Week 1 hands on exercise solutions

Design patterns and Principles

Exercise 1: Implementing the Singleton Pattern

Scenario:

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

Steps:

1. **Create a New Java Project:**
   - Create a new Java project named **SingletonPatternExample**.
2. **Define a Singleton Class:**
   - Create a class named Logger that has a private static instance of itself.
   - Ensure the constructor of Logger is private.
   - Provide a public static method to get the instance of the Logger class.
3. **Implement the Singleton Pattern:**
   - Write code to ensure that the Logger class follows the Singleton design pattern.
4. **Test the Singleton Implementation:**
   - Create a test class to verify that only one instance of Logger is created and used across the application.

Solution:

Code:

lab - InventoryManagementSystem/src/Implementing_Singleton_Pattern/Logger.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer

- InventoryManagementSystem
  - JRE System Library [JavaSE-17]
  - src
    - Implementing_Singleton_Pattern
      - Logger.java
      - TestSingleton.java
    - module-info.java

Logger.java | TestSingleton.java

```java
package Implementing_Singleton_Pattern;

public class Logger {

    private static Logger loggerInstance;

    private Logger() {
        System.out.println("Logger Initialized...");
    }

    public static Logger getInstance() {
        if (loggerInstance == null) {
            loggerInstance = new Logger();
        }
        return loggerInstance;
    }

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

Task List

Find All Activate...

Outline

- Implementing_Singleton_Pattern
  - Logger
    - loggerInstance : Logger
    - Logger()
    - getInstance() : Logger
    - log(String) : void

Problems Javadoc Declaration Console

<terminated> TestSingleton [Java Application] C:\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe (21-Jun-2025, 7:42:45 pm – 7:42:46 pm) [pid: 4404]

```
Logger Initialized...
Log: This is the first log message.
Log: This is the second log message.
Both logger instances are the same. Singleton confirmed.
```

Writable | Smart Insert | 23 : 1 : 476

lab - InventoryManagementSystem/src/Implementing_Singleton_Pattern/TestSingleton.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer

- InventoryManagementSystem
  - JRE System Library [JavaSE-17]
  - src
    - Implementing_Singleton_Pattern
      - Logger.java
      - TestSingleton.java
    - module-info.java

Logger.java | TestSingleton.java

```java
package Implementing_Singleton_Pattern;

public class TestSingleton {
    public static void main(String[] args) {
        Logger logger1 = Logger.getInstance();
        Logger logger2 = Logger.getInstance();

        logger1.log("This is the first log message.");
        logger2.log("This is the second log message.");

        if (logger1 == logger2) {
            System.out.println("Both logger instances are the same. Singleton confirmed.");
        } else {
            System.out.println("Logger instances are different. Singleton failed.");
        }
    }
}
```

Task List

Find All Activate...

Outline

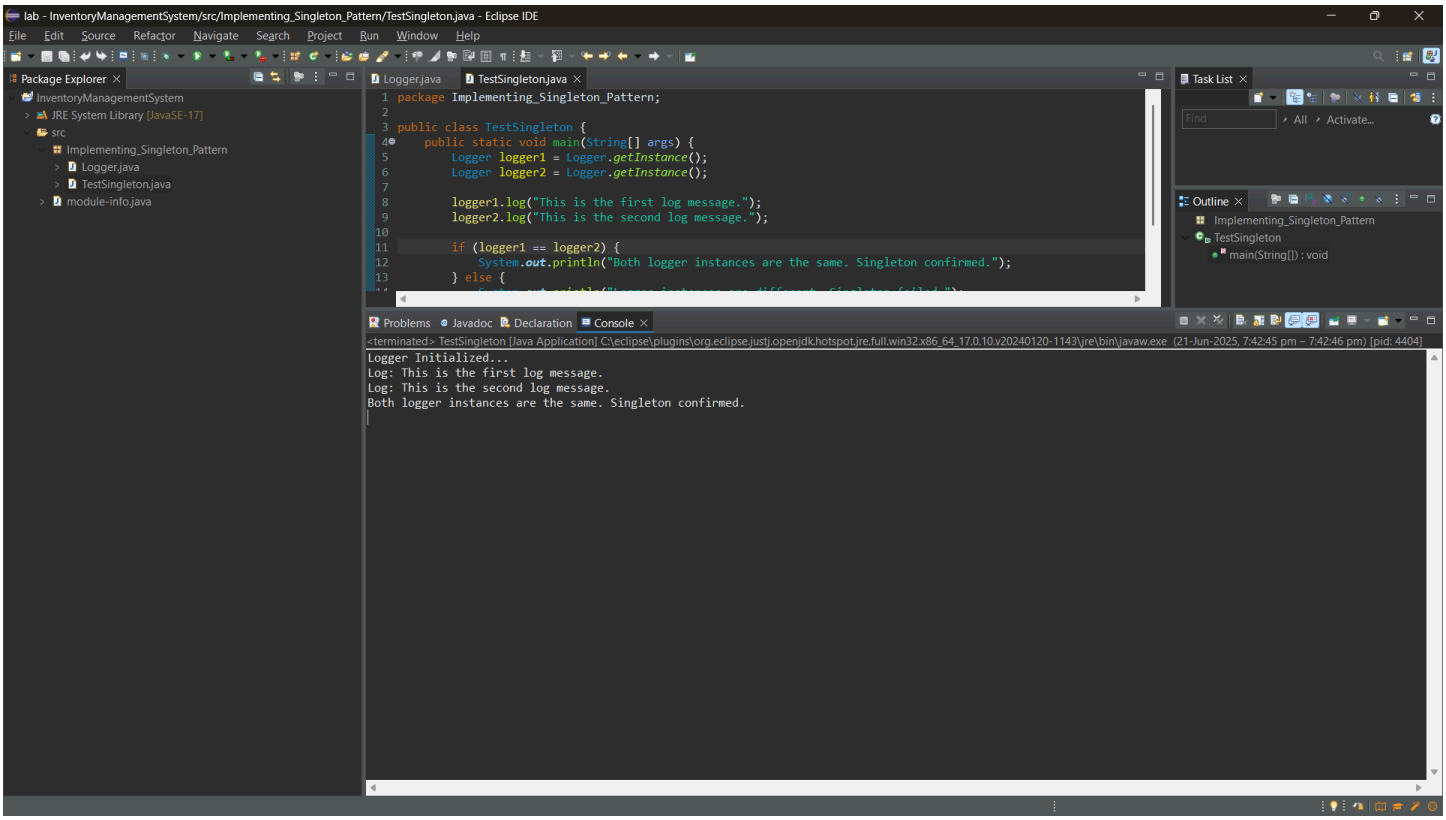
- Implementing_Singleton_Pattern
  - TestSingleton
    - main(String[]) : void

Problems Javadoc Declaration Console

<terminated> TestSingleton [Java Application] C:\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe (21-Jun-2025, 7:42:45 pm – 7:42:46 pm) [pid: 4404]

```
Logger Initialized...
Log: This is the first log message.
Log: This is the second log message.
Both logger instances are the same. Singleton confirmed.
```

Writable | Smart Insert | 11 : 1 : 332



Exercise 2: Implementing the Factory Method Pattern

Scenario:

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

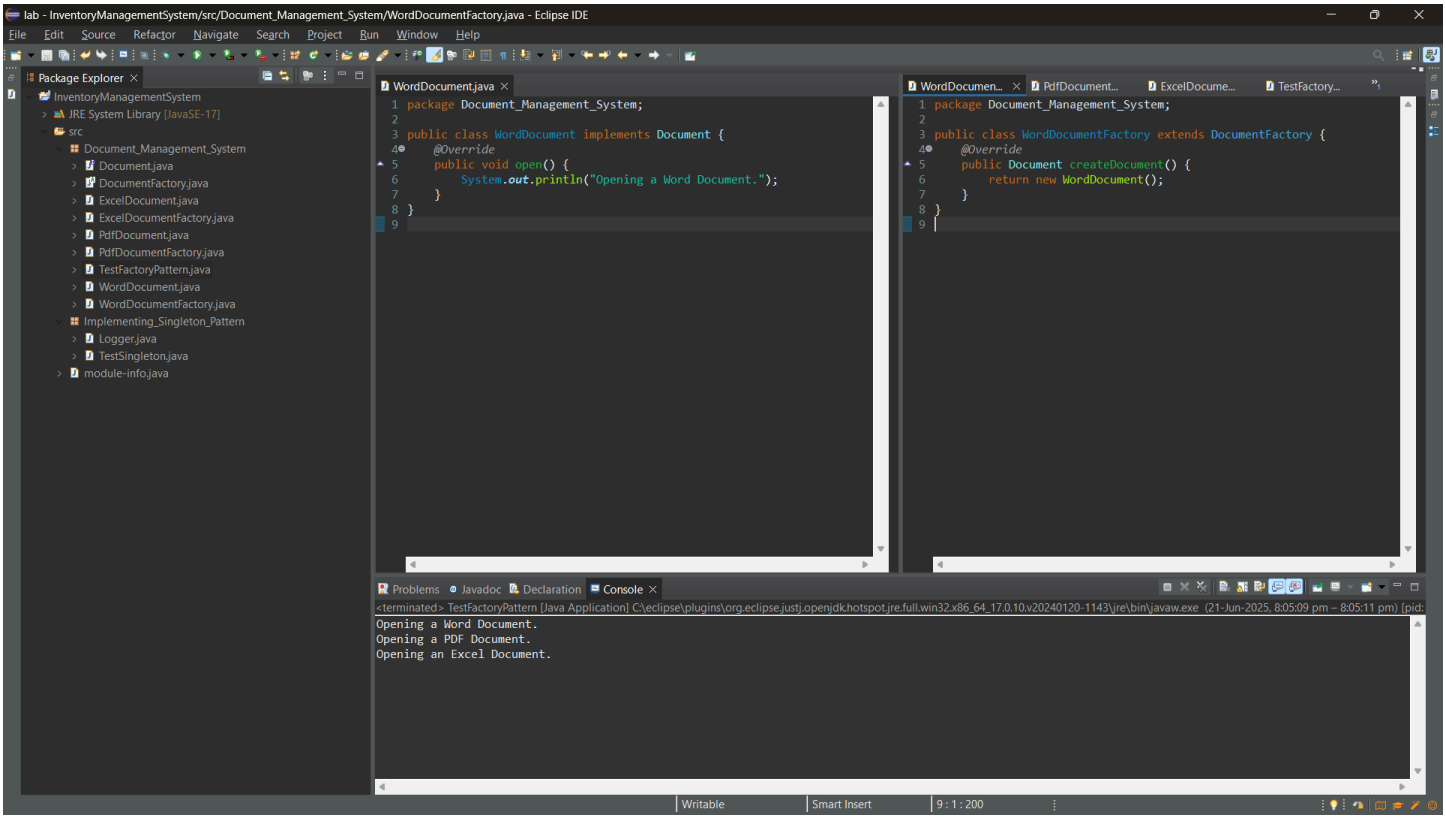
Steps:

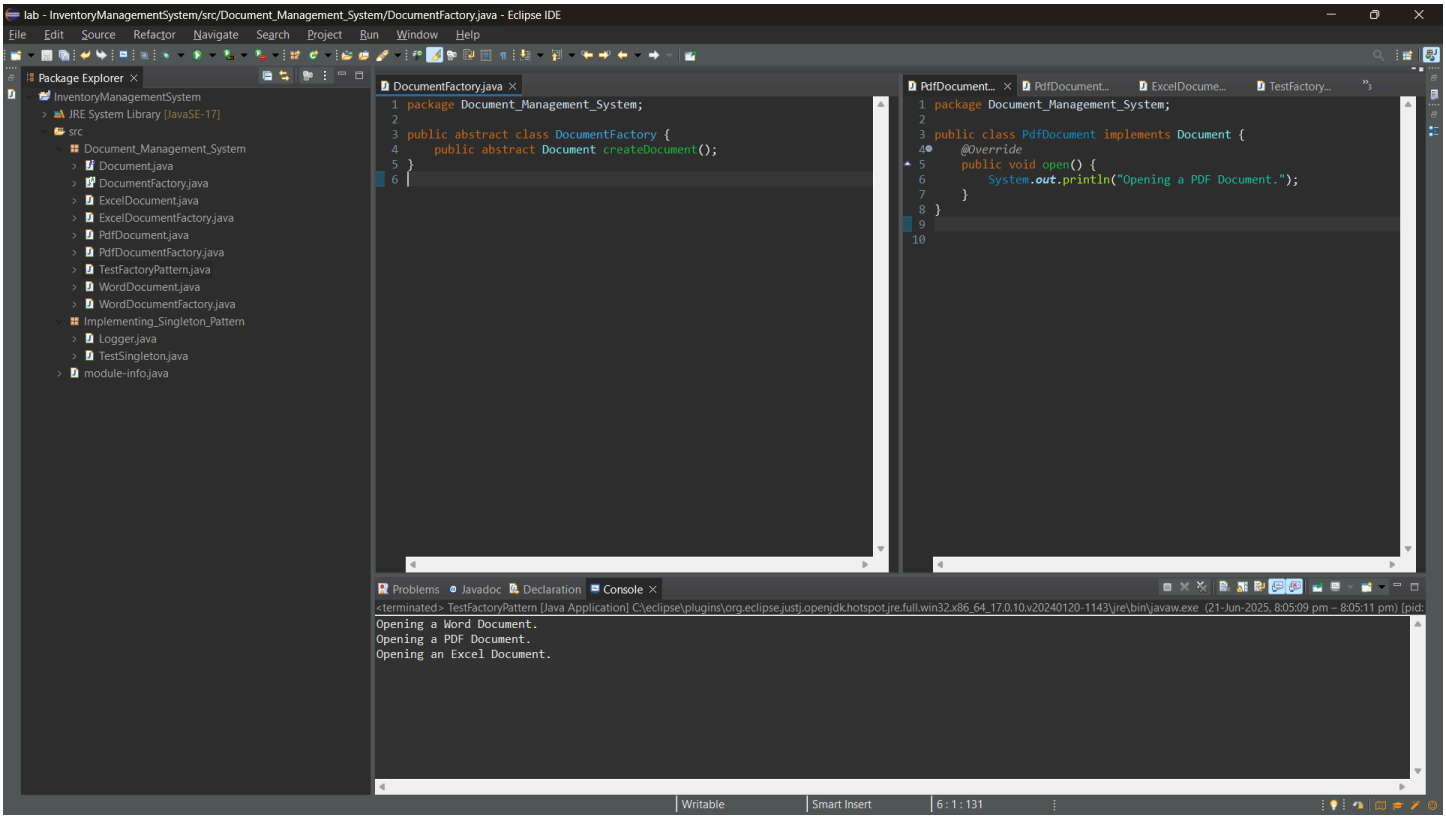
1. **Create a New Java Project:**
   - Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   - Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document Classes:**
   - Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **Implement the Factory Method:**
   - Create an abstract class **DocumentFactory** with a method **createDocument()**.
   - Create concrete factory classes for each document type that extends DocumentFactory and implements the **createDocument()** method.
5. **Test the Factory Method Implementation:**

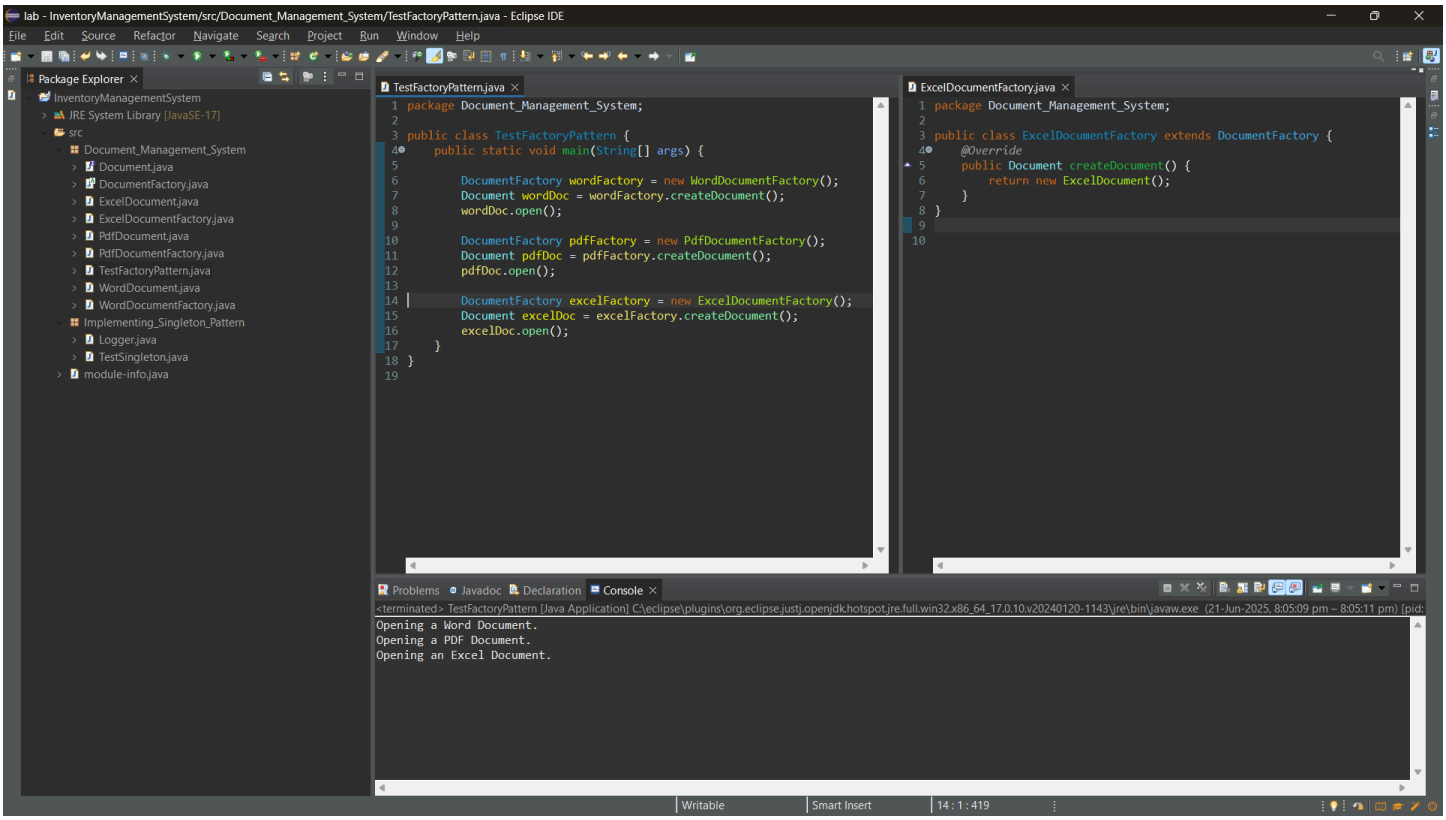
- Create a test class to demonstrate the creation of different document types using the factory method.

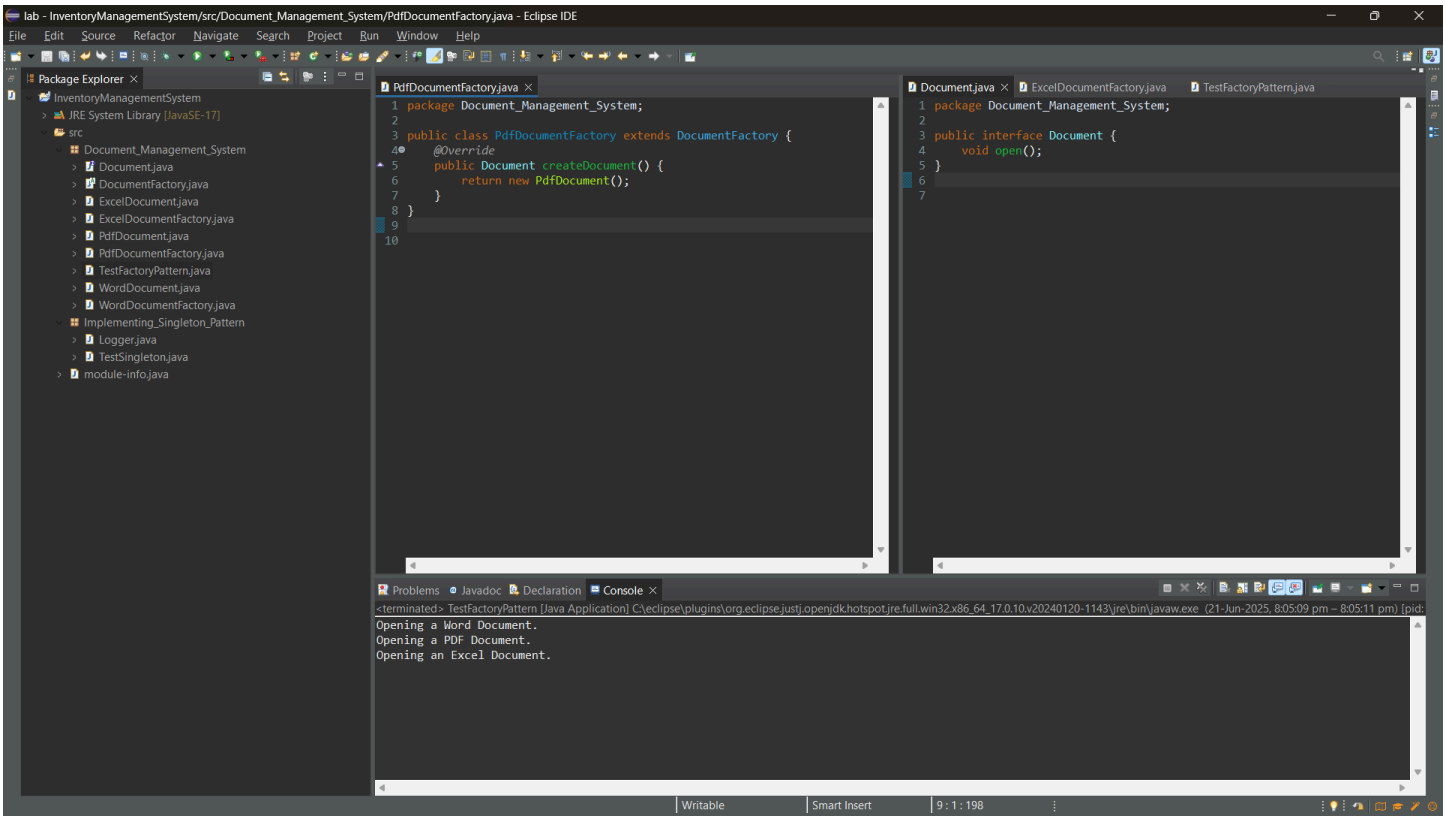
Solution:

Code:









Exercise 3: Implementing the Builder Pattern

Scenario:

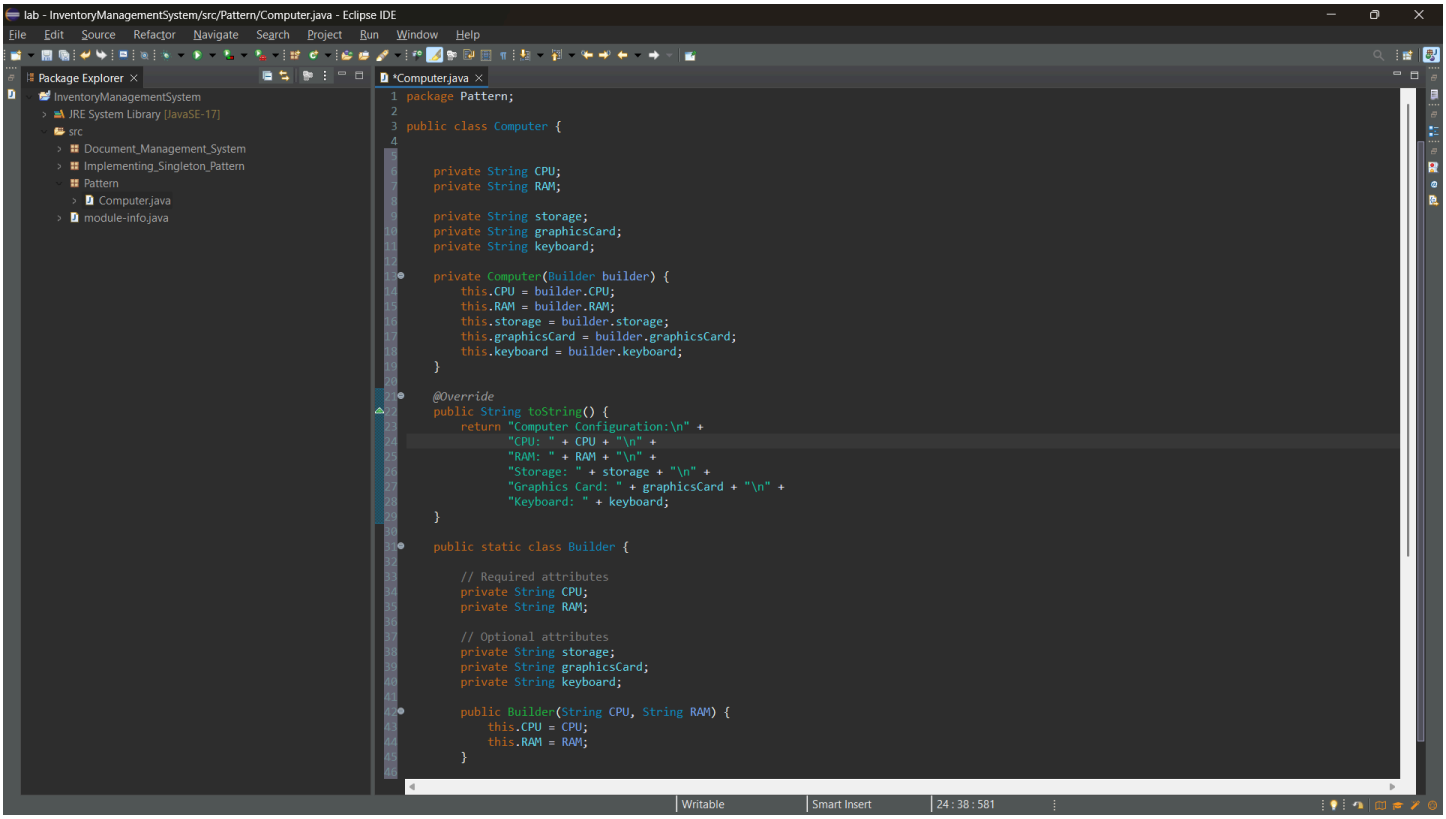
You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

Steps:

1. **Create a New Java Project:**
   - Create a new Java project named **BuilderPatternExample**.
2. **Define a Product Class:**
   - Create a class **Computer** with attributes like **CPU**, **RAM**, **Storage**, etc.
3. **Implement the Builder Class:**
   - Create a static nested Builder class inside Computer with methods to set each attribute.
   - Provide a **build()** method in the Builder class that returns an instance of Computer.
4. **Implement the Builder Pattern:**
   - Ensure that the **Computer** class has a private constructor that takes the **Builder** as a parameter.
5. **Test the Builder Implementation:**
   - Create a test class to demonstrate the creation of different configurations of Computer using the Builder pattern.

Solution:

Code:



lab - InventoryManagementSystem/src/Pattern/Computer.java - Eclipse IDE

```java
    public String toString() {
        return "Computer Configuration:\n" +
                "CPU: " + CPU + "\n" +
                "RAM: " + RAM + "\n" +
                "Storage: " + storage + "\n" +
                "Graphics Card: " + graphicsCard + "\n" +
                "Keyboard: " + keyboard;
    }

    public static class Builder {

        // Required attributes
        private String CPU;
        private String RAM;

        // Optional attributes
        private String storage;
        private String graphicsCard;
        private String keyboard;

        public Builder(String CPU, String RAM) {
            this.CPU = CPU;
            this.RAM = RAM;
        }

        public Builder setStorage(String storage) {
            this.storage = storage;
            return this;
        }

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

        public Builder setKeyboard(String keyboard) {
            this.keyboard = keyboard;
            return this;
        }

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

lab - InventoryManagementSystem/src/Pattern/TestBuilderPattern.java - Eclipse IDE

```java
package Pattern;

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

        Computer basicComputer = new Computer.Builder("Intel i3", "8GB")
                                        .build();

        // Create a high-end gaming computer
        Computer gamingComputer = new Computer.Builder("Intel i9", "32GB")
                                        .setStorage("1TB SSD")
                                        .setGraphicsCard("NVIDIA RTX 4090")
                                        .setKeyboard("Mechanical Keyboard")
                                        .build();

        System.out.println(basicComputer);
        System.out.println();
        System.out.println(gamingComputer);
    }
}
```

```
<terminated> TestBuilderPattern [Java Application] C:\eclipse\plugins\org.eclipse.justj.ope
Computer Configuration:
CPU: Intel i3
RAM: 8GB
Storage: null
Graphics Card: null
Keyboard: null

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

Exercise 4: Implementing the Adapter Pattern

Scenario:

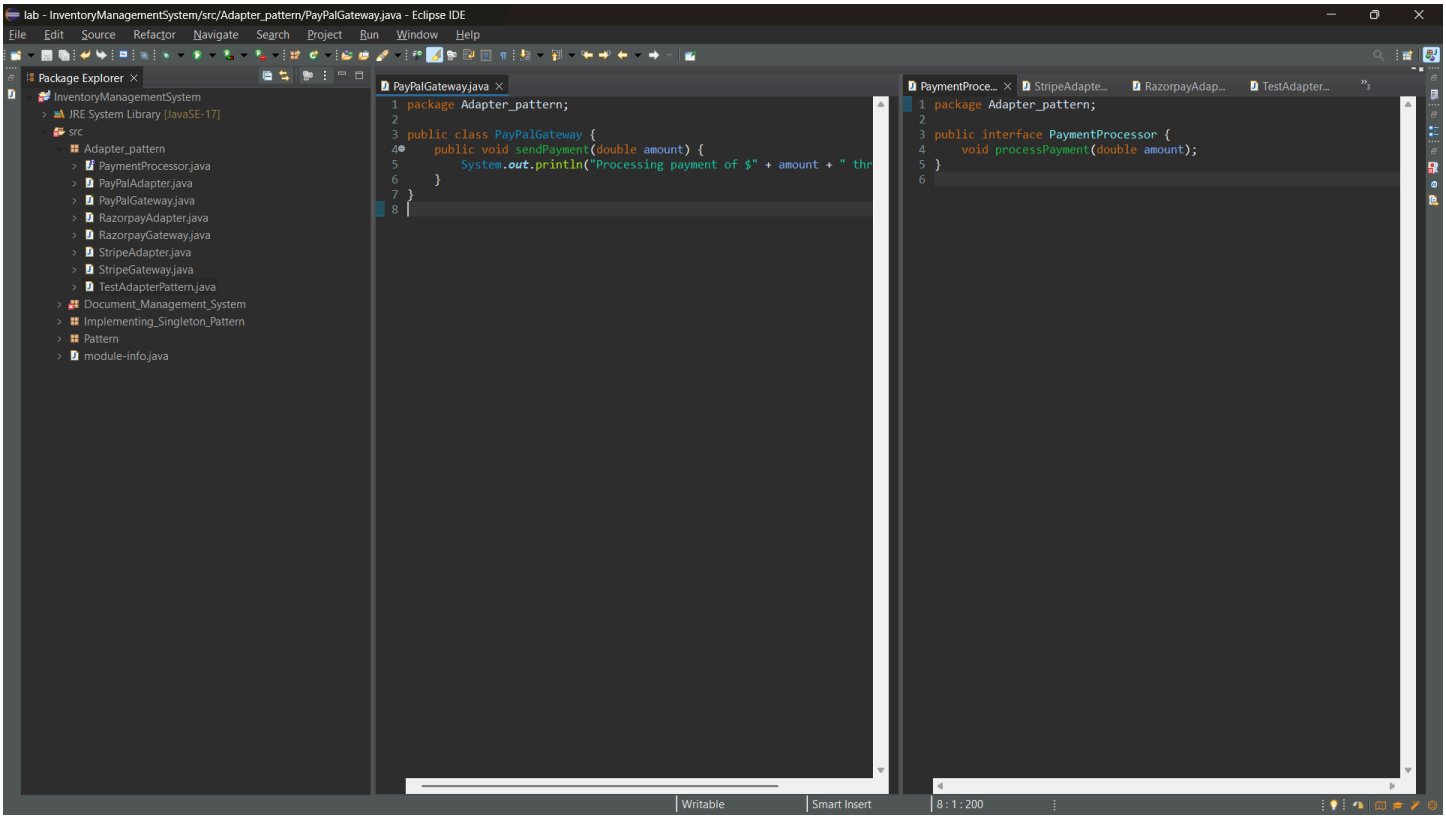
You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

Steps:

1. **Create a New Java Project:**
   - Create a new Java project named **AdapterPatternExample**.
2. **Define Target Interface:**
   - Create an interface **PaymentProcessor** with methods like **processPayment()**.
3. **Implement Adaptee Classes:**
   - Create classes for different payment gateways with their own methods.
4. **Implement the Adapter Class:**
   - Create an adapter class for each payment gateway that implements PaymentProcessor and translates the calls to the gateway-specific methods.
5. **Test the Adapter Implementation:**
   - Create a test class to demonstrate the use of different payment gateways through the adapter.

Solution:

Code:



lab - InventoryManagementSystem/src/Adapter_pattern/PayPalAdapter.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer

- InventoryManagementSystem
  - JRE System Library [JavaSE-17]
  - src
    - Adapter_pattern
      - PaymentProcessor.java
      - PayPalAdapter.java
      - PayPalGateway.java
      - RazorpayAdapter.java
      - RazorpayGateway.java
      - StripeAdapter.java
      - StripeGateway.java
      - TestAdapterPattern.java
    - Document_Management_System
    - Implementing_Singleton_Pattern
    - Pattern
    - module-info.java

PayPalAdapter.java

```java
package Adapter_pattern;
public class PayPalAdapter implements PaymentProcessor {

    private PayPalGateway payPalGateway;

    public PayPalAdapter(PayPalGateway payPalGateway) {
        this.payPalGateway = payPalGateway;
    }

    @Override
    public void processPayment(double amount) {
        payPalGateway.sendPayment(amount);
    }
}
```

StripeGateway.ja... | StripeAdapter.ja... | RazorpayAdapter.... | TestAdapterPatt...

```java
package Adapter_pattern;

public class StripeGateway {
    public void makeStripePayment(double amount) {
        System.out.println("Processing payment of $" + amount + " thr
    }
}
```

Writable | Smart Insert | 15 : 1 : 359

lab - InventoryManagementSystem/src/Adapter_pattern/RazorpayGateway.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer

- InventoryManagementSystem
  - JRE System Library [JavaSE-17]
  - src
    - Adapter_pattern
      - PaymentProcessor.java
      - PayPalAdapter.java
      - PayPalGateway.java
      - RazorpayAdapter.java
      - RazorpayGateway.java
      - StripeAdapter.java
      - StripeGateway.java
      - TestAdapterPattern.java
    - Document_Management_System
    - Implementing_Singleton_Pattern
    - Pattern
    - module-info.java

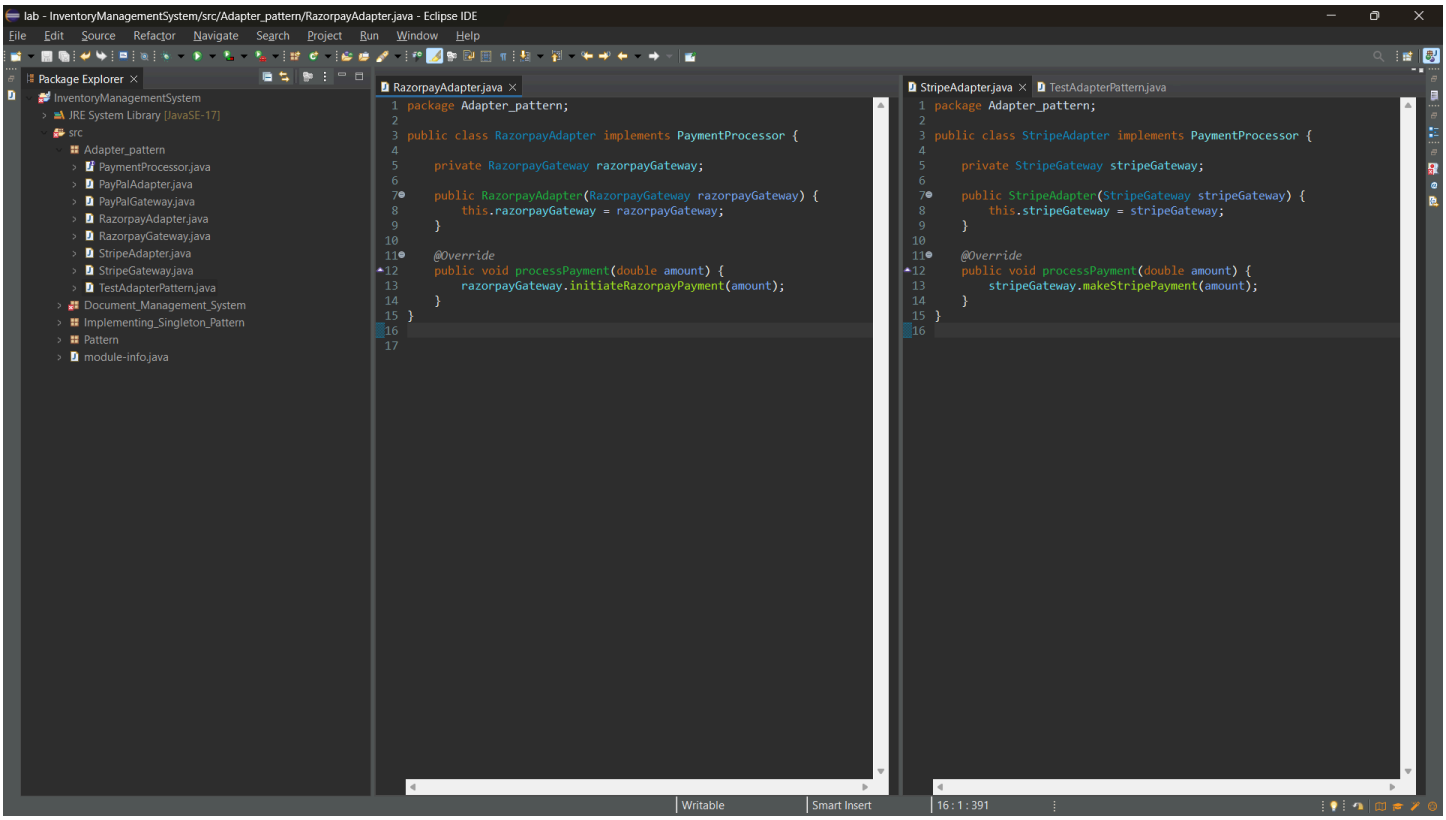
RazorpayGateway.java

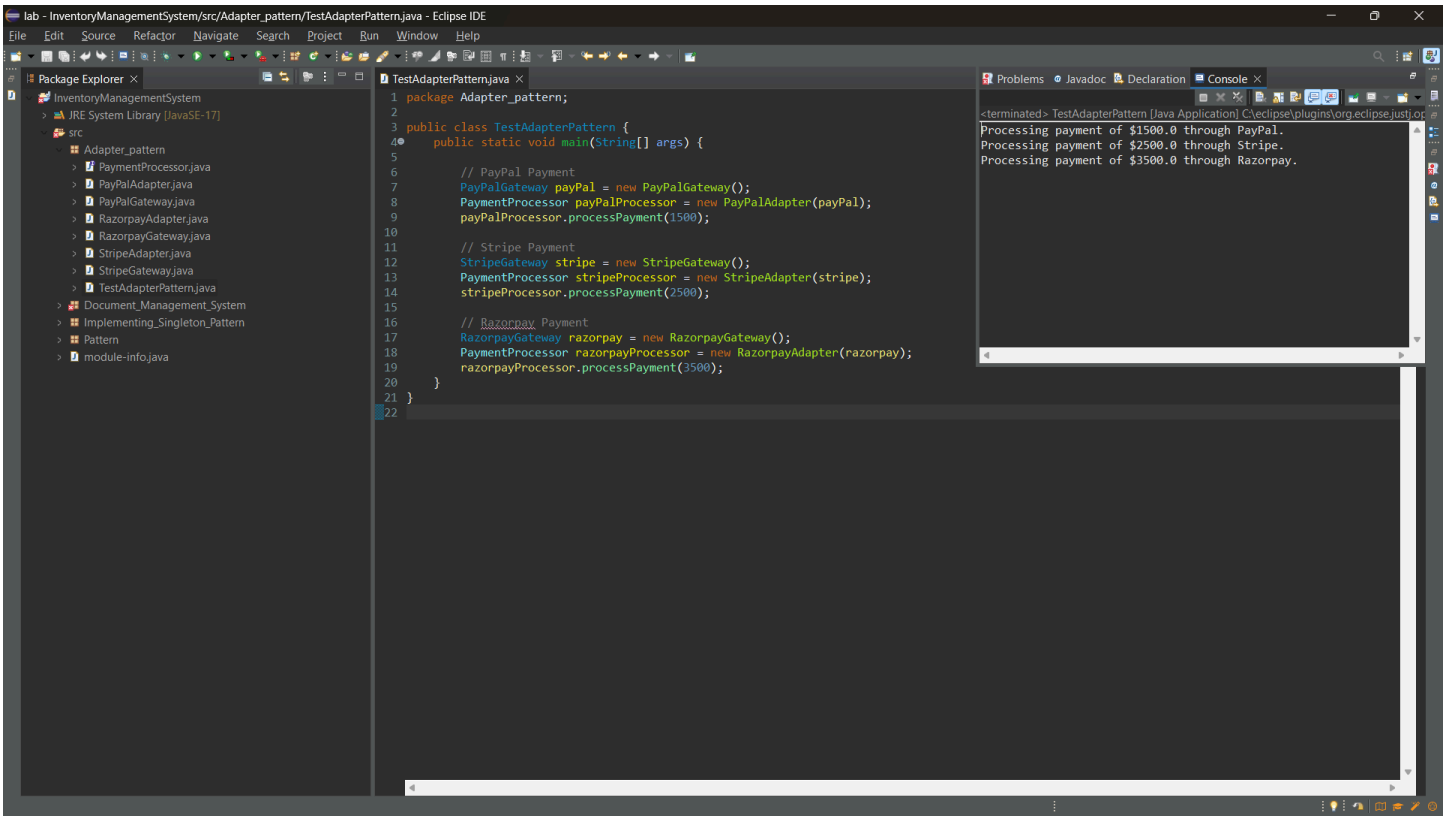
```java
package Adapter_pattern;
public class RazorpayGateway {
    public void initiateRazorpayPayment(double amount) {
        System.out.println("Processing payment of $" + amount + " thr
    }
}
```

PayPalAdapte... | StripeAdapte... | RazorpayAdap... | TestAdapter...

```java
package Adapter_pattern;

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

        // PayPal Payment
        PayPalGateway payPal = new PayPalGateway();
        PaymentProcessor payPalProcessor = new PayPalAdapter(payPal);
        payPalProcessor.processPayment(1500);

        // Stripe Payment
        StripeGateway stripe = new StripeGateway();
        PaymentProcessor stripeProcessor = new StripeAdapter(stripe);
        stripeProcessor.processPayment(2500);

        // Razorpay Payment
        RazorpayGateway razorpay = new RazorpayGateway();
        PaymentProcessor razorpayProcessor = new RazorpayAdapter(razo
        razorpayProcessor.processPayment(3500);
    }
}
```

Writable | Smart Insert | 7 : 1 : 214





Exercise 5: Implementing the Decorator Pattern

Scenario:

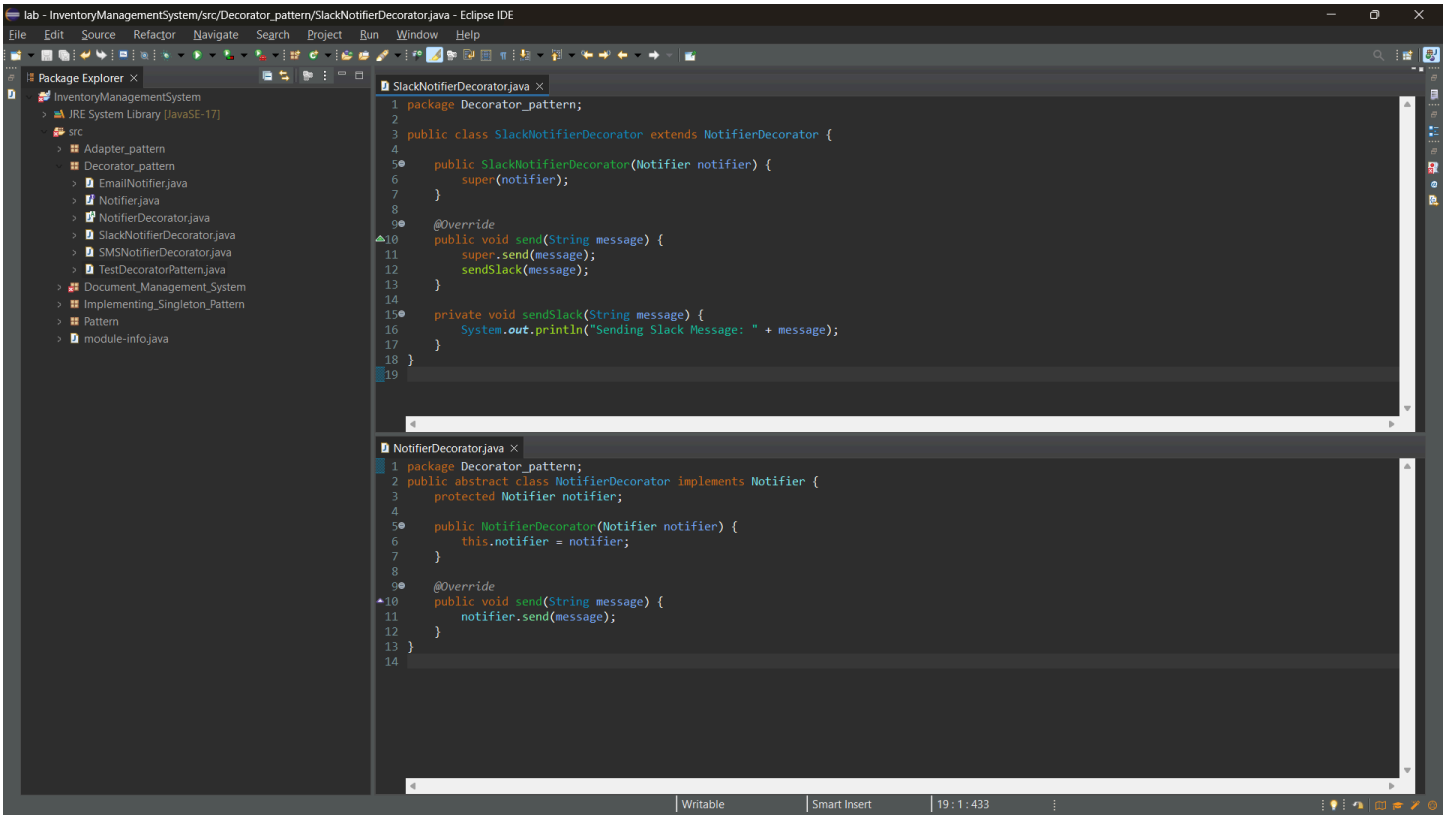
You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically.

Steps:

1. **Create a New Java Project:**
   - Create a new Java project named **DecoratorPatternExample**.
2. **Define Component Interface:**
   - Create an interface **Notifier** with a method **send()**.
3. **Implement Concrete Component:**
   - Create a class **EmailNotifier** that implements Notifier.
4. **Implement Decorator Classes:**
   - Create abstract decorator class **NotifierDecorator** that implements **Notifier** and holds a reference to a **Notifier** object.
   - Create concrete decorator classes like **SMSNotifierDecorator**, **SlackNotifierDecorator** that extend **NotifierDecorator**.
5. **Test the Decorator Implementation:**
   - Create a test class to demonstrate sending notifications via multiple channels using decorators.

Solution:

Code:



lab - InventoryManagementSystem/src/Decorator_pattern/TestDecoratorPattern.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer

- InventoryManagementSystem
  - JRE System Library [JavaSE-17]
  - src
    - Adapter_pattern
    - Decorator_pattern
      - EmailNotifier.java
      - Notifier.java
      - NotifierDecorator.java
      - SlackNotifierDecorator.java
      - SMSNotifierDecorator.java
      - TestDecoratorPattern.java
    - Document_Management_System
    - Implementing_Singleton_Pattern
    - Pattern
    - module-info.java

SMSNotifierDecorator.java | TestDecoratorPattern.java

```java
package Decorator_pattern;

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

        // Base notification via Email
        Notifier emailNotifier = new EmailNotifier();

        // Adding SMS notification
        Notifier smsNotifier = new SMSNotifierDecorator(emailNotifier);

        // Adding Slack notification
        Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);

        // Send notification through all channels
        slackNotifier.send("Your package has been shipped.");
    }
}
```

Problems | Javadoc | Declaration | Console

```
<terminated> TestDecoratorPattern [Java Application] C:\eclipse\plugins\org.eclipse.justj.
Sending Email: Your package has been shipped.
Sending SMS: Your package has been shipped.
Sending Slack Message: Your package has been shipped.
```

lab - InventoryManagementSystem/src/Decorator_pattern/EmailNotifier.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer

- InventoryManagementSystem
  - JRE System Library [JavaSE-17]
  - src
    - Adapter_pattern
    - Decorator_pattern
      - EmailNotifier.java
      - Notifier.java
      - NotifierDecorator.java
      - SlackNotifierDecorator.java
      - SMSNotifierDecorator.java
      - TestDecoratorPattern.java
    - Document_Management_System
    - Implementing_Singleton_Pattern
    - Pattern
    - module-info.java

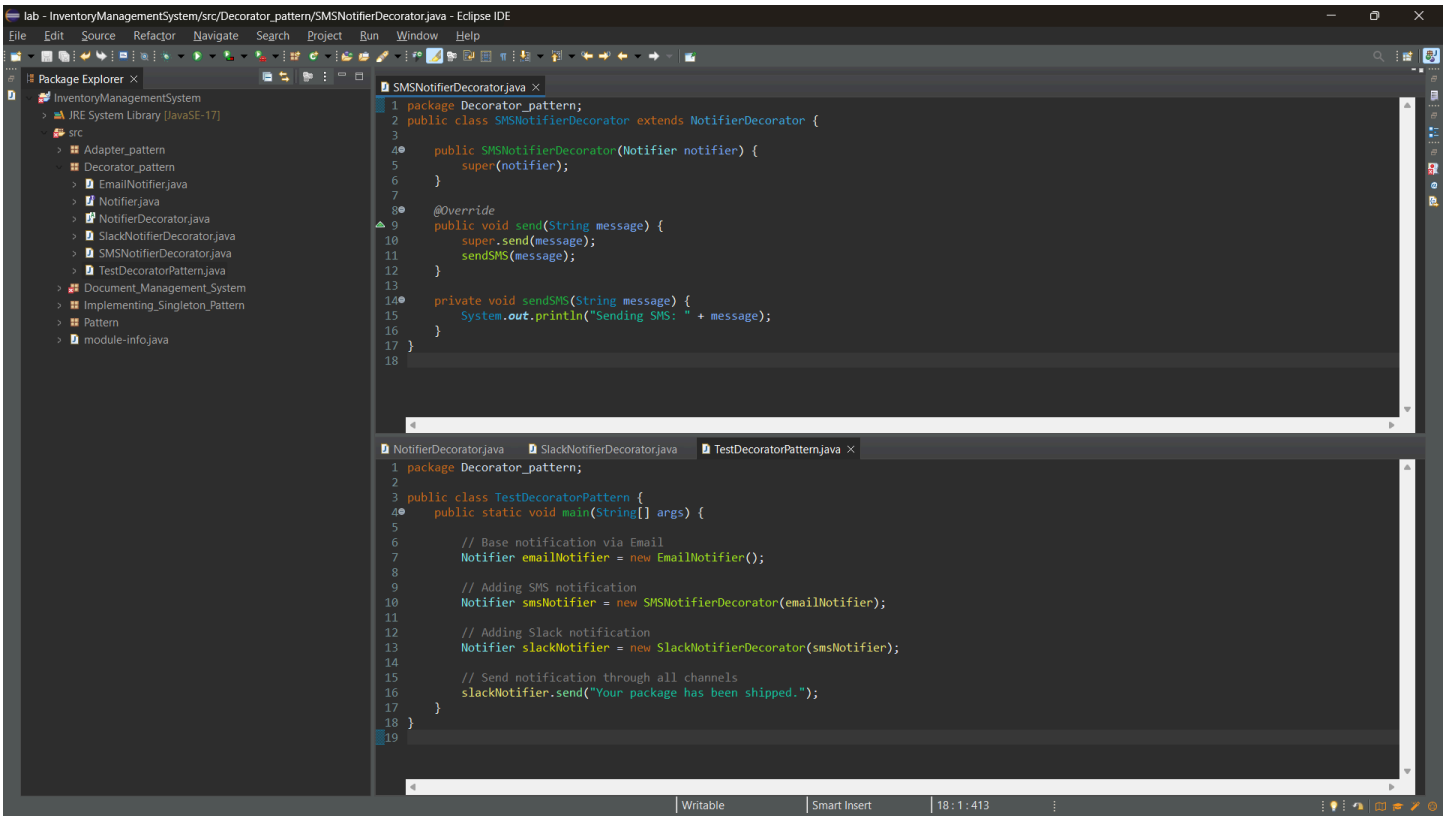
EmailNotifier.java

```java
package Decorator_pattern;
public class EmailNotifier implements Notifier {
    @Override
    public void send(String message) {
        System.out.println("Sending Email: " + message);
    }
}
```

Notifier.java | NotifierDecorator.java | SMSNotifierDecorator.java | SlackNotifierDecorator.java | TestDecoratorPattern.java

```java
package Decorator_pattern;

public interface Notifier {
    void send(String message);
}
```

Writable | Smart Insert | 8 : 1 : 201



Exercise 6: Implementing the Proxy Pattern

Scenario:

You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching.

Steps:

1. **Create a New Java Project:**
   - Create a new Java project named **ProxyPatternExample**.
2. **Define Subject Interface:**
   - Create an interface Image with a method **display()**.
3. **Implement Real Subject Class:**
   - Create a class **RealImage** that implements Image and loads an image from a remote server.
4. **Implement Proxy Class:**
   - Create a class **ProxyImage** that implements Image and holds a reference to RealImage.
   - Implement lazy initialization and caching in **ProxyImage**.
5. **Test the Proxy Implementation:**
   - Create a test class to demonstrate the use of **ProxyImage** to load and display images.

Solution:

Code:

lab - InventoryManagementSystem/src/Proxy_pattern/RealImage.java - Eclipse IDE

```java
public class RealImage implements Image {

    private String fileName;

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

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

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

```
<terminated> TestProxyPattern [Java Application] C:\eclipse\plugins\org.eclipse.justj.open
Loading Photo1.jpg from remote server...
Displaying Photo1.jpg

Displaying Photo1.jpg

Loading Photo2.jpg from remote server...
Displaying Photo2.jpg
```

```java
package Proxy_pattern;

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

        Image image1 = new ProxyImage("Photo1.jpg");
        Image image2 = new ProxyImage("Photo2.jpg");

        // Image will be loaded only when displayed
        image1.display(); // Loads from server
        System.out.println();
        image1.display(); // Cached, no loading
        System.out.println();
        image2.display(); // Loads from server
    }
}
```

lab - InventoryManagementSystem/src/Proxy_pattern/Image.java - Eclipse IDE

```java
package Proxy_pattern;

public class ProxyImage implements Image {

    private String fileName;
    private RealImage realImage;

    public ProxyImage(String fileName) {
        this.fileName = fileName;
    }

    @Override
    public void display() {
        if (realImage == null) {
            realImage = new RealImage(fileName); // Lazy initialization
        }
        realImage.display();
    }
}
```

```java
package Proxy_pattern;
public interface Image {
    void display();
}
```

Exercise 7: Implementing the Observer Pattern

Scenario:

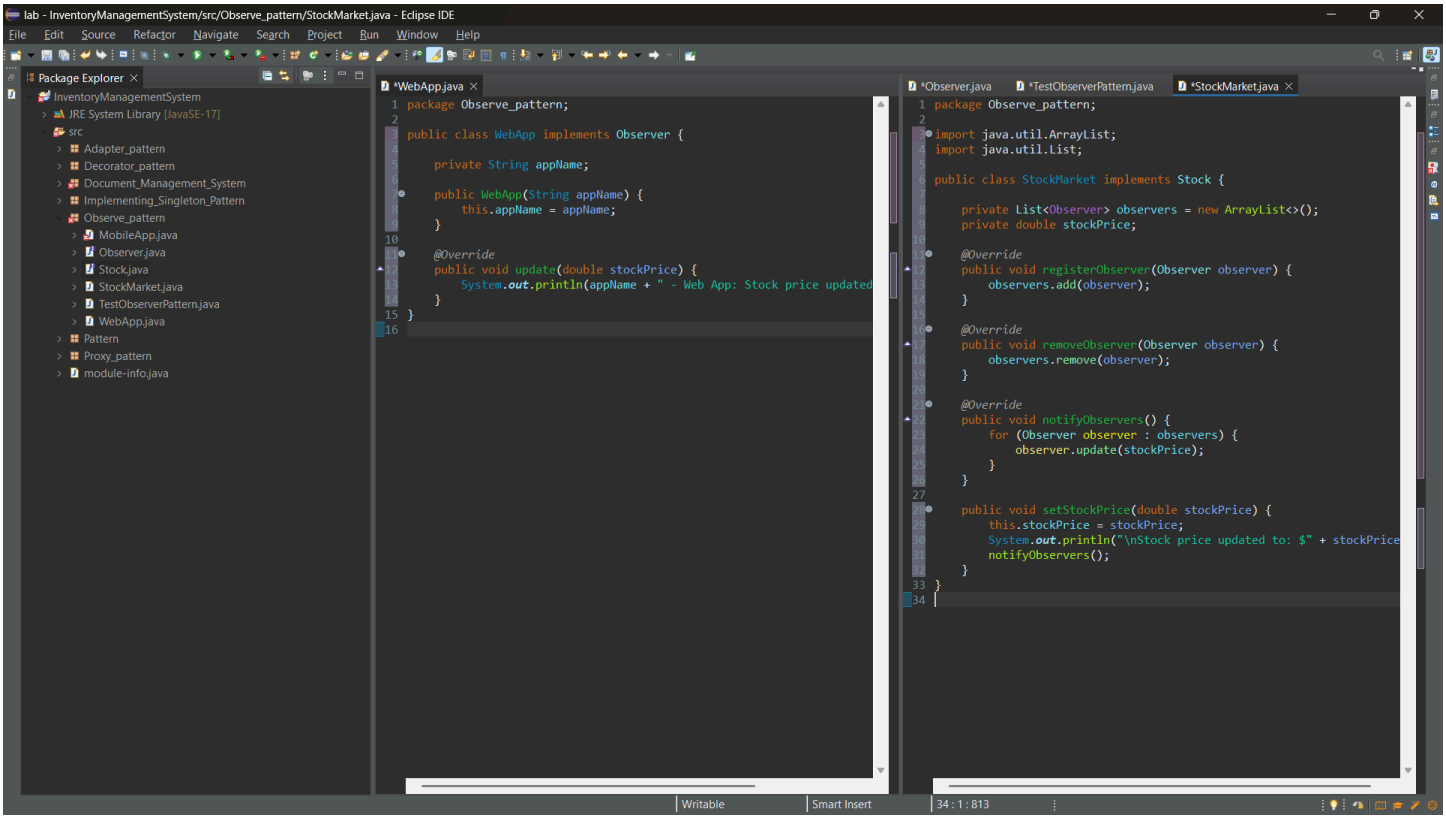
You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

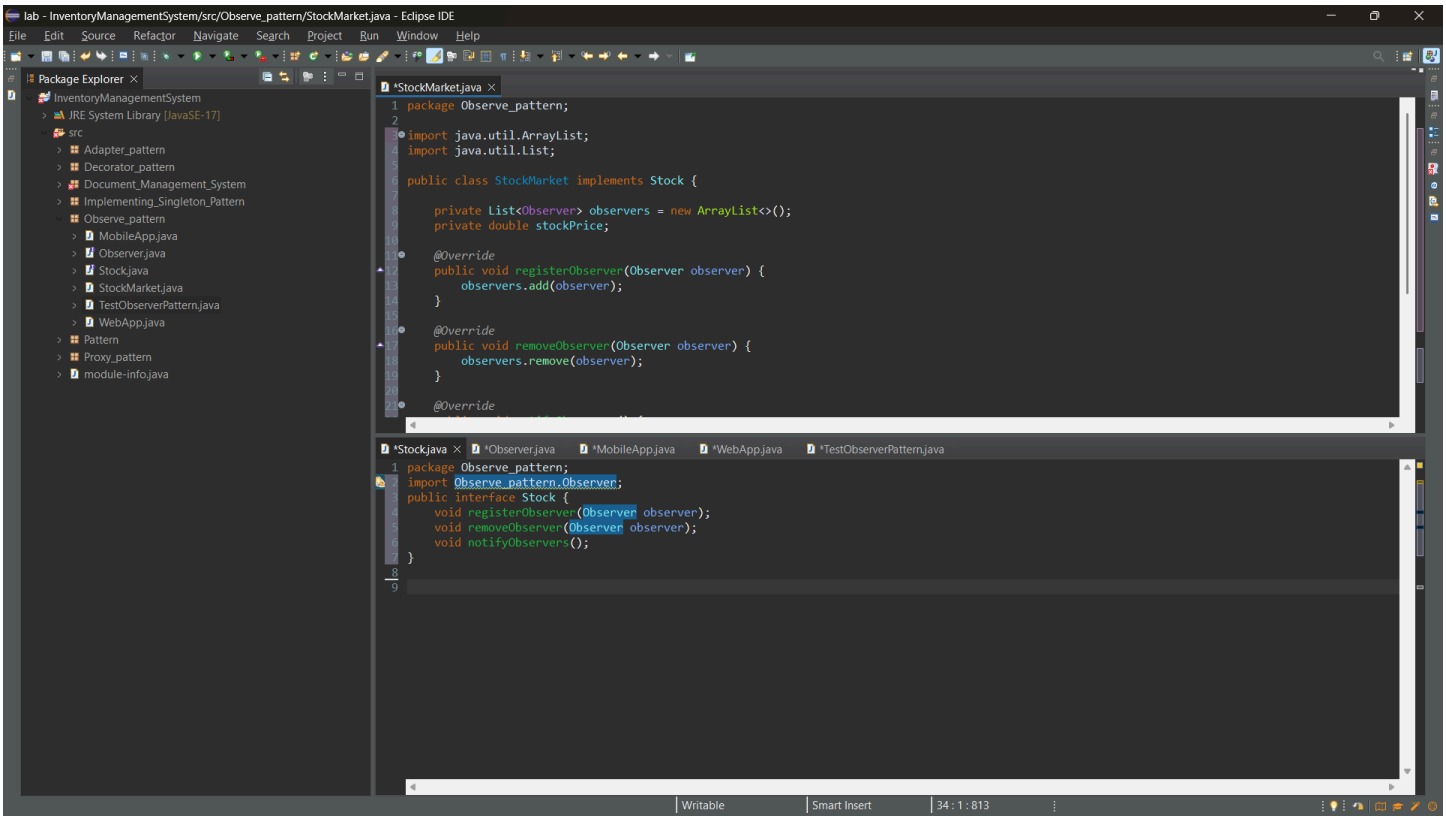
Steps:

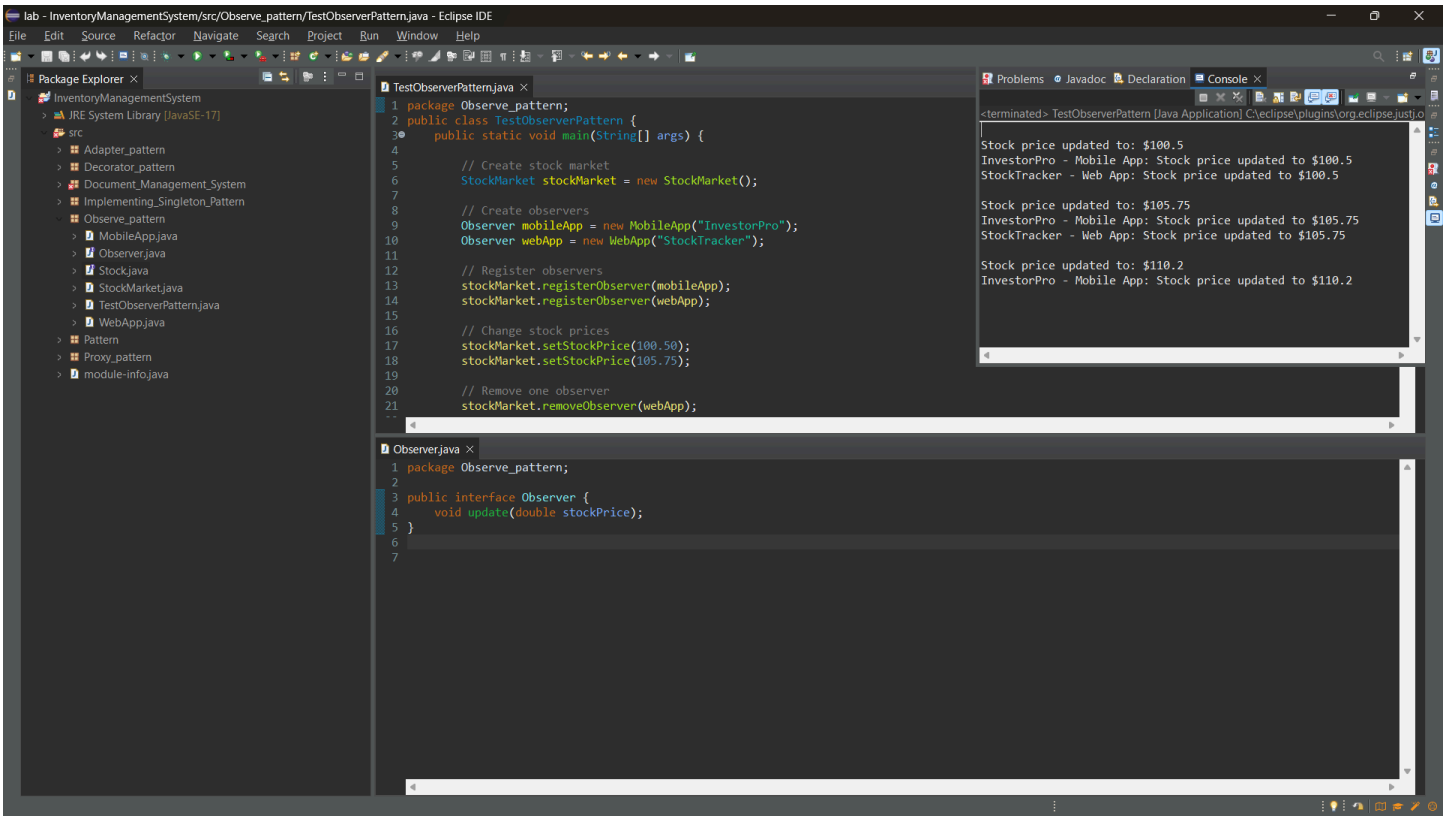
1. **Create a New Java Project:**
   - Create a new Java project named **ObserverPatternExample**.
2. **Define Subject Interface:**
   - Create an interface **Stock** with methods to **register**, **deregister**, and **notify** observers.
3. **Implement Concrete Subject:**
   - Create a class **StockMarket** that implements **Stock** and maintains a list of observers.
4. **Define Observer Interface:**
   - Create an interface Observer with a method **update().**
5. **Implement Concrete Observers:**
   - Create classes **MobileApp**, **WebApp** that implement Observer.
6. **Test the Observer Implementation:**
   - Create a test class to demonstrate the registration and notification of observers.

Solution:

Code







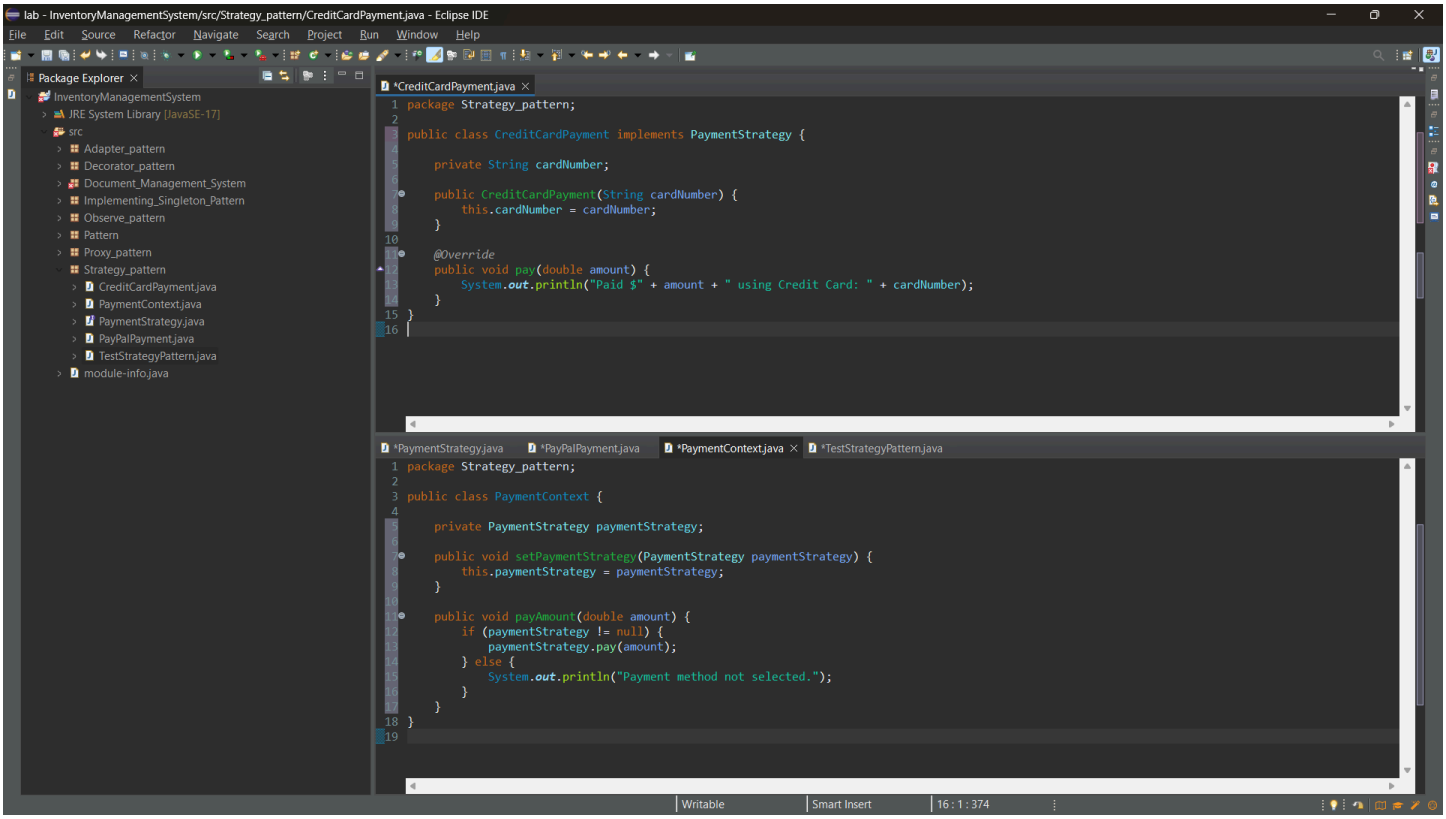
Exercise 8: Implementing the Strategy Pattern

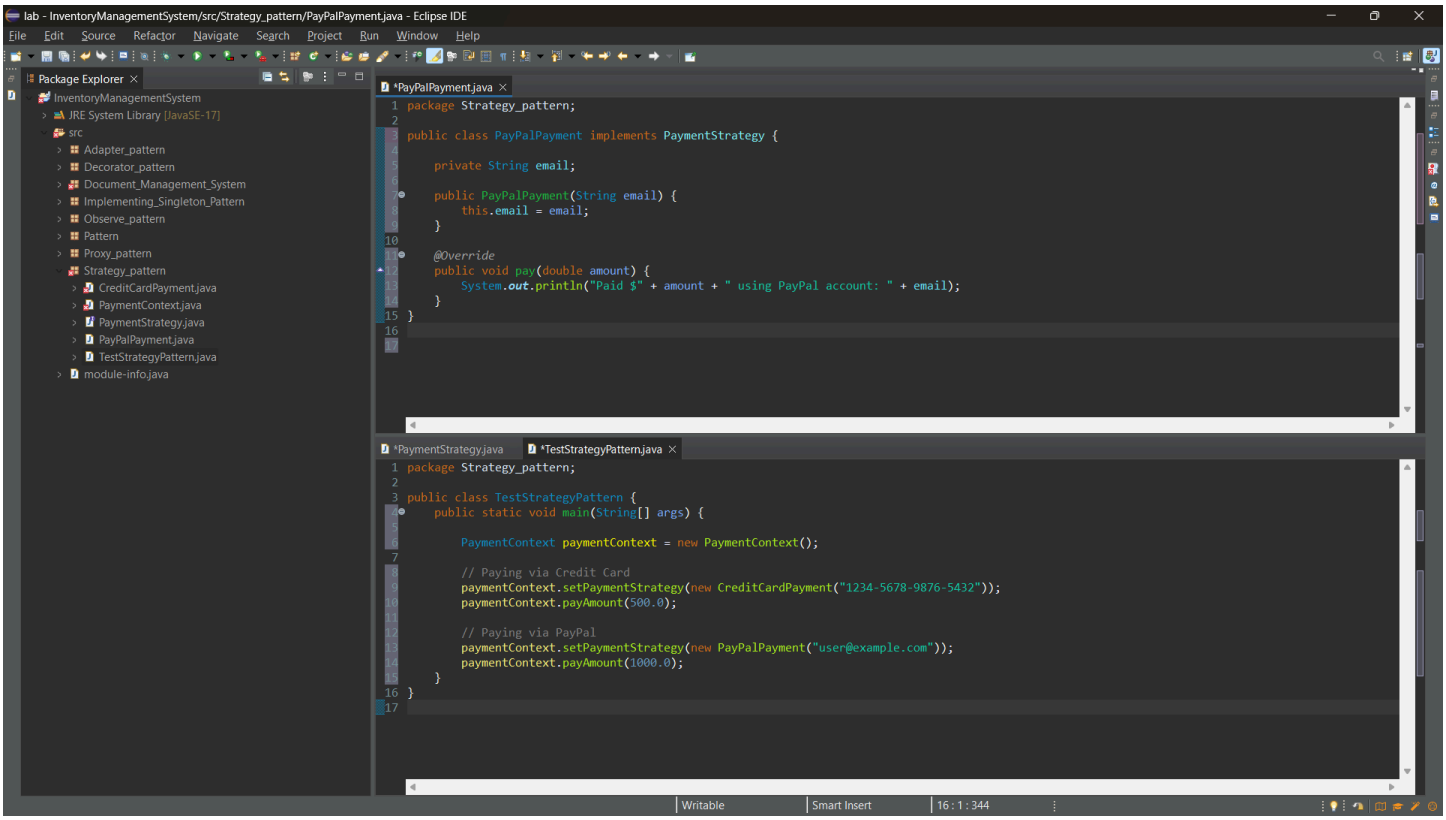
Scenario:

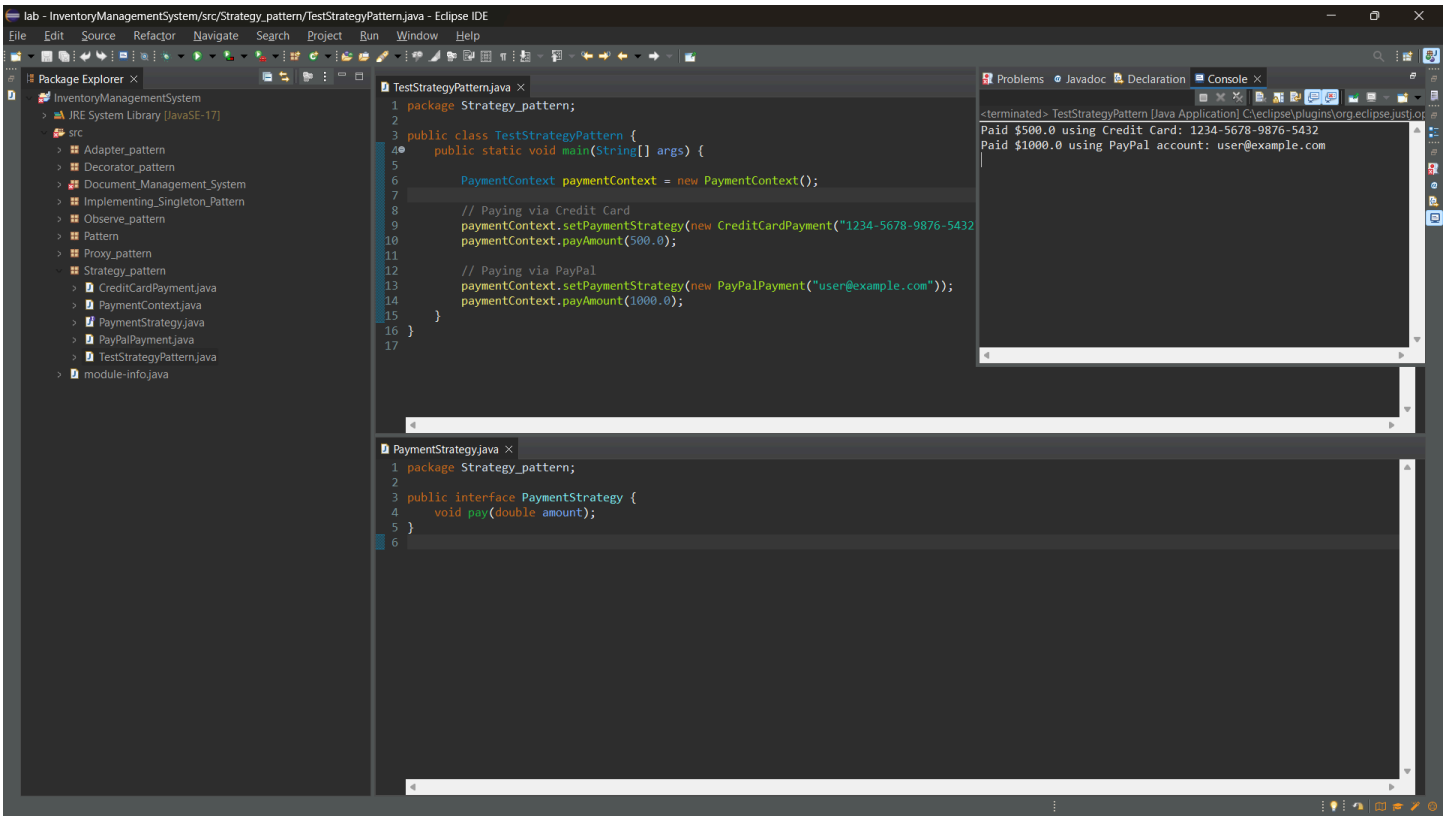
You are developing a payment system where different payment methods (e.g., Credit Card, PayPal) can be selected at runtime. Use the Strategy Pattern to achieve this.

Steps:

1. **Create a New Java Project:**
   - Create a new Java project named **StrategyPatternExample**.
2. **Define Strategy Interface:**
   - Create an interface PaymentStrategy with a method **pay()**.
3. **Implement Concrete Strategies:**
   - Create classes **CreditCardPayment**, **PayPalPayment** that implement **PaymentStrategy**.
4. **Implement Context Class:**
   - Create a class **PaymentContext** that holds a reference to **PaymentStrategy** and a method to execute the strategy.
5. **Test the Strategy Implementation:**
   - Create a test class to demonstrate selecting and using different payment strategies.





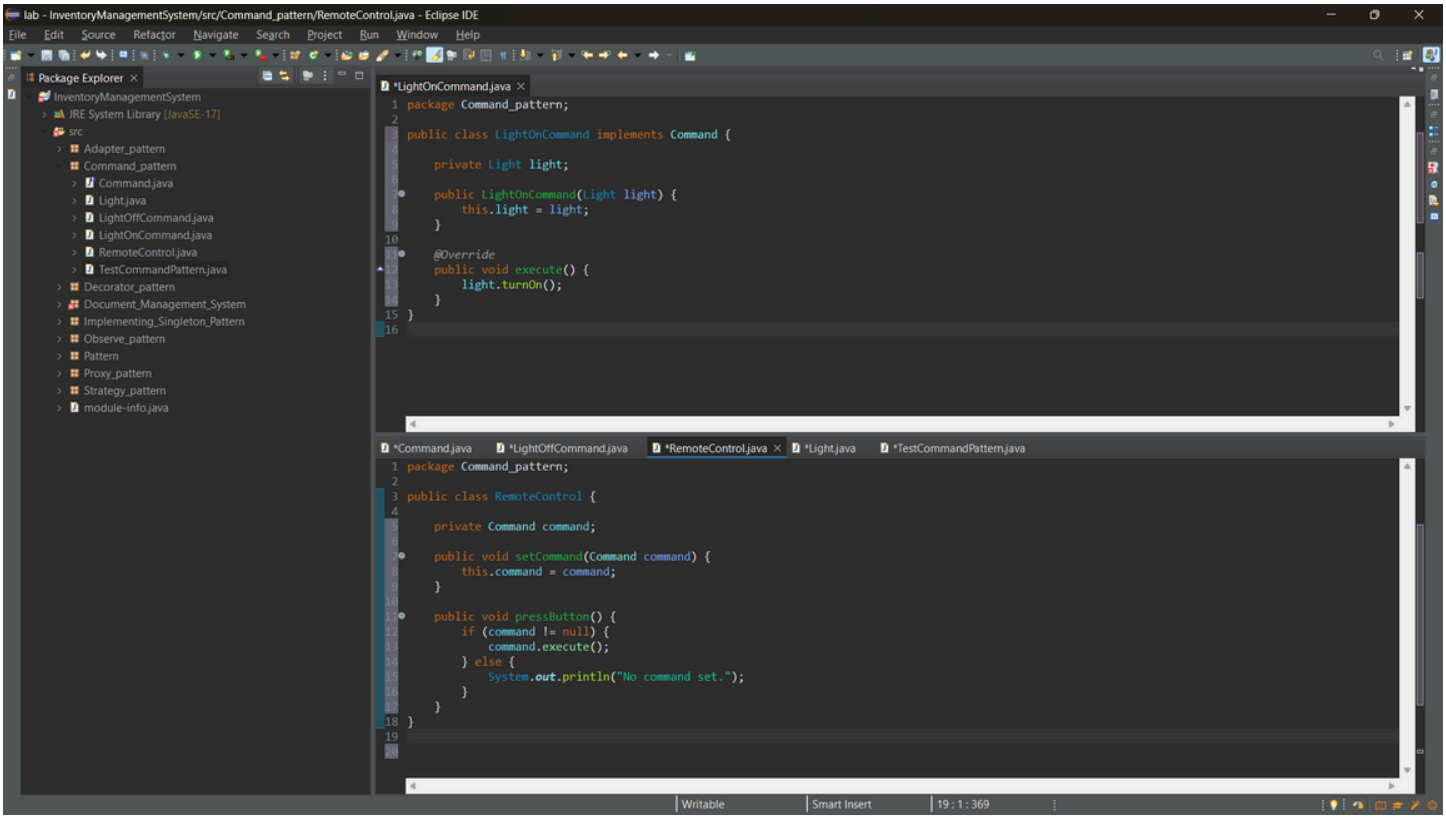


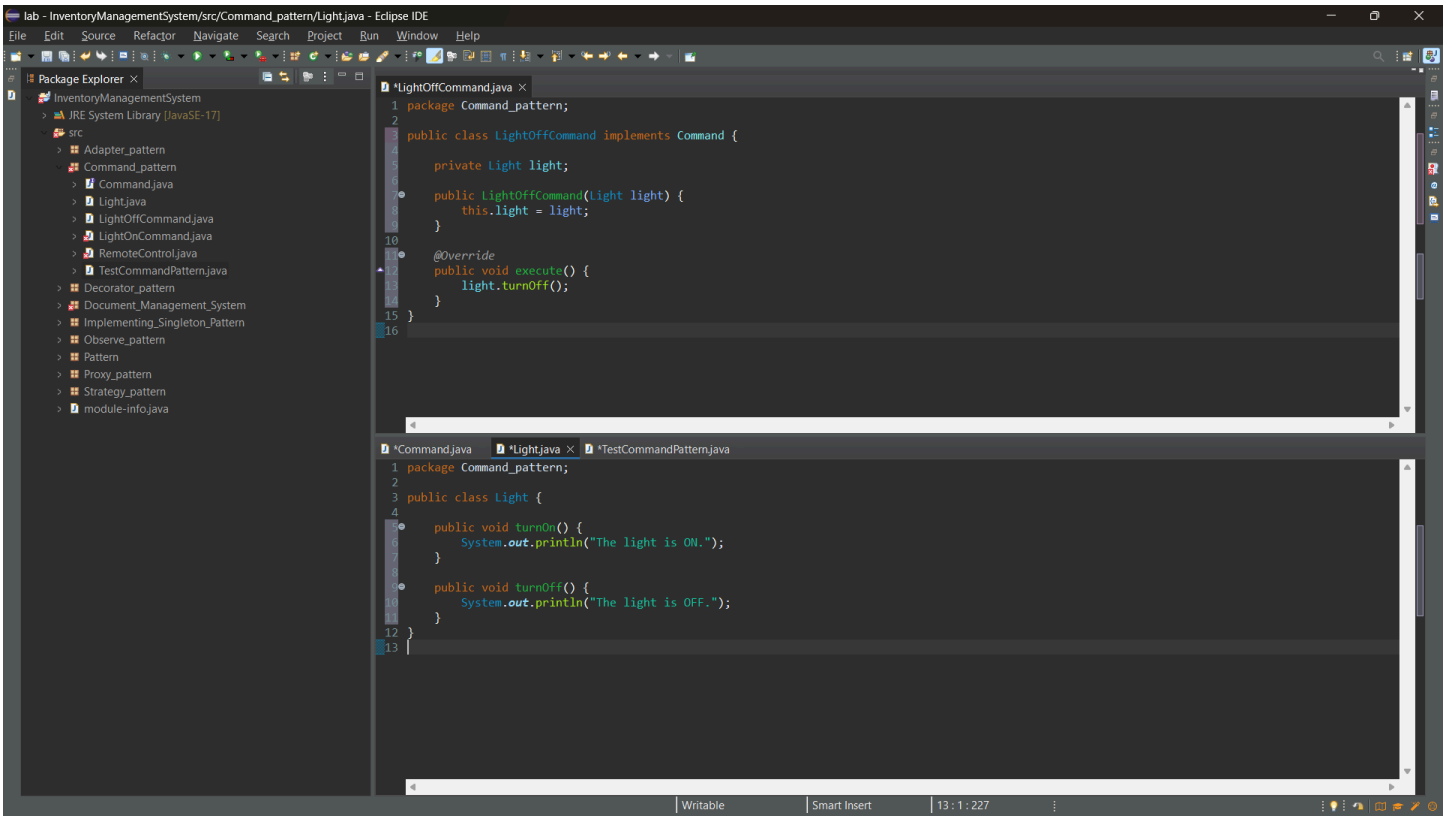
Exercise 9: Implementing the Command Pattern

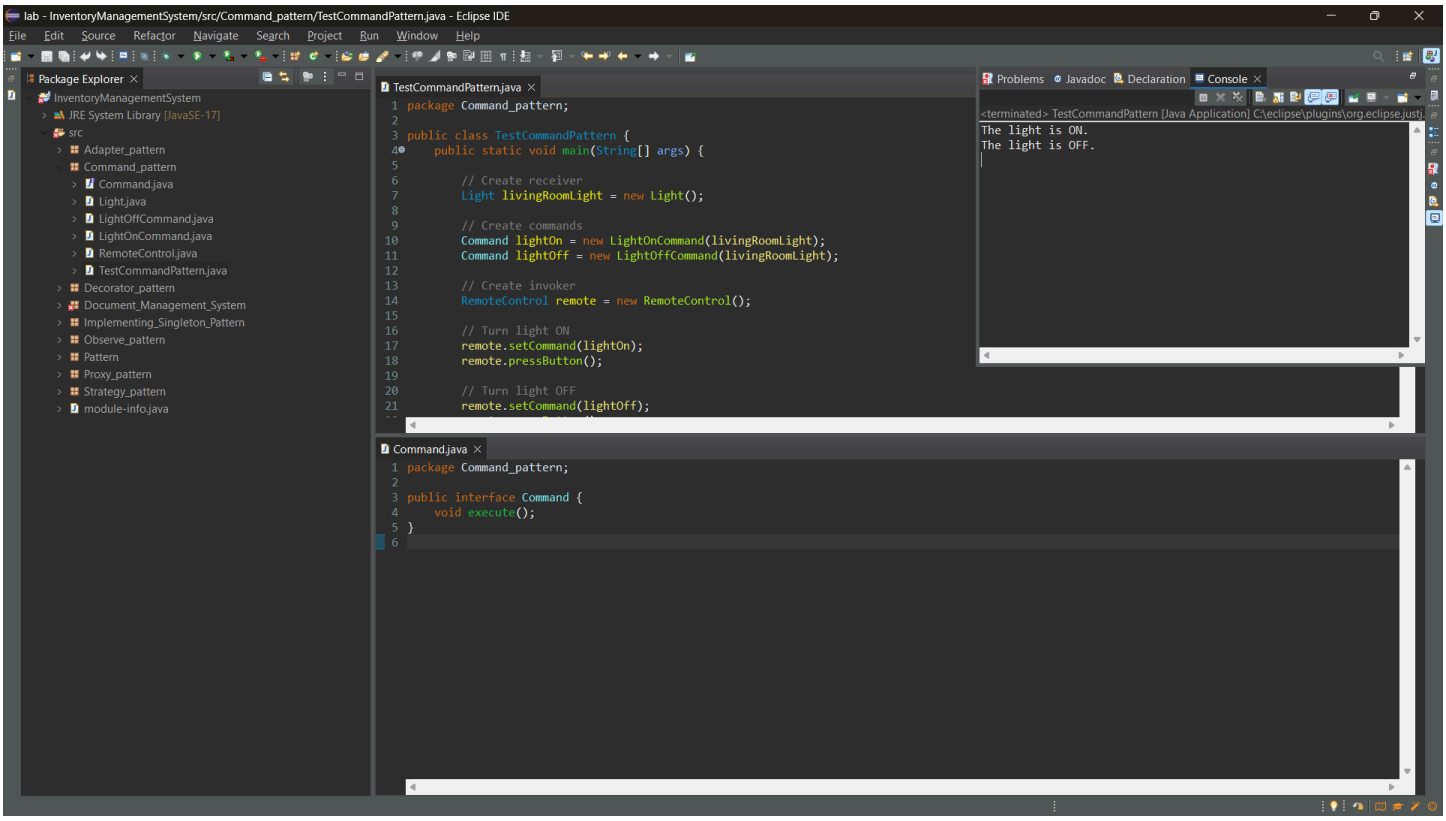
Scenario: You are developing a home automation system where commands can be issued to turn devices on or off. Use the Command Pattern to achieve this.

Steps:

1. **Create a New Java Project:**
   - Create a new Java project named **CommandPatternExample**.
2. **Define Command Interface:**
   - Create an interface Command with a method **execute()**.
3. **Implement Concrete Commands:**
   - Create classes **LightOnCommand**, **LightOffCommand** that implement Command.
4. **Implement Invoker Class:**
   - Create a class **RemoteControl** that holds a reference to a Command and a method to execute the command.
5. **Implement Receiver Class:**
   - Create a class **Light** with methods to turn on and off.
6. **Test the Command Implementation:**
   - Create a test class to demonstrate issuing commands using the **RemoteControl**.







Data Structures and Algorithms

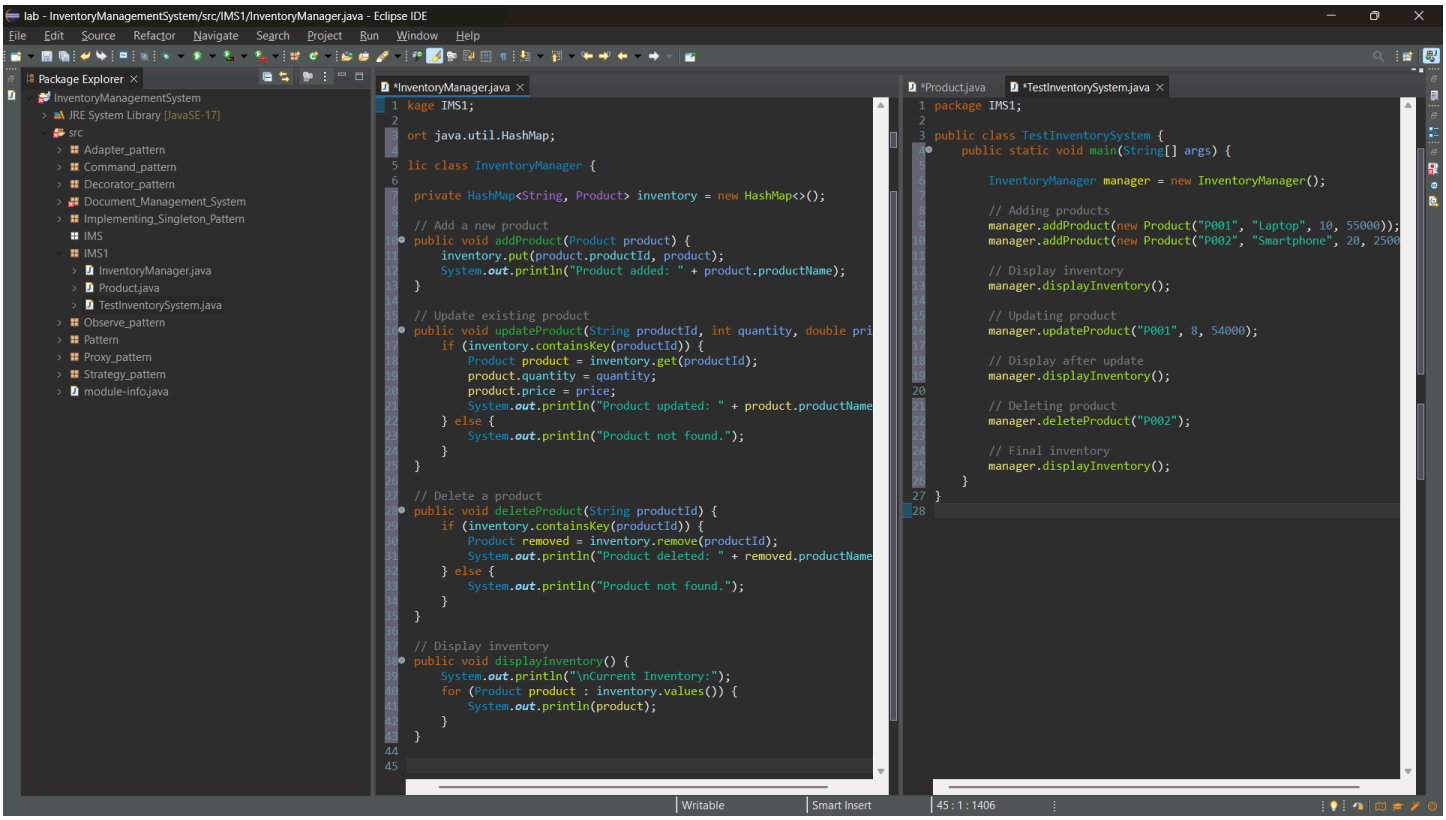
Exercise 1: Inventory Management System

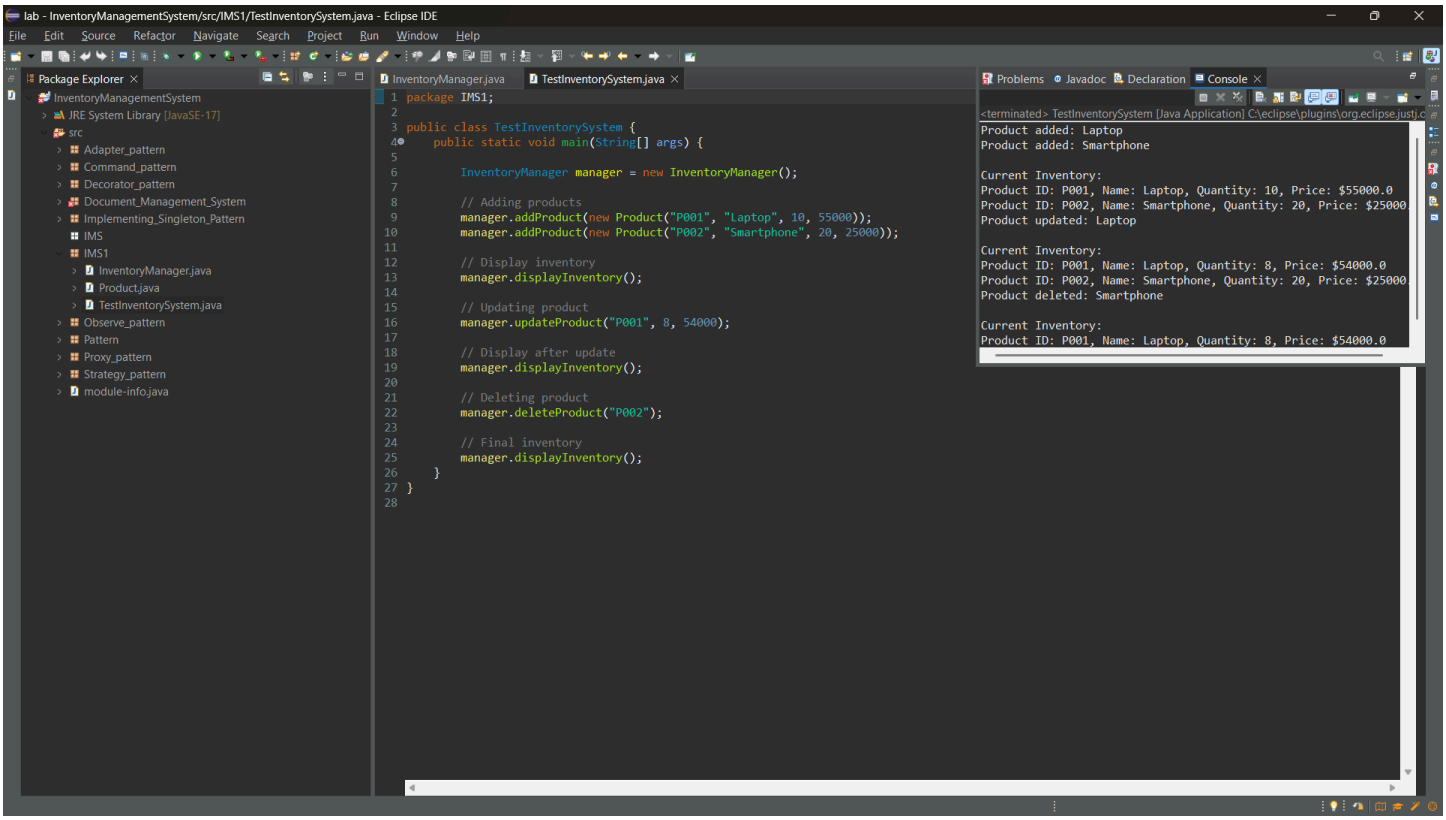
Scenario:

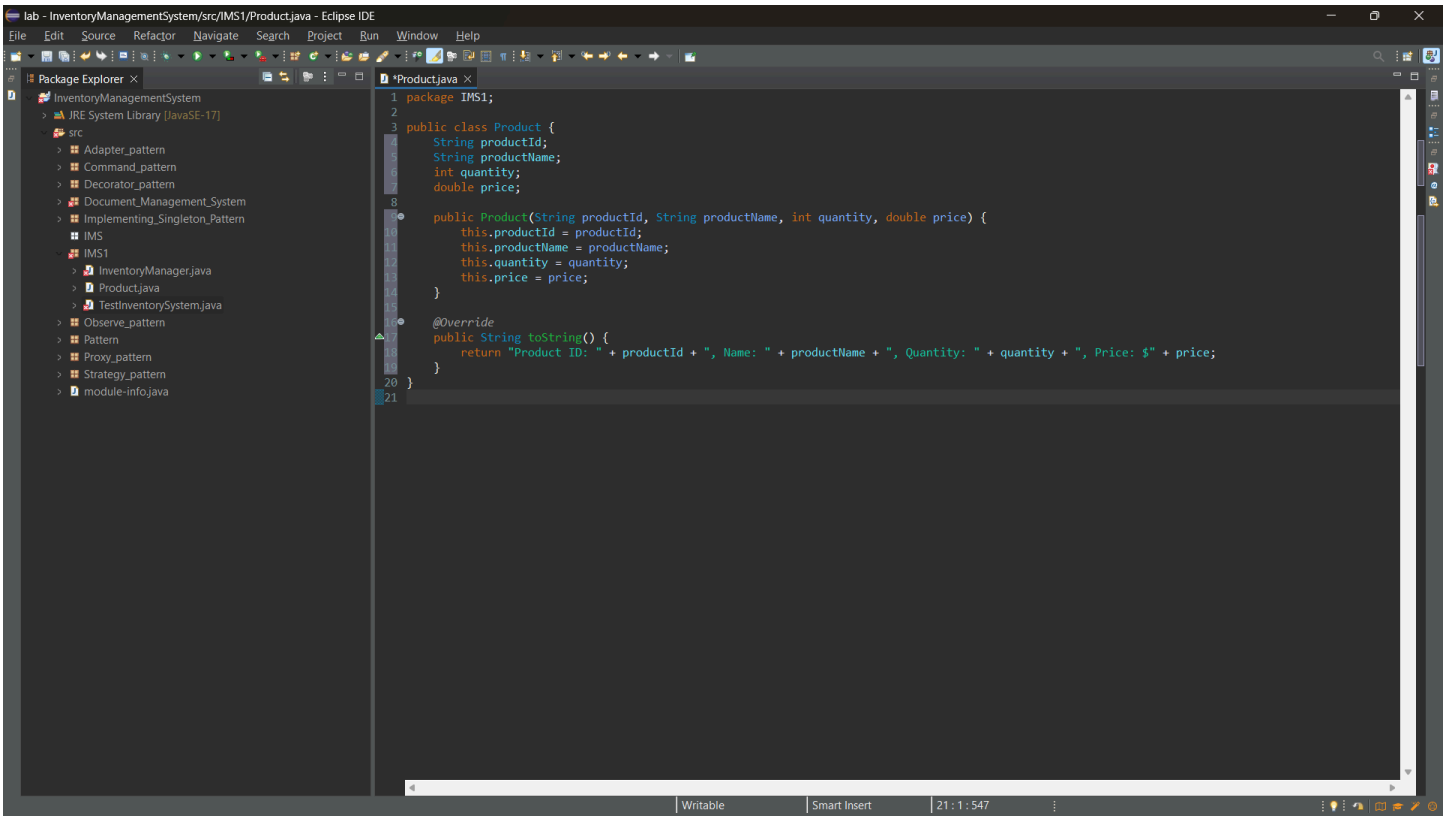
You are developing an inventory management system for a warehouse. Efficient data storage and retrieval are crucial.

Steps:

1. **Understand the Problem:**
   - Explain why data structures and algorithms are essential in handling large inventories.
   - Discuss the types of data structures suitable for this problem.
2. **Setup:**
   - Create a new project for the inventory management system.
3. **Implementation:**
   - Define a class Product with attributes like **productId**, **productName**, **quantity**, and **price**.
   - Choose an appropriate data structure to store the products (e.g., ArrayList, HashMap).
   - Implement methods to add, update, and delete products from the inventory.
4. **Analysis:**
   - Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.
   - Discuss how you can optimize these operations.







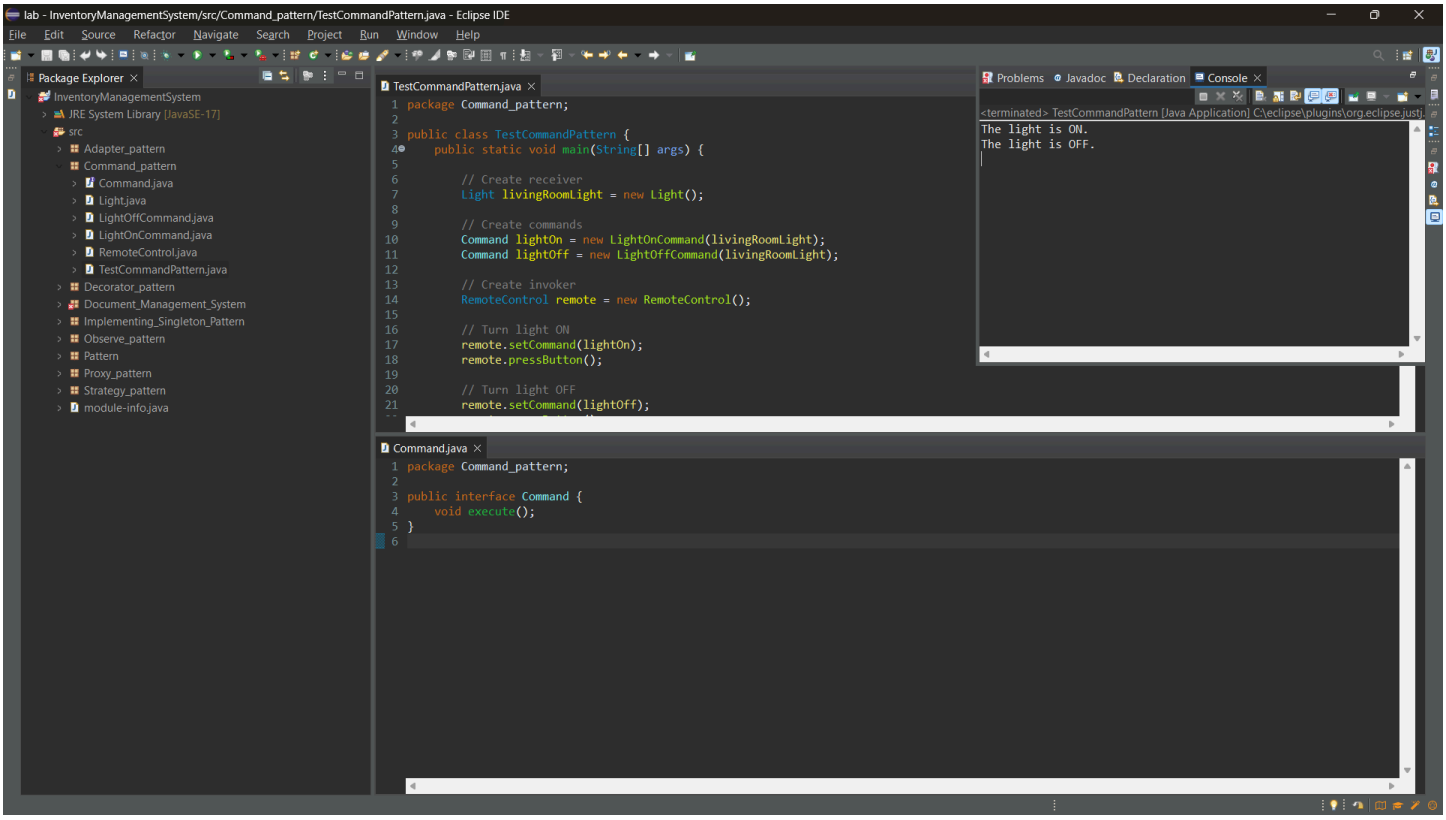
Exercise 2: E-commerce Platform Search Function

Scenario:

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

Steps:

1. **Understand Asymptotic Notation:**
   - Explain Big O notation and how it helps in analyzing algorithms.
   - Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   - Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   - Implement linear search and binary search algorithms.
   - Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   - Compare the time complexity of linear and binary search algorithms.
   - Discuss which algorithm is more suitable for your platform and why.



lab - InventoryManagementSystem/src/Strategy_pattern/CreditCardPayment.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer

- InventoryManagementSystem
  - JRE System Library [JavaSE-17]
  - src
    - Adapter_pattern
    - Decorator_pattern
    - Document_Management_System
    - Implementing_Singleton_Pattern
    - Observe_pattern
    - Pattern
    - Proxy_pattern
    - Strategy_pattern
      - CreditCardPayment.java
      - PaymentContext.java
      - PaymentStrategy.java
      - PayPalPayment.java
      - TestStrategyPattern.java
    - module-info.java

*CreditCardPayment.java

```java
package Strategy_pattern;

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: " + cardNumber);
    }
}
```

*PaymentStrategy.java | *PayPalPayment.java | *PaymentContext.java | *TestStrategyPattern.java

```java
package Strategy_pattern;

public class PaymentContext {

    private PaymentStrategy paymentStrategy;

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

    public void payAmount(double amount) {
        if (paymentStrategy != null) {
            paymentStrategy.pay(amount);
        } else {
            System.out.println("Payment method not selected.");
        }
    }
}
```

Writable | Smart Insert | 16 : 1 : 374

lab - InventoryManagementSystem/src/Strategy_pattern/PayPalPayment.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer

- InventoryManagementSystem
  - JRE System Library [JavaSE-17]
  - src
    - Adapter_pattern
    - Decorator_pattern
    - Document_Management_System
    - Implementing_Singleton_Pattern
    - Observe_pattern
    - Pattern
    - Proxy_pattern
    - Strategy_pattern
      - CreditCardPayment.java
      - PaymentContext.java
      - PaymentStrategy.java
      - PayPalPayment.java
      - TestStrategyPattern.java
    - module-info.java

*PayPalPayment.java

```java
package Strategy_pattern;

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);
    }
}
```

*PaymentStrategy.java | *TestStrategyPattern.java

```java
package Strategy_pattern;

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

        PaymentContext paymentContext = new PaymentContext();

        // Paying via Credit Card
        paymentContext.setPaymentStrategy(new CreditCardPayment("1234-5678-9876-5432"));
        paymentContext.payAmount(500.0);

        // Paying via PayPal
        paymentContext.setPaymentStrategy(new PayPalPayment("user@example.com"));
        paymentContext.payAmount(1000.0);
    }
}
```

Writable | Smart Insert | 16 : 1 : 344