Design Patterns: Part 2

Zineb ELANSARI

November 25, 2025

Contents

1 Exercise 1: Navigation System with Strategy Pattern

1.1 Task 1: Class Diagram

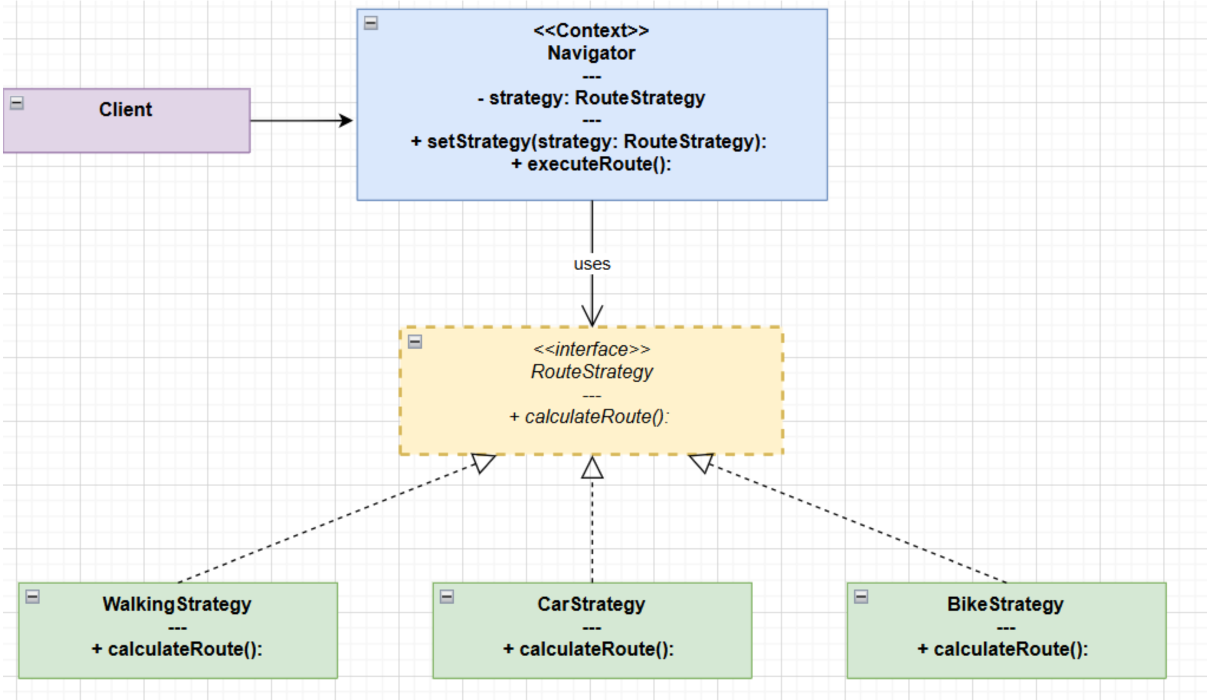

Figure 1: Strategy Pattern Navigation System

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 {
    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 {
    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 {
    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;

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

    public void executeRoute(String origin, String destination) {
        if (strategy == null) {
            System.out.println("No strategy set!");
            return;
        }
        strategy.calculateRoute(origin, destination);
    }
}

// Client code
public class NavigationApp {
    public static void main(String[] args) {
        Navigator navigator = new Navigator();

        // Using walking strategy
        navigator.setStrategy(new WalkingStrategy());
        navigator.executeRoute("Home", "Office");

        System.out.println();

        // Switching to car strategy at runtime
        navigator.setStrategy(new CarStrategy());
        navigator.executeRoute("Home", "Office");

        System.out.println();

        // Switching to bike strategy
        navigator.setStrategy(new BikeStrategy());
        navigator.executeRoute("Home", "Office");
    }
}
```

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

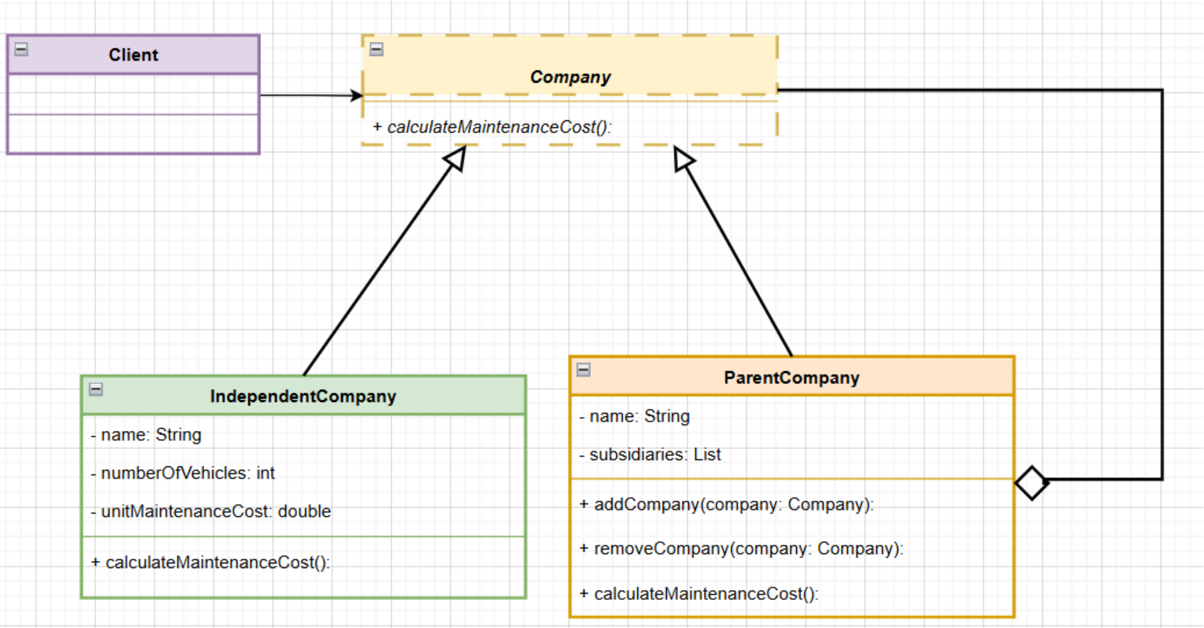
Client
Company
+ calculateMaintenanceCost():
IndependentCompany
- name: String
- numberOfVehicles: int
- unitMaintenanceCost: double
+ calculateMaintenanceCost():
ParentCompany
- name: String
- subsidiaries: List
+ addCompany(company: Company):
+ removeCompany(company: Company):
+ calculateMaintenanceCost():

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) {
```

```java
        this.name = name;
        this.numberOfVehicles = numberOfVehicles;
        this.unitMaintenanceCost = unitMaintenanceCost;
    }

    @Override
    public double calculateMaintenanceCost() {
        double cost = numberOfVehicles * unitMaintenanceCost;
        System.out.println(name + " maintenance cost: " + cost);
        return cost;
    }
}

// Composite
public class ParentCompany implements Company {
    private String name;
    private List<Company> subsidiaries;

    public ParentCompany(String name) {
        this.name = name;
        this.subsidiaries = new ArrayList<>();
    }

    public void addCompany(Company company) {
        subsidiaries.add(company);
    }

    public void removeCompany(Company company) {
        subsidiaries.remove(company);
    }

    @Override
    public double calculateMaintenanceCost() {
        double totalCost = 0;
        System.out.println(name + " calculating total maintenance cost:");
        for (Company company : subsidiaries) {
            totalCost += company.calculateMaintenanceCost();
        }
        System.out.println(name + " total cost: " + totalCost);
        return totalCost;
    }
}

// Client code
public class MaintenanceSystem {
    public static void main(String[] args) {
        // Create independent companies
        Company company1 = new IndependentCompany("TechCorp", 10, 500);
        Company company2 = new IndependentCompany("AutoFleet", 15, 450);
        Company company3 = new IndependentCompany("LogisTrans", 8, 550);

        // Create parent company and add subsidiaries
        ParentCompany parentCompany = new ParentCompany("MegaCorp");
        parentCompany.addCompany(company1);
        parentCompany.addCompany(company2);

        // Create another parent company
        ParentCompany superParent = new ParentCompany("SuperHolding");
        superParent.addCompany(parentCompany);
        superParent.addCompany(company3);

        // Calculate maintenance cost uniformly
        System.out.println(" Calculating Maintenance Costs ");
        double totalCost = superParent.calculateMaintenanceCost();
        System.out.println("Grand Total: " + totalCost);
    }
}
```

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

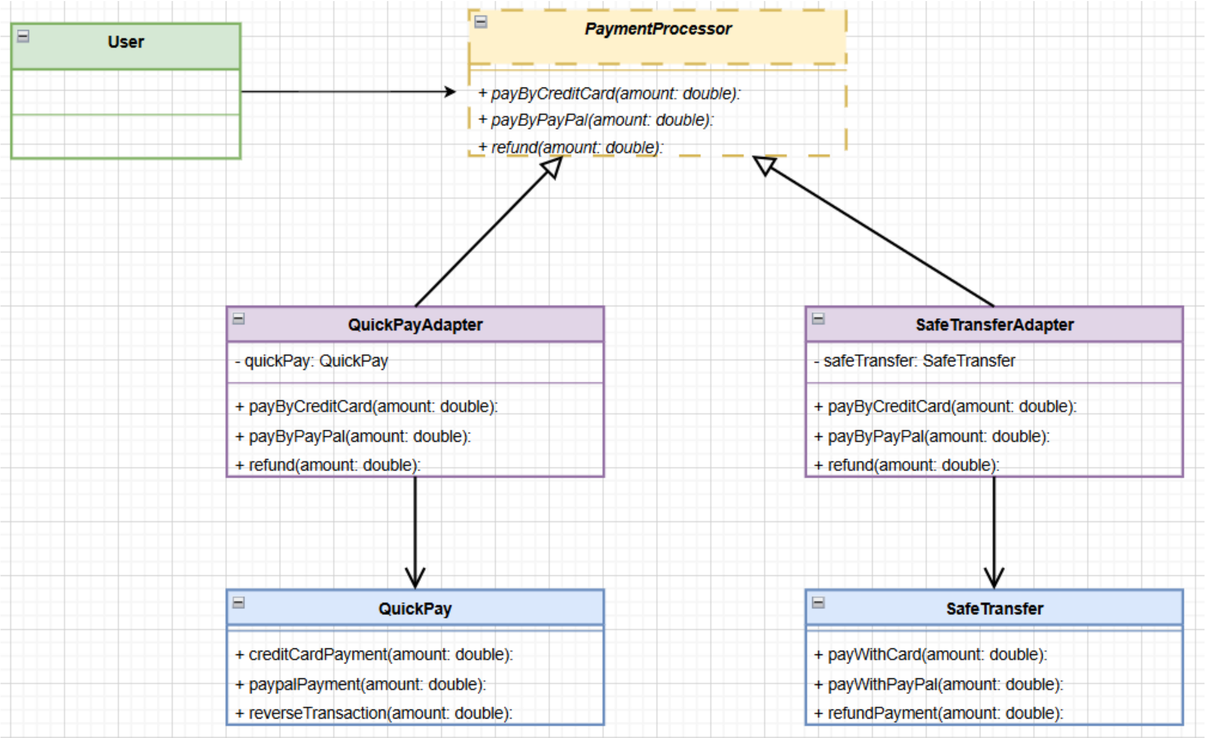
User
PaymentProcessor
+ payByCreditCard(amount: double):
+ payByPayPal(amount: double):
+ refund(amount: double):
QuickPayAdapter
- quickPay: QuickPay
+ payByCreditCard(amount: double):
+ payByPayPal(amount: double):
+ refund(amount: double):
SafeTransferAdapter
- safeTransfer: SafeTransfer
+ payByCreditCard(amount: double):
+ payByPayPal(amount: double):
+ refund(amount: double):
QuickPay
+ creditCardPayment(amount: double):
+ paypalPayment(amount: double):
+ reverseTransaction(amount: double):
SafeTransfer
+ payWithCard(amount: double):
+ payWithPayPal(amount: double):
+ refundPayment(amount: double):

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;

    public QuickPayAdapter(QuickPay quickPay) {
        this.quickPay = quickPay;
    }

    public void payByCreditCard(double amount) {
        quickPay.creditCardPayment(amount);
    }

    @Override
    public void payByPayPal(double amount) {
        quickPay.paypalPayment(amount);
    }

    public void refund(double amount) {
        quickPay.reverseTransaction(amount);
    }
}

// Adapter 2: SafeTransferAdapter
public class SafeTransferAdapter implements PaymentProcessor {
    private SafeTransfer safeTransfer;

    public SafeTransferAdapter(SafeTransfer safeTransfer) {
        this.safeTransfer = safeTransfer;
    }

    @Override
    public void payByCreditCard(double amount) {
        safeTransfer.payWithCard(amount);
    }

    public void payByPayPal(double amount) {
        safeTransfer.payWithPayPal(amount);
    }


    public void refund(double amount) {
        safeTransfer.refundPayment(amount);
    }
}

// Client code
public class User {
    public static void main(String[] args) {
        // Using QuickPay through adapter
        PaymentProcessor processor1 =
            new QuickPayAdapter(new QuickPay());
        System.out.println(" Using QuickPay ");
        processor1.payByCreditCard(100.50);
        processor1.payByPayPal(75.25);
        processor1.refund(25.00);

        System.out.println();

        // Using SafeTransfer through adapter
        PaymentProcessor processor2 =
            new SafeTransferAdapter(new SafeTransfer());
        System.out.println(" Using SafeTransfer ");
        processor2.payByCreditCard(200.00);
        processor2.payByPayPal(150.75);
        processor2.refund(50.00);
    }
}
```

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

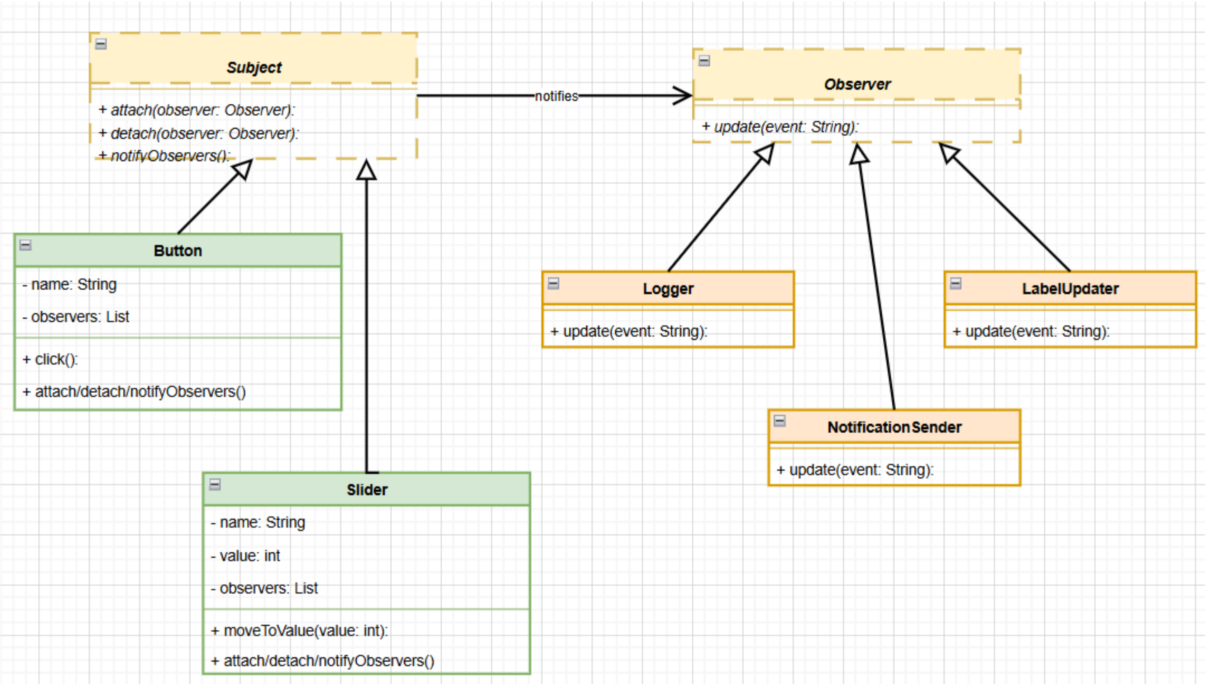

Figure 4: GUI Dashboard with Observer Pattern class diagram

4.3 Java Implementation

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

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

```java
}

// 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<>();
    }


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


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


    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) {
        this.name = name;
        this.observers = new ArrayList<>();
        this.value = 50; // default value
    }


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


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


    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(name + " moved to " + value);
        }
    }

    public void moveToValue(int newValue) {
        this.value = newValue;
        System.out.println("[" + name + " moved to " + value + "]");
        notifyObservers();
    }
}

// Concrete Observer: Logger
public class Logger implements Observer {
    public void update(String event) {
        System.out.println("Logger: Logging interaction - " + event);
    }
}

// Concrete Observer: LabelUpdater
public class LabelUpdater implements Observer {

    public void update(String event) {
        System.out.println("LabelUpdater: Updating label - Last action: " +
                         event);
    }
}

// Concrete Observer: NotificationSender
public class NotificationSender implements Observer {
    public void update(String event) {
        System.out.println("NotificationSender: Sending alert for " + event);
    }
}

// Client code
public class DashboardApp {
    public static void main(String[] args) {
        // Create GUI elements
        Button submitButton = new Button("SubmitButton");
        Button cancelButton = new Button("CancelButton");
        Slider volumeSlider = new Slider("VolumeSlider");
        Slider brightnessSlider = new Slider("BrightnessSlider");

        // Create observers
        Logger logger = new Logger();
        LabelUpdater labelUpdater = new LabelUpdater();
        NotificationSender notificationSender = new NotificationSender();

        // Attach observers to SubmitButton
        submitButton.attach(logger);
        submitButton.attach(labelUpdater);

        // Attach observers to VolumeSlider
        volumeSlider.attach(logger);
        volumeSlider.attach(notificationSender);

        // Attach observers to CancelButton
        cancelButton.attach(logger);
        cancelButton.attach(labelUpdater);

        // Simulate user interactions
        System.out.println(" User Interaction 1 ");
        submitButton.click();

        System.out.println("\n User Interaction 2 ");
        volumeSlider.moveToValue(75);

        System.out.println("\n User Interaction 3 ");
        cancelButton.click();

        System.out.println("\n User Interaction 4 ");
        brightnessSlider.attach(logger);
        brightnessSlider.attach(notificationSender);
        brightnessSlider.moveToValue(30);
    }
}
```