

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:

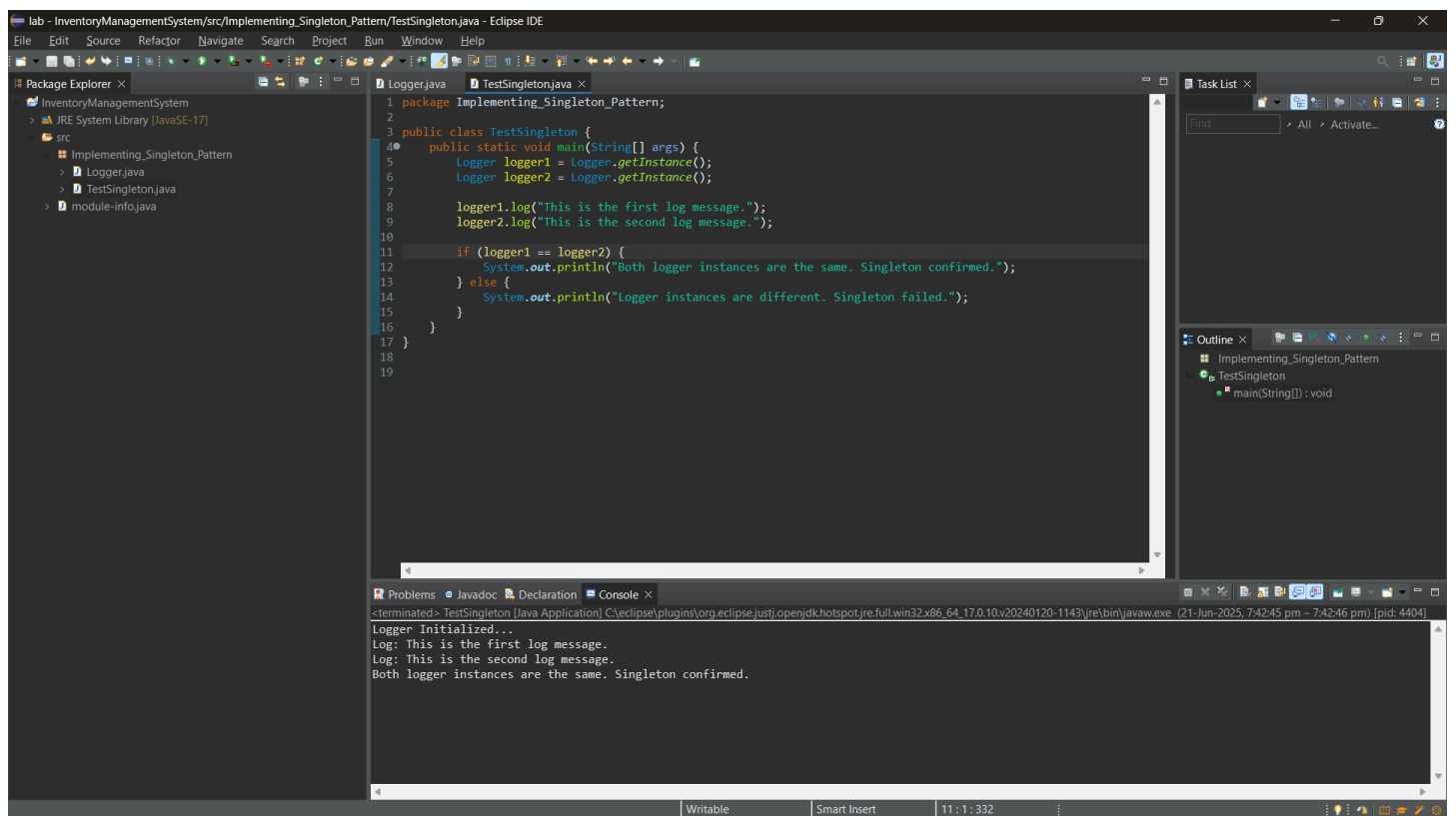
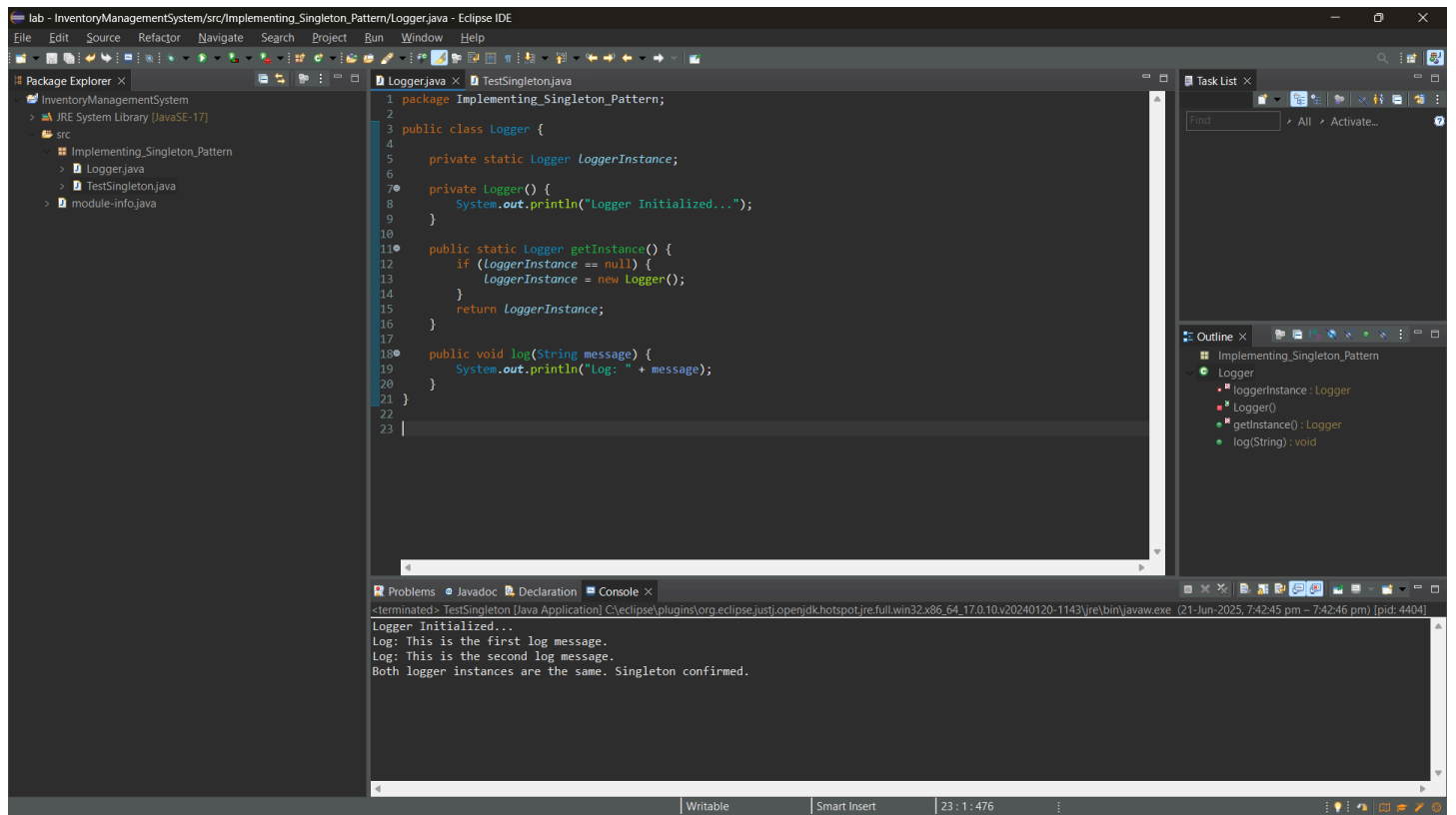
- Write code to ensure that the `Logger` class follows the Singleton design pattern.

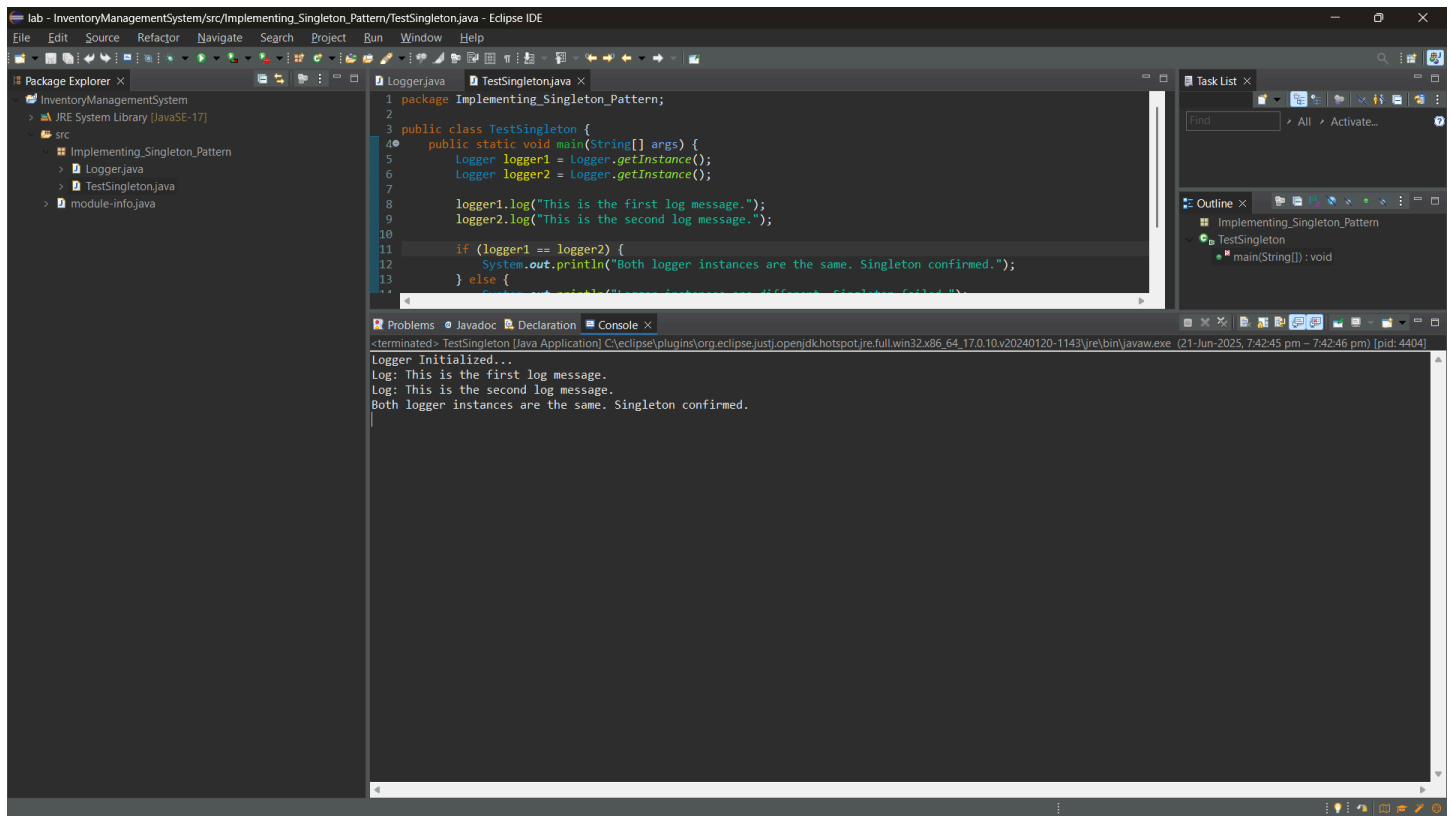
4. Test the Singleton Implementation:

- Create a test class to verify that only one instance of `Logger` is created and used across the application.

Solution:

Code:





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.

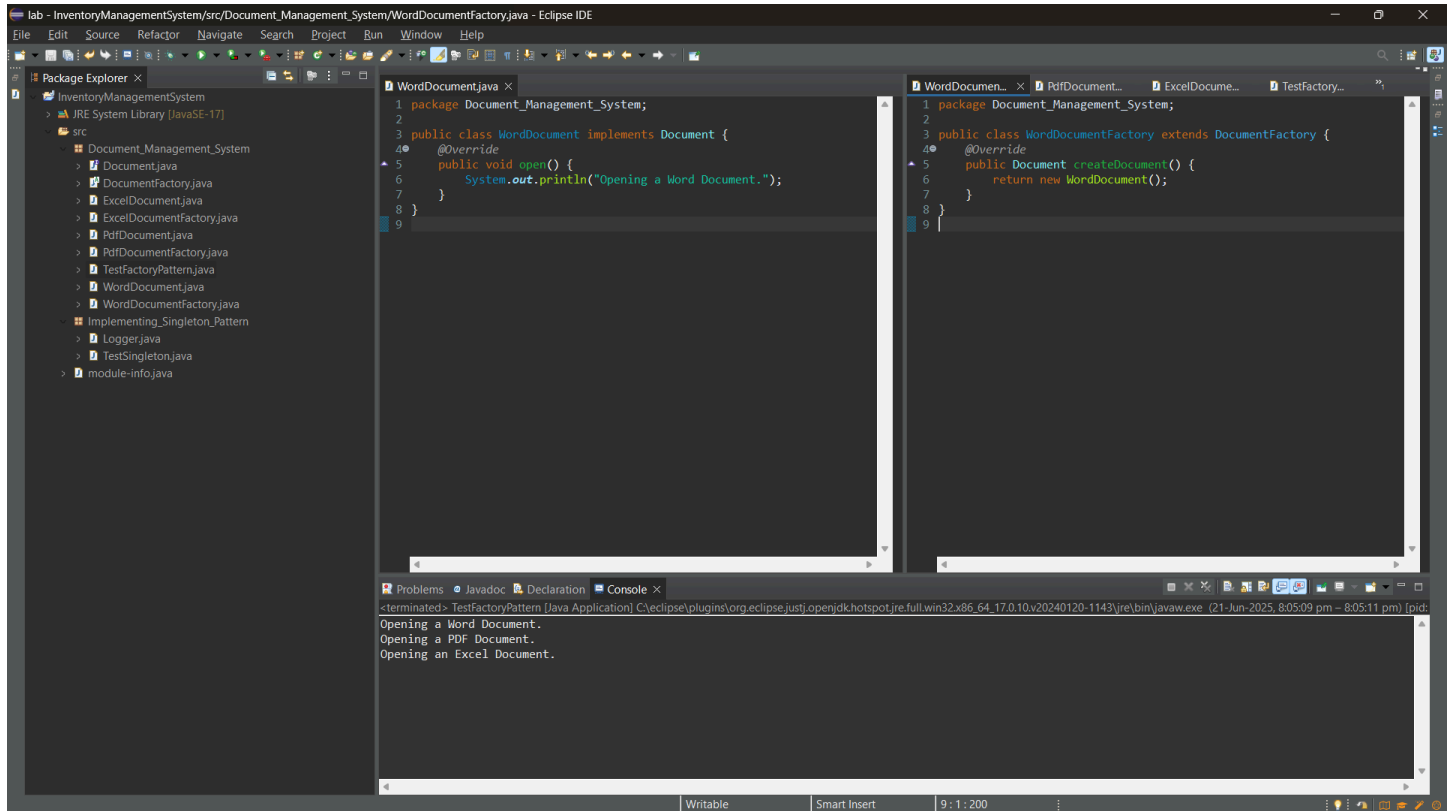
Steps:

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:



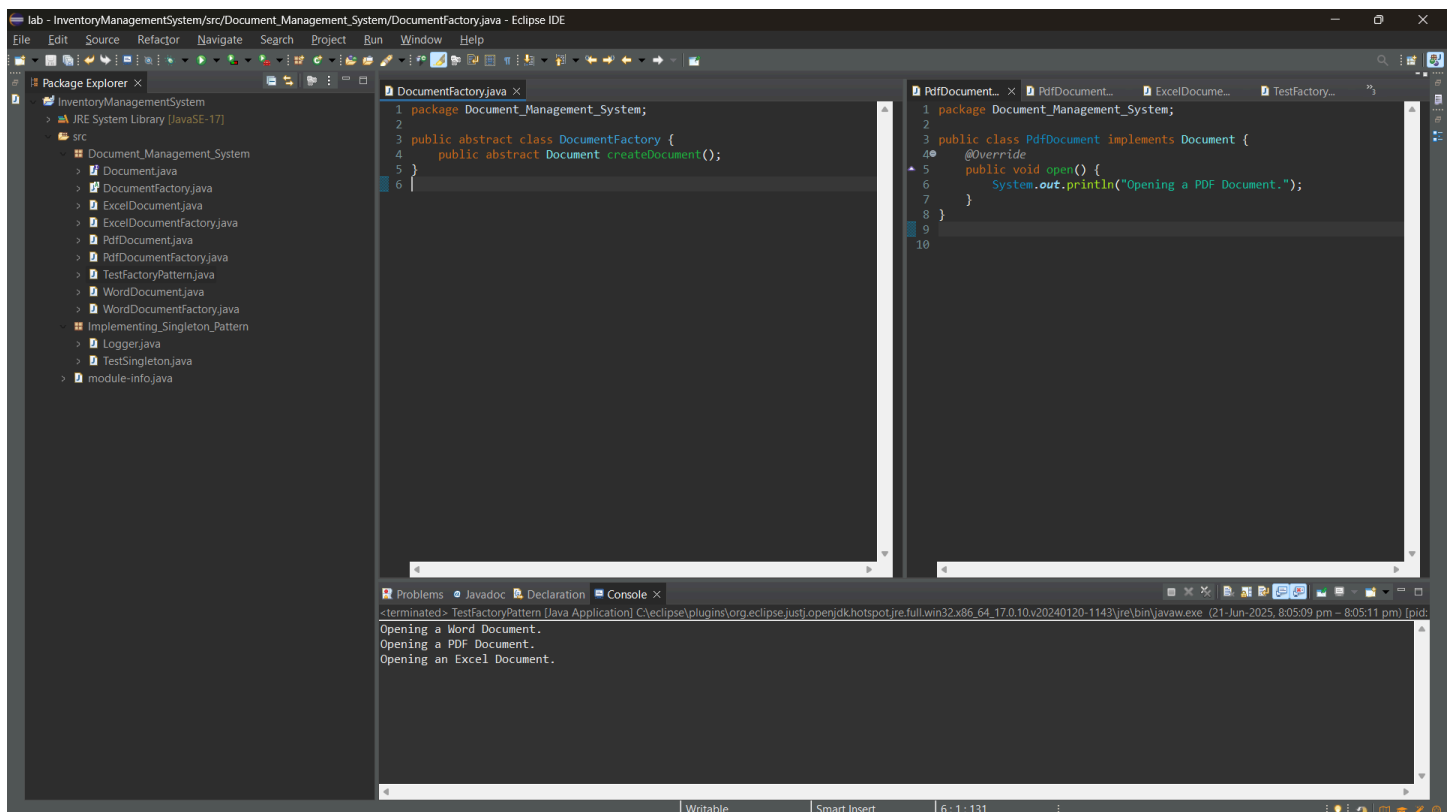
The screenshot shows the Eclipse IDE with the following code:

```
WordDocumentFactory.java
1 package Document_Management_System;
2
3 public class WordDocumentFactory extends DocumentFactory {
4     @Override
5     public Document createDocument() {
6         return new WordDocument();
7     }
8 }
9
```

```
WordDocument.java
1 package Document_Management_System;
2
3 public class WordDocument implements Document {
4     @Override
5     public void open() {
6         System.out.println("Opening a Word Document.");
7     }
8 }
9
```

The console output shows:

```
<terminated> TestFactoryPattern [Java Application] C:\eclipse\plugins\org.eclipse.justi.openjdk hotspot\jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe (21-Jun-2025, 8:05:09 pm - 8:05:11 pm) [pid:
Opening a Word Document.
Opening a PDF Document.
Opening an Excel Document.
```



The screenshot shows the Eclipse IDE with the following code:

```
DocumentFactory.java
1 package Document_Management_System;
2
3 public abstract class DocumentFactory {
4     public abstract Document createDocument();
5 }
6
```

```
PdfDocument.java
1 package Document_Management_System;
2
3 public class PdfDocument implements Document {
4     @Override
5     public void open() {
6         System.out.println("Opening a PDF Document.");
7     }
8 }
9
10
```

The console output shows:

```
<terminated> TestFactoryPattern [Java Application] C:\eclipse\plugins\org.eclipse.justi.openjdk hotspot\jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe (21-Jun-2025, 8:05:09 pm - 8:05:11 pm) [pid:
Opening a Word Document.
Opening a PDF Document.
Opening an Excel Document.
```

lab - InventoryManagementSystem/src/Document_Management_System/TestFactoryPattern.java - Eclipse IDE

Package Explorer: InventoryManagementSystem > JRE System Library [JavaSE-17] > src > Document_Management_System > DocumentFactory.java, ExcelDocumentFactory.java, PdfDocumentFactory.java, WordDocumentFactory.java, Implementing_Singleton_Pattern, Logger.java, TestSingleton.java, module-info.java

TestFactoryPattern.java

```
1 package Document_Management_System;
2
3 public class TestFactoryPattern {
4     public static void main(String[] args) {
5
6         DocumentFactory wordFactory = new WordDocumentFactory();
7         Document wordDoc = wordFactory.createDocument();
8         wordDoc.open();
9
10        DocumentFactory pdfFactory = new PdfDocumentFactory();
11        Document pdfDoc = pdfFactory.createDocument();
12        pdfDoc.open();
13
14        DocumentFactory excelFactory = new ExcelDocumentFactory();
15        Document excelDoc = excelFactory.createDocument();
16        excelDoc.open();
17    }
18 }
19
```

ExcelDocumentFactory.java

```
1 package Document_Management_System;
2
3 public class ExcelDocumentFactory extends DocumentFactory {
4     @Override
5     public Document createDocument() {
6         return new ExcelDocument();
7     }
8 }
9
10
```

Console

```
<terminated> TestFactoryPattern [Java Application] C:\eclipse\plugins\org.eclipse.justi.openjdk hotspot\jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe (21-Jun-2025, 8:05:09 pm) [pid:
Opening a Word Document.
Opening a PDF Document.
Opening an Excel Document.
```

lab - InventoryManagementSystem/src/Document_Management_System/PdfDocumentFactory.java - Eclipse IDE

Package Explorer: InventoryManagementSystem > JRE System Library [JavaSE-17] > src > Document_Management_System > DocumentFactory.java, ExcelDocumentFactory.java, PdfDocumentFactory.java, WordDocumentFactory.java, Implementing_Singleton_Pattern, Logger.java, TestSingleton.java, module-info.java

PdfDocumentFactory.java

```
1 package Document_Management_System;
2
3 public class PdfDocumentFactory extends DocumentFactory {
4     @Override
5     public Document createDocument() {
6         return new PdfDocument();
7     }
8 }
9
10
```

Document.java

```
1 package Document_Management_System;
2
3 public interface Document {
4     void open();
5 }
6
7
```

Console

```
<terminated> TestFactoryPattern [Java Application] C:\eclipse\plugins\org.eclipse.justi.openjdk hotspot\jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe (21-Jun-2025, 8:05:11 pm) [pid:
Opening a Word Document.
Opening a PDF Document.
Opening an Excel Document.
```

Exercise 3: Implementing the Builder Pattern

Scenario:

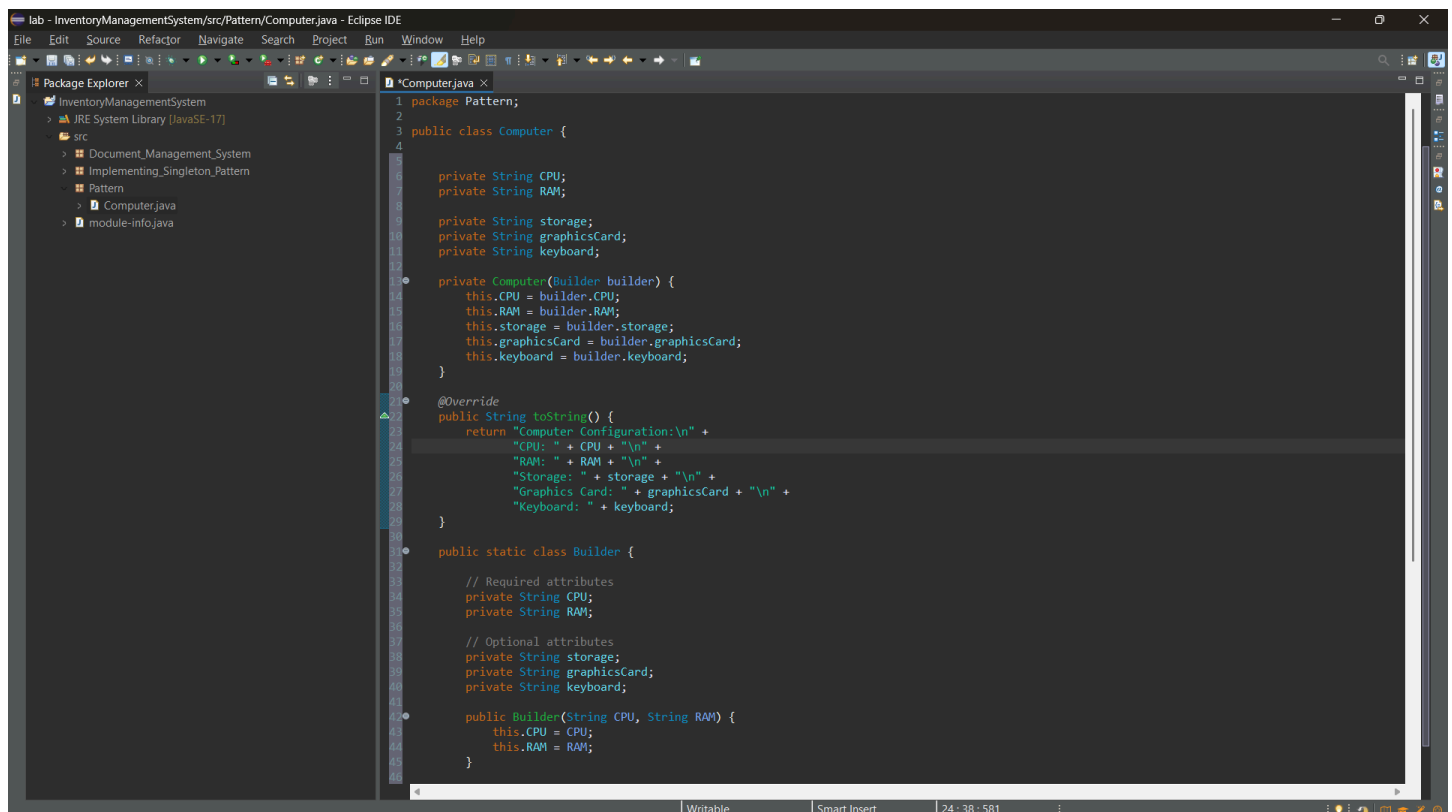
You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

Steps:

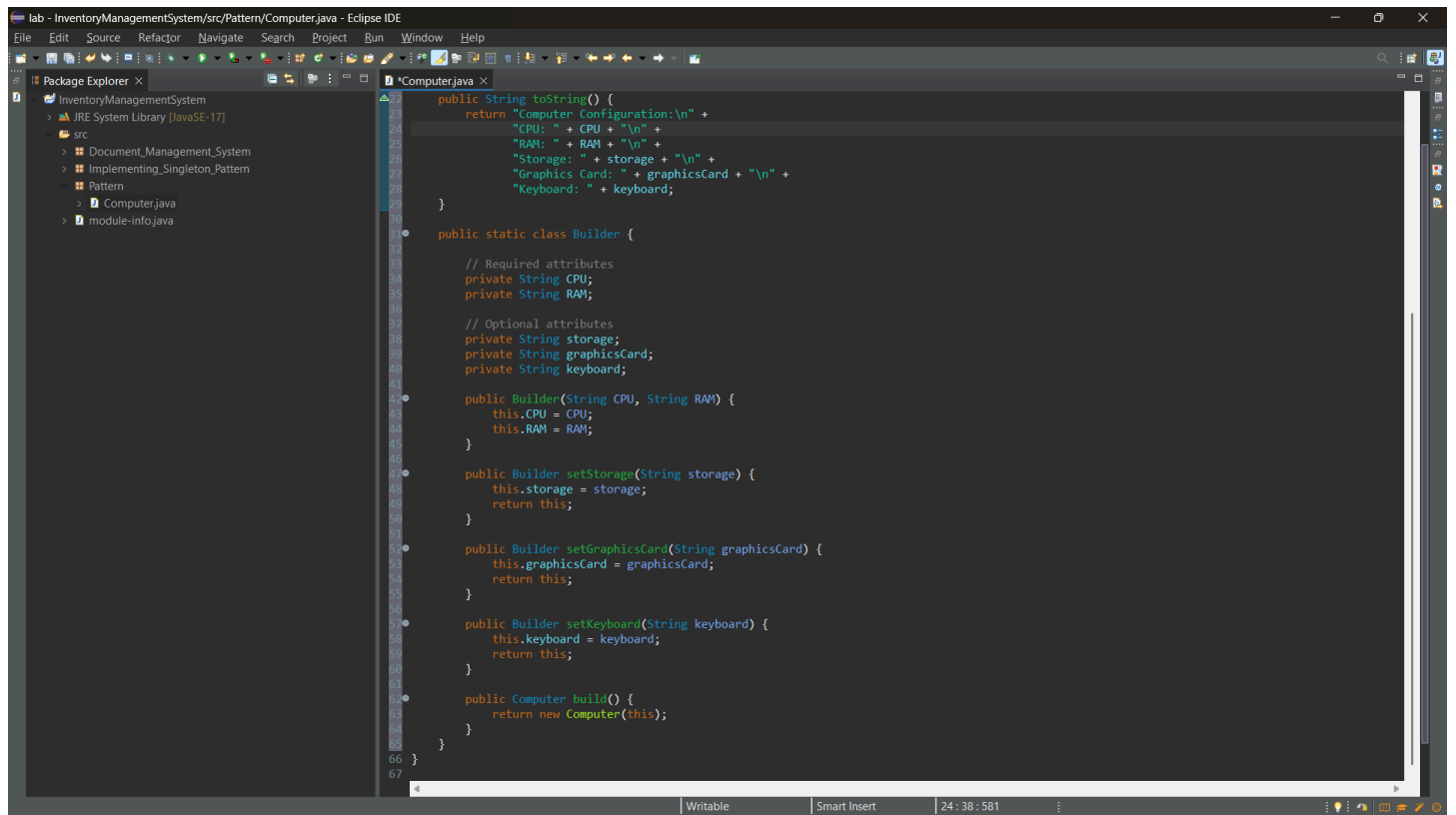
1. **Create a New Java Project:**
 - Create a new Java project named **BuilderPatternExample**.
2. **Define a Product Class:**
 - Create a class **Computer** with attributes like **CPU**, **RAM**, **Storage**, etc.
3. **Implement the Builder Class:**
 - Create a static nested Builder class inside Computer 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:

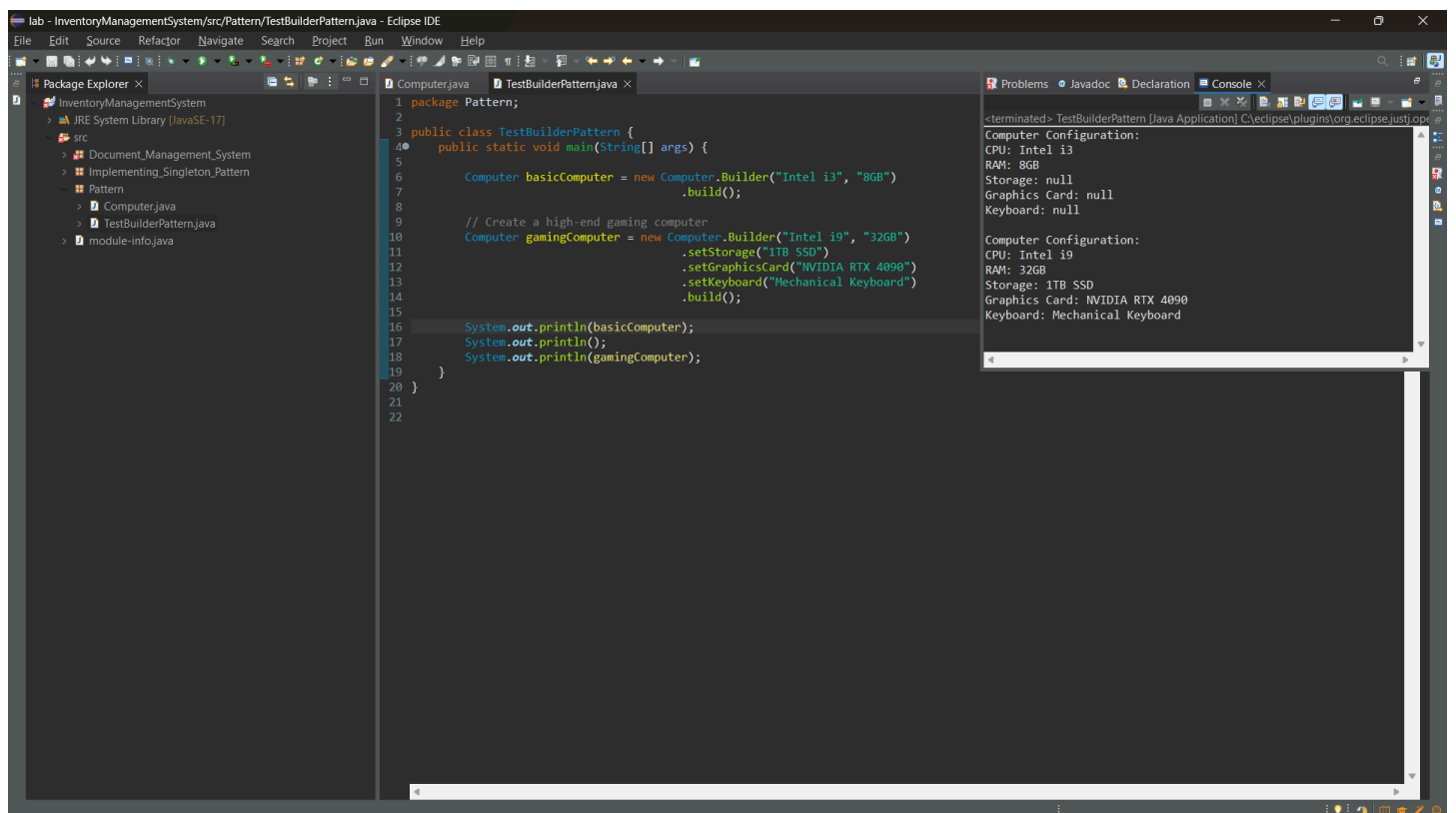


```
1 package Pattern;
2
3 public class Computer {
4
5     private String CPU;
6     private String RAM;
7
8     private String storage;
9     private String graphicsCard;
10    private String keyboard;
11
12    • private Computer(Builder builder) {
13        this.CPU = builder.CPU;
14        this.RAM = builder.RAM;
15        this.storage = builder.storage;
16        this.graphicsCard = builder.graphicsCard;
17        this.keyboard = builder.keyboard;
18    }
19
20    • @Override
21    public String toString() {
22        return "Computer Configuration:\n" +
23            "CPU: " + CPU + "\n" +
24            "RAM: " + RAM + "\n" +
25            "Storage: " + storage + "\n" +
26            "Graphics Card: " + graphicsCard + "\n" +
27            "Keyboard: " + keyboard;
28    }
29
30    • public static class Builder {
31
32        // Required attributes
33        private String CPU;
34        private String RAM;
35
36        // Optional attributes
37        private String storage;
38        private String graphicsCard;
39        private String keyboard;
40
41        • public Builder(String CPU, String RAM) {
42            this.CPU = CPU;
43            this.RAM = RAM;
44        }
45    }
46 }
```



The screenshot shows the Eclipse IDE with the file `Computer.java` open. The Package Explorer on the left shows the project structure: `InventoryManagementSystem` with sub-packages `src`, `Document_Management_System`, `Implementing_Singleton_Pattern`, `Pattern`, `Computer.java`, and `module-info.java`. The main editor displays the code for `Computer.java`, which includes a `toString()` method and a `Builder` class.

```
23 public String toString() {
24     return "Computer Configuration:\n" +
25           "CPU: " + CPU + "\n" +
26           "RAM: " + RAM + "\n" +
27           "Storage: " + storage + "\n" +
28           "Graphics Card: " + graphicsCard + "\n" +
29           "Keyboard: " + keyboard;
30 }
31
32 public static class Builder {
33
34     // Required attributes
35     private String CPU;
36     private String RAM;
37
38     // Optional attributes
39     private String storage;
40     private String graphicsCard;
41     private String keyboard;
42
43     public Builder(String CPU, String RAM) {
44         this.CPU = CPU;
45         this.RAM = RAM;
46     }
47
48     public Builder setStorage(String storage) {
49         this.storage = storage;
50         return this;
51     }
52
53     public Builder setGraphicsCard(String graphicsCard) {
54         this.graphicsCard = graphicsCard;
55         return this;
56     }
57
58     public Builder setKeyboard(String keyboard) {
59         this.keyboard = keyboard;
60         return this;
61     }
62
63     public Computer build() {
64         return new Computer(this);
65     }
66 }
67 }
```



The screenshot shows the Eclipse IDE with the file `TestBuilderPattern.java` open. The Package Explorer on the left shows the project structure: `InventoryManagementSystem` with sub-packages `src`, `Document_Management_System`, `Implementing_Singleton_Pattern`, `Pattern`, `Computer.java`, `TestBuilderPattern.java`, and `module-info.java`. The main editor displays the code for `TestBuilderPattern.java`, which includes a `main` method that creates and prints two computer configurations. The Console on the right shows the output of the program.

```
1 package Pattern;
2
3 public class TestBuilderPattern {
4     public static void main(String[] args) {
5
6         Computer basicComputer = new Computer.Builder("Intel i3", "8GB")
7             .build();
8
9         // Create a high-end gaming computer
10        Computer gamingComputer = new Computer.Builder("Intel i9", "32GB")
11            .setStorage("1TB SSD")
12            .setGraphicsCard("NVIDIA RTX 4090")
13            .setKeyboard("Mechanical Keyboard")
14            .build();
15
16        System.out.println(basicComputer);
17        System.out.println();
18        System.out.println(gamingComputer);
19    }
20 }
21
22 }
```

Console Output:

```
<terminated> TestBuilderPattern [Java Application] C:\eclipse\plugins\org.eclipse.justj4p...
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:

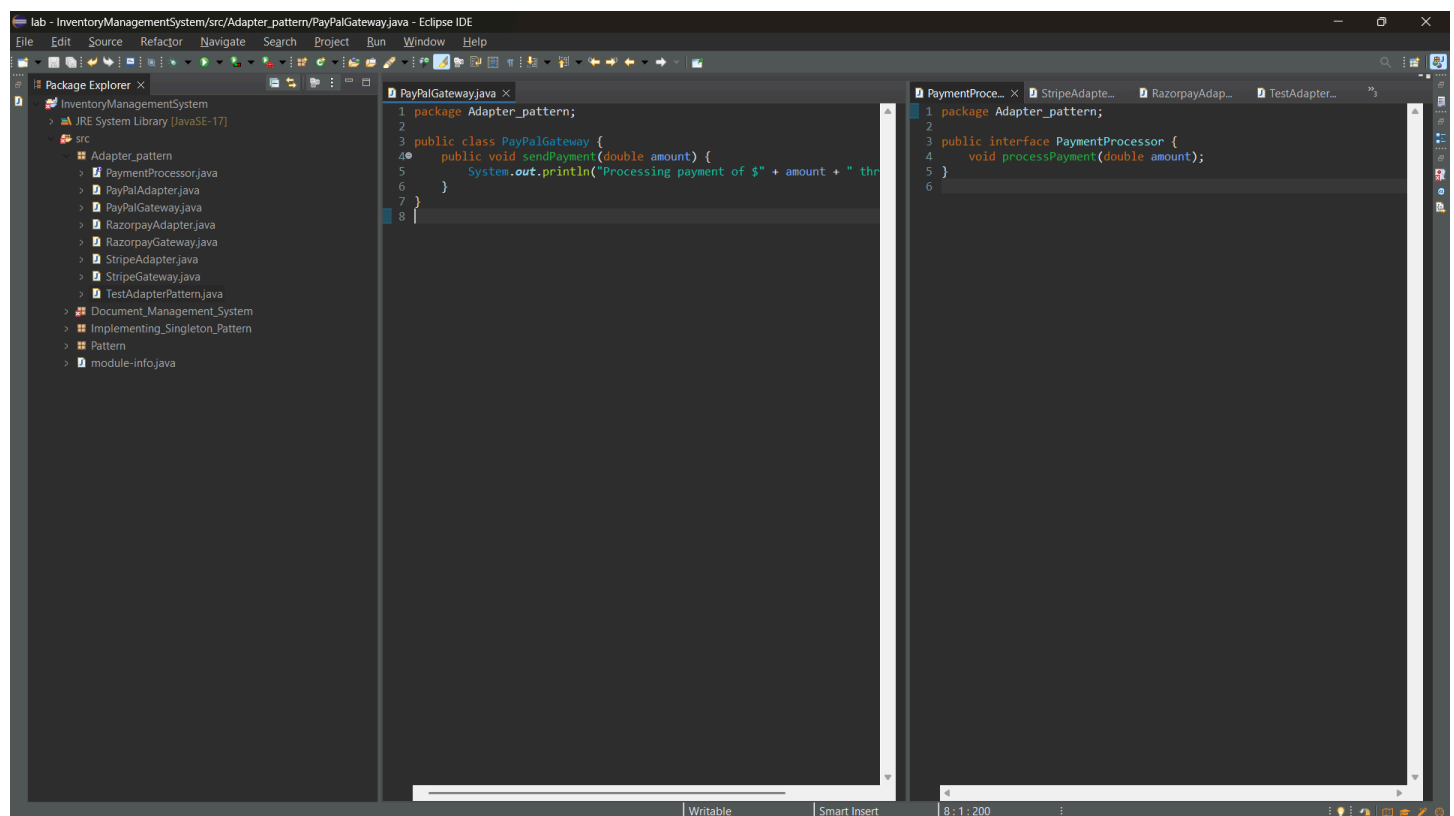
You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

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:



The screenshot displays the Eclipse IDE interface with a project named 'InventoryManagementSystem'. The Package Explorer on the left shows the project structure, including a package named 'Adapter_pattern' containing several classes: 'PaymentProcessor.java', 'PayPalGateway.java', 'PayPalGatewayAdapter.java', 'RazorpayGatewayAdapter.java', 'StripeGatewayAdapter.java', 'StripeGateway.java', 'TestAdapterPattern.java', 'Document_Management_System', 'Implementing_Singleton_Pattern', 'Pattern', and 'module-info.java'. The main editor area shows the code for 'PayPalGateway.java' and 'PaymentProcessor.java'. The 'PayPalGateway.java' file contains the following code:

```
1 package Adapter_pattern;
2
3 public class PayPalGateway {
4     public void sendPayment(double amount) {
5         System.out.println("Processing payment of $" + amount + " through PayPal");
6     }
7 }
8
```

The 'PaymentProcessor.java' file contains the following code:

```
1 package Adapter_pattern;
2
3 public interface PaymentProcessor {
4     void processPayment(double amount);
5 }
6
```

The status bar at the bottom indicates 'Writable', 'Smart Insert', and '8 : 1 : 200'.

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

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer X

- 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 X

```
1 package Adapter_pattern;
2 public class PayPalAdapter implements PaymentProcessor {
3
4     private PayPalGateway payPalGateway;
5
6     public PayPalAdapter(PayPalGateway payPalGateway) {
7         this.payPalGateway = payPalGateway;
8     }
9
10    @Override
11    public void processPayment(double amount) {
12        payPalGateway.sendPayment(amount);
13    }
14 }
15
```

StripeGateway.java X StripeAdapter.java X RazorpayAdapter.java X TestAdapterPattern.java X

```
1 package Adapter_pattern;
2
3 public class StripeGateway {
4     public void makeStripePayment(double amount) {
5         System.out.println("Processing payment of $" + amount + " through Stripe");
6     }
7 }
8
```

15:1:359

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

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer X

- 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 X

```
1 package Adapter_pattern;
2 public class RazorpayGateway {
3     public void initiateRazorpayPayment(double amount) {
4         System.out.println("Processing payment of $" + amount + " through Razorpay");
5     }
6 }
7
```

PayPalAdapter.java X StripeAdapter.java X RazorpayAdapter.java X TestAdapterPattern.java X

```
1 package Adapter_pattern;
2
3 public class TestAdapterPattern {
4     public static void main(String[] args) {
5
6         // PayPal Payment
7         PayPalGateway paypal = new PayPalGateway();
8         PaymentProcessor paypalProcessor = new PayPalAdapter(paypal);
9         paypalProcessor.processPayment(1500);
10
11         // Stripe Payment
12         StripeGateway stripe = new StripeGateway();
13         PaymentProcessor stripeProcessor = new StripeAdapter(stripe);
14         stripeProcessor.processPayment(2500);
15
16         // Razorpay Payment
17         RazorpayGateway razorpay = new RazorpayGateway();
18         PaymentProcessor razorpayProcessor = new RazorpayAdapter(razorpay);
19         razorpayProcessor.processPayment(3500);
20     }
21 }
22
```

7:1:214

```
1 package Adapter_pattern;
2
3 public class RazorpayAdapter implements PaymentProcessor {
4
5     private RazorpayGateway razorpayGateway;
6
7     public RazorpayAdapter(RazorpayGateway razorpayGateway) {
8         this.razorpayGateway = razorpayGateway;
9     }
10
11     @Override
12     public void processPayment(double amount) {
13         razorpayGateway.initiateRazorpayPayment(amount);
14     }
15 }
16
17
```

```
1 package Adapter_pattern;
2
3 public class StripeAdapter implements PaymentProcessor {
4
5     private StripeGateway stripeGateway;
6
7     public StripeAdapter(StripeGateway stripeGateway) {
8         this.stripeGateway = stripeGateway;
9     }
10
11     @Override
12     public void processPayment(double amount) {
13         stripeGateway.makeStripePayment(amount);
14     }
15 }
16
```

```
1 package Adapter_pattern;
2
3 public class TestAdapterPattern {
4     public static void main(String[] args) {
5
6         // PayPal Payment
7         PayPalGateway paypal = new PayPalGateway();
8         PaymentProcessor paypalProcessor = new PayPalAdapter(paypal);
9         paypalProcessor.processPayment(1500);
10
11         // Stripe Payment
12         StripeGateway stripe = new StripeGateway();
13         PaymentProcessor stripeProcessor = new StripeAdapter(stripe);
14         stripeProcessor.processPayment(2500);
15
16         // Razorpay Payment
17         RazorpayGateway razorpay = new RazorpayGateway();
18         PaymentProcessor razorpayProcessor = new RazorpayAdapter(razorpay);
19         razorpayProcessor.processPayment(3500);
20     }
21 }
22
```

<terminated> TestAdapterPattern [Java Application] C:\eclipse\plugins\org.eclipse.justic4
Processing payment of \$1500.0 through PayPal.
Processing payment of \$2500.0 through Stripe.
Processing payment of \$3500.0 through Razorpay.

Exercise 5: Implementing the Decorator Pattern

Scenario:

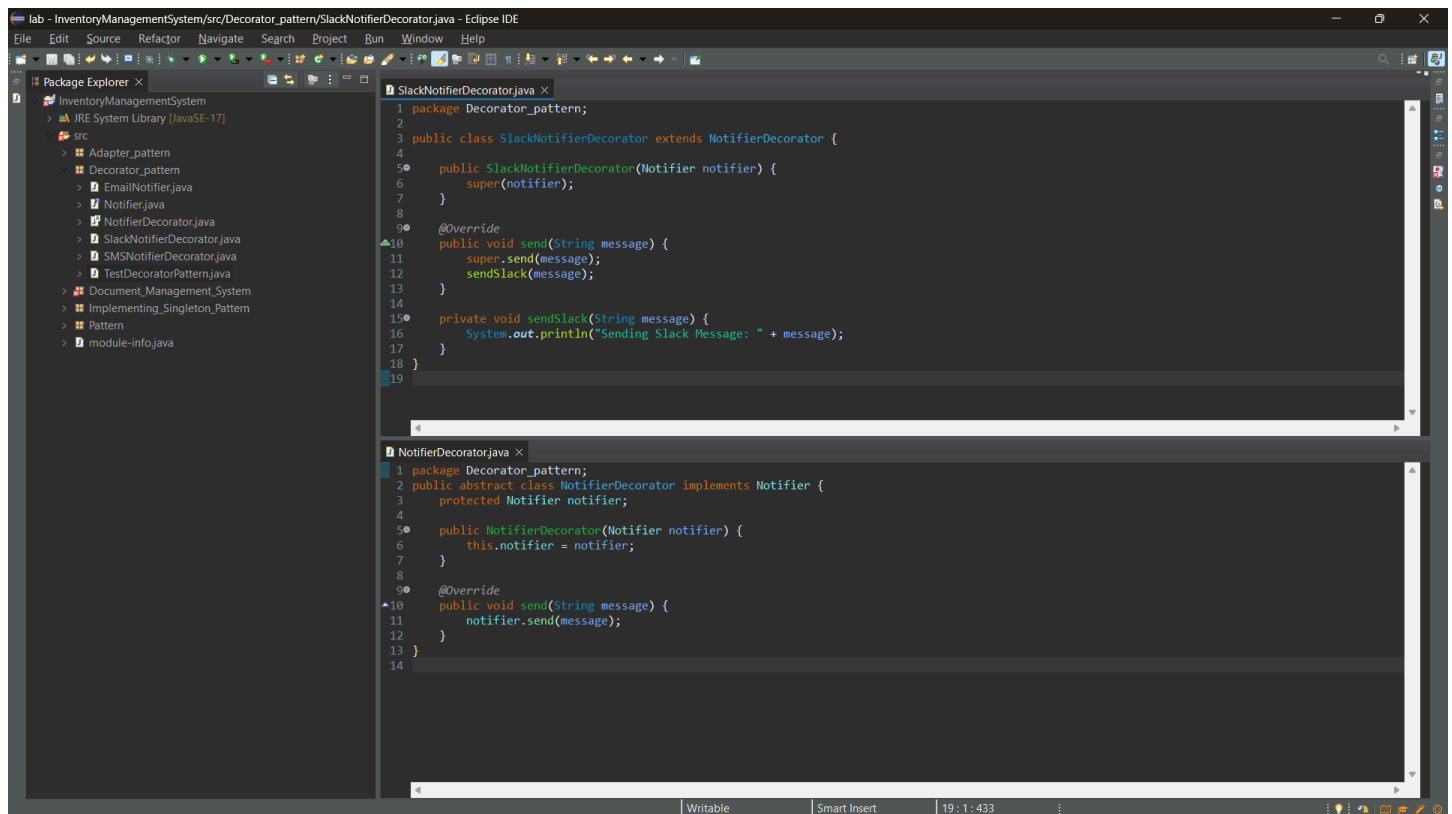
You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically.

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:



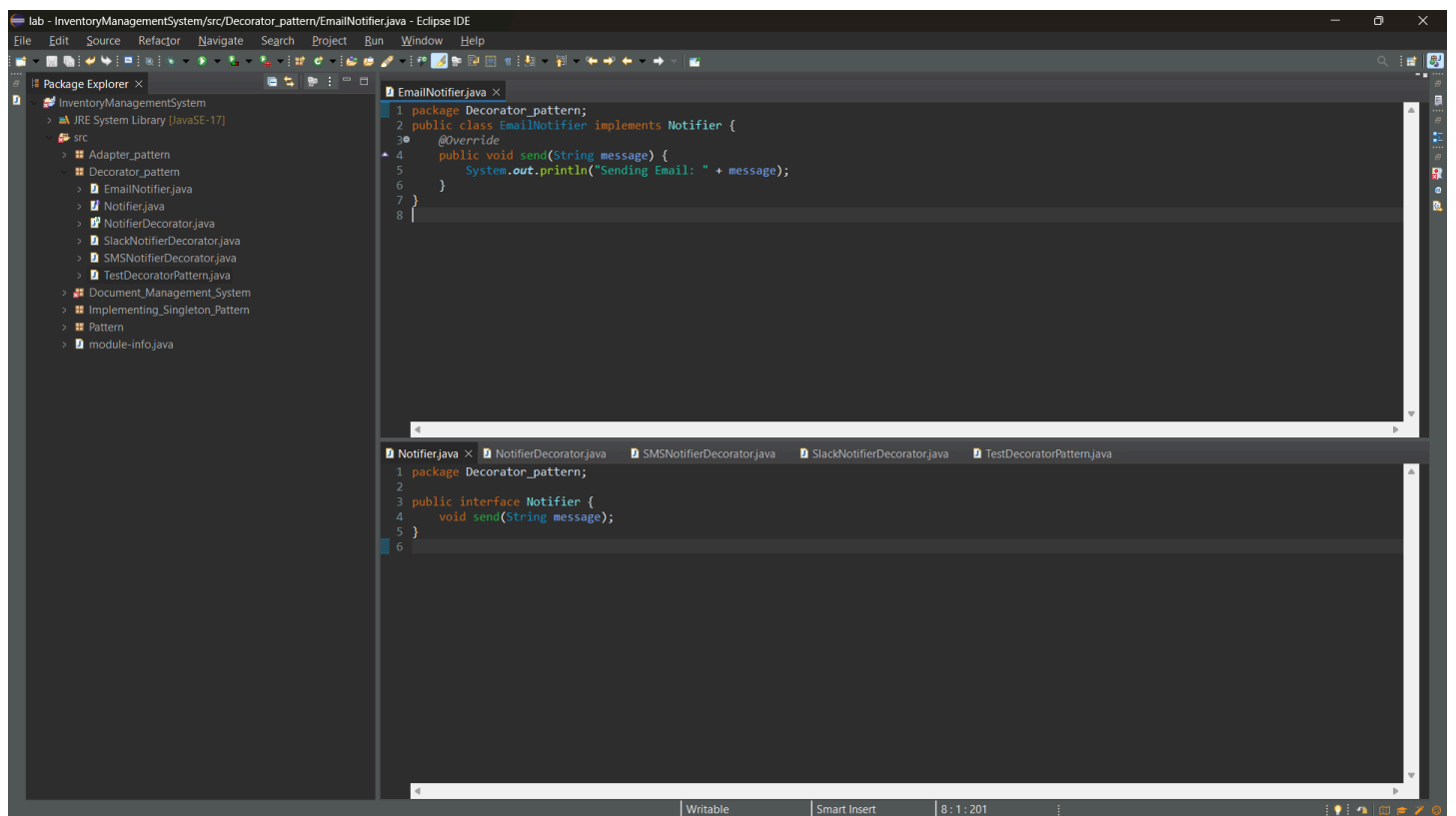
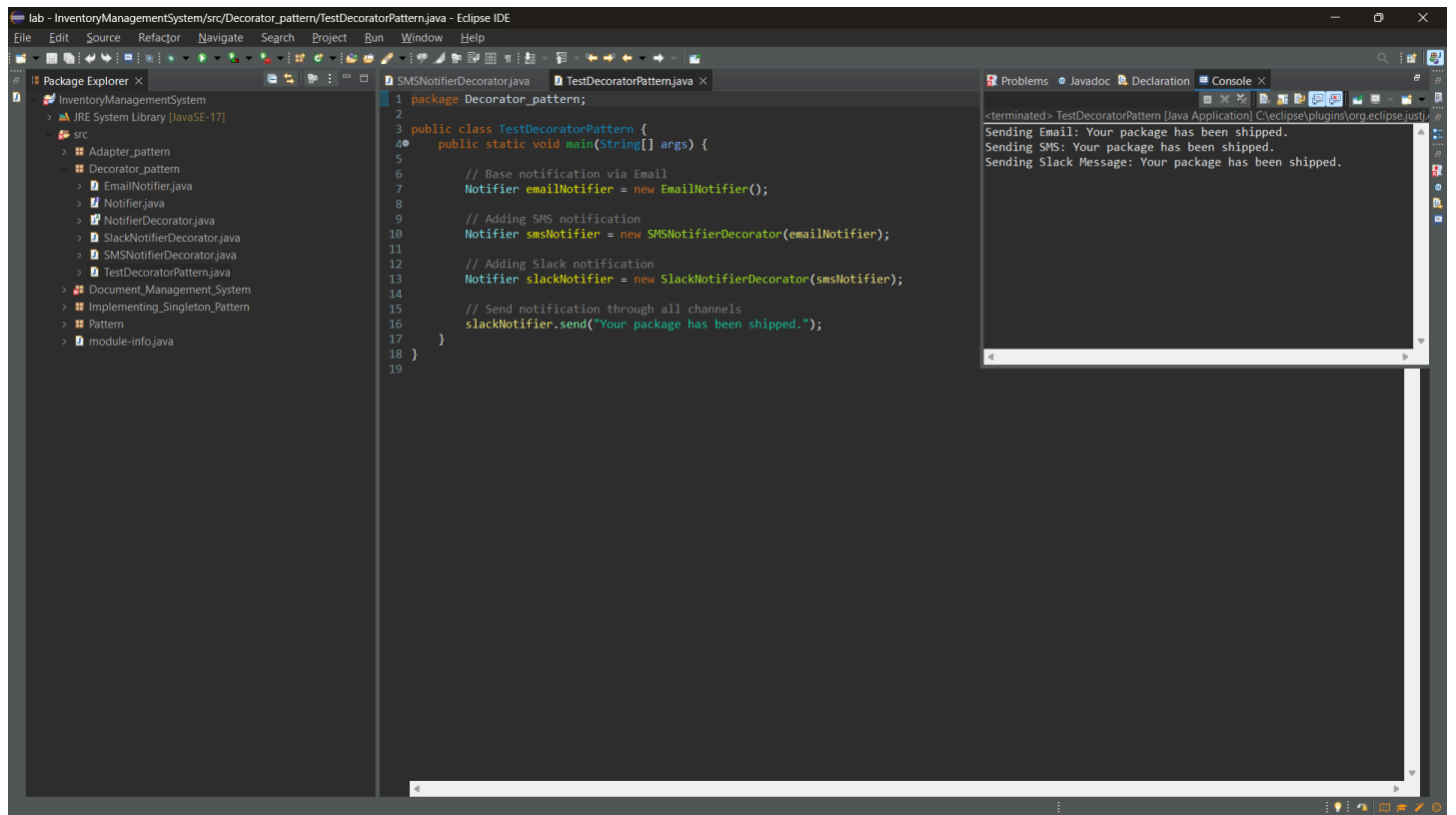
The screenshot shows the Eclipse IDE with two Java files open in the editor. The Package Explorer on the left shows the project structure: 'InventoryManagementSystem' with sub-packages 'src', 'Decorator_pattern', and 'src'. The 'Decorator_pattern' package contains 'EmailNotifier.java', 'Notifier.java', 'NotifierDecorator.java', 'SMSNotifierDecorator.java', and 'SlackNotifierDecorator.java'. The 'src' package contains 'Document_Management_System', 'Implementing_Singleton_Pattern', 'Pattern', and 'module-info.java'.

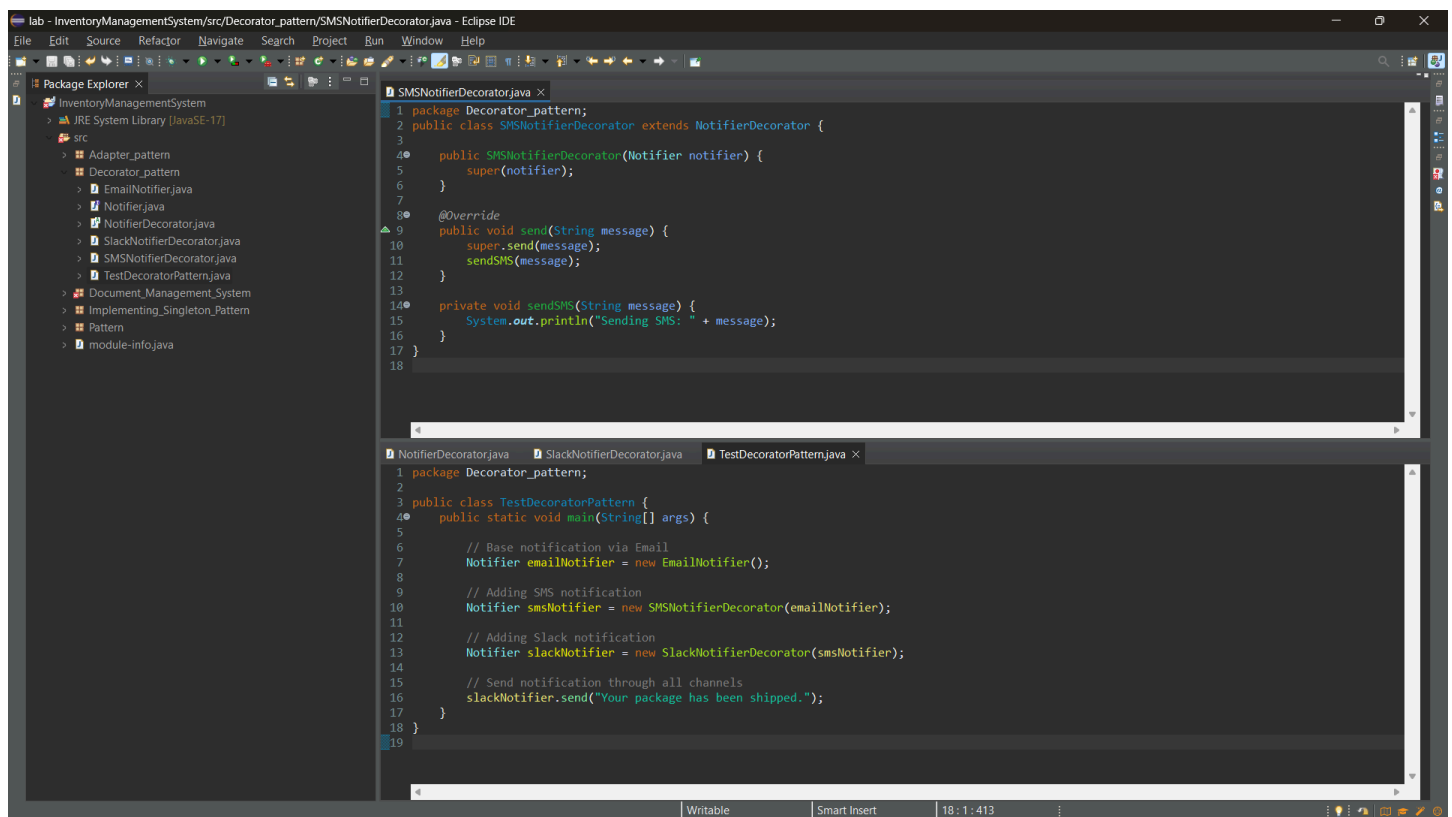
The top editor shows the **SlackNotifierDecorator.java** file:

```
1 package Decorator_pattern;
2
3 public class SlackNotifierDecorator extends NotifierDecorator {
4
5     public SlackNotifierDecorator(Notifier notifier) {
6         super(notifier);
7     }
8
9     @Override
10    public void send(String message) {
11        super.send(message);
12        sendSlack(message);
13    }
14
15    private void sendSlack(String message) {
16        System.out.println("Sending Slack Message: " + message);
17    }
18 }
19
```

The bottom editor shows the **NotifierDecorator.java** file:

```
1 package Decorator_pattern;
2 public abstract class NotifierDecorator implements Notifier {
3     protected Notifier notifier;
4
5     public NotifierDecorator(Notifier notifier) {
6         this.notifier = notifier;
7     }
8
9     @Override
10    public void send(String message) {
11        notifier.send(message);
12    }
13 }
14
```





Exercise 6: Implementing the Proxy Pattern

Scenario:

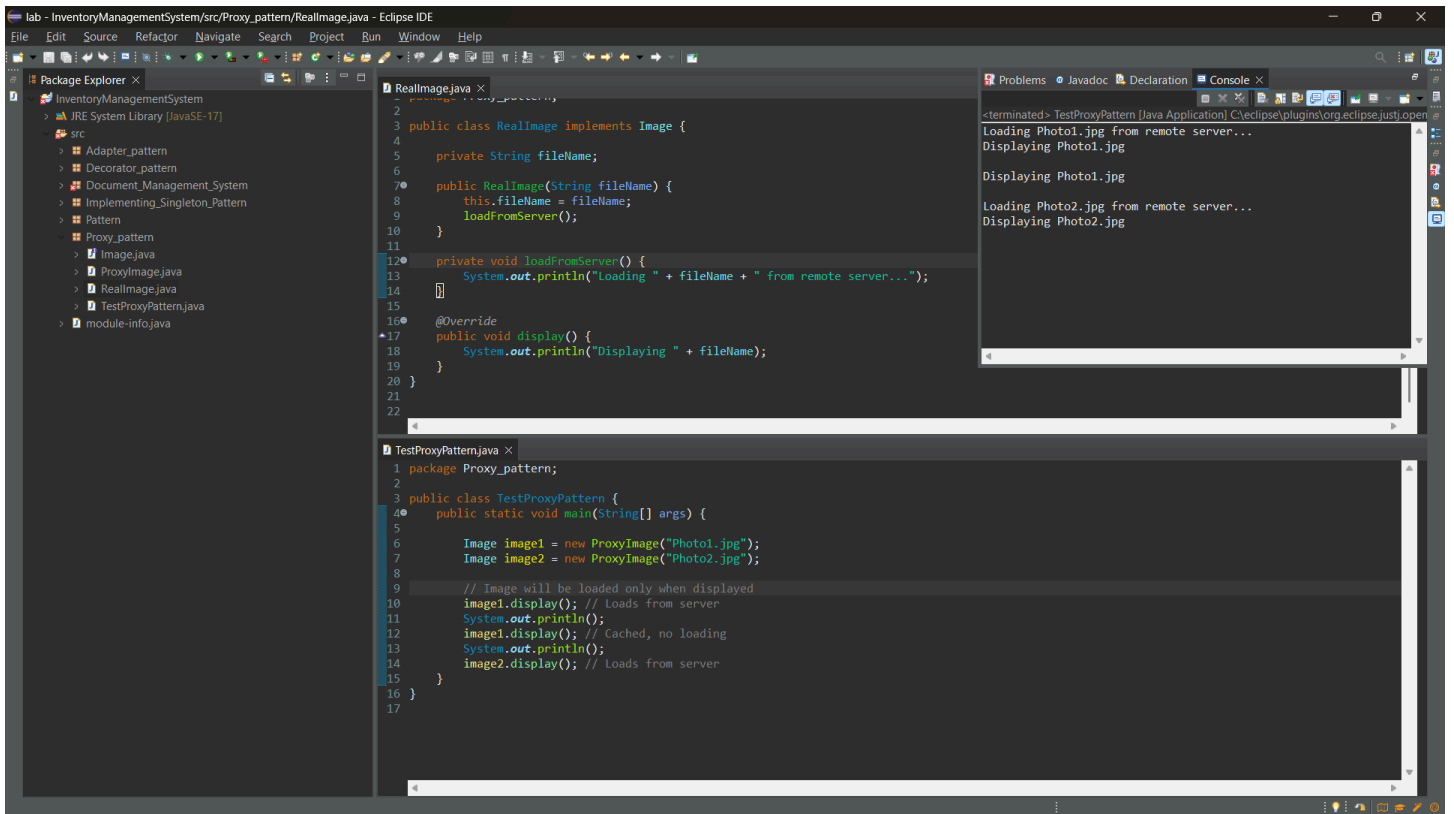
You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching.

Steps:

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

Solution:

Code:



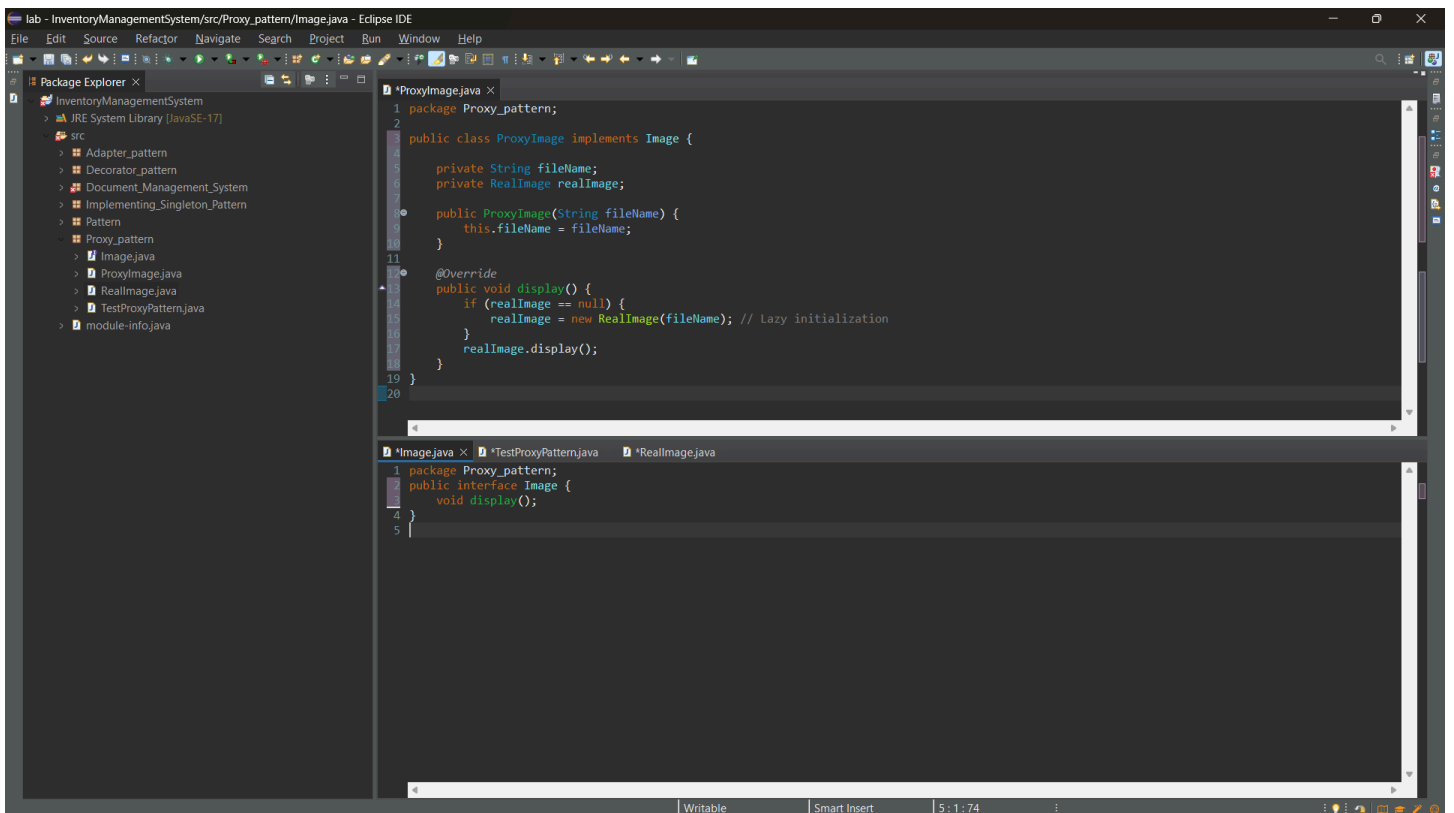
```
lab - InventoryManagementSystem/src/Proxy_pattern/RealImage.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer
InventoryManagementSystem
  JRE System Library [JavaSE-17]
  src
    Adapter_pattern
    Decorator_pattern
    Document_Management_System
    Implementing_Singleton_Pattern
    Pattern
    Proxy_pattern
      Image.java
      ProxyImage.java
      RealImage.java
      TestProxyPattern.java
      module-info.java

RealImage.java
1 package Proxy_pattern;
2
3 public class RealImage implements Image {
4
5     private String fileName;
6
7     public RealImage(String fileName) {
8         this.fileName = fileName;
9         loadFromServer();
10    }
11
12    private void loadFromServer() {
13        System.out.println("Loading " + fileName + " from remote server...");
14    }
15
16    @Override
17    public void display() {
18        System.out.println("Displaying " + fileName);
19    }
20 }
21
22

TestProxyPattern.java
1 package Proxy_pattern;
2
3 public class TestProxyPattern {
4     public static void main(String[] args) {
5
6         Image image1 = new ProxyImage("Photo1.jpg");
7         Image image2 = new ProxyImage("Photo2.jpg");
8
9         // Image will be loaded only when displayed
10        image1.display(); // Loads from server
11        System.out.println();
12        image1.display(); // Cached, no loading
13        System.out.println();
14        image2.display(); // Loads from server
15    }
16 }
17

Problems Javadoc Declaration Console
<terminated> TestProxyPattern [Java Application] C:\eclipse\plugins\org.eclipse.justi.open
Loading Photo1.jpg from remote server...
Displaying Photo1.jpg
Displaying Photo1.jpg
Loading Photo2.jpg from remote server...
Displaying Photo2.jpg
```



```
lab - InventoryManagementSystem/src/Proxy_pattern/Image.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
    Pattern
    Proxy_pattern
      Image.java
      ProxyImage.java
      RealImage.java
      TestProxyPattern.java
      module-info.java

ProxyImage.java
1 package Proxy_pattern;
2
3 public class ProxyImage implements Image {
4
5     private String fileName;
6     private RealImage realImage;
7
8     public ProxyImage(String fileName) {
9         this.fileName = fileName;
10    }
11
12    @Override
13    public void display() {
14        if (realImage == null) {
15            realImage = new RealImage(fileName); // lazy initialization
16        }
17        realImage.display();
18    }
19 }
20

Image.java
1 package Proxy_pattern;
2
3 public interface Image {
4     void display();
5 }

Writable Smart Insert 5:1:74
```

Exercise 7: Implementing the Observer Pattern

Scenario:

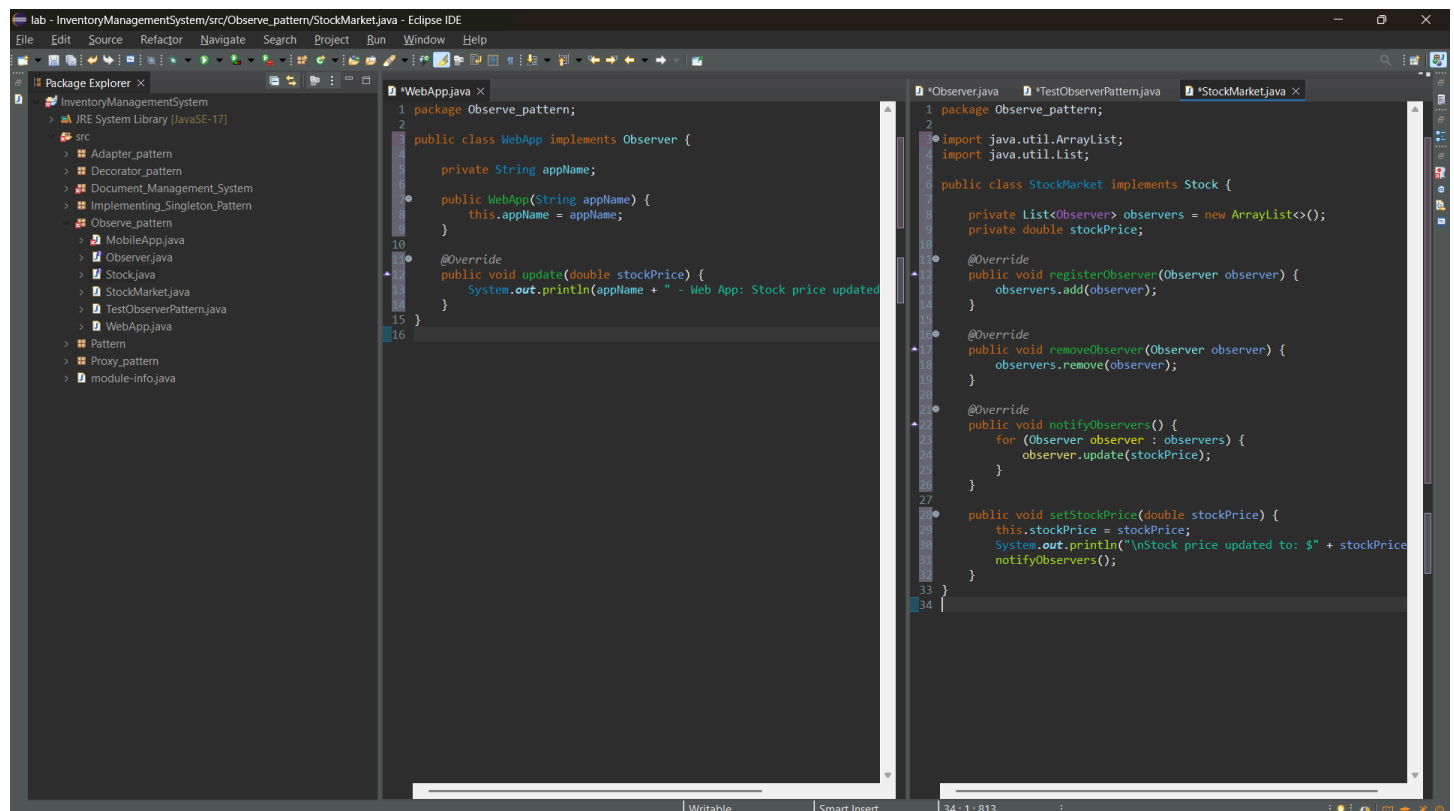
You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

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



```
lab - InventoryManagementSystem/src/Observe_pattern/StockMarket.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
      MobileApp.java
      Observer.java
      Stock.java
      StockMarket.java
      TestObserverPattern.java
      WebApp.java
    Pattern
    Proxy_pattern
    module-info.java

WebApp.java
1 package Observe_pattern;
2 public class WebApp implements Observer {
3     private String appName;
4     public WebApp(String appName) {
5         this.appName = appName;
6     }
7     @Override
8     public void update(double stockPrice) {
9         System.out.println(appName + " - Web App: Stock price updated to: " + stockPrice);
10    }
11 }

Observer.java
1 package Observe_pattern;
2 import java.util.ArrayList;
3 import java.util.List;
4 public class Observer {
5     private List<Observer> observers = new ArrayList<>();
6     private double stockPrice;
7     @Override
8     public void registerObserver(Observer observer) {
9         observers.add(observer);
10    }
11     @Override
12     public void removeObserver(Observer observer) {
13         observers.remove(observer);
14    }
15     @Override
16     public void notifyObservers() {
17         for (Observer observer : observers) {
18             observer.update(stockPrice);
19         }
20    }
21     public void setStockPrice(double stockPrice) {
22         this.stockPrice = stockPrice;
23         System.out.println("Stock price updated to: $" + stockPrice);
24         notifyObservers();
25    }
26 }

StockMarket.java
1 package Observe_pattern;
2 import java.util.ArrayList;
3 import java.util.List;
4 public class StockMarket implements Stock {
5     private List<Observer> observers = new ArrayList<>();
6     private double stockPrice;
7     @Override
8     public void registerObserver(Observer observer) {
9         observers.add(observer);
10    }
11     @Override
12     public void removeObserver(Observer observer) {
13         observers.remove(observer);
14    }
15     @Override
16     public void notifyObservers() {
17         for (Observer observer : observers) {
18             observer.update(stockPrice);
19         }
20    }
21     public void setStockPrice(double stockPrice) {
22         this.stockPrice = stockPrice;
23         System.out.println("Stock price updated to: $" + stockPrice);
24         notifyObservers();
25    }
26 }
```

The screenshot shows the Eclipse IDE with the 'InventoryManagementSystem' project. The Package Explorer on the left shows the project structure. The main editor displays the 'StockMarket.java' file, which implements the 'Stock' interface. The code includes imports for 'java.util.ArrayList' and 'java.util.List', and defines a 'StockMarket' class with a 'registerObserver' method, a 'removeObserver' method, and an '@Override' annotation. The 'TestObserverPattern.java' file is also visible in the editor, showing the 'main' method that creates a 'StockMarket' object, registers observers, and updates stock prices.

```
1 package Observe_pattern;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class StockMarket implements Stock {
7
8     private List<Observer> observers = new ArrayList<>();
9     private double stockPrice;
10
11     @Override
12     public void registerObserver(Observer observer) {
13         observers.add(observer);
14     }
15
16     @Override
17     public void removeObserver(Observer observer) {
18         observers.remove(observer);
19     }
20
21     @Override
22     public void notifyObservers() {
23         // Implementation of notifyObservers
24     }
25 }
```

The screenshot shows the Eclipse IDE with the 'InventoryManagementSystem' project. The Package Explorer on the left shows the project structure. The main editor displays the 'TestObserverPattern.java' file, which implements the 'Observer' interface. The code includes imports for 'java.util.ArrayList' and 'java.util.List', and defines a 'TestObserverPattern' class with a 'main' method that creates a 'StockMarket' object, registers observers, and updates stock prices. The 'Observer.java' file is also visible in the editor, showing the 'Observer' interface with an 'update' method. The 'Console' window on the right shows the output of the program, displaying stock price updates for 'InvestorPro' and 'StockTracker'.

```
1 package Observe_pattern;
2
3 public class TestObserverPattern {
4     public static void main(String[] args) {
5
6         // Create stock market
7         StockMarket stockMarket = new StockMarket();
8
9         // Create observers
10        Observer mobileApp = new MobileApp("InvestorPro");
11        Observer webApp = new WebApp("StockTracker");
12
13        // Register observers
14        stockMarket.registerObserver(mobileApp);
15        stockMarket.registerObserver(webApp);
16
17        // Change stock prices
18        stockMarket.setStockPrice(100.50);
19        stockMarket.setStockPrice(105.75);
20
21        // Remove one observer
22        stockMarket.removeObserver(webApp);
23    }
24 }
```

```
1 package Observe_pattern;
2
3 public interface Observer {
4     void update(double stockPrice);
5 }
6
7
```

Console Output:

```
<terminated> TestObserverPattern [Java Application] C:\eclipse\plugins\org.eclipse.justi.o
Stock price updated to: $100.5
InvestorPro - Mobile App: Stock price updated to $100.5
StockTracker - Web App: Stock price updated to $100.5

Stock price updated to: $105.75
InvestorPro - Mobile App: Stock price updated to $105.75
StockTracker - Web App: Stock price updated to $105.75

Stock price updated to: $110.2
InvestorPro - Mobile App: Stock price updated to $110.2
```

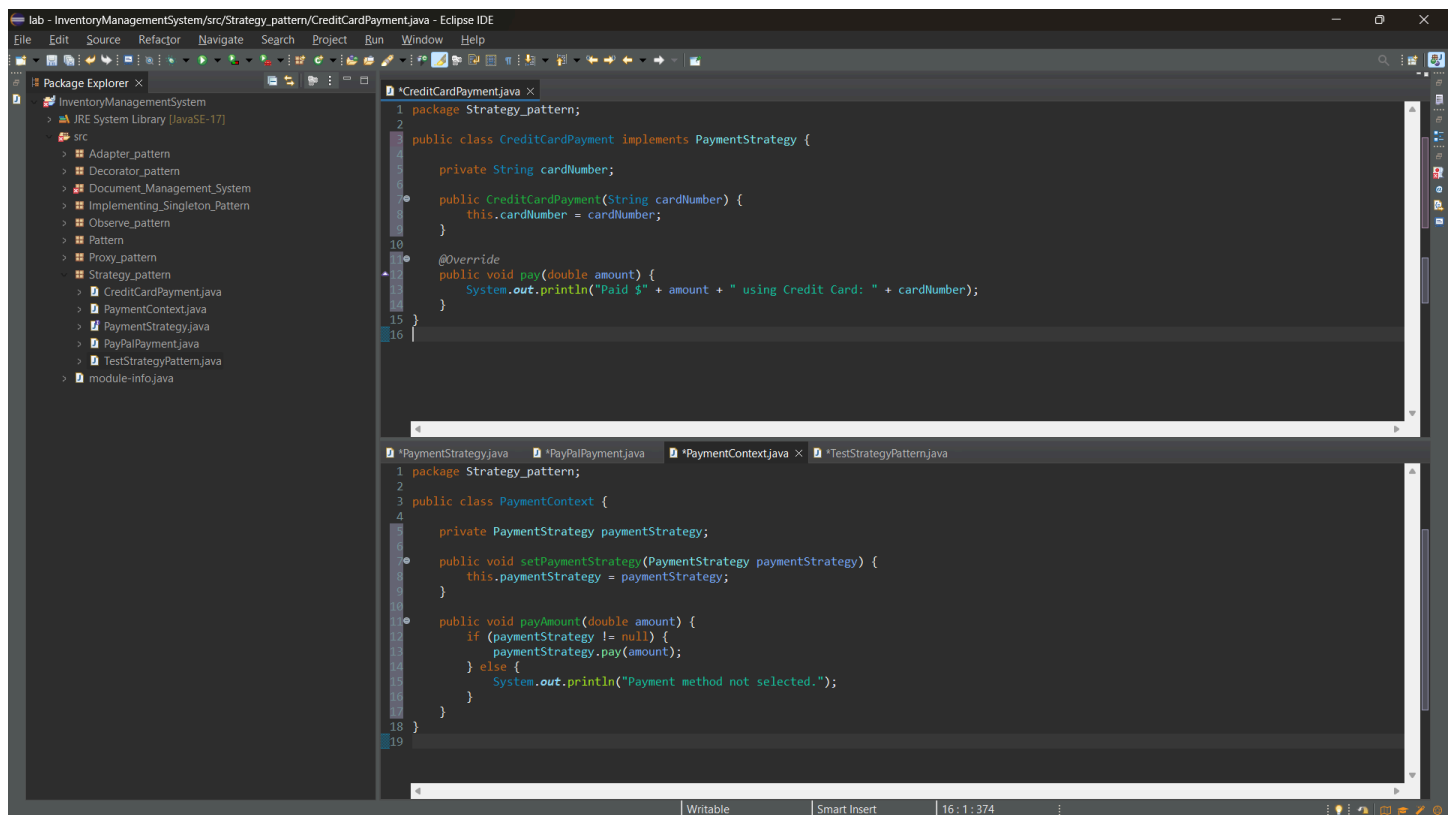
Exercise 8: Implementing the Strategy Pattern

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.



```
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
1 package Strategy_pattern;
2
3 public class CreditCardPayment implements PaymentStrategy {
4
5     private String cardNumber;
6
7     public CreditCardPayment(String cardNumber) {
8         this.cardNumber = cardNumber;
9     }
10
11     @Override
12     public void pay(double amount) {
13         System.out.println("Paid $" + amount + " using Credit Card: " + cardNumber);
14     }
15 }
16

PaymentStrategy.java
1 package Strategy_pattern;
2
3 public interface PaymentStrategy {
4     void pay(double amount);
5 }

PaymentContext.java
1 package Strategy_pattern;
2
3 public class PaymentContext {
4
5     private PaymentStrategy paymentStrategy;
6
7     public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
8         this.paymentStrategy = paymentStrategy;
9     }
10
11     public void payAmount(double amount) {
12         if (paymentStrategy != null) {
13             paymentStrategy.pay(amount);
14         } else {
15             System.out.println("Payment method not selected.");
16         }
17     }
18 }
19

TestStrategyPattern.java
1 package Strategy_pattern;
2
3 public class TestStrategyPattern {
4
5     public static void main(String[] args) {
6         PaymentContext context = new PaymentContext();
7         CreditCardPayment cardPayment = new CreditCardPayment("1234 5678 9010 1010");
8         context.setPaymentStrategy(cardPayment);
9         context.payAmount(100);
10
11         PayPalPayment paypalPayment = new PayPalPayment("test@example.com", "1234567890");
12         context.setPaymentStrategy(paypalPayment);
13         context.payAmount(200);
14     }
15 }
16 }
```

The screenshot shows the Eclipse IDE with the following components:

- Package Explorer:** Displays the project structure for 'InventoryManagementSystem'. The 'src' folder contains several pattern-related packages, including 'Strategy_pattern' which contains 'CreditCardPayment.java', 'PaymentContext.java', 'PaymentStrategy.java', 'PayPalPayment.java', and 'TestStrategyPattern.java'.
- Main Editor:** Shows the code for 'PayPalPayment.java'. It defines a class that implements the 'PaymentStrategy' interface. The class has a private 'email' field, a constructor to set the email, and an overridden 'pay' method that prints the payment details using the email.
- Test Editor:** Shows the code for 'TestStrategyPattern.java'. It contains a 'main' method that creates a 'PaymentContext', sets the 'PaymentStrategy' to 'CreditCardPayment' and 'PayPalPayment', and calls 'payAmount' to demonstrate the functionality.

```
1 package Strategy_pattern;
2
3 public class PayPalPayment implements PaymentStrategy {
4
5     private String email;
6
7     public PayPalPayment(String email) {
8         this.email = email;
9     }
10
11     @Override
12     public void pay(double amount) {
13         System.out.println("Paid $" + amount + " using PayPal account: " + email);
14     }
15 }
16
```

```
1 package Strategy_pattern;
2
3 public class TestStrategyPattern {
4     public static void main(String[] args) {
5
6         PaymentContext paymentContext = new PaymentContext();
7
8         // Paying via Credit Card
9         paymentContext.setPaymentStrategy(new CreditCardPayment("1234-5678-9876-5432"));
10        paymentContext.payAmount(500.0);
11
12        // Paying via PayPal
13        paymentContext.setPaymentStrategy(new PayPalPayment("user@example.com"));
14        paymentContext.payAmount(1000.0);
15    }
16 }
17
```

This screenshot shows the same Eclipse IDE environment after running the test. The 'Console' window is open, displaying the output of the 'TestStrategyPattern' application. The output shows two payment transactions: one for \$500.0 using a credit card and another for \$1000.0 using a PayPal account.

```
<terminated> TestStrategyPattern [Java Application] C:\eclipse\plugins\org.eclipse.justi.or
Paid $500.0 using Credit Card: 1234-5678-9876-5432
Paid $1000.0 using PayPal account: user@example.com
```

The code editors below the console show the same source files as the previous screenshot, but the 'TestStrategyPattern.java' file is now partially obscured by the console output.

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:

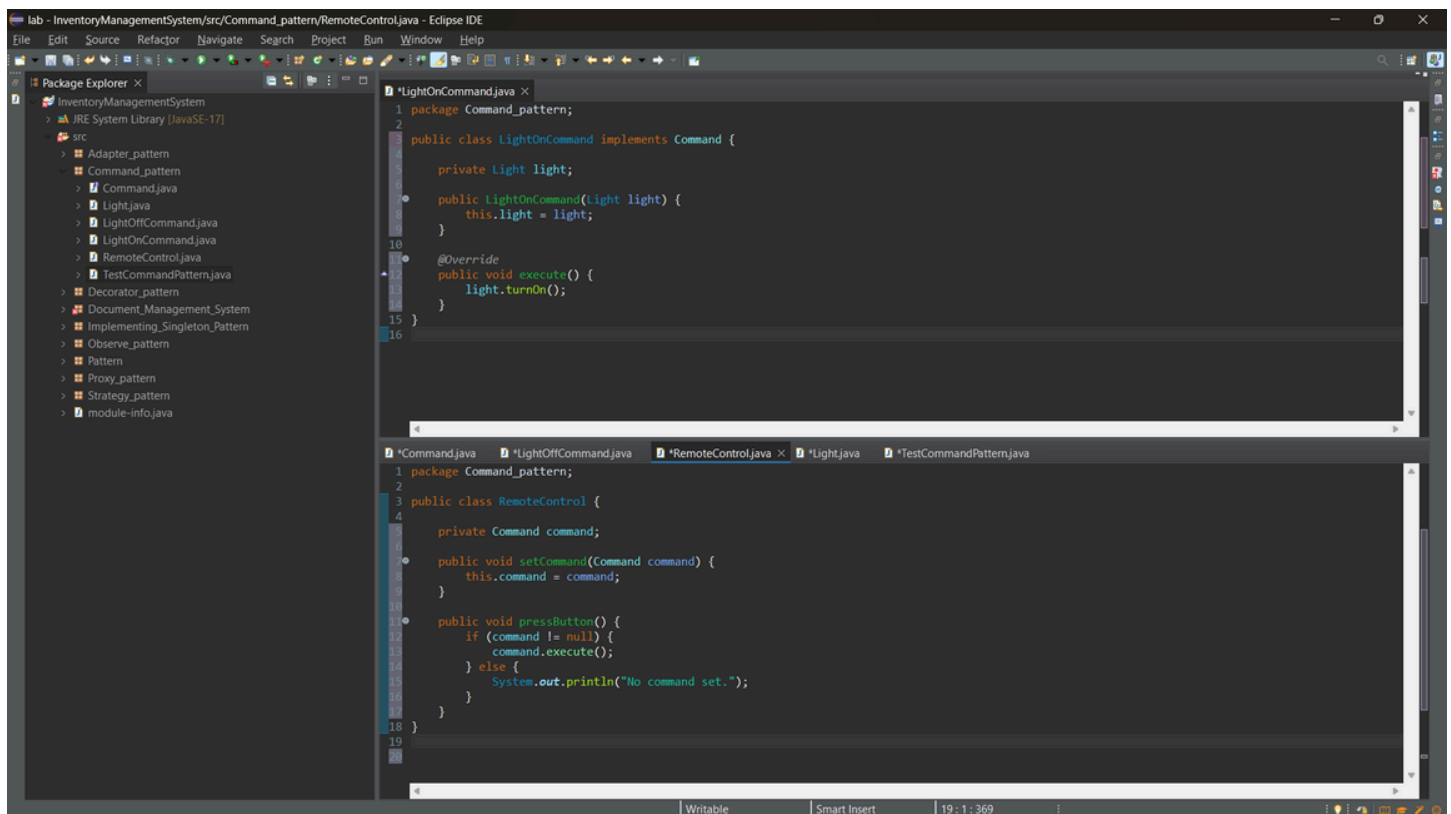
- Create a class **RemoteControl** that holds a reference to a **Command** and a method to execute the command.

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**.



```
lab - InventoryManagementSystem/src/Command_pattern/Light.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer
InventoryManagementSystem
  JRE System Library [JavaSE-17]
  src
    Adapter_pattern
    Command_pattern
    Command.java
    Light.java
    LightOffCommand.java
    LightOnCommand.java
    RemoteControl.java
    TestCommandPattern.java
    Decorator_pattern
    Document_Management_System
    Implementing_Singleton_Pattern
    Observe_pattern
    Pattern
    Proxy_pattern
    Strategy_pattern
    module-info.java

LightOffCommand.java
1 package Command_pattern;
2
3 public class LightOffCommand implements Command {
4
5     private Light light;
6
7     public LightOffCommand(Light light) {
8         this.light = light;
9     }
10
11     @Override
12     public void execute() {
13         light.turnOff();
14     }
15 }
16

Command.java
1 package Command_pattern;
2
3 public class Light {
4
5     public void turnOn() {
6         System.out.println("The light is ON.");
7     }
8
9     public void turnOff() {
10        System.out.println("The light is OFF.");
11    }
12 }
13
```

```
lab - InventoryManagementSystem/src/Command_pattern/TestCommandPattern.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer
InventoryManagementSystem
  JRE System Library [JavaSE-17]
  src
    Adapter_pattern
    Command_pattern
    Command.java
    Light.java
    LightOffCommand.java
    LightOnCommand.java
    RemoteControl.java
    TestCommandPattern.java
    Decorator_pattern
    Document_Management_System
    Implementing_Singleton_Pattern
    Observe_pattern
    Pattern
    Proxy_pattern
    Strategy_pattern
    module-info.java

TestCommandPattern.java
1 package Command_pattern;
2
3 public class TestCommandPattern {
4     public static void main(String[] args) {
5
6         // Create receiver
7         Light livingRoomLight = new Light();
8
9         // Create commands
10        Command lightOn = new LightOnCommand(livingRoomLight);
11        Command lightOff = new LightOffCommand(livingRoomLight);
12
13        // Create invoker
14        RemoteControl remote = new RemoteControl();
15
16        // Turn light ON
17        remote.setCommand(lightOn);
18        remote.pressButton();
19
20        // Turn light OFF
21        remote.setCommand(lightOff);
22    }
23 }

Command.java
1 package Command_pattern;
2
3 public interface Command {
4     void execute();
5 }
6

Problems Javadoc Declaration Console
<terminated> TestCommandPattern [Java Application] C:\eclipse\plugins\org.eclipse.just
The light is ON.
The light is OFF.
```

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:

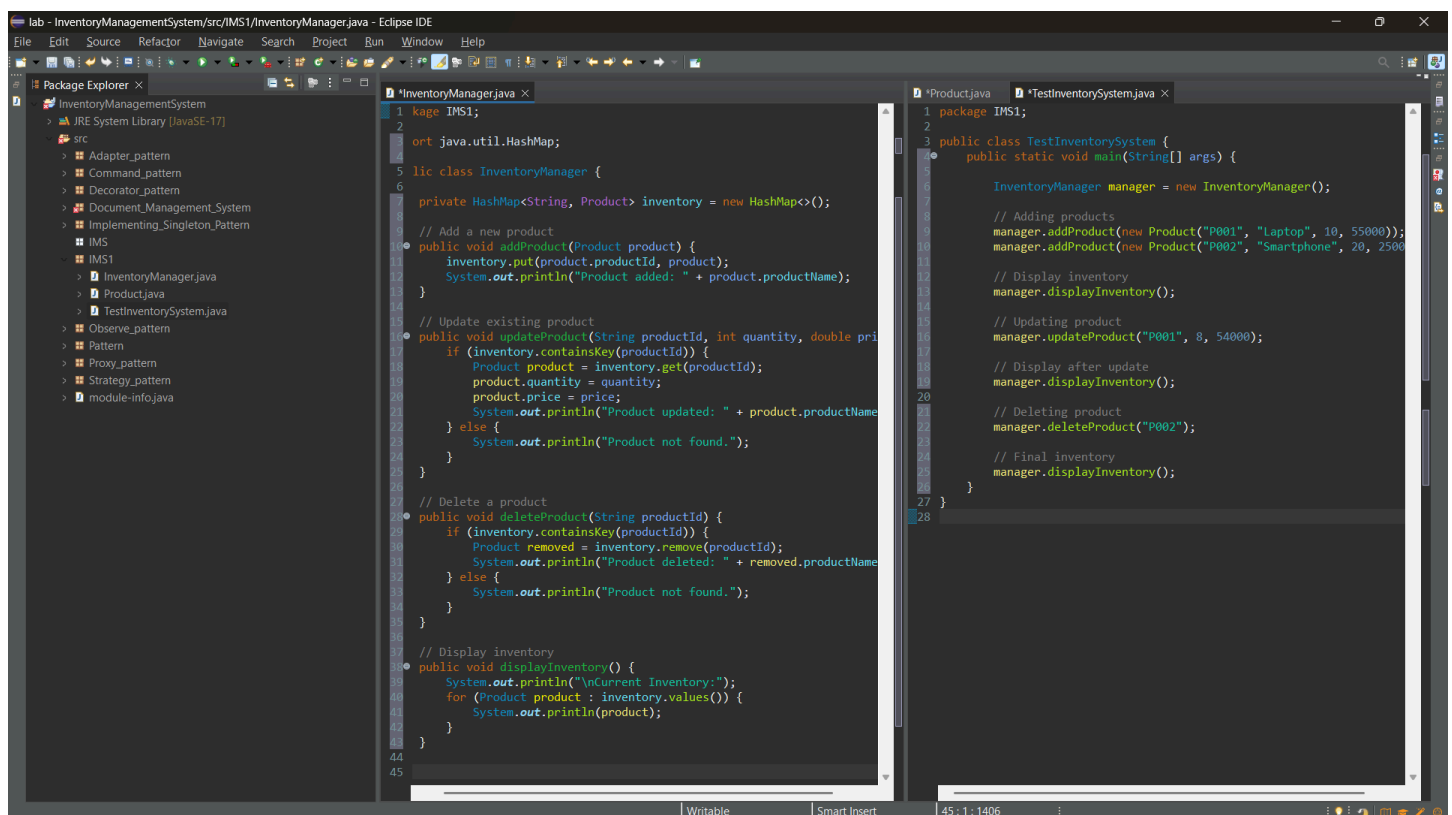
- Create a new project for the inventory management system.

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.



```
lab - InventoryManagementSystem/src/IMS1/InventoryManager.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer X
InventoryManagementSystem
  JRE System Library [JavaSE-17]
  src
    Adapter_pattern
    Command_pattern
    Decorator_pattern
    Document_Management_System
    Implementing_Singleton_Pattern
    IMS
    IMS1
    InventoryManager.java
    Product.java
    TestInventorySystem.java
  Observe_pattern
  Pattern
  Proxy_pattern
  Strategy_pattern
  module-info.java

IMS1/InventoryManager.java
1 package IMS1;
2 import java.util.HashMap;
3
4 public class InventoryManager {
5     private HashMap<String, Product> inventory = new HashMap<>();
6
7     // Add a new product
8     public void addProduct(Product product) {
9         inventory.put(product.productId, product);
10        System.out.println("Product added: " + product.productName);
11    }
12
13    // Update existing product
14    public void updateProduct(String productId, int quantity, double price) {
15        if (inventory.containsKey(productId)) {
16            Product product = inventory.get(productId);
17            product.quantity = quantity;
18            product.price = price;
19            System.out.println("Product updated: " + product.productName);
20        } else {
21            System.out.println("Product not found.");
22        }
23    }
24
25    // Delete a product
26    public void deleteProduct(String productId) {
27        if (inventory.containsKey(productId)) {
28            Product removed = inventory.remove(productId);
29            System.out.println("Product deleted: " + removed.productName);
30        } else {
31            System.out.println("Product not found.");
32        }
33    }
34
35    // Display inventory
36    public void displayInventory() {
37        System.out.println("\nCurrent Inventory:");
38        for (Product product : inventory.values()) {
39            System.out.println(product);
40        }
41    }
42
43 }
44
45

IMS1/Product.java
1 package IMS1;
2
3 public class Product {
4     private String productId;
5     private String productName;
6     private int quantity;
7     private double price;
8
9     public Product(String productId, String productName, int quantity, double price) {
10        this.productId = productId;
11        this.productName = productName;
12        this.quantity = quantity;
13        this.price = price;
14    }
15
16     public String getProductId() {
17        return productId;
18    }
19
20     public String getProductName() {
21        return productName;
22    }
23
24     public int getQuantity() {
25        return quantity;
26    }
27
28     public double getPrice() {
29        return price;
30    }
31
32     @Override
33     public String toString() {
34        return "Product [productId=" + productId + ", productName=" + productName + ", quantity=" + quantity + ", price=" + price + "]";
35    }
36 }
37
38

IMS1/TestInventorySystem.java
1 package IMS1;
2
3 public class TestInventorySystem {
4     public static void main(String[] args) {
5
6         InventoryManager manager = new InventoryManager();
7
8         // Adding products
9         manager.addProduct(new Product("P001", "Laptop", 10, 55000));
10        manager.addProduct(new Product("P002", "Smartphone", 20, 2500));
11
12        // Display inventory
13        manager.displayInventory();
14
15        // Updating product
16        manager.updateProduct("P001", 8, 54000);
17
18        // Display after update
19        manager.displayInventory();
20
21        // Deleting product
22        manager.deleteProduct("P002");
23
24        // Final inventory
25        manager.displayInventory();
26    }
27 }
28
```

The screenshot shows the Eclipse IDE with the `TestInventorySystem.java` file open. The code defines an `InventoryManager` class and a `TestInventorySystem` class. The `main` method in `TestInventorySystem` performs the following actions:

- Creates an `InventoryManager` instance.
- Adds two products: "P001" (Laptop, 10 units, \$55000) and "P002" (Smartphone, 20 units, \$25000).
- Displays the inventory.
- Updates the quantity of "P001" to 8.
- Displays the inventory again.
- Deletes "P002".
- Displays the final inventory.

The console output shows the following sequence of events:

```
<terminated> TestInventorySystem [Java Application] C:\eclipse\plugins\org.eclipse.justi.c
Product added: Laptop
Product added: Smartphone

Current Inventory:
Product ID: P001, Name: Laptop, Quantity: 10, Price: $55000.0
Product ID: P002, Name: Smartphone, Quantity: 20, Price: $25000.0
Product updated: Laptop

Current Inventory:
Product ID: P001, Name: Laptop, Quantity: 8, Price: $54000.0
Product ID: P002, Name: Smartphone, Quantity: 20, Price: $25000.0
Product deleted: Smartphone

Current Inventory:
Product ID: P001, Name: Laptop, Quantity: 8, Price: $54000.0
```

The screenshot shows the Eclipse IDE with the `Product.java` file open. The code defines a `Product` class with the following attributes and methods:

```
package IMS1;

public class Product {
    String productId;
    String productName;
    int quantity;
    double price;

    public Product(String productId, String productName, int quantity, double price) {
        this.productId = productId;
        this.productName = productName;
        this.quantity = quantity;
        this.price = price;
    }

    @Override
    public String toString() {
        return "Product ID: " + productId + ", Name: " + productName + ", Quantity: " + quantity + ", Price: $" + price;
    }
}
```

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:

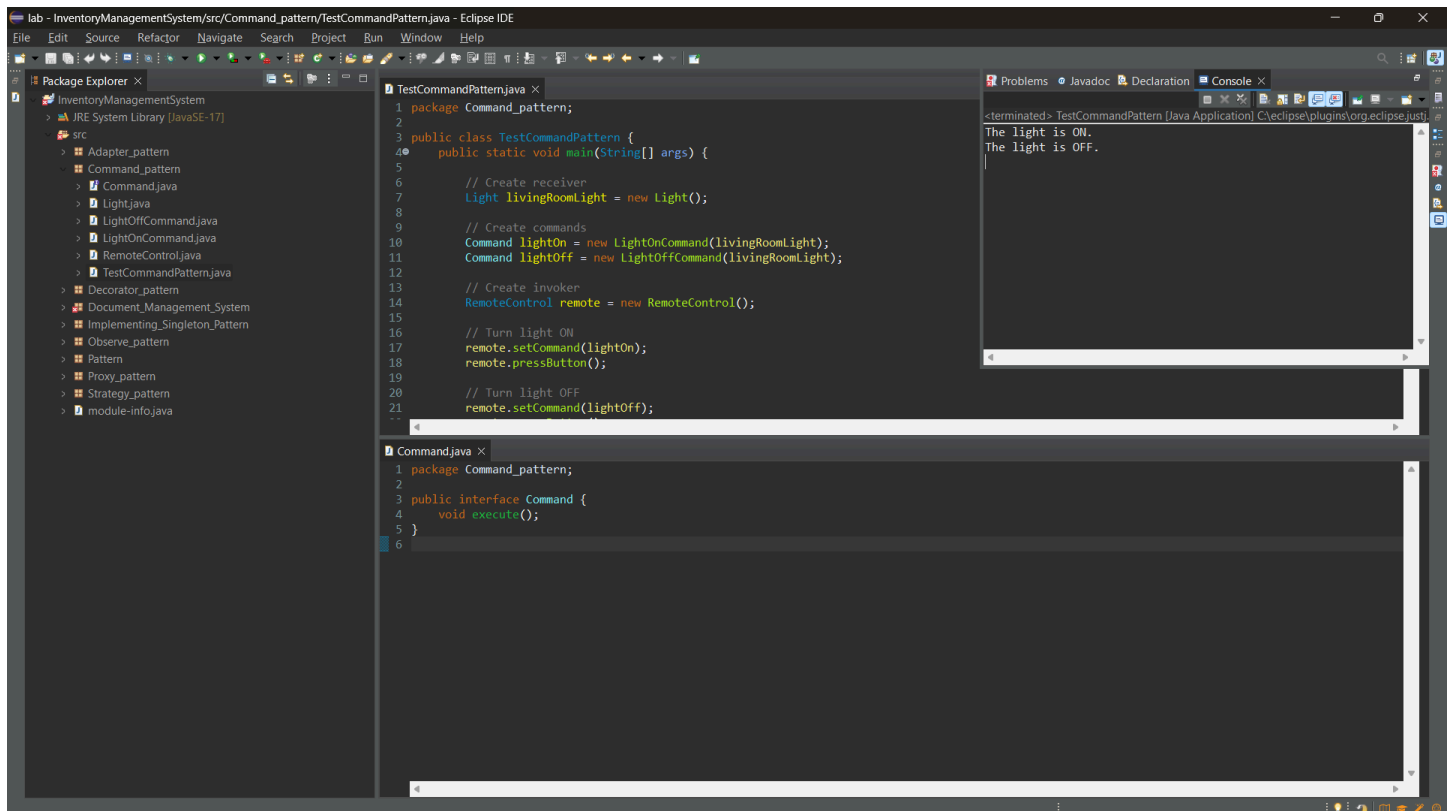
- Create a class **Product** with attributes for searching, such as **productId**, **productName**, and **category**.

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/Command_pattern/TestCommandPattern.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer
InventoryManagementSystem
  JRE System Library [JavaSE-17]
  src
    Adapter_pattern
    Command_pattern
    Command.java
    Light.java
    LightOffCommand.java
    LightOnCommand.java
    RemoteControl.java
    TestCommandPattern.java
    Decorator_pattern
    Document_Management_System
    Implementing_Singleton_Pattern
    Observe_pattern
    Pattern
    Proxy_pattern
    Strategy_pattern
    module-info.java

TestCommandPattern.java
1 package Command_pattern;
2
3 public class TestCommandPattern {
4     public static void main(String[] args) {
5
6         // Create receiver
7         Light livingRoomLight = new Light();
8
9         // Create commands
10        Command lightOn = new LightOnCommand(livingRoomLight);
11        Command lightOff = new LightOffCommand(livingRoomLight);
12
13        // Create invoker
14        RemoteControl remote = new RemoteControl();
15
16        // Turn light ON
17        remote.setCommand(lightOn);
18        remote.pressButton();
19
20        // Turn light OFF
21        remote.setCommand(lightOff);
22    }
23 }

Command.java
1 package Command_pattern;
2
3 public interface Command {
4     void execute();
5 }
6

Problems Javadoc Declaration Console
<terminated> TestCommandPattern [Java Application] C:\eclipse\plugins\org.eclipse.jdt
The light is ON.
The light is OFF.
```

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

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer

- InventoryManagementSystem
 - 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

```
1 package Strategy_pattern;
2
3 public class CreditCardPayment implements PaymentStrategy {
4
5     private String cardNumber;
6
7     public CreditCardPayment(String cardNumber) {
8         this.cardNumber = cardNumber;
9     }
10
11     @Override
12     public void pay(double amount) {
13         System.out.println("Paid $" + amount + " using Credit Card: " + cardNumber);
14     }
15 }
16
```

*PaymentStrategy.java

```
1 package Strategy_pattern;
2
3 public class PaymentContext {
4
5     private PaymentStrategy paymentStrategy;
6
7     public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
8         this.paymentStrategy = paymentStrategy;
9     }
10
11     public void payAmount(double amount) {
12         if (paymentStrategy != null) {
13             paymentStrategy.pay(amount);
14         } else {
15             System.out.println("Payment method not selected.");
16         }
17     }
18 }
19
```

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
 - 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

```
1 package Strategy_pattern;
2
3 public class PayPalPayment implements PaymentStrategy {
4
5     private String email;
6
7     public PayPalPayment(String email) {
8         this.email = email;
9     }
10
11     @Override
12     public void pay(double amount) {
13         System.out.println("Paid $" + amount + " using PayPal account: " + email);
14     }
15 }
16
```

*PaymentStrategy.java

```
1 package Strategy_pattern;
2
3 public class TestStrategyPattern {
4     public static void main(String[] args) {
5
6         PaymentContext paymentContext = new PaymentContext();
7
8         // Paying via Credit Card
9         paymentContext.setPaymentStrategy(new CreditCardPayment("1234-5678-9876-5432"));
10        paymentContext.payAmount(500.0);
11
12        // Paying via PayPal
13        paymentContext.setPaymentStrategy(new PayPalPayment("user@example.com"));
14        paymentContext.payAmount(1000.0);
15    }
16 }
17
```

Writable Smart Insert 16:1:344