

# Design Patterns: Part 2

Zineb ELANSARI

November 25, 2025

## Contents

<b>1</b>	<b>Exercise 1: Navigation System with Strategy Pattern</b>	<b>2</b>
1.1	Task 1: Class Diagram . . . . .	2
1.2	Task 2: Java Implementation . . . . .	3
<b>2</b>	<b>Exercise 2: Vehicle Maintenance System</b>	<b>5</b>
2.1	Design Pattern . . . . .	5
2.2	Class Diagram . . . . .	5
2.3	Java Implementation . . . . .	5
<b>3</b>	<b>Exercise 3: Payment System with Adapter Pattern</b>	<b>8</b>
3.1	Design Pattern . . . . .	8
3.2	Participants . . . . .	8
3.3	Class Diagram . . . . .	8
3.4	Java Implementation . . . . .	9
<b>4</b>	<b>Exercise 4: GUI Dashboard with Observer Pattern</b>	<b>12</b>
4.1	Design Pattern . . . . .	12
4.2	Class Diagram . . . . .	12
4.3	Java Implementation . . . . .	12

# 1 Exercise 1: Navigation System with Strategy Pattern

## 1.1 Task 1: Class Diagram

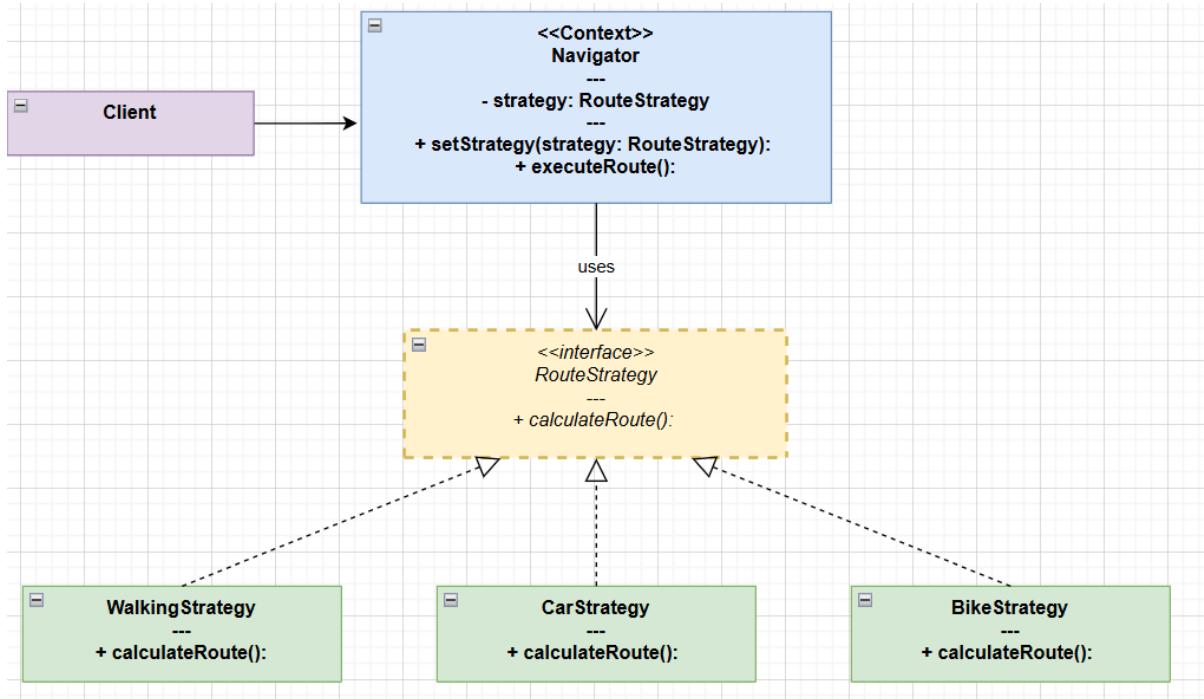


Figure 1: Strategy Pattern Navigation System

### Answers to Questions:

- Role of Navigator:** The Navigator acts as the **Context** in the Strategy pattern. It maintains a reference to a **RouteStrategy** and delegates the route calculation to the current strategy.
- Why Navigator depends on RouteStrategy interface:** This dependency allows the Navigator to work with any concrete strategy without knowing its implementation details. It promotes loose coupling and follows the Dependency Inversion Principle.
- SOLID Principles Applied:**
  - Open/Closed Principle (OCP):** The system is open for extension (new strategies can be added) but closed for modification (Navigator doesn't need changes).
  - Dependency Inversion Principle (DIP):** Navigator depends on the abstraction (**RouteStrategy** interface), not concrete implementations.
  - Single Responsibility Principle (SRP):** Each strategy class has one reason to change its specific routing algorithm.

## 1.2 Task 2: Java Implementation

```
1 // RouteStrategy interface
2 public interface RouteStrategy {
3     void calculateRoute(String origin, String destination);
4 }
5
6 // WalkingStrategy concrete implementation
7 public class WalkingStrategy implements RouteStrategy {
8     public void calculateRoute(String origin, String destination)
9     {
10         System.out.println("Calculating walking route from " +
11             origin +
12                 " to " + destination);
13         System.out.println("Walking: Using pedestrian paths, " +
14                         "estimated time: 45 minutes");
15     }
16 }
17
18 // CarStrategy concrete implementation
19 public class CarStrategy implements RouteStrategy {
20     public void calculateRoute(String origin, String destination)
21     {
22         System.out.println("Calculating car route from " + origin +
23             " to " + destination);
24         System.out.println("Car: Using highways and main roads, " +
25             "estimated time: 15 minutes");
26     }
27 }
28
29 // BikeStrategy concrete implementation
30 public class BikeStrategy implements RouteStrategy {
31     public void calculateRoute(String origin, String destination)
32     {
33         System.out.println("Calculating bike route from " +
34             origin +
35                 " to " + destination);
36         System.out.println("Bike: Using bike lanes and side
37             streets, " +
38                 "estimated time: 25 minutes");
39     }
40 }
41
42 // Navigator (Context)
43 public class Navigator {
44     private RouteStrategy strategy;
45
46     public void setStrategy(RouteStrategy strategy) {
47         this.strategy = strategy;
48     }
49 }
```

```
42     }
43
44     public void executeRoute(String origin, String destination) {
45         if (strategy == null) {
46             System.out.println("No strategy set!");
47             return;
48         }
49         strategy.calculateRoute(origin, destination);
50     }
51 }
52
53 // Client code
54 public class NavigationApp {
55     public static void main(String[] args) {
56         Navigator navigator = new Navigator();
57
58         // Using walking strategy
59         navigator.setStrategy(new WalkingStrategy());
60         navigator.executeRoute("Home", "Office");
61
62         System.out.println();
63
64         // Switching to car strategy at runtime
65         navigator.setStrategy(new CarStrategy());
66         navigator.executeRoute("Home", "Office");
67
68         System.out.println();
69
70         // Switching to bike strategy
71         navigator.setStrategy(new BikeStrategy());
72         navigator.executeRoute("Home", "Office");
73     }
74 }
```

## 2 Exercise 2: Vehicle Maintenance System

### 2.1 Design Pattern

The best design pattern for this problem is the **Composite Pattern**. This pattern allows us to treat individual objects (Independent companies) and compositions of objects (Parent companies) uniformly.

### 2.2 Class Diagram

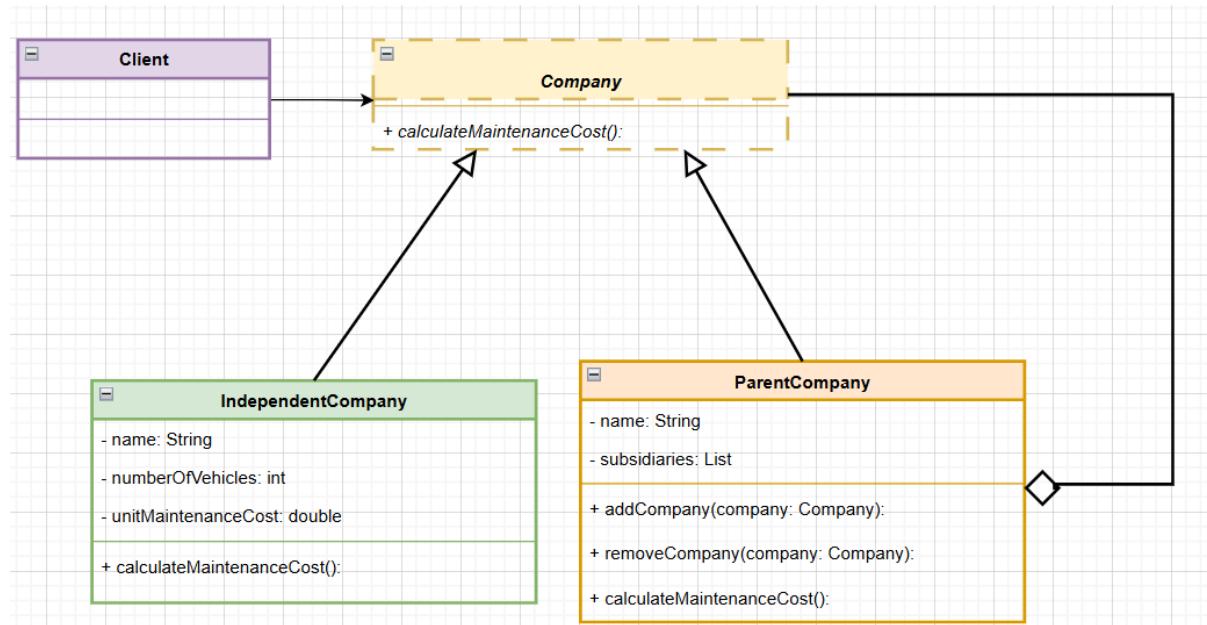


Figure 2: Composite Pattern Vehicle Maintenance System Class Diagram

### 2.3 Java Implementation

```

1 import java.util.ArrayList;
2 import java.util.List;
3
4 // Component interface
5 public interface Company {
6     double calculateMaintenanceCost();
7 }
8
9 // Leaf
10 public class IndependentCompany implements Company {
11     private String name;
12     private int numberofVehicles;
13     private double unitMaintenanceCost;
14
15     public IndependentCompany(String name, int numberofVehicles,
16                               double unitMaintenanceCost) {
17
18     }
19 }
20
21 // Composite
22 public class ParentCompany {
23     private String name;
24     private List<Company> subsidiaries;
25
26     public void addCompany(Company company) {
27         subsidiaries.add(company);
28     }
29
30     public void removeCompany(Company company) {
31         subsidiaries.remove(company);
32     }
33
34     public double calculateMaintenanceCost() {
35         double totalCost = 0;
36         for (Company company : subsidiaries) {
37             totalCost += company.calculateMaintenanceCost();
38         }
39         return totalCost;
40     }
41 }
42
43 // Client
44 public class Client {
45     public void calculateMaintenanceCost() {
46         // ...
47     }
48 }
49
50 // Main
51 public class Main {
52     public static void main(String[] args) {
53         ParentCompany parentCompany = new ParentCompany();
54         parentCompany.addCompany(new IndependentCompany("Company A", 10, 100));
55         parentCompany.addCompany(new IndependentCompany("Company B", 20, 150));
56         parentCompany.addCompany(new IndependentCompany("Company C", 30, 200));
57
58         Client client = new Client();
59         client.calculateMaintenanceCost();
60     }
61 }
  
```

```

17     this.name = name;
18     this.numberOfVehicles = numberOfVehicles;
19     this.unitMaintenanceCost = unitMaintenanceCost;
20 }
21
22 @Override
23 public double calculateMaintenanceCost() {
24     double cost = numberOfVehicles * unitMaintenanceCost;
25     System.out.println(name + " maintenance cost: " + cost);
26     return cost;
27 }
28 }
29
30 // Composite
31 public class ParentCompany implements Company {
32     private String name;
33     private List<Company> subsidiaries;
34
35     public ParentCompany(String name) {
36         this.name = name;
37         this.subsidiaries = new ArrayList<>();
38     }
39
40     public void addCompany(Company company) {
41         subsidiaries.add(company);
42     }
43
44     public void removeCompany(Company company) {
45         subsidiaries.remove(company);
46     }
47
48     @Override
49     public double calculateMaintenanceCost() {
50         double totalCost = 0;
51         System.out.println(name + " calculating total maintenance
52             cost:");
53         for (Company company : subsidiaries) {
54             totalCost += company.calculateMaintenanceCost();
55         }
56         System.out.println(name + " total cost: " + totalCost);
57         return totalCost;
58     }
59 }
60
61 // Client code
62 public class MaintenanceSystem {
63     public static void main(String[] args) {
64         // Create independent companies
65         Company company1 = new IndependentCompany("TechCorp", 10,
66             500);

```

```
65 Company company2 = new IndependentCompany("AutoFleet",
66     15, 450);
67 Company company3 = new IndependentCompany("LogisTrans",
68     8, 550);
69
70 // Create parent company and add subsidiaries
71 ParentCompany parentCompany = new ParentCompany("MegaCorp");
72 parentCompany.addCompany(company1);
73 parentCompany.addCompany(company2);
74
75 // Create another parent company
76 ParentCompany superParent = new ParentCompany("SuperHolding");
77 superParent.addCompany(parentCompany);
78 superParent.addCompany(company3);
79
80 // Calculate maintenance cost uniformly
81 System.out.println(" Calculating Maintenance Costs ");
82 double totalCost = superParent.calculateMaintenanceCost();
83
84 System.out.println("Grand Total: " + totalCost);
85 }
86 }
```

### 3 Exercise 3: Payment System with Adapter Pattern

#### 3.1 Design Pattern

The appropriate design pattern is the **Adapter Pattern**. It allows incompatible interfaces to work together by creating an adapter that translates one interface into another.

#### 3.2 Participants

- **Target:** PaymentProcessor interface
- **Adaptee 1:** QuickPay class
- **Adaptee 2:** SafeTransfer class
- **Adapter 1:** QuickPayAdapter implements PaymentProcessor
- **Adapter 2:** SafeTransferAdapter implements PaymentProcessor
- **Client:** Uses PaymentProcessor interface

#### 3.3 Class Diagram

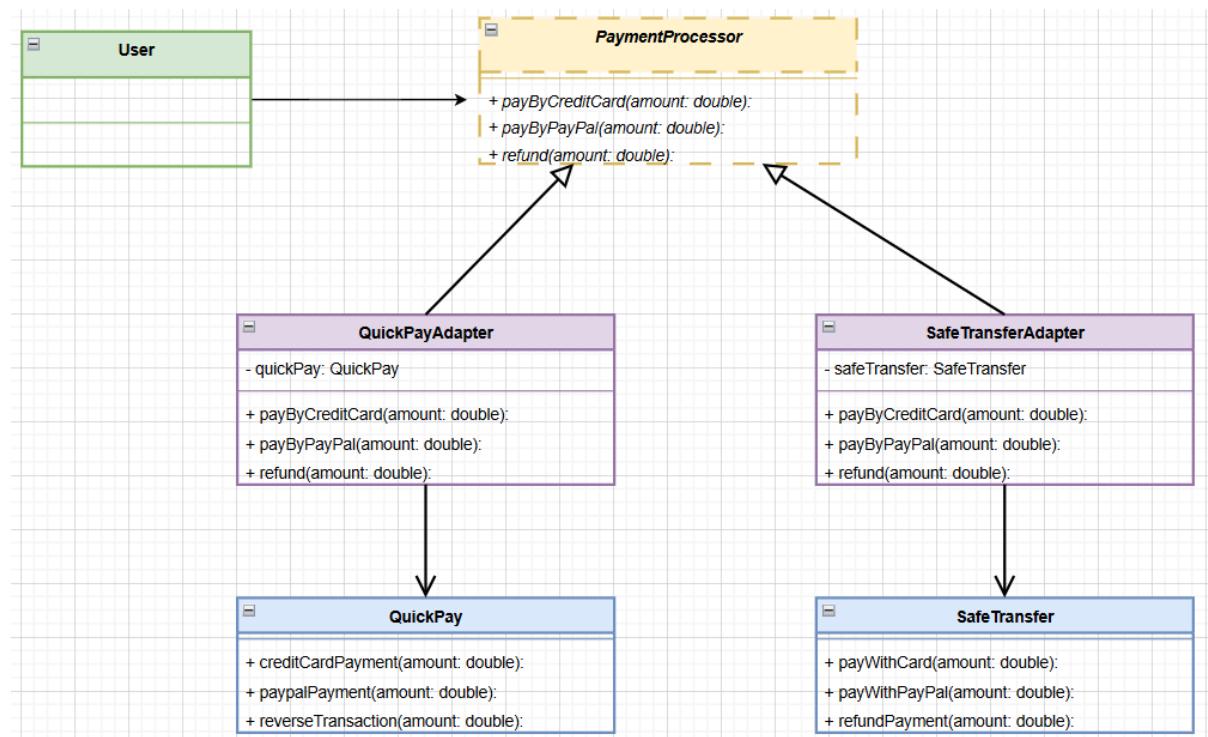


Figure 3: Payment System with Adapter Pattern Class diagram

### 3.4 Java Implementation

```

1 // Target interface
2 public interface PaymentProcessor {
3     void payByCreditCard(double amount);
4     void payByPayPal(double amount);
5     void refund(double amount);
6 }
7
8 // Adaptee 1: QuickPay
9 public class QuickPay {
10    public void creditCardPayment(double amount) {
11        System.out.println("QuickPay: Processing credit card
12            payment " +
13                amount);
14    }
15
16    public void paypalPayment(double amount) {
17        System.out.println("QuickPay: Processing PayPal payment " +
18            amount);
19    }
20
21    public void reverseTransaction(double amount) {
22        System.out.println("QuickPay: Reversing transaction " +
23            amount);
24    }
25 }
26
27 // Adaptee 2: SafeTransfer
28 public class SafeTransfer {
29    public void payWithCard(double amount) {
30        System.out.println("SafeTransfer: Paying with credit card
31            " +
32                amount);
33    }
34
35    public void payWithPayPal(double amount) {
36        System.out.println("SafeTransfer: Paying with PayPal " +
37            amount);
38    }
39 }
40
41 // Adapter 1: QuickPayAdapter
42 public class QuickPayAdapter implements PaymentProcessor {
43     private QuickPay quickPay;

```

```

44
45     public QuickPayAdapter(QuickPay quickPay) {
46         this.quickPay = quickPay;
47     }
48
49     public void payByCreditCard(double amount) {
50         quickPay.creditCardPayment(amount);
51     }
52
53     @Override
54     public void payByPayPal(double amount) {
55         quickPay.paypalPayment(amount);
56     }
57
58     public void refund(double amount) {
59         quickPay.reverseTransaction(amount);
60     }
61 }
62
63 // Adapter 2: SafeTransferAdapter
64 public class SafeTransferAdapter implements PaymentProcessor {
65     private SafeTransfer safeTransfer;
66
67     public SafeTransferAdapter(SafeTransfer safeTransfer) {
68         this.safeTransfer = safeTransfer;
69     }
70
71     @Override
72     public void payByCreditCard(double amount) {
73         safeTransfer.payWithCard(amount);
74     }
75
76     public void payByPayPal(double amount) {
77         safeTransfer.payWithPayPal(amount);
78     }
79
80
81     public void refund(double amount) {
82         safeTransfer.refundPayment(amount);
83     }
84 }
85
86 // Client code
87 public class User {
88     public static void main(String[] args) {
89         // Using QuickPay through adapter
90         PaymentProcessor processor1 =
91             new QuickPayAdapter(new QuickPay());
92         System.out.println(" Using QuickPay ");
93         processor1.payByCreditCard(100.50);
94         processor1.payByPayPal(75.25);

```

```
95     processor1.refund(25.00);
96
97     System.out.println();
98
99     // Using SafeTransfer through adapter
100    PaymentProcessor processor2 =
101        new SafeTransferAdapter(new SafeTransfer());
102    System.out.println(" Using SafeTransfer ");
103    processor2.payByCreditCard(200.00);
104    processor2.payByPayPal(150.75);
105    processor2.refund(50.00);
106 }
107 }
```

## 4 Exercise 4: GUI Dashboard with Observer Pattern

### 4.1 Design Pattern

The most suitable design pattern is the **Observer Pattern**. This pattern defines a one to many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

#### Why Observer Pattern?

- Multiple components need to react to changes in GUI elements
- Loose coupling between subjects (Buttons, Sliders) and observers (Logger, LabelUpdater, NotificationSender)
- Components can be added or removed dynamically
- Ensures efficient and prompt notification of changes

### 4.2 Class Diagram

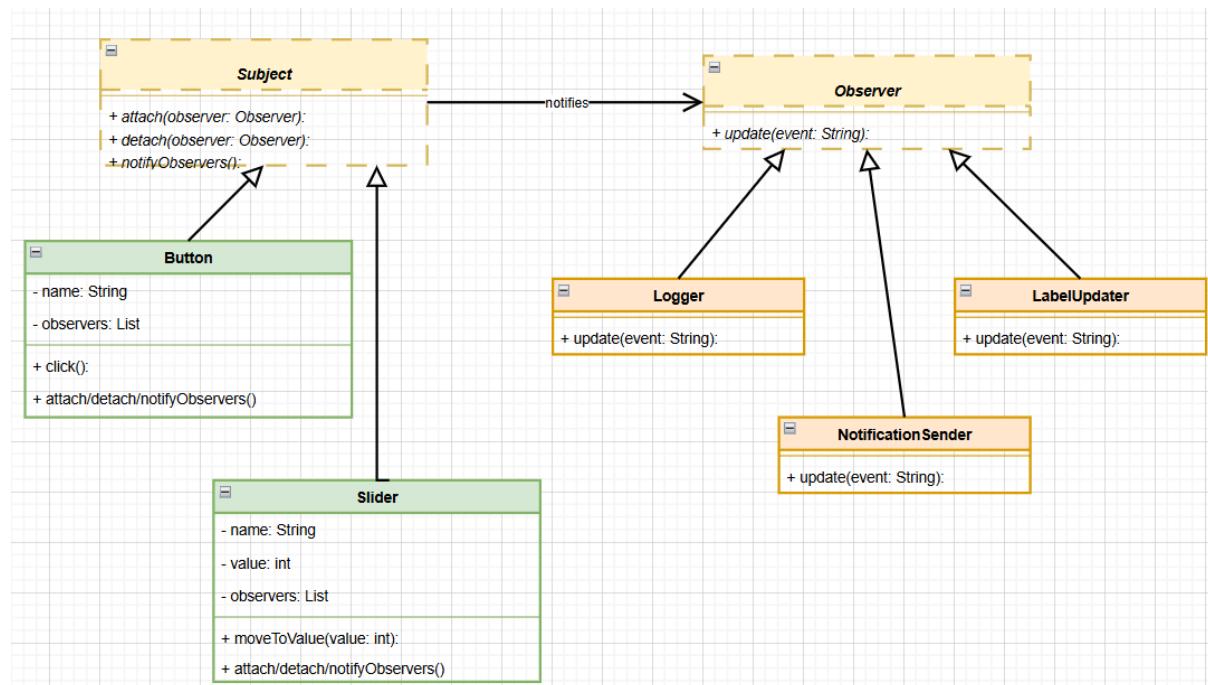


Figure 4: GUI Dashboard with Observer Pattern class diagram

### 4.3 Java Implementation

```

1 import java.util.ArrayList;
2 import java.util.List;
3
4 // Observer interface
5 public interface Observer {
6     void update(String event);
  
```

```

7  }
8
9  // Subject interface
10 public interface Subject {
11     void attach(Observer observer);
12     void detach(Observer observer);
13     void notifyObservers();
14 }
15
16 // Concrete Subject: Button
17 public class Button implements Subject {
18     private String name;
19     private List<Observer> observers;
20     private String lastAction;
21
22     public Button(String name) {
23         this.name = name;
24         this.observers = new ArrayList<>();
25     }
26
27
28     public void attach(Observer observer) {
29         observers.add(observer);
30     }
31
32
33     public void detach(Observer observer) {
34         observers.remove(observer);
35     }
36
37
38     public void notifyObservers() {
39         for (Observer observer : observers) {
40             observer.update(name + " clicked");
41         }
42     }
43
44     public void click() {
45         System.out.println("[ " + name + " was clicked]");
46         lastAction = "clicked";
47         notifyObservers();
48     }
49 }
50
51 // Concrete Subject: Slider
52 public class Slider implements Subject {
53     private String name;
54     private List<Observer> observers;
55     private int value;
56
57     public Slider(String name) {

```

```

58     this.name = name;
59     this.observers = new ArrayList<>();
60     this.value = 50; // default value
61 }
62
63
64     public void attach(Observer observer) {
65         observers.add(observer);
66     }
67
68
69     public void detach(Observer observer) {
70         observers.remove(observer);
71     }
72
73
74     public void notifyObservers() {
75         for (Observer observer : observers) {
76             observer.update(name + " moved to " + value);
77         }
78     }
79
80     public void moveToValue(int newValue) {
81         this.value = newValue;
82         System.out.println("[" + name + " moved to " + value + "]"
83             );
84         notifyObservers();
85     }
86
87 // Concrete Observer: Logger
88 public class Logger implements Observer {
89     public void update(String event) {
90         System.out.println("Logger: Logging interaction - " +
91             event);
92     }
93 }
94
95 // Concrete Observer: LabelUpdater
96 public class LabelUpdater implements Observer {
97
98     public void update(String event) {
99         System.out.println("LabelUpdater: Updating label - Last
100            action: " +
101                event);
102    }
103
104 // Concrete Observer: NotificationSender
105 public class NotificationSender implements Observer {
106     public void update(String event) {

```

```

106     System.out.println("NotificationSender: Sending alert for
107         " + event);
108 }
109
110 // Client code
111 public class DashboardApp {
112     public static void main(String[] args) {
113         // Create GUI elements
114         Button submitButton = new Button("SubmitButton");
115         Button cancelButton = new Button("CancelButton");
116         Slider volumeSlider = new Slider("VolumeSlider");
117         Slider brightnessSlider = new Slider("BrightnessSlider");
118
119         // Create observers
120         Logger logger = new Logger();
121         LabelUpdater labelUpdater = new LabelUpdater();
122         NotificationSender notificationSender = new
123             NotificationSender();
124
125         // Attach observers to SubmitButton
126         submitButton.attach(logger);
127         submitButton.attach(labelUpdater);
128
129         // Attach observers to VolumeSlider
130         volumeSlider.attach(logger);
131         volumeSlider.attach(notificationSender);
132
133         // Attach observers to CancelButton
134         cancelButton.attach(logger);
135         cancelButton.attach(labelUpdater);
136
137         // Simulate user interactions
138         System.out.println(" User Interaction 1 ");
139         submitButton.click();
140
141         System.out.println("\n User Interaction 2 ");
142         volumeSlider.moveToValue(75);
143
144         System.out.println("\n User Interaction 3 ");
145         cancelButton.click();
146
147         System.out.println("\n User Interaction 4 ");
148         brightnessSlider.attach(logger);
149         brightnessSlider.attach(notificationSender);
150         brightnessSlider.moveToValue(30);
151     }
}

```