

Digital Nurture 4.0  
Deep Skilling Handbook  
Java FSE -Solution

# DESIGN PATTERNS

## Exercise 1: Implementing the Singleton Pattern:

### Program:

```
package DesignPatterns;

public class SingletonPatternExample {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        logger1.log("First log message");

        Logger logger2 = Logger.getInstance();

        logger2.log("Second log message");

        if (logger1 == logger2) {

            System.out.println("Both logger instances are the same (Singleton verified)");

        } else {

            System.out.println("Different instances (Singleton violated)");

        }

    }

}

class Logger {

    private static final Logger instance = new Logger();

    private Logger() {

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

    }

    public static Logger getInstance() {

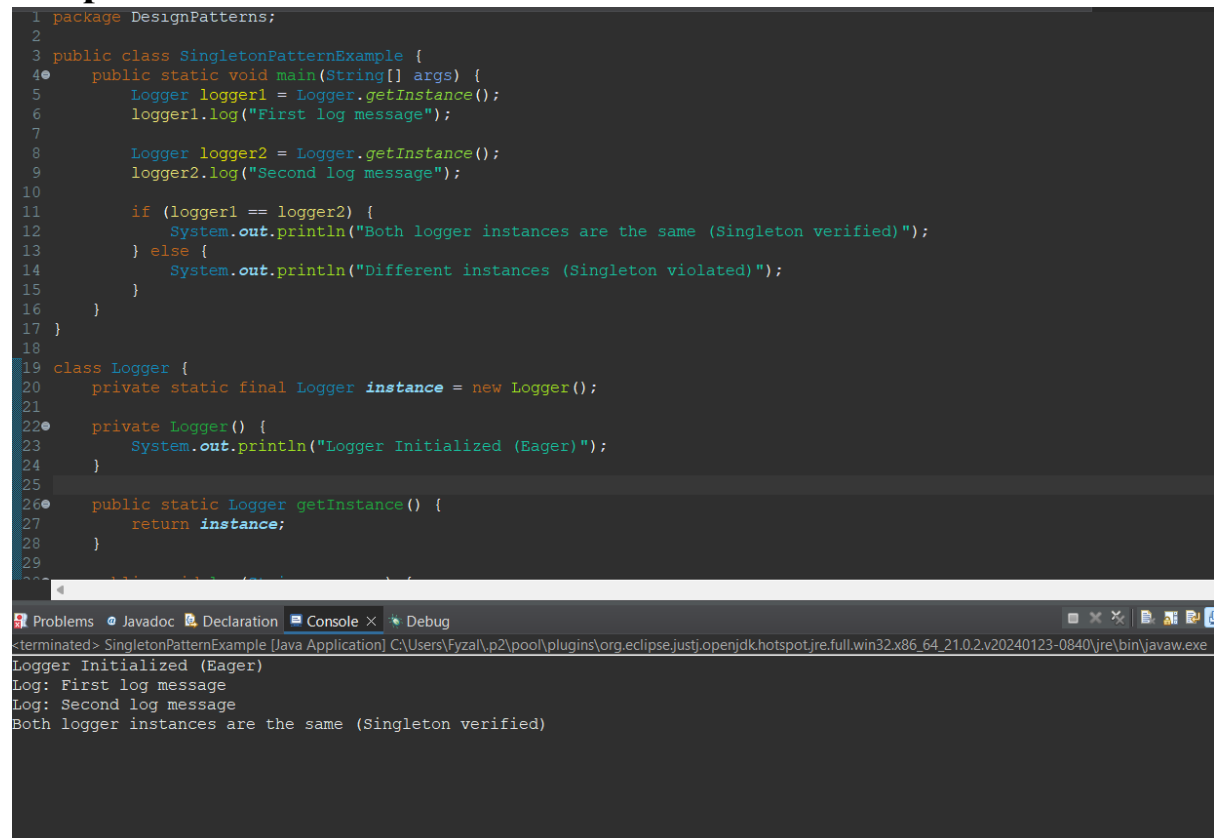
        return instance;

    }

}
```

```
public void log(String message) {  
    System.out.println("Log: " + message);  
}  
}
```

## Output:



The screenshot shows an IDE with a Java code editor and a console window. The code defines a Singleton pattern for a Logger class. The main method creates two logger instances and checks if they are the same. The console output shows the logger being initialized and the two instances being identical, confirming the Singleton pattern.

```
1 package DesignPatterns;  
2  
3 public class SingletonPatternExample {  
4     public static void main(String[] args) {  
5         Logger logger1 = Logger.getInstance();  
6         logger1.log("First log message");  
7  
8         Logger logger2 = Logger.getInstance();  
9         logger2.log("Second log message");  
10  
11         if (logger1 == logger2) {  
12             System.out.println("Both logger instances are the same (Singleton verified)");  
13         } else {  
14             System.out.println("Different instances (Singleton violated)");  
15         }  
16     }  
17 }  
18  
19 class Logger {  
20     private static final Logger instance = new Logger();  
21  
22     private Logger() {  
23         System.out.println("Logger Initialized (Eager)");  
24     }  
25  
26     public static Logger getInstance() {  
27         return instance;  
28     }  
29  
30     public void log(String message) {  
31         System.out.println("Log: " + message);  
32     }  
33 }
```

Problems Javadoc Declaration Console × Debug  
<terminated> SingletonPatternExample [Java Application] C:\Users\Fyza\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86\_64\_21.0.2.v20240123-0840\jre\bin\javaw.exe  
Logger Initialized (Eager)  
Log: First log message  
Log: Second log message  
Both logger instances are the same (Singleton verified)

## Exercise 2: Implementing the Factory Method Pattern

### PROGRAM:

```
package DesignPatterns;

public class FactoryMethodPatternExample {

    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();
    }
}

interface Document {

    void open();
}

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");  
    }  
}
```

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

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

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

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

## OUTPUT:

```
1 package DesignPatterns;  
2  
3 public class FactoryMethodPatternExample {  
4     public static void main(String[] args) {  
5         DocumentFactory wordFactory = new WordDocumentFactory();  
6         Document wordDoc = wordFactory.createDocument();  
7         wordDoc.open();  
8  
9         DocumentFactory pdfFactory = new PdfDocumentFactory();  
10        Document pdfDoc = pdfFactory.createDocument();  
11        pdfDoc.open();  
12  
13        DocumentFactory excelFactory = new ExcelDocumentFactory();  
14        Document excelDoc = excelFactory.createDocument();  
15        excelDoc.open();  
16    }  
17 }  
18  
19 interface Document {  
20     void open();  
21 }  
22  
23 class WordDocument implements Document {  
24     public void open() {  
25         System.out.println("Opening Word Document");  
26     }  
27 }  
28  
29 class PdfDocument implements Document {  
30     public void open() {  
31         System.out.println("Opening PDF Document");  
32     }  
33 }  
34  
35 class ExcelDocument implements Document {  
36     public void open() {  
37         System.out.println("Opening Excel Document");  
38     }  
39 }
```

Problems Javadoc Declaration Console × Debug

<terminated> FactoryMethodPatternExample [Java Application] C:\Users\Fyza\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86\_64\_21.0.2.v20240123-0840\jre\bin\java

Opening Word Document  
Opening PDF Document  
Opening Excel Document

## Exercise 3: Implementing the Builder Pattern

### PROGRAM:

```
package DesignPatterns;

public class BuilderPatternExample {

    public static void main(String[] args) {

        Computer gamingPC = new Computer.Builder()
            .setCPU("Intel i9")
            .setRAM("32GB")
            .setStorage("1TB SSD")
            .setOperatingSystem("Windows 11 Pro")
            .build();

        Computer officePC = new Computer.Builder()
            .setCPU("Intel i5")
            .setRAM("16GB")
            .setStorage("512GB SSD")
            .setOperatingSystem("Windows 12")
            .build();

        System.out.println("Gaming PC: " + gamingPC);
        System.out.println("Office PC: " + officePC);
    }
}

class Computer {

    private String CPU;

    private String RAM;

    private String storage;
```

```
private String operatingSystem;
```

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

```
public String toString() {  
    return "CPU: " + CPU + ", RAM: " + RAM + ", Storage: " + storage + ", OS: " +  
operatingSystem;  
}
```

```
public static class Builder {  
    private String CPU;  
    private String RAM;  
    private String storage;  
    private String operatingSystem;
```

```
    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 setOperatingSystem(String operatingSystem) {

        this.operatingSystem = operatingSystem;

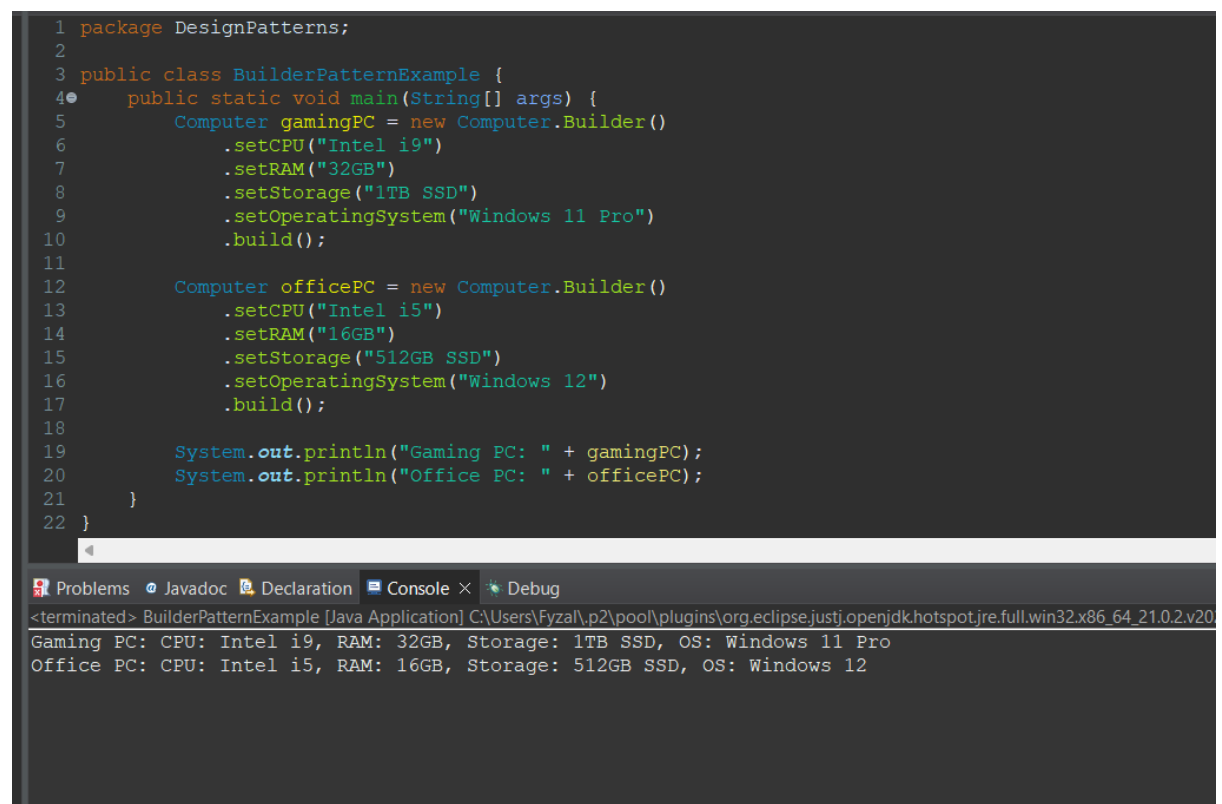
        return this;
    }

    public Computer build() {

        return new Computer(this);
    }
}
}

```

## OUTPUT:



The screenshot shows an IDE with a Java file named `BuilderPatternExample` in the `DesignPatterns` package. The code defines a `Computer` class with a `Builder` pattern and a `main` method that creates and builds two `Computer` objects: a gaming PC and an office PC. The output window shows the results of the `build` method, displaying the specifications for each PC.

```

1 package DesignPatterns;
2
3 public class BuilderPatternExample {
4     public static void main(String[] args) {
5         Computer gamingPC = new Computer.Builder()
6             .setCPU("Intel i9")
7             .setRAM("32GB")
8             .setStorage("1TB SSD")
9             .setOperatingSystem("Windows 11 Pro")
10            .build();
11
12        Computer officePC = new Computer.Builder()
13            .setCPU("Intel i5")
14            .setRAM("16GB")
15            .setStorage("512GB SSD")
16            .setOperatingSystem("Windows 12")
17            .build();
18
19        System.out.println("Gaming PC: " + gamingPC);
20        System.out.println("Office PC: " + officePC);
21    }
22 }

```

Problems Javadoc Declaration Console × Debug

<terminated> BuilderPatternExample [Java Application] C:\Users\Fyzal\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86\_64\_21.0.2.v20

Gaming PC: CPU: Intel i9, RAM: 32GB, Storage: 1TB SSD, OS: Windows 11 Pro  
Office PC: CPU: Intel i5, RAM: 16GB, Storage: 512GB SSD, OS: Windows 12

## Exercise 4: Implementing the Adapter Pattern

### PROGRAM:

```
package DesignPatterns;

class AdapterPatternExample {

    public static void main(String[] args) {

        PaymentProcessor gpayAdapter = new GPayAdapter(new GPayGateway());

        gpayAdapter.processPayment(500);

        PaymentProcessor paytmAdapter = new PaytmAdapter(new PaytmGateway());

        paytmAdapter.processPayment(750);

    }

}

interface PaymentProcessor {

    void processPayment(int amountInRupees);

}

class GPayGateway {

    public void payUsingGPay(int amount) {

        System.out.println("GPay processed payment of Rs." + amount);

    }

}

class PaytmGateway {

    public void doPaytmPayment(int amount) {

        System.out.println("Paytm processed payment of Rs." + amount);

    }

}

class GPayAdapter implements PaymentProcessor {

    private GPayGateway gpay;

    public GPayAdapter(GPayGateway gpay) {

        this.gpay = gpay;    }
```

```

    public void processPayment(int amountInRupees) {
        gpay.payUsingGPay(amountInRupees);
    }
}

```

```

class PaytmAdapter implements PaymentProcessor {
    private PaytmGateway paytm;

    public PaytmAdapter(PaytmGateway paytm) {
        this.paytm = paytm;
    }

    public void processPayment(int amountInRupees) {
        paytm.doPaytmPayment(amountInRupees);
    }
}

```

## OUTPUT:

```

1 package DesignPatterns;
2
3 class AdapterPatternExample {
4     public static void main(String[] args) {
5         PaymentProcessor gpayAdapter = new GPayAdapter(new GPayGateway());
6         gpayAdapter.processPayment(500);
7
8         PaymentProcessor paytmAdapter = new PaytmAdapter(new PaytmGateway());
9         paytmAdapter.processPayment(750);
10    }
11 }
12
13 interface PaymentProcessor {
14     void processPayment(int amountInRupees);
15 }
16
17 class GPayGateway {
18     public void payUsingGPay(int amount) {
19         System.out.println("GPay processed payment of Rs." + amount);
20     }
21 }
22

```

Problems Javadoc Declaration Console × Debug

<terminated> AdapterPatternExample [Java Application] C:\Users\Fyza\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86\_64\_21.0.2.v20240123

GPay processed payment of Rs.500  
Paytm processed payment of Rs.750

## Exercise 5: Implementing the Decorator Pattern

### PROGRAM:

```
package DesignPatterns;

public class DecoratorPatternExample {

    public static void main(String[] args) {

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

        notifier.send("Your OTP is 839201");

    }

}

interface Notifier {

    void send(String message);

}

class EmailNotifier implements Notifier {

    public void send(String message) {

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

    }

}

abstract class NotifierDecorator implements Notifier {

    protected Notifier notifier;

    public NotifierDecorator(Notifier notifier) {

        this.notifier = notifier;

    }

    public void send(String message) {

        notifier.send(message);

    }

}
```

```

    }
}

class SMSNotifier extends NotifierDecorator {

    public SMSNotifier(Notifier notifier) {

        super(notifier);

    }

    public void send(String message) {

        notifier.send(message);

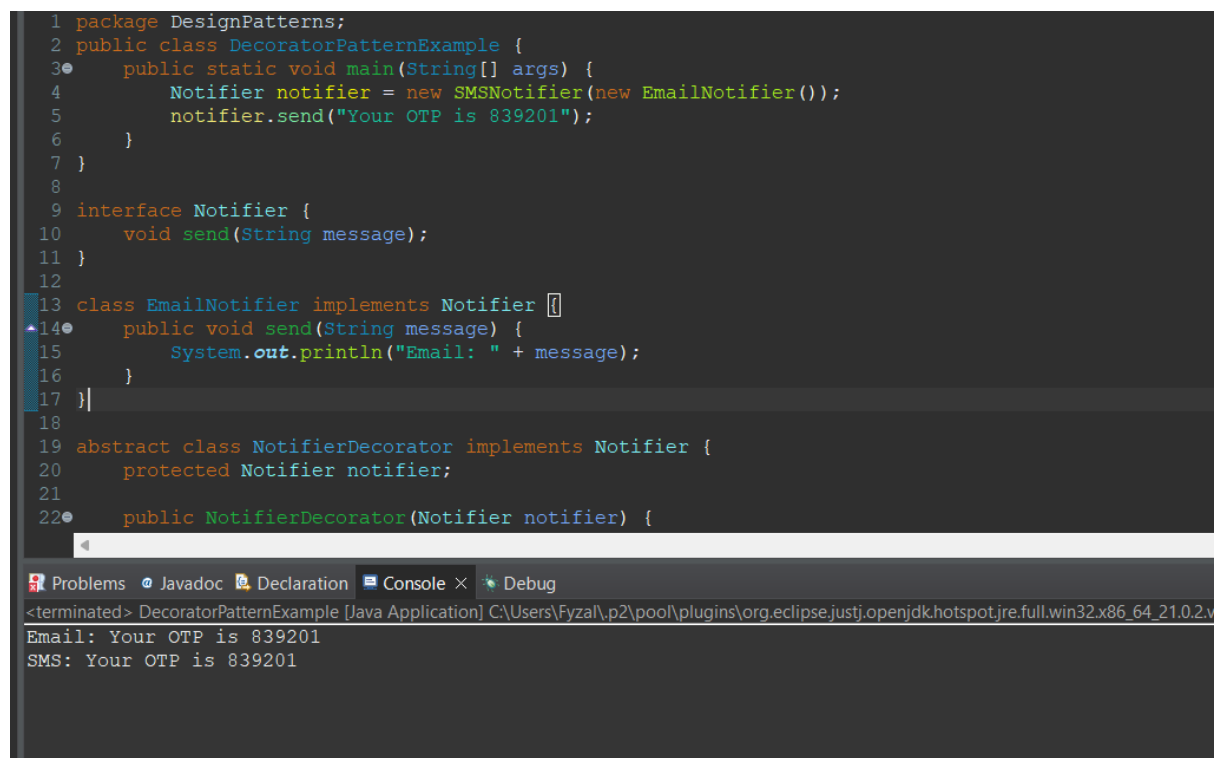
        System.out.println("SMS: " + message);

    }

}

```

## OUTPUT:



The screenshot shows an IDE with a Java code editor and a console window. The code implements the Decorator Pattern for sending messages via Email or SMS. The main method creates an SMSNotifier that wraps an EmailNotifier and sends the message "Your OTP is 839201". The console output shows the message being sent via Email first, and then via SMS.

```

1 package DesignPatterns;
2 public class DecoratorPatternExample {
3     public static void main(String[] args) {
4         Notifier notifier = new SMSNotifier(new EmailNotifier());
5         notifier.send("Your OTP is 839201");
6     }
7 }
8
9 interface Notifier {
10     void send(String message);
11 }
12
13 class EmailNotifier implements Notifier {
14     public void send(String message) {
15         System.out.println("Email: " + message);
16     }
17 }
18
19 abstract class NotifierDecorator implements Notifier {
20     protected Notifier notifier;
21
22     public NotifierDecorator(Notifier notifier) {

```

Problems Javadoc Declaration Console × Debug

<terminated> DecoratorPatternExample [Java Application] C:\Users\Fyza\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86\_64\_21.0.2.v

Email: Your OTP is 839201

SMS: Your OTP is 839201

## Exercise 6: Implementing the Proxy Pattern

### PROGRAM:

```
package DesignPatterns;

public class ProxyPatternExample {
    public static void main(String[] args) {
        Image image1 = new ProxyImage("city.jpg");
        Image image2 = new ProxyImage("city.jpg");
        image1.display();
        image2.display();
    }
}

interface Image {
    void display();
}

class RealImage implements Image {
    private String filename;

    public RealImage(String filename) {
        this.filename = filename;
        System.out.println("Loading image : " + filename);
    }

    public void display() {
        System.out.println("Displaying image: " + filename);
    }
}
```

```

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();
    }
}

```

## OUTPUT:

```

1 package DesignPatterns;
2 public class ProxyPatternExample {
3     public static void main(String[] args) {
4         Image image1 = new ProxyImage("city.jpg");
5         Image image2 = new ProxyImage("city.jpg");
6
7         image1.display();
8         image2.display();
9     }
10 }
11
12 interface Image {
13     void display();
14 }
15
16 class RealImage implements Image {
17     private String filename;
18
19     public RealImage(String filename) {
20         this.filename = filename;
21         System.out.println("Loading image : " + filename);
22     }

```

Problems Javadoc Declaration Console × Debug

<terminated> ProxyPatternExample [Java Application] C:\Users\Fyzal\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full

Loading image : city.jpg  
 Displaying image: city.jpg  
 Loading image : city.jpg  
 Displaying image: city.jpg

## Exercise 7: Implementing the Observer Pattern

### PROGRAM:

```
package DesignPatterns;

import java.util.*;

public class ObserverPatternExample {

    public static void main(String[] args) {

        StockMarket stockMarket = new StockMarket();

        Observer mobileApp = new MobileApp("MobileApp");
        Observer webApp = new WebApp("WebApp");

        stockMarket.register(mobileApp);
        stockMarket.register(webApp);

        stockMarket.setPrice("TATA", 840.5);
        stockMarket.setPrice("RELIANCE", 2512.3);

        stockMarket.deregister(webApp);

        stockMarket.setPrice("INFY", 1540.0);
    }
}

interface Stock {

    void register(Observer o);

    void deregister(Observer o);

    void notifyObservers(String stock, double price);

}
```



```

class StockMarket implements Stock {
    private List<Observer> observers = new ArrayList<>();
    private Map<String, Double> stockPrices = new HashMap<>();

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

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

    public void notifyObservers(String stock, double price) {
        for (Observer o : observers) {
            o.update(stock, price);
        }
    }

    public void setPrice(String stock, double price) {
        stockPrices.put(stock, price);
        notifyObservers(stock, price);
    }
}

interface Observer {
    void update(String stock, double price);
}

class MobileApp implements Observer {
    private String name;

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

```

```

    }

    public void update(String stock, double price) {

        System.out.println(name + " received update: " + stock + " = " + price);

    }

}

class WebApp implements Observer {

    private String name;

    public WebApp(String name) {

        this.name = name;

    }

    public void update(String stock, double price) {

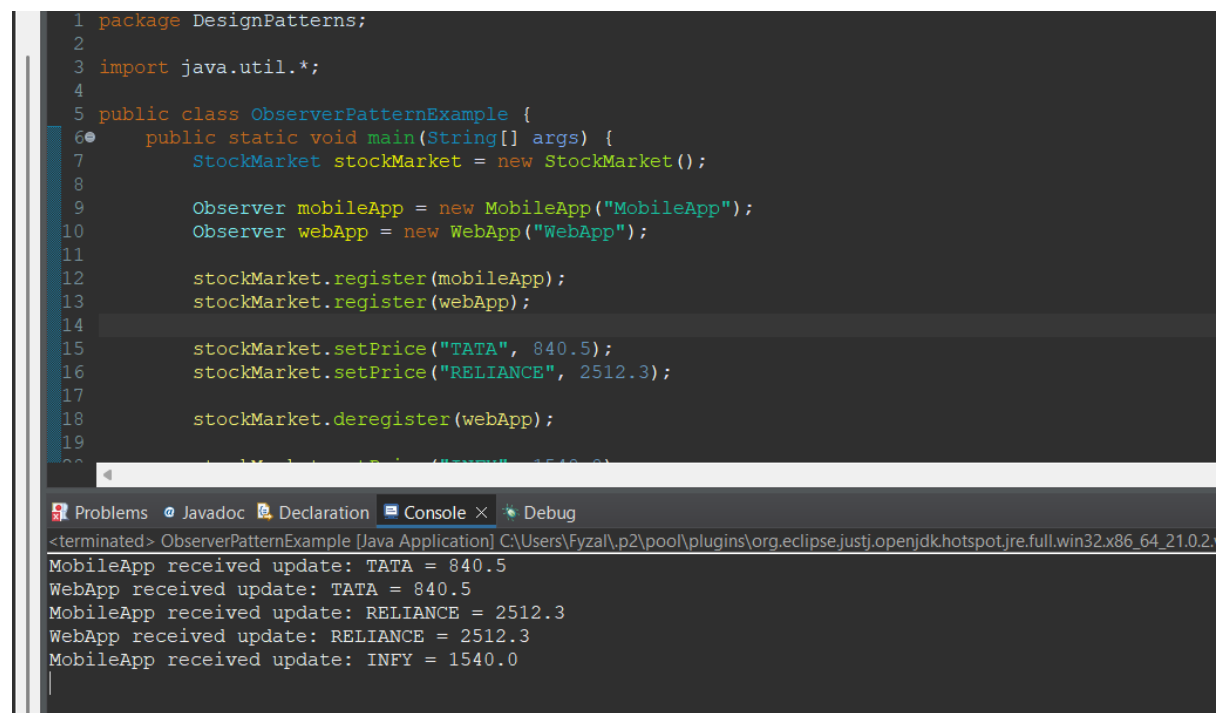
        System.out.println(name + " received update: " + stock + " = " + price);

    }

}

```

## OUTPUT:



The screenshot shows an IDE with a Java file named `ObserverPatternExample.java`. The code defines a `StockMarket` class and two observer classes, `MobileApp` and `WebApp`. The `main` method in `ObserverPatternExample` creates a `StockMarket` instance, registers `MobileApp` and `WebApp` as observers, sets prices for "TATA", "RELIANCE", and "INFY", and then deregisters `WebApp`.

The console output shows the following sequence of events:

```

<terminated> ObserverPatternExample [Java Application] C:\Users\Fyza\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_21.0.2
MobileApp received update: TATA = 840.5
WebApp received update: TATA = 840.5
MobileApp received update: RELIANCE = 2512.3
WebApp received update: RELIANCE = 2512.3
MobileApp received update: INFY = 1540.0

```

## Exercise 8: Implementing the Strategy Pattern

### PROGRAM:

```
package DesignPatterns;

public class StrategyPatternExample {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();

        context.setStrategy(new CreditCardPayment("Fyzal", "489948181811"));

        context.pay(250.0);

        context.setStrategy(new UpiPayment("fyzal@upi"));

        context.pay(120.5);

    }

}

interface PaymentStrategy {

    void pay(double amount);

}

class CreditCardPayment implements PaymentStrategy {

    private String cardHolderName;

    private String cardNumber;

    public CreditCardPayment(String cardHolderName, String cardNumber) {

        this.cardHolderName = cardHolderName;

        this.cardNumber = cardNumber;

    }

    public void pay(double amount) {

        System.out.println("Paid Rs." + amount + " using Credit Card by " + cardHolderName);

    }

}
```

```

class UpiPayment implements PaymentStrategy {
    private String upiId;

    public UpiPayment(String upiId) {
        this.upiId = upiId;
    }

    public void pay(double amount) {
        System.out.println("Paid Rs" + amount + " using UPI ID " + upiId);
    }
}

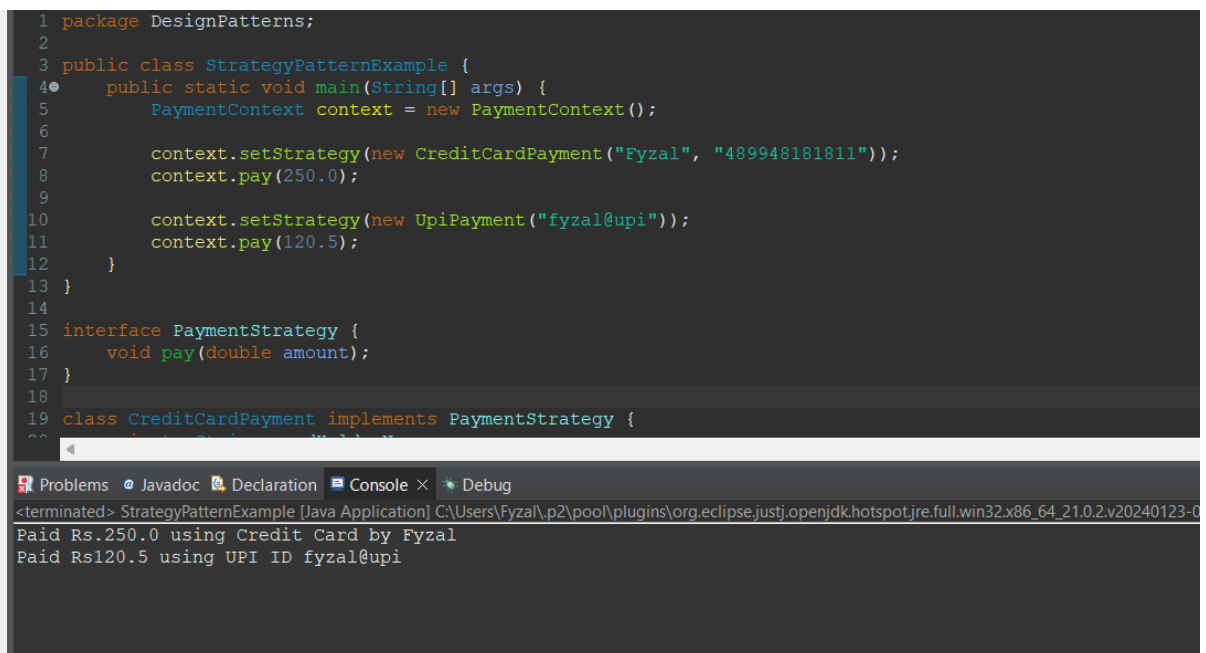
class PaymentContext {
    private PaymentStrategy strategy;

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

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

```

## OUTPUT:



The screenshot shows an IDE with a Java file named `StrategyPatternExample.java`. The code defines a `PaymentStrategy` interface with a `pay(double amount)` method. It has two concrete implementations: `CreditCardPayment` and `UPIPayment`. The `PaymentContext` class uses the Strategy pattern to delegate the `pay` method to the appropriate `PaymentStrategy` implementation. The `main` method in `StrategyPatternExample` creates a `PaymentContext` object, sets it to `CreditCardPayment` and then `UPIPayment`, and calls `pay` on the context with amounts 250.0 and 120.5 respectively.

```

1 package DesignPatterns;
2
3 public class StrategyPatternExample {
4     public static void main(String[] args) {
5         PaymentContext context = new PaymentContext();
6
7         context.setStrategy(new CreditCardPayment("Fyzal", "489948181811"));
8         context.pay(250.0);
9
10        context.setStrategy(new UPIPayment("fyzal@upi"));
11        context.pay(120.5);
12    }
13 }
14
15 interface PaymentStrategy {
16     void pay(double amount);
17 }
18
19 class CreditCardPayment implements PaymentStrategy {
20     private String name;
21     private String cardNo;
22
23     CreditCardPayment(String name, String cardNo) {
24         this.name = name;
25         this.cardNo = cardNo;
26     }
27
28     void pay(double amount) {
29         System.out.println("Paid Rs." + amount + " using Credit Card by " + name);
30     }
31 }
32
33 class UPIPayment implements PaymentStrategy {
34     private String upiId;
35
36     UPIPayment(String upiId) {
37         this.upiId = upiId;
38     }
39
40     void pay(double amount) {
41         System.out.println("Paid Rs." + amount + " using UPI ID " + upiId);
42     }
43 }
44
45 class PaymentContext {
46     private PaymentStrategy strategy;
47
48     void setStrategy(PaymentStrategy strategy) {
49         this.strategy = strategy;
50     }
51
52     void pay(double amount) {
53         strategy.pay(amount);
54     }
55 }

```

The console output shows the execution results:

```

<terminated> StrategyPatternExample [Java Application] C:\Users\Fyzal\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.2.v20240123-0
Paid Rs.250.0 using Credit Card by Fyzal
Paid Rs120.5 using UPI ID fyzal@upi

```

## Exercise 9: Implementing the Command Pattern

### PROGRAM:

```
package DesignPatterns;

public class CommandPatternExample {

    public static void main(String[] args) {

        Light light = new Light();

        Command lightOn = new LightOnCommand(light);

        Command lightOff = new LightOffCommand(light);

        RemoteControl remote = new RemoteControl();

        remote.setCommand(lightOn);

        remote.pressButton();

        remote.setCommand(lightOff);

        remote.pressButton();

    }

}

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;  
    }  
}
```

$$\}$$

## OUTPUT:

[illegible]

## Exercise 10: Implementing the MVC Pattern

### PROGRAM:

```
package DesignPatterns;

public class MVCPatternExample {
    public static void main(String[] args) {
        Student model = new Student("Fyzal", "22UIT034", "B");
        StudentView view = new StudentView();
        StudentController controller = new StudentController(model, view);

        controller.updateView();

        controller.setStudentName("Fyzal K");
        controller.setStudentGrade("A+");

        controller.updateView();
    }
}

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;
    }
}
```



```
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 setGrade(String grade) {  
    this.grade = grade;  
}  
}
```

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

```
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 void setStudentGrade(String grade) {

    model.setGrade(grade);

}

public void updateView() {

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

}

}

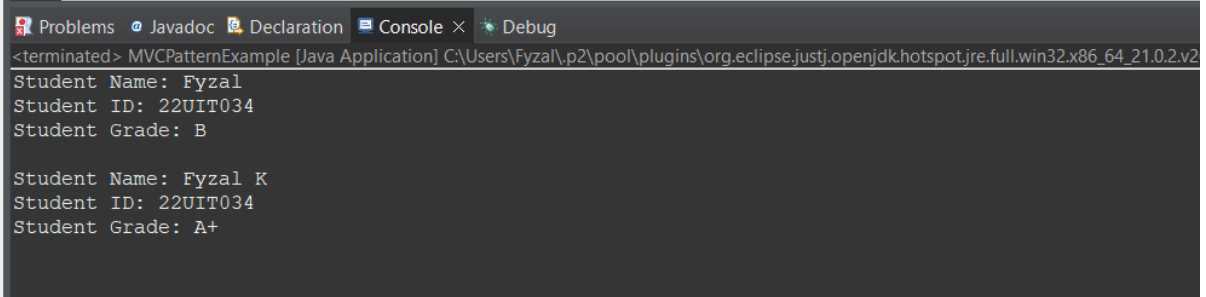
```

## OUTPUT:

```

1 package DesignPatterns;
2
3 public class MVCPatternExample {
4     public static void main(String[] args) {
5         Student model = new Student("Fyzal", "22UIT034", "B");
6         StudentView view = new StudentView();
7         StudentController controller = new StudentController(model, view);
8
9         controller.updateView();
10
11         controller.setStudentName("Fyzal K");
12         controller.setStudentGrade("A+");
13
14         controller.updateView();
15     }
16 }
17
18 class Student {
19     private String name;
20     private String id;
21     private String grade;
22
23     public Student(String name, String id, String grade) {
24         this.name = name;
25         this.id = id;
26         this.grade = grade;
27     }
28
29     public String getName() {
30         return name;
31     }
32
33     public String getId() {
34         return id;
35     }
36
37     public String getGrade() {
38         return grade;
39     }
40
41     public void setName(String name) {
42         this.name = name;
43     }
44
45     public void setId(String id) {
46         this.id = id;
47     }
48
49     public void setGrade(String grade) {
50         this.grade = grade;
51     }
52 }

```


 The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The output displays the execution of the MVCPatternExample application. It shows the initial state of a student (Name: Fyzal, ID: 22UIT034, Grade: B) and the state after updates (Name: Fyzal K, ID: 22UIT034, Grade: A+).

## Exercise 11: Implementing Dependency Injection

### PROGRAM:

```
package DesignPatterns;

public class DependencyInjectionExample {

    public static void main(String[] args) {

        CustomerRepository repository = new CustomerRepositoryImpl();

        CustomerService service = new CustomerService(repository);

        Customer customer = service.findCustomerById("22UIT034");

        System.out.println("Customer Found: " + customer.getName() + ", ID: " +
customer.getId());

    }

}

interface CustomerRepository {

    Customer findCustomerById(String id);

}

class CustomerRepositoryImpl implements CustomerRepository {

    public Customer findCustomerById(String id) {

        return new Customer(id, "Fyzal K");

    }

}

class CustomerService {

    private final CustomerRepository customerRepository;

    public CustomerService(CustomerRepository customerRepository) {

        this.customerRepository = customerRepository;

    }

    public Customer findCustomerById(String id) {

        return customerRepository.findCustomerById(id);

    }

}
```

```

    }
}

class Customer {

    private final String id;

    private final String name;

    public Customer(String id, String name) {

        this.id = id;

        this.name = name;

    }

    public String getId() {

        return id;

    }

    public String getName() {

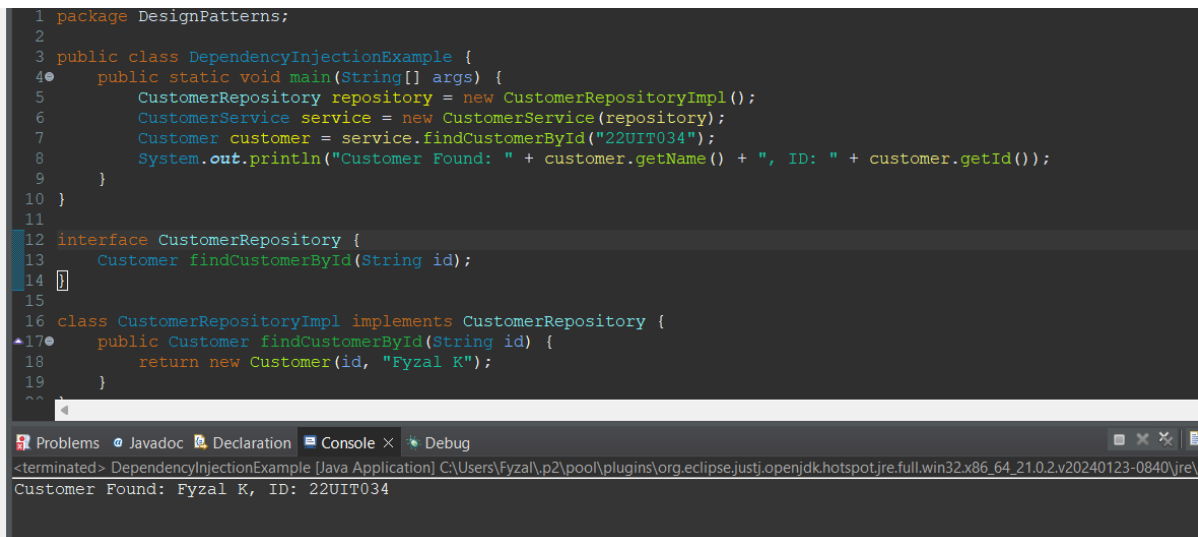
        return name;

    }

}

```

## OUTPUT:



The screenshot shows an IDE with a Java code editor and a console window. The code defines a package, a main class, an interface, and an implementation. The console output shows the result of running the application.

```

1 package DesignPatterns;
2
3 public class DependencyInjectionExample {
4     public static void main(String[] args) {
5         CustomerRepository repository = new CustomerRepositoryImpl();
6         CustomerService service = new CustomerService(repository);
7         Customer customer = service.findCustomerById("22UIT034");
8         System.out.println("Customer Found: " + customer.getName() + ", ID: " + customer.getId());
9     }
10 }
11
12 interface CustomerRepository {
13     Customer findCustomerById(String id);
14 }
15
16 class CustomerRepositoryImpl implements CustomerRepository {
17     public Customer findCustomerById(String id) {
18         return new Customer(id, "Fyzal K");
19     }
20 }

```

Console Output:

```

<terminated> DependencyInjectionExample [Java Application] C:\Users\Fyzal\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_21.0.2.v20240123-0840\jre\
Customer Found: Fyzal K, ID: 22UIT034

```