

## 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:

The screenshot displays the IntelliJ IDEA IDE interface. The top toolbar shows the 'Run' button (a green play icon). The 'Project' view on the left lists the project structure, including 'SingletonPattern' and its sub-packages. The 'Main' class is selected. The editor window shows the following Java code:

```

package Design.SingletonPattern;

public class Main {

    public static void main(String[] args) {

        // Get logger instance 1
        Logger logger1 = Logger.getInstance();
        logger1.log("This is the first log message.");

        // Get logger instance 2
        Logger logger2 = Logger.getInstance();
        logger2.log("This is the second log message.");
    }
}

```

The 'Run' window at the bottom shows the output of the program:

```

C:\Users\sadha\.jdk\openjdk-23.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\lib\idea_rt.jar=58932:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\bin" -Didea.config.path=C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\conf -Didea.copyright.notification=false -Didea.log=C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\log\idea.log -Didea.platform.prefix=idea -Didea.version=2024.3.3 -jar C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\bin\idea_rt.jar 58932
Logger initialized.
Log: This is the first log message.
Log: This is the second log message.
Both logger instances are the same (singleton works).
Process finished with exit code 0

```

The status bar at the bottom indicates the file encoding is UTF-8 and the line length is 4 spaces.

## 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

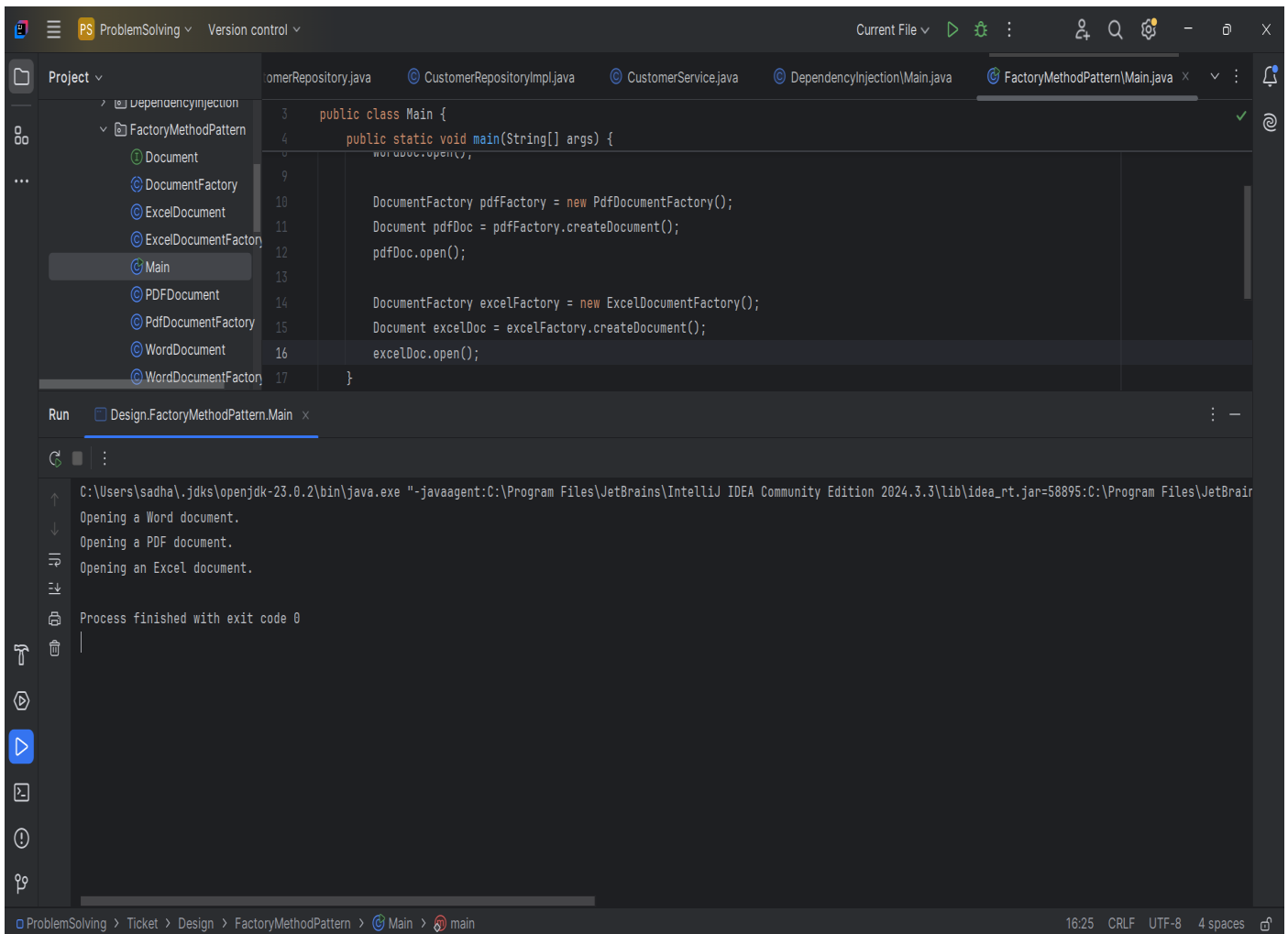
    public Document createDocument() {

        return new WordDocument();

    }

}
```

## OUTPUT:



The screenshot displays the IntelliJ IDEA IDE interface. The top toolbar shows the 'Run' button (a green play icon). The 'Project' view on the left shows the package structure: `DependencyInjection` > `FactoryMethodPattern`, which contains `Document`, `DocumentFactory`, `ExcelDocument`, `ExcelDocumentFactory`, `Main` (selected), `PDFDocument`, `PdfDocumentFactory`, `WordDocument`, and `WordDocumentFactory`. The editor shows the `Main.java` file with the following code:

```
3 public class Main {
4     public static void main(String[] args) {
5         wordDoc.open();
6
7         DocumentFactory pdfFactory = new PdfDocumentFactory();
8         Document pdfDoc = pdfFactory.createDocument();
9         pdfDoc.open();
10
11         DocumentFactory excelFactory = new ExcelDocumentFactory();
12         Document excelDoc = excelFactory.createDocument();
13         excelDoc.open();
14     }
15 }
```

The 'Run' window at the bottom shows the execution output for `Design.FactoryMethodPattern.Main`. The command executed is `C:\Users\sadha\.jdk\openjdk-23.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\lib\idea_rt.jar=58895:C:\Program Files\JetBrains\JetBrain`. The output shows the sequence of operations: 'Opening a Word document.', 'Opening a PDF document.', and 'Opening an Excel document.', followed by 'Process finished with exit code 0'.

## 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);

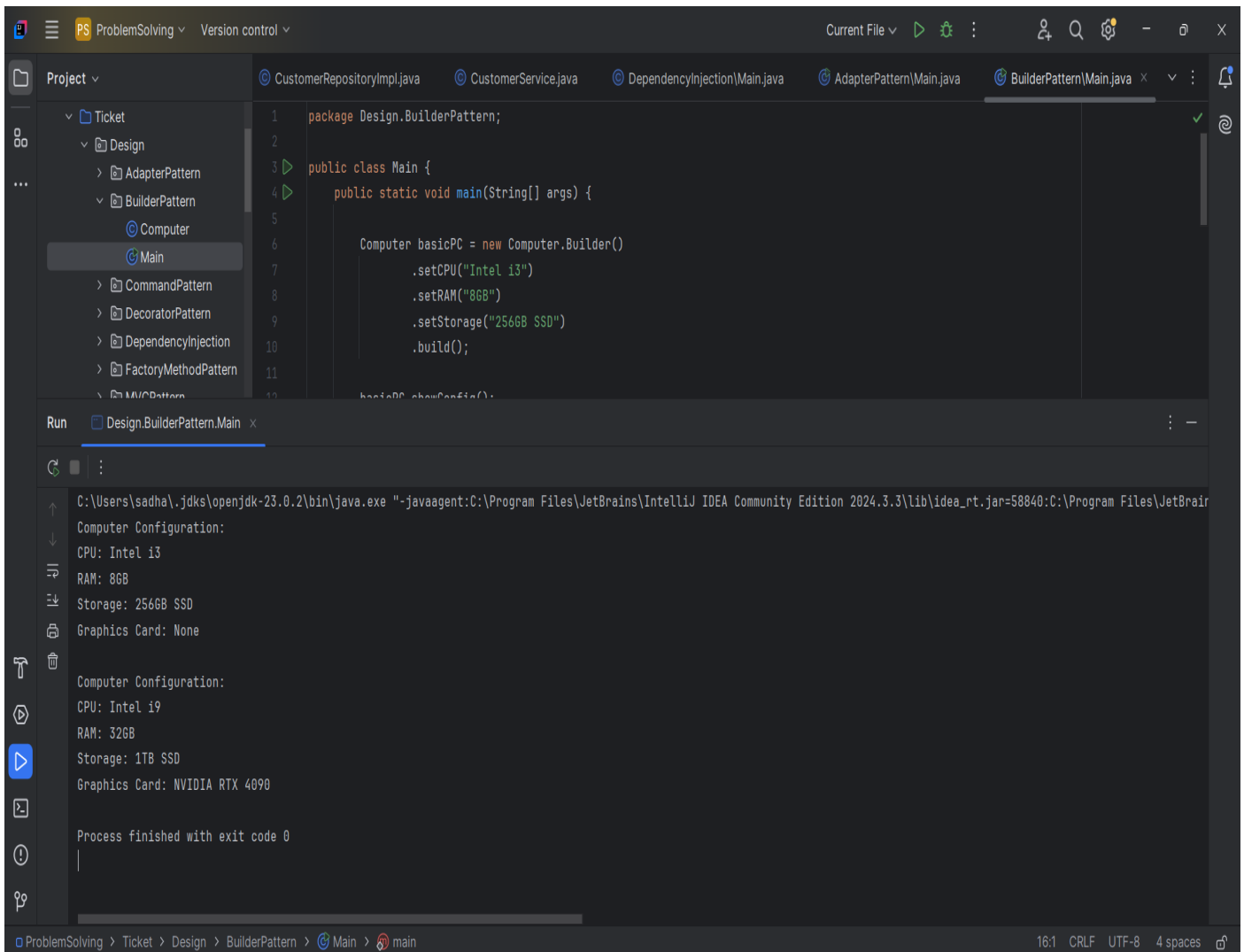
    }

}

}
```



## Output:



The screenshot shows the IntelliJ IDEA IDE with a project named 'ProblemSolving'. The 'Project' view on the left shows a package structure: 'Ticket' > 'Design' > 'BuilderPattern' > 'Main'. The 'Main.java' file is open in the editor, showing the following code:

```
1 package Design.BuilderPattern;
2
3 public class Main {
4     public static void main(String[] args) {
5
6         Computer basicPC = new Computer.Builder()
7             .setCPU("Intel i3")
8             .setRAM("8GB")
9             .setStorage("256GB SSD")
10            .build();
11
12        basicPC.showConfig();
13    }
14 }
```

The 'Run' view at the bottom shows the output of the program:

```
C:\Users\sadha\.jdk\openjdk-23.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\lib\idea_rt.jar=58840:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\bin" -Didea.config.path=C:\Users\sadha\.idea -Didea.home.path=C:\Users\sadha\.idea -Didea.platform.prefix=IntelliJ -jar C:\Users\sadha\.idea\workspace\ProblemSolving\out\production\ProblemSolving\Main.class
Computer Configuration:
CPU: Intel i3
RAM: 8GB
Storage: 256GB SSD
Graphics Card: None

Computer Configuration:
CPU: Intel i9
RAM: 32GB
Storage: 1TB SSD
Graphics Card: NVIDIA RTX 4090

Process finished with exit code 0
```

The status bar at the bottom indicates the file encoding is UTF-8 and the line length is 16:1.

## 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 {

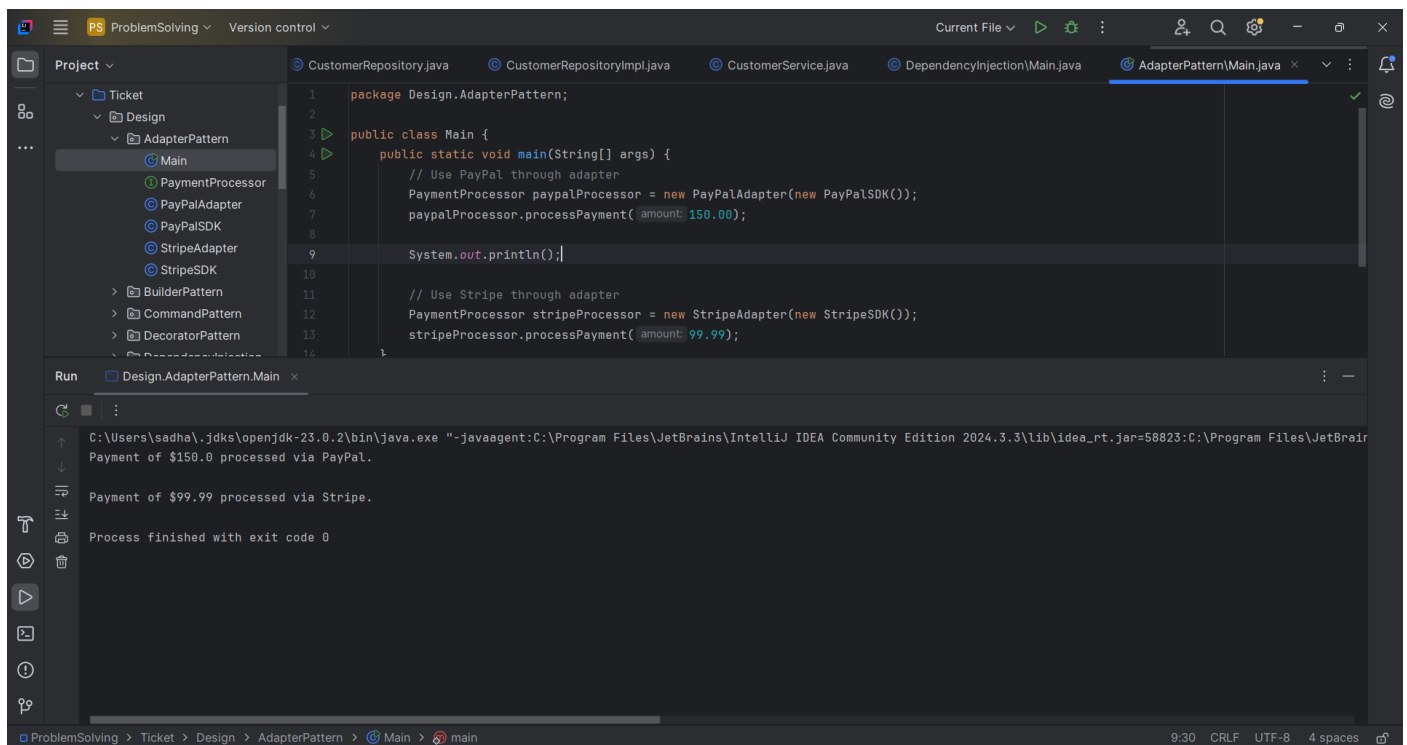
    public void makePayment(double amountInCents) {

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

    }

}
```

## OUTPUT:



The screenshot shows the IntelliJ IDEA IDE with a project named 'ProblemSolving'. The 'Project' view on the left shows a package structure: 'Ticket' > 'Design' > 'AdapterPattern'. Inside 'AdapterPattern', there is a 'Main' class and several interfaces and adapters: 'PaymentProcessor', 'PayPalAdapter', 'PayPalSDK', 'StripeAdapter', and 'StripeSDK'. The 'Main' class is selected, and its code is visible in the editor. The code implements the AdapterPattern by creating 'PayPalAdapter' and 'StripeAdapter' to implement the 'PaymentProcessor' interface. The 'main' method demonstrates the usage by creating a 'paypalProcessor' and a 'stripeProcessor' and calling 'processPayment' with amounts 150.00 and 99.99 respectively. The 'Run' window at the bottom shows the output: 'Payment of \$150.0 processed via PayPal.' and 'Payment of \$99.99 processed via Stripe.', followed by 'Process finished with exit code 0'.

```
1 package Design.AdapterPattern;
2
3 public class Main {
4     public static void main(String[] args) {
5         // Use PayPal through adapter
6         PaymentProcessor paypalProcessor = new PayPalAdapter(new PayPalSDK());
7         paypalProcessor.processPayment( amount: 150.00);
8
9         System.out.println();
10
11        // Use Stripe through adapter
12        PaymentProcessor stripeProcessor = new StripeAdapter(new StripeSDK());
13        stripeProcessor.processPayment( amount: 99.99);
14    }
15 }
```

Run: Design.AdapterPattern.Main

C:\Users\sadha\jdk\openjdk-23.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\lib\idea\_rt.jar=58823:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\bin" -Didea.config.path=C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\conf -Didea.copyright.path=C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\copyright -Didea.home.path=C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\bin -Didea.platform.prefix=IntelliJ -Didea.vendor.name=JetBrains -Djava.awt.headless=true -Djava.class.path=C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\bin\idea\_rt.jar -Djava.library.path=C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\bin\lib\jbr -Didea.config.path=C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\conf -Didea.copyright.path=C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\copyright -Didea.home.path=C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\bin -Didea.platform.prefix=IntelliJ -Didea.vendor.name=JetBrains -Djava.awt.headless=true -Djava.class.path=C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\bin\idea\_rt.jar -Djava.library.path=C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\bin\lib\jbr

Payment of \$150.0 processed via PayPal.

Payment of \$99.99 processed via Stripe.

Process finished with exit code 0

## 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);

    }

}

```

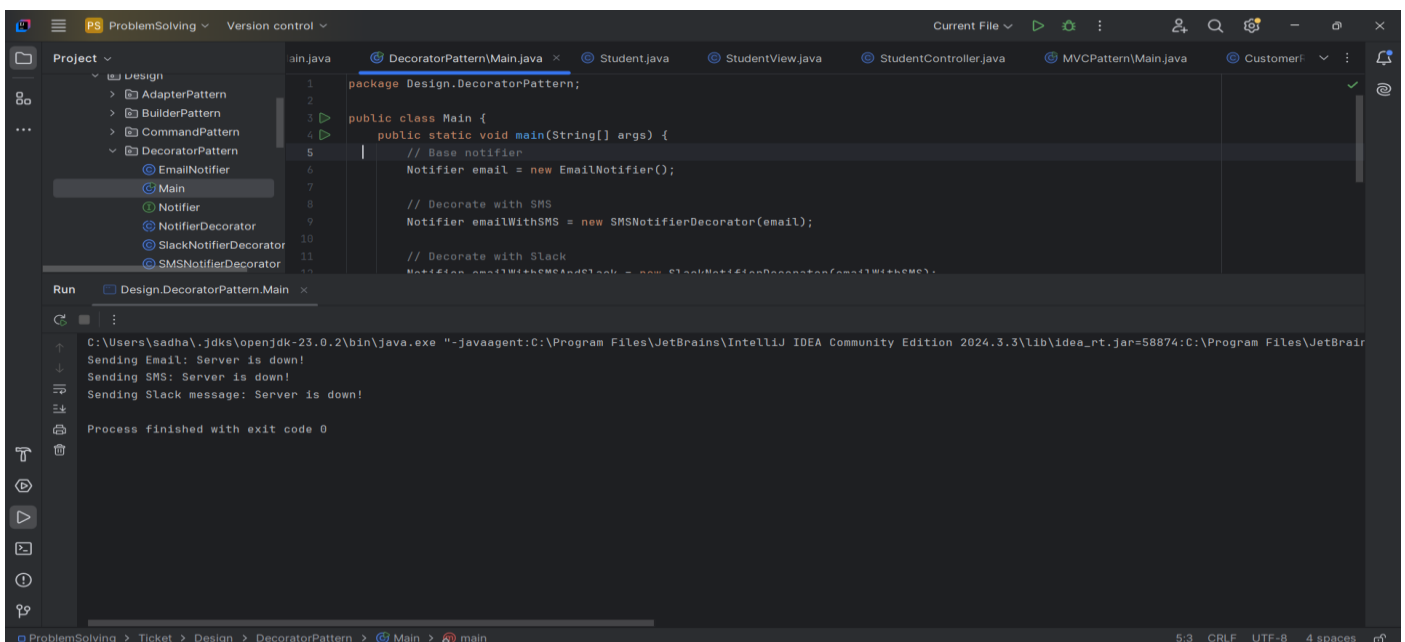
@Override

```
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:



The screenshot shows the IntelliJ IDEA IDE with the following details:

- Project View:** A tree on the left shows the project structure under 'Design' > 'DecoratorPattern'. It includes files like EmailNotifier, Main, Notifier, NotifierDecorator, SlackNotifierDecorator, and SMSNotifierDecorator.
- Code Editor:** The 'Main.java' file is open, showing a 'Main' class with a 'main' method. The code creates an 'EmailNotifier', decorates it with 'SMSNotifierDecorator', and then with 'SlackNotifierDecorator' before calling 'send'.
- Run Console:** The output at the bottom shows the execution results:  
C:\Users\sadha\.jdk\openjdk-23.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\lib\idea\_rt.jar=58874:C:\Program Files\JetBrains\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\bin" -Dfile.encoding=UTF-8  
Sending Email: Server is down!  
Sending SMS: Server is down!  
Sending Slack message: Server is down!  
Process finished with exit code 0

## 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 ReallImage reallImage;

    private String fileName;

    public ProxyImage(String fileName) {

        this.fileName = fileName;

    }

    @Override

    public void display() {

        if (reallImage == null) {

            reallImage = new ReallImage(fileName);

        } else {
```

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

### **Image.java**

```
package Design.ProxyPattern;

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

### **Realimage.java:**

```
package Design.ProxyPattern;

public class ReallImage implements Image {
    private String fileName;

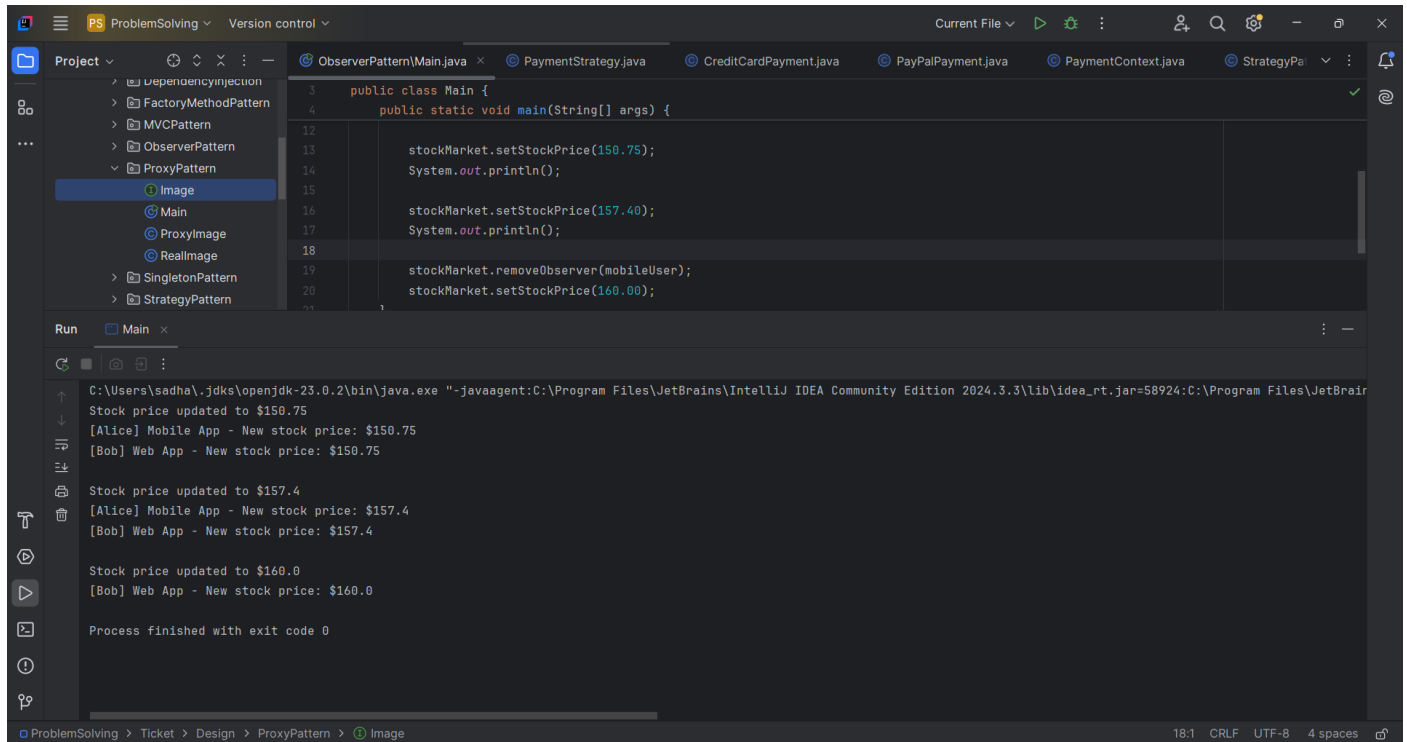
    public ReallImage(String fileName) {
        this.fileName = fileName;
        loadFromRemoteServer();
    }

    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:

```
package 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;
```

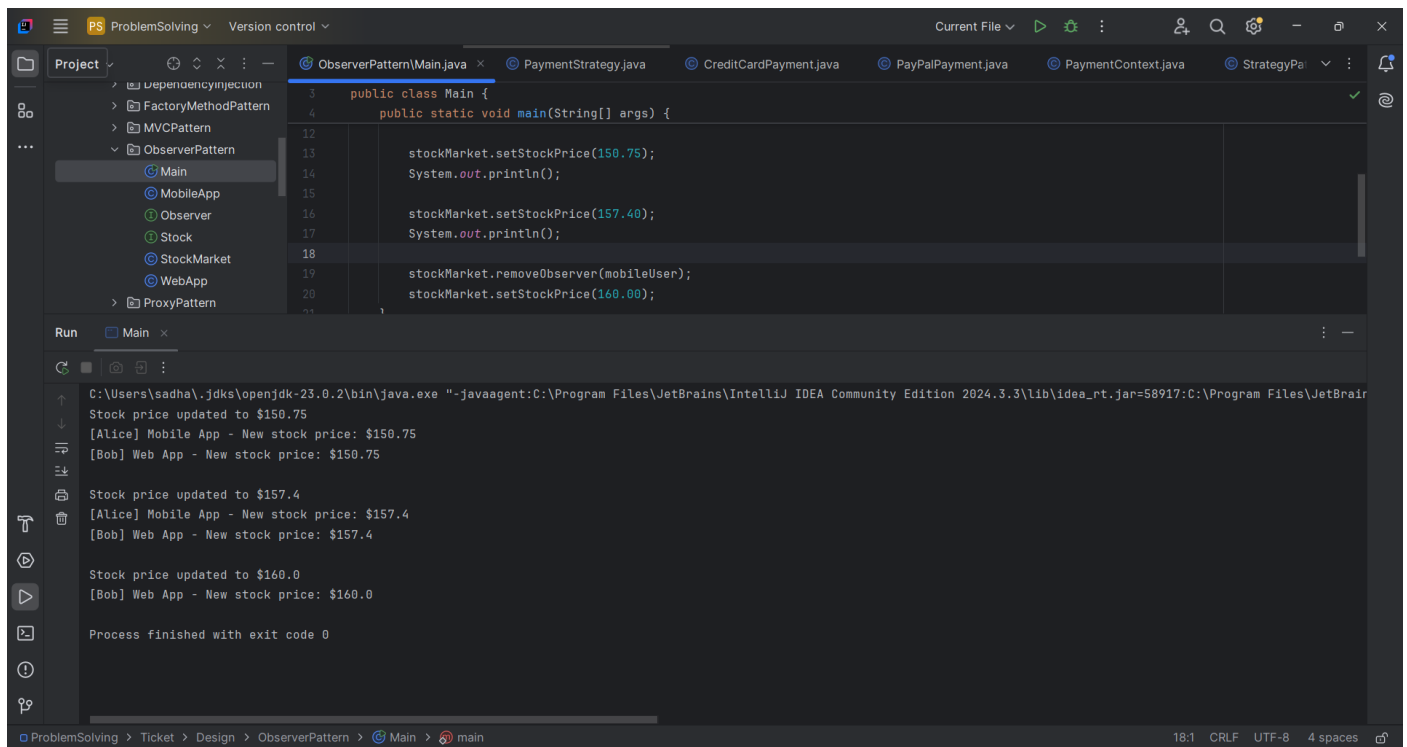
```
public class WebApp implements Observer {  
    private String name;
```

```
    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) {

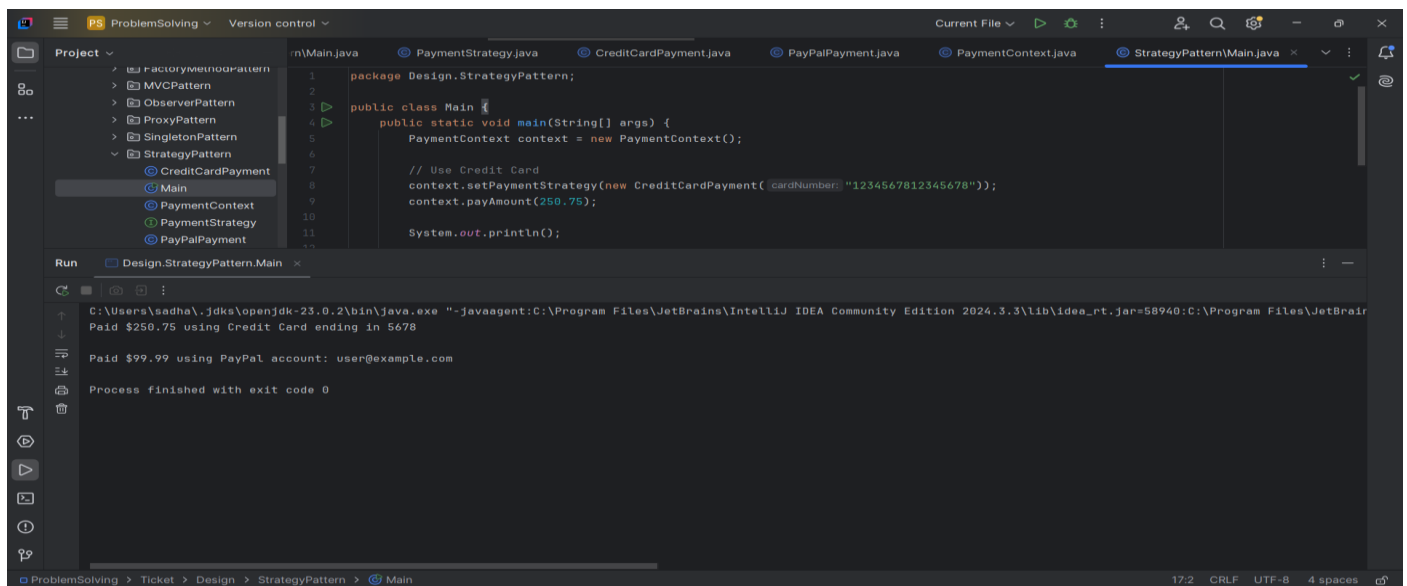
        this.strategy = strategy;
    }

    public void payAmount(double amount) {

        if (strategy == null) {

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

            strategy.pay(amount);
        }
    }
}
```



## 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 {

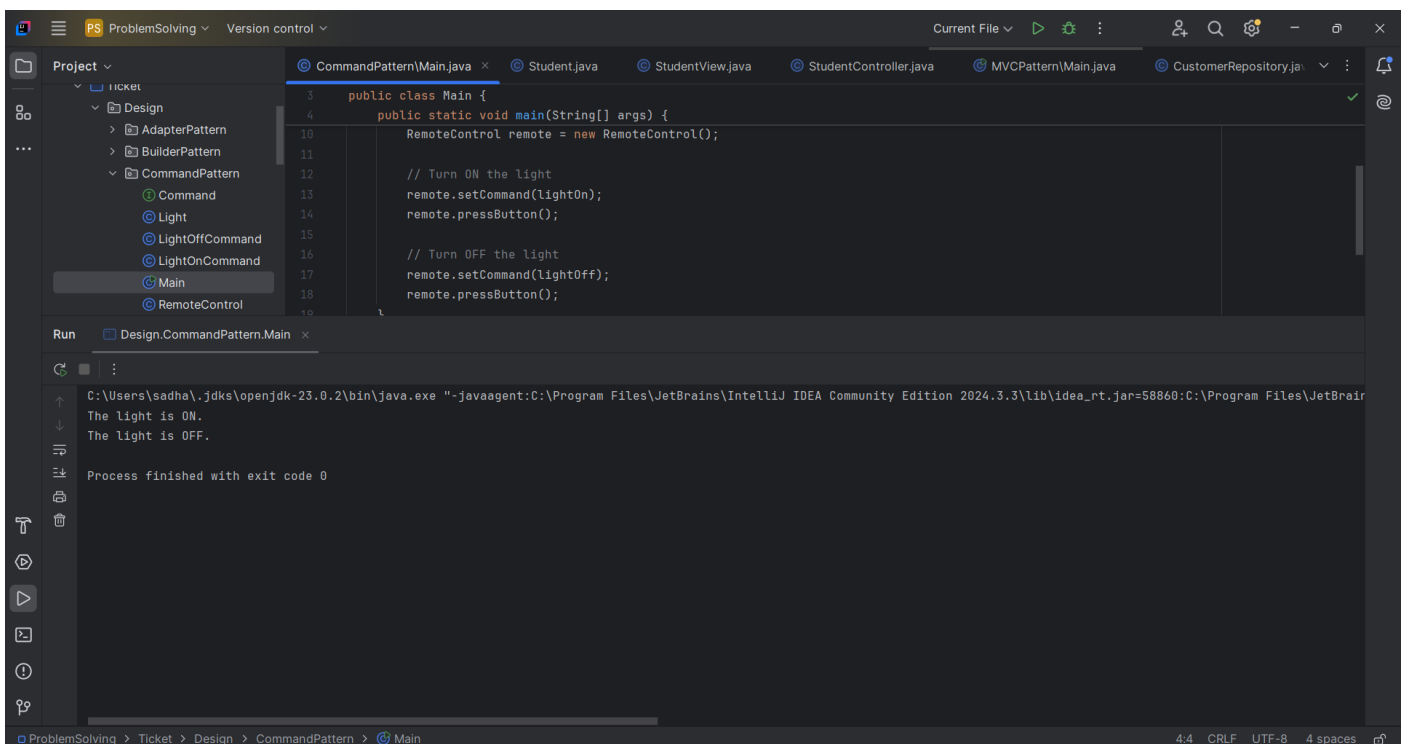
            System.out.println("No command set.");

        }

    }

}
```

## Output:



The screenshot displays the IntelliJ IDEA interface. The Project view on the left shows the project structure: Ticket > Design > CommandPattern > Main. The Main.java file is open in the editor, showing the following code:

```
3 public class Main {
4     public static void main(String[] args) {
5         RemoteControl remote = new RemoteControl();
6
7         // Turn ON the light
8         remote.setCommand(lightOn);
9         remote.pressButton();
10
11         // Turn OFF the light
12         remote.setCommand(lightOff);
13         remote.pressButton();
14     }
15 }
```

The Run window at the bottom shows the output of the program:

```
C:\Users\sadha\jdk\openjdk-23.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\lib\idea_rt.jar=58860:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\bin" -Didea.config.path=C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\conf -Didea.system.path=C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\lib\idea_rt.jar -Didea.version=2024.3.3 -jar C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\bin\idea_rt.jar 58860
The light is ON.
The light is OFF.
Process finished with exit code 0
```

The status bar at the bottom indicates the file is Main.java, with encoding 4:4, CRLF, UTF-8, and 4 spaces.

### **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);

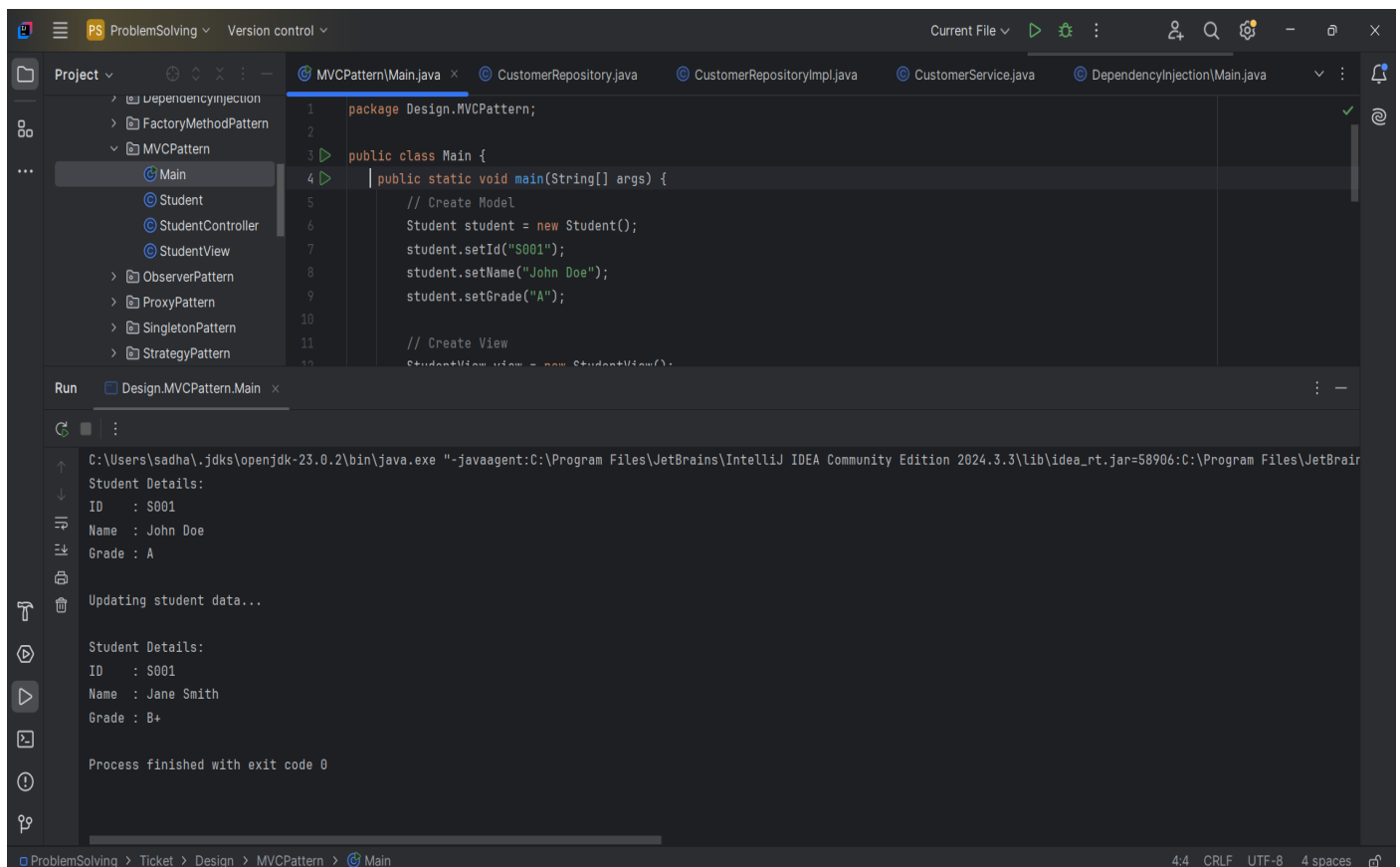
        System.out.println("Name : " + name);

        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:**

