Design Patterns: Part 2

Nour el houda

November 26, 2025

1 Exercise 1: Flexible Navigation with the Strategy Pattern

1.1 Task 1: Class Diagram

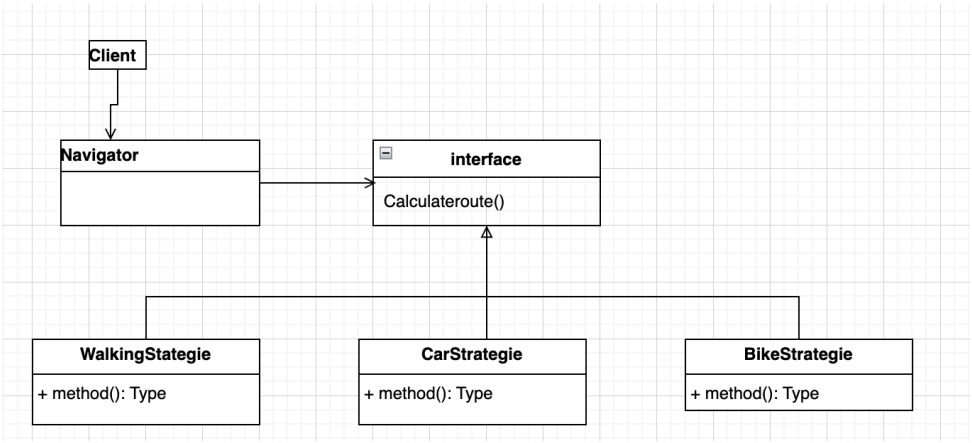

Figure 1: Class diagram for Navigation System using Strategy Pattern

Answers to Questions:

1. **What role does the Navigator class play?**

   The Navigator class plays the role of the **Context** in the Strategy pattern.

2. **Why does Navigator depend on the RouteStrategy interface?**

   - Navigator doesn't need to know about specific strategy implementations
   - New strategies can be added without changing Navigator's code
   - All strategies are treated uniformly through the common interface

3. **Which SOLID principles are applied in this design?**

   - **Single Responsibility Principle (SRP)**: Each strategy class has one responsibility (calculating a specific type of route), and Navigator has one responsibility (managing route calculation)

- **Open/Closed Principle (OCP)**: The system is open for extension (new strategies can be added) but closed for modification (Navigator doesn't need to change)
- **Dependency Inversion Principle (DIP)**: Navigator depends on the RouteStrategy abstraction, not concrete implementations

1.2 Task 2: Java Implementation

1.2.1 RouteStrategy Interface

```
// Strategy Interface
public interface RouteStrategy {
    void calculateRoute(String start, String destination);
}
```

1.2.2 Concrete Strategy Classes

```
// Concrete Strategy: Walking
public class WalkingStrategy implements RouteStrategy {
    @Override
    public void calculateRoute(String start, String destination) {
        System.out.println("Calculating walking route from " + start +
                           " to " + destination);
        System.out.println("Route: Use pedestrian paths and sidewalks");
        System.out.println("Estimated time: 45 minutes");
    }
}
```

```
// Concrete Strategy: Car
public class CarStrategy implements RouteStrategy {
    @Override
    public void calculateRoute(String start, String destination) {
        System.out.println("Calculating car route from " + start +
                           " to " + destination);
        System.out.println("Route: Use highways and main roads");
        System.out.println("Estimated time: 15 minutes");
    }
}
```

```
// Concrete Strategy: Bike (Optional)
public class BikeStrategy implements RouteStrategy {
    @Override
    public void calculateRoute(String start, String destination) {
        System.out.println("Calculating bike route from " + start +
                           " to " + destination);
        System.out.println("Route: Use bike lanes and quiet streets");
        System.out.println("Estimated time: 25 minutes");
    }
}
```

1.2.3 Context Class

```java
// Context: Navigator
public class Navigator {
    private RouteStrategy routeStrategy;

    // Constructor
    public Navigator(RouteStrategy routeStrategy) {
        this.routeStrategy = routeStrategy;
    }

    // Method to change strategy at runtime
    public void setRouteStrategy(RouteStrategy routeStrategy) {
        this.routeStrategy = routeStrategy;
    }

    // Delegate to strategy
    public void buildRoute(String start, String destination) {
        System.out.println("\n=== Navigator Building Route ===");
        routeStrategy.calculateRoute(start, destination);
        System.out.println("================================\n");
    }
}
```

1.2.4 Client/Main Class

```java
// Client
public class NavigationClient {
    public static void main(String[] args) {
        // Create navigator with walking strategy
        Navigator navigator = new Navigator(new WalkingStrategy());
        navigator.buildRoute("Home", "Office");

        // Change to car strategy at runtime
        navigator.setRouteStrategy(new CarStrategy());
        navigator.buildRoute("Office", "Gym");

        // Change to bike strategy
        navigator.setRouteStrategy(new BikeStrategy());
        navigator.buildRoute("Gym", "Restaurant");
    }
}
```

2 Exercise 2: Vehicle Maintenance System

2.1 Question 1: Appropriate Design Pattern

The **Composite Pattern** is best suited to model this problem

2.2 Question 2: Class Diagram

[Insert your class diagram image here]

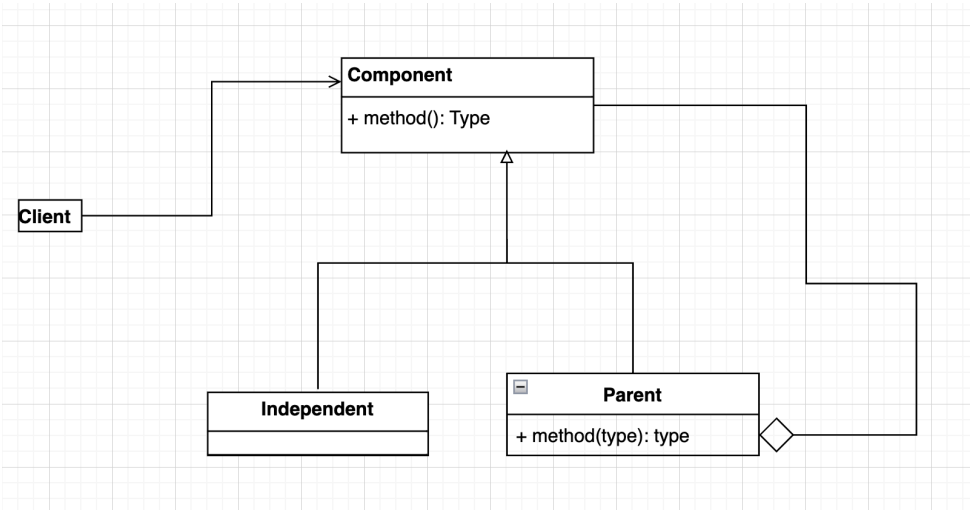

Figure 2: Class diagram for Vehicle Maintenance System using Composite Pattern

2.3 Question 3: Java Implementation

2.3.1 Component Interface

```
// Component: Common interface for both leaf and composite
public interface Company {
    double calculateMaintenanceCost();
    String getName();
}
```

2.3.2 Leaf Class

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

    public IndependentCompany(String name, int numberOfVehicles,
                              double unitMaintenanceCost) {
        this.name = name;
        this.numberOfVehicles = numberOfVehicles;
        this.unitMaintenanceCost = unitMaintenanceCost;
    }

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

    @Override
    public String getName() {
        return name;
```

```
    }
}
```

2.3.3 Composite Class

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

// Composite: Parent Company
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);
        System.out.println("Added " + company.getName() +
                           " to " + name);
    }

    public void removeCompany(Company company) {
        subsidiaries.remove(company);
        System.out.println("Removed " + company.getName() +
                           " from " + name);
    }

    @Override
    public double calculateMaintenanceCost() {
        System.out.println("\n" + name + " (Parent Company) - " +
                           "Calculating total maintenance cost:");
        double totalCost = 0;

        for (Company company : subsidiaries) {
            totalCost += company.calculateMaintenanceCost();
        }

        System.out.println(name + " Total: $" + totalCost);
        return totalCost;
    }

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

2.3.4 Client/Main Class

```
// Client
public class VehicleMaintenanceClient {
    public static void main(String[] args) {
        // Create independent companies (leaves)
```

```
        Company companyA = new IndependentCompany("Company A", 10, 500);
        Company companyB = new IndependentCompany("Company B", 5, 600);
        Company companyC = new IndependentCompany("Company C", 8, 550);
        Company companyD = new IndependentCompany("Company D", 12, 450);

        // Create parent companies (composites)
        ParentCompany westRegion = new ParentCompany("West Region");
        westRegion.addCompany(companyA);
        westRegion.addCompany(companyB);

        ParentCompany eastRegion = new ParentCompany("East Region");
        eastRegion.addCompany(companyC);
        eastRegion.addCompany(companyD);

        // Create top-level parent company
        ParentCompany headquarters = new ParentCompany("Headquarters");
        headquarters.addCompany(westRegion);
        headquarters.addCompany(eastRegion);

        // Calculate maintenance cost uniformly
        System.out.println("\n========== MAINTENANCE COST CALCULATION
    ==========");
        double totalCost = headquarters.calculateMaintenanceCost();
        System.out.println("\n========== GRAND TOTAL: $" + totalCost +
                           " ==========");
    }
}
```

3 Exercise 3: Payment System Integration

3.1 Question 1: Appropriate Design Pattern

We should use the **Adapter Pattern** .

3.2 Question 2: Participants and Class Diagram

Participants:

- **Target**: `PaymentProcessor` (the interface expected by clients)
- **Adaptee 1**: `QuickPay` (existing service with incompatible interface)
- **Adaptee 2**: `SafeTransfer` (existing service with incompatible interface)
- **Adapter 1**: `QuickPayAdapter` (adapts QuickPay to PaymentProcessor)
- **Adapter 2**: `SafeTransferAdapter` (adapts SafeTransfer to PaymentProcessor)
- **Client**: Uses PaymentProcessor interface to process payments

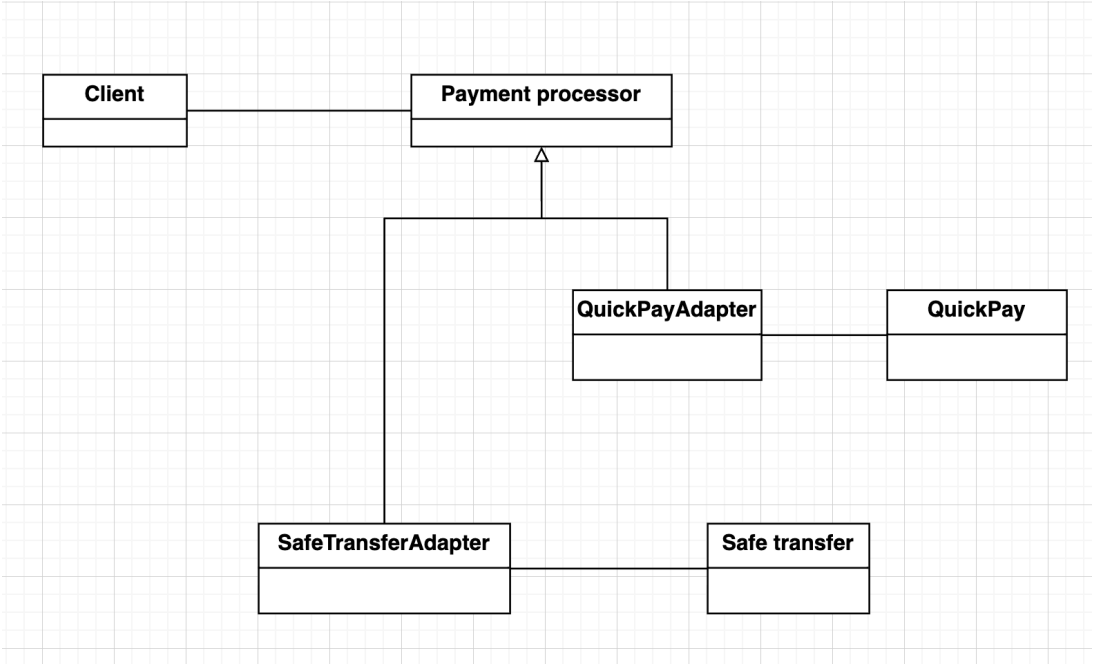

Figure 3: Class diagram for Payment System using Adapter Pattern

3.3 Question 3: Java Implementation

3.3.1 Target Interface

```
// Target interface - what the client expects
public interface PaymentProcessor {
    void payByCreditCard(double amount);
    void payByPayPal(double amount);
    void refund(double amount);
}
```

3.3.2 Adaptee Classes (Given)

```
// 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)
    ;
    }
}
```

```java
// 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)
    ;
    }
}
```

3.3.3 Adapter Classes

```java
// Adapter for QuickPay
public class QuickPayAdapter implements PaymentProcessor {
    private QuickPay quickPay;

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

    @Override
    public void payByCreditCard(double amount) {
        // Adapt the interface
        quickPay.creditCardPayment(amount);
    }

    @Override
    public void payByPayPal(double amount) {
        // Adapt the interface
        quickPay.paypalPayment(amount);
    }

    @Override
    public void refund(double amount) {
        // Adapt the interface
        quickPay.reverseTransaction(amount);
    }
}
```

```java
// Adapter for SafeTransfer
public class SafeTransferAdapter implements PaymentProcessor {
    private SafeTransfer safeTransfer;

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

    @Override
    public void payByCreditCard(double amount) {
```

```
        // Adapt the interface
        safeTransfer.payWithCard(amount);
    }

    @Override
    public void payByPayPal(double amount) {
        // Adapt the interface
        safeTransfer.payWithPayPal(amount);
    }

    @Override
    public void refund(double amount) {
        // Adapt the interface
        safeTransfer.refundPayment(amount);
    }
}
```

3.3.4 Client/Main Class

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

        // Use QuickPay through adapter
        System.out.println("--- Using QuickPay Service ---");
        PaymentProcessor quickPayProcessor = new QuickPayAdapter();
        quickPayProcessor.payByCreditCard(100.00);
        quickPayProcessor.payByPayPal(50.00);
        quickPayProcessor.refund(25.00);

        System.out.println();

        // Use SafeTransfer through adapter
        System.out.println("--- Using SafeTransfer Service ---");
        PaymentProcessor safeTransferProcessor = new SafeTransferAdapter
    ();
        safeTransferProcessor.payByCreditCard(200.00);
        safeTransferProcessor.payByPayPal(75.00);
        safeTransferProcessor.refund(30.00);

        System.out.println("\n
    ================================================");
    }
}
```

4 Exercise 4: GUI Dashboard with Reactive Components

4.1 Question 1: Appropriate Design Pattern

The appropriate design pattern is **Observer Pattern** .

4.2 Question 2: Class Diagram

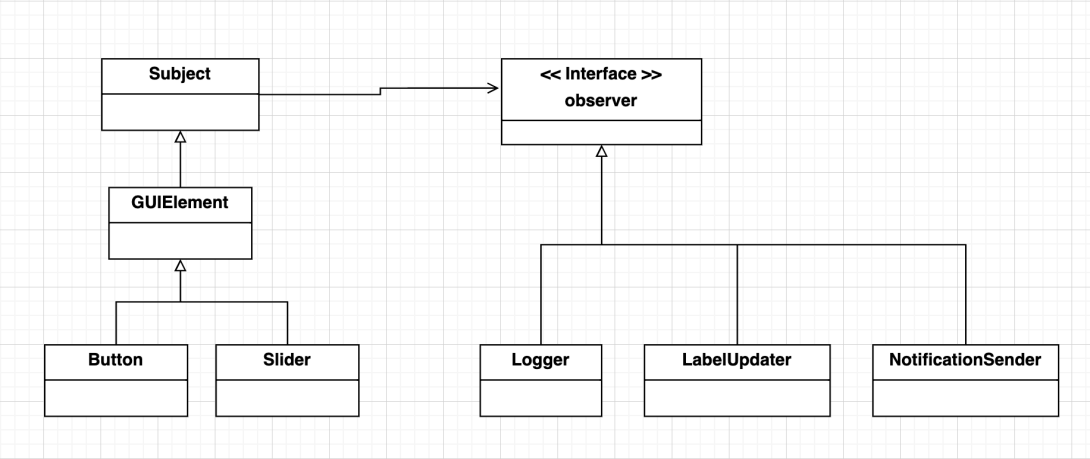

Figure 4: Class diagram for GUI Dashboard using Observer Pattern

4.3 Question 3: Java Implementation

4.3.1 Observer Interface

```
// Observer interface
public interface Observer {
    void update(String elementName, String action);
}
```

4.3.2 Subject Interface

```
import java.util.List;

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

4.3.3 Concrete Observers

```
// Concrete Observer: Logger
public class Logger implements Observer {
    @Override
    public void update(String elementName, String action) {
        System.out.println("Logger: Logging interaction - " +
                           elementName + " " + action);
    }
}
```

```java
// Concrete Observer: LabelUpdater
public class LabelUpdater implements Observer {
    @Override
    public void update(String elementName, String action) {
        System.out.println("LabelUpdater: Updating label - Last action: " +
                           elementName + " " + action);
    }
}
```

```java
// Concrete Observer: NotificationSender
public class NotificationSender implements Observer {
    @Override
    public void update(String elementName, String action) {
        System.out.println("NotificationSender: Sending alert for " +
                           elementName + " " + action);
    }
}
```

4.3.4 Abstract GUI Element (Subject)

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

// Abstract GUI Element (Concrete Subject base)
public abstract class GUIElement implements Subject {
    private List<Observer> observers;
    protected String name;

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

    @Override
    public void attach(Observer observer) {
        observers.add(observer);
        System.out.println("Attached observer to " + name);
    }

    @Override
    public void detach(Observer observer) {
        observers.remove(observer);
        System.out.println("Detached observer from " + name);
    }

    @Override
    public void notifyObservers(String action) {
        System.out.println("\n--- " + name + " " + action + " ---");
        for (Observer observer : observers) {
            observer.update(name, action);
        }
        System.out.println();
    }
}
```

4.3.5 Concrete GUI Elements

```java
// Concrete Subject: Button
public class Button extends GUIElement {

    public Button(String name) {
        super(name);
    }

    public void click() {
        System.out.println(name + " was clicked!");
        notifyObservers("clicked");
    }
}
```

```java
// Concrete Subject: Slider
public class Slider extends GUIElement {
    private int value;

    public Slider(String name) {
        super(name);
        this.value = 0;
    }

    public void setValue(int value) {
        this.value = value;
        System.out.println(name + " moved to value: " + value);
        notifyObservers("moved to " + value);
    }

    public int getValue() {
        return value;
    }
}
```

4.3.6 Client/Main Class

```java
// Client
public class GUIDashboardClient {
    public static void main(String[] args) {
        System.out.println("========== GUI Dashboard System ==========\n
    ");

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

        // Create GUI elements
        Button submitButton = new Button("SubmitButton");
        Button cancelButton = new Button("CancelButton");
        Slider volumeSlider = new Slider("VolumeSlider");
        Slider brightnessSlider = new Slider("BrightnessSlider");

        // Attach observers to SubmitButton
        System.out.println("--- Setting up SubmitButton ---");
        submitButton.attach(logger);
```

```
        submitButton.attach(labelUpdater);

        // Attach observers to VolumeSlider
        System.out.println("\n--- Setting up VolumeSlider ---");
        volumeSlider.attach(logger);
        volumeSlider.attach(notificationSender);

        // Attach observers to CancelButton
        System.out.println("\n--- Setting up CancelButton ---");
        cancelButton.attach(logger);
        cancelButton.attach(labelUpdater);
        cancelButton.attach(notificationSender);

        // Simulate user interactions
        System.out.println("\n========== User Interactions ==========");

        submitButton.click();

        volumeSlider.setValue(75);

        cancelButton.click();

        brightnessSlider.attach(logger);
        brightnessSlider.setValue(50);

        System.out.println("========================================")
    ;
     }
}
```