DESIGN PATTERNS

Exercise 1: Implementing the Singleton Pattern.

Main.java:

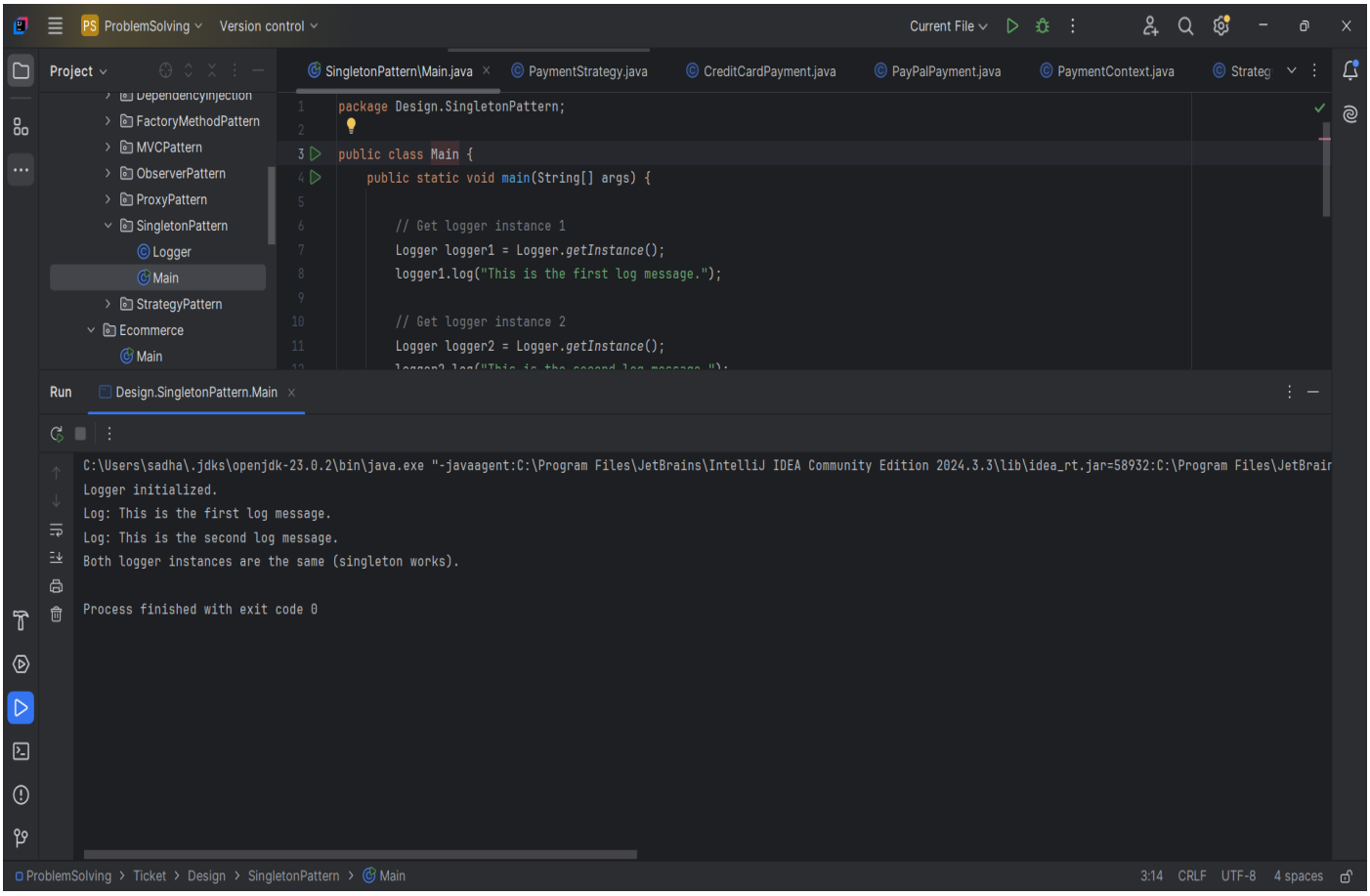
```
package Design.SingletonPattern;
public class Main {
    public static void main(String[] args) {
        Logger logger1 = Logger.getInstance();
        logger1.log("This is the first log message.");

        Logger logger2 = Logger.getInstance();
        logger2.log("This is the second log message.");
        if (logger1 == logger2) {
            System.out.println("Both logger instances are the same (singleton works).");
        } else {
            System.out.println("Logger instances are different (singleton failed).");
        }
    }
}
```

Logger.java:

```
package Design.SingletonPattern;
public class Logger {
    private static Logger instance;
    private Logger() {
        System.out.println("Logger initialized.");
    }
    public static Logger getInstance() {
        if (instance == null) {
```

```
        instance = new Logger(); // only initialized once
    }
    return instance;
  }
  public void log(String message) {
    System.out.println("Log: " + message);
  }
}
```

OUTPUT:



Exercise 2: Implementing the Factory Method Pattern

document.java:

```
package Design.FactoryMethodPattern;

public interface Document {

   void open();

}
```

Documentfactory.java:

```
package Design.FactoryMethodPattern;

public abstract class DocumentFactory {

 public abstract Document createDocument();

}
```

Exceldocument.java:

```
package Design.FactoryMethodPattern;

public class ExcelDocument implements Document {

   @Override

   public void open() {

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

   }

}
```

Exceldocument factory.java:

```
package Design.FactoryMethodPattern;

public class ExcelDocumentFactory extends DocumentFactory {

   @Override

   public Document createDocument() {

      return new ExcelDocument();

   }

}
```

Main.java:

```
package Design.FactoryMethodPattern;
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();
    }
}
```

Pdfdocument.java:

```
package Design.FactoryMethodPattern;
public class PDFDocument implements Document {
    @Override
    public void open() {
        System.out.println("Opening a PDF document.");
    }
}
```

Pdfdocumentfactory.java:

```
package Design.FactoryMethodPattern;

public class PdfDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new PDFDocument();

    }

}
```

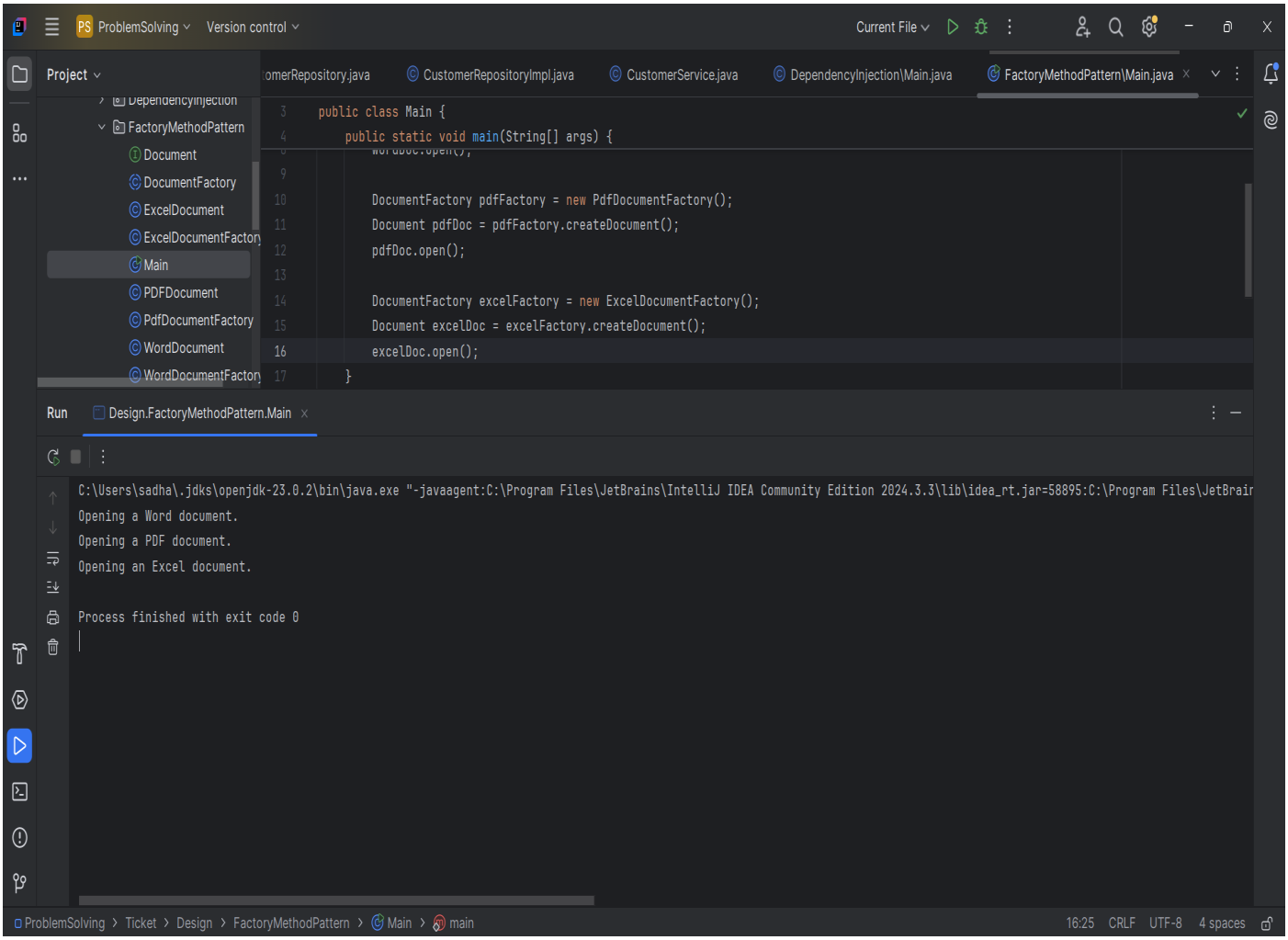
Worddocument.java:

```
package Design.FactoryMethodPattern;

public class WordDocument implements Document {

    @Override

    public void open() {

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

    }

}
```

Worddocumentfactory.java:

```
package Design.FactoryMethodPattern;

public class WordDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new WordDocument();

    }

}
```

OUTPUT:



Exercise 3: Implementing the Builder Pattern

Main.java:

```
package Design.BuilderPattern;

public class Main {

    public static void main(String[] args) {

        Computer basicPC = new Computer.Builder()

                .setCPU("Intel i3")

                .setRAM("8GB")

                .setStorage("256GB SSD")

                .build();
```

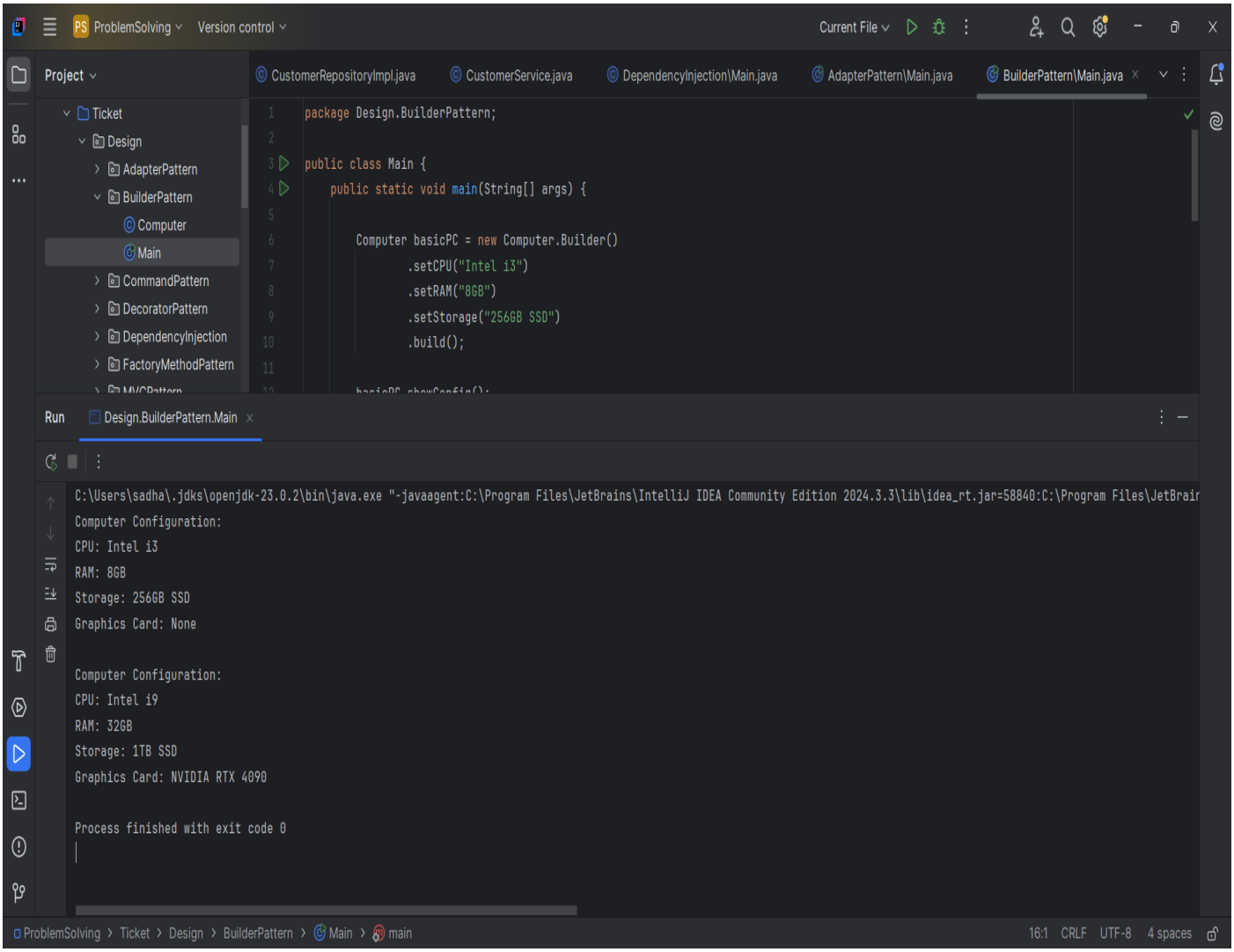
```
        basicPC.showConfig();
        System.out.println();

        Computer gamingPC = new Computer.Builder()
                .setCPU("Intel i9")
                .setRAM("32GB")
                .setStorage("1TB SSD")
                .setGraphicsCard("NVIDIA RTX 4090")
                .build();
        gamingPC.showConfig();
    }
}
```

Computer.java:

```
package Design.BuilderPattern;
public 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 != null ? graphicsCard : "None"));
    }

    public 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);
        }
    }
}
```

Output:



Exercise 4: Implementing the Adapter Pattern

Main.java:

```
package Design.AdapterPattern;

public class Main {

    public static void main(String[] args) {

        // Use PayPal through adapter

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

        paypalProcessor.processPayment(150.00);
```

```
        System.out.println();
        PaymentProcessor stripeProcessor = new StripeAdapter(new StripeSDK());
        stripeProcessor.processPayment(99.99);
    }
}
```

Paymentprocessor.java:

```
package Design.AdapterPattern;
public interface PaymentProcessor {
    void processPayment(double amount);
}
```

Paypaladapter.java:

```
package Design.AdapterPattern;

public class PayPalAdapter implements PaymentProcessor {
    private PayPalSDK payPalSDK;

    public PayPalAdapter(PayPalSDK payPalSDK) {
        this.payPalSDK = payPalSDK;
    }

    @Override
    public void processPayment(double amount) {
        payPalSDK.sendPayment(amount); // adapt to PayPalSDK
    }
}
```

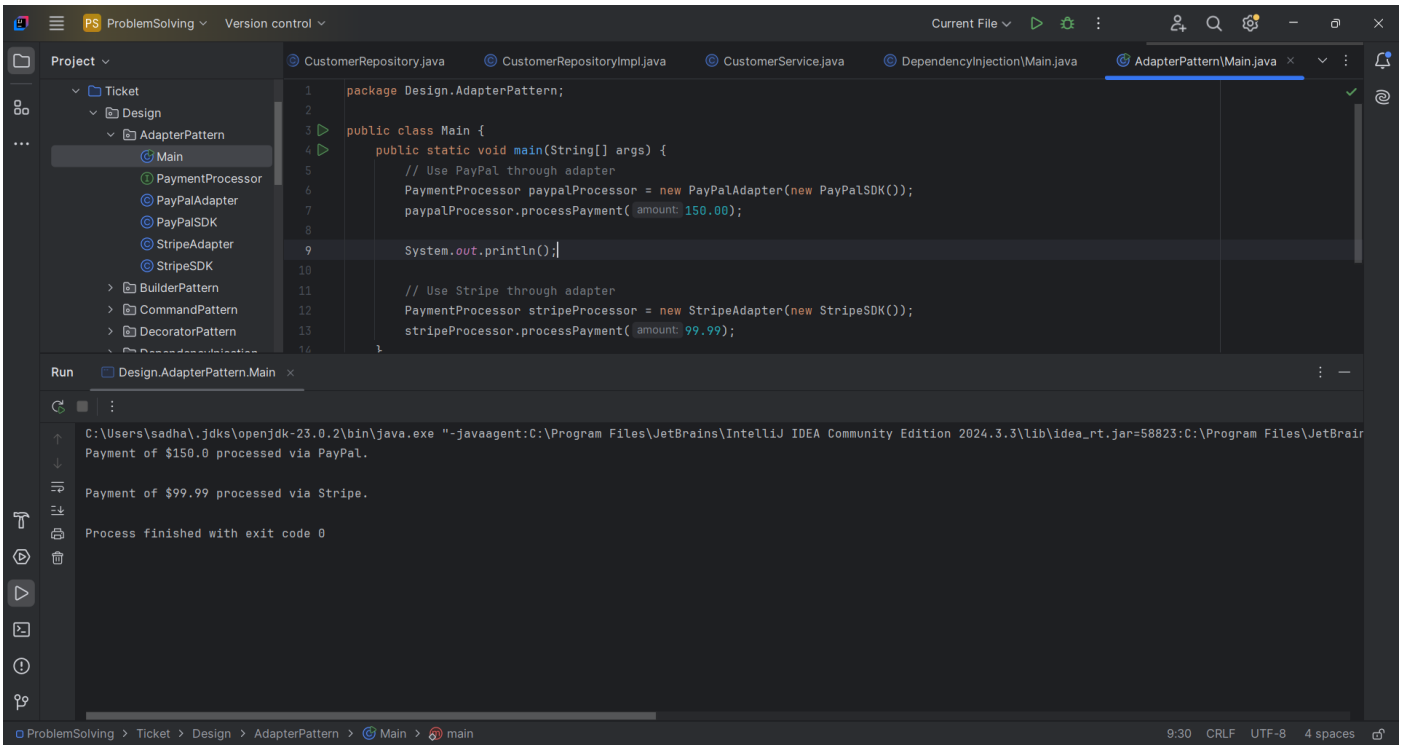
Stripe adapter.java:

```
package Design.AdapterPattern;

public class StripeAdapter implements PaymentProcessor {

    private StripeSDK stripeSDK;


    public StripeAdapter(StripeSDK stripeSDK) {

        this.stripeSDK = stripeSDK;

    }

    @Override

    public void processPayment(double amount) {

        stripeSDK.makePayment(amount * 100);

    }

}
```

Stripesdk.java:

```
package Design.AdapterPattern;

public class StripeSDK {

    public void makePayment(double amountInCents) {

        System.out.println("Payment of $" + (amountInCents / 100) + " processed via Stripe.");

    }

}
```

OUTPUT:



Exercise 5: Implementing the Decorator Pattern

Emailnotifier.java:

```
package Design.DecoratorPattern;

public class EmailNotifier implements Notifier {

    @Override

    public void send(String message) {

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

    }

}
```

main.java:

```
package Design.DecoratorPattern;

public class Main {

    public static void main(String[] args) {
```

```
        Notifier email = new EmailNotifier();
        Notifier emailWithSMS = new SMSNotifierDecorator(email);

        Notifier emailWithSMSAndSlack = new SlackNotifierDecorator(emailWithSMS);
        emailWithSMSAndSlack.send("Server is down!");
    }
}
```

Notifier.java:

```
package Design.DecoratorPattern;
public interface Notifier {
    void send(String message);
}
```

Notifierdecorator.java:

```
package Design.DecoratorPattern;
public abstract class NotifierDecorator implements Notifier {
    protected Notifier wrappedNotifier;

    public NotifierDecorator(Notifier notifier) {
        this.wrappedNotifier = notifier;
    }
    @Override
    public void send(String message) {
        wrappedNotifier.send(message);
    }
}
```

Slacknotifier decorator.java:

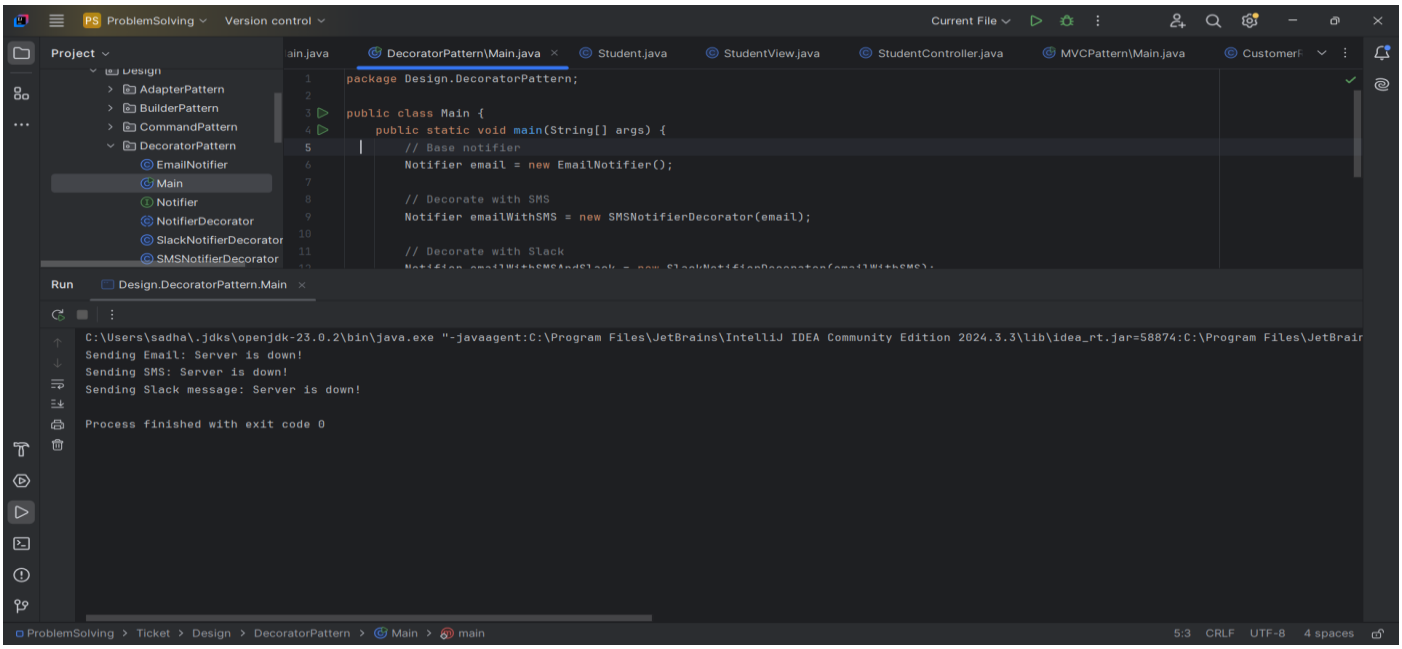
```
package Design.DecoratorPattern;
public class SlackNotifierDecorator extends NotifierDecorator {
    public SlackNotifierDecorator(Notifier notifier) {
        super(notifier);
    }
```

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

Smsnotifier decorator.java:

```
package Design.DecoratorPattern;
public class SMSNotifierDecorator extends NotifierDecorator {
    public SMSNotifierDecorator(Notifier notifier) {
        super(notifier);
    }
    @Override
    public void send(String message) {
        super.send(message);
        System.out.println("Sending SMS: " + message);
    }
}
```

Output:



Exercise 6: Implementing the Proxy Pattern

Main.java:

```
package Design.ProxyPattern;
public class Main {
    public static void main(String[] args) {
        Image image1 = new ProxyImage("photo1.jpg");
        image1.display();

        System.out.println();
        image1.display();

        System.out.println();
        Image image2 = new ProxyImage("photo2.png");
        image2.display();
    }
}
```

Proxyimage.java:

```
package Design.ProxyPattern;
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);
        } else {
```

```
            System.out.println("Using cached image: " + fileName);
        }
        realImage.display();
    }
}
```

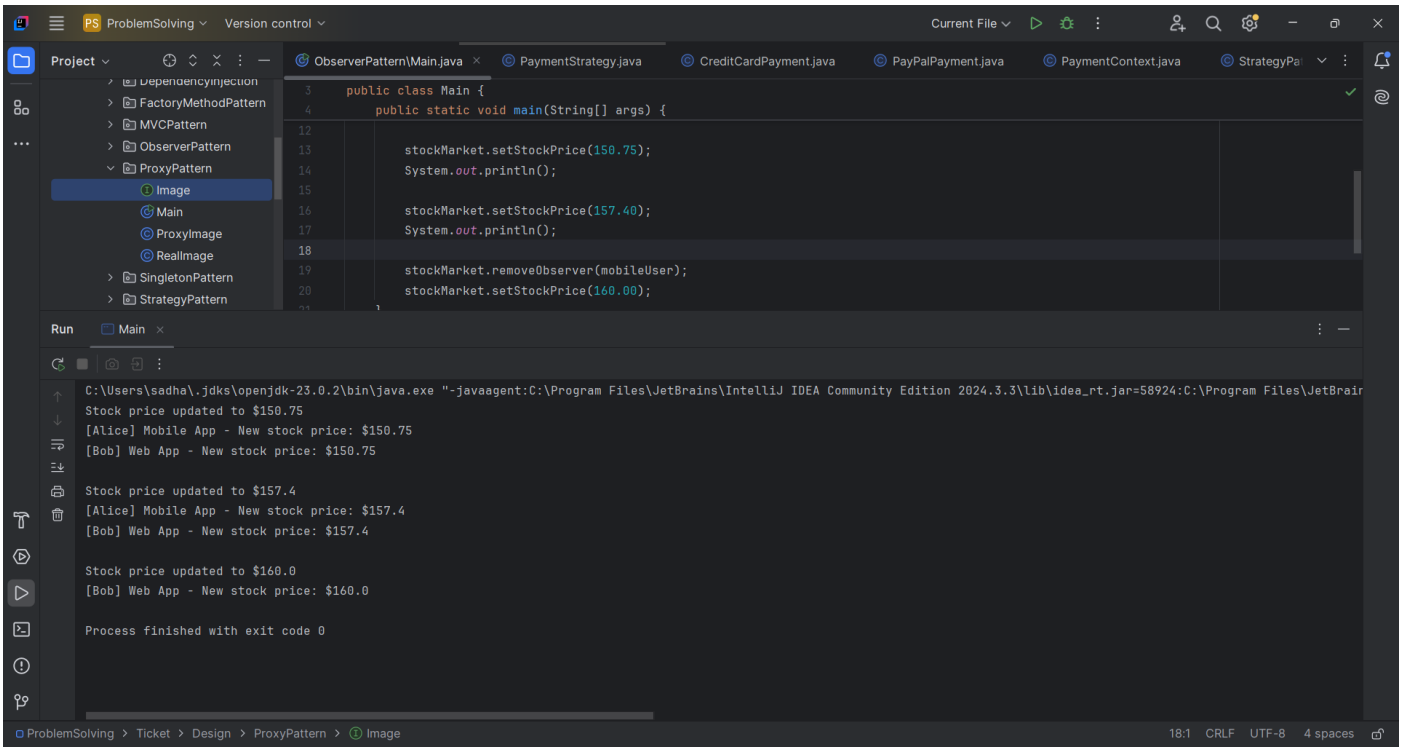
Image.java

```
package Design.ProxyPattern;
public interface Image {
    void display();
}
```

Realimage.java:

```
package Design.ProxyPattern;
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);
    }
    @Override
    public void display() {
        System.out.println("Displaying image: " + fileName);
    }
}
```

OUTPUT:



Exercise 7: Implementing the Observer Pattern

Main.java:

```
ackage Design.ObserverPattern;


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


      Observer mobileUser = new MobileApp("Alice");
      Observer webUser = new WebApp("Bob");


      stockMarket.registerObserver(mobileUser);
      stockMarket.registerObserver(webUser);


      stockMarket.setStockPrice(150.75);
      System.out.println();
```

```
        stockMarket.setStockPrice(157.40);
        System.out.println();

        stockMarket.removeObserver(mobileUser);
        stockMarket.setStockPrice(160.00);
    }
}
```

Mobileapp.java:

```
package Design.ObserverPattern;

public class MobileApp implements Observer {
    private String name;

    public MobileApp(String name) {
        this.name = name;
    }

    @Override
    public void update(double stockPrice) {
        System.out.println("[" + name + "] Mobile App - New stock price: $" + stockPrice);
    }
}
```

Observer.java:

```
package Design.ObserverPattern;

public interface Observer {
    void update(double stockPrice);
}
```

Stock.java:

```
package Design.ObserverPattern;

public interface Stock {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}
```

Stockmarket.java:

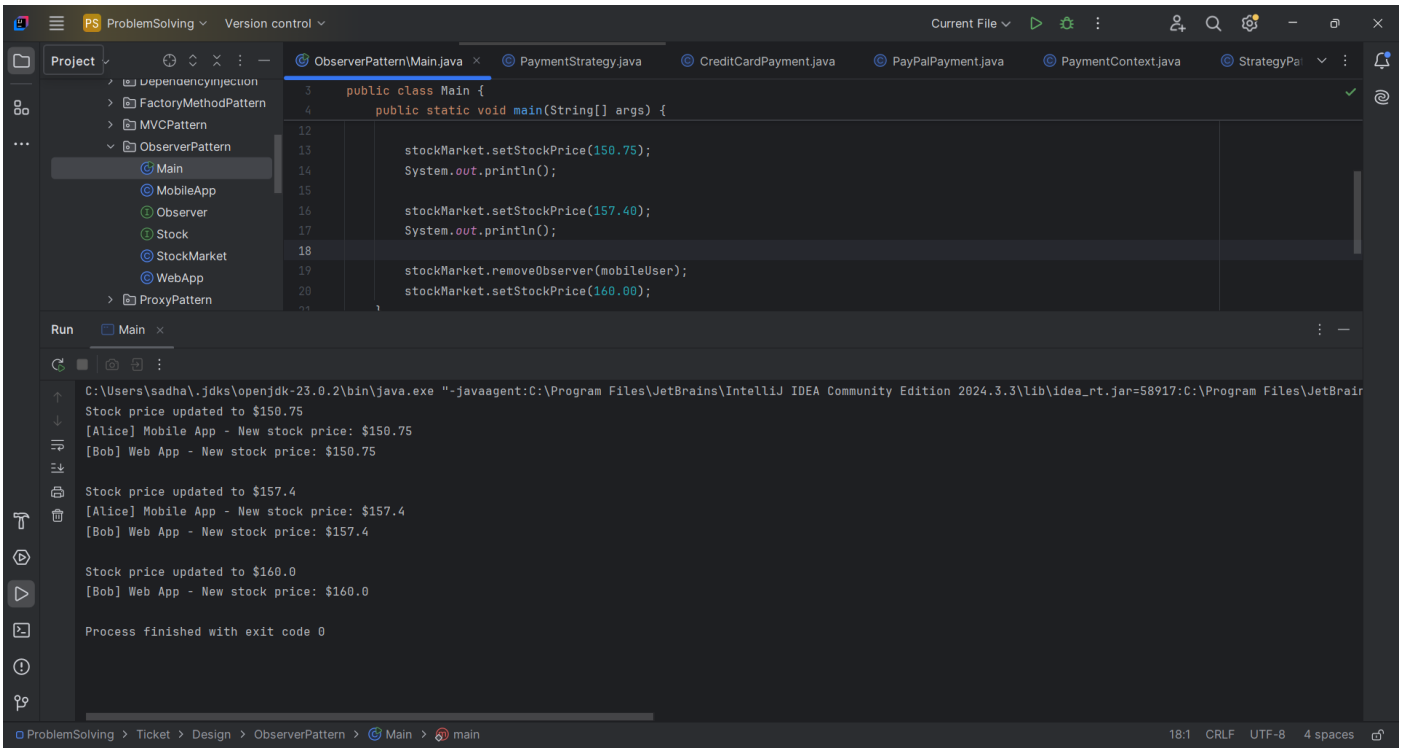
```
package Design.ObserverPattern;

import java.util.ArrayList;
import java.util.List;

public class StockMarket implements Stock {
    private List<Observer> observers = new ArrayList<>();
    private double stockPrice;

    @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();
    }
}
```

Webapp.java:

```
package Design.ObserverPattern;

public class WebApp implements Observer {
    private String name;

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

    @Override
    public void update(double stockPrice) {
        System.out.println("[" + name + "] Web App - New stock price: $" + stockPrice);
    }
}
```

Output:



Exercise 8: Implementing the Strategy Pattern

Main.java:

```
package Design.StrategyPattern;

public class Main {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();


        context.setPaymentStrategy(new CreditCardPayment("1234567812345678"));

        context.payAmount(250.75);


        System.out.println();

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

        context.payAmount(99.99);

    }

}
```

Creditcardpayment.java:

```
package Design.StrategyPattern;


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 ending in " +
                cardNumber.substring(cardNumber.length() - 4));
    }
}
```

Paymentcontext.java:

```
package Design.StrategyPattern;
public class PaymentContext {
    private PaymentStrategy strategy;


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


    public void payAmount(double amount) {
        if (strategy == null) {
            System.out.println("No payment method selected.");
        } else {
            strategy.pay(amount);
        }
```

```
    }
}
```

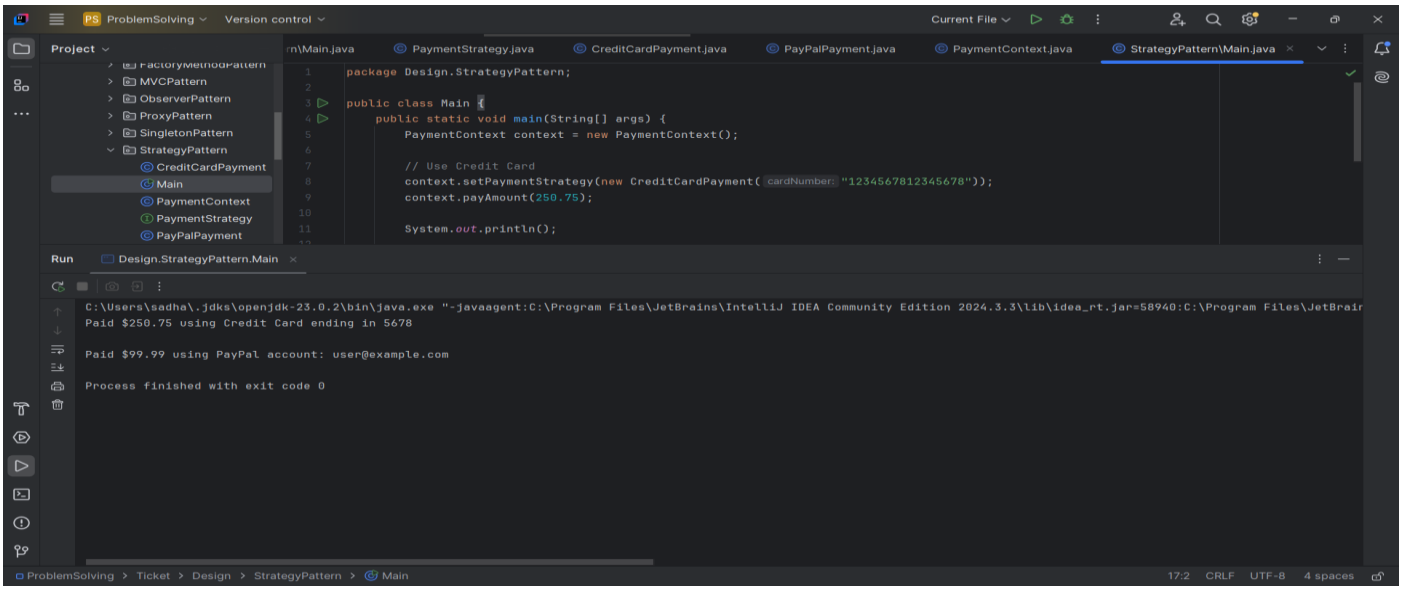
Paymentstrategy.java:

```
package Design.StrategyPattern;
public interface PaymentStrategy {
    void pay(double amount);
}
```

Paypalpayment.java:

```
package Design.StrategyPattern;
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);
    }
}
```

Output:



Exercise 9: Implementing the Command Pattern

Main.java

```
package Design.CommandPattern;

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();


        // Turn ON the light

        remote.setCommand(lightOn);

        remote.pressButton();


        // Turn OFF the light

        remote.setCommand(lightOff);

        remote.pressButton();

    }

}
```

Command.java:

```
package Design.CommandPattern;

public interface Command {

    void execute();

}
```

Light.java:

```
package Design.CommandPattern;


public class Light {

    public void turnOn() {
```

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

    public void turnOff() {
        System.out.println("The light is OFF.");
    }
}
```

LightOffCommand.java:

```
package Design.CommandPattern;
public class LightOffCommand implements Command {
    private Light light;
    public LightOffCommand(Light light) {
        this.light = light;
    }
    @Override
    public void execute() {
        light.turnOff();
    }
}
```

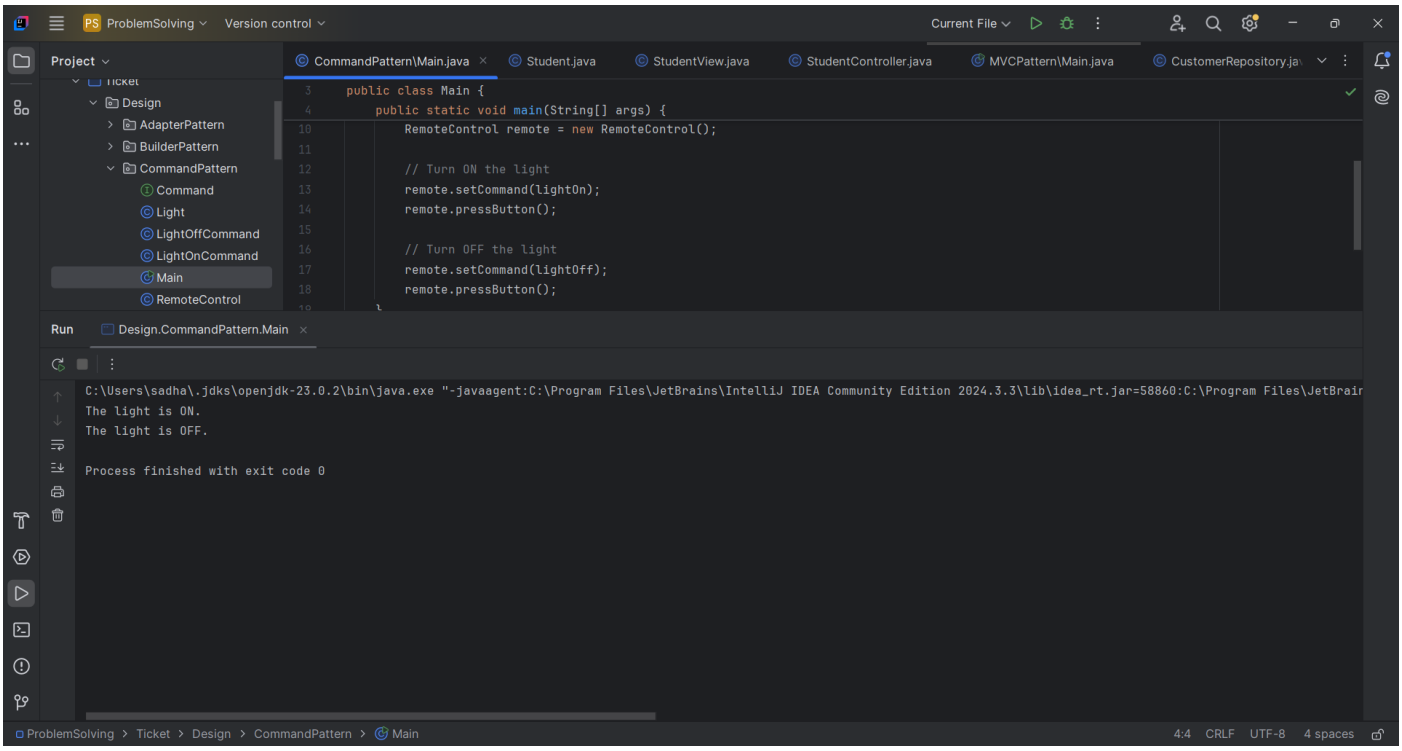
LightOnCommand.java:

```
package Design.CommandPattern;
public class LightOnCommand implements Command {
    private Light light;
    public LightOnCommand(Light light) {
        this.light = light;
    }
    @Override
    public void execute() {
        light.turnOn();
    }
}
```

RemoteControl.java:

```
package Design.CommandPattern;

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 set.");

        }

    }

}
```

Output:



Main.java

```
package Design.MVCPattern;


public class Main {
    public static void main(String[] args) {


        Student student = new Student();
        student.setId("S001");
        student.setName("John Doe");
        student.setGrade("A");


        StudentView view = new StudentView();


        StudentController controller = new StudentController(student, view);


        controller.updateView();


        System.out.println("\nUpdating student data...\n");


        controller.setStudentName("Jane Smith");
        controller.setStudentGrade("B+");


        controller.updateView();
    }
}
```

Student.java:

```
package Design.MVCPattern;
public class Student {
    private String id;
    private String name;
```

```
    private String grade;

    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getGrade() {
        return grade;
    }
    public void setGrade(String grade) {
        this.grade = grade;
    }
}
```

Studentcontroller.java:

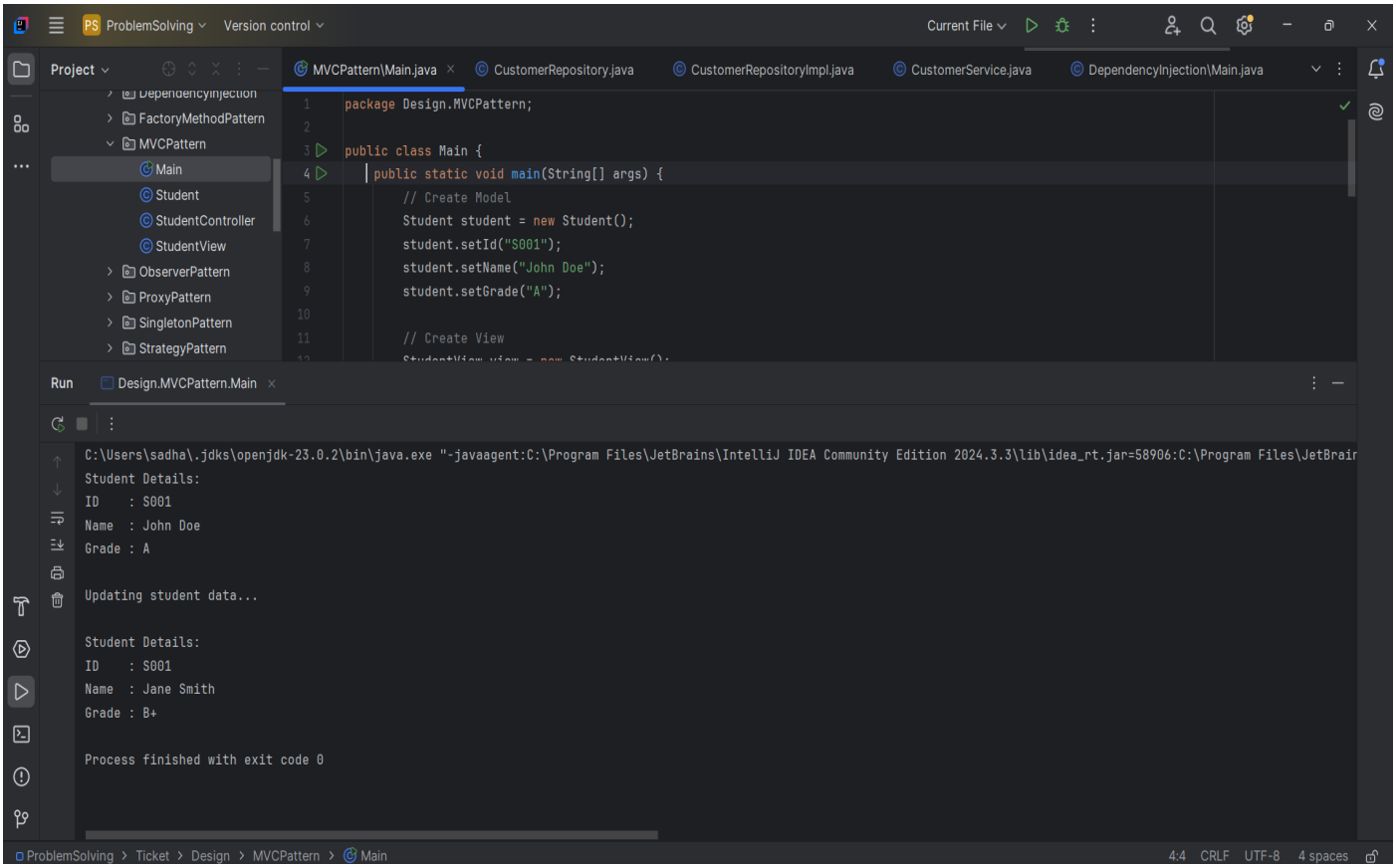
```
package Design.MVCPattern;
public class StudentController {
    private Student model;
    private StudentView view;

    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();
    }
```

Studentview.java:

```
package Design.MVCPattern;
public class StudentView {
    public void displayStudentDetails(String id, String name, String grade) {
        System.out.println("Student Details:");
        System.out.println("ID    : " + id);
        System.out.println("Name  : " + name);
        System.out.println("Grade : " + grade);
    }
}
```

Output:



Exercise 11: Implementing Dependency Injection

Main.java:

package Design.DependencyInjection;

public class Main {

```
    public static void main(String[] args) {

        CustomerRepository repository = new CustomerRepositoryImpl();
        CustomerService service = new CustomerService(repository);
        service.getCustomerDetails("CUST001");
    }
}
```

CustomerService.java

```
package Design.DependencyInjection;
public class CustomerService {
    private CustomerRepository customerRepository;

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

    public void getCustomerDetails(String id) {
        String customer = customerRepository.findCustomerById(id);
        System.out.println("Customer found: " + customer);
    }
}
```

CustomerRepositoryImpl.java:

```
package Design.DependencyInjection;
public class CustomerRepositoryImpl implements CustomerRepository {

    @Override
    public String findCustomerById(String id) {
        return "Customer{id='" + id + "', name='Alice Johnson'}";
    }
}
```

CustomerRepository.java:

```
package Design.DependencyInjection;

public interface CustomerRepository {

    String findCustomerById(String id);

}
```

Output: