**Exercise 1: Singleton Pattern – Logger**

Logger.java

public class Logger {

private static Logger instance;

private Logger() {}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

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

}

}

TestLogger.java

public class TestLogger {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

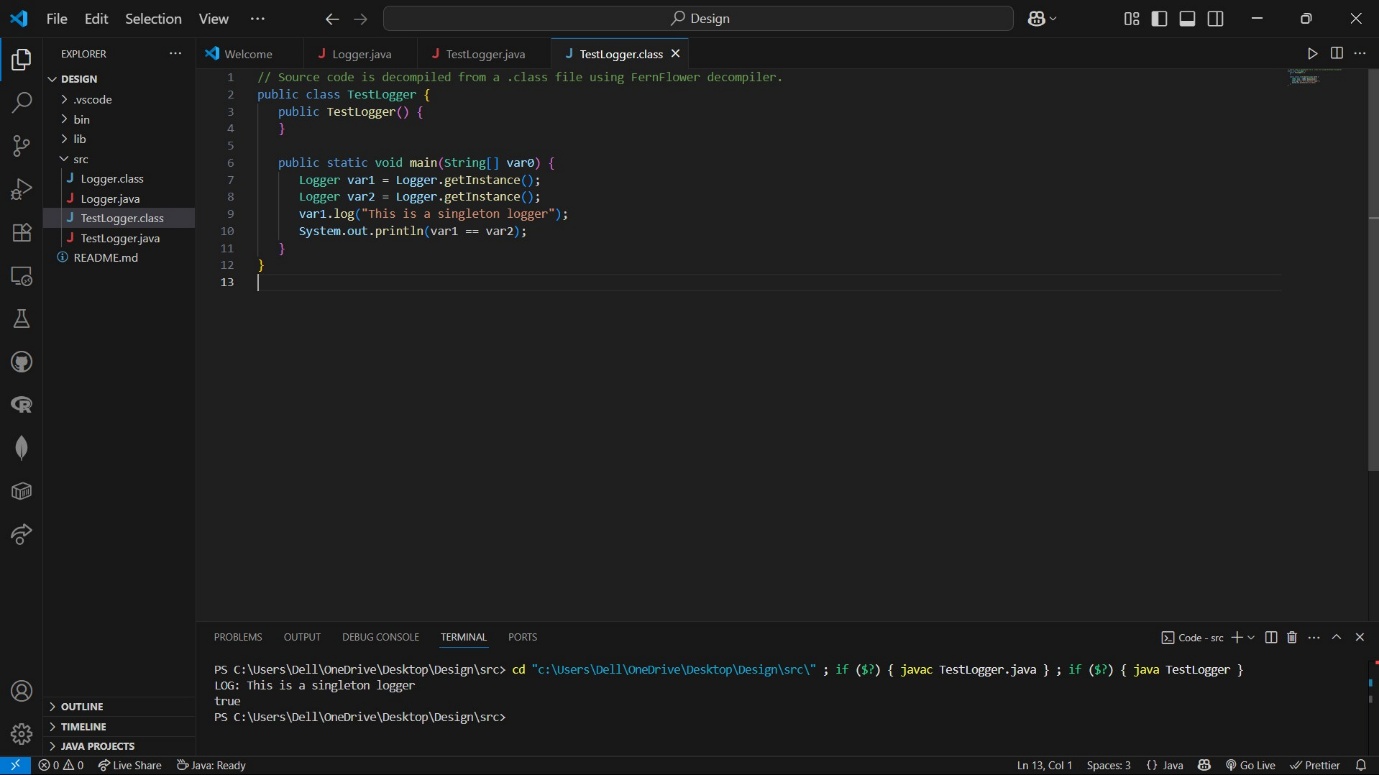
Logger logger2 = Logger.getInstance();

logger1.log("This is a singleton logger");

System.out.println(logger1 == logger2); Should print true

}

}



**Exercise 2: Factory Method Pattern – Documents**

Document.java

interface Document {

void open();

}

WordDocument.java

class WordDocument implements Document {

public void open() {

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

}

}

PdfDocument.java

class PdfDocument implements Document {

public void open() {

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

}

}

ExcelDocument.java

class ExcelDocument implements Document {

public void open() {

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

}

}

DocumentFactory.java

abstract class DocumentFactory {

abstract Document createDocument();

}

WordFactory.java

class WordFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

TestFactory.java

public class TestFactory {

public static void main(String[] args) {

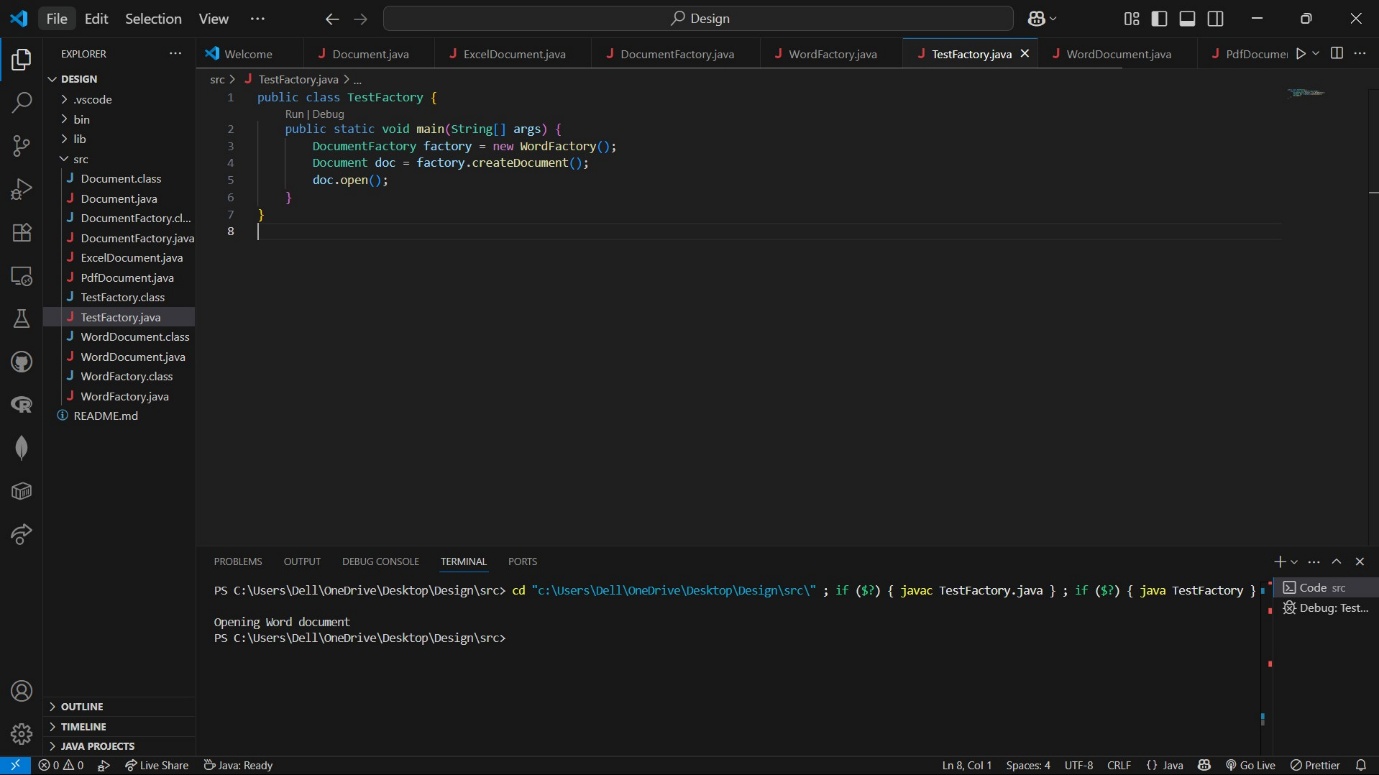
DocumentFactory factory = new WordFactory();

Document doc = factory.createDocument();

doc.open();

}

}



**Exercise 3: Builder Pattern – Computer**

Computer.java

public class Computer {

private String CPU, RAM, storage;

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

}

public static class Builder {

private String CPU, RAM, storage;

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 Computer build() {

return new Computer(this);

}

}

public void specs() {

System.out.println("CPU: " + CPU + ", RAM: " + RAM + ", Storage: " + storage);

}

}

TestBuilder.java

public class TestBuilder {

public static void main(String[] args) {

Computer pc = new Computer.Builder()

.setCPU("Intel i7")

.setRAM("16GB")

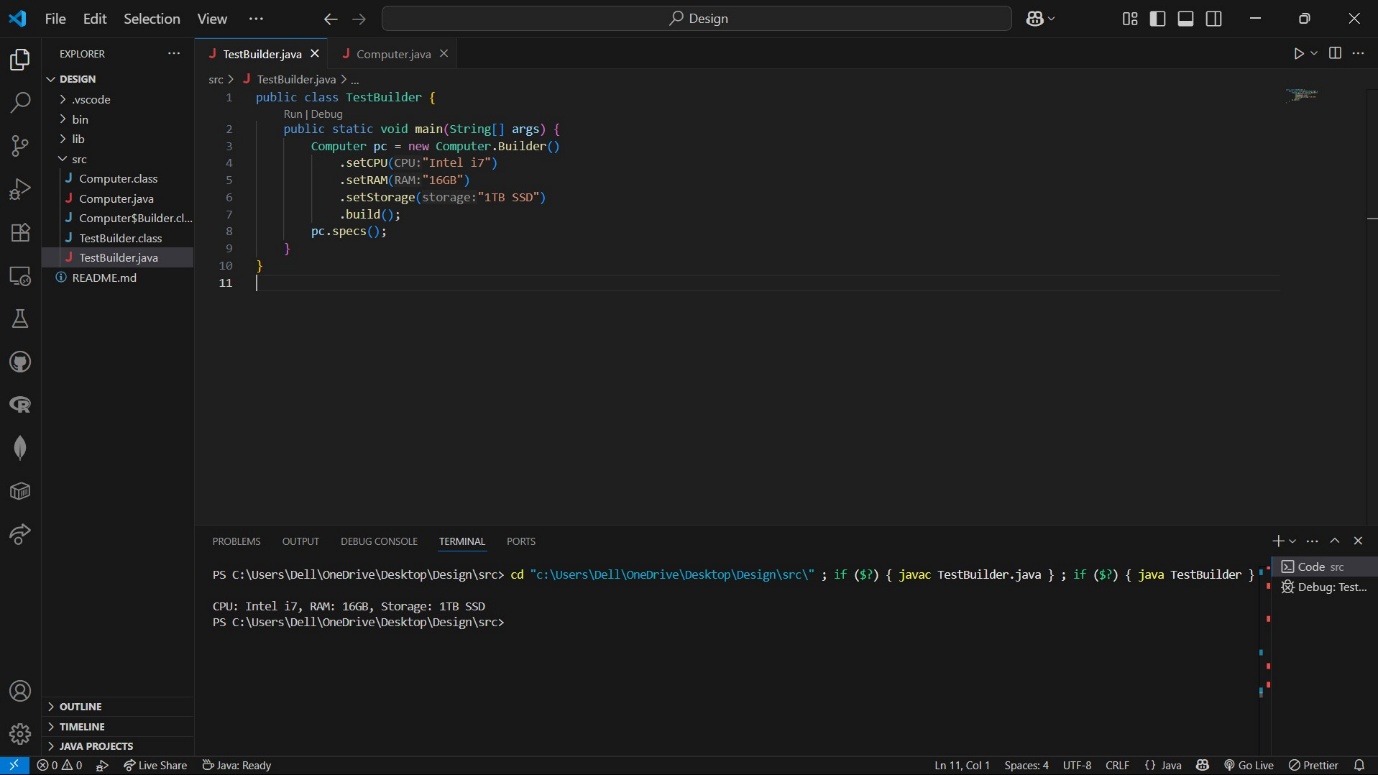
.setStorage("1TB SSD")

.build();

pc.specs();

}

}



**Exercise 4: Adapter Pattern – Payment Gateways**

PaymentProcessor.java

interface PaymentProcessor {

void processPayment();

}

PayPal.java (Adaptee)

class PayPal {

public void sendPayment() {

System.out.println("Paying via PayPal...");

}

}

PayPalAdapter.java

class PayPalAdapter implements PaymentProcessor {

private PayPal payPal = new PayPal();

public void processPayment() {

payPal.sendPayment();

}

}

TestAdapter.java

public class TestAdapter {

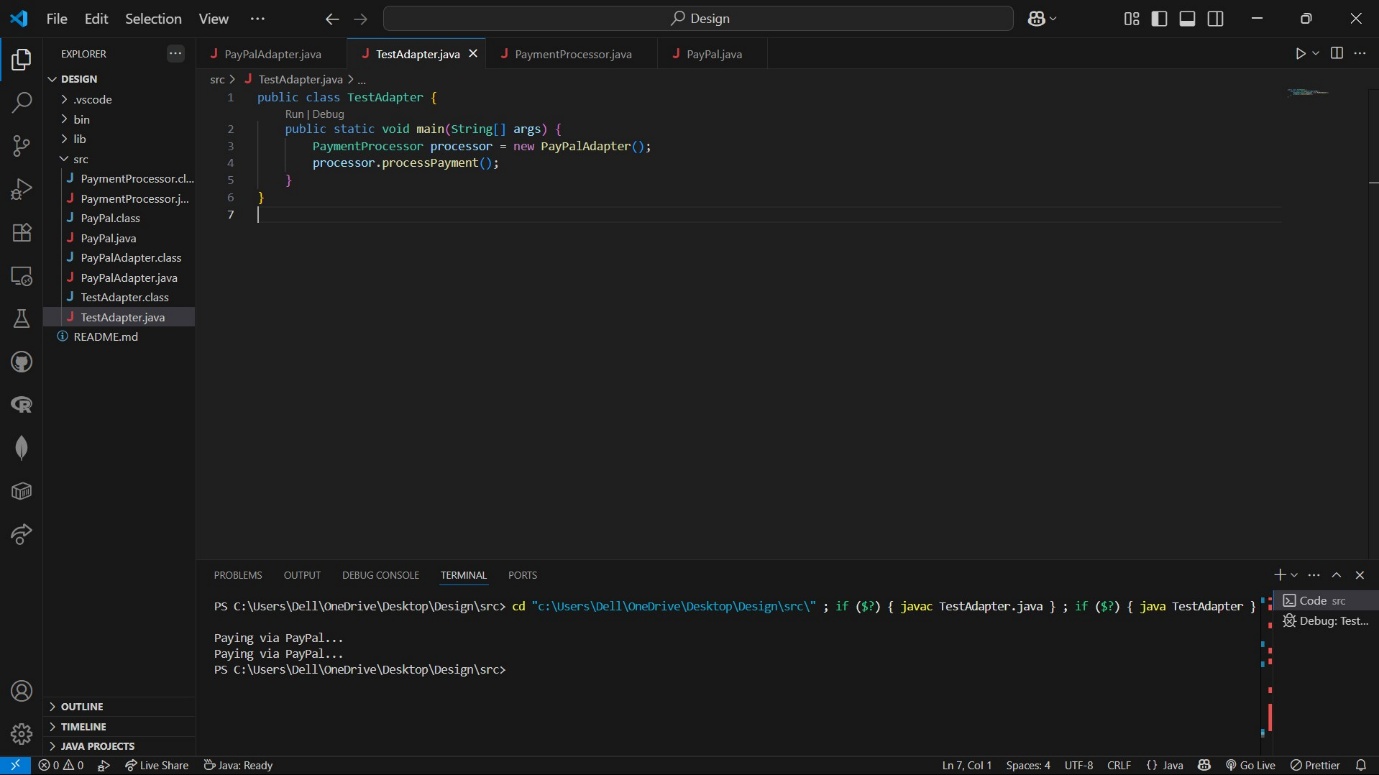
public static void main(String[] args) {

PaymentProcessor processor = new PayPalAdapter();

processor.processPayment();

}

}



**Exercise 5: Decorator Pattern – Notifications**

Notifier.java

interface Notifier {

void send(String message);

}

EmailNotifier.java

class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

NotifierDecorator.java

abstract class NotifierDecorator implements Notifier {

protected Notifier wrappee;

public NotifierDecorator(Notifier notifier) {

this.wrappee = notifier;

}

public void send(String message) {

wrappee.send(message);

}

}

SMSNotifierDecorator.java

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

TestDecorator.java

public class TestDecorator {

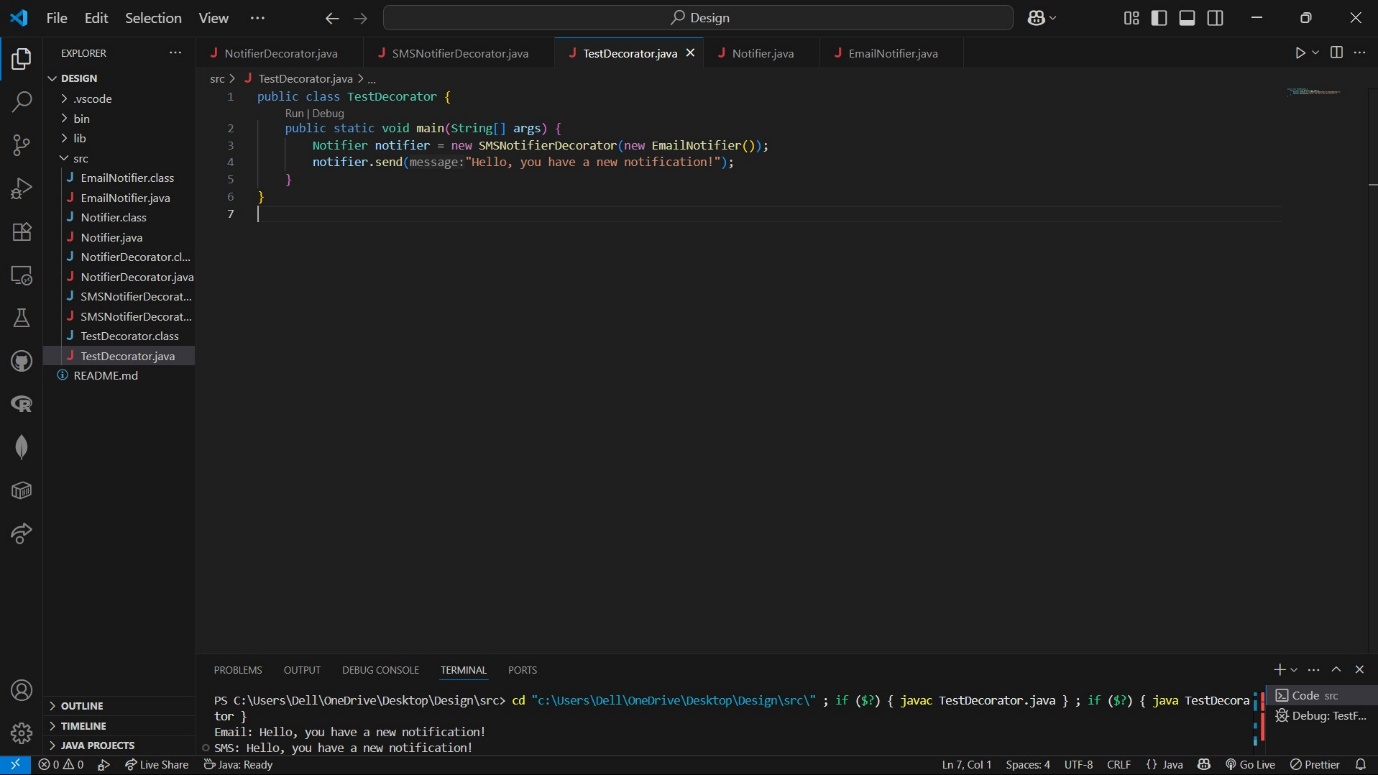
public static void main(String[] args) {

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

notifier.send("Hello, you have a new notification!");

}

}



**Exercise 6: Proxy Pattern – Image Viewer**

Image.java

interface Image {

void display();

}

RealImage.java

class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromDisk();

}

private void loadFromDisk() {

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

}

public void display() {

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

}

}

ProxyImage.java

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);

}

realImage.display();

}

}

TestProxy.java

public class TestProxy {

public static void main(String[] args) {

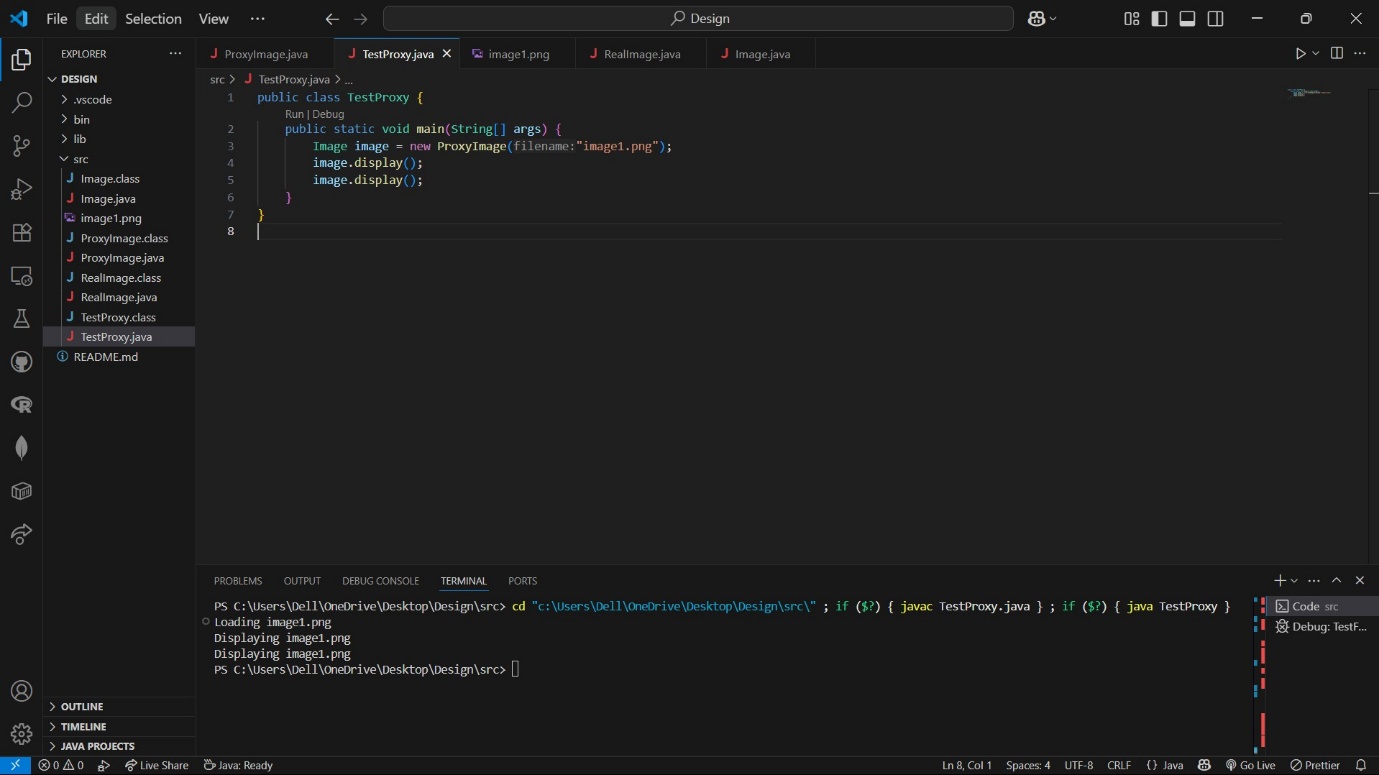
Image image = new ProxyImage("image1.png");

image.display(); Loads and displays

image.display(); Displays from cache

}

}



**Exercise 7: Observer Pattern – Stock Market**

Observer.java

interface Observer {

void update(float price);

}

Stock.java

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

StockMarket.java

import java.util.\*;

class StockMarket implements Stock {

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

private float price;

public void setPrice(float price) {

this.price = price;

notifyObservers();

}

public void register(Observer o) {

observers.add(o);

}

public void deregister(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(price);

}

}

}

MobileApp.java

class MobileApp implements Observer {

public void update(float price) {

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

}

}

TestObserver.java

public class TestObserver {

public static void main(String[] args) {

StockMarket market = new StockMarket();

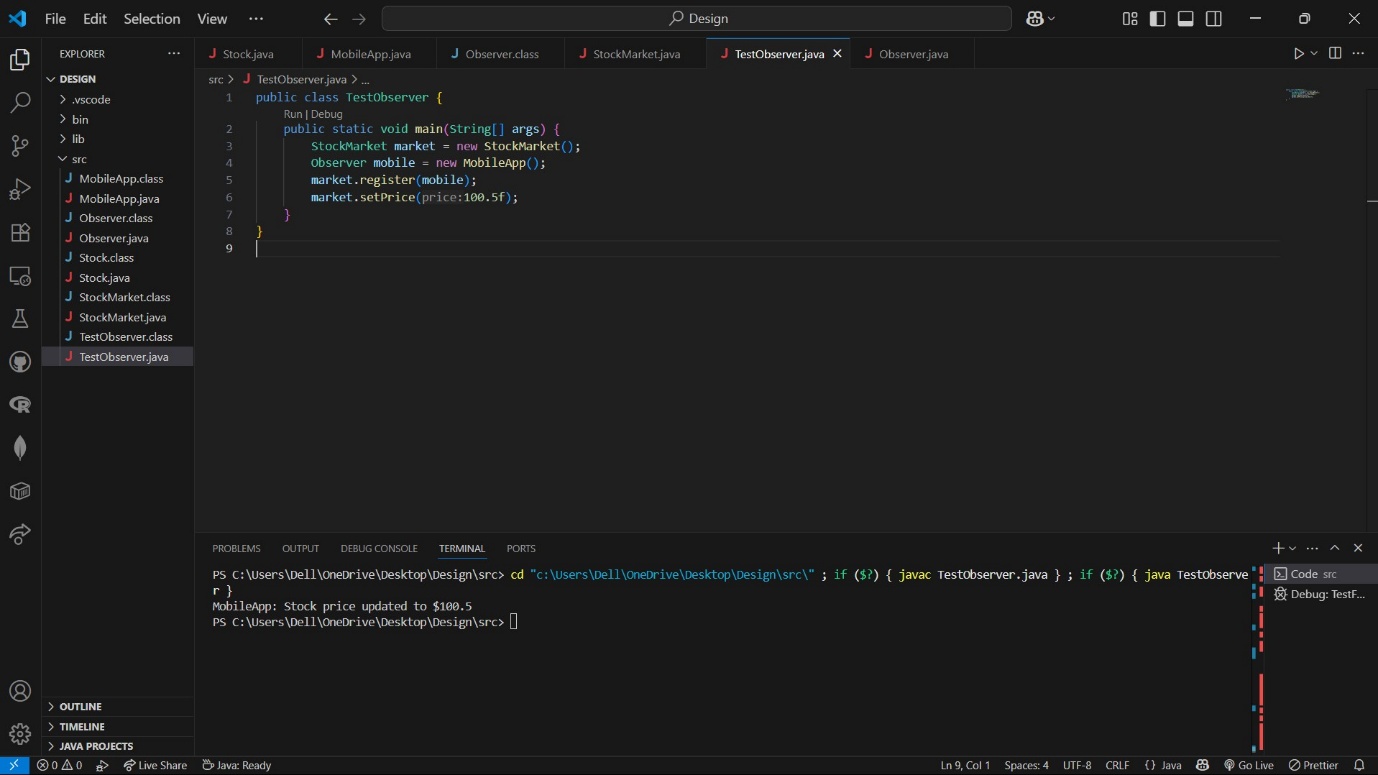
Observer mobile = new MobileApp();

market.register(mobile);

market.setPrice(100.5f);

}

}



**Exercise 8: Strategy Pattern – Payment**

PaymentStrategy.java

interface PaymentStrategy {

void pay(double amount);

}

CreditCardPayment.java

class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

System.out.println("Paid $" + amount + " with Credit Card");

}

}

PaymentContext.java

class PaymentContext {

private PaymentStrategy strategy;

public void setStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void pay(double amount) {

strategy.pay(amount);

}

}

TestStrategy.java

public class TestStrategy {

public static void main(String[] args) {

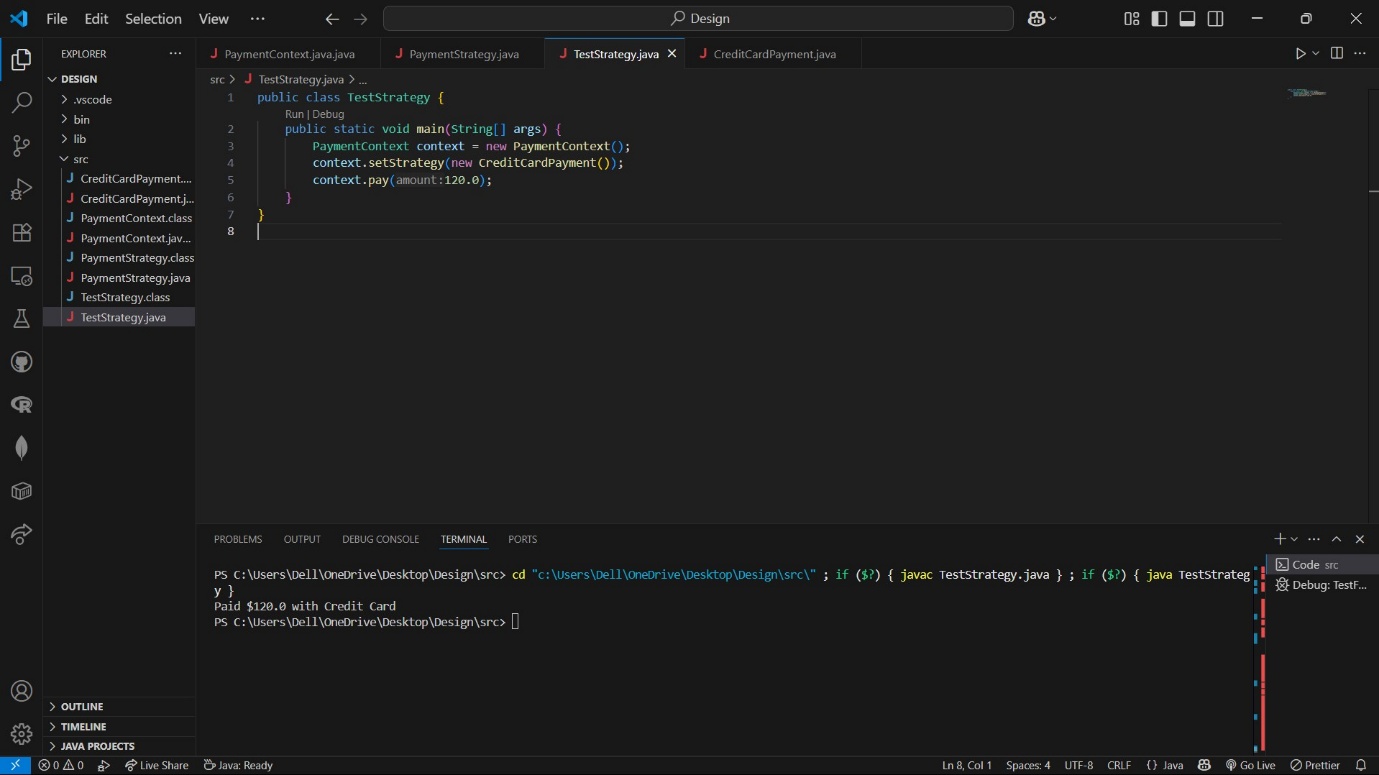
PaymentContext context = new PaymentContext();

context.setStrategy(new CreditCardPayment());

context.pay(120.0);

}

}



**Exercise 9: Command Pattern – Home Automation**

Command.java

interface Command {

void execute();

}

Light.java

class Light {

void on() {

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

}

void off() {

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

}

}

LightOnCommand.java

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.on();

}

}

RemoteControl.java

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

TestCommand.java

public class TestCommand {

public static void main(String[] args) {

Light light = new Light();

Command command = new LightOnCommand(light);

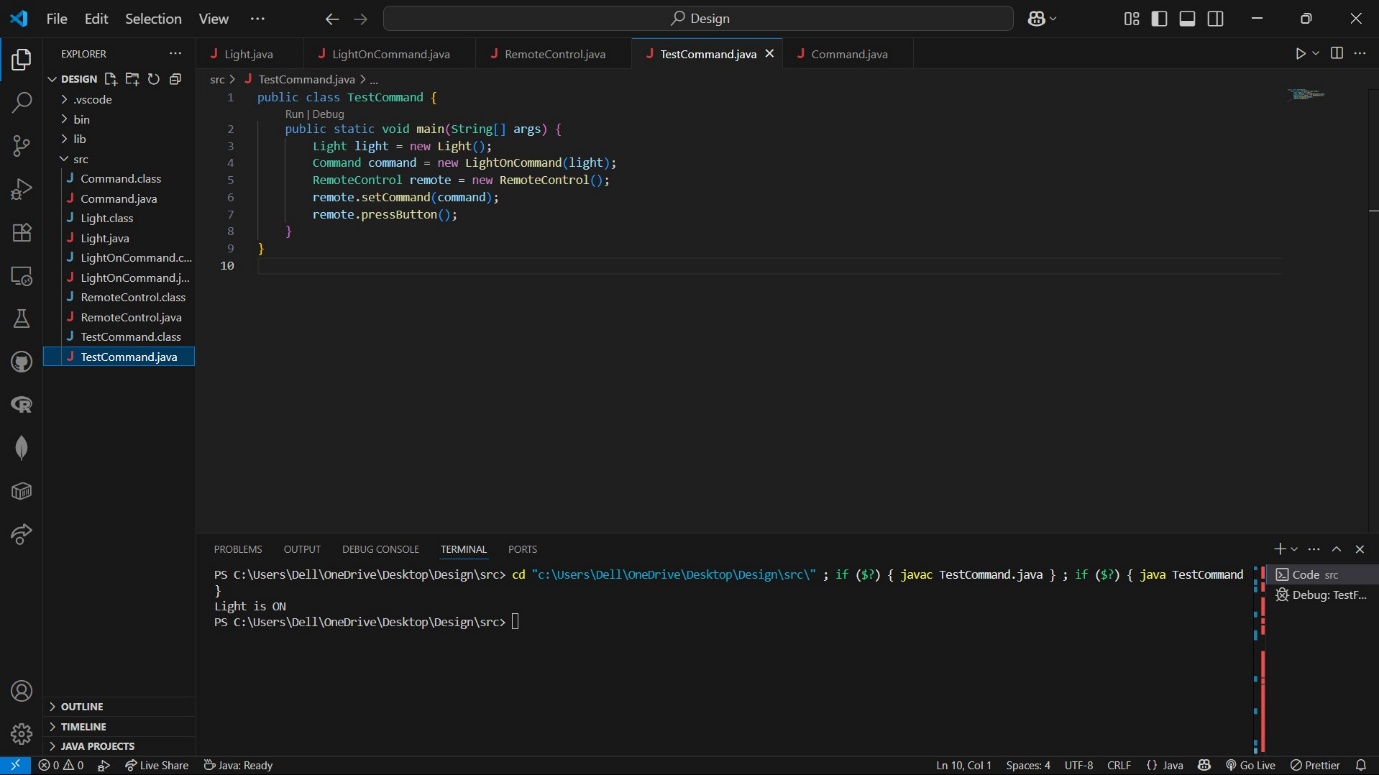
RemoteControl remote = new RemoteControl();

remote.setCommand(command);

remote.pressButton();

}

}



**Exercise 10: MVC Pattern – Student Records**

Student.java

class Student {

private String name;

private int id;

private String grade;

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

this.name = name;

this.id = id;

this.grade = grade;

}

public String getName() { return name; }

public int getId() { return id; }

public String getGrade() { return grade; }

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

}

StudentView.java

class StudentView {

public void displayStudentDetails(Student s) {

System.out.println("Student: " + s.getName() + ", ID: " + s.getId() + ", Grade: " + s.getGrade());

}

}

StudentController.java

class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void updateGrade(String grade) {

model.setGrade(grade);

}

public void display() {

view.displayStudentDetails(model);

}

}

TestMVC.java

public class TestMVC {

public static void main(String[] args) {

Student s = new Student("Alice", 1, "B");

StudentView v = new StudentView();

StudentController c = new StudentController(s, v);

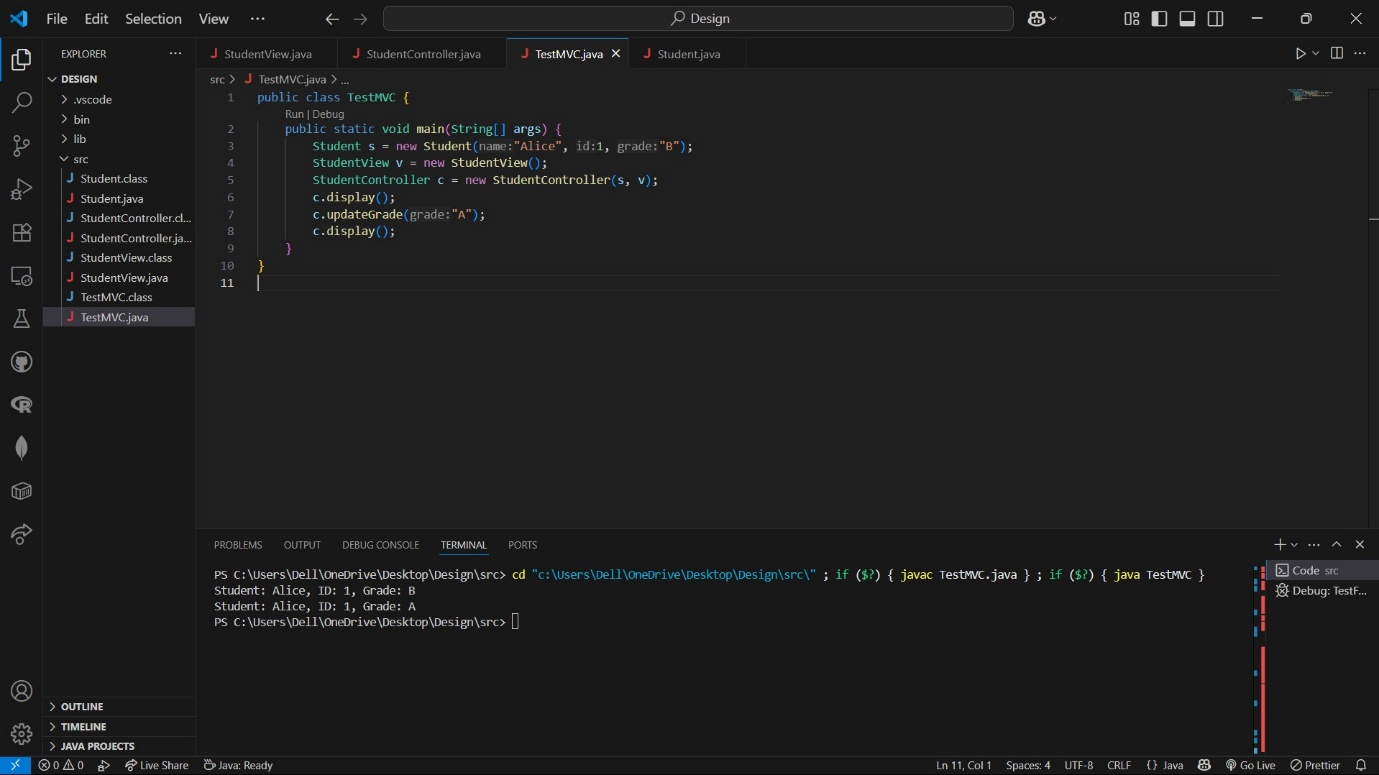
c.display();

c.updateGrade("A");

c.display();

}

}



**Exercise 11: Dependency Injection – Customer Management**

CustomerRepository.java

interface CustomerRepository {

String findCustomerById(int id);

}

CustomerRepositoryImpl.java

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(int id) {

return "Customer#" + id;

}

}

CustomerService.java

class CustomerService {

private CustomerRepository repo;

public CustomerService(CustomerRepository repo) {

this.repo = repo;

}

public void getCustomer(int id) {

System.out.println("Found: " + repo.findCustomerById(id));

}

}

TestDI.java

public class TestDI {

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.getCustomer(101);

}

}

