## 6.19 Mẫu thiết kế STRATEGY

### 6.19.1 Mở đầu

Mẫu thiết kế *Strategy* trông đơn giản nhất trong các mẫu thiết kế, nó cung cấp tính linh hoạt rất lớn cho code của bạn. Mẫu này được sử dụng hầu khắp mọi nơi, ngay cả trong việc kết hợp với các mẫu thiết kế khác. Các mẫu mà chúng ta đã bàn trước đây có quan hệ với mẫu này, hoặc trực tiếp hoặc gián tiếp. Sau này này bạn sẽ nhận được ý tưởng về mẫu này quan trọng như thế nào.

Để hiểu mẫu thiết kế *Strategy*, giả sử chúng ta muốn tạo định dạng văn bản cho trình soan thảo văn bản. Mỗi người cần phải quan tâm đến trình soạn thảo văn bản. Một trình soạn thảo văn bản có các định dạng văn bản khác nhau để định dạng văn bản. Chúng ta có thể tạo các định dạng văn bản khác nhau và sau đó truyền cái được yêu cầu cho trình soạn thảo văn bản, như vậy trình soạn thảo có khả năng định dạng văn bản như đã yêu cầu.

Trình soạn thảo văn bản *(text editor*) sẽ giữ tham chiếu đến giao diện chung của định dạng văn bản (*text formatter*) và công việc của trình soạn thảo là sẽ truyền *text* cho *formatter* để định dạng *text*.

Giả sử chúng ta cài đặt điều này sử dụng mẫu thiết kế *Strategy* mà sẽ là cho code linh hoạt và dễ bảo trì. Nhưng trước đó, chúng ta sẽ tìm hiểu thêm về mẫu thiết kế *Strategy*.

### 6.19.2 Mẫu thiết kế Strategy là gì

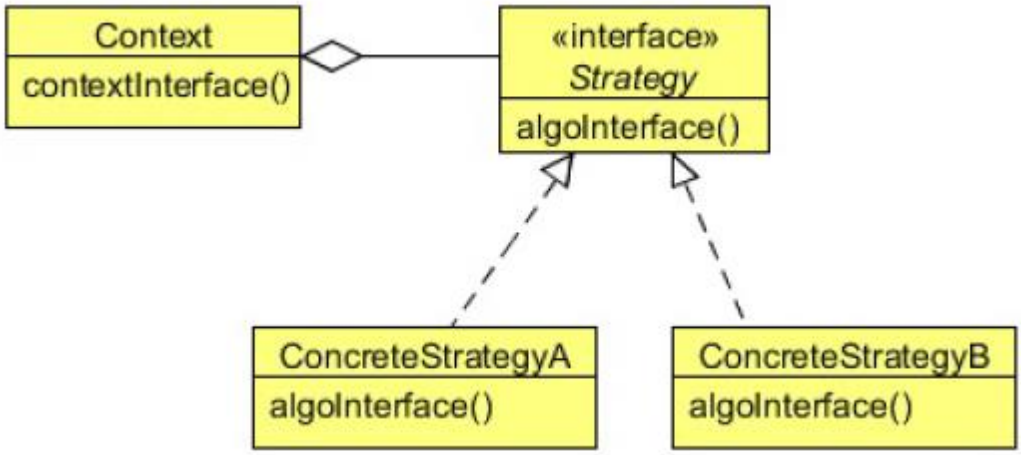
Mẫu thiết kế *Strategy* định nghĩa một họ các thuật toán, đóng gói mỗi một trong chúng và làm cho chúng có thể trao đổi nhau được. Chiến lược *Strategy* cho phép thuật toán đa dạng độc lập khỏi *client* mà sử dụng nó.

Mẫu *Strategy* là hữu ích khi có một tập các thuật toán liên quan và đối tượng *client* cần có khả năng lựa chọn động một thuật toán từ tập đó mà phù hợp với yêu cầu hiện tại. Mẫu *Strategy* đề xuất giữ cài đặt của mỗi thuật toán trong một lớp riêng. Mỗi thuật toán này đóng gói trong một lớp riêng được tham chiếu như một chiến lược *Strategy*. Một đối tượng mà sử dụng đối tượng *Strategy* thường được tham chiếu như *context object*.

Với các đối tượng *Strategy* khác nhau, thay đổi hành vi cho đối tượng *Context* đơn giản là thay đổi hành vi đối tượng *Strategy* của nó đến cái mà cài đặt thuật toán mong muốn. Để cho phép đối tượng *Context* truy cập đến các đối tượng *Strategy* khác nhau theo cách trơn tru, mọi đối tượng *Strategy* cần phải được thiết kế để có cùng một giao diện. Trong lập trình Java, điều này được thực hiện bằng cách thiết kế mỗi đối tượng *Strategy* hoặc như cài đặt của một giao diện chung hoặc như lớp con của lớp trừu tượng chung mà khai báo giao diện chung được yêu cầu.

Một lớp các thuật toán liên quan được đóng gói trong tập các lớp *Strategy* trong một phân cấp lớp, mỗi *client* có thể chọn trong số các thuật toán đó bằng cách lấy và khởi tạo một lớp *Strategy* phù hợp. Để thay đổi hành vi của *Context*, đối tượng client cần cấu hình *context* với khởi tạo

được chọn. Kiểu bố trí này hoàn toàn tách cài đặt thuật toán ra khỏi *context* mà sử dụng nó. Kết quả là khi cài đặt thuật toán đang có được thay đổi hoặc thuật toán mới được bổ sung vào nhóm, cả hai *context* và đối tượng *client* mà sử dụng *context* đều không bị ảnh hưởng.




**Strategy**

- Khai báo giao diện chung cho mọi thuật toán hỗ trợ. *Context* sử dụng giao diện này để gọi thuật toán được định nghĩa bởi *ConcreteStrategy*.

**ConcreteStrategy**

- Cài đặt thuật toán sử dụng giao diện *Strategy*.

**Context**

- Được cấu hình với đối tượng *ConcreteStrategy*.
- Bảo trì tham chiếu đến đối tượng *Strategy*.
- Có thể định nghĩa một giao diện mà cho phép *Strategy* truy cập đến dữ liệu của nó.

### 6.19.3 Cài đặt mẫu thiết kế Strategy

Dưới đây là giao diện định dạng văn bản *text formatter* mà được cài đặt bởi mọi *concrete formatter*.

```
package com.javacodegeeks.patterns.strategypattern;

public interface TextFormatter {

        public void format(String text);

}
```

Giao diện trên đây chứa chỉ một phương thức *format*, được dùng để định dạng văn bản.

```
package com.javacodegeeks.patterns.strategypattern;

public class CapTextFormatter implements TextFormatter{

	@Override
	public void format(String text) {
		System.out.println("[CapTextFormatter]: "+text.toUpperCase());
	}

}
```

Lớp trên đây *CapTextFormatter*, là một *concrete formatter* mà cài đặt giao diện *TextFormatter* và lớp này được sử dụng để thay đổi văn bản thành chữ in hoa.

```
package com.javacodegeeks.patterns.strategypattern;

public class LowerTextFormatter implements TextFormatter{

	@Override
	public void format(String text) {
		System.out.println("[LowerTextFormatter]: "+text.toLowerCase());
	}

}
```

*LowerTextFormatter*, là một *concrete formatter* mà cài đặt giao diện *TextFormatter* và lớp này được sử dụng để thay đổi văn bản thành chữ viết nhỏ.

```
package com.javacodegeeks.patterns.strategypattern;

public class TextEditor {

	private final TextFormatter textFormatter;

	public TextEditor(TextFormatter textFormatter){
		this.textFormatter = textFormatter;
	}

	public void publishText(String text){
		textFormatter.format(text);
	}

}
```

Lớp trên là lớp *TextEditor*, mà giữ tham chiếu đến giao diện *TextFormatter*. Lớp này chứa phương thức *publishText* mà chuyển tiếp *text* đến cho *Formatter* để in văn bản theo định dạng mong muốn.

Bây giờ, chúng ta sẽ kiểm tra code trên.

```
package com.javacodegeeks.patterns.strategypattern;

public class TestStrategyPattern {

        public static void main(String[] args) {
                TextFormatter formatter = new CapTextFormatter();
                TextEditor editor = new TextEditor(formatter);
                editor.publishText("Testing text in caps formatter");

                formatter = new LowerTextFormatter();
                editor = new TextEditor(formatter);
                editor.publishText("Testing text in lower formatter");

        }

}
```

Code trên sẽ in ra kết quả sau:

```
[CapTextFormatter]: TESTING TEXT IN CAPS FORMATTER
[LowerTextFormatter]: testing text in lower formatter
```

Trong lớp trên, trước hết chúng ta tạo *CapTextFormatter* và gán nó cho khởi tạo *TextEditor*. Sau đó chúng ta gọi phương thức *publishText* để truyền *text* đầu vào cho nó.

Một lần nữa, chúng ta lại làm lại như vậy, nhưng lần này đối tượng *LowerTextFormatter* sẽ được truyền cho *TextEditor*.

Đầu ra rõ ràng chỉ ra rằng, định dạng văn bản khác nhau đã được tạo ra bởi các *text editor* khác nhau vì nó sử dụng các *text formatter* khác nhau.

Ưu điểm chính của mẫu thiết kế *Strategy* là chúng ta nâng cao code mà không phải lo lắng gì nhiều. Chúng ta có thể bổ sung *text Formatter* mới mà không ảnh hưởng đến code hiện thời. Điều này sẽ làm cho code của chúng ta được bảo trì và linh hoạt. Mẫu thiết kế này cũng cung cấp nguyên lý thiết kế "*mã cho giao diện*" - "*code to interface*".

### 6.19.4 Khi nào sử dụng mẫu thiết kế Strategy

Sử dụng mẫu *Strategy* khi

- Nhiều lớp liên quan khác nhau chỉ ở hành vi của chúng. Các chiến lược *Strategies* cung cấp cách cấu hình một lớp với một trong nhiều hành vi.
- Bạn cần các phương án khác nhau của một thuật toán. Chẳng hạn, bạn cần định nghĩa thuật toán phản ánh các cân bằng không gian – thời gian khác nhau. Các chiến lược *Strategies* có thể được sử dụng khi các phương án được cài đặt như một phân cấp lớp của thuật toán.
- Một thuật toán sử dụng dữ liệu mà *client* không cần biết về nó. Sử dụng mẫu *Strategy* để tránh mở thuật toán phức tạp chuyên biệt cho cấu trúc dữ liệu.
- Một lớp định nghĩa nhiều hành vi, và nó trông như lệnh có đa điều kiện trong thao tác của nó. Thay vì nhiều điều kiện, chuyển các nhánh điều kiện liên quan vào lớp *Strategy* riêng của chúng.