02

Comportamentos compostos por outros comportamentos e o Decorator

Nos capítulos anteriores criamos nossa abstração para os impostos através da hierarquia que começa pela interface `Imposto` e por meio dela podemos calcular os impostos dos orçamentos. Um exemplo desses cálculos, é dado pelo código abaixo:

```
public class TesteDeImpostos {

    public static void main(String[] args) {
        Imposto iss = new ISS();

        Orcamento orcamento = new Orcamento(500.0);

        double valor = iss.calcula(orcamento);

        System.out.println(valor);
    }
}
```

No entanto, o exemplo demonstra um simples cálculo. Em muitos projetos, pode ser necessário criar comportamentos que sejam compostos por outros comportamentos. Um exemplo seria calcularmos o ICMS em cima do ISS, como no código abaixo:

```
public class TesteDeImpostos {

  public static void main(String[] args) {
        Imposto issComIcms = new ISSComICMS();

        Orcamento orcamento = new Orcamento(500.0);

        double valor = issComIcms.calcula(orcamento);

        System.out.println(valor);
    }
}
```

Repare que para isso, tivemos que criar uma nova classe, chamada `ISSComICMS`. A codificação dessa classe seria uma simples sequência de cálculos envolvendo os dois impostos. No entanto, o mais importante a se notar nesse ponto, é o fato de que sempre que quisermos cálculos compostos de impostos, temos que criar uma nova classe. Ou seja, se quisermos calcular ISS com o fictício ICCP, criaríamos uma classe `ISSComICCP`. Imagine se tivermos 3 impostos: serão 8 combinações; 4 impostos já seriam 16! Trabalhoso demais de implementar! Com isso, percebemos que nosso código é pouco flexível, pois teríamos que saber de antemão as várias combinações possíveis de impostos e criarmos uma classe para cada.

Pensando no momento do uso das classes de impostos e tendo em vista que queremos ter diferentes combinações de impostos sem precisarmos criar classes novas, poderíamos fazer os impostos, opcionalmente, depender de outro imposto. Assim, teríamos algo como:

```
public class TesteDeImpostos {

  public static void main(String[] args) {
        Imposto impostoComplexo = new ISS(new ICMS());

        Orcamento orcamento = new Orcamento(500.0);

        double valor = impostoComplexo.calcula(orcamento);

        System.out.println(valor);
    }
}
```

No código acima, note que não precisamos criar mais nenhuma classe e apenas compomos os comportamentos com as que já possuíamos. Mas o código acima ainda não compila! A classe `ISS` precisa receber outros impostos. E se quiséssemos fazer o processo inverso, ou seja, calcular o ISS, em cima do ICMS, precisaríamos que a classe `ICMS` recebesse também outros impostos. E pela flexibilidade, deveria poder ser recebido qualquer imposto.

Com isso, percebemos que em nosso caso, todos os impostos, podem receber outros impostos. Podemos explicitar isso através de construtores na classe Imposto. Caso você possua uma interface chamada `Imposto`, nesse momento, pode-se transformá-la em uma classe abstrata, para podermos adicionar os construtores:

```
public abstract class Imposto {

    protected final Imposto outroImposto;
    public Imposto(Imposto outroImposto) {
        this.outroImposto = outroImposto;
    }

    public abstract double calcula(Orcamento orcamento);
}
```

O próximo passo é fazermos as classes filhas de `Imposto` também terem os construtores, delegando para o construtor de `Imposto`. Para isso, um exemplo é a classe ISS:

```
public class ISS extends Imposto {

    public ISS(Imposto outroImposto) {
        super(outroImposto);
    }

    public double calcula(Orcamento orcamento) {
        return orcamento.getValor() * 0.06;
    }

}
```

Nesse momento, já conseguimos fazer com que os impostos recebam outros impostos, mas ainda não estamos compondo os cálculos. Nesse caso do ISS, queremos que o `calcula`, devolva os 6% do orçamento mais o resultado do cálculo do outro imposto. Com isso, nosso método `calcula` precisa também do outro imposto, então, vamos utilizá-lo:

```
public class ISS extends Imposto {

    public ISS(Imposto outroImposto) {
        super(outroImposto);
    }

    public double calcula(Orcamento orcamento) {
        return orcamento.getValor() * 0.06 + calculoDoOutroImposto(orcamento);
    }

    private double calculoDoOutroImposto(Orcamento orcamento) {
      return outroImposto.calcula(orcamento);
    }

}
```

Esse método `calculoDoOutroImposto` será o mesmo em todos os impostos. Para evitar repetição de código, ele pode estar na classe `Imposto`, garantindo que todos que herdam de `Imposto` possam chamá-lo para delegar para o outro imposto, caso ele tenha sido definido. Nesse caso, a implementação na classe `Imposto` seria:

```
public abstract class Imposto {

    // construtor e atributo

    protected double calculoDoOutroImposto(Orcamento orcamento) {
      return outroImposto.calcula(orcamento);
    }

    public abstract double calcula(Orcamento orcamento);
}
```

Mas e no final, quando não existirem mais impostos a serem calculados? O `outroImposto` será nulo? Devemos permitir que um imposto não tenha mais um próximo imposto a ser calculado. Para isso, vamos adicionar um construtor default na classe `Imposto`, e fazer com que o método `calculoDoOutroImposto()` agora trate o caso de não haver um próximo:

```
public abstract class Imposto {

    private final Imposto outroImposto;

    public Imposto(Imposto outroImposto) {
        this.outroImposto = outroImposto;
    }

  // construtor default
  public Imposto() {
    this.outroImposto = null;
  }

    protected double calculoDoOutroImposto(Orcamento orcamento) {
      // tratando o caso do proximo imposto nao existir!
      if(outroImposto == null) return 0;
      return outroImposto.calcula(orcamento);
    }
```

```
    public abstract double calcula(Orcamento orcamento);
}
```

Repare que o método `calculoDoOutroImposto` invoca o método `calcula` caso outro imposto tenha sido definido. Caso contrário, apenas retorna 0, não influenciando na composiçao do cálculo.

Por questões de implementação, precisaremos escrever construtores default em todos os nossos impostos agora para que todos eles funcionem de maneira adequada.

Pronto, agora conseguimos inclusive compor o comportamento. Ao executarmos o código `new ISS(new ICMS())` o `ICMS` será guardado como o outro imposto e quando o cálculo for realizado, o método `calculoDoOutroImposto` que está sendo chamado no `calcula` se encarregará de invocar o cálculo do outro imposto.

Quando compomos comportamento, através de classes que recebem objetos do mesmo tipo que elas mesmas (nesse caso, `ISS` que é um `Imposto`, recebe em seu construtor outro `Imposto`) para fazerem parte de seu comportamento, de uma maneira que seu uso é definido a partir do que se passou para a instanciação dos objetos, é o que caracteriza o Design Pattern chamado Decorator.