Design Patterns: Part 2 - Solutions

Mohamed AIT LAHCEN

November 27, 2025

1 Exercise 1: Navigation System with Strategy Pattern

1.1 Task 1: Class Diagram

- strategy: RouteStrategy

+ setStrategy(RouteStrategy)
+ executeRoute(String, String) ; (interface) [draw, rectangle, right of=navigator, xshift=4cm] **¡¡interface¿¿**
RouteStrategy

+ calculateRoute(String, String) ; (walking) [draw, rectangle, below of=interface, xshift=-2cm, yshift=-1.5cm] **WalkingStrategy**

+ calculateRoute(String, String) ; (car) [draw, rectangle, right of=walking, xshift=2cm]
CarStrategy

+ calculateRoute(String, String) ; (bike) [draw, rectangle, right of=car, xshift=2cm]
BikeStrategy

+ calculateRoute(String, String) ;
[-¿] (client) – (navigator); [-¿] (navigator) – (interface); [dashed,-¿] (walking) – (interface);

[dashed,-¿] (car) – (interface); [dashed,-¿] (bike) – (interface);

1.2 Questions

1. What role does the Navigator class play?

The Navigator class acts as the Context in the Strategy pattern. It holds a reference to a RouteStrategy and delegates the route calculation to the current strategy without knowing implementation details.

2. Why does Navigator depend on the RouteStrategy interface?

Navigator depends on RouteStrategy interface to achieve loose coupling. This allows changing strategies at runtime without modifying Navigator's code, following the dependency inversion principle.

3. Which SOLID principles are applied?

- Open/Closed Principle: Navigator is open for extension (new strategies) but closed for modification
- Dependency Inversion Principle: Navigator depends on abstraction (RouteStrategy) not concrete implementations
- Single Responsibility Principle: Each strategy class has one responsibility

1.3 Task 2: Java Implementation

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

public class WalkingStrategy implements RouteStrategy {
    @Override
    public void calculateRoute(String start, String destination) {
        System.out.println("Walking route from " + start + " to " + destination);
        System.out.println("Taking pedestrian paths, estimated time: 45 mins");
    }
}

public class CarStrategy implements RouteStrategy {
    @Override
    public void calculateRoute(String start, String destination) {
        System.out.println("Car route from " + start + " to " + destination);
        System.out.println("Taking highways, estimated time: 15 mins");
    }
}

public class BikeStrategy implements RouteStrategy {
    @Override
    public void calculateRoute(String start, String destination) {
        System.out.println("Bike route from " + start + " to " + destination);
        System.out.println("Taking bike lanes, estimated time: 25 mins");
    }
}

public class Navigator {
    private RouteStrategy strategy;

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

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

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

        navigator.setStrategy(new WalkingStrategy());
        navigator.executeRoute("Home", "Office");

        System.out.println();

        navigator.setStrategy(new CarStrategy());
        navigator.executeRoute("Home", "Office");

        System.out.println();

        navigator.setStrategy(new BikeStrategy());
        navigator.executeRoute("Home", "Office");
    }
}
```

2 Exercise 2: Vehicle Maintenance System

2.1 Design Pattern Selection

The best suited pattern is the **Composite Pattern**. This pattern allows treating individual objects and compositions uniformly, perfect for representing company hierarchies where parent companies contain independent companies.

2.2 Class Diagram

+ calculateMaintenanceCost(): double ; (leaf) [draw, rectangle, below left of=component, xshift=-1.5cm, yshift=-1.5cm] **IndependentCompany**

- vehicleCount: int
- unitCost: double

+ calculateMaintenanceCost(): double ; (composite) [draw, rectangle, below right of=component, xshift=1.5cm, yshift=-1.5cm] **ParentCompany**

- subsidiaries: List¡Company¿

+ addCompany(Company)
+ removeCompany(Company)
+ calculateMaintenanceCost(): double ;
[dashed,-¿] (leaf) – (component); [dashed,-¿] (composite) – (component); [-¿] (composite) – node[right] contains (component);

2.3 Java Implementation

```java
public interface Company {
    double calculateMaintenanceCost();
}

public class IndependentCompany implements Company {
    private String name;
    private int vehicleCount;
    private double unitCost;

    public IndependentCompany(String name, int vehicleCount,
       double unitCost) {
        this.name = name;
        this.vehicleCount = vehicleCount;
        this.unitCost = unitCost;
    }

    @Override
    public double calculateMaintenanceCost() {
        return vehicleCount * unitCost;
    }

    public String getName() {
        return name;
    }
}

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

    public String getName() {
        return name;
    }
}

public class MaintenanceTest {
    public static void main(String[] args) {
        IndependentCompany company1 = new IndependentCompany("SubCo␣A", 10, 500);
        IndependentCompany company2 = new IndependentCompany("SubCo␣B", 15, 450);
        IndependentCompany company3 = new IndependentCompany("SubCo␣C", 8, 600);

        ParentCompany parent = new ParentCompany("Main␣Corporation");
        parent.addCompany(company1);
        parent.addCompany(company2);
        parent.addCompany(company3);

        System.out.println("Company1␣cost:␣$" + company1.calculateMaintenanceCost());
        System.out.println("Total␣parent␣company␣cost:␣$" + parent.calculateMaintenanceCost());
    }
}
```

3 Exercise 3: Payment System Integration

3.1 Design Pattern Selection

The **Adapter Pattern** should be used. It allows incompatible interfaces to work together by wrapping existing classes with a new interface that clients expect.

3.2 Participants and Class Diagram

Participants:

- Target: PaymentProcessor interface
- Adaptee: QuickPay and SafeTransfer classes
- Adapter: QuickPayAdapter and SafeTransferAdapter
- Client: Uses PaymentProcessor interface

+ payByCreditCard(double)
+ payByPayPal(double)
+ refund(double) ; (adapter1) [draw, rectangle, below left of=target, xshift=-2cm, yshift=-2cm] **QuickPayAdapter**

- quickPay: QuickPay

+ payByCreditCard(double)
+ payByPayPal(double)
+ refund(double) ; (adapter2) [draw, rectangle, below right of=target, xshift=2cm, yshift=-2cm] **SafeTransferAdapter**

- safeTransfer: SafeTransfer

+ payByCreditCard(double)
+ payByPayPal(double)
+ refund(double) ; (adaptee1) [draw, rectangle, below of=adapter1, yshift=-1.5cm] **QuickPay**

+ creditCardPayment(double)
+ paypalPayment(double)
+ reverseTransaction(double) ; (adaptee2) [draw, rectangle, below of=adapter2, yshift=-1.5cm] **SafeTransfer**

+ payWithCard(double)
+ payWithPayPal(double)
+ refundPayment(double) ;

[dashed,-¿] (adapter1) – (target); [dashed,-¿] (adapter2) – (target); [-¿] (adapter1) – (adaptee1); [-¿] (adapter2) – (adaptee2);

3.3 Java Implementation

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

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);
    }
}

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);
    }
}

public class QuickPayAdapter implements PaymentProcessor {
    private QuickPay quickPay;

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

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

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

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

public class SafeTransferAdapter implements PaymentProcessor {
    private SafeTransfer safeTransfer;

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

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

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

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

public class PaymentClient {
    public static void main(String[] args) {
        PaymentProcessor quickPayProcessor = new QuickPayAdapter(new QuickPay());
        quickPayProcessor.payByCreditCard(100.0);
        quickPayProcessor.payByPayPal(50.0);
        quickPayProcessor.refund(25.0);

        System.out.println();

        PaymentProcessor safeTransferProcessor = new SafeTransferAdapter(new SafeTransfer());
        safeTransferProcessor.payByCreditCard(200.0);
        safeTransferProcessor.payByPayPal(75.0);
```

```
            safeTransferProcessor.refund(30.0);
        }
}
```

4 Exercise 4: GUI Dashboard

4.1 Design Pattern Selection

The **Observer Pattern** is most suitable. It allows multiple objects to be notified when a subject changes state, perfect for GUI components that need to react to user interactions.

Why? The pattern decouples the GUI elements (subjects) from the components that react to them (observers), allowing flexible addition/removal of observers without modifying the subjects.

4.2 Class Diagram

+ attach(Observer)
+ detach(Observer)
+ notifyObservers() ; (observer) [draw, rectangle, right of=subject, xshift=5cm]
¡¡interface¿¿
Observer

+ update(String) ; (button) [draw, rectangle, below of=subject, xshift=-2cm, yshift=-2cm] **Button**

- observers: List¡Observer¿
- name: String

+ click() ; (slider) [draw, rectangle, right of=button, xshift=3cm] **Slider**

- observers: List¡Observer¿
- name: String

+ move(int) ; (logger) [draw, rectangle, below of=observer, xshift=-2.5cm, yshift=-3cm]
Logger

+ update(String) ; (label) [draw, rectangle, right of=logger, xshift=2.5cm]
LabelUpdater

+ update(String) ; (notif) [draw, rectangle, right of=label, xshift=2.5cm]
NotificationSender

+ update(String) ;
[dashed,-¿] (button) – (subject); [dashed,-¿] (slider) – (subject); [-¿] (button) – (observer); [-¿] (slider) – (observer); [dashed,-¿] (logger) – (observer); [dashed,-¿] (label) – (observer);

[dashed,-¿] (notif) – (observer);

4.3 Java Implementation

```
public interface Observer {
    void update(String message);
}

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

public class Button implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String name;

    public Button(String name) {
        this.name = name;
    }

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

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

    @Override
    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }

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

public class Slider implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String name;

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

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

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

    @Override
    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }

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

public class Logger implements Observer {
    @Override
    public void update(String message) {
        System.out.println("Logger: Logging action - " + message);
    }
}

public class LabelUpdater implements Observer {
    @Override
    public void update(String message) {
        System.out.println("LabelUpdater: Updating label with - " + message);
    }
}

public class NotificationSender implements Observer {
    @Override
    public void update(String message) {
        System.out.println("NotificationSender: Sending alert for " + message);
    }
}

public class Dashboard {
    public static void main(String[] args) {
        Button submitButton = new Button("SubmitButton");
        Button cancelButton = new Button("CancelButton");
        Slider volumeSlider = new Slider("VolumeSlider");

        Logger logger = new Logger();
        LabelUpdater labelUpdater = new LabelUpdater();
        NotificationSender notificationSender = new
          NotificationSender();

        submitButton.attach(logger);
        submitButton.attach(labelUpdater);

        volumeSlider.attach(logger);
        volumeSlider.attach(notificationSender);

        submitButton.click();
        System.out.println();

        volumeSlider.move(75);
        System.out.println();

        cancelButton.attach(logger);
        cancelButton.click();
    }
}
```