02

Estruturas de dados e o Visitor

No capítulo anterior, montamos expressões da nossa calculadora científica de uma maneira bem orientada a objetos. Veja, por exemplo, o código abaixo:

```
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);
```

Atualmente esse nosso código consegue interpretar a expressão (ou expressões aninhadas) e dar o resultado final.

Mas agora imagine que precisemos navegar nessa árvore de elementos para fazer outras coisas. Coisas essas como, por exemplo, imprimir de uma maneira formatada a estrutura da árvore. Ou seja, para a expressão acima, precisamos imprimir: **((10 - 5) + (2 + 10))**.

Cada nó da árvore (`Soma` , `Subtracao` , `Numero`) tem uma saída diferente. Por exemplo, imprimir um `Numero` basta imprimir o seu conteúdo. Imprimir uma soma, precisamos imprimir um "(", e aí imprimir o nó da esquerda (que pode ser uma outra expressão), depois o sinal de "+", depois a expressão da direita, e por fim, um ")".

Mas poderíamos também ter outras maneiras de imprimir essa árvore, como usando notação pré-fixa. Precisamos então encontrar uma maneira genérica de navegar por essa árvore, e fazer coisas diferentes, dependendo do "navegador".

A primeira classe que criaremos então é uma classe que sabe o que fazer, para cada tipo de nó:

```
public class Impressora {

    public void visitaSoma(Soma soma) {
        System.out.print("(");
        VISITA ( esquerda  ) ;
        System.out.print(" + ");
        VISITA ( direita  ) ;
        System.out.print(")");

    }

    public void visitaSubtracao(Subtracao subtracao) {
        System.out.print("(");
        VISITA ( esquerda  ) ;
        System.out.print(" - ");
        VISITA ( direita  ) ;
        System.out.print(")");
    }

    public void visitaNumero(Numero numero) {
        System.out.print(numero.getNumero());

    }
}
```

Veja só, cada método faz exatamente como falamos. Mas temos um problema, não conseguimos saber como visitar os nós da esquerda e direita, pois não sabemos qual o tipo do nó que está lá dentro!

Precisamos de alguma forma fazer com que, se o nó for `Soma`, ele deve executar o `visitaSoma`, e assim por diante.

Uma maneira de resolver isso, é criar um método dentro de cada `Expressao`, que invocará o método certo na `Impressora`:

```
public interface Expressao {
    int avalia();
    void aceita(Visitor visitor);
}

public class Subtracao implements Expressao {
    // todo codigo aqui

    @Override
    public void aceita(Impressora visitor) {
        visitor.visitaSubtracao(this);
    }
}

public class Soma implements Expressao {
    // todo codigo aqui

    @Override
    public void aceita(Impressora visitor) {
        visitor.visitaSoma(this);
    }
}

public class Numero implements Expressao {
    // todo codigo aqui

    @Override
    public void aceita(Impressora visitor) {
        visitor.visitaNumero(this);
    }
}
```

Veja que cada método `aceita()` invoca o seu respectivo método. Dizemos então que a expressão "está aceitando" o visitor.

Com isso funcionando, podemos agora corrigir a Impressora, pois agora basta invocar o `aceita()`, e a classe se encarregará de chamar o método certo:

```
public class ImpressoraVisitor {

    public void visitaSoma(Soma soma) {
        System.out.print("(");
        soma.getEsquerda().aceita(this);
        System.out.print(" + ");
        soma.getDireita().aceita(this);
        System.out.print(")");
```

```
    }

    public void visitaSubtracao(Subtracao subtracao) {
        System.out.print("(");
        subtracao.getEsquerda().aceita(this);
        System.out.print(" - ");
        subtracao.getDireita().aceita(this);
        System.out.print(")");
    }

    public void visitaNumero(Numero numero) {
        System.out.print(numero.getNumero());

    }
}
```

Pronto! Basta testar:

```
        Visitor visitor = new Impressora();
        conta.aceita(visitor);
```

Quando temos uma árvore, e precisamos navegar nessa árvore de maneira organizada, podemos usar um **Visitor**, que foi o padrão de projeto implementado nessa aula.

É comum inclusive que o código faça sempre referência a uma interface de Visitor, e não de uma classe concreta. Assim, conseguimos trocar facilmente o visitor que visitará a árvore:

```
public interface Visitor {
    void visitaSoma(Soma soma);
    void visitaSubtracao(Subtracao subtracao);
    void visitaNumero(Numero numero);
}

public class Impressora implements Visitor {
   // ...
}

// Visitor e nao mais Impressora
public void aceita(Visitor visitor) {
  // ...
}
```