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 = Subtracao(Numero(10), Numero(5))
expressao_direita = Soma(Numero(2), Numero(10))
expressao_conta = Soma(expressao_esquerda, expressao_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:

```
class Impressora(object):

    def visita_soma(self, soma):

        print '(',
        # visita expressao_esquerda de Soma
        print '+',
        # visita expressao_direita de Soma
        print ')',

    def visita_subtracao(self, subtracao):
        print '(',
        # visita expressao_esquerda da Subtracao
        print '-',
        # visita expressao_direita da Subtracao
        print ')',

    def visita_numero(self, numero):

        print numero.avalia(),
```

Não sabemos o tipo de nó? E agora? Como interpretar?

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 `visita_soma`, e assim por diante.

Visitando cada nó

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

```
class Subtracao(object):

    def __init__(self, expressao_esquerda, expressao_direita):
        self.__expressao_esquerda = expressao_esquerda
        self.__expressao_direita = expressao_direita

    def avalia(self):
        return (self.__expressao_esquerda.avalia()
            - self.__expressao_direita.avalia())

    @property
    def expressao_esquerda(self):
        return self.__expressao_esquerda

    @property
    def expressao_direita(self):
        return self.__expressao_direita

    def aceita(self, visitor):
        visitor.visita_subtracao(self)

class Soma(object):

    def __init__(self, expressao_esquerda, expressao_direita):
        self.__expressao_esquerda = expressao_esquerda
        self.__expressao_direita = expressao_direita

    def avalia(self):
        return (self.__expressao_esquerda.avalia()
            + self.__expressao_direita.avalia())

    def aceita(self, visitor):
        visitor.visita_soma(self)

    @property
    def expressao_esquerda(self):
        return self.__expressao_esquerda

    @property
    def expressao_direita(self):
        return self.__expressao_direita

class Numero(object):

    def __init__(self, numero):
        self.__numero = numero

    def avalia(self):
        return self.__numero
```

```
    def aceita(self, visitor):
        visitor.visita_numero(self)
```

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:

```
class Impressora(object):

    def visita_soma(self, soma):

        print '(',
        soma.expressao_esquerda.aceita(self)
        print '+',
        soma.expressao_direita.aceita(self)
        print ')',

    def visita_subtracao(self, subtracao):
        print '(',
        subtracao.expressao_esquerda.aceita(self)
        print '-',
        subtracao.expressao_direita.aceita(self)
        print ')',

    def visita_numero(self, numero):

        print numero.avalia(),
```

Pronto! Basta testar:

```
if __name__ == '__main__':

    expressao_esquerda = Subtracao(Numero(10), Numero(5))
    expressao_direita = Soma(Numero(2), Numero(10))
    expressao_conta = Soma(expressao_esquerda, expressao_direita)

    visitor = Impressora()
    expressao_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.

Quantos visitors desejarmos

Podemos criar quantos visitors desejarmos, até mesmo um que exiba a expressão por extenso, por exemplo. Porém, para que o novo visitor funcione, ele precisa ter os métodos `visita_soma`, `visita_subtracao` e `visita_numero`. Sabemos que pela natureza dinâmica da linguagem Python e do Duck Tying, o método `aceita` da expressões pode aceitar qualquer objeto que tenham esses métodos. Porém, nada nos impede de criarmos uma classe abstrata que contenha esses métodos como

abstratos, obrigando a todos que herdarem a classe a implementarem esses métodos. Aliás, usamos bastante classes abstratas no primeiro treinamento de Design Patterns.