6.21 Mẫu thiết kế INTERPRETER

6.21.1 Mở đầu

Mẫu thiết kế *Interpreter* là mẫu có trọng trách cao (heavy-duty). Nó gộp tất cả những gì đi kèm với ngôn ngữ lập trình của riêng bạn lại hoặc kiểm soát cái đã có bằng cách tạo bộ thông dịch *interpreter* cho ngôn ngữ đó. Để sử dụng mẫu này, bạn cần biết một chút về văn phạm hình thức đi kèm với một ngôn ngữ. Như bạn có thể hình dung, nó là một trong số các mẫu mà người phát triển thực tế không sử dụng hàng ngày, vì việc tạo một ngôn ngữ của riêng bạn không phải là việc nhiều người làm.

Chẳng hạn, định nghĩa biểu thức trong ngôn ngữ mới của bạn có thể trông giống như đoạn sau trong ngôn ngữ hình thức:

```
expression ::=<command> | <repetition> | <sequence>
```

Mỗi biểu thức trong ngôn ngữ của bạn, khi đó, có thể được tạo nên từ các biểu thức lệnh command, phép lặp repetition hoặc các biểu thức tuần tự. Mỗi mục có thể được thể hiện như một đối tượng với phương thức *interpret* để dịch ngôn ngữ mới của bạn vào một cái gì đó bạn có thể chạy trên Java.

Để minh họa sử dụng mẫu thiết kế *Interpreter* giả sử chúng ta tạo một ví dụ để tính một biểu thức toán đơn giản, nhưng trước đó, chúng ta sẽ tìm hiểu thêm về mẫu *Interpreter* trong mục sau.

6.21.2 Mẫu thiết kế Interpreter là gì

Cho một ngôn ngữ, định nghĩa một thể hiện cho một văn phạm của nó với một bộ thông dịch mà sử dụng thể hiện đó để dịch các câu trong ngôn ngữ đó.

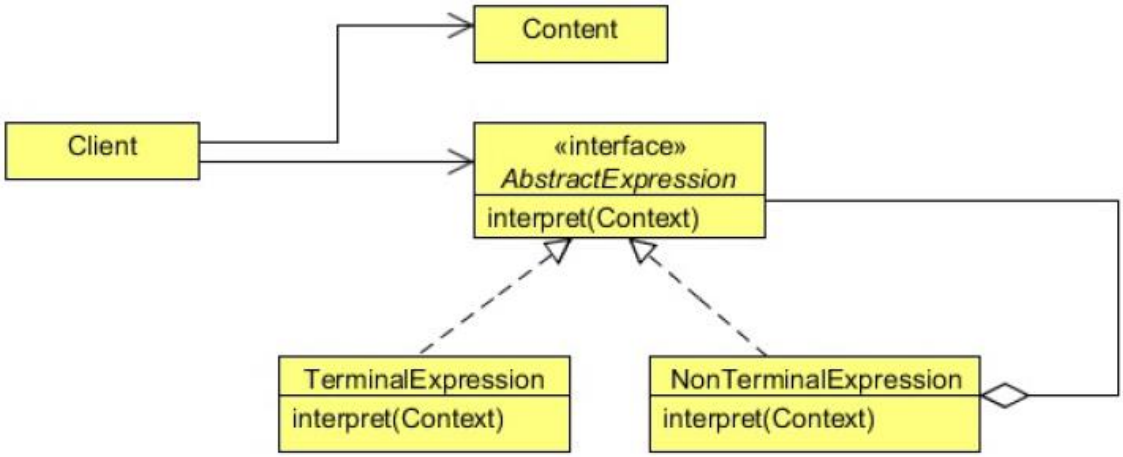
Nó chung, ngôn ngữ là được sinh ra từ tập các qui tắc văn phạm. Các câu khác nhau có thể được tạo nên bởi việc tuân theo các qui tắc văn phạm này. Đôi khi một ứng dụng có thể cần để xử lý các xuất hiện lặp của các yêu cầu tương tự mà là sự kết hợp của tập các qui tắc văn phạm. Các yêu cầu này là khác nhau nhưng là tương tự theo nghĩa chúng được tạo nên từ cùng một tập các qui tắc.

Một ví dụ đơn giản của điều trên có thể là tập các biểu thức số học khác nhau được đưa vào chương trình tính toán. Mặc dù mỗi biểu thức như vậy là khác nhau, chúng đều được tạo nên sử dụng các qui tắc cơ bản mà hợp lại là bộ văn phạm của ngôn ngữ các biểu thức số học.

Trong các trường hợp như vậy, thay vì xử lý mỗi sự kết hợp khác nhau của các qui tắc như một trường hợp riêng biệt, ứng dụng sẽ có ích hơn nếu có khả năng thông dịch sự kết hợp tổng quan của các qui tắc. Mẫu *Interpreter* có thể được sử dụng để thiết kế khả năng như vậy trong một ứng dụng sao cho các ứng dụng và người sử dụng khác có thể đặc tả các thao tác sử dụng ngôn ngữ đơn giản được định nghĩa bởi tập các qui tắc văn phạm.

Một phân cấp lớp có thể được thiết kế để biểu diễn tập các qui tắc văn phạm với mỗi lớp trong phân cấp đó biểu diễn một qui tắc văn phạm riêng biệt. Một *module Interpreter* có thể được thiết kế để thông dịch các câu được tạo nên sử dụng phân cấp lớp được thiết kế bên trên và tiến hành các thao tác cần thiết.

Bởi vì lớp khác nhau biểu diễn một qui tắc văn phạm, nên số các lớp tăng lên cùng số qui tắc văn phạm. Một ngôn ngữ với các qui tắc văn phạm phức tạp và mở rộng đòi hỏi số lớp rất lớn. Mẫu *Interpreter* làm việc tốt nhất khi văn phạm là đơn giản. Có văn phạm đơn giản sẽ tránh sự cần thiết phải có nhiều lớp tương ứng với một tập phức tạp các qui tắc kéo theo mà là khó quản trị và bảo trì.



AbstractExpression

- Khai báo một thao tác *Interpret* trừu tượng mà là chung cho mọi đỉnh trên cây cú pháp trừu tượng.

TerminalExpression

- Cài đặt một thao tác *Interpret* tượng ứng với ký hiệu tận cùng trong văn phạm.
- Một khởi tạo được đòi hỏi cho mỗi ký hiệu tận cùng trong một câu.

NonterminalExpression

- Một lớp như vậy được yêu cầu cho mỗi qui tắc R ::= R_1 R_2 …R_n trong văn phạm.
- Bảo trì các biến khởi tạo của kiểu *AbstractExpression* cho mỗi ký hiệu R_1 cho đến R_n.
- Cài đặt một thao tác *Interpret* cho mỗi ký hiệu chưa tận trong văn phạm. *Interpret* thông thường gọi đệ qui đến chính nó trên các biến biểu diễn R_1 đến R_n.

Context

- Chứa thông tin mà là tổng thể cho bộ thông dịch *Interpreter*.

Client

- Xây dựng một cây cú pháp trừu tượng (hoặc là cho trước) biểu diễn một câu cụ thể trong ngôn ngữ mà văn phạm định nghĩa. Cây cú pháp trừu tượng được lắp ghép từ các khởi tạo của các lớp *NonterminalExpression* và *TerminalExpression.*
- Triệu gọi thao tác *Interpret.*

6.21.3 Cài đặt thiết kế mẫu Interpreter

```
package com.javacodegeeks.patterns.interpreterpattern;

public interface Expression {
        public int interpret();
}
```

Giao diện trên được sử dụng bởi mọi biểu thức cụ thể khác nhau và ghi đè phương thức *Interpret* để định nghĩa thao tác chuyên biệt của chúng trên biểu thức đó.

Sau đây là các lớp biểu thức chuyên biệt của thao tác.

```
package com.javacodegeeks.patterns.interpreterpattern;

public class Add implements Expression{

        private final Expression leftExpression;
        private final Expression rightExpression;

        public Add(Expression leftExpression,Expression rightExpression ){
                this.leftExpression = leftExpression;
                this.rightExpression = rightExpression;
        }
        @Override
        public int interpret() {
                return leftExpression.interpret() + rightExpression.interpret();
        }

}
```

```
package com.javacodegeeks.patterns.interpreterpattern;

public class Product implements Expression{

        private final Expression leftExpression;
        private final Expression rightExpression;

        public Product(Expression leftExpression,Expression rightExpression ){
                this.leftExpression = leftExpression;
                this.rightExpression = rightExpression;
        }
        @Override
        public int interpret() {
                return leftExpression.interpret() * rightExpression.interpret();
        }
}
```

```
package com.javacodegeeks.patterns.interpreterpattern;

public class Substract implements Expression{

        private final Expression leftExpression;
        private final Expression rightExpression;

        public Substract(Expression leftExpression,Expression rightExpression ){
                this.leftExpression = leftExpression;
                this.rightExpression = rightExpression;
        }
        @Override
        public int interpret() {

                return leftExpression.interpret() - rightExpression.interpret();
        }

}
```

```
package com.javacodegeeks.patterns.interpreterpattern;

public class Number implements Expression{

        private final int n;

        public Number(int n){
                this.n = n;
        }
        @Override
        public int interpret() {
                return n;
        }

}
```

Dưới đây là lớp tiện ích tùy chọn mà chứa các phương thức tiện ích khác nhau được dùng để tính toán biểu thức.

```
package com.javacodegeeks.patterns.interpreterpattern;

public class ExpressionUtils {

	public static boolean isOperator(String s) {
		if (s.equals("+") || s.equals("-") || s.equals("*"))
			return true;
		else
			return false;
	}

	public static Expression getOperator(String s, Expression left, Expression right) {
		switch (s) {
		case "+":
			return new Add(left, right);
		case "-":
			return new Substract(left, right);
		case "*":
			return new Product(left, right);
		}
		return null;
	}

}
```

Bây giờ chúng ta kiểm tra ví dụ trên.

```
package com.javacodegeeks.patterns.interpreterpattern;

import java.util.Stack;

public class TestInterpreterPattern {

	public static void main(String[] args) {

		String tokenString = "7 3 - 2 1 + *";
		Stack<Expression> stack = new Stack<>();
		String[] tokenArray = tokenString.split(" ");
		for (String s : tokenArray) {

			if (ExpressionUtils.isOperator(s)) {
				Expression rightExpression = stack.pop();
				Expression leftExpression = stack.pop();
				Expression operator = ExpressionUtils.getOperator(s,  ↩
				    leftExpression,rightExpression);
				int result = operator.interpret();
				stack.push(new Number(result));
			} else {
				Expression i = new Number(Integer.parseInt(s));
				stack.push(i);
			}
		}
		System.out.println("( "+tokenString+" ): "+stack.pop().interpret());

	}


}
```

Code trên sẽ cho ra kết quả sau.

```
( 7 3 -2 1 + * ):12
```

Lưu ý rằng chúng ta đã sử dụng biểu thức hậu tố để tính toán.

Nếu bạn không biết về hậu tố, dưới đây là tóm tắt thông tin về nó. Có ba cách ký hiệu cho biểu thức toán học: trung tố, hậu tố và tiền tố.

- Ký hiệu trung tố là ký hiệu số hoc và logic chung, mà ở đó phép toán được viết ở giữa các toán hạng mà nó thao tác như 3 + 4.
- Hậu tố hay ký hiệu Balan viết ngược là cách ký hiệu toán học mà ở đó mỗi phép toán được viết sau các toán hạng của nó như 3 4 +.
- Tiền tố hay ký hiệu Balan là ký hiệu toán học mà ở đó mỗi phép toán được viết trước các toán hạng của nó như + 3 4.

Ký hiệu trung tố thường được sử dụng trong biểu thức toán học. Hai ký hiệu khác được sử dụng như cú pháp cho biểu thức toán học cho bởi Interpreter cho các ngôn ngữ lập trình.

Trong lớp trên, chúng ta khai báo hậu tố của một biểu thức trong biến *tokenString*. Sau đó ta tách *tokenString* và gán nó cho một mảng, *tokenArray*. Khi lặp các token từng cái một, trước hết chúng ta kiểm tra xem *token* là phép toán hay toán hạng. Nếu *token* là toán hạng, ta truyền nó cho ngăn xếp, nhưng nếu nó là phép toán ta kéo hai toán hạng trên cùng ra khỏi ngăn xếp. Phương thức *getOperation* từ *ExpressionUtils* trả về lớp biểu thức tương ứng tương ứng với phép toán được truyền cho nó.

Sau đó, chúng ta thông dịch nhận kết quả và đẩy lại ngăn xếp. Sau khi lặp xong *tokenList* chúng ta nhận được kết quả cuối cùng.

6.21.4 Khi nào sử dụng mẫu thiết kế Interpreter

Sử dụng mẫu *Interprete*r khi có một ngôn ngữ để thông dịch, và bạn có thể biểu diễn các câu trong ngôn ngữ đó như các cây cú pháp trừu tượng. Mẫu *Interpreter* làm việc tốt nhất khi

- Văn phạm là đơn giản. Đối với văn phạm phức tạp, phân cấp lớp cho văn phạm trở nên lớn và khó quản trị. Các công cụ như bộ sinh bộ phân tích là lựa chọn tốt hơn trong trường hợp này. Chúng có thể thông dịch các biểu thức mà không cần xây dựng cây cú pháp trừu tượng mà có thể tiết kiệm không gian và thời gian.
- Tính hiệu quả không là vấn đề quan trọng ở đây Hầu hết các bộ thông dịch hiệu quả thường không được cài đặt bởi thông dịch cây cú pháp trực tiếp, nhưng trước hết dịch chúng sang dạng khác. Chẳng hạn, các biểu thức chính qui thường được biến đổi vào máy trạng thái. Nhưng ngay cả khi đó, bộ biến đổi có thể được cài đặt bởi mẫu *Interpreter*, như vậy mẫu này vẫn được áp dụng.