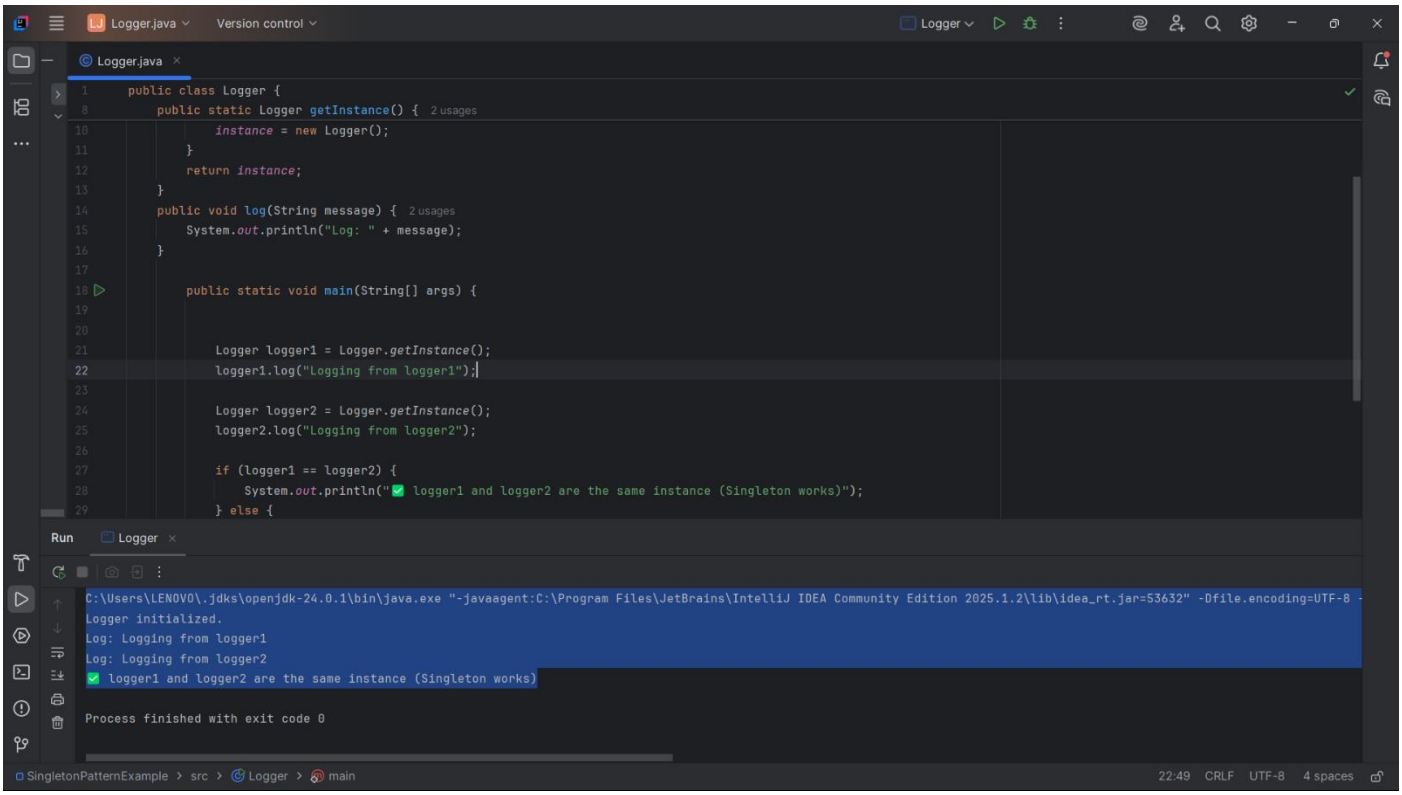
DESIGN PRINCIPLES AND DESIGN PATTERNS EXERCISES

Exercise 1: Implementing the Singleton Pattern

```
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);
    }

        public static void main(String[] args) {


            Logger logger1 = Logger.getInstance();
            logger1.log("Logging from logger1");

            Logger logger2 = Logger.getInstance();
            logger2.log("Logging from logger2");

            if (logger1 == logger2) {
                System.out.println("☑ logger1 and logger2 are the same instance
(Singleton works)");
            } else {
                System.out.println("❌ Different instances detected (Singleton failed)");
            }
        }
    }
```

OUTPUT:

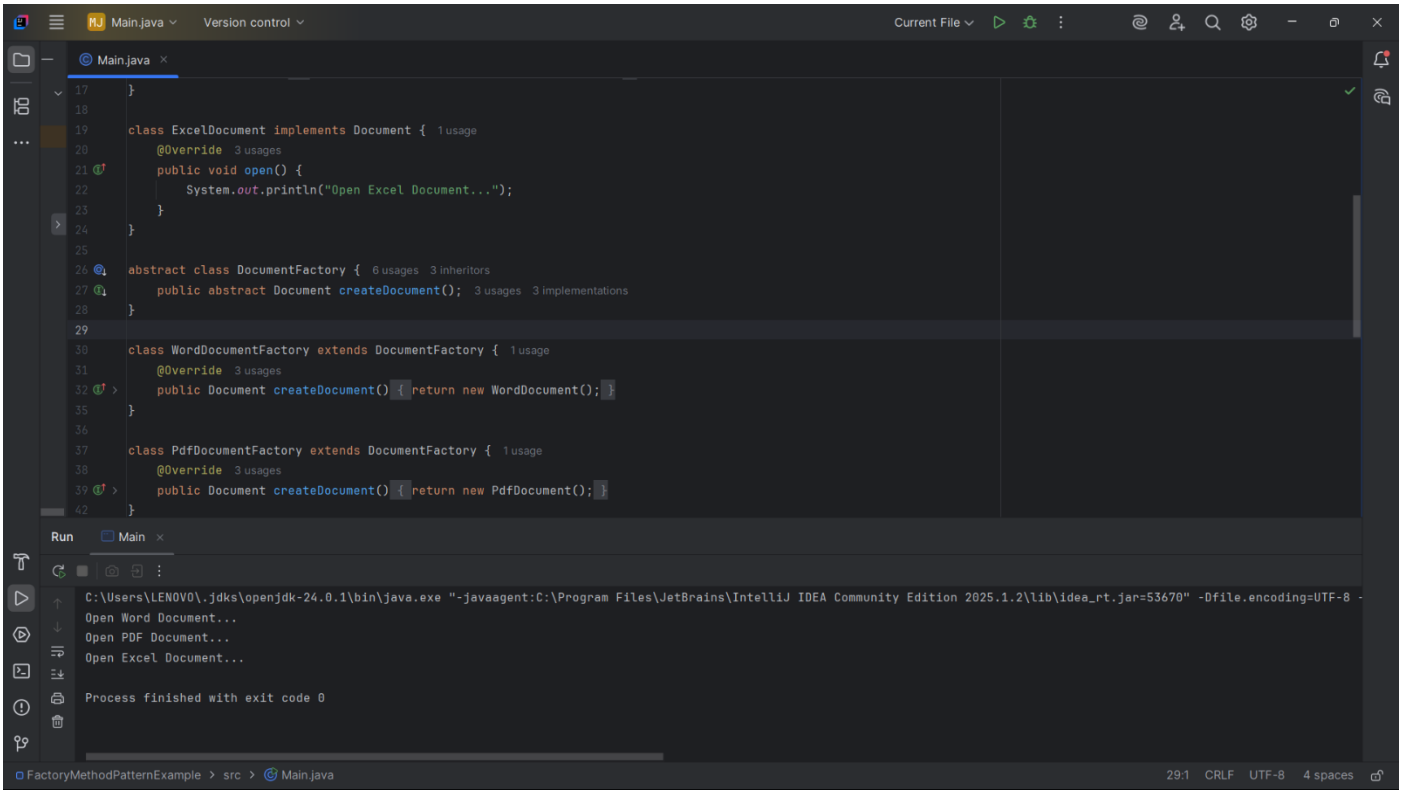


Exercise 2: Implementing the Factory Method Pattern

```
interface Document {
    void open();
}

class WordDocument implements Document {
    @Override
    public void open() {
        System.out.println("Open Word Document...");
    }
}

class PdfDocument implements Document {
    @Override
    public void open() {
        System.out.println("Open PDF Document...");
    }
}

class ExcelDocument implements Document {
    @Override
    public void open() {
        System.out.println("Open Excel Document...");
    }
}
```

```
abstract class DocumentFactory {
    public abstract Document createDocument();
}

class WordDocumentFactory extends DocumentFactory {
    @Override
    public Document createDocument() {
        return new WordDocument();
    }
}

class PdfDocumentFactory extends DocumentFactory {
    @Override
    public Document createDocument() {
        return new PdfDocument();
    }
}

class ExcelDocumentFactory extends DocumentFactory {
    @Override
    public Document createDocument() {
        return new ExcelDocument();
    }
}

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();
    }
}
```

OUTPUT:

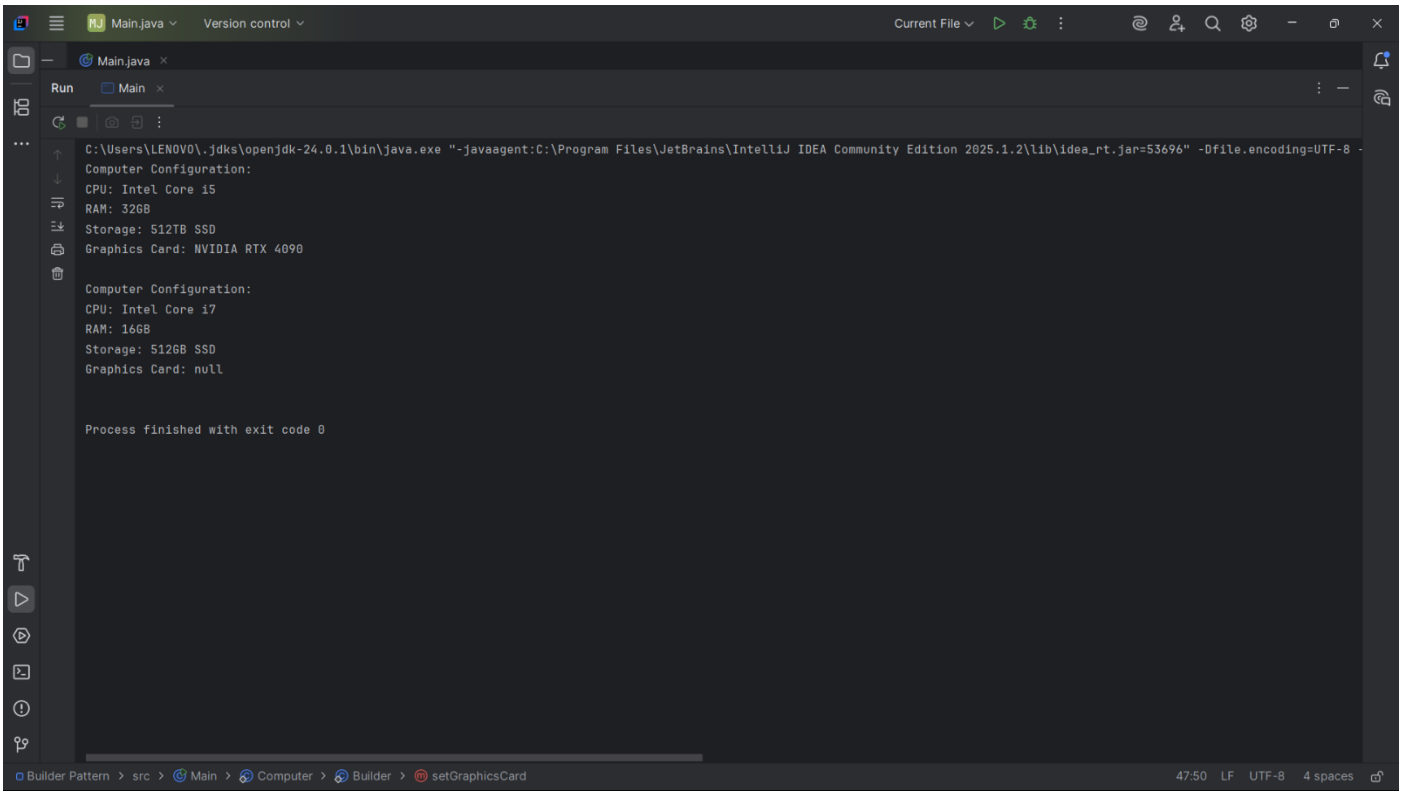


Exercise 3: Implementing the Builder Pattern

```
public class Main {

    static class Computer {
        private final String CPU;
        private final String RAM;
        private final String storage;
        private final String graphicsCard;

        private Computer(Builder builder) {
            this.CPU = builder.CPU;
            this.RAM = builder.RAM;
            this.storage = builder.storage;
            this.graphicsCard = builder.graphicsCard;
        }

        public void showConfig() {
            System.out.println("Computer Configuration:");
            System.out.println("CPU: " + CPU);
            System.out.println("RAM: " + RAM);
            System.out.println("Storage: " + storage);
            System.out.println("Graphics Card: " + graphicsCard);
            System.out.println();
        }

        static class Builder {
```

```
            private String CPU;
            private String RAM;
            private String storage;
            private String graphicsCard;

            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 setGraphicsCard(String graphicsCard) {
                this.graphicsCard = graphicsCard;
                return this;
            }

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

    public static void main(String[] args) {
        Computer gamingPC = new Computer.Builder()
                .setCPU("Intel Core i5")
                .setRAM("32GB")
                .setStorage("512TB SSD")
                .setGraphicsCard("NVIDIA RTX 4090")
                .build();

        Computer officePC = new Computer.Builder()
                .setCPU("Intel Core i7")
                .setRAM("16GB")
                .setStorage("512GB SSD")
                .build();

        gamingPC.showConfig();
        officePC.showConfig();
    }
}
```

OUTPUT:

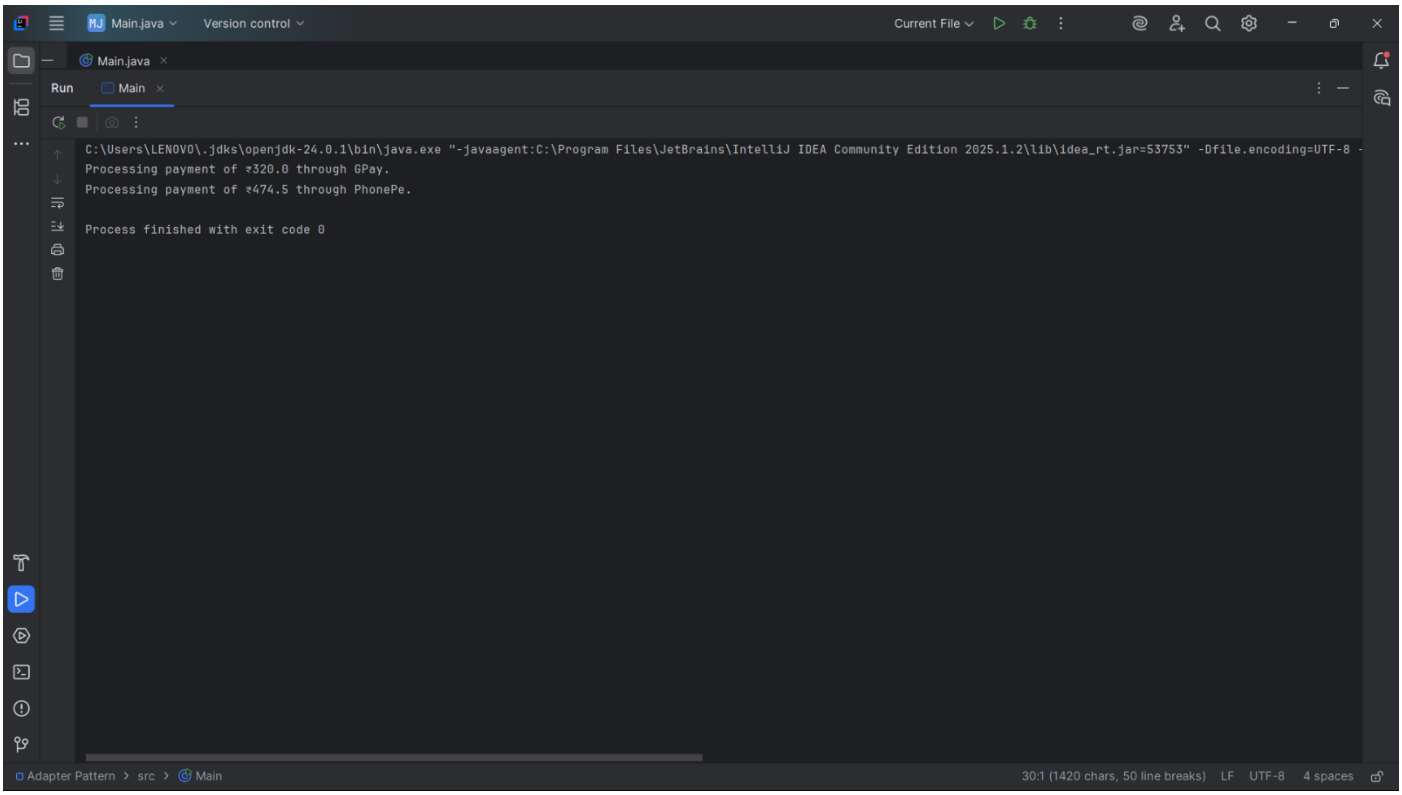


Exercise 4: Implementing the Adapter Pattern

```
public class Main {

    interface PaymentProcessor {
        void processPayment(double amount);
    }

    static class GPayGateway {
        void makePayment(double amount) {
            System.out.println("Processing payment of ₹" + amount + " through GPay.");
        }
    }

    static class PhonePeGateway {
        void doTransaction(double amount) {
            System.out.println("Processing payment of ₹" + amount + " through PhonePe.");
        }
    }

    static class GPayAdapter implements PaymentProcessor {
        private GPayGateway gPayGateway;

        public GPayAdapter(GPayGateway gPayGateway) {
            this.gPayGateway = gPayGateway;
        }

        public void processPayment(double amount) {
```

```
            gPayGateway.makePayment(amount);
        }
    }

    static class PhonePeAdapter implements PaymentProcessor {
        private PhonePeGateway phonePeGateway;

        public PhonePeAdapter(PhonePeGateway phonePeGateway) {
            this.phonePeGateway = phonePeGateway;
        }

        public void processPayment(double amount) {
            phonePeGateway.doTransaction(amount);
        }
    }

    public static void main(String[] args) {
        PaymentProcessor gpay = new GPayAdapter(new GPayGateway());
        gpay.processPayment(320.00);

        PaymentProcessor phonePe = new PhonePeAdapter(new PhonePeGateway());
        phonePe.processPayment(474.50);
    }
}
```

OUTPUT:

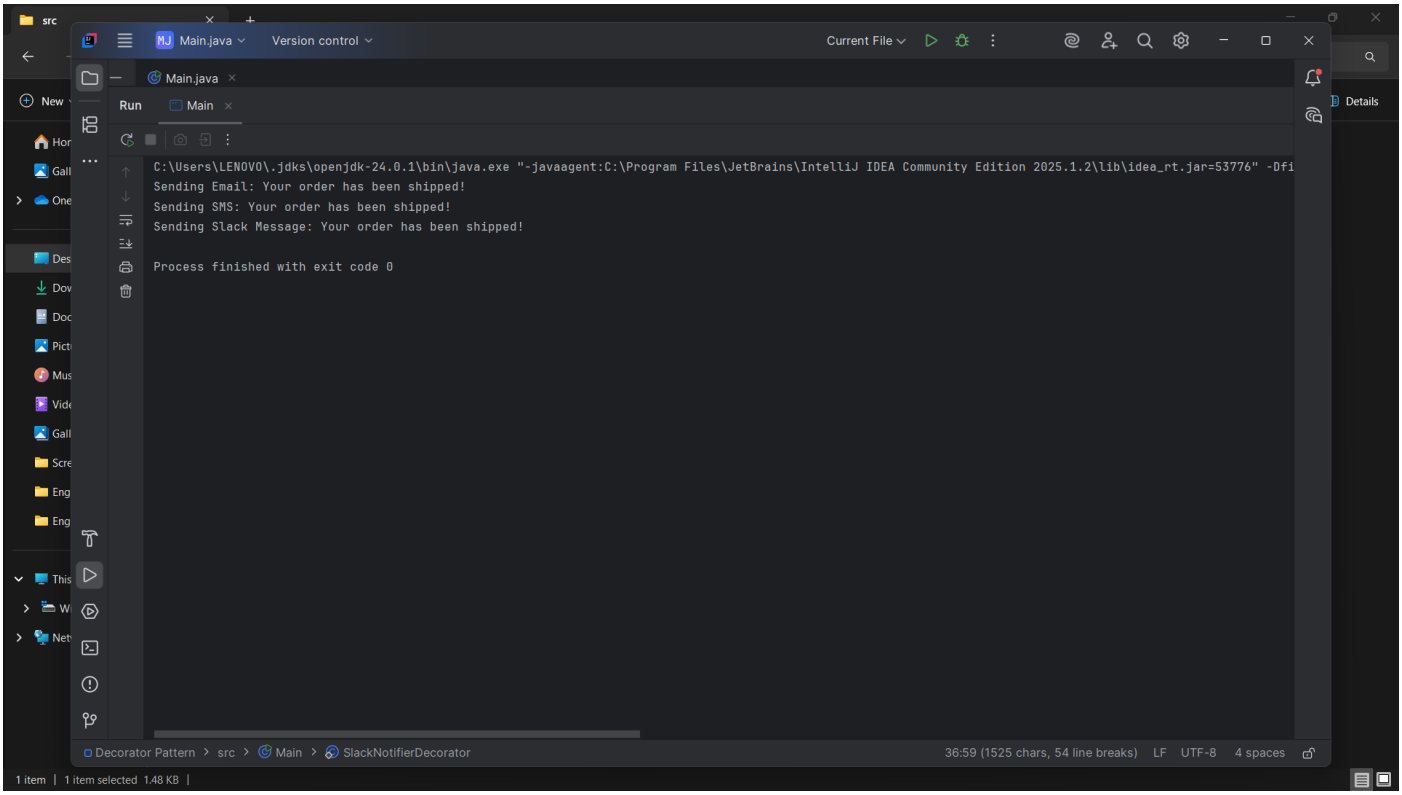


Exercise 5: Implementing the Decorator Pattern

```
public class Main {

    interface Notifier {
        void send(String message);
    }

    static class EmailNotifier implements Notifier {
        public void send(String message) {
            System.out.println("Sending Email: " + message);
        }
    }

    static abstract class NotifierDecorator implements Notifier {
        protected Notifier notifier;

        public NotifierDecorator(Notifier notifier) {
            this.notifier = notifier;
        }

        public void send(String message) {
            notifier.send(message);
        }
    }

    static class SMSNotifierDecorator extends NotifierDecorator {
        public SMSNotifierDecorator(Notifier notifier) {
            super(notifier);
        }

        public void send(String message) {
            super.send(message);
            System.out.println("Sending SMS: " + message);
        }
    }

    static class SlackNotifierDecorator extends NotifierDecorator {
        public SlackNotifierDecorator(Notifier notifier) {
            super(notifier);
        }

        public void send(String message) {
            super.send(message);
            System.out.println("Sending Slack Message: " + message);
        }
    }

    public static void main(String[] args) {
        Notifier notifier = new EmailNotifier();
        Notifier smsNotifier = new SMSNotifierDecorator(notifier);
        Notifier slackAndSmsNotifier = new SlackNotifierDecorator(smsNotifier);

        slackAndSmsNotifier.send("Your order has been shipped!");
```

```
    }
}
```

OUTPUT:

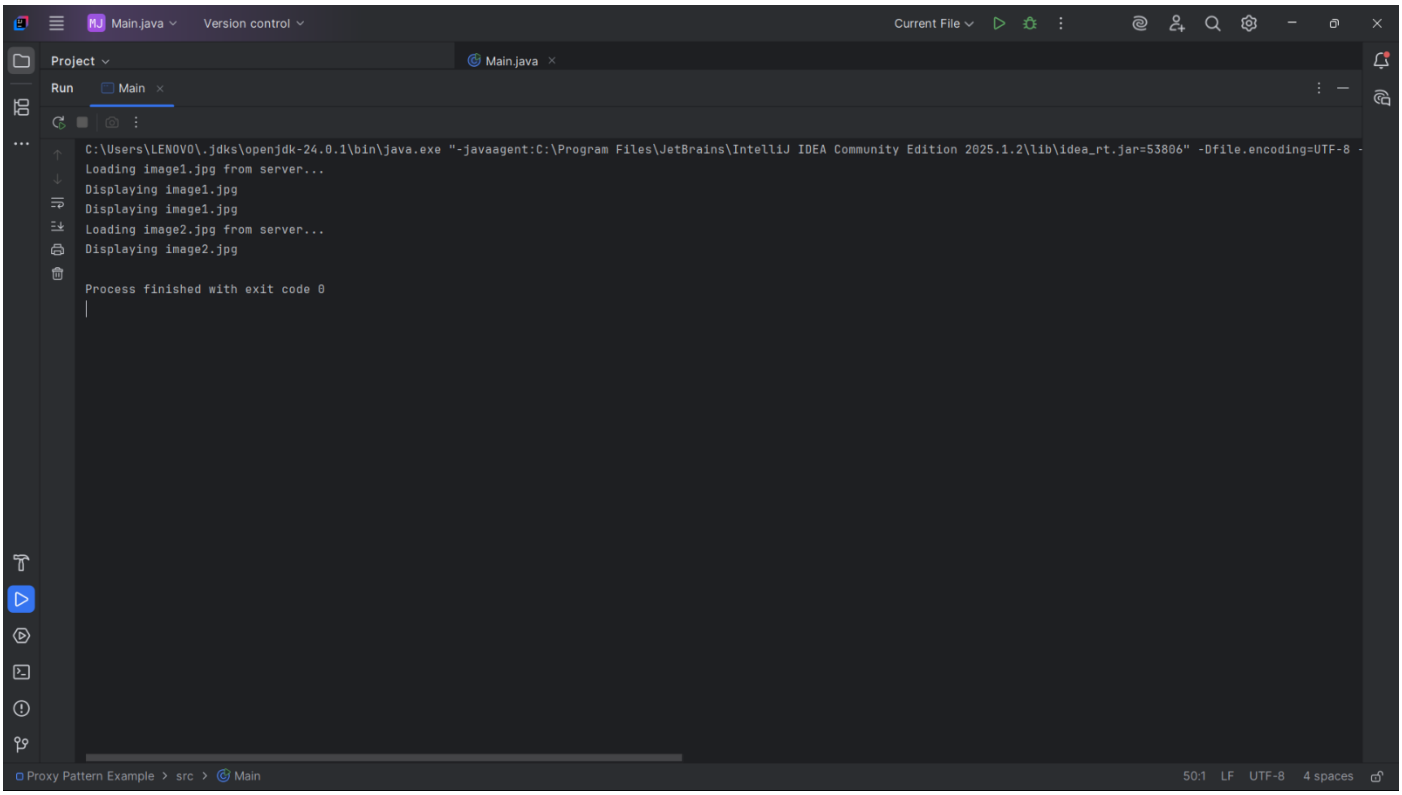


Exercise 6: Implementing the Proxy Pattern

```
public class Main {
    interface Image {
        void display();
    }

    static class RealImage implements Image {
        private final String filename;

        public RealImage(String filename) {
            this.filename = filename;
            loadFromServer();
        }

        private void loadFromServer() {
            System.out.println("Loading " + filename + " from server...");
        }

        @Override
        public void display() {
            System.out.println("Displaying " + filename);
```

```
        }
    }

    static class ProxyImage implements Image {
        private final 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();
        }
    }

    public static void main(String[] args) {
        Image image1 = new ProxyImage("image1.jpg");
        image1.display();  // Loads from server
        image1.display();  // Uses cache

        Image image2 = new ProxyImage("image2.jpg");
        image2.display();  // Loads from server
    }
}
```

OUTPUT:

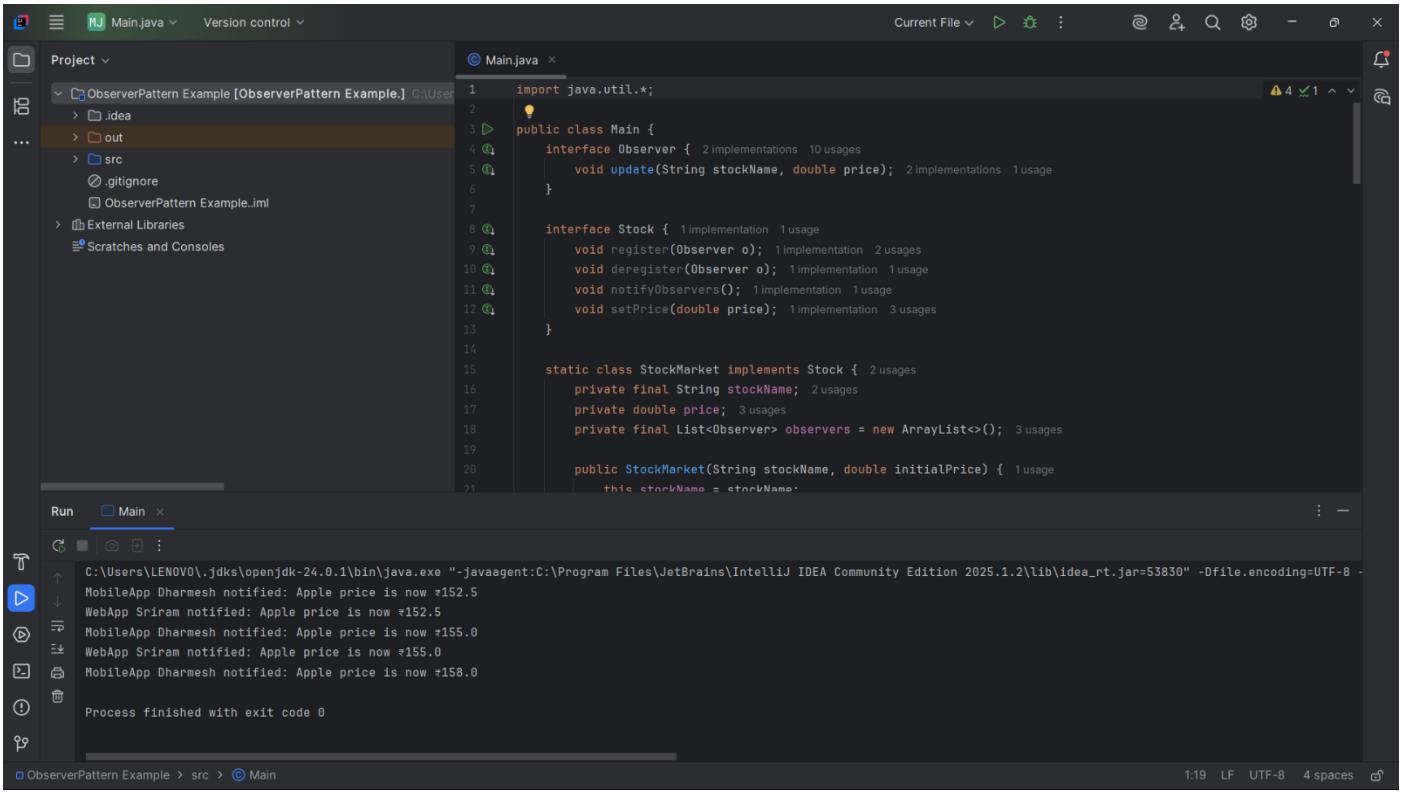


Exercise 7: Implementing the Observer Pattern

```
import java.util.*;

public class Main {
    interface Observer {
        void update(String stockName, double price);
    }

    interface Stock {
        void register(Observer o);
        void deregister(Observer o);
        void notifyObservers();
        void setPrice(double price);
    }

    static class StockMarket implements Stock {
        private final String stockName;
        private double price;
        private final List<Observer> observers = new ArrayList<>();

        public StockMarket(String stockName, double initialPrice) {
            this.stockName = stockName;
            this.price = initialPrice;
        }

        @Override
        public void register(Observer o) {
            observers.add(o);
        }

        @Override
        public void deregister(Observer o) {
            observers.remove(o);
        }

        @Override
        public void notifyObservers() {
            for (Observer o : observers) {
                o.update(stockName, price);
            }
        }

        @Override
        public void setPrice(double price) {
            this.price = price;
            notifyObservers();
        }
    }

    static class MobileApp implements Observer {
        private final String user;
```

```
        public MobileApp(String user) {
            this.user = user;
        }

        @Override
        public void update(String stockName, double price) {
            System.out.println("MobileApp " + user + " notified: " + stockName + " price
is now ₹" + price);
        }
    }

    static class WebApp implements Observer {
        private final String user;

        public WebApp(String user) {
            this.user = user;
        }

        @Override
        public void update(String stockName, double price) {
            System.out.println("WebApp " + user + " notified: " + stockName + " price is
now ₹" + price);
        }
    }

    public static void main(String[] args) {
        StockMarket appleStock = new StockMarket("Apple", 150.0);

        Observer mobileUser = new MobileApp("Dharmesh");
        Observer webUser = new WebApp("Sriram");

        appleStock.register(mobileUser);
        appleStock.register(webUser);

        appleStock.setPrice(152.5);
        appleStock.setPrice(155.0);

        appleStock.deregister(webUser);
        appleStock.setPrice(158.0);
    }
}
```

OUTPUT:

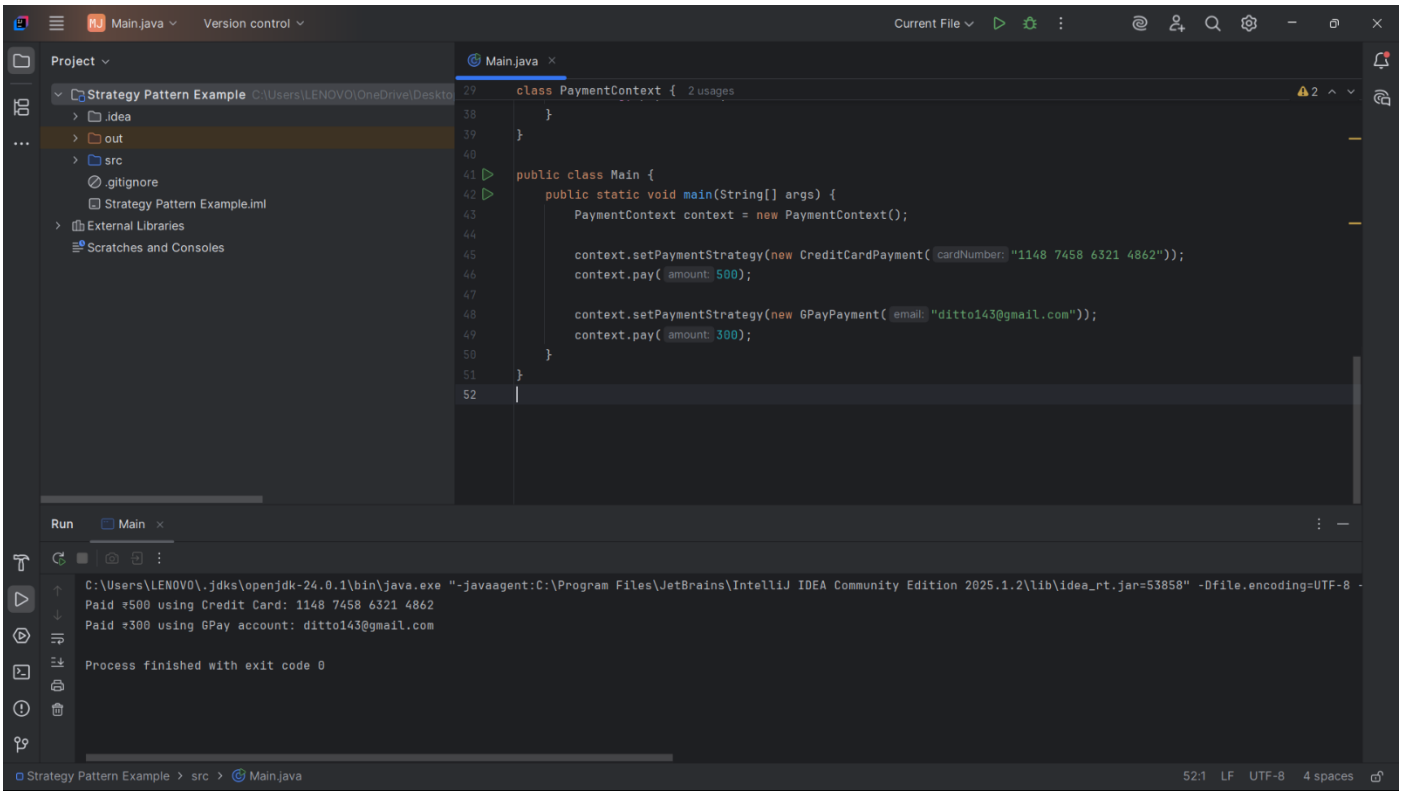


Exercise 8: Implementing the Strategy Pattern

```
interface PaymentStrategy {
    void pay(int amount);
}

class CreditCardPayment implements PaymentStrategy {
    private String cardNumber;

    public CreditCardPayment(String cardNumber) {
        this.cardNumber = cardNumber;
    }

    public void pay(int amount) {
        System.out.println("Paid ₹" + amount + " using Credit Card: " + cardNumber);
    }
}

class GPayPayment implements PaymentStrategy {
    private String email;

    public GPayPayment(String email) {
        this.email = email;
    }

    public void pay(int amount) {
        System.out.println("Paid ₹" + amount + " using GPay account: " + email);
    }
}
```

```
class PaymentContext {
    private PaymentStrategy strategy;

    public void setPaymentStrategy(PaymentStrategy strategy) {
        this.strategy = strategy;
    }

    public void pay(int amount) {
        strategy.pay(amount);
    }
}

public class Main {
    public static void main(String[] args) {
        PaymentContext context = new PaymentContext();

        context.setPaymentStrategy(new CreditCardPayment("1148 7458 6321 4862"));
        context.pay(500);

        context.setPaymentStrategy(new GPayPayment("ditto143@gmail.com"));
        context.pay(300);
    }
}
```

OUTPUT:



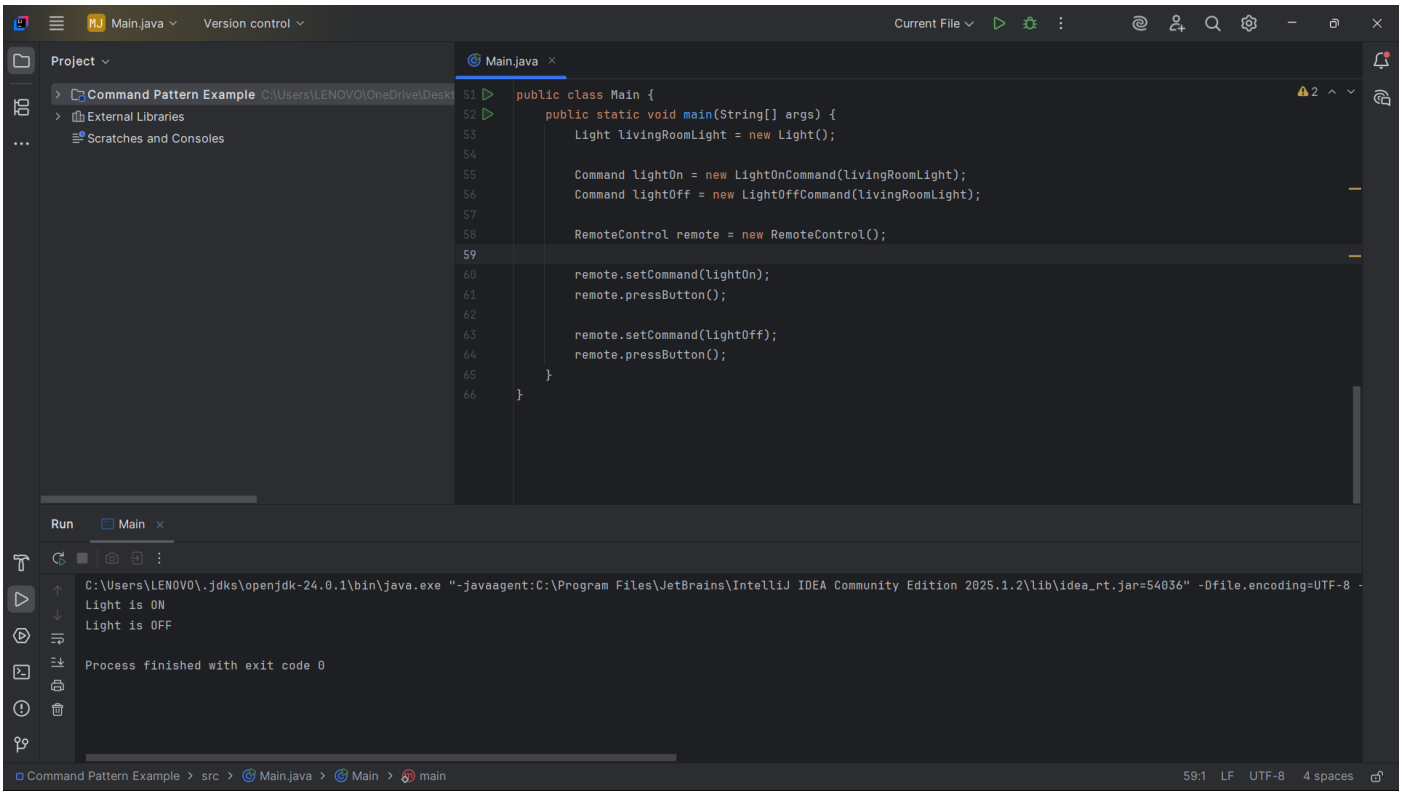
Exercise 9: Implementing the Command Pattern

```
interface Command {
    void execute();
```

```
}

class Light {
    public void turnOn() {
        System.out.println("Light is ON");
    }

    public void turnOff() {
        System.out.println("Light is OFF");
    }
}

class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.turnOn();
    }
}

class LightOffCommand implements Command {
    private Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.turnOff();
    }
}

class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}

public class Main {
    public static void main(String[] args) {
        Light livingRoomLight = new Light();

        Command lightOn = new LightOnCommand(livingRoomLight);
        Command lightOff = new LightOffCommand(livingRoomLight);
```

```
        RemoteControl remote = new RemoteControl();

        remote.setCommand(lightOn);
        remote.pressButton();

        remote.setCommand(lightOff);
        remote.pressButton();
    }
}
```

OUTPUT:



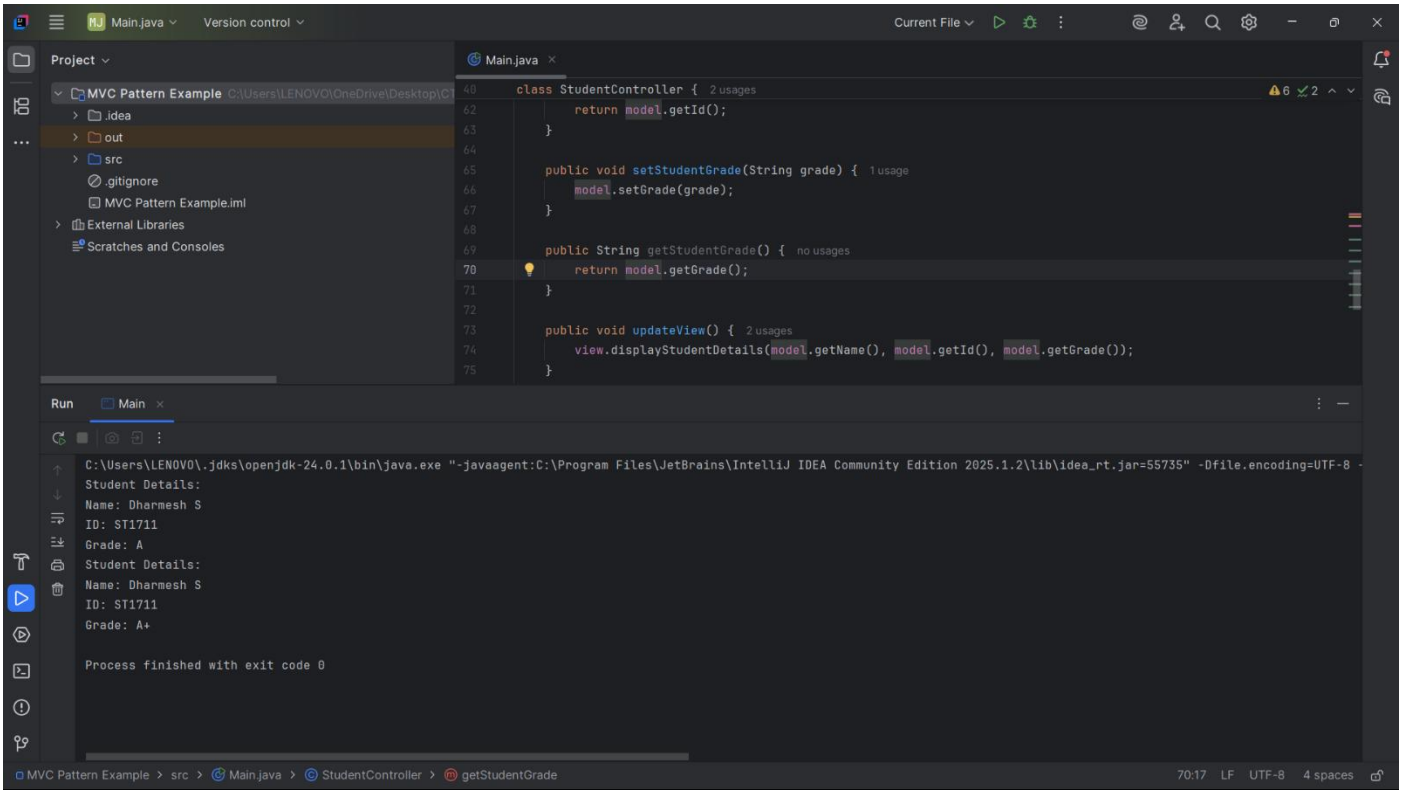
Exercise 10: Implementing the MVC Pattern

```
class Student {
    private String name;
    private String id;
    private String grade;

    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;
    }
}

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 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());
    }
}

public class Main {
    public static void main(String[] args) {
        Student model = new Student();
        model.setName("Dharmesh S");
        model.setId("ST1711");
        model.setGrade("A");

        StudentView view = new StudentView();
        StudentController controller = new StudentController(model, view);

        controller.updateView();

        controller.setStudentGrade("A+");
        controller.setStudentName("Dharmesh S ");

        controller.updateView();
    }
}
```

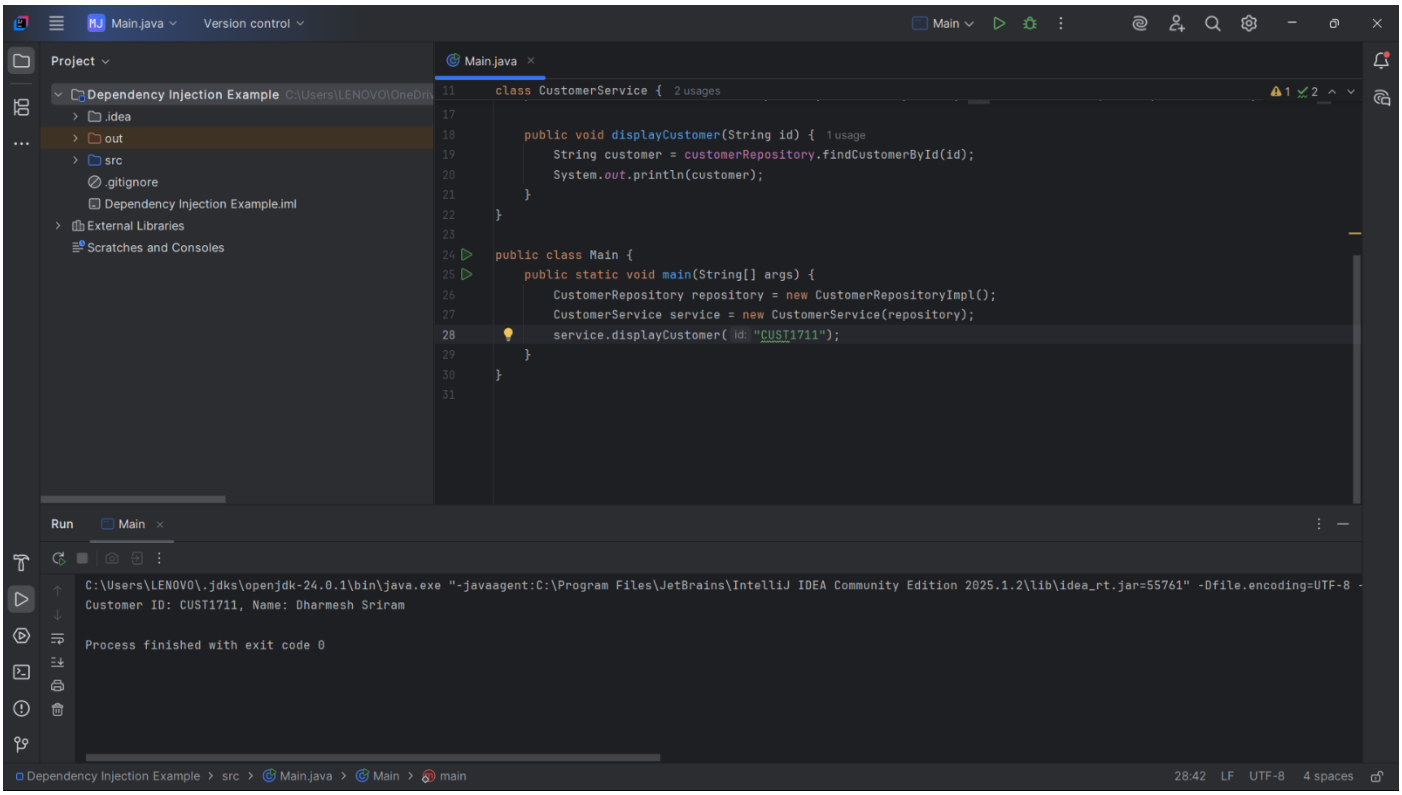
OUTPUT:



Exercise 11: Implementing Dependency Injection

```
interface CustomerRepository {
    String findCustomerById(String id);
}

class CustomerRepositoryImpl implements CustomerRepository {
    public String findCustomerById(String id) {
        return "Customer ID: " + id + ", Name: Dharmesh Sriram";
    }
}

class CustomerService {
    private CustomerRepository customerRepository;

    public CustomerService(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

    public void displayCustomer(String id) {
        String customer = customerRepository.findCustomerById(id);
        System.out.println(customer);
    }
}

public class Main {
    public static void main(String[] args) {
        CustomerRepository repository = new CustomerRepositoryImpl();
        CustomerService service = new CustomerService(repository);
        service.displayCustomer("CUST1711");
    }
}
```

OUTPUT:


Main.java
Version control
Main
Project
Dependency Injection Example C:\Users\LENOVO\OneDriv
.idea
out
src
.gitignore
Dependency Injection Example.iml
External Libraries
Scratches and Consoles
Main.java
class CustomerService { 2 usages
public void displayCustomer(String id) { 1 usage
String customer = customerRepository.findCustomerById(id);
System.out.println(customer);
}
}
public class Main {
public static void main(String[] args) {
CustomerRepository repository = new CustomerRepositoryImpl();
CustomerService service = new CustomerService(repository);
service.displayCustomer(id: "CUST1711");
}
}
Run
Main
C:\Users\LENOVO\.jdks\openjdk-24.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2025.1.2\lib\idea_rt.jar=55761" -Dfile.encoding=UTF-8 -
Customer ID: CUST1711, Name: Dharmesh Sriram
Process finished with exit code 0
Dependency Injection Example › src › Main.java › Main › main
28:42 LF UTF-8 4 spaces