02

Muitos Descontos e o Chain of Responsibility

Nosso orçamento pode receber um desconto de acordo com o tipo da venda que será efetuada. Por exemplo, se o cliente comprou mais de 5 ítens, ele recebe 10% de desconto; se ele fez uma compra casada de alguns produtos, recebe 5% de desconto, e assim por diante.

Em uma implementação tipicamente procedural, teríamos algo do tipo:

```
public class CalculadorDeDescontos {
  public double calcula(Orcamento orcamento) {
    // verifica primeira regra de possível desconto
    if(orcamento.getItens().size() > 5) {
      return orcamento.getValor() * 0.1;
    }

    // verifica segunda regra de possível desconto
    else if(orcamento.getValor() > 500.0) {
      return orcamento.getValor() * 0.07;
    }

    // verifica terceira, quarta, quinta regra de possível desconto ...
    // um monte de ifs daqui pra baixo
  }
}
```

Ver esse monte de `if` s em sequência nos lembra o capítulo anterior, na qual extraímos cada um deles para uma classe específica. Vamos repetir o feito, já que com classes menores, o código se torna mais simples de entender:

```
public class DescontoPorMaisDeCincoItens {
  public double desconta(Orcamento orcamento) {
    if(orcamento.getItens().size() > 5) {
      return orcamento.getValor() * 0.1;
    }
    else {
      return 0;
    }
  }
}
```

```
public class DescontoPorMaisDeQuinhentosReais {
  public double desconta(Orcamento orcamento) {
    if(orcamento.getValor() > 500) {
      return orcamento.getValor() * 0.07;
    }
    else {
      return 0;
    }
  }
}
```

Veja que tivemos que sempre colocar um `return 0;` , afinal o método precisa sempre retornar um double. Vamos agora substituir o conjunto de `if` s na classe `CalculadorDeDesconto` . Repare que essa classe procura pelo desconto que deve ser aplicado; caso o anterior não seja válido, tenta o próximo.

```
public class CalculadorDeDescontos {
  public double calcula(Orcamento orcamento) {
    // vai chamando os descontos na ordem até que algum deles dê diferente de zero...
    double desconto = new DescontoPorMaisDeCincoItens().desconta(orcamento);
    if(desconto == 0)
      desconto = new DescontoPorMaisDeQuinhentosReais().desconta(orcamento);
    if(desconto == 0)
      desconto = new ProximoDesconto().desconta(orcamento);
    // ...

    return desconto;
  }
}
```

O código já está um pouco melhor. Cada regra de negócio está em sua respectiva classe. O problema agora é como fazer essa sequência de descontos ser aplicada na ordem, pois precisamos colocar mais um `if` sempre que um novo desconto aparecer.

Precisávamos fazer com que um desconto qualquer, caso não deva ser executado, automaticamente passe para o próximo, até encontrar um que faça sentido. Algo do tipo:

```
public class DescontoPorMaisDeQuinhentosReais {
  public double desconta(Orcamento orcamento) {
    if(orcamento.getValor() > 500) {
      return orcamento.getValor() * 0.07;
    }
    else {
      return PROXIMO DESCONTO;
    }
  }
}
```

Todos os descontos têm algo em comum. Todos eles calculam o desconto dado um orçamento. Podemos criar uma abstração para representar um desconto genérico. Por exemplo:

```
public interface Desconto {
  double desconta(Orcamento orcamento);
}

public class DescontoPorMaisDeCincoItens implements Desconto { ... }
public class DescontoPorMaisDeQuinhentosReais implements Desconto { ... }
```

Para fazer aquele código funcionar agora, basta fazer com que todo desconto receba um próximo desconto! Observe o código abaixo:

```
public interface Desconto {
  double desconta(Orcamento orcamento);
```

```
  void setProximo(Desconto proximo);
}

public class DescontoPorMaisDeCincoItens implements Desconto {
  private Desconto proximo;

  public void setProximo(Desconto proximo) {
    this.proximo = proximo;
  }

  public double desconta(Orcamento orcamento) {
    if(orcamento.getItens().size() > 5) {
      return orcamento.getValor() * 0.1;
    }
    else {
      return proximo.desconta(orcamento);
    }
  }
}



public class DescontoPorMaisDeQuinhentosReais implements Desconto {
  private Desconto proximo;

  public void setProximo(Desconto proximo) {
    this.proximo = proximo;
  }

  public double desconta(Orcamento orcamento) {
    if(orcamento.getValor() > 500) {
      return orcamento.getValor() * 0.07;
    }
    else {
      return proximo.desconta(orcamento);
    }
  }
}
```

Ou seja, se o orçamento atende a regra de um desconto, o mesmo já calcula o desconto. Caso contrário, ele passa para o "próximo" desconto, qualquer que seja esse próximo desconto.

Basta agora plugarmos todas essas classes juntas. Veja que um desconto recebe um "próximo". Para o desconto, pouco importa qual é o próximo desconto. Eles estão totalmente desacoplados um do outro!

```
public class CalculadorDeDescontos {
  public double calcula(Orcamento orcamento) {
    Desconto d1 = new DescontoPorMaisDeCincoItens ();
    Desconto d2 = new DescontoPorMaisDeQuinhentosReais();

    d1.setProximo(d2);

    return d1.desconta(orcamento);
  }
}
```

```
public class TestaDescontos {
  public static void main(String[] args) {
    CalculadorDeDescontos calculador = new CalculadorDeDescontos();

        Orcamento orcamento = new Orcamento(500.0);
        orcamento.adicionaItem(new Item("CANETA", 250.0));
        orcamento.adicionaItem(new Item("LAPIS", 250.0));

        double desconto = calculador.calcula(orcamento);

        System.out.println(desconto);

  }
}
```

Esses descontos formam como se fosse uma "corrente", ou seja, um ligado ao outro. Daí o nome do padrão de projeto: Chain of Responsibility. A ideia do padrão é resolver problemas como esses: de acordo com o cenário, devemos realizar alguma ação. Ao invés de escrevermos código procedural, e deixarmos um único método descobrir o que deve ser feito, quebramos essas responsabilidades em várias diferentes classes, e as unimos como uma corrente.

Nosso problema é só fazer o algoritmo parar agora. Se ele não encontrar nenhum desconto válido, o valor deve ser 0. Vamos criar a classe `SemDesconto`, que será o fim da corrente.

```
public class SemDesconto implements Desconto {

    public double desconta(Orcamento orcamento) {
        return 0;
    }

    public void setProximo(Desconto desconto) {
        // nao tem!
    }
}

public class CalculadorDeDescontos {
  public double calcula(Orcamento orcamento) {
    Desconto d1 = new DescontoPorMaisDeCincoItens();
    Desconto d2 = new DescontoPorMaisDeQuinhentosReais();
    Desconto d3 = new SemDesconto();

    d1.setProximo(d2);
    d2.setProximo(d3);

    return d1.desconta(orcamento);
  }
}
```

A classe `SemDesconto` não atribui o próximo desconto, pois ela não possui um próximo. Na realidade, ela é o ponto final da nossa cadeia de responsabilidades.

Note também que nossa classe `Orcamento` cresceu, e agora recebe ítens também. A mudança é simples:

```
public class Orcamento {

    private double valor;
    private List<Item> itens;

    public Orcamento(double valor) {
        this.valor = valor;
        this.itens = new ArrayList<Item>();
    }

    public double getValor() {
        return valor;
    }

    public List<Item> getItens() {
        return Collections.unmodifiableList(itens);
    }

    public void adicionaItem(Item item) {
        itens.add(item);
    }

}
public class Item {

    private String nome;
    private double valor;

    public Item(String nome, double valor) {
        this.nome = nome;
        this.valor = valor;
    }

    public String getNome() {
        return nome;
    }

    public double getValor() {
      return valor;
    }

}
```