Design Patterns: Part 2

Imane Fouad, UM6P

Exercise 1: Flexible Navigation with the Strategy Pattern

1- The strategy design pattern

Define a family of algorithms, encapsulate each one, and make them **interchangeable**. Strategy lets the algorithm vary **independently** from clients that use it.

Participants:

- **Strategy (interface)**: Defines a common method that all algorithms must implement, so the Context can use them interchangeably.
- **ConcreteStrategy**: Implements the algorithm in a specific way.
- **Context**: Holds a reference to a Strategy and delegates execution to it. It can switch strategies at runtime.
- **Client**: Chooses which ConcreteStrategy to use and provides it to the Context

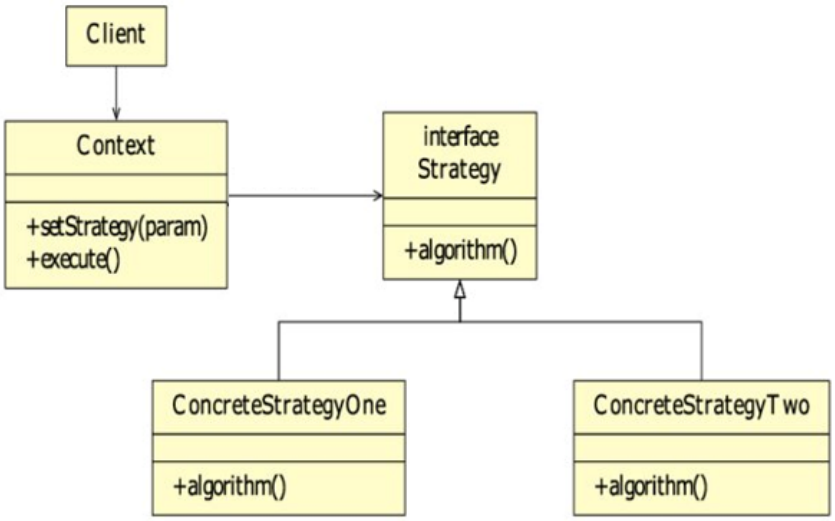

Figure 1: Class diagram of strategy design pattern

Several precisions:

- The Context must hold a reference to the Strategy interface.
- The Strategy interface defines the common operation(s) that all concrete strategies implement.
- Each Concrete Strategy must implement the interface and provide its own algorithm.

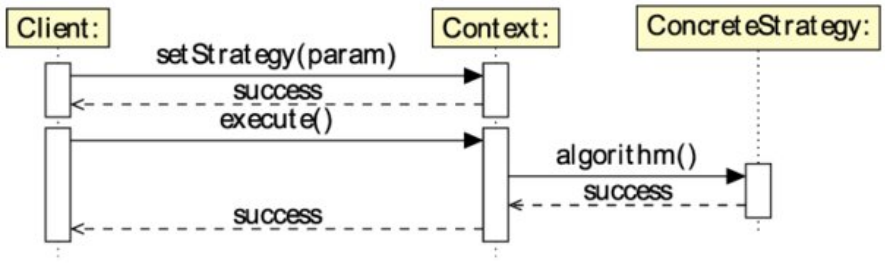

Figure 2: Sequence diagram of strategy design pattern

- The Context delegates execution of the algorithm to the Strategy, it does not implement the algorithm itself.
- The Client selects which concrete strategy to use and provides it to the Context.
- It should be possible to swap the strategy at runtime by giving the Context a different strategy object.

2- Scenario

You are building a Navigation System (like Google Maps or Waze). The system must be able to calculate a route using different strategies:

- WalkingStrategy
- CarStrategy
- BikeStrategy (optional extension)

Each strategy uses a different algorithm to compute the route. A Navigator allows the client to choose a travel strategy. The client sets the strategy, and the Navigator executes the route using the chosen method.

The user (client) should be able to change the routing strategy at runtime without modifying the Navigator's internal logic.

The goal is to apply the **Strategy Pattern** to make the system flexible and maintainable.

Task 1: Build the corresponding class diagram with respect to the Strategy design pattern.

> **Questions:**
>
> 1. What role does the Navigator class play?
> 2. Why does Navigator depend on the RouteStrategy interface?
> 3. Which SOLID principles are applied in this design?

Task 2: Implement the solution using Java.

Exercise 2:

Within a vehicle sales system, we want to represent client companies, in particular to offer them maintenance packages for the vehicles they own. To do this, we need to calculate the maintenance cost of their vehicles, which depends on the number of vehicles a company owns, and the unit maintenance cost per vehicle Companies can either be: Independent, or Parent companies to which we can add independent companies. It must be possible to calculate the maintenance cost for both parent companies and independent companies in a uniform way.

1. Which design pattern is best suited to model this problem?
2. Build the class diagram.
3. Implement the solution in Java.

Exercise 3:

You are developing a modern e-commerce platform. The system expects a standard interface for payment processing:

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

However, the platform must integrate with two existing third-party payment services, each with a different API:

Payment Service A: QuickPay

```java
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);
    }
}
```

Payment Service B: SafeTransfer

```
1
2  public class SafeTransfer {
3      public void payWithCard(double amount) {
4          System.out.println("SafeTransfer: Paying with credit card $" +
               amount);
5      }
6
7      public void payWithPayPal(double amount) {
8          System.out.println("SafeTransfer: Paying with PayPal $" + amount);
9      }
10
11     public void refundPayment(double amount) {
12         System.out.println("SafeTransfer: Refunding payment $" + amount);
13     }
14 }
```

1. You want the user to be able to use the new PaymentProcessor interface without modifying the existing payment services. Which design pattern should you use?
2. Identify different participants and build the corresponding class diagram for your solution.
3. Implement the solution in Java.

Exercise 4:

We are building a GUI dashboard that has multiple interactive elements: Buttons (e.g., Submit, Cancel) and Sliders (e.g., Volume control, Brightness control) Multiple components need to react to changes in these GUI elements: Logger (logs user interactions), LabelUpdater (updates a GUI label showing the last action), NotificationSender (pops up alerts for important interactions).

Example Actions

Clicking SubmitButton → Notifies Logger and LabelUpdater.

Moving VolumeSlider → Notifies Logger and NotificationSender.

You need to ensure that each component reacts efficiently and promptly to changes in the buttons and sliders.

Note: You do not need to implement actual GUI components. For this exercise, it is sufficient for each component to print messages to the console to simulate their behavior. *Exp: NotificationSender: Sending alert for VolumeSlider*

1. Which design pattern is most suitable for this scenario and why?
2. Build the corresponding class diagram.
3. Implement the solution in Java.