Design Patterns: Part 2

SAFAR Fatima Ezzahra

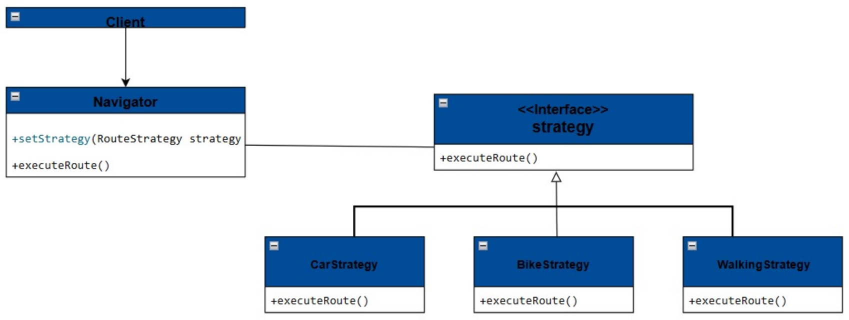
Design Patterns Course

November 25, 2025

Contents

1 Exercise 1: Flexible Navigation with Strategy Pattern

1.1 Class Diagram



Recommended size: Width = \textwidth, maintain aspect ratio

1.2 Answers to Questions

Question 1: What role does the Navigator class play?

The Navigator class plays the role of the *Context* in the Strategy pattern. It maintains a reference to a RouteStrategy object and delegates the route calculation to the current strategy.

Question 2: Why does Navigator depend on the RouteStrategy interface?

Navigator depends on the RouteStrategy interface (not concrete implementations) to achieve:

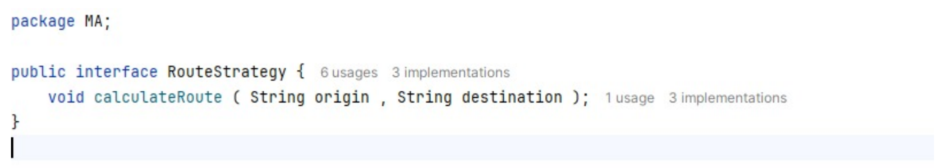
- **Loose coupling**: Navigator doesn't need to know about specific strategy implementations
- **Flexibility**: Strategies can be changed at runtime
- **Extensibility**: New strategies can be added without modifying Navigator

Question 3: Which SOLID principles are applied in this design?

- **Single Responsibility Principle (SRP)**: Each strategy class has one responsibility - implementing a specific routing algorithm
- **Open/Closed Principle (OCP)**: The system is open for extension (new strategies) but closed for modification (Navigator doesn't change)
- **Liskov Substitution Principle (LSP)**: Any RouteStrategy implementation can be substituted without affecting Navigator's behavior
- **Dependency Inversion Principle (DIP)**: Navigator depends on the RouteStrategy abstraction, not concrete implementations

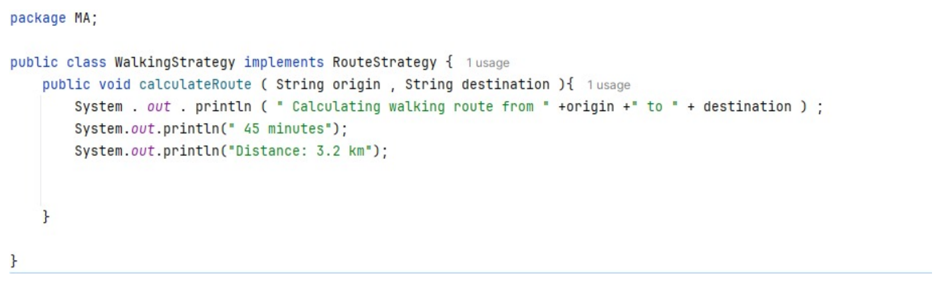
1.3 Java Implementation

1.3.1 RouteStrategy.java



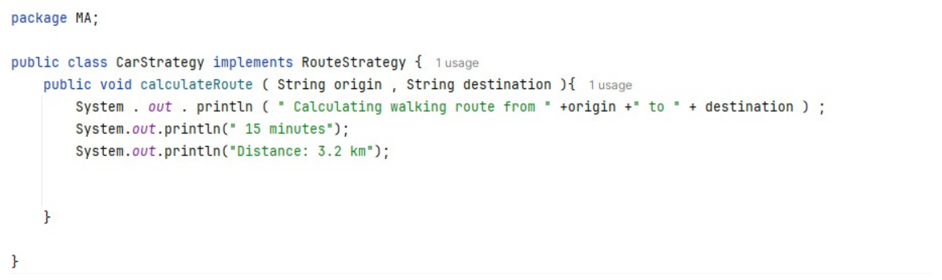
```java
package MA;

public interface RouteStrategy {
    void calculateRoute ( String origin , String destination );
}
```

1.3.2 WalkingStrategy.java



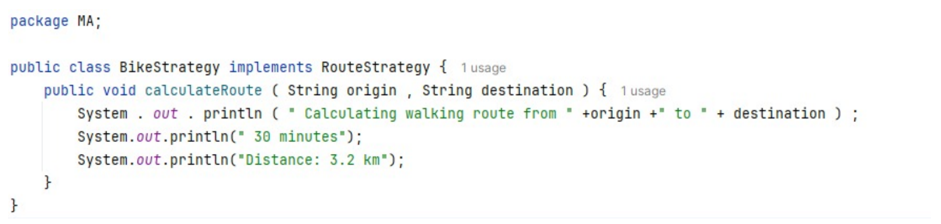
```java
package MA;

public class WalkingStrategy implements RouteStrategy {
    public void calculateRoute ( String origin , String destination ){
        System . out . println ( " Calculating walking route from " +origin +" to " + destination ) ;
        System.out.println(" 45 minutes");
        System.out.println("Distance: 3.2 km");


    }

}
```

1.3.3 CarStrategy.java



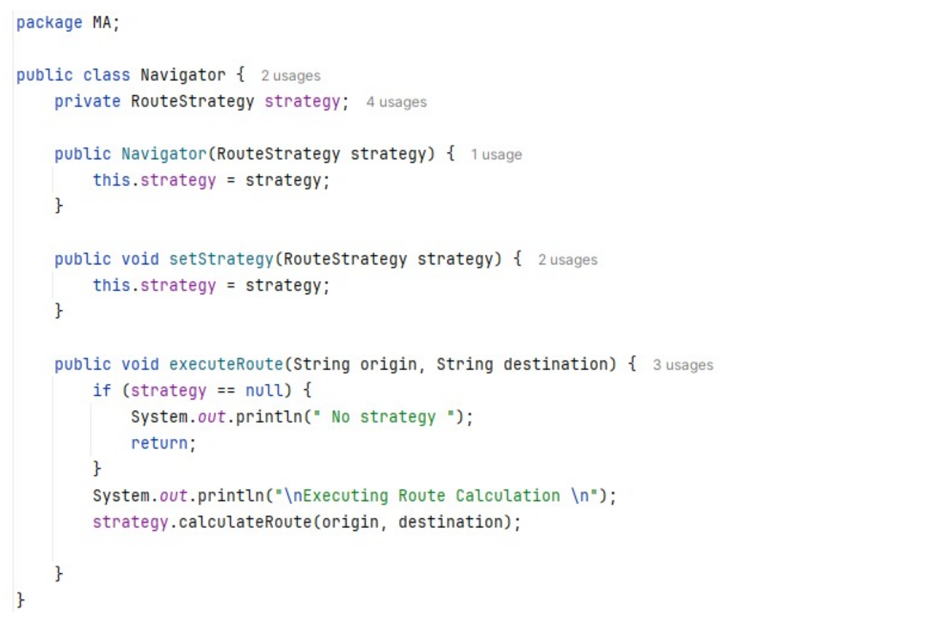
```java
package MA;

public class CarStrategy implements RouteStrategy {
    public void calculateRoute ( String origin , String destination ){
        System . out . println ( " Calculating walking route from " +origin +" to " + destination ) ;
        System.out.println(" 15 minutes");
        System.out.println("Distance: 3.2 km");


    }

}
```

1.3.4 BikeStrategy.java



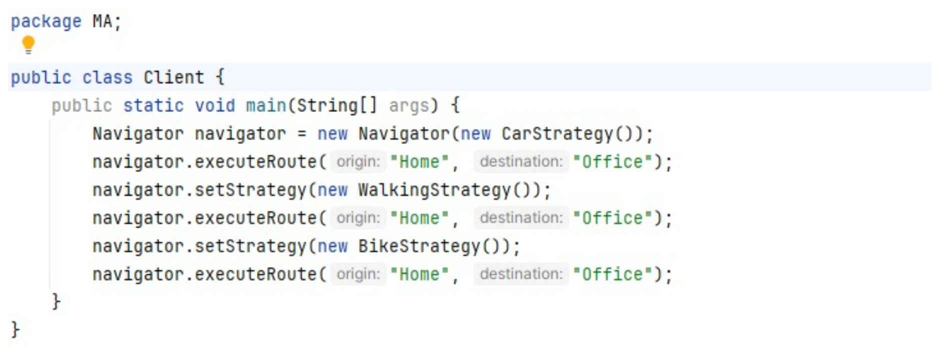
```
package MA;

public class BikeStrategy implements RouteStrategy {  1 usage
    public void calculateRoute ( String origin , String destination ) {  1 usage
        System . out . println ( " Calculating walking route from " +origin +" to " + destination ) ;
        System.out.println(" 30 minutes");
        System.out.println("Distance: 3.2 km");
    }
}
```

1.3.5 Navigator.java



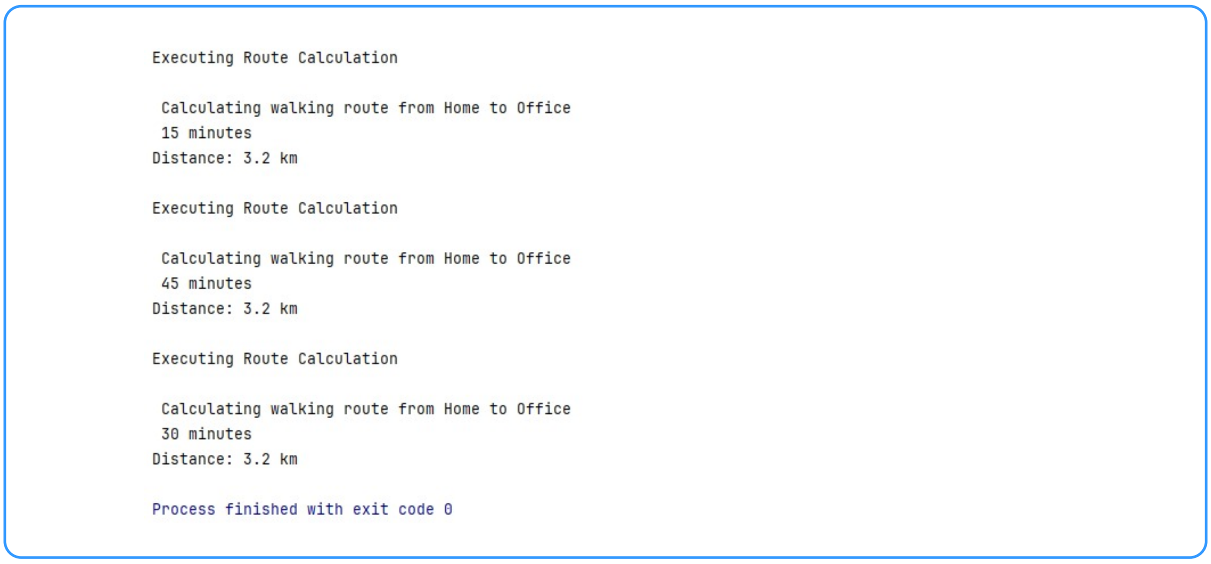
```
package MA;

public class Navigator {  2 usages
    private RouteStrategy strategy;  4 usages

    public Navigator(RouteStrategy strategy) {  1 usage
        this.strategy = strategy;
    }

    public void setStrategy(RouteStrategy strategy) {  2 usages
        this.strategy = strategy;
    }

    public void executeRoute(String origin, String destination) {  3 usages
        if (strategy == null) {
            System.out.println(" No strategy ");
            return;
        }
        System.out.println("\nExecuting Route Calculation \n");
        strategy.calculateRoute(origin, destination);

    }
}
```

1.3.6 Client.java



```
package MA;

public class Client {
    public static void main(String[] args) {
        Navigator navigator = new Navigator(new CarStrategy());
        navigator.executeRoute( origin: "Home",  destination: "Office");
        navigator.setStrategy(new WalkingStrategy());
        navigator.executeRoute( origin: "Home",  destination: "Office");
        navigator.setStrategy(new BikeStrategy());
        navigator.executeRoute( origin: "Home",  destination: "Office");
    }
}
```

1.3.7 Output



```
Executing Route Calculation

 Calculating walking route from Home to Office
 15 minutes
Distance: 3.2 km

Executing Route Calculation

 Calculating walking route from Home to Office
 45 minutes
Distance: 3.2 km

Executing Route Calculation

 Calculating walking route from Home to Office
 30 minutes
Distance: 3.2 km

Process finished with exit code 0
```

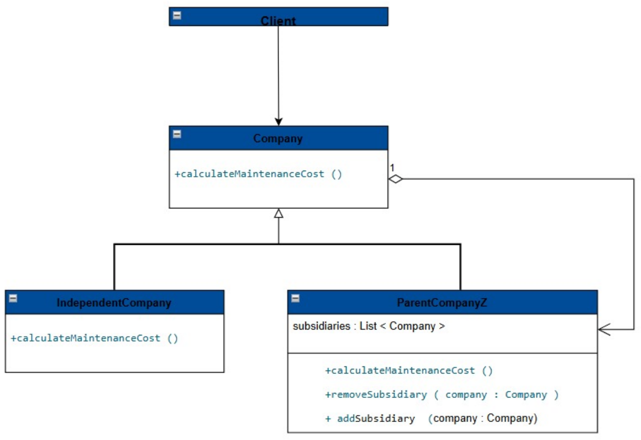
2 Exercise 2: Vehicle Maintenance with Composite Pattern

2.1 Design Pattern Identification

The **Composite Pattern** is best suited for this problem because:

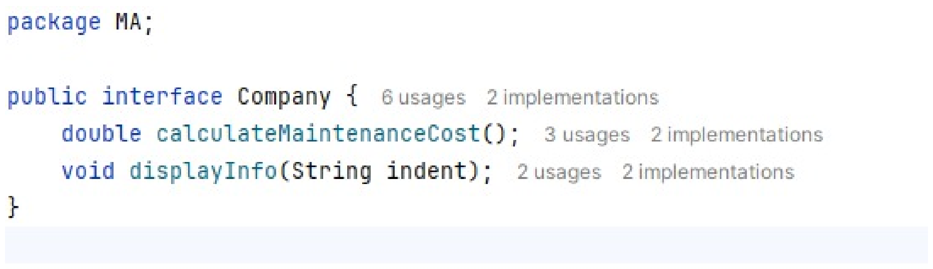
- We need to represent a tree structure (parent companies containing independent companies)
- We want to treat individual objects (independent companies) and compositions (parent companies) uniformly
- Operations (calculating maintenance cost) should work the same way on both leaf and composite nodes

2.2 Class Diagram

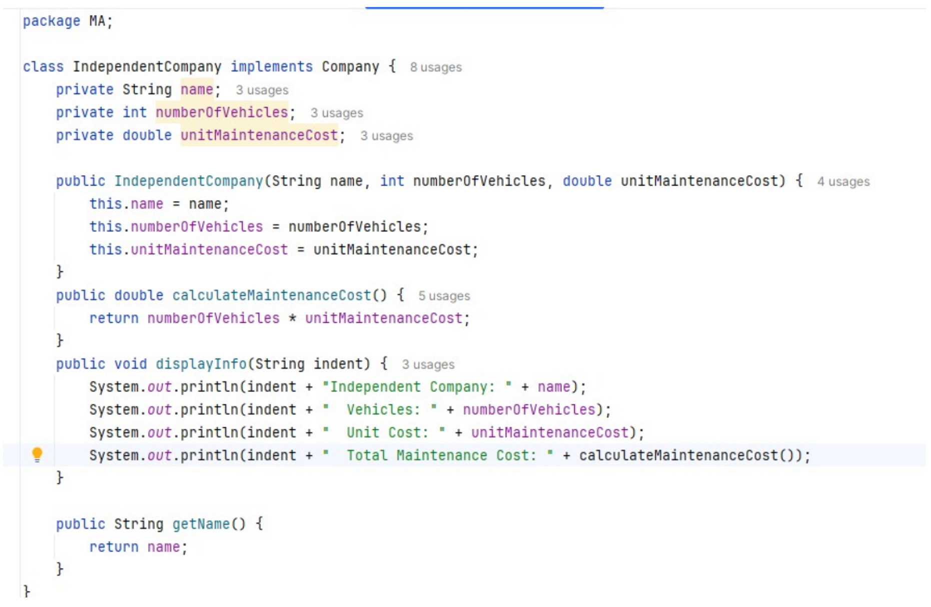

Client
Company
+calculateMaintenanceCost ()
1
IndependentCompany
+calculateMaintenanceCost ()
ParentCompanyZ
subsidiaries : List < Company >
+calculateMaintenanceCost ()
+removeSubsidiary (company : Company)
+ addSubsidiary (company : Company)

2.3 Java Implementation

2.3.1 Company.java

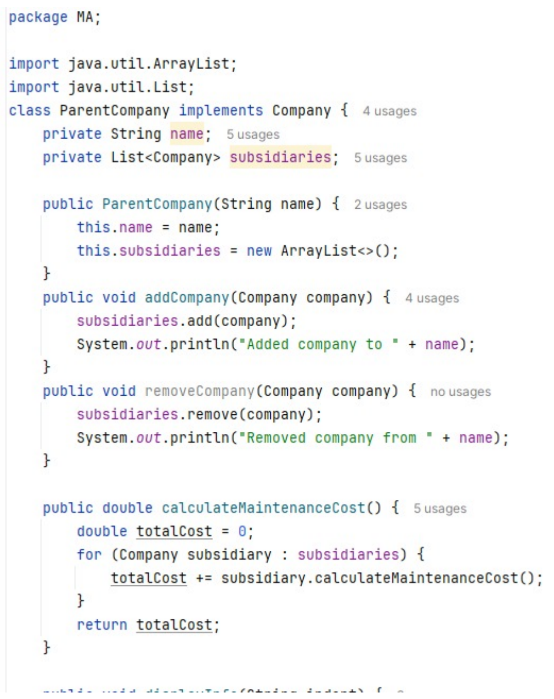

```java
package MA;

public interface Company {  6 usages  2 implementations
    double calculateMaintenanceCost();  3 usages  2 implementations
    void displayInfo(String indent);  2 usages  2 implementations
}
```

2.3.2 IndependentCompany.java

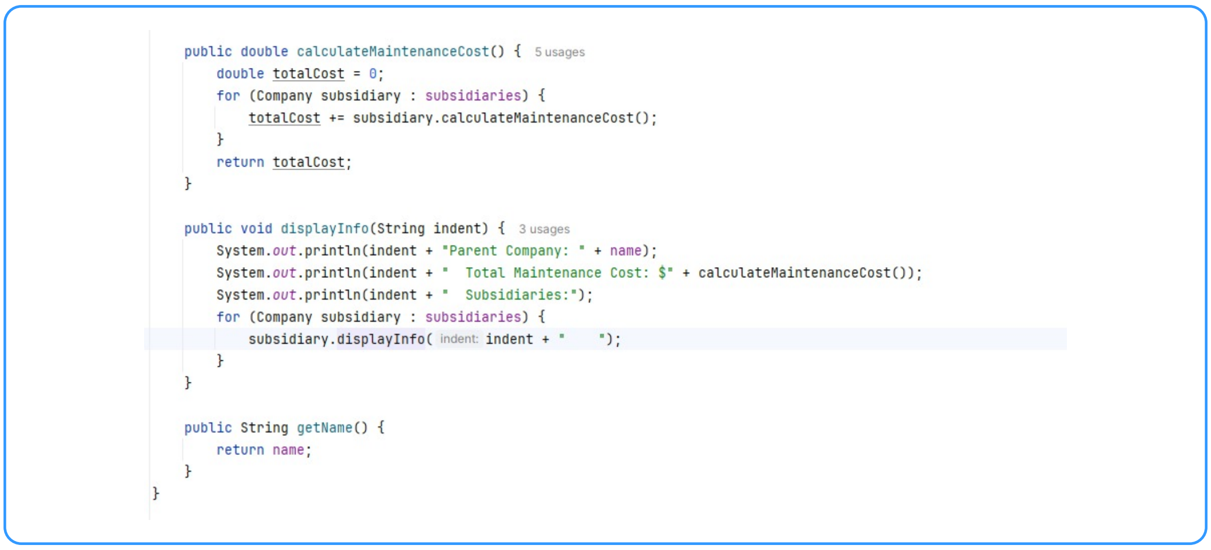


```java
package MA;

class IndependentCompany implements Company {  8 usages
    private String name;  3 usages
    private int numberOfVehicles;  3 usages
    private double unitMaintenanceCost;  3 usages

    public IndependentCompany(String name, int numberOfVehicles, double unitMaintenanceCost) {  4 usages
        this.name = name;
        this.numberOfVehicles = numberOfVehicles;
        this.unitMaintenanceCost = unitMaintenanceCost;
    }
    public double calculateMaintenanceCost() {  5 usages
        return numberOfVehicles * unitMaintenanceCost;
    }
    public void displayInfo(String indent) {  3 usages
        System.out.println(indent + "Independent Company: " + name);
        System.out.println(indent + "  Vehicles: " + numberOfVehicles);
        System.out.println(indent + "  Unit Cost: " + unitMaintenanceCost);
        System.out.println(indent + "  Total Maintenance Cost: " + calculateMaintenanceCost());
    }

    public String getName() {
        return name;
    }
}
```

2.3.3 ParentCompany.java

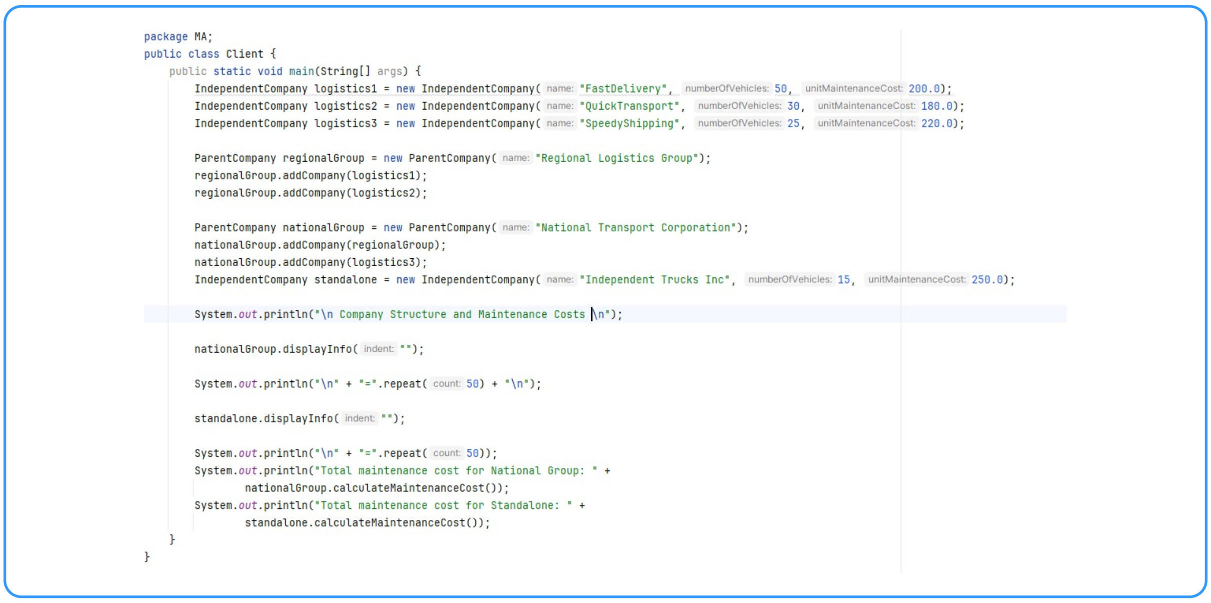


```java
package MA;

import java.util.ArrayList;
import java.util.List;
class ParentCompany implements Company {  4 usages
    private String name;  5 usages
    private List<Company> subsidiaries;  5 usages

    public ParentCompany(String name) {  2 usages
        this.name = name;
        this.subsidiaries = new ArrayList<>();
    }
    public void addCompany(Company company) {  4 usages
        subsidiaries.add(company);
        System.out.println("Added company to " + name);
    }
    public void removeCompany(Company company) {  no usages
        subsidiaries.remove(company);
        System.out.println("Removed company from " + name);
    }

    public double calculateMaintenanceCost() {  5 usages
        double totalCost = 0;
        for (Company subsidiary : subsidiaries) {
            totalCost += subsidiary.calculateMaintenanceCost();
        }
        return totalCost;
    }
```



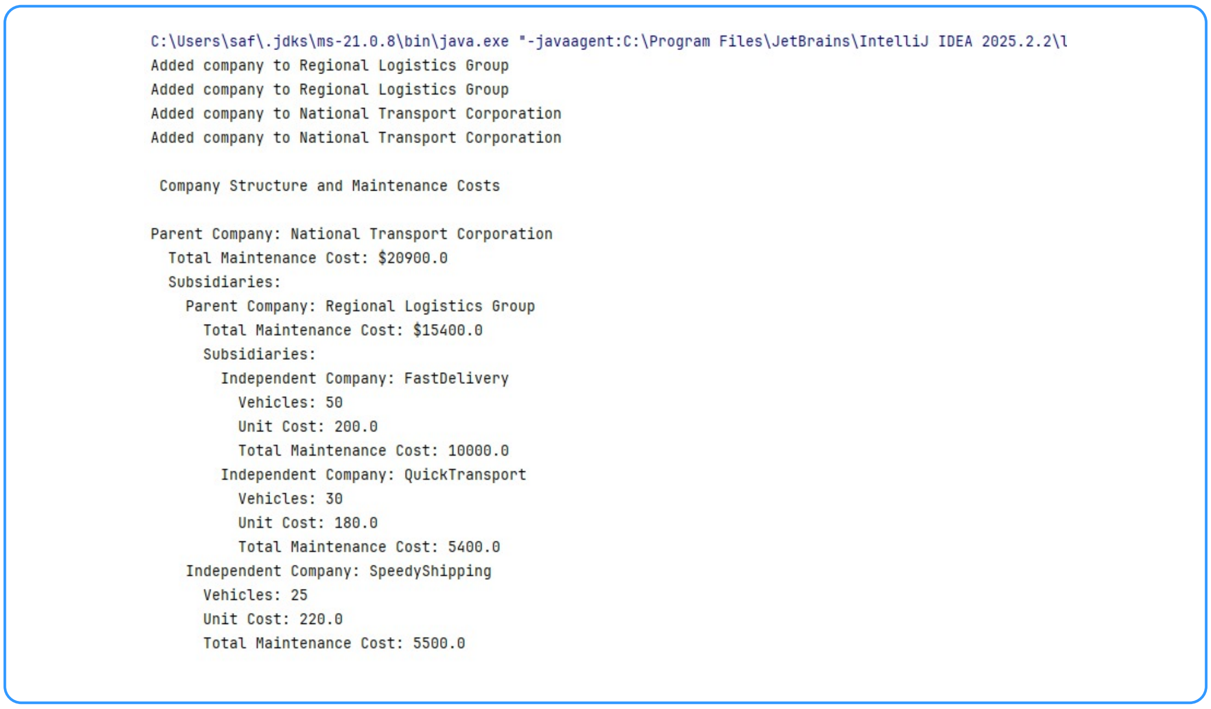
```java
    public double calculateMaintenanceCost() {  5 usages
        double totalCost = 0;
        for (Company subsidiary : subsidiaries) {
            totalCost += subsidiary.calculateMaintenanceCost();
        }
        return totalCost;
    }

    public void displayInfo(String indent) {  3 usages
        System.out.println(indent + "Parent Company: " + name);
        System.out.println(indent + "  Total Maintenance Cost: $" + calculateMaintenanceCost());
        System.out.println(indent + "  Subsidiaries:");
        for (Company subsidiary : subsidiaries) {
            subsidiary.displayInfo( indent: indent + "    ");
        }
    }

    public String getName() {
        return name;
    }
}
```

2.3.4 Client.java

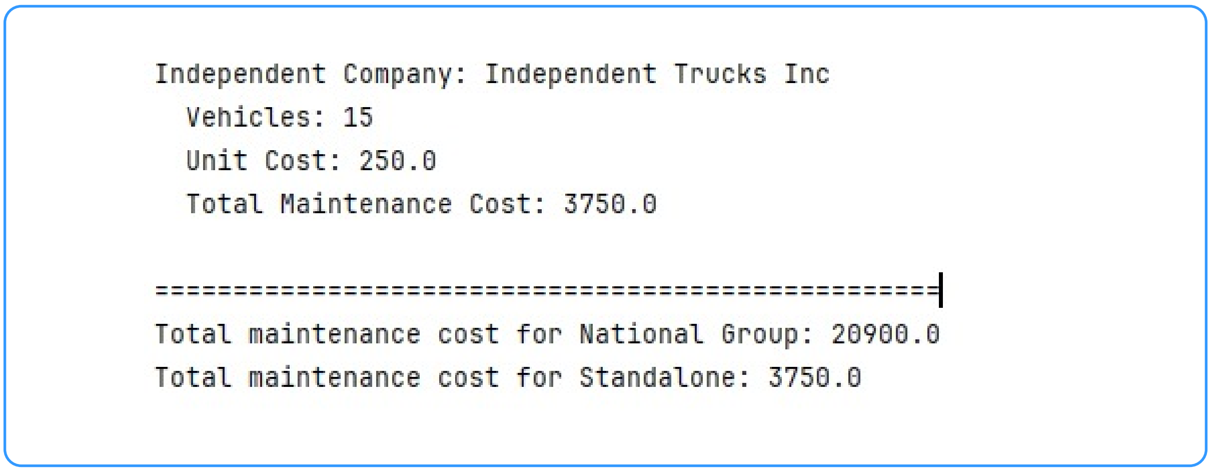


```java
package MA;
public class Client {
    public static void main(String[] args) {
        IndependentCompany logistics1 = new IndependentCompany( name: "FastDelivery",  numberOfVehicles: 50,  unitMaintenanceCost: 200.0);
        IndependentCompany logistics2 = new IndependentCompany( name: "QuickTransport",  numberOfVehicles: 30,  unitMaintenanceCost: 180.0);
        IndependentCompany logistics3 = new IndependentCompany( name: "SpeedyShipping",  numberOfVehicles: 25,  unitMaintenanceCost: 220.0);

        ParentCompany regionalGroup = new ParentCompany( name: "Regional Logistics Group");
        regionalGroup.addCompany(logistics1);
        regionalGroup.addCompany(logistics2);

        ParentCompany nationalGroup = new ParentCompany( name: "National Transport Corporation");
        nationalGroup.addCompany(regionalGroup);
        nationalGroup.addCompany(logistics3);
        IndependentCompany standalone = new IndependentCompany( name: "Independent Trucks Inc",  numberOfVehicles: 15,  unitMaintenanceCost: 250.0);

        System.out.println("\n Company Structure and Maintenance Costs \n");

        nationalGroup.displayInfo( indent: "");

        System.out.println("\n" + "=".repeat( count: 50) + "\n");

        standalone.displayInfo( indent: "");

        System.out.println("\n" + "=".repeat( count: 50));
        System.out.println("Total maintenance cost for National Group: " +
                nationalGroup.calculateMaintenanceCost());
        System.out.println("Total maintenance cost for Standalone: " +
                standalone.calculateMaintenanceCost());
    }
}
```

2.3.5 Output



```
C:\Users\saf\.jdks\ms-21.0.8\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2025.2.2\l
Added company to Regional Logistics Group
Added company to Regional Logistics Group
Added company to National Transport Corporation
Added company to National Transport Corporation

 Company Structure and Maintenance Costs

Parent Company: National Transport Corporation
  Total Maintenance Cost: $20900.0
  Subsidiaries:
    Parent Company: Regional Logistics Group
      Total Maintenance Cost: $15400.0
      Subsidiaries:
        Independent Company: FastDelivery
          Vehicles: 50
          Unit Cost: 200.0
          Total Maintenance Cost: 10000.0
        Independent Company: QuickTransport
          Vehicles: 30
          Unit Cost: 180.0
          Total Maintenance Cost: 5400.0
    Independent Company: SpeedyShipping
      Vehicles: 25
      Unit Cost: 220.0
      Total Maintenance Cost: 5500.0
```



```
Independent Company: Independent Trucks Inc
  Vehicles: 15
  Unit Cost: 250.0
  Total Maintenance Cost: 3750.0

==================================================
Total maintenance cost for National Group: 20900.0
Total maintenance cost for Standalone: 3750.0
```

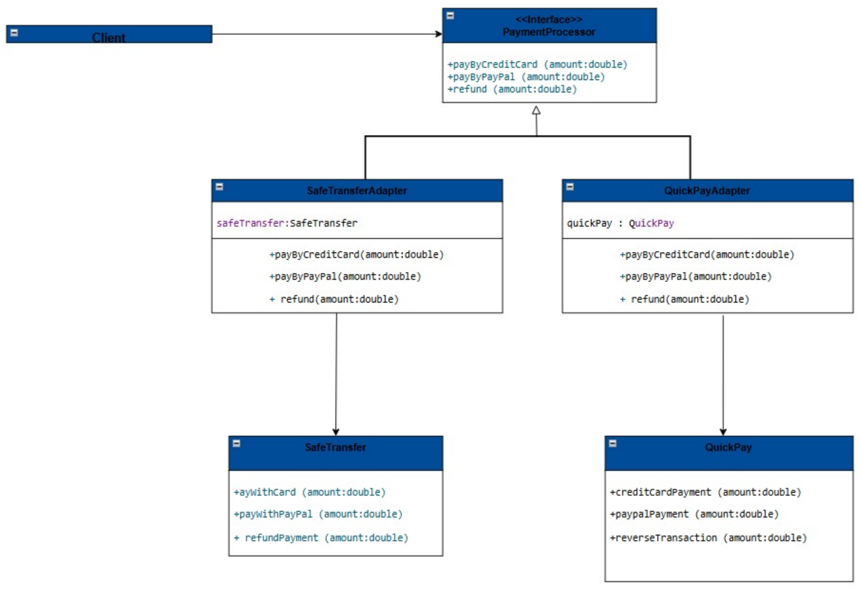
3 Exercise 3: Payment Integration with Adapter Pattern

3.1 Design Pattern Identification

The **Adapter Pattern** should be used because:

- We need to integrate third-party services with incompatible interfaces
- We cannot modify the existing payment services (QuickPay and SafeTransfer)
- We want to provide a uniform interface (PaymentProcessor) to the client
- The adapter translates calls from the target interface to the adaptee interface

3.2 Class Diagram

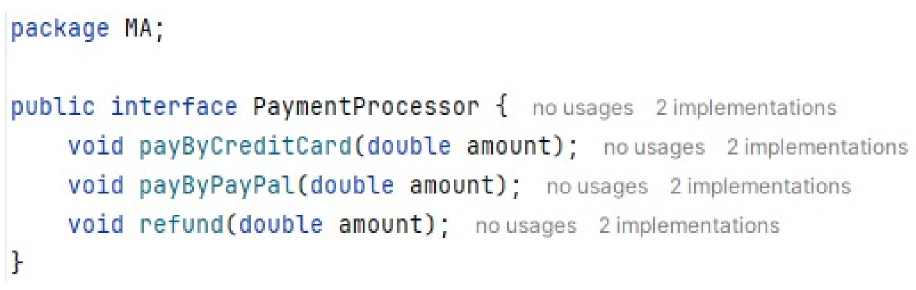

Client
<<Interface>>
PaymentProcessor
+payByCreditCard (amount:double)
+payByPayPal (amount:double)
+refund (amount:double)
SafeTransferAdapter
safeTransfer:SafeTransfer
+payByCreditCard(amount:double)
+payByPayPal(amount:double)
+ refund(amount:double)
QuickPayAdapter
quickPay : QuickPay
+payByCreditCard(amount:double)
+payByPayPal(amount:double)
+ refund(amount:double)
SafeTransfer
+aywithCard (amount:double)
+paywithPayPal (amount:double)
+ refundPayment (amount:double)
QuickPay
+creditCardPayment (amount:double)
+paypalPayment (amount:double)
+reverseTransaction (amount:double)

3.3 Participants

- **Target (PaymentProcessor)**: The interface expected by the client
- **Adapters (QuickPayAdapter, SafeTransferAdapter)**: Convert the target interface to the adaptee's interface
- **Adaptees (QuickPay, SafeTransfer)**: The existing third-party services with incompatible interfaces
- **Client**: Uses the PaymentProcessor interface

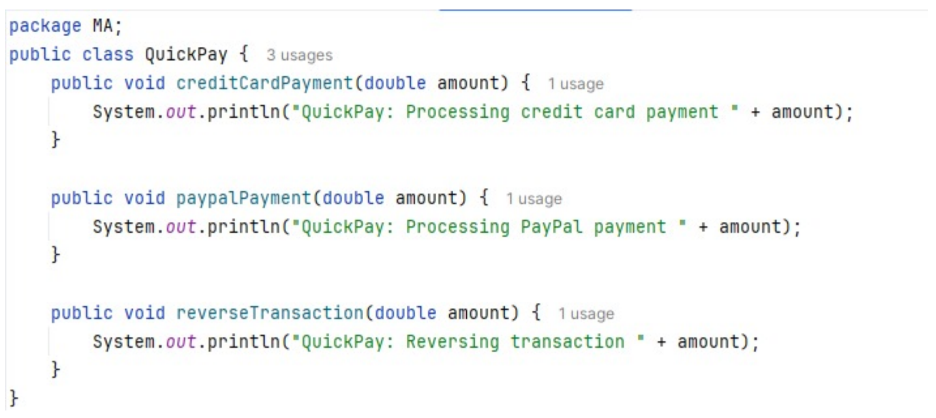
3.4 Java Implementation

3.4.1 PaymentProcessor.java



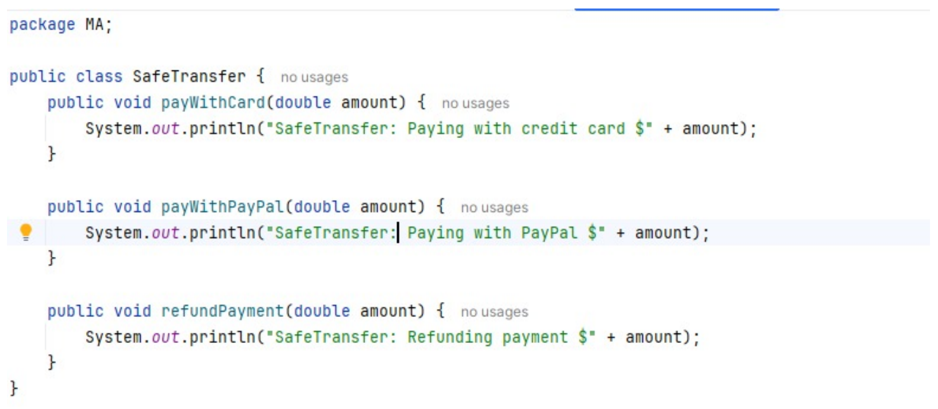
```java
package MA;

public interface PaymentProcessor {  no usages  2 implementations
    void payByCreditCard(double amount);  no usages  2 implementations
    void payByPayPal(double amount);  no usages  2 implementations
    void refund(double amount);  no usages  2 implementations
}
```

3.4.2 QuickPay.java



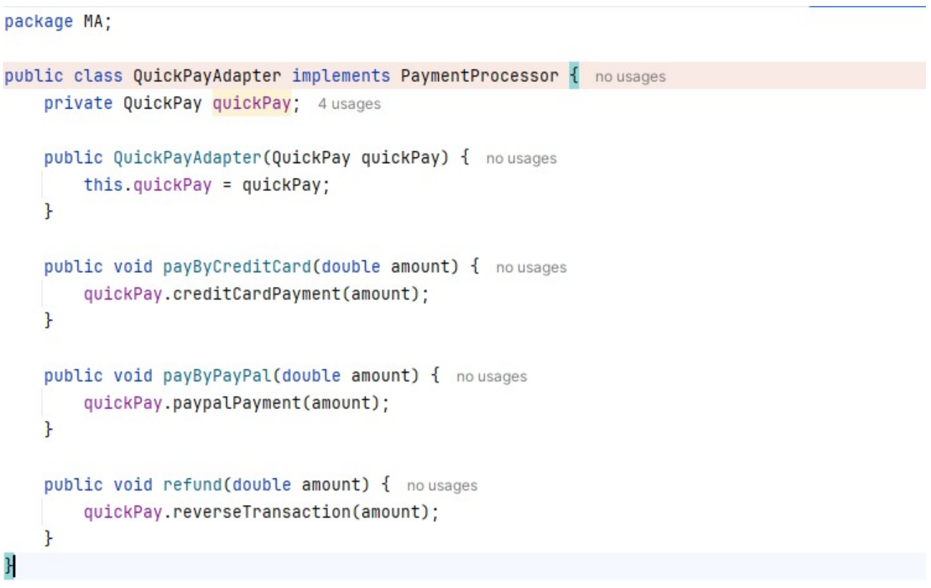
```java
package MA;
public class QuickPay {  3 usages
    public void creditCardPayment(double amount) {  1 usage
        System.out.println("QuickPay: Processing credit card payment " + amount);
    }

    public void paypalPayment(double amount) {  1 usage
        System.out.println("QuickPay: Processing PayPal payment " + amount);
    }

    public void reverseTransaction(double amount) {  1 usage
        System.out.println("QuickPay: Reversing transaction " + amount);
    }
}
```

3.4.3 SafeTransfer.java



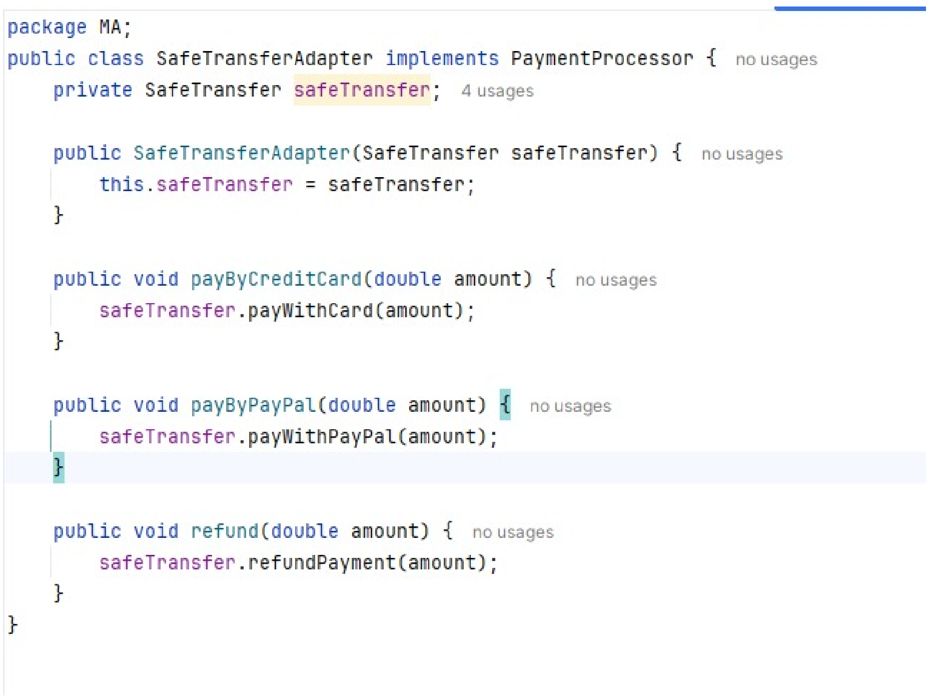
```java
package MA;

public class SafeTransfer {  no usages
    public void payWithCard(double amount) {  no usages
        System.out.println("SafeTransfer: Paying with credit card $" + amount);
    }

    public void payWithPayPal(double amount) {  no usages
        System.out.println("SafeTransfer: Paying with PayPal $" + amount);
    }

    public void refundPayment(double amount) {  no usages
        System.out.println("SafeTransfer: Refunding payment $" + amount);
    }
}
```

3.4.4 QuickPayAdapter.java



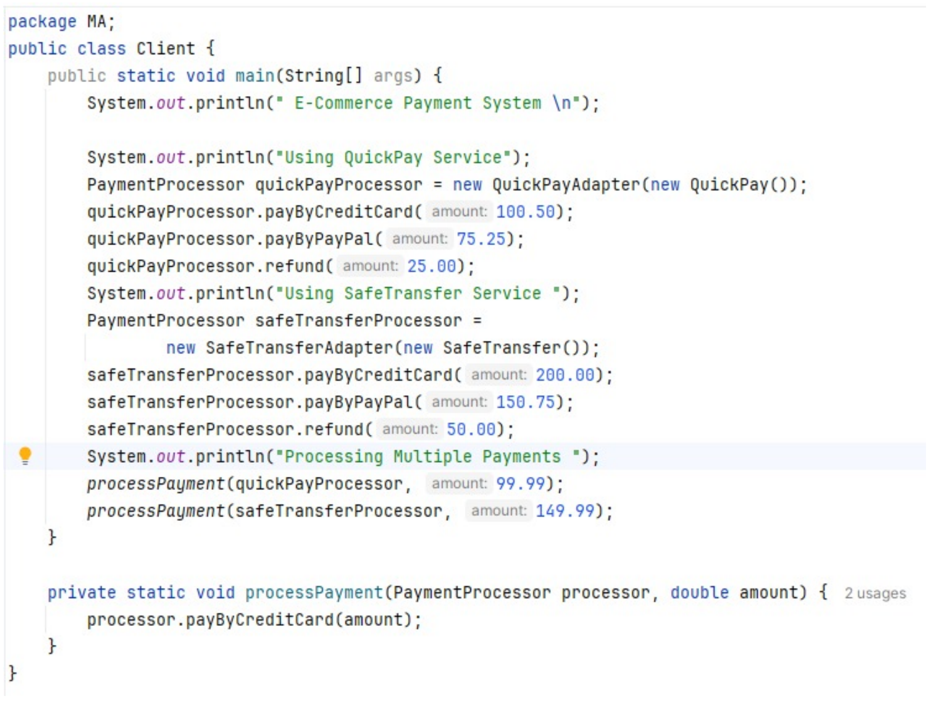
```
package MA;

public class QuickPayAdapter implements PaymentProcessor {  no usages
    private QuickPay quickPay;  4 usages

    public QuickPayAdapter(QuickPay quickPay) {  no usages
        this.quickPay = quickPay;
    }

    public void payByCreditCard(double amount) {  no usages
        quickPay.creditCardPayment(amount);
    }

    public void payByPayPal(double amount) {  no usages
        quickPay.paypalPayment(amount);
    }

    public void refund(double amount) {  no usages
        quickPay.reverseTransaction(amount);
    }
}
```

3.4.5 SafeTransferAdapter.java



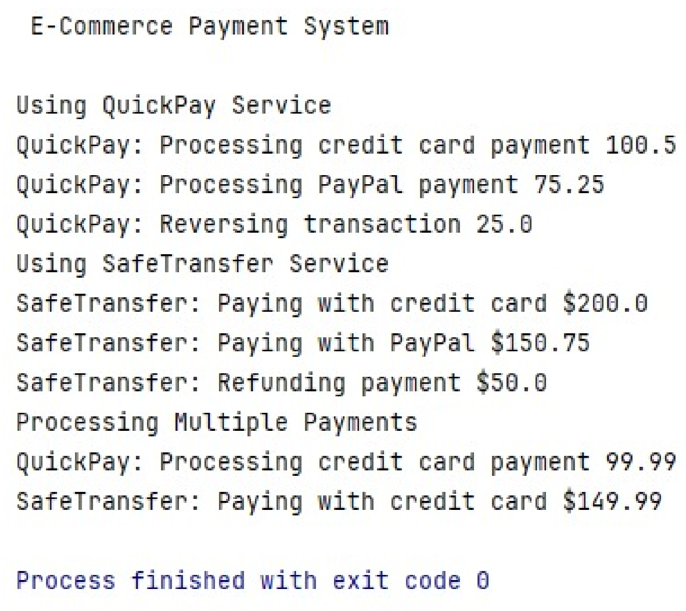
```
package MA;
public class SafeTransferAdapter implements PaymentProcessor {  no usages
    private SafeTransfer safeTransfer;  4 usages

    public SafeTransferAdapter(SafeTransfer safeTransfer) {  no usages
        this.safeTransfer = safeTransfer;
    }

    public void payByCreditCard(double amount) {  no usages
        safeTransfer.payWithCard(amount);
    }

    public void payByPayPal(double amount) {  no usages
        safeTransfer.payWithPayPal(amount);
    }

    public void refund(double amount) {  no usages
        safeTransfer.refundPayment(amount);
    }
}
```

3.4.6 Client.java



```java
package MA;
public class Client {
    public static void main(String[] args) {
        System.out.println(" E-Commerce Payment System \n");

        System.out.println("Using QuickPay Service");
        PaymentProcessor quickPayProcessor = new QuickPayAdapter(new QuickPay());
        quickPayProcessor.payByCreditCard( amount: 100.50);
        quickPayProcessor.payByPayPal( amount: 75.25);
        quickPayProcessor.refund( amount: 25.00);
        System.out.println("Using SafeTransfer Service ");
        PaymentProcessor safeTransferProcessor =
                new SafeTransferAdapter(new SafeTransfer());
        safeTransferProcessor.payByCreditCard( amount: 200.00);
        safeTransferProcessor.payByPayPal( amount: 150.75);
        safeTransferProcessor.refund( amount: 50.00);
        System.out.println("Processing Multiple Payments ");
        processPayment(quickPayProcessor,  amount: 99.99);
        processPayment(safeTransferProcessor,  amount: 149.99);
    }

    private static void processPayment(PaymentProcessor processor, double amount) {  2 usages
        processor.payByCreditCard(amount);
    }
}
```

3.4.7 Output



```
 E-Commerce Payment System

Using QuickPay Service
QuickPay: Processing credit card payment 100.5
QuickPay: Processing PayPal payment 75.25
QuickPay: Reversing transaction 25.0
Using SafeTransfer Service
SafeTransfer: Paying with credit card $200.0
SafeTransfer: Paying with PayPal $150.75
SafeTransfer: Refunding payment $50.0
Processing Multiple Payments
QuickPay: Processing credit card payment 99.99
SafeTransfer: Paying with credit card $149.99

Process finished with exit code 0
```

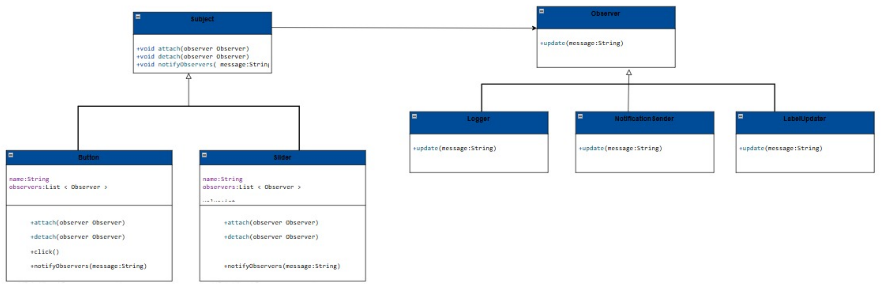
4 Exercise 4: GUI Dashboard with Observer Pattern

4.1 Design Pattern Identification

The **Observer Pattern** is most suitable for this scenario because:

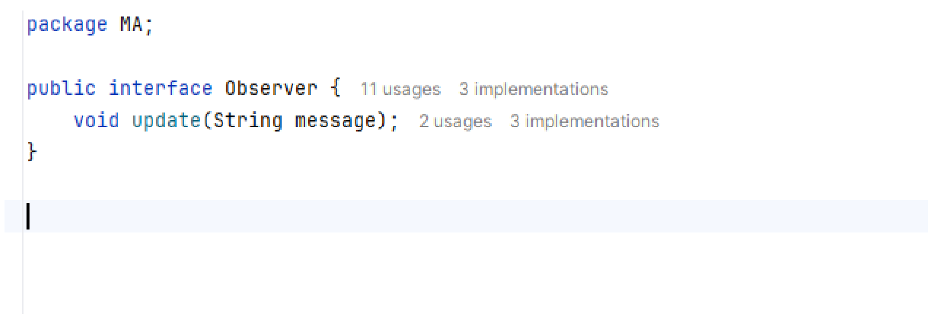
- Multiple components need to be notified when GUI elements change state
- The relationship is one-to-many: one subject (button/slider) to many observers
- Observers should react independently and automatically to state changes
- The system needs loose coupling between GUI elements and their observers
- We want to add/remove observers dynamically at runtime

4.2 Class Diagram


Subject
+void attach(observer Observer)
+void detach(observer Observer)
+void notifyObservers(message:Strin
Observer
+update(message:String)
Button
name:String
observers:List < Observer >
+attach(observer Observer)
+detach(observer Observer)
+click()
+notifyObservers(message:String)
Slider
name:String
observers:List < Observer >
+attach(observer Observer)
+detach(observer Observer)
+notifyObservers(message:String)
Logger
+update(message:String)
NotificationSender
+update(message:String)
LabelUpdater
+update(message:String)

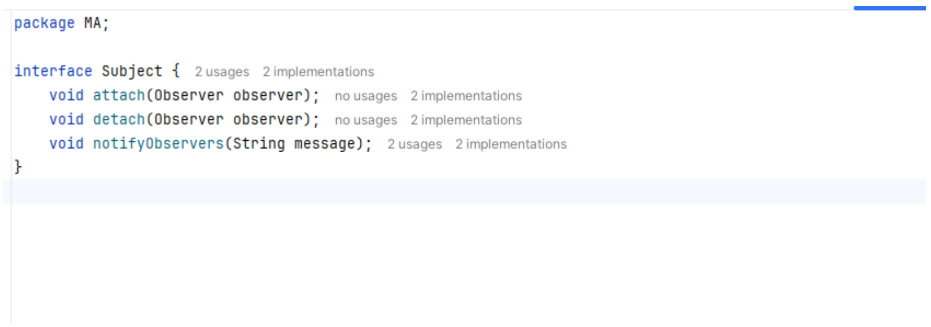
4.3 Java Implementation

4.3.1 Observer.java



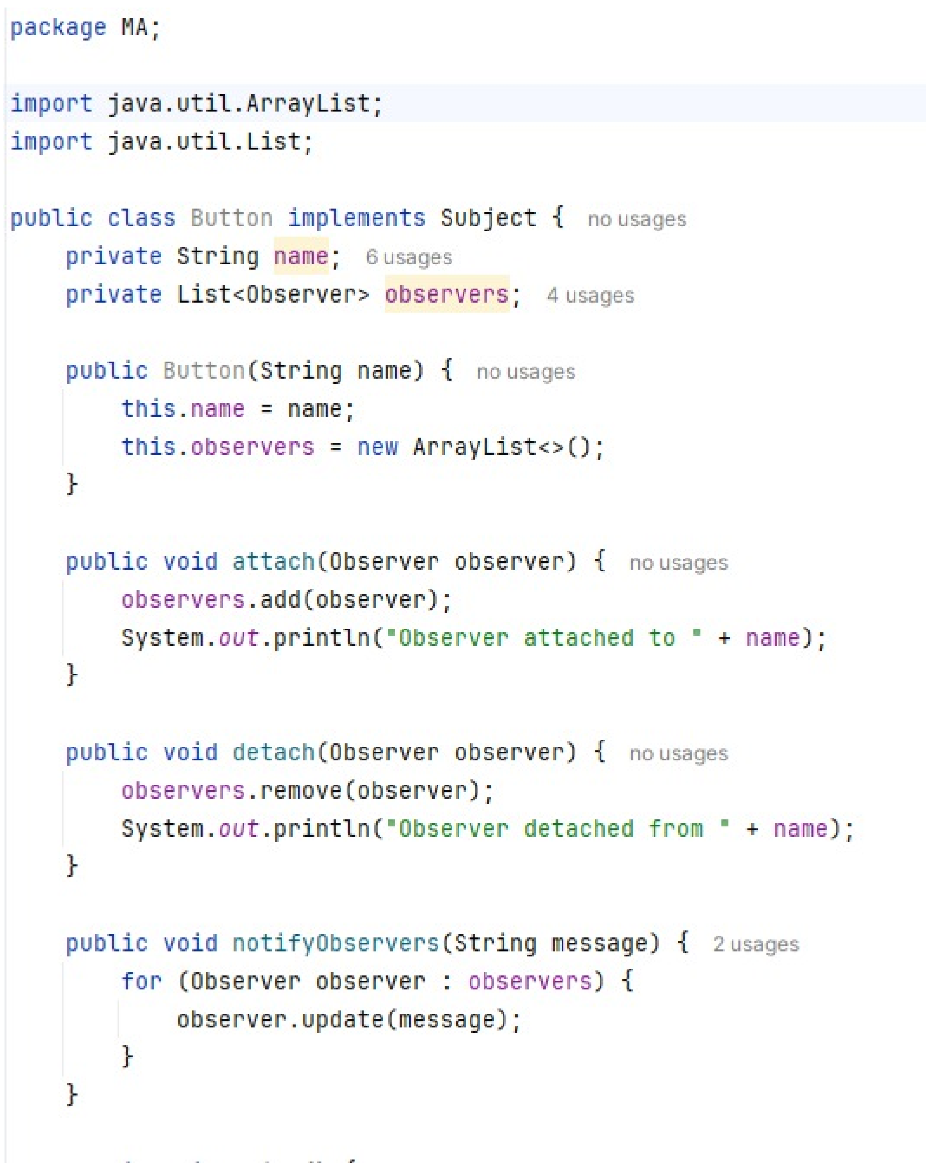
```
package MA;

public interface Observer {  11 usages   3 implementations
    void update(String message);  2 usages   3 implementations
}
```

4.3.2 Subject.java

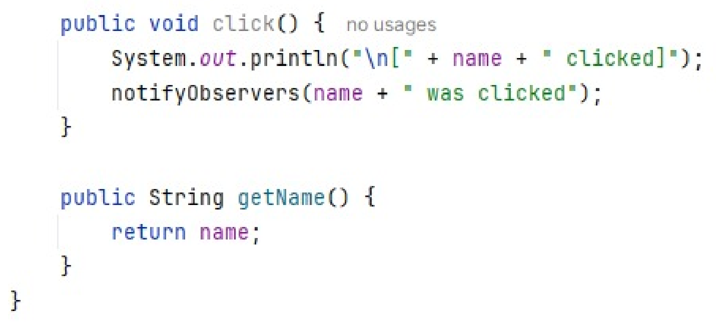


```
package MA;

interface Subject {  2 usages  2 implementations
    void attach(Observer observer);  no usages  2 implementations
    void detach(Observer observer);  no usages  2 implementations
    void notifyObservers(String message);  2 usages  2 implementations
}
```

4.3.3 Button.java

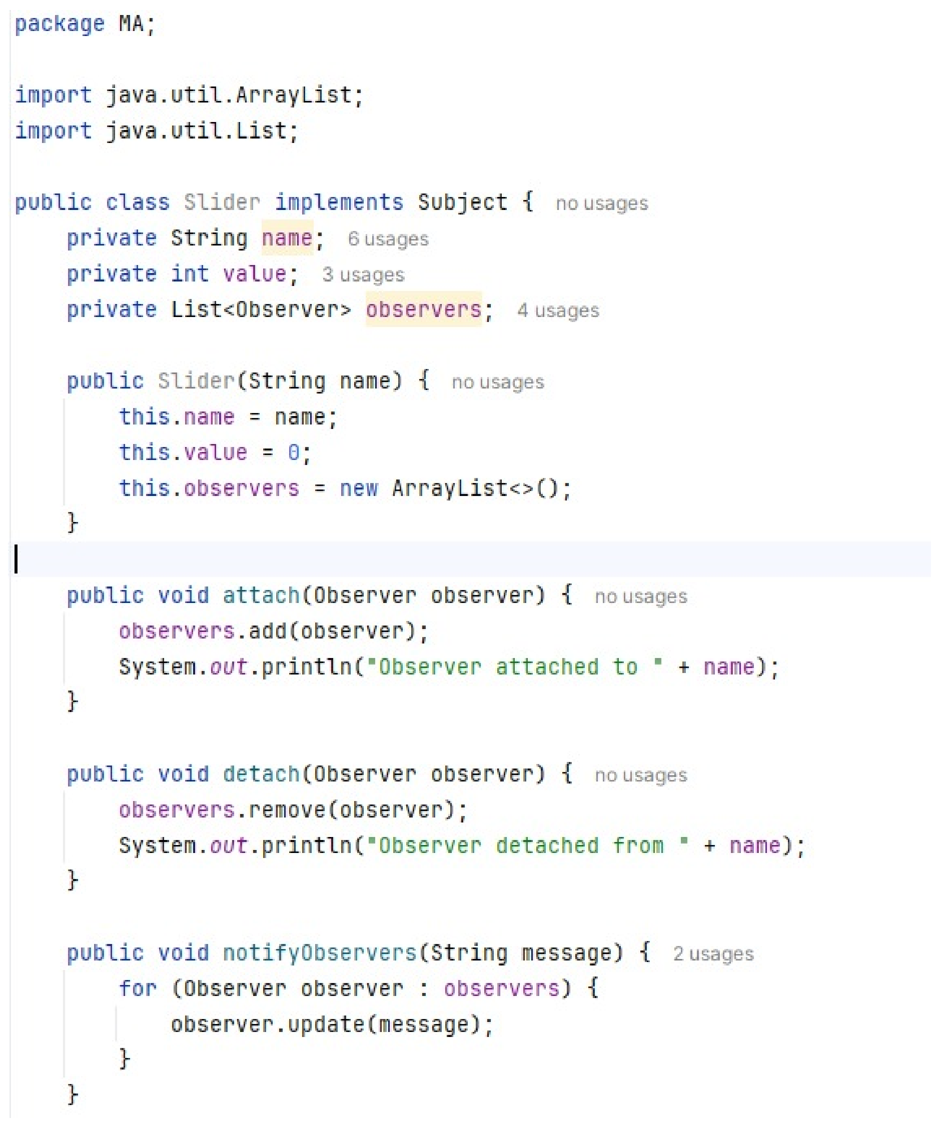


```
package MA;

import java.util.ArrayList;
import java.util.List;

public class Button implements Subject {  no usages
    private String name;  6 usages
    private List<Observer> observers;  4 usages

    public Button(String name) {  no usages
        this.name = name;
        this.observers = new ArrayList<>();
    }

    public void attach(Observer observer) {  no usages
        observers.add(observer);
        System.out.println("Observer attached to " + name);
    }

    public void detach(Observer observer) {  no usages
        observers.remove(observer);
        System.out.println("Observer detached from " + name);
    }

    public void notifyObservers(String message) {  2 usages
        for (Observer observer : observers) {
            observer.update(message);
        }
    }
```



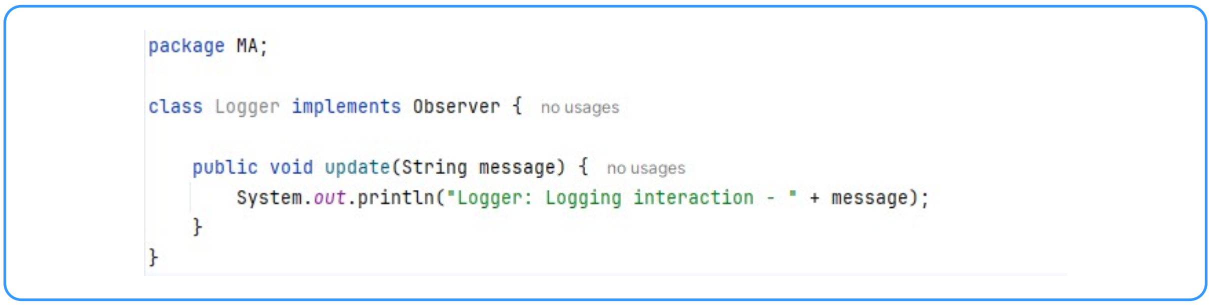
```
        public void click() {  no usages
            System.out.println("\n[" + name + " clicked]");
            notifyObservers(name + " was clicked");
        }

        public String getName() {
            return name;
        }
    }
```

4.3.4 Slider.java



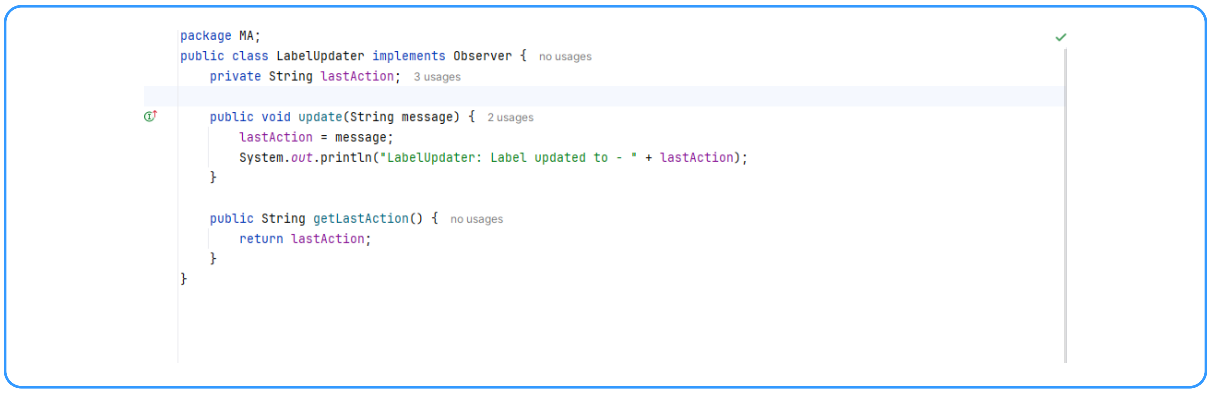
```
    package MA;

    import java.util.ArrayList;
    import java.util.List;

    public class Slider implements Subject {  no usages
        private String name;  6 usages
        private int value;  3 usages
        private List<Observer> observers;  4 usages

        public Slider(String name) {  no usages
            this.name = name;
            this.value = 0;
            this.observers = new ArrayList<>();
        }

        public void attach(Observer observer) {  no usages
            observers.add(observer);
            System.out.println("Observer attached to " + name);
        }

        public void detach(Observer observer) {  no usages
            observers.remove(observer);
            System.out.println("Observer detached from " + name);
        }

        public void notifyObservers(String message) {  2 usages
            for (Observer observer : observers) {
                observer.update(message);
            }
        }
```

4.3.5 Logger.java



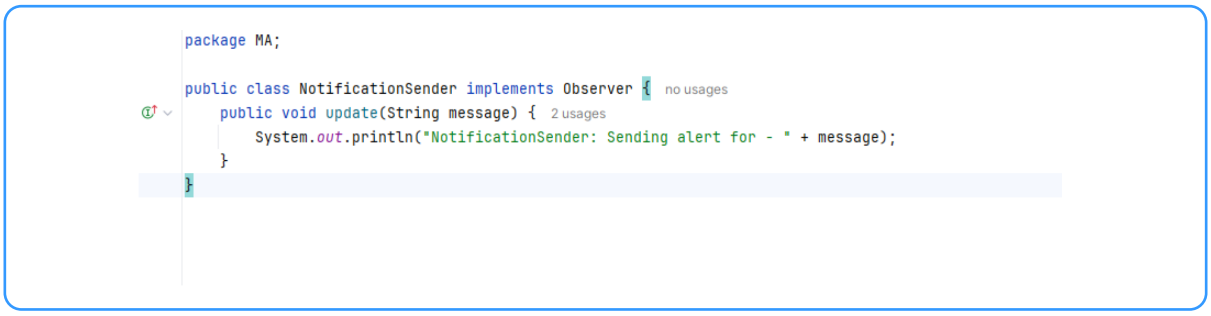
```
package MA;

class Logger implements Observer {  no usages

    public void update(String message) {  no usages
        System.out.println("Logger: Logging interaction - " + message);
    }
}
```

4.3.6 LabelUpdater.java



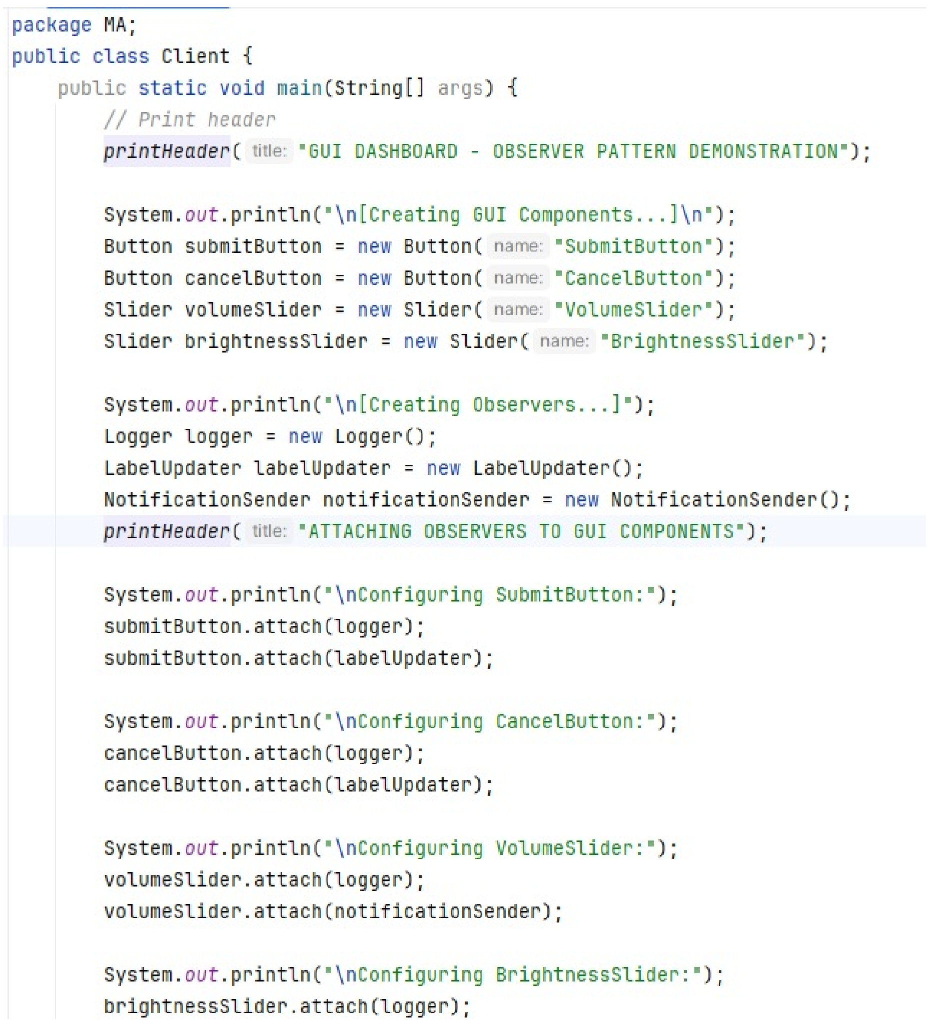
```
package MA;
public class LabelUpdater implements Observer {  no usages
    private String lastAction;  3 usages

    public void update(String message) {  2 usages
        lastAction = message;
        System.out.println("LabelUpdater: Label updated to - " + lastAction);
    }

    public String getLastAction() {  no usages
        return lastAction;
    }
}
```

4.3.7 NotificationSender.java

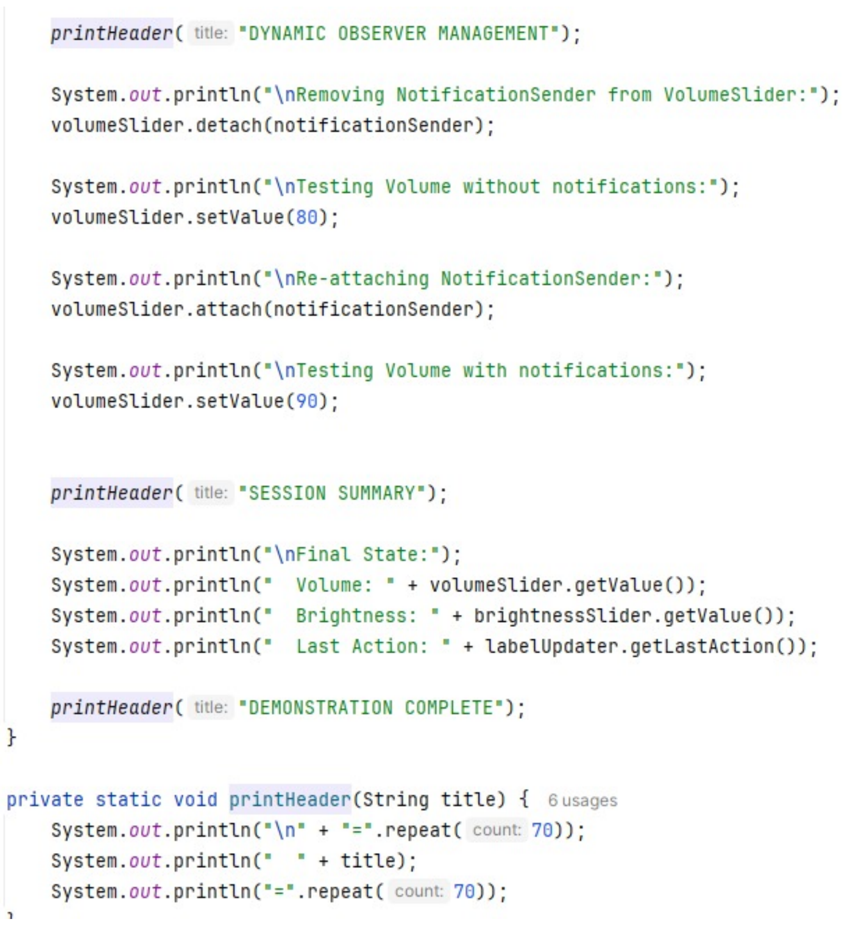


```
package MA;

public class NotificationSender implements Observer {  no usages
    public void update(String message) {  2 usages
        System.out.println("NotificationSender: Sending alert for - " + message);
    }
}
```

4.3.8 Client.java

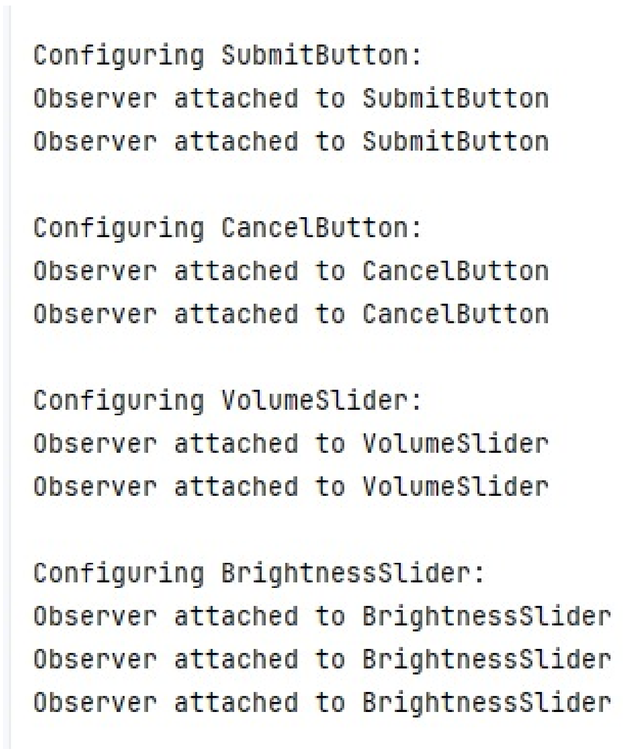


```java
package MA;
public class Client {
    public static void main(String[] args) {
        // Print header
        printHeader( title: "GUI DASHBOARD - OBSERVER PATTERN DEMONSTRATION");

        System.out.println("\n[Creating GUI Components...]\n");
        Button submitButton = new Button( name: "SubmitButton");
        Button cancelButton = new Button( name: "CancelButton");
        Slider volumeSlider = new Slider( name: "VolumeSlider");
        Slider brightnessSlider = new Slider( name: "BrightnessSlider");

        System.out.println("\n[Creating Observers...]");
        Logger logger = new Logger();
        LabelUpdater labelUpdater = new LabelUpdater();
        NotificationSender notificationSender = new NotificationSender();
        printHeader( title: "ATTACHING OBSERVERS TO GUI COMPONENTS");

        System.out.println("\nConfiguring SubmitButton:");
        submitButton.attach(logger);
        submitButton.attach(labelUpdater);

        System.out.println("\nConfiguring CancelButton:");
        cancelButton.attach(logger);
        cancelButton.attach(labelUpdater);

        System.out.println("\nConfiguring VolumeSlider:");
        volumeSlider.attach(logger);
        volumeSlider.attach(notificationSender);

        System.out.println("\nConfiguring BrightnessSlider:");
        brightnessSlider.attach(logger);
```

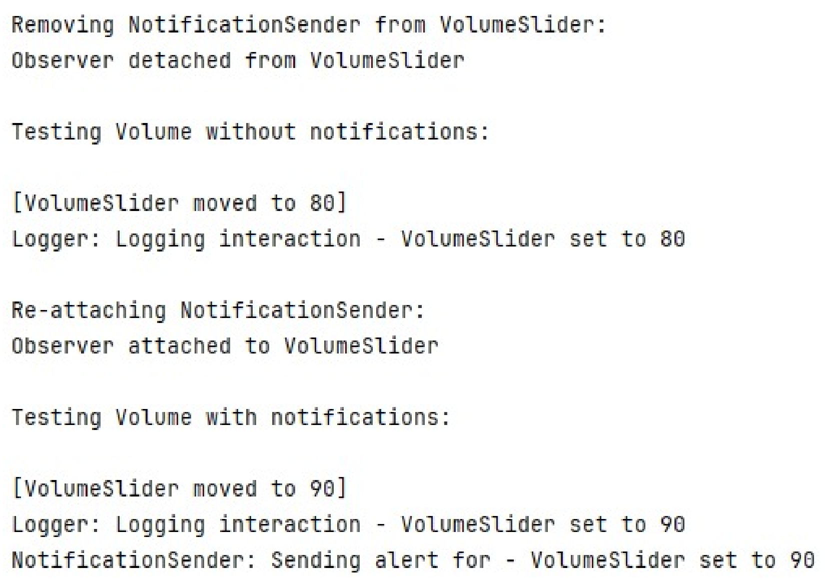


```
        printHeader( title: "DYNAMIC OBSERVER MANAGEMENT");

        System.out.println("\nRemoving NotificationSender from VolumeSlider:");
        volumeSlider.detach(notificationSender);

        System.out.println("\nTesting Volume without notifications:");
        volumeSlider.setValue(80);

        System.out.println("\nRe-attaching NotificationSender:");
        volumeSlider.attach(notificationSender);

        System.out.println("\nTesting Volume with notifications:");
        volumeSlider.setValue(90);


        printHeader( title: "SESSION SUMMARY");

        System.out.println("\nFinal State:");
        System.out.println("  Volume: " + volumeSlider.getValue());
        System.out.println("  Brightness: " + brightnessSlider.getValue());
        System.out.println("  Last Action: " + labelUpdater.getLastAction());

        printHeader( title: "DEMONSTRATION COMPLETE");
    }

    private static void printHeader(String title) {  6 usages
        System.out.println("\n" + "=".repeat( count: 70));
        System.out.println("  " + title);
        System.out.println("=".repeat( count: 70));
    }
```

4.3.9 Output.java



```
Configuring SubmitButton:
Observer attached to SubmitButton
Observer attached to SubmitButton

Configuring CancelButton:
Observer attached to CancelButton
Observer attached to CancelButton

Configuring VolumeSlider:
Observer attached to VolumeSlider
Observer attached to VolumeSlider

Configuring BrightnessSlider:
Observer attached to BrightnessSlider
Observer attached to BrightnessSlider
Observer attached to BrightnessSlider
```



```
Removing NotificationSender from VolumeSlider:
Observer detached from VolumeSlider

Testing Volume without notifications:

[VolumeSlider moved to 80]
Logger: Logging interaction - VolumeSlider set to 80

Re-attaching NotificationSender:
Observer attached to VolumeSlider

Testing Volume with notifications:

[VolumeSlider moved to 90]
Logger: Logging interaction - VolumeSlider set to 90
NotificationSender: Sending alert for - VolumeSlider set to 90
```