Digital Nurture 4.0

Deep Skilling Handbook

Java FSE -Solution

DESIGN PATTERNS

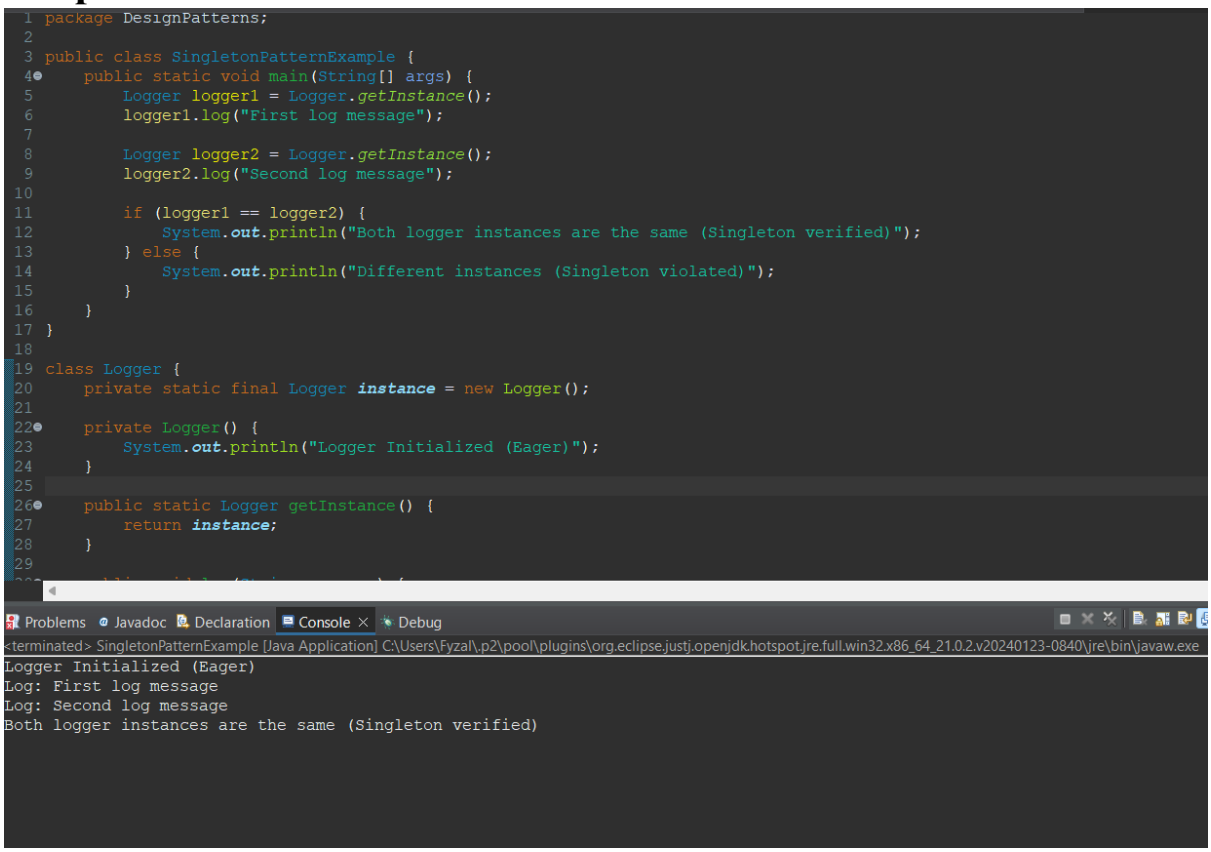
Exercise 1: Implementing the Singleton Pattern:

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:



```
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;
    }
```

Problems Javadoc Declaration Console Debug

<terminated> SingletonPatternExample [Java Application] C:\Users\Fyzal\.p2\pool\plugins\org.eclipse.justj.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

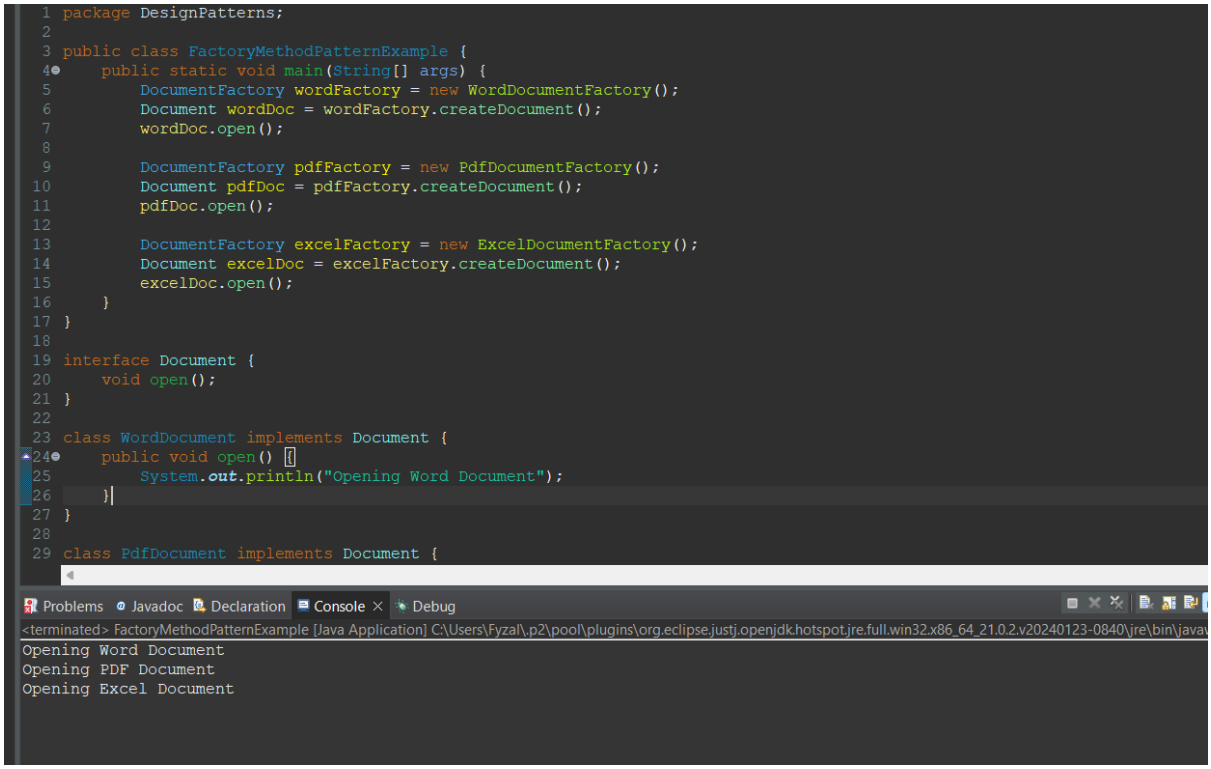
PROGRAM:

```
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:



```
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 {
```

```
Problems  Javadoc  Declaration  Console  Debug
<terminated> FactoryMethodPatternExample [Java Application] C:\Users\Fyzal\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.2.v20240123-0840\jre\bin\javaw
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:

```
 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
```

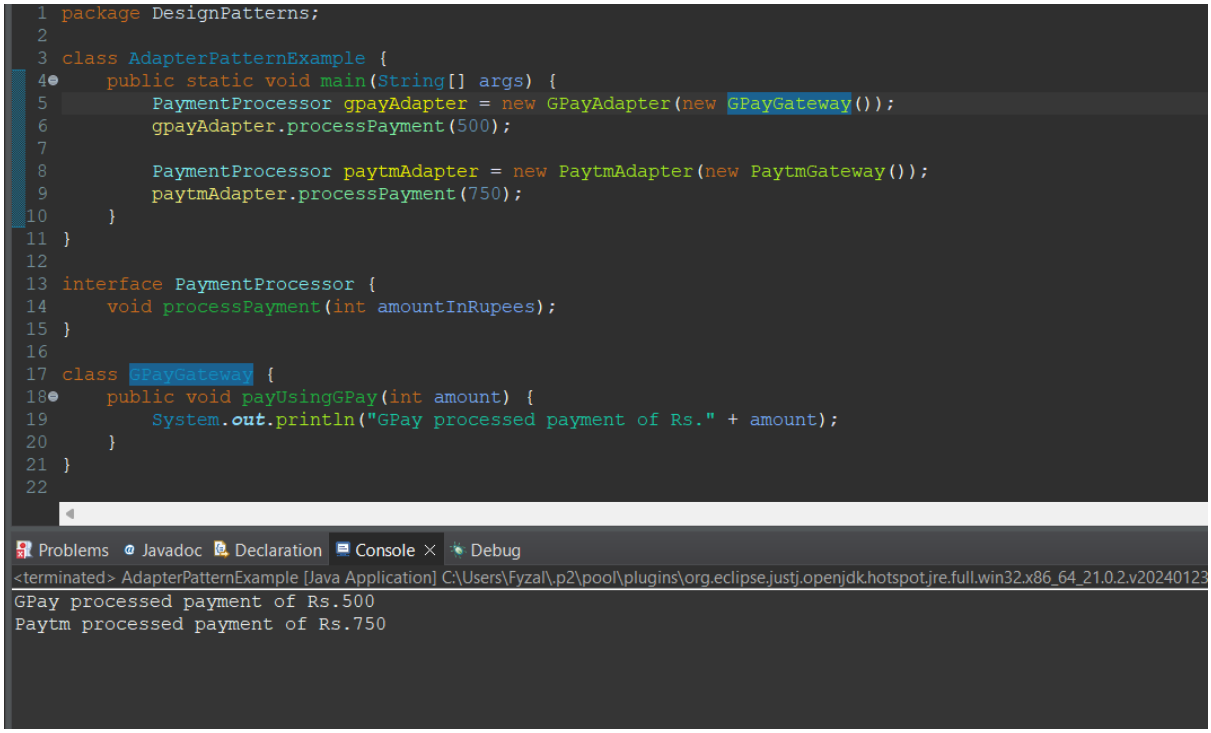
Exercise 4: Implementing the Adapter Pattern

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:



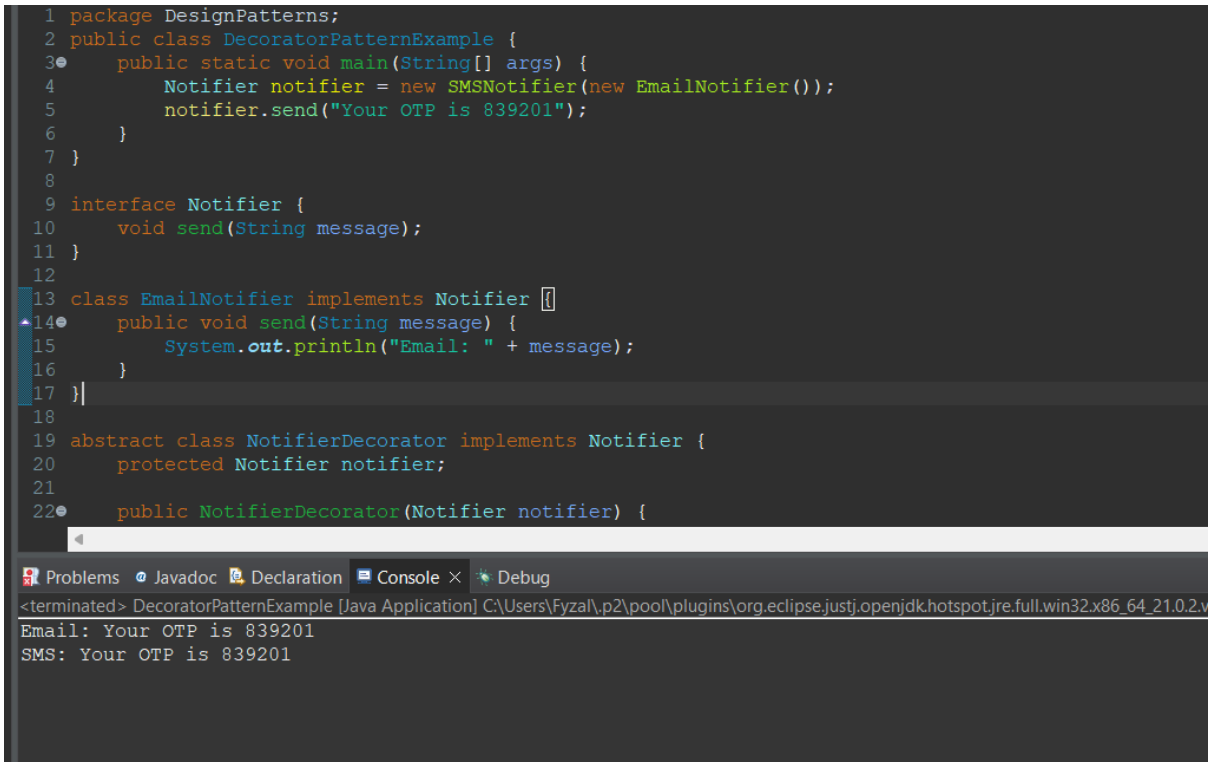
Exercise 5: Implementing the Decorator Pattern

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:



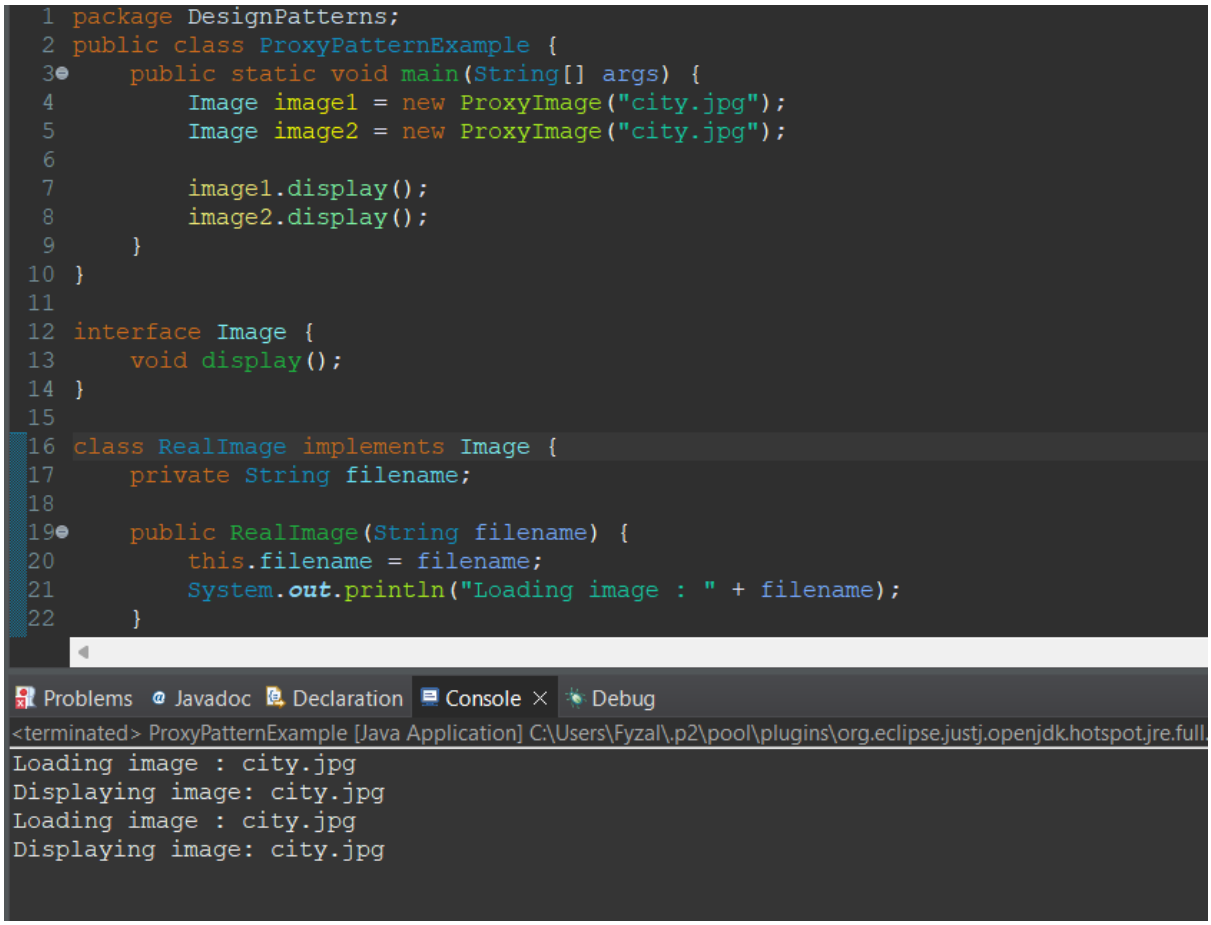
Exercise 6: Implementing the Proxy Pattern

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:



Exercise 7: Implementing the Observer Pattern

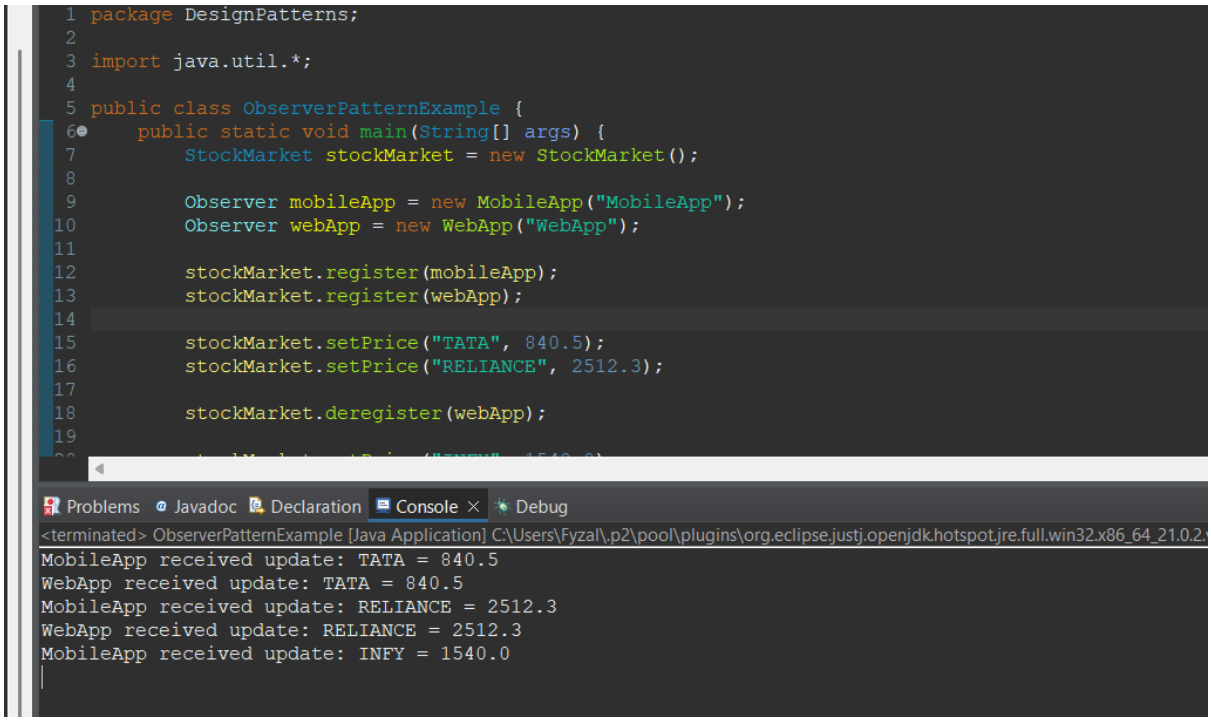
PROGRAM:

```
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:



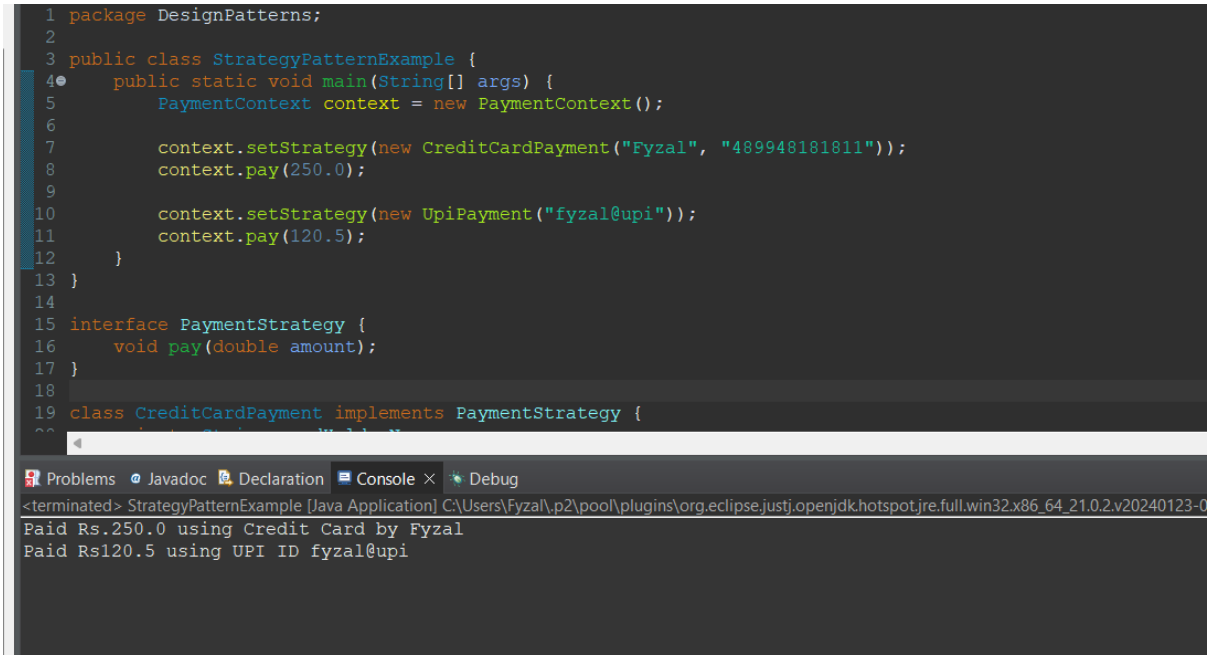
Exercise 8: Implementing the Strategy Pattern

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:



Exercise 9: Implementing the Command Pattern

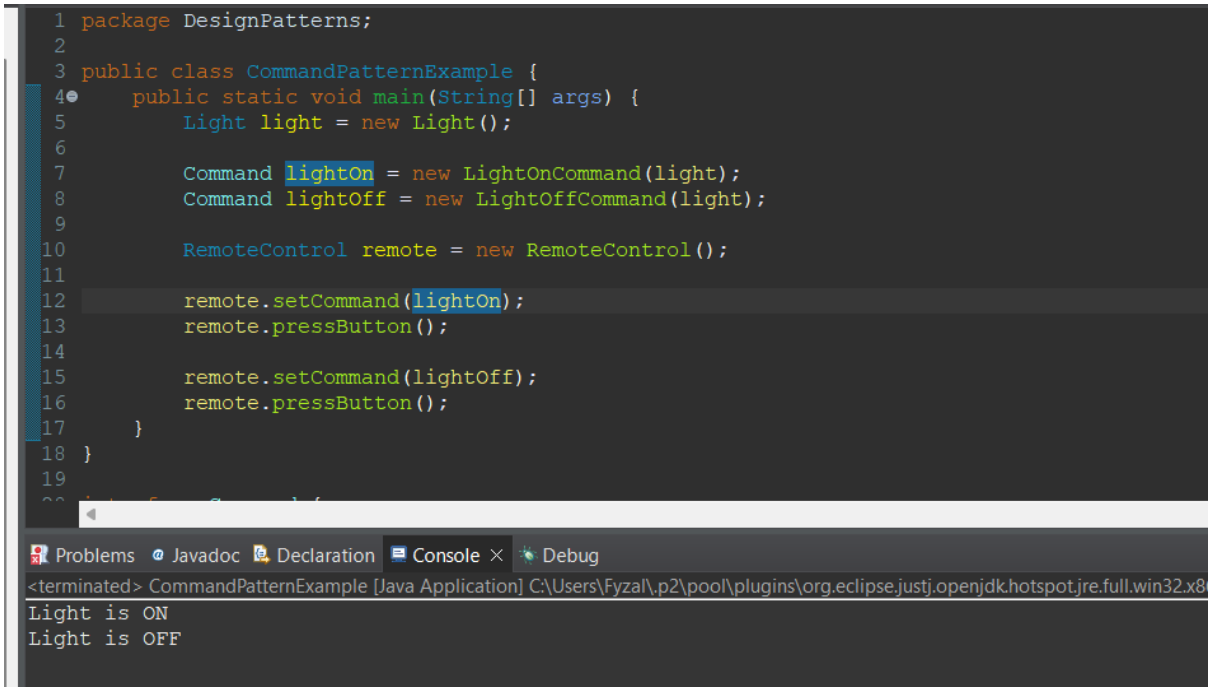
PROGRAM:

```
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;
    }
```

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

OUTPUT:



```
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();
    }
}
```

```
Problems  Javadoc  Declaration  Console  Debug
<terminated> CommandPatternExample [Java Application] C:\Users\Fyzal\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x8
Light is ON
Light is OFF
```

Exercise 10: Implementing the MVC Pattern

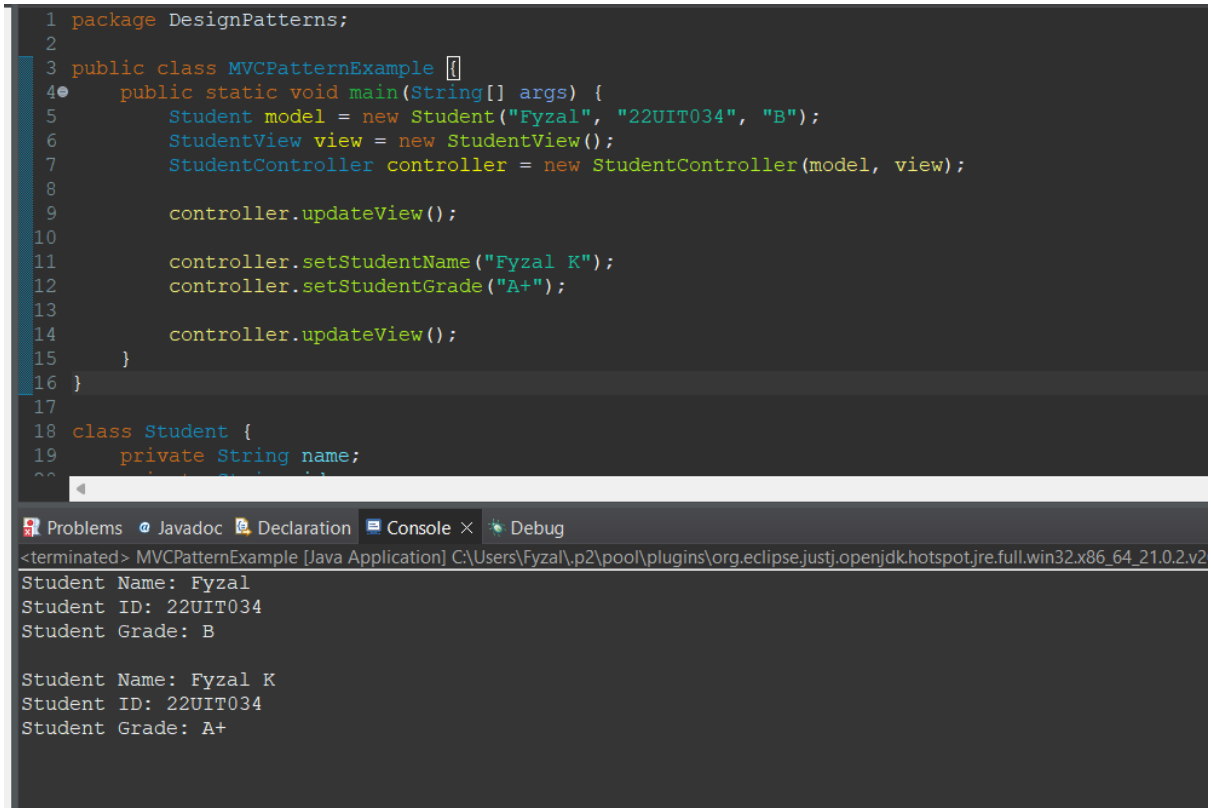
PROGRAM:

```
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:



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: