Design Patterns: Part 2

Zineb ELANSARI

November 23, 2025

Contents

1 Exercise 1: Navigation System with Strategy Pattern

1.1 Task 1: Class Diagram

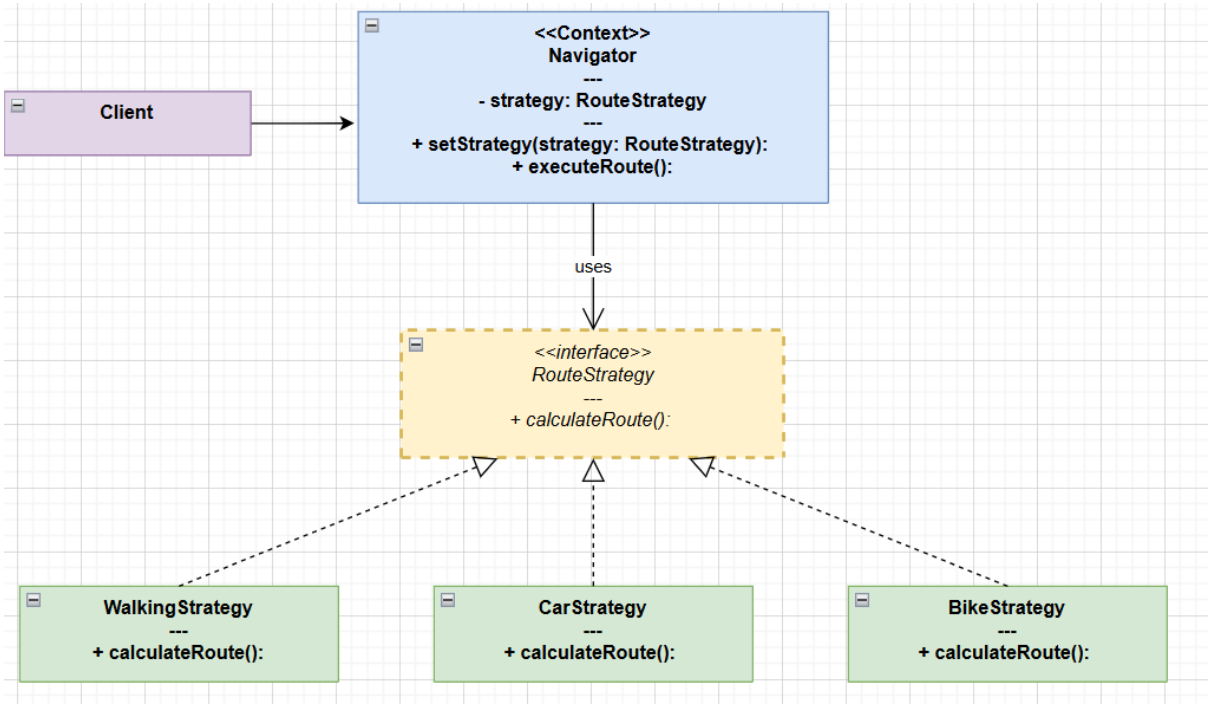

Figure 1: Strategy Pattern Navigation System

Answers to Questions:

1. **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.

2. **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.

3. **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

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

// WalkingStrategy concrete implementation
public class WalkingStrategy implements RouteStrategy {
    @Override
    public void calculateRoute(String origin, String destination) {
        System.out.println("Calculating walking route from " + origin +
                           " to " + destination);
        System.out.println("Walking: Using pedestrian paths, " +
                           "estimated time: 45 minutes");
    }
}

// CarStrategy concrete implementation
public class CarStrategy implements RouteStrategy {
    @Override
    public void calculateRoute(String origin, String destination) {
        System.out.println("Calculating car route from " + origin +
                           " to " + destination);
        System.out.println("Car: Using highways and main roads, " +
                           "estimated time: 15 minutes");
    }
}

// BikeStrategy concrete implementation
public class BikeStrategy implements RouteStrategy {
    @Override
    public void calculateRoute(String origin, String destination) {
        System.out.println("Calculating bike route from " + origin +
                           " to " + destination);
        System.out.println("Bike: Using bike lanes and side streets, " +
                           "estimated time: 25 minutes");
    }
}

// Navigator (Context)
public class Navigator {
    private RouteStrategy strategy;
```

```
42
43      public void setStrategy(RouteStrategy strategy) {
44          this.strategy = strategy;
45      }
46
47      public void executeRoute(String origin, String destination) {
48          if (strategy == null) {
49              System.out.println("No strategy set!");
50              return;
51          }
52          strategy.calculateRoute(origin, destination);
53      }
54  }
55
56  // Client code
57  public class NavigationApp {
58      public static void main(String[] args) {
59          Navigator navigator = new Navigator();
60
61          // Using walking strategy
62          navigator.setStrategy(new WalkingStrategy());
63          navigator.executeRoute("Home", "Office");
64
65          System.out.println();
66
67          // Switching to car strategy at runtime
68          navigator.setStrategy(new CarStrategy());
69          navigator.executeRoute("Home", "Office");
70
71          System.out.println();
72
73          // Switching to bike strategy
74          navigator.setStrategy(new BikeStrategy());
75          navigator.executeRoute("Home", "Office");
76      }
77  }
```

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

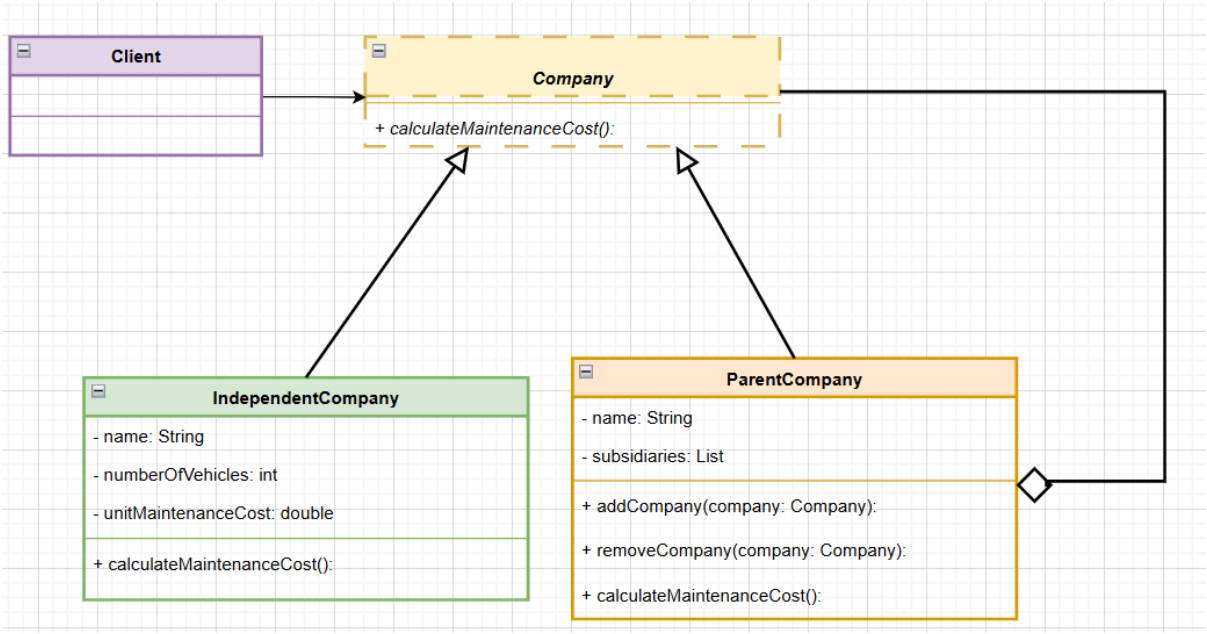

Figure 2: Composite Pattern Vehicle Maintenance System Class Diagram

2.3 Java Implementation

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

// Component interface
public interface Company {
    double calculateMaintenanceCost();
}

// Leaf
public class IndependentCompany implements Company {
    private String name;
    private int numberOfVehicles;
    private double unitMaintenanceCost;

    public IndependentCompany(String name, int numberOfVehicles,
                              double unitMaintenanceCost) {
```

```
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
                cost:");
52          for (Company company : subsidiaries) {
53              totalCost += company.calculateMaintenanceCost();
54          }
55          System.out.println(name + " total cost: $" + totalCost);
56          return totalCost;
57      }
58  }
59
60  // Client code
61  public class MaintenanceSystem {
62      public static void main(String[] args) {
63          // Create independent companies
64          Company company1 = new IndependentCompany("TechCorp", 10,
                500);
```

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

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

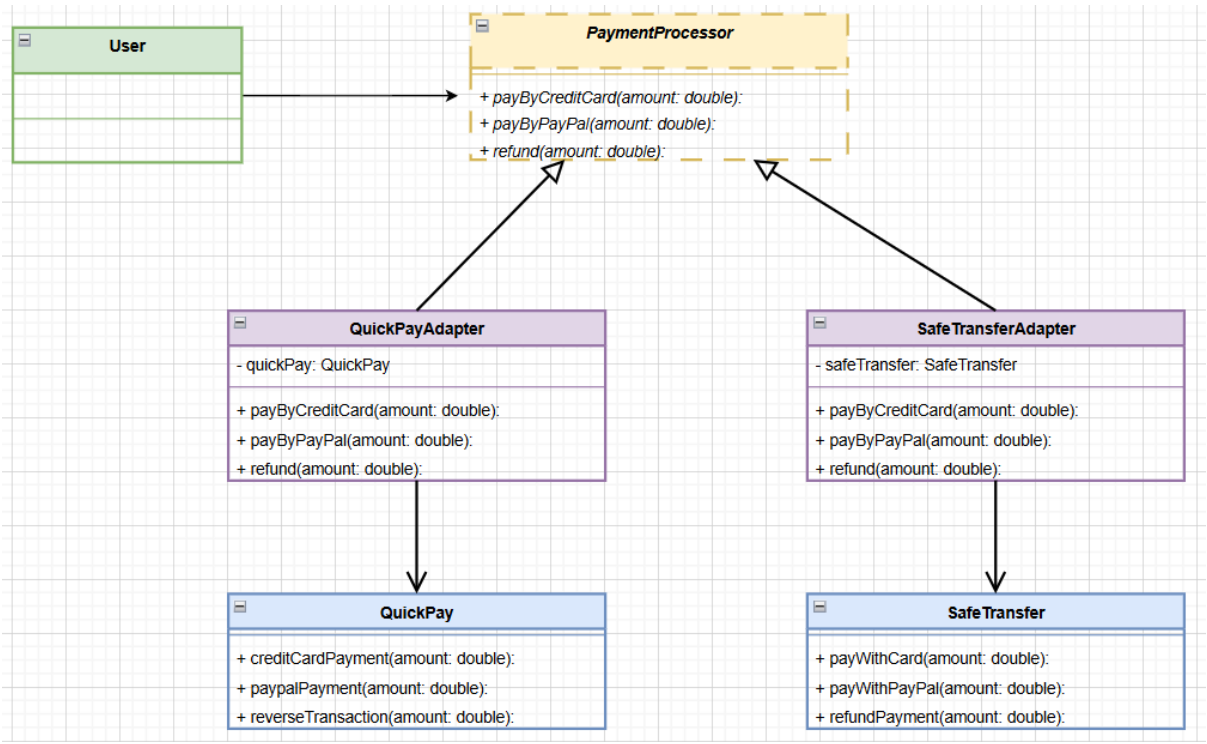

Figure 3: Payment System with Adapter Pattern Class diagram

3.4 Java Implementation

```java
// Target interface
public interface PaymentProcessor {
    void payByCreditCard(double amount);
    void payByPayPal(double amount);
    void refund(double amount);
}

// Adaptee 1: QuickPay
public class QuickPay {
    public void creditCardPayment(double amount) {
        System.out.println("QuickPay: Processing credit card payment $" +
                           amount);
    }

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

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

// Adaptee 2: SafeTransfer
public class SafeTransfer {
    public void payWithCard(double amount) {
        System.out.println("SafeTransfer: Paying with credit card $" +
                           amount);
    }

    public void payWithPayPal(double amount) {
        System.out.println("SafeTransfer: Paying with PayPal $" +
                           amount);
    }

    public void refundPayment(double amount) {
        System.out.println("SafeTransfer: Refunding payment $" + amount);
    }
}

// Adapter 1: QuickPayAdapter
public class QuickPayAdapter implements PaymentProcessor {
    private QuickPay quickPay;
```

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

```
95          System.out.println(" Using QuickPay ");
96          processor1.payByCreditCard(100.50);
97          processor1.payByPayPal(75.25);
98          processor1.refund(25.00);
99
100         System.out.println();
101
102         // Using SafeTransfer through adapter
103         PaymentProcessor processor2 =
104             new SafeTransferAdapter(new SafeTransfer());
105         System.out.println(" Using SafeTransfer ");
106         processor2.payByCreditCard(200.00);
107         processor2.payByPayPal(150.75);
108         processor2.refund(50.00);
109     }
110 }
```

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

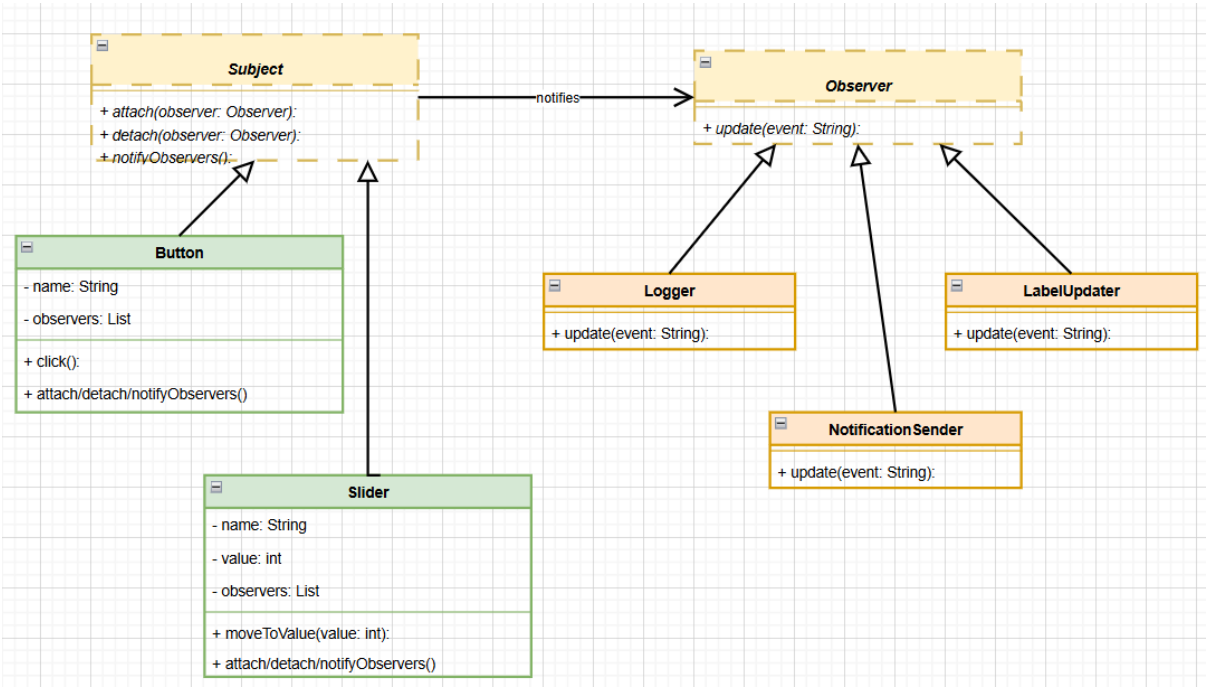

Figure 4: GUI Dashboard with Observer Pattern class diagram

4.3 Java Implementation

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

// Observer interface
public interface Observer {
    void update(String event);
```

```
}

// Subject interface
public interface Subject {
    void attach(Observer observer);
    void detach(Observer observer);
    void notifyObservers();
}

// Concrete Subject: Button
public class Button implements Subject {
    private String name;
    private List<Observer> observers;
    private String lastAction;

    public Button(String name) {
        this.name = name;
        this.observers = new ArrayList<>();
    }

    @Override
    public void attach(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void detach(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(name + " clicked");
        }
    }

    public void click() {
        System.out.println("[" + name + " was clicked]");
        lastAction = "clicked";
        notifyObservers();
    }
}

// Concrete Subject: Slider
public class Slider implements Subject {
    private String name;
    private List<Observer> observers;
    private int value;

    public Slider(String name) {
```

```
58          this.name = name;
59          this.observers = new ArrayList<>();
60          this.value = 50; // default value
61      }
62
63      @Override
64      public void attach(Observer observer) {
65          observers.add(observer);
66      }
67
68      @Override
69      public void detach(Observer observer) {
70          observers.remove(observer);
71      }
72
73      @Override
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          notifyObservers();
84      }
85  }
86
87  // Concrete Observer: Logger
88  public class Logger implements Observer {
89      @Override
90      public void update(String event) {
91          System.out.println("Logger: Logging interaction - " +
                event);
92      }
93  }
94
95  // Concrete Observer: LabelUpdater
96  public class LabelUpdater implements Observer {
97      @Override
98      public void update(String event) {
99          System.out.println("LabelUpdater: Updating label - Last
                action: " +
100                             event);
101     }
102 }
103
104 // Concrete Observer: NotificationSender
105 public class NotificationSender implements Observer {
```

```
106     @Override
107     public void update(String event) {
108         System.out.println("NotificationSender: Sending alert for
                " + event);
109     }
110 }
111 
112 // Client code
113 public class DashboardApp {
114     public static void main(String[] args) {
115         // Create GUI elements
116         Button submitButton = new Button("SubmitButton");
117         Button cancelButton = new Button("CancelButton");
118         Slider volumeSlider = new Slider("VolumeSlider");
119         Slider brightnessSlider = new Slider("BrightnessSlider");
120 
121         // Create observers
122         Logger logger = new Logger();
123         LabelUpdater labelUpdater = new LabelUpdater();
124         NotificationSender notificationSender = new
              NotificationSender();
125 
126         // Attach observers to SubmitButton
127         submitButton.attach(logger);
128         submitButton.attach(labelUpdater);
129 
130         // Attach observers to VolumeSlider
131         volumeSlider.attach(logger);
132         volumeSlider.attach(notificationSender);
133 
134         // Attach observers to CancelButton
135         cancelButton.attach(logger);
136         cancelButton.attach(labelUpdater);
137 
138         // Simulate user interactions
139         System.out.println("=== User Interaction 1 ===");
140         submitButton.click();
141 
142         System.out.println("\n=== User Interaction 2 ===");
143         volumeSlider.moveToValue(75);
144 
145         System.out.println("\n=== User Interaction 3 ===");
146         cancelButton.click();
147 
148         System.out.println("\n=== User Interaction 4 ===");
149         brightnessSlider.attach(logger);
150         brightnessSlider.attach(notificationSender);
151         brightnessSlider.moveToValue(30);
152     }
153 }
```