02

DSLs e o Interpreter

Muitas vezes temos problemas que são bem representados por uma árvore. Suponha que devamos fazer uma calculadora científica, que é composta por diversas operações, desde as mais simples até as mais complexas.

Por exemplo, imagine que seu sistema deve conseguir resolver a expressão "(2+3)-4/2". Ele deve ser responsável por entender essa string e interpretá-la.

É, com certeza um trabalho complicado. É algo parecido, por exemplo, com o que a JVM faz quando lê o bytecode. Você consegue pensar em uma implementação pra isso que seja simples? É difícil!

Dado que temos muitas expressões diferentes em uma calculadora, como por exemplo, adição, subtração, etc, precisamos achar uma maneira simples de lidar com essa complexidade, e a ideia de criar novas expressões.

Imagine então que representemos essas operações como classes. Pensando em abstrações, poderíamos ter uma interface `Expressao`, e criar classes `Soma` e `Subtração`, as implementam:

```
Expressao conta = new Soma(new Numero(10), new Numero(20));
```

Podemos ter expressões ainda mais complicadas:

```
Expressao esquerda = new Subtracao(new Numero(10), new Numero(5));
Expressao direita = new Soma(new Numero(2), new Numero(10));

Expressao conta = new Soma(esquerda, direita);
```

Veja que uma soma recebe no construtor duas outras "expressões", que podem ser um número ou mesmo uma outra expressão.

Vamos ver a implementação disso, em código:

```
public interface Expressao {

}

public class Subtracao implements Expressao {

    private Expressao esquerda;
    private Expressao direita;

    public Subtracao(Expressao esquerda, Expressao direita) {
        this.esquerda = esquerda;
        this.direita = direita;

    }
}

public class Soma implements Expressao {
```

```
    private Expressao esquerda;
    private Expressao direita;

    public Soma(Expressao esquerda, Expressao direita) {
        this.esquerda = esquerda;
        this.direita = direita;
    }
}
```

Veja que a classe `Soma` e a classe `Subtracao` cada uma guarda outras duas expressões dentro. O que faz sentido, já que a subtração subtrai o número da esquerda com o da direita. Análogo para a soma.

Precisamos também implementar a classe `Numero`, que representa um simples número:

```
public class Numero implements Expressao {

    private int numero;
    public Numero(int numero) {
        this.numero = numero;
    }
}
```

Excelente. Já temos a nossa estrutura de dados pronta. Ela consegue bem representar qualquer expressão que queremos. Basta instanciar os objetos corretos.

O próximo passo agora é interpretar essa "árvore". Já que todos os elementos devem ser interpretados; podemos garantir isso colocando o método `avalia()` na interface, que devolve um inteiro (afinal, esse é o resultado do nosso processamento).

```
public interface Expressao {
    int avalia();
}
```

Vamos começar pela classe `Numero`. Interpretar um `Numero` é fácil: basta retornar ele próprio.

```
public class Numero implements Expressao {

    private int numero;
    public Numero(int numero) {
        this.numero = numero;
    }

    @Override
    public int avalia() {
        return numero;
    }
}
```

Já o da soma ou da subtração, é um pouco mais complicado. Precisamos "interpretar" o lado da esquerda primeiro, depois o da direita, e aí sim fazer a conta:

```
public class Soma implements Expressao {

    private Expressao esquerda;
    private Expressao direita;

    public Soma(Expressao esquerda, Expressao direita) {
        this.esquerda = esquerda;
        this.direita = direita;
    }

    @Override
    public int avalia() {
        int resultadoDaEsquerda = esquerda.avalia();
        int resultadoDaDireita = direita.avalia();
        return resultadoDaEsquerda + resultadoDaDireita;
    }
}

public class Subtracao implements Expressao {

    private Expressao esquerda;
    private Expressao direita;

    public Subtracao(Expressao esquerda, Expressao direita) {
        this.esquerda = esquerda;
        this.direita = direita;

    }

    @Override
    public int avalia() {
        int resultadoDaEsquerda = esquerda.avalia();
        int resultadoDaDireita = direita.avalia();
        return resultadoDaEsquerda - resultadoDaDireita;
    }
}
```

Veja que cada `avalia()` agora sabe interpretar sua própria expressão, e expressões podem avaliar outras expressões, podemos tentar calcular o resultado de uma expressão inteira.

```
public class Programa {

    public static void main(String[] args) {

        Expressao esquerda = new Subtracao(new Numero(10), new Numero(5));
        Expressao direita = new Soma(new Numero(2), new Numero(10));

        Expressao conta = new Soma(esquerda, direita);

        int resultado = conta.avalia();
        System.out.println(resultado);
    }
}
```

Veja que nossa árvore consegue interpretar, e calcular o resultado final. Quando temos expressões que devem ser avaliadas, e a transformamos em uma estrutura de dados, e depois fazemos com que a própria árvore se avalie, damos o nome de **Interpreter**.

O padrão é bastante útil quando temos que implementar interpretadores para DSLs, ou coisas similares. É um padrão bem complicado, mas bastante interessante.