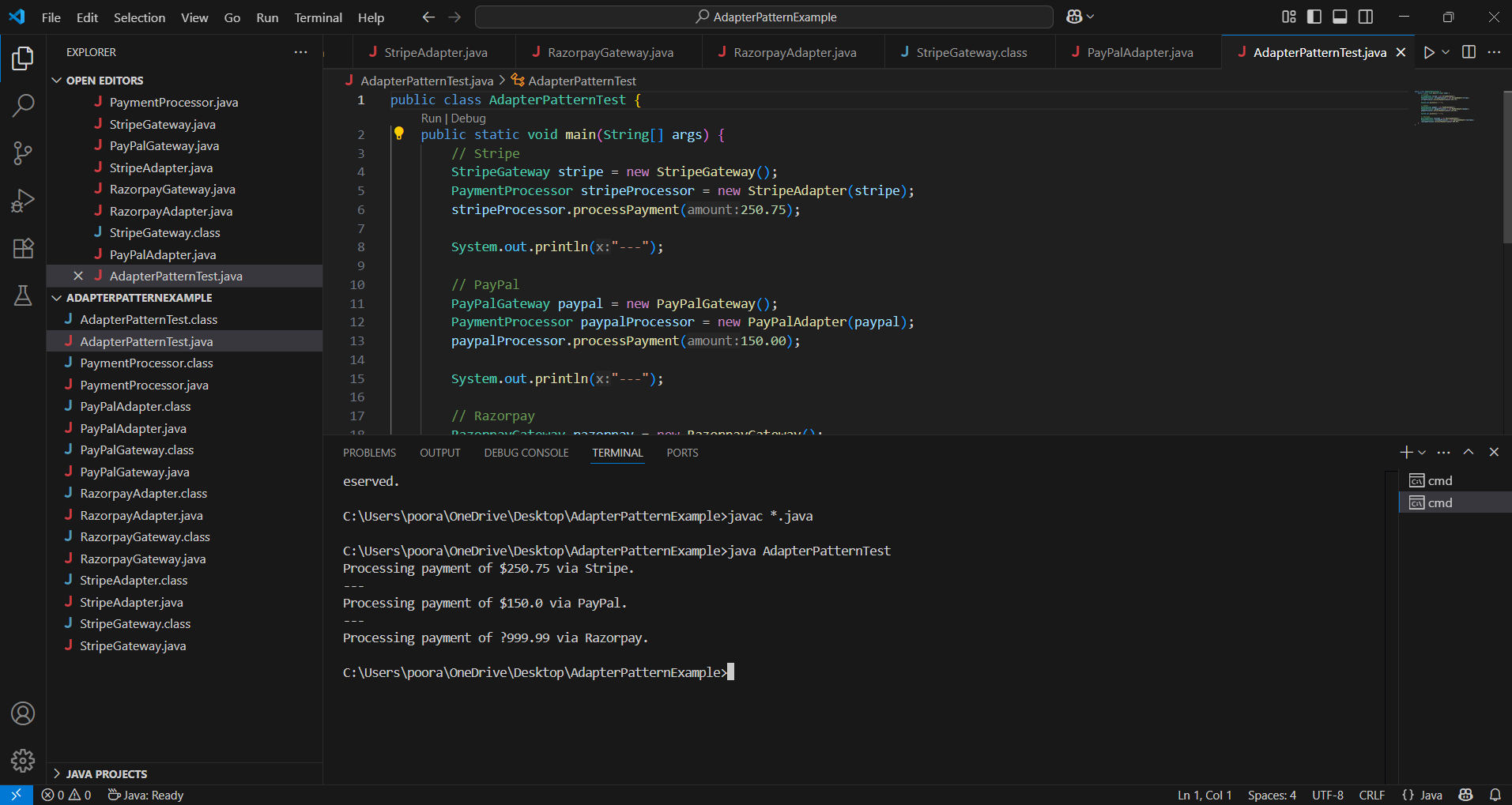
PaymentProcessor razorpayProcessor = new RazorpayAdapter(razorpay);

razorpayProcessor.processPayment(999.99);

}

}

**Output screenshot:**

****

**Exercise 5: Implementing the Decorator Pattern**

**Scenario:**

You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically.

**Project name : DecoratorPatternExample**

**Notifier.java**

public interface Notifier {

void send(String message);

}

**EmailNotifier.java**

public class EmailNotifier implements Notifier {

@Override

public void send(String message) {

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

}

}

**NotifierDecorator.java**

public abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

@Override

public void send(String message) {

notifier.send(message); // Delegation

}

}

**SMSNotifierDecorator.java**

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message); // existing channel

sendSMS(message); // added functionality

}

private void sendSMS(String message) {

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

}

}

**SlackNotifierDecorator.java**

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message); // existing channel(s)

sendSlack(message); // added functionality

}

private void sendSlack(String message) {

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

}

}

**DecoratorPatternTest.java**

public class DecoratorPatternTest {

public static void main(String[] args) {

// Base notifier: Email only

Notifier emailNotifier = new EmailNotifier();

// Add SMS

Notifier smsAndEmailNotifier = new SMSNotifierDecorator(emailNotifier);

// Add Slack on top of Email and SMS

Notifier fullNotifier = new SlackNotifierDecorator(smsAndEmailNotifier);

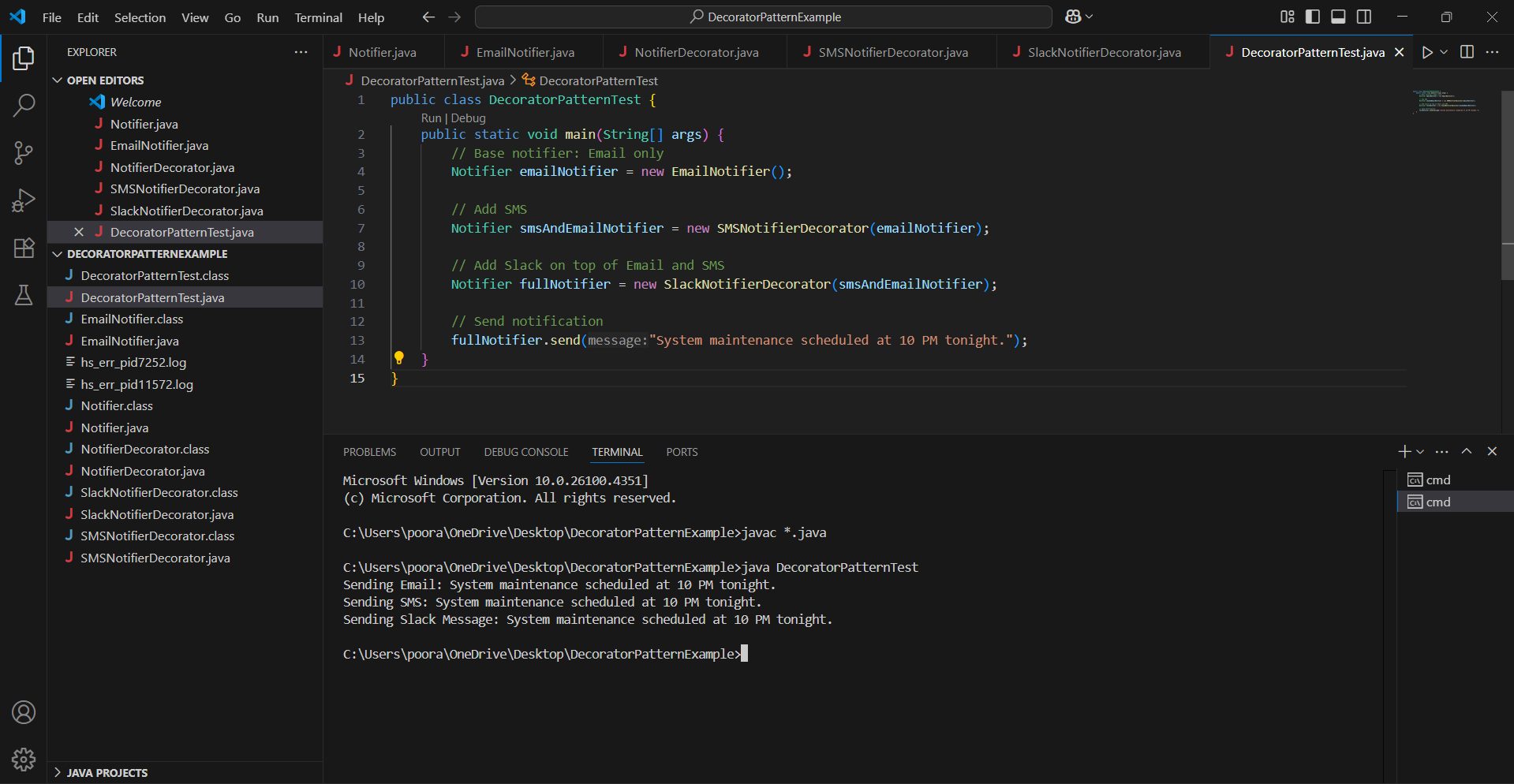
// Send notification

fullNotifier.send("System maintenance scheduled at 10 PM tonight.");

}

}

**Output screenshot:**

****

**Exercise 6: Implementing the Proxy Pattern**

**Scenario:**

You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching.

**Project name : ProxyPatternExample**

**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 image from remote server: " + filename);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

@Override

public void display() {

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

}

}

**ProxyImage.java**

public class ProxyImage implements Image {

private String filename;

private RealImage realImage;

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("nature.jpg");

// Image will be loaded from remote server on first display

image1.display();

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

// Image will be displayed from cache (not reloaded)

image1.display();

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

// Different image - will trigger a new load

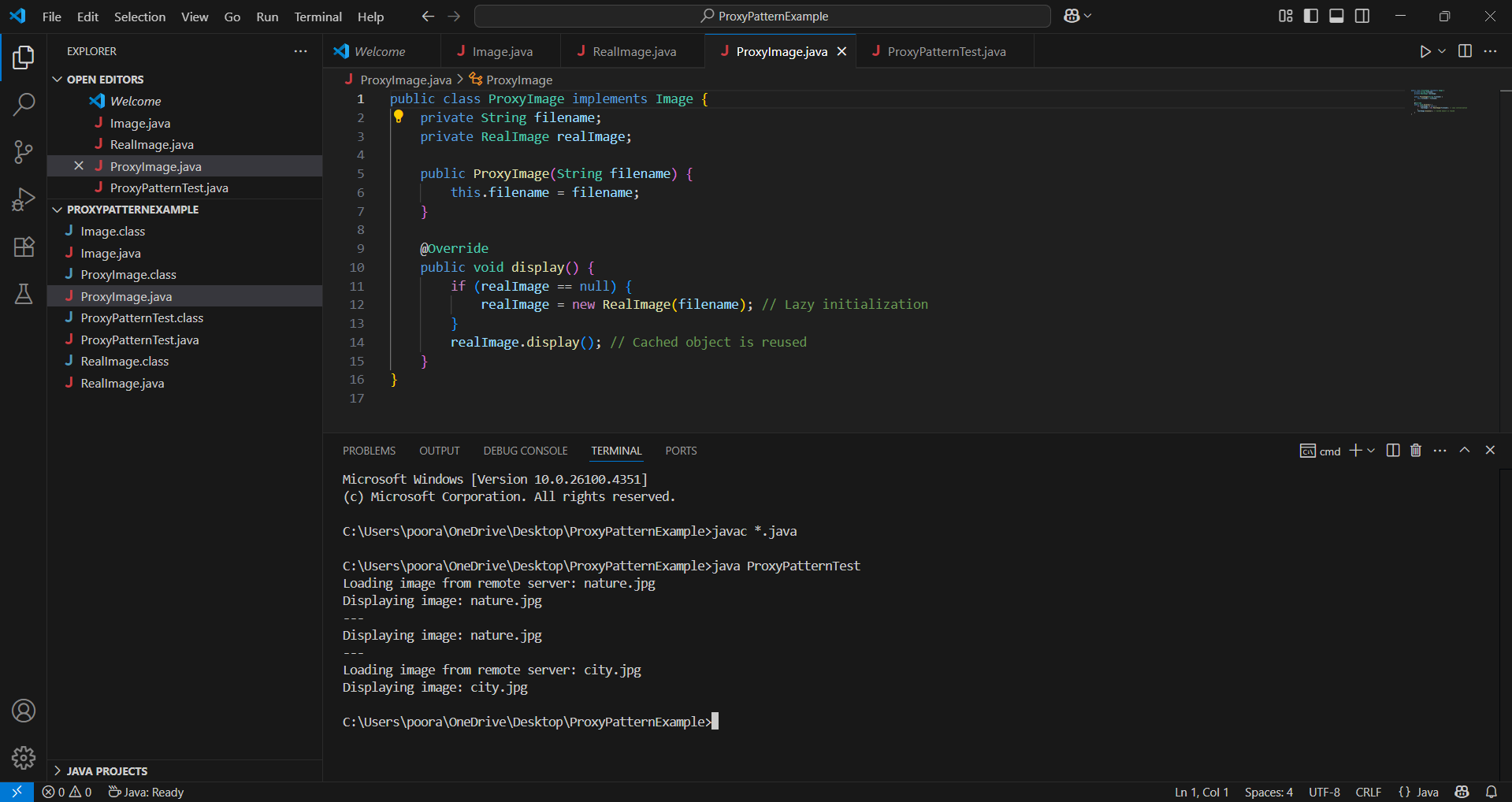
Image image2 = new ProxyImage("city.jpg");

image2.display();

}

}

**Output screenshot:**

****

**Exercise 7: Implementing the Observer Pattern**

**Scenario:**

You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

**Project name : ObserverPatternExample**

**Observer.java**

public interface Observer {

void update(String stockName, double stockPrice);

}

**Stock.java**

public interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

**StockMarket.java**

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

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

private String stockName;

private double stockPrice;

@Override

public void registerObserver(Observer o) {

observers.add(o);

}

@Override

public void removeObserver(Observer o) {

observers.remove(o);

}

@Override

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockName, stockPrice);

}

}

public void setStockPrice(String stockName, double price) {

this.stockName = stockName;

this.stockPrice = price;

notifyObservers();

}

}

**MobileApp.java**

public class MobileApp implements Observer {

private String appName;

public MobileApp(String appName) {

this.appName = appName;

}

@Override

public void update(String stockName, double stockPrice) {

System.out.println(appName + " (Mobile App) - Stock Update: " + stockName + " is now ₹" + stockPrice);

}

}

**WebApp.java**

public class WebApp implements Observer {

private String appName;

public WebApp(String appName) {

this.appName = appName;

}

@Override

public void update(String stockName, double stockPrice) {

System.out.println(appName + " (Web App) - Stock Update: " + stockName + " is now ₹" + stockPrice);

}

}

**ObserverPatternTest.java**

public class ObserverPatternTest {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobileApp = new MobileApp("BullsEye");

Observer webApp = new WebApp("MarketWatch");

stockMarket.registerObserver(mobileApp);

stockMarket.registerObserver(webApp);

stockMarket.setStockPrice("TCS", 3475.50);

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

stockMarket.setStockPrice("INFY", 1510.75);

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

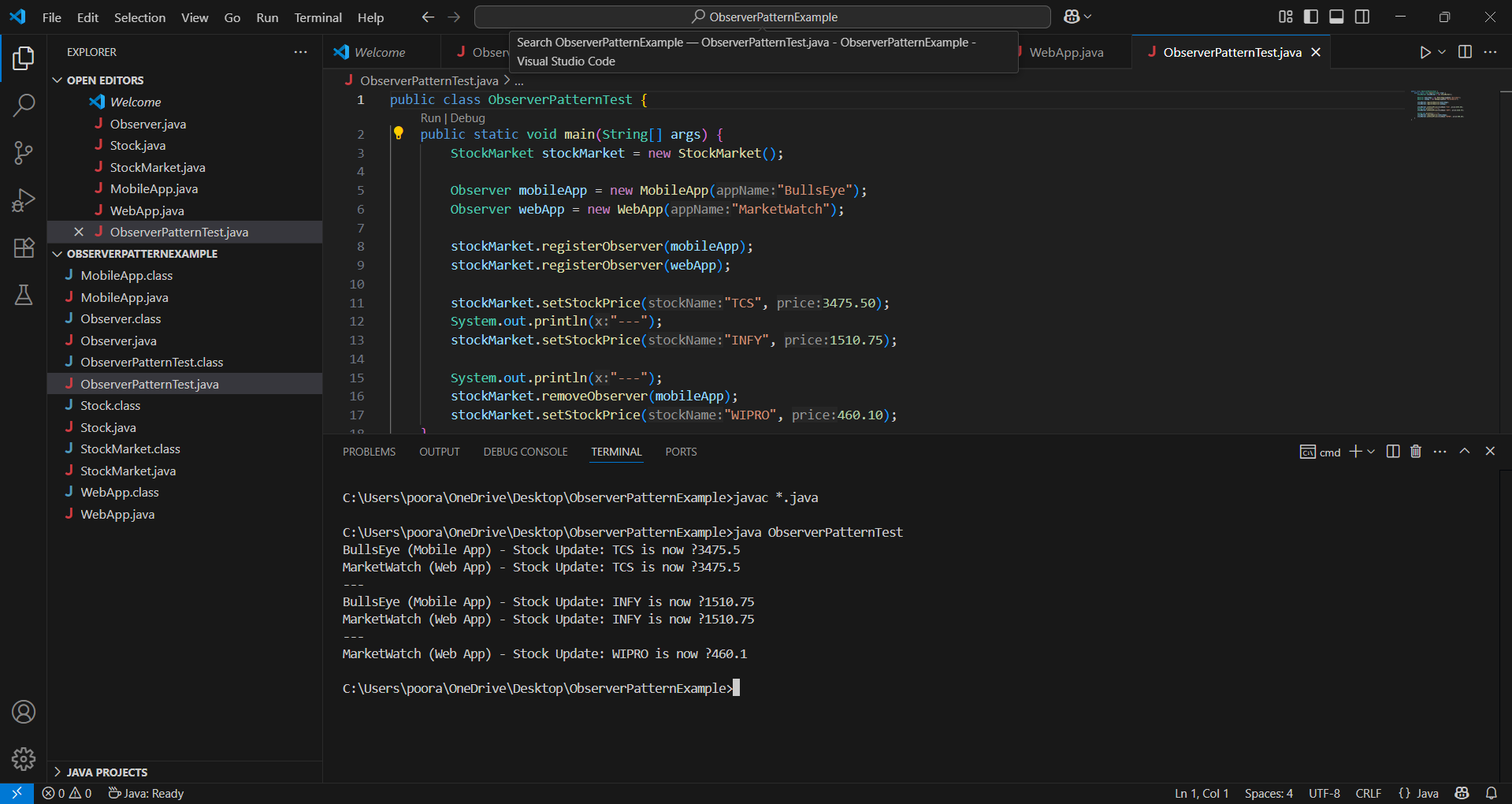
stockMarket.removeObserver(mobileApp);

stockMarket.setStockPrice("WIPRO", 460.10);

}

}

**Output screenshot:**

****

**Exercise 8: Implementing the Strategy Pattern**

**Scenario:**

You are developing a payment system where different payment methods (e.g., Credit Card, PayPal) can be selected at runtime. Use the Strategy Pattern to achieve this.

**Project name : StrategyPatternExample**

**PaymentStrategy.java**

public interface PaymentStrategy {

void pay(double amount);

}

**CreditCardPayment.java**

public class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String cardHolder;

public CreditCardPayment(String cardNumber, String cardHolder) {

this.cardNumber = cardNumber;

this.cardHolder = cardHolder;

}

@Override

public void pay(double amount) {

System.out.println("Paid ₹" + amount + " using Credit Card [" + cardHolder + "]");

}

}

**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 [" + email + "]");

}

}

**PaymentContext.java**

public class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void executePayment(double amount) {

if (strategy == null) {

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

} else {

strategy.pay(amount);

}

}

}

**StrategyPatternTest.java**

public class StrategyPatternTest {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

// Paying via Credit Card

PaymentStrategy creditCard = new CreditCardPayment("1234-5678-9012-3456", "Poorani Sakthivel");

context.setPaymentStrategy(creditCard);

context.executePayment(1500.00);

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

// Paying via PayPal

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

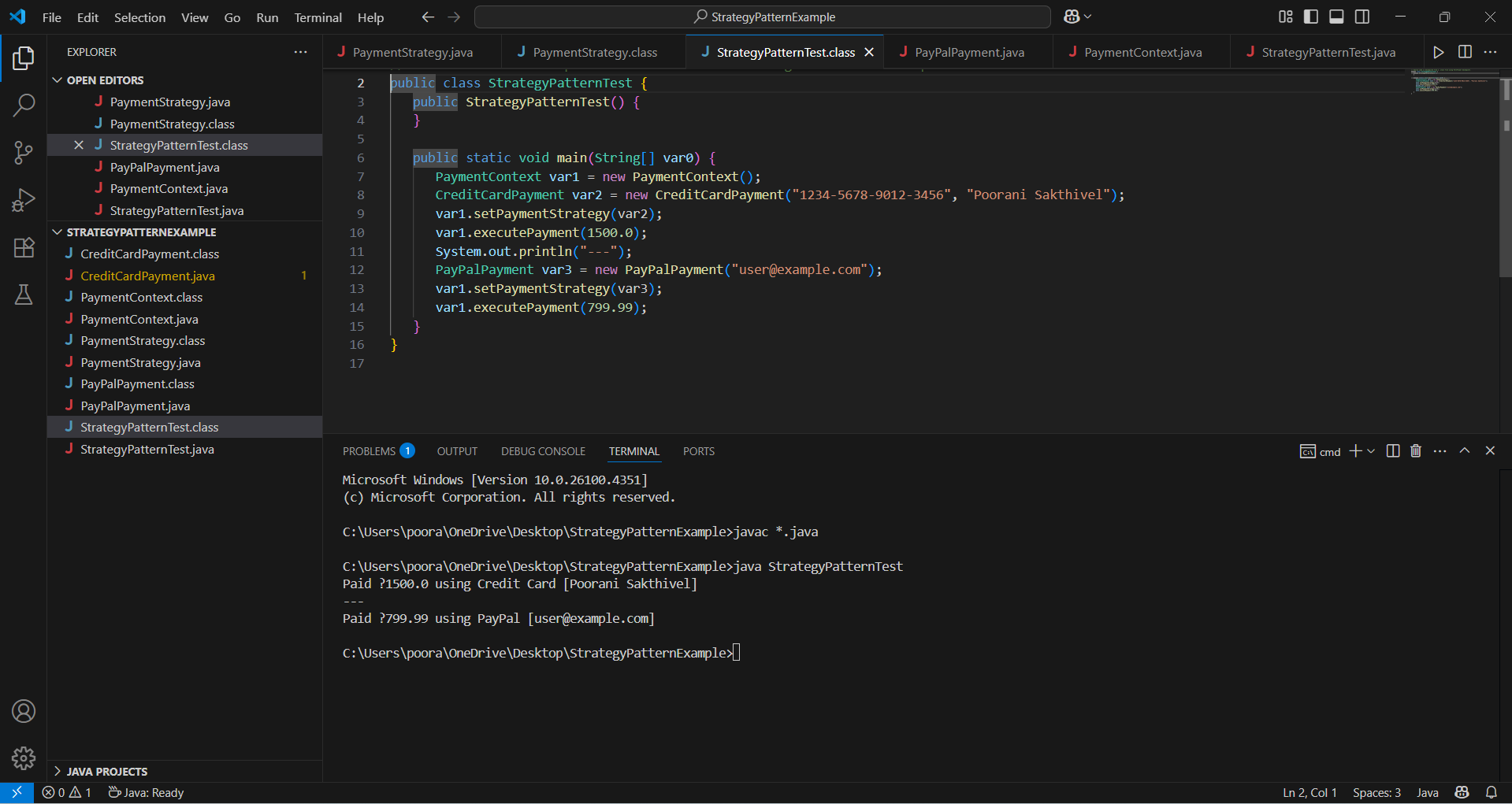
context.setPaymentStrategy(paypal);

context.executePayment(799.99);

}

}

**Output screenshot:**

****

**Exercise 9: Implementing the Command Pattern**

**Scenario:**

You are developing a home automation system where commands can be issued to turn devices on or off. Use the Command Pattern to achieve this.

**Project name : CommandPatternExample**

**Command.java**

public interface Command {

void execute();

}

**Light.java**

public class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

**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.");

}

}

}

**CommandPatternTest.java**

public class CommandPatternTest {

public static void main(String[] args) {

// Receiver

Light livingRoomLight = new Light();

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

// Invoker

RemoteControl remote = new RemoteControl();

// Turn light ON

remote.setCommand(lightOn);

remote.pressButton();

// Turn light OFF

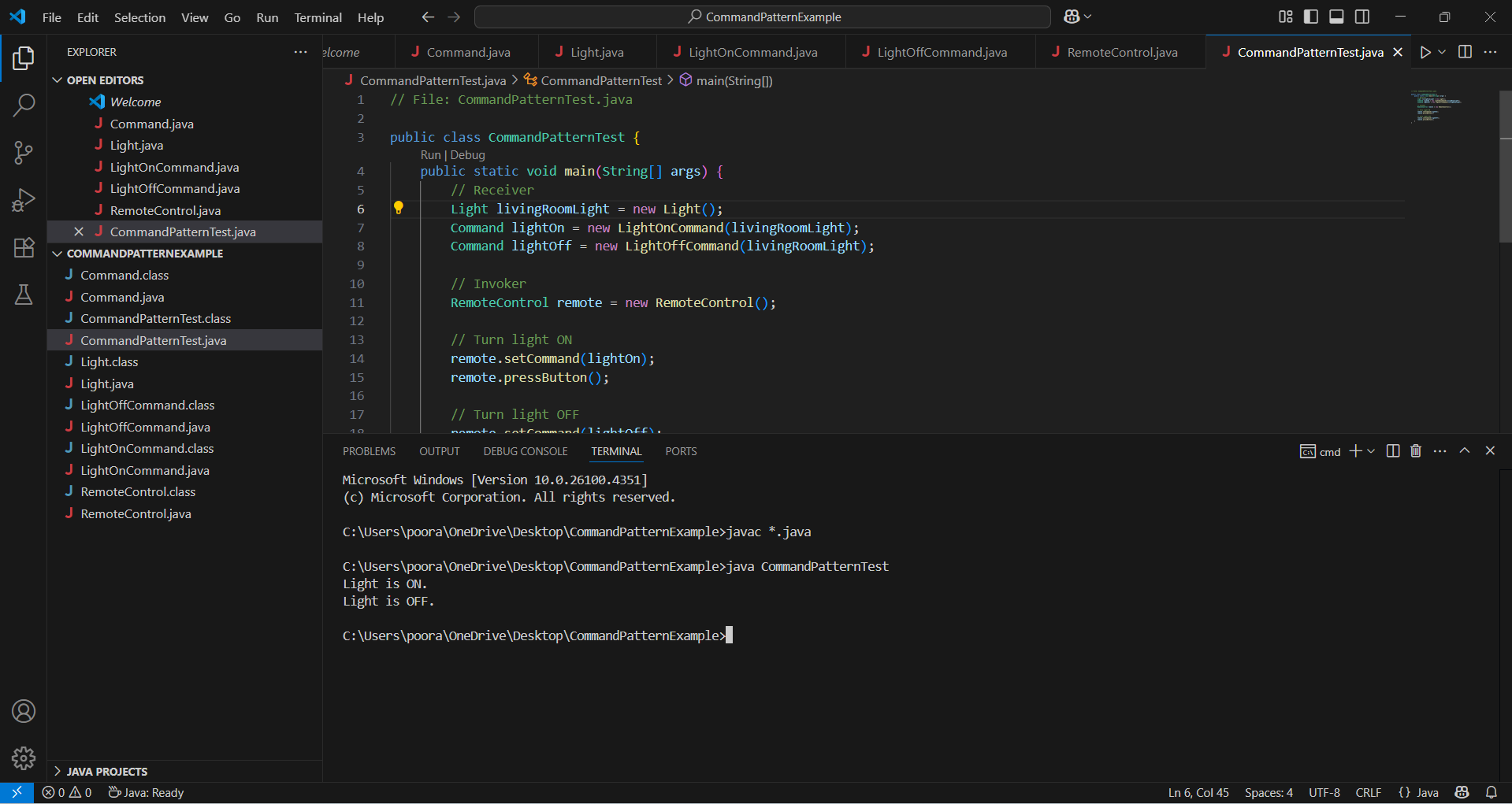
remote.setCommand(lightOff);

remote.pressButton();

}

}

**Output screenshot:**

****

**Exercise 10: Implementing the MVC Pattern**

**Scenario:**

You are developing a simple web application for managing student records using the MVC pattern.

**Project name : MVCPatternExample**

**Student.java**

public class Student {

private String name;

private String id;

private String 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 void setStudentId(String id) {

model.setId(id);

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public String getStudentName() {

return model.getName();

}

public String getStudentId() {

return model.getId();

}

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) {

// Create model and view

Student student = new Student();

StudentView view = new StudentView();

// Create controller

StudentController controller = new StudentController(student, view);

// Set data

controller.setStudentName("Poorani Sakthivel");

controller.setStudentId("S12345");

controller.setStudentGrade("A");

// Display student data

controller.updateView();

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

// Update and re-display

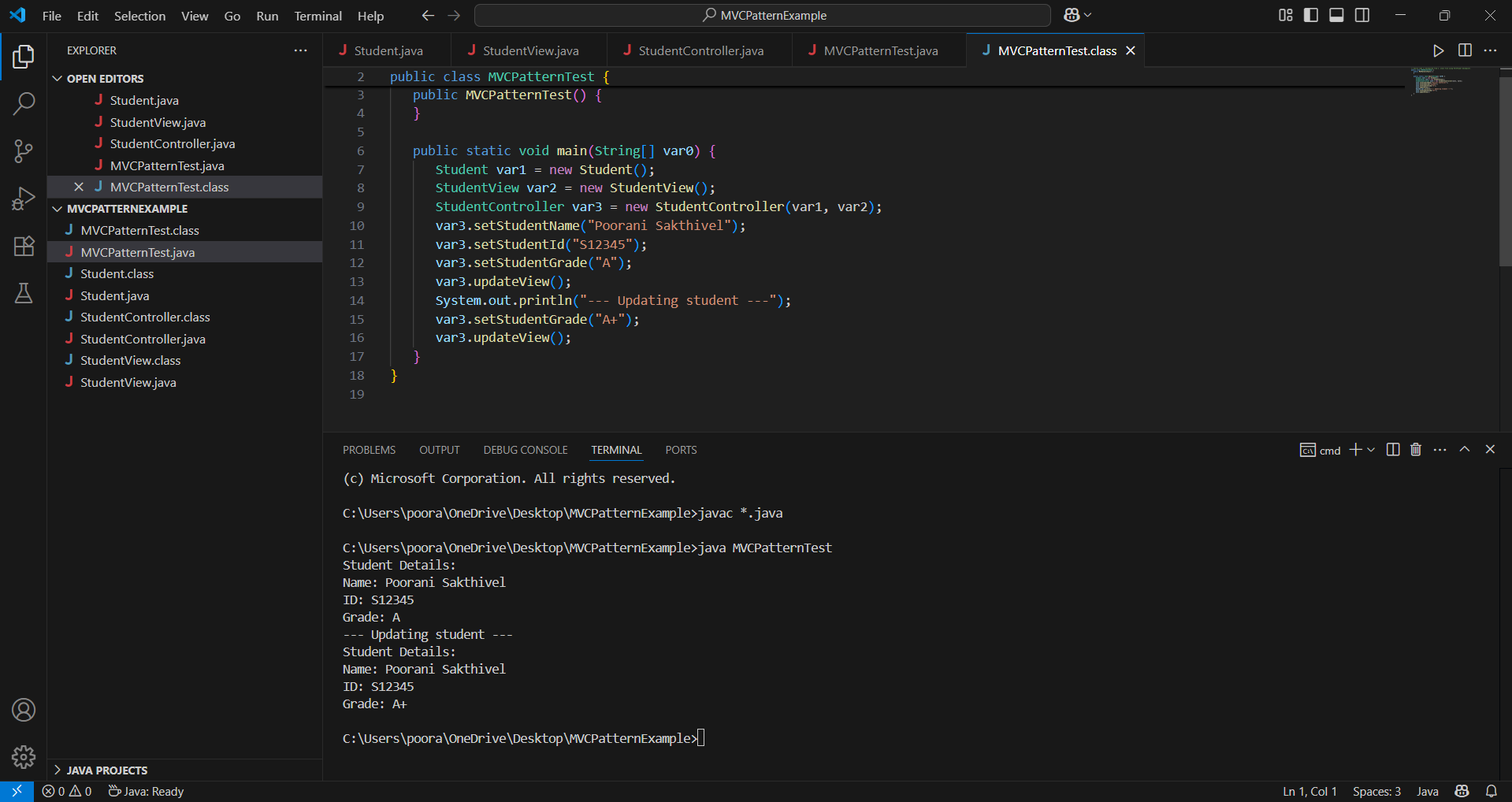
controller.setStudentGrade("A+");

controller.updateView();

}

}

**Output screenshot:**

****

**Exercise 11: Implementing Dependency Injection**

**Scenario:**

You are developing a customer management application where the service class depends on a repository class. Use Dependency Injection to manage these dependencies.

**Project name : DependencyInjectionExample**

**CustomerRepository.java**

public interface CustomerRepository {

Customer findCustomerById(String customerId);

}

**Customer.java**

public class Customer {

private String id;

private String name;

public Customer(String id, String name) {

this.id = id;

this.name = name;

}

public String getId() {

return id;

}

public String getName() {

return name;

}

}

**CustomerRepositoryImpl.java**

public class CustomerRepositoryImpl implements CustomerRepository {

@Override

public Customer findCustomerById(String customerId) {

return new Customer(customerId, "Poorani Sakthivel");

}

}

**CustomerService.java**

public class CustomerService {

private CustomerRepository customerRepository;

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void displayCustomerDetails(String customerId) {

Customer customer = customerRepository.findCustomerById(customerId);

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

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

}

}

**DependencyInjectionTest.java**

public class DependencyInjectionTest {

public static void main(String[] args) {

// Manual DI

CustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

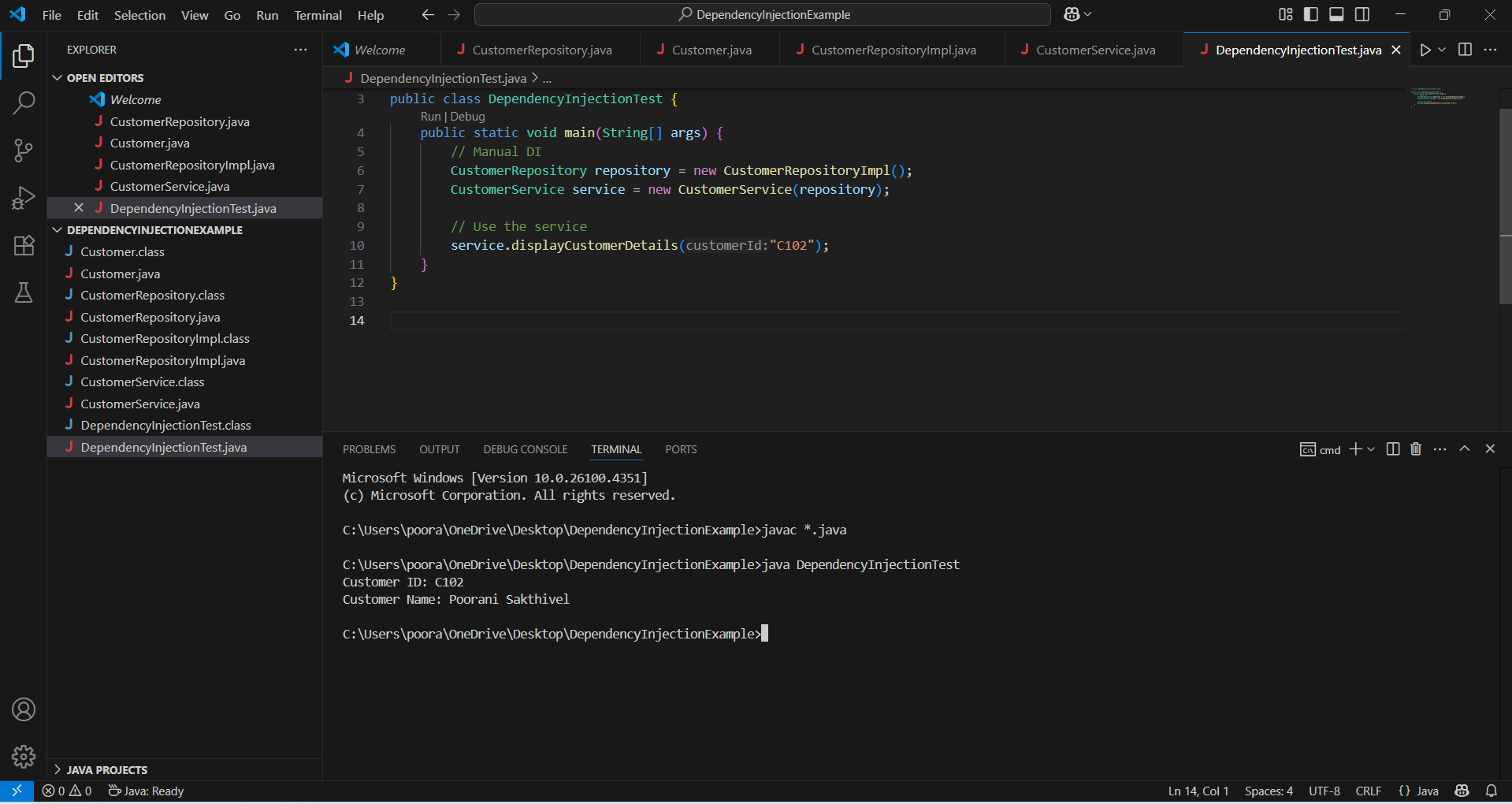
// Use the service

service.displayCustomerDetails("C102");

}

}

**Output screenshot:**

****