**Exercise 1: Implementing the Singleton Pattern**

**Logger.java**

package designpatternandprinciple.SingletonPatternExample;

public class Logger {

    // Step 1: Static instance

    private static Logger instance;

    // Step 2: Private constructor to prevent instantiation

    private Logger() {

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

    }

    // Step 3: Public method to get the instance

    public static Logger getInstance() {

        if (instance == null) {

            instance = new Logger();

        }

        return instance;

    }

    // Logging method

    public void log(String message) {

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

    }

}

**Main.java**

package designpatternandprinciple.SingletonPatternExample;

public class Main {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        logger1.log("This is the first log.");

        Logger logger2 = Logger.getInstance();

        logger2.log("This is the second log.");

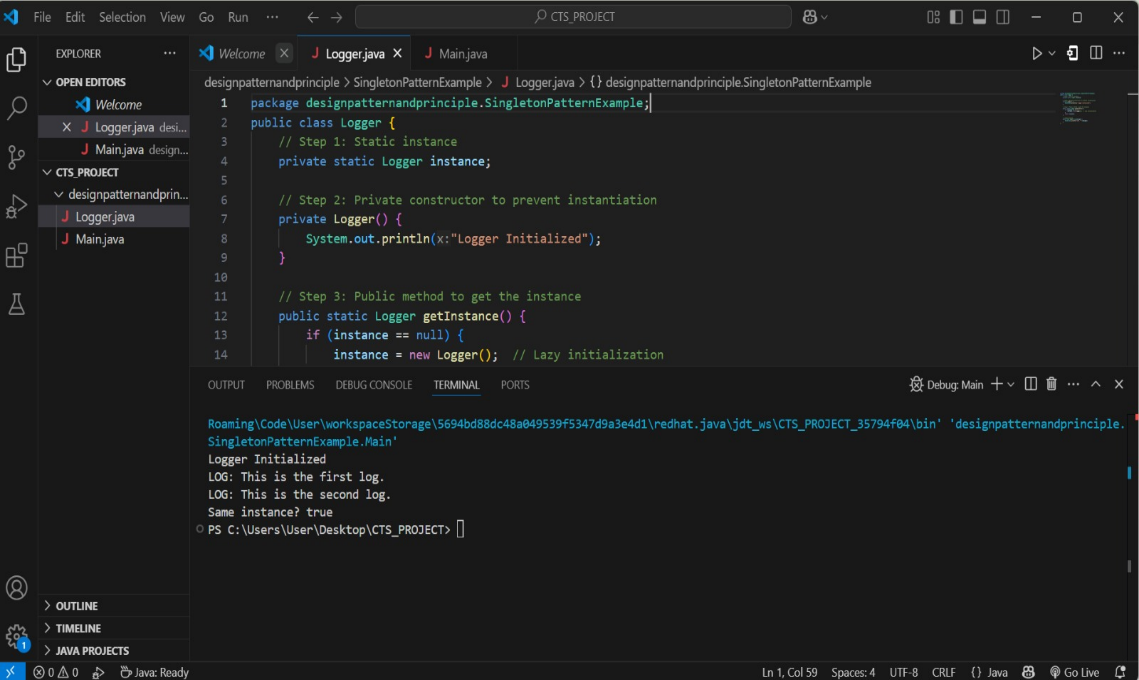
        // Check if both logger1 and logger2 are same

        System.out.println("Same instance? " + (logger1 == logger2));

    }

}

**Output**

****

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

**FactoryMethodPatternExample.java**

package designpatternandprinciple;

interface Document {

    void open();

}

// Concrete Document Classes

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...");

    }

}

// Factory Abstract Class

abstract class DocumentFactory {

    public abstract Document createDocument();

}

// Concrete Factories

class WordFactory extends DocumentFactory {

    public Document createDocument() {

        return new WordDocument();

    }

}

class PdfFactory extends DocumentFactory {

    public Document createDocument() {

        return new PdfDocument();

    }

}

class ExcelFactory extends DocumentFactory {

    public Document createDocument() {

        return new ExcelDocument();

    }

}

public class FactoryMethodPatternExample {

  public static void main(String[] args) {

        DocumentFactory wordFactory = new WordFactory();

        Document wordDoc = wordFactory.createDocument();

        wordDoc.open();

        DocumentFactory pdfFactory = new PdfFactory();

        Document pdfDoc = pdfFactory.createDocument();

        pdfDoc.open();

        DocumentFactory excelFactory = new ExcelFactory();

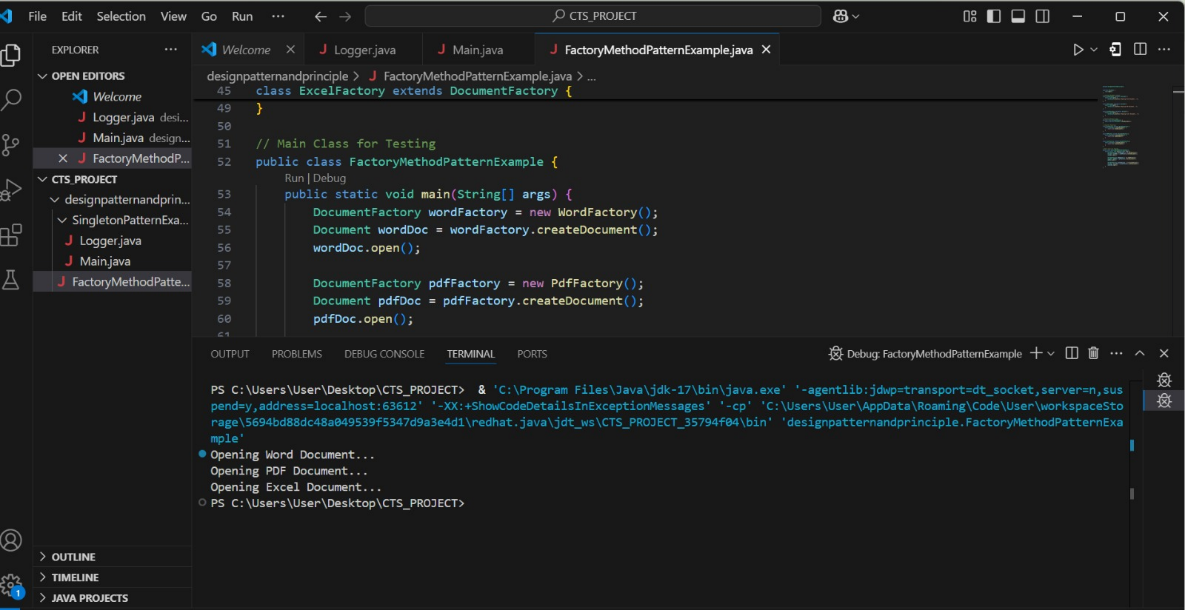
        Document excelDoc = excelFactory.createDocument();

        excelDoc.open();

    }

}

**Output**

****

**Exercise 3: Implementing the Builder Pattern**

**BuilderPatternExample.java**

// Product class

class Computer {

// Required attributes

private String CPU;

private String RAM;

// Optional attributes

private String storage;

private String graphicsCard;

// Private constructor to enforce Builder usage

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

}

public void showSpecs() {

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("-------------------------");

}

// 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);

}

}

}

public class BuilderPatternExample {

public static void main(String[] args) {

// Basic configuration

Computer basicPC = new Computer.Builder("Intel i3", "4GB")

.build();

// Gaming configuration

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

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 4090")

.build();

System.out.println("Basic PC:");

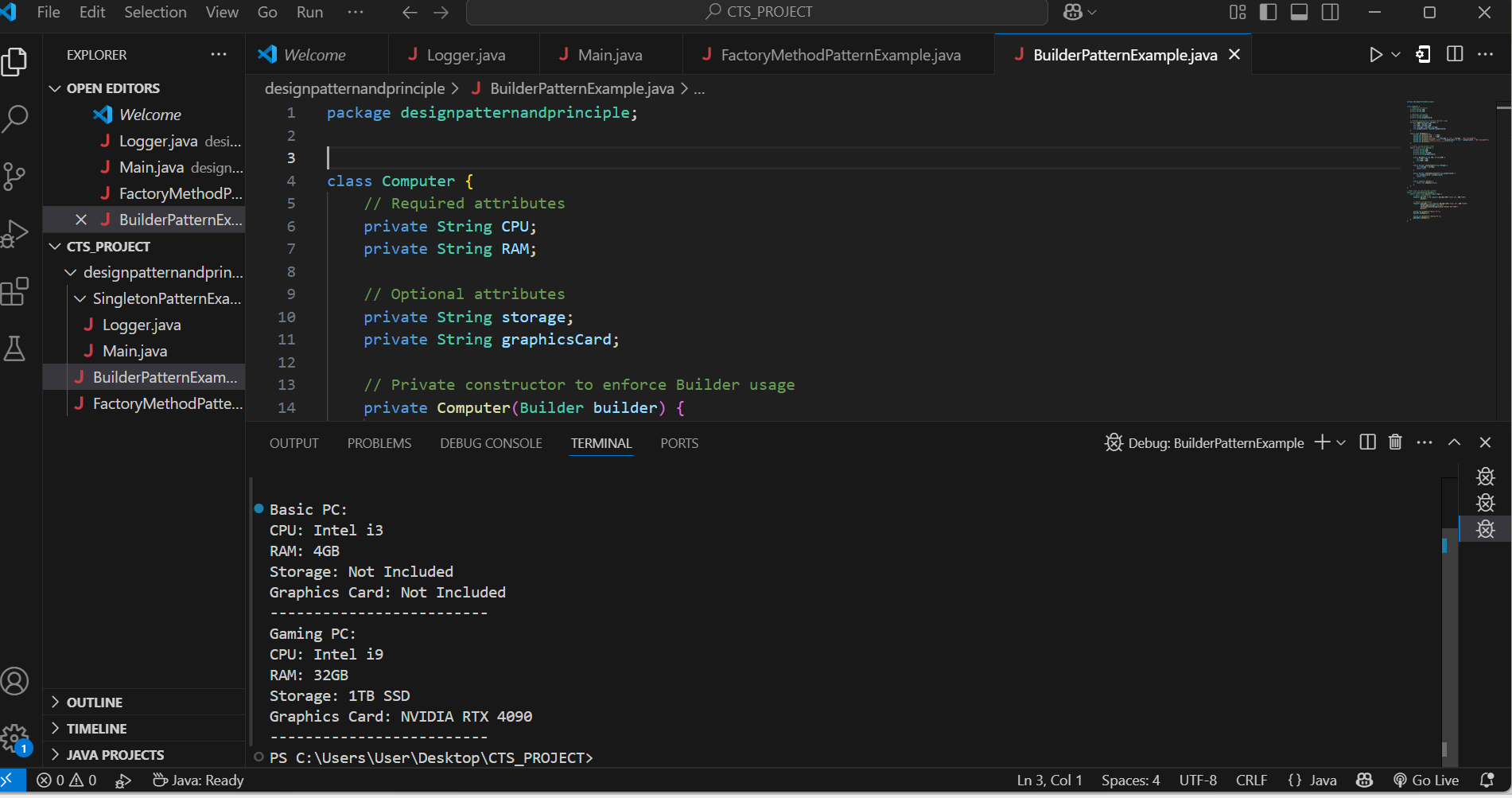
basicPC.showSpecs();

System.out.println("Gaming PC:");

gamingPC.showSpecs();

}}

**Output**

****

**Exercise 4: Implementing the Adapter Pattern**

**AdapterPatternExample.java**

package designpatternandprinciple;

// No package used

// Target Interface

interface PaymentProcessor {

    void processPayment(double amount);

}

// Adaptee 1 - PayPal

class PayPalGateway {

    public void sendPayment(double amount) {

        System.out.println("Payment of ₹" + amount + " processed via PayPal.");

    }

}

// Adaptee 2 - RazorPay

class RazorPayGateway {

    public void makePayment(double amount) {

        System.out.println("Payment of ₹" + amount + " processed via RazorPay.");

    }

}

// Adapter for PayPal

class PayPalAdapter implements PaymentProcessor {

    private PayPalGateway payPal;

    public PayPalAdapter() {

        this.payPal = new PayPalGateway();

    }

    public void processPayment(double amount) {

        payPal.sendPayment(amount);

    }

}

// Adapter for RazorPay

class RazorPayAdapter implements PaymentProcessor {

    private RazorPayGateway razorPay;

    public RazorPayAdapter() {

        this.razorPay = new RazorPayGateway();

    }

    public void processPayment(double amount) {

        razorPay.makePayment(amount);

    }

}

// Client

public class AdapterPatternExample {

    public static void main(String[] args) {

        PaymentProcessor paypal = new PayPalAdapter();

        paypal.processPayment(1500.0);

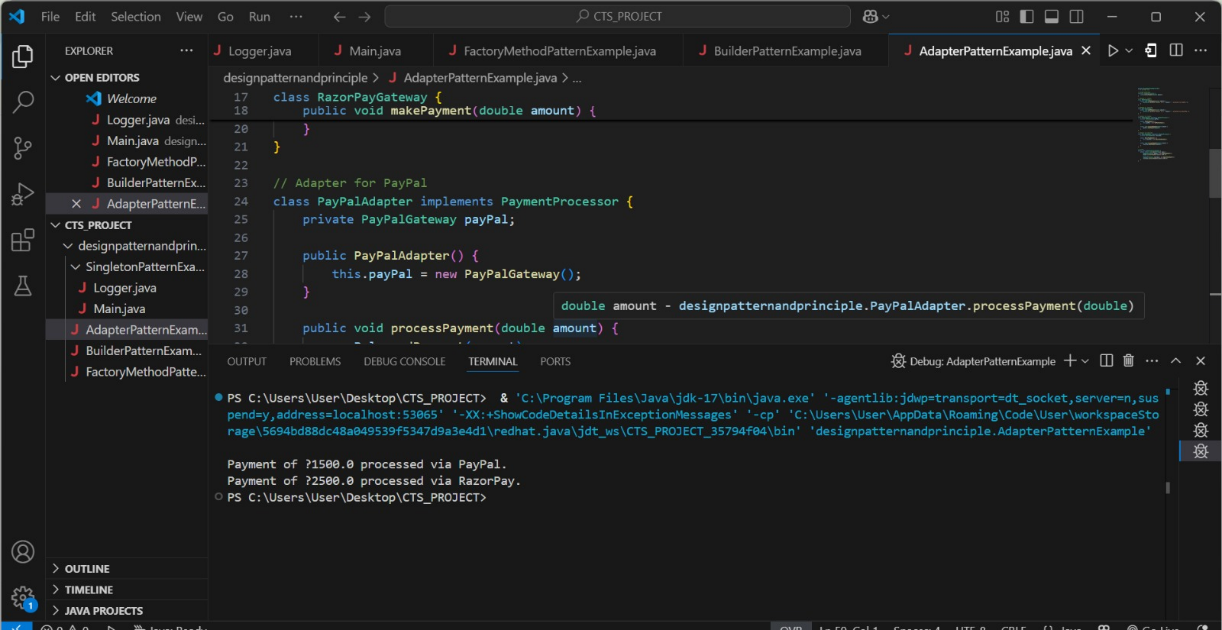
        PaymentProcessor razorpay = new RazorPayAdapter();

        razorpay.processPayment(2500.0);

    }

}

**Output**



**Exercise 5: Implementing the Decorator Pattern**

**DecoratorPatternExample.java**

// Component Interface

interface Notifier {

void send(String message);

}

// Concrete Component

class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

// Abstract Decorator

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send(String message) {

notifier.send(message); // delegate to base

}

}

// SMS Decorator

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

// Slack Decorator

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

// Client

public class DecoratorPatternExample {

public static void main(String[] args) {

// Send Email only

Notifier emailOnly = new EmailNotifier();

emailOnly.send("Server down!");

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

// Send Email + SMS

Notifier emailAndSMS = new SMSNotifierDecorator(new EmailNotifier());

emailAndSMS.send("Server down!");

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

// Send Email + SMS + Slack

Notifier allChannels = new SlackNotifierDecorator(

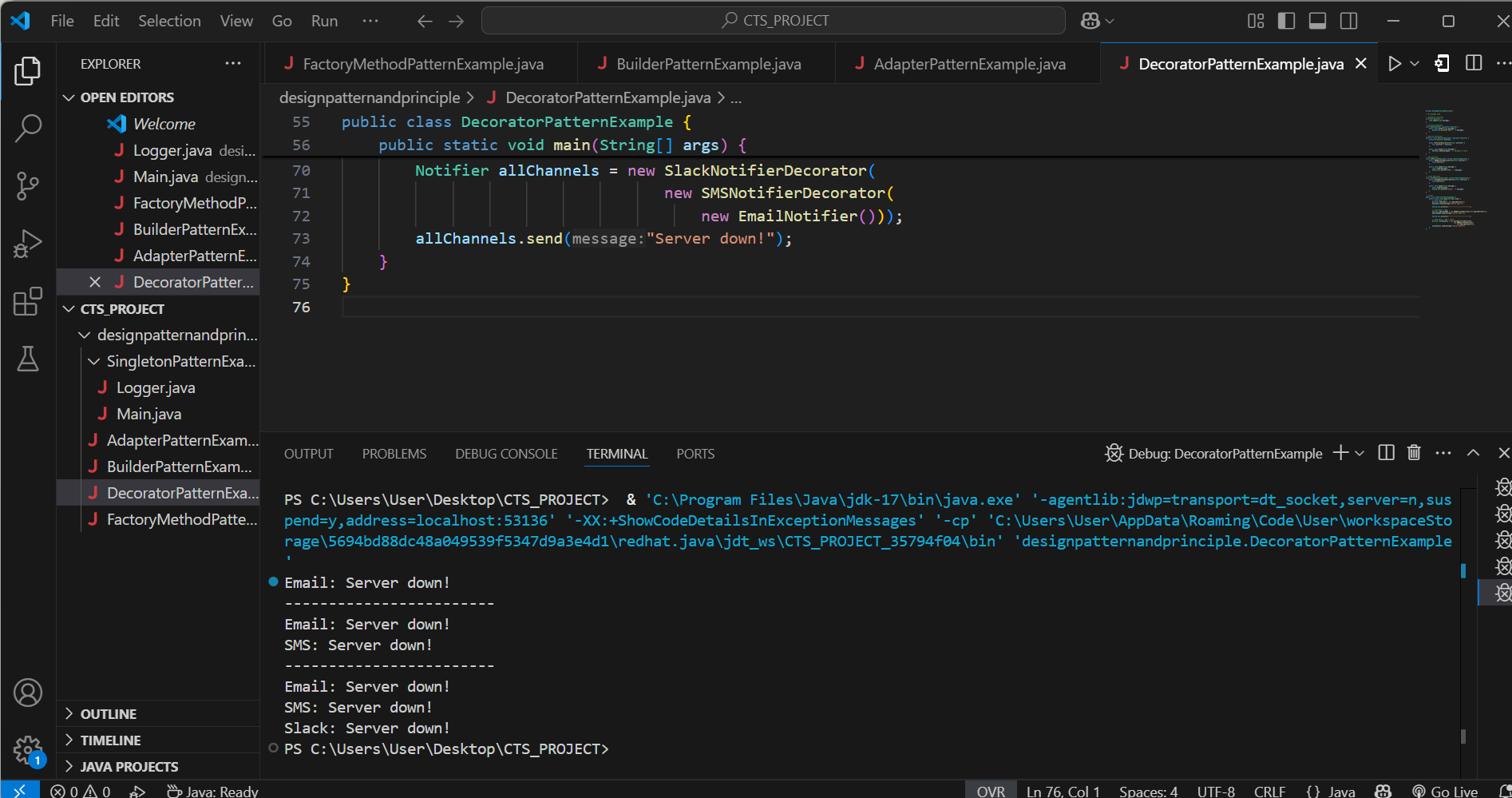
new SMSNotifierDecorator(

new EmailNotifier()));

allChannels.send("Server down!");

}}

**Output**

****

**Exercise 6: Implementing the Proxy Pattern**

**ProxyPatternExample.java**

// Subject Interface

interface Image {

void display();

}

// Real Subject

class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromDisk();

}

private void loadFromDisk() {

System.out.println("Loading image from disk: " + filename);

}

public void display() {

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

}

}

// Proxy

class ProxyImage implements Image {

private RealImage realImage;

private String filename;

public ProxyImage(String filename) {

this.filename = filename;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(filename); // Lazy loading

} else {

System.out.println("Using cached image for: " + filename);

}

realImage.display();

}

}

// Client

public class ProxyPatternExample {

public static void main(String[] args) {

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

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

image1.display(); // loads and displays

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

image1.display(); // cached

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

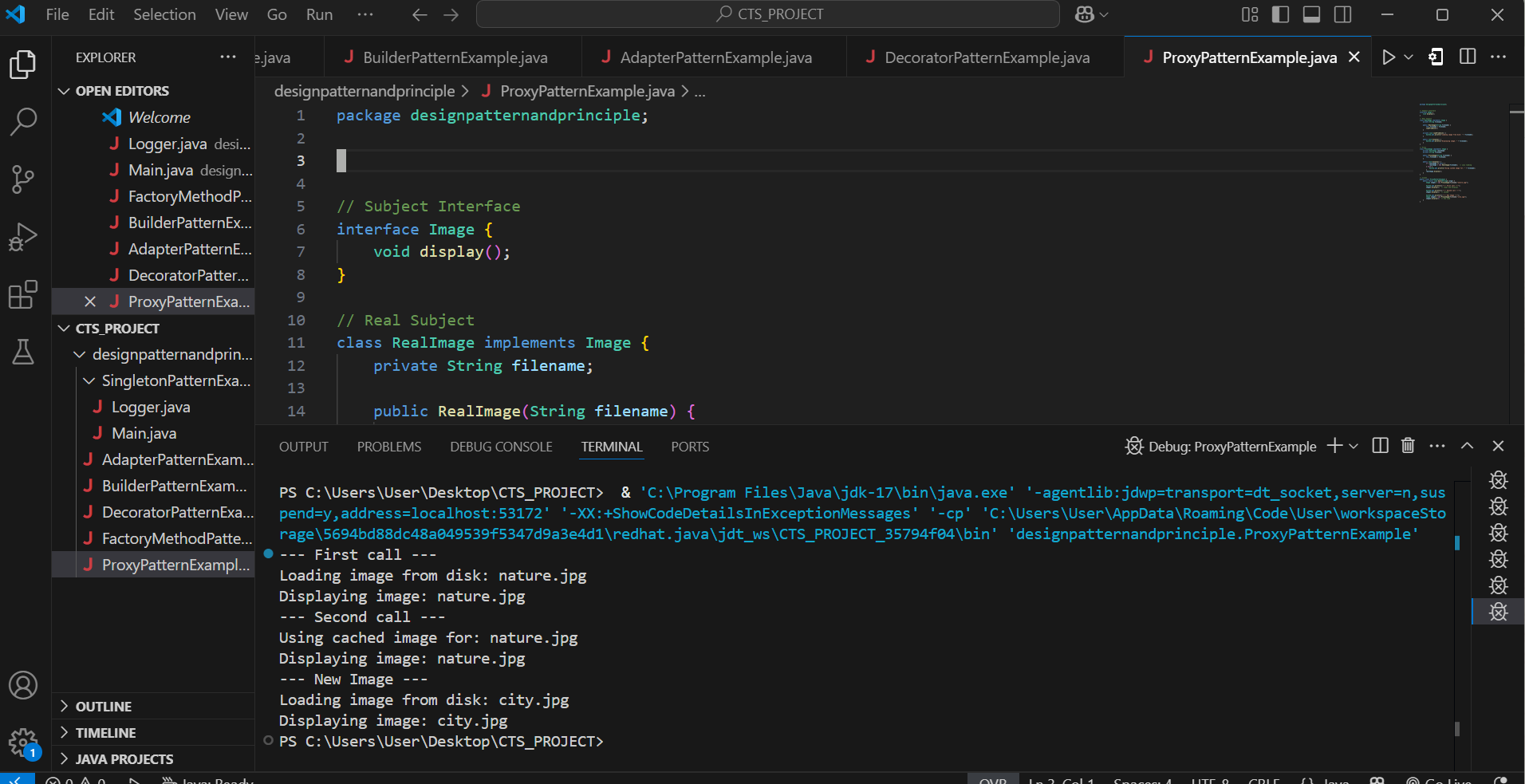
Image image2 = new ProxyImage("city.jpg");

image2.display(); // new load

}

}

**Output**

****

**Exercise 7: Implementing the Observer Pattern**

**ObserverPatternExample.java**

import java.util.\*;

// Observer Interface

interface Observer {

void update(String stock, double price);

}

// Subject Interface

interface Stock {

void register(Observer observer);

void deregister(Observer observer);

void notifyObservers();

}

// Concrete Subject

class StockMarket implements Stock {

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

private String stockName;

private double stockPrice;

public void setStock(String stockName, double stockPrice) {

this.stockName = stockName;

this.stockPrice = stockPrice;

notifyObservers();

}

public void register(Observer observer) {

observers.add(observer);

}

public void deregister(Observer observer) {

observers.remove(observer);

}

public void notifyObservers() {

for (Observer observer : observers) {

observer.update(stockName, stockPrice);

}

}

}

// Concrete Observers

class MobileApp implements Observer {

public void update(String stock, double price) {

System.out.println("MobileApp - " + stock + " updated to ₹" + price);

}

}

class WebApp implements Observer {

public void update(String stock, double price) {

System.out.println("WebApp - " + stock + " updated to ₹" + price);

}

}

// Client

public class ObserverPatternExample {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobile = new MobileApp();

Observer web = new WebApp();

market.register(mobile);

market.register(web);

System.out.println("--- Stock Update: INFY ---");

market.setStock("INFY", 1532.75);

System.out.println("--- Stock Update: TCS ---");

market.setStock("TCS", 3420.10);

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

market.deregister(web);

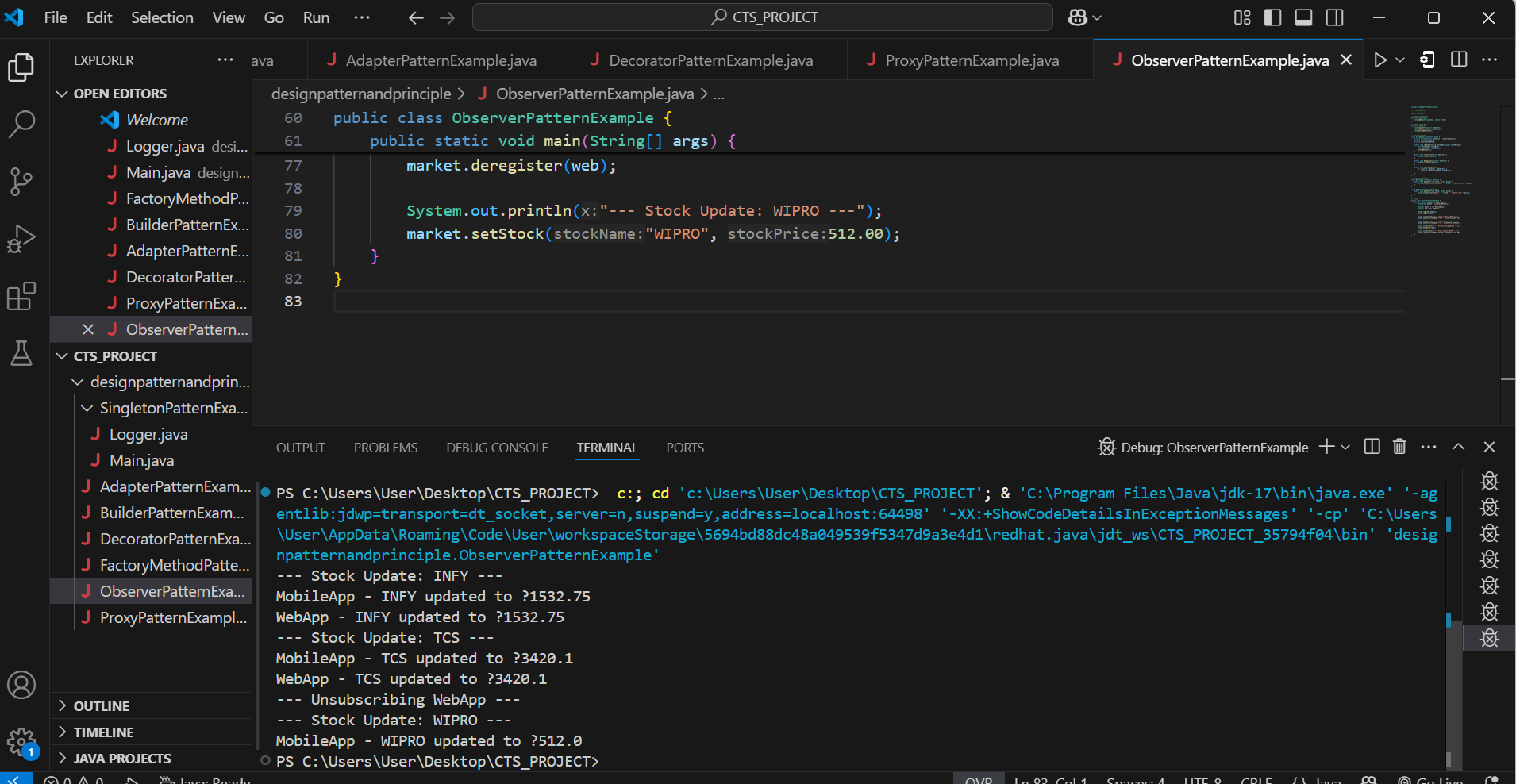
System.out.println("--- Stock Update: WIPRO ---");

market.setStock("WIPRO", 512.00);

}

}

**Output**

****

**Exercise 8: Implementing the Strategy Pattern**

**StrategyPatternExample.java**

// Strategy Interface

interface PaymentStrategy {

void pay(double amount);

}

// Concrete Strategies

class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

public CreditCardPayment(String cardNumber) {

this.cardNumber = cardNumber;

}

public void pay(double amount) {

System.out.println("Paid ₹" + amount + " using Credit Card: " + cardNumber);

}

}

class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

public void pay(double amount) {

System.out.println("Paid ₹" + amount + " using PayPal: " + email);

}

}

// Context Class

class PaymentContext {

private PaymentStrategy strategy;

public void setStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void makePayment(double amount) {

if (strategy != null) {

strategy.pay(amount);

} else {

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

}

}

}

// Client

public class StrategyPatternExample {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

System.out.println("Customer chooses Credit Card:");

context.setStrategy(new CreditCardPayment("1234-5678-9012-3456"));

context.makePayment(1200.0);

System.out.println("Customer chooses PayPal:");

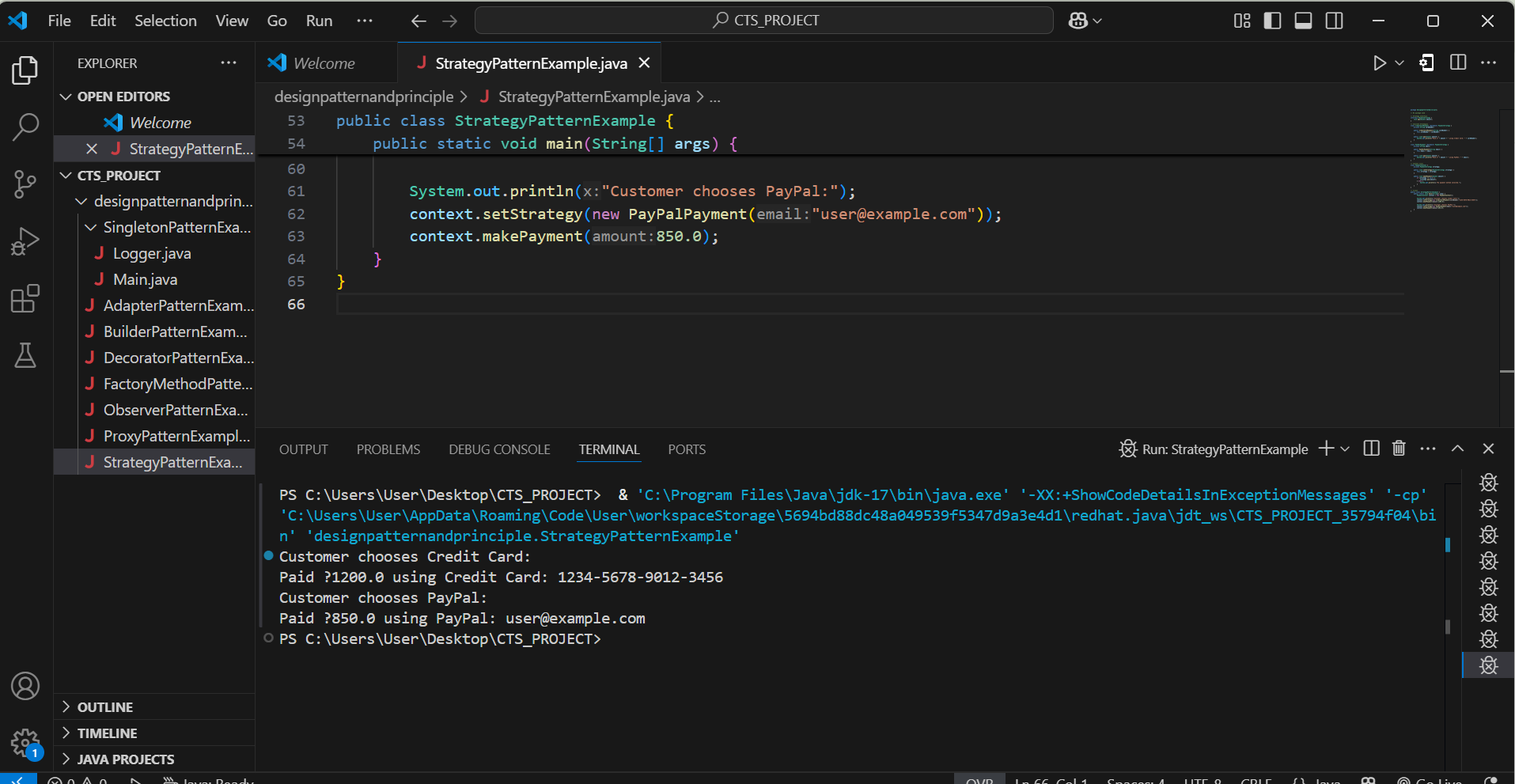
context.setStrategy(new PayPalPayment("user@example.com"));

context.makePayment(850.0);

}

}

**Output**



**Exercise 9: Implementing the Command Pattern**

CommandPatternExample.java

// No package used

// Command Interface

interface Command {

void execute();

}

// Receiver Class

class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

// Concrete Command to turn ON

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

// Concrete Command to turn OFF

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

// Invoker

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.");

}

}

}

// Client

public class CommandPatternExample {

public static void main(String[] args) {

Light bedroomLight = new Light();

Command lightOn = new LightOnCommand(bedroomLight);

Command lightOff = new LightOffCommand(bedroomLight);

RemoteControl remote = new RemoteControl();

System.out.println("Press ON button:");

remote.setCommand(lightOn);

remote.pressButton();

System.out.println("Press OFF button:");

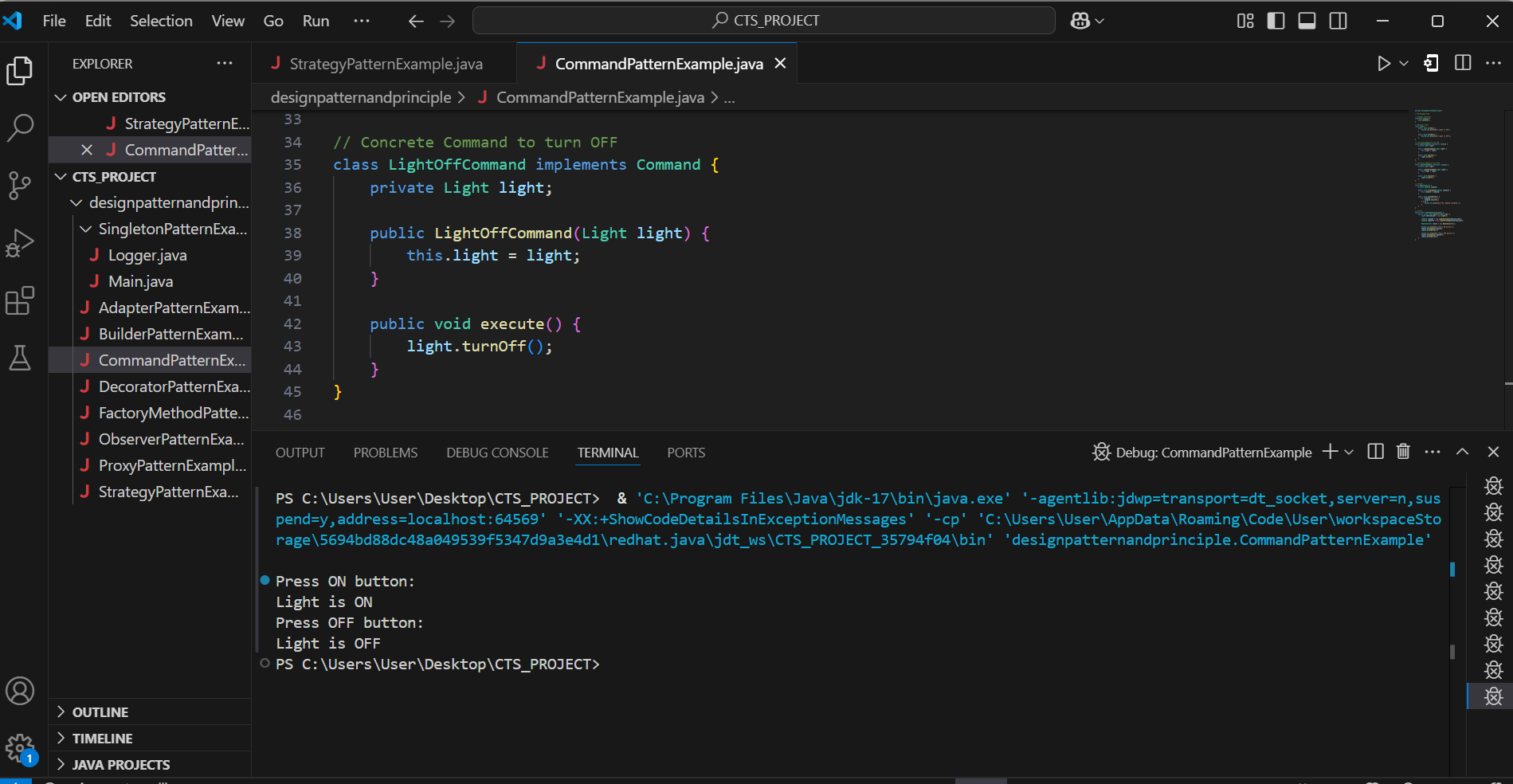
remote.setCommand(lightOff);

remote.pressButton();

}

}

**Output**



**Exercise 10: Implementing the MVC Pattern**

**MVCPatternExample.java**

// No package used

// Model

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 & Setters

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 setId(String id) { this.id = id; }

public void setGrade(String grade) { this.grade = grade; }

}

// View

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);

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

}

}

// Controller

class StudentController {

private Student student;

private StudentView view;

public StudentController(Student student, StudentView view) {

this.student = student;

this.view = view;

}

public void updateView() {

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

}

public void setStudentName(String name) {

student.setName(name);

}

public void setStudentGrade(String grade) {

student.setGrade(grade);

}

}

// Client

public class MVCPatternExample {

public static void main(String[] args) {

// Step 1: Create Model

Student student = new Student("Mathii", "CTS1023", "A");

// Step 2: Create View

StudentView view = new StudentView();

// Step 3: Create Controller

StudentController controller = new StudentController(student, view);

// Step 4: Show Initial Data

controller.updateView();

// Step 5: Update Data through Controller

controller.setStudentName("Mathi Selvaraj");

controller.setStudentGrade("A+");

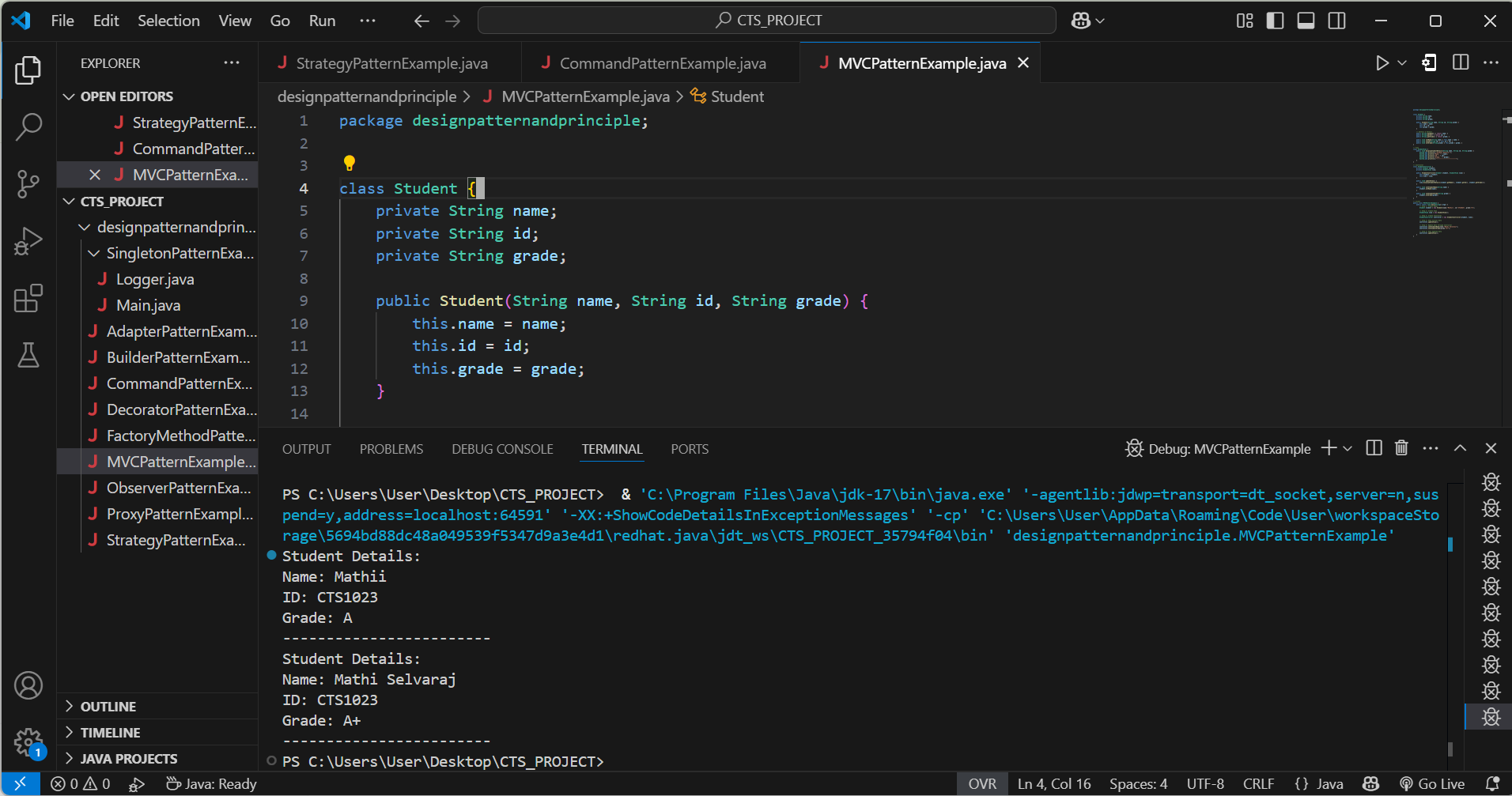
// Step 6: Show Updated Data

controller.updateView();

}

}

**Output**

****

**Exercise 11: Implementing Dependency Injection**

**DependencyInjectionExample.java**

// Repository Interface

interface CustomerRepository {

String findCustomerById(String id);

}

// Concrete Repository

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

// Dummy implementation

return "Customer{id='" + id + "', name='Mathii', email='mathii@example.com'}";

}

}

// Service Class (depends on Repository)

class CustomerService {

private CustomerRepository repository;

// Constructor-based Dependency Injection

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void getCustomerDetails(String id) {

String customer = repository.findCustomerById(id);

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

}

}

// Client

public class DependencyInjectionExample {

public static void main(String[] args) {

// Injecting dependency manually

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

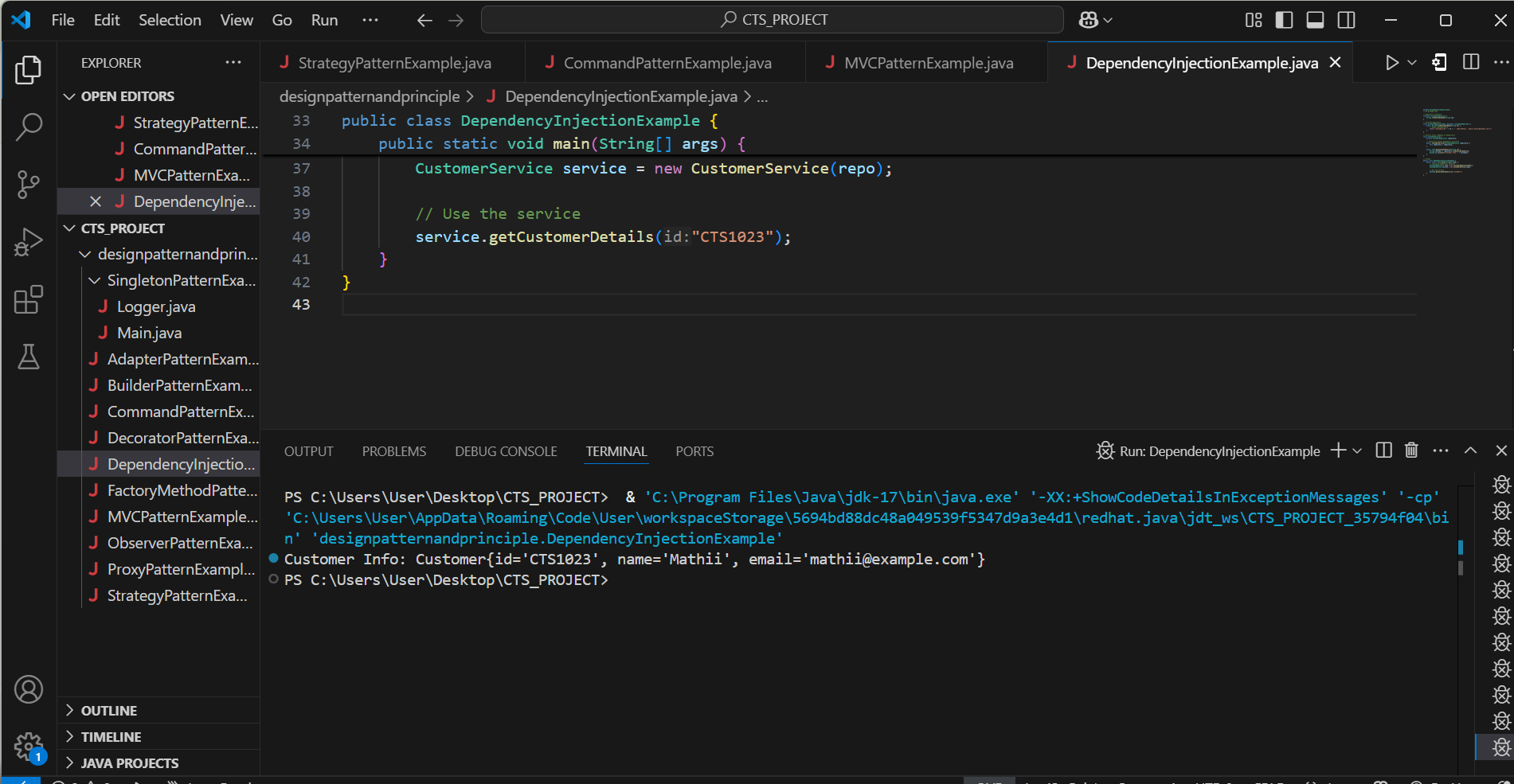
// Use the service

service.getCustomerDetails("CTS1023");

}

}

**Output**

****