02

Estados que variam e o State

Nossos orçamentos podem ter diferentes estados durante o seu ciclo de vida. Um orçamento nasce "Em aprovação", e pode virar "Aprovado" ou "Reprovado". Ao final de todo o processo, deverá ser "Finalizado".

Dependendo do estado que o orçamento se encontra, algumas ações podem ser diferentes. Por exemplo, podemos adicionar um desconto extra ao orçamento: quando o orçamento está em aprovação, a empresa oferece 5% a mais de desconto; quando já está aprovado, a empresa oferece 2% de desconto. Orçamentos reprovados e finalizados não recebem nada de desconto extra.

Veja o código abaixo:

```
class TestesDoDescontoExtra {
  public static void main(String[] args) {
    Orcamento reforma = new Orcamento(500.0);

    reforma.aplicaDescontoExtra(); // resultado aqui depende da situação corrente do orçamer
  }
}
```

Em implementações mais procedurais, desenvolvedores geralmente optam por representar o estado do objeto por meio de constantes. Por exemplo:

```
class Orcamento {
  public static final int EM_APROVACAO = 1;
  public static final int APROVADO = 2;
  public static final int REPROVADO = 3;
  public static final int FINALIZADO = 4;

  // ... resto da classe aqui
}
```

E, no método que aplica o desconto, é comum a utilização de vários `ifs` para tal:

```
class Orcamento {
  public static final int EM_APROVACAO = 1;
  public static final int APROVADO = 2;
  public static final int REPROVADO = 3;
  public static final int FINALIZADO = 4;

  private double valor;
  private int estadoAtual;

  // ... resto da classe aqui

  public void aplicaDescontoExtra() {
    if(estadoAtual == EM_APROVACAO) valor = valor - (valor * 0.05);
    else if(estadoAtual == APROVADO) valor = valor - (valor * 0.02);
    else throw new RuntimeException("Orçamentos reprovados não recebem desconto extra!");
```

```
  }
}
```

Esse código tende a se tornar difícil de manter por vários motivos: 1) geralmente outros comportamentos dessa classe também variam de acordo com o estado do objeto; 2) a cada novo estado, um novo `if` deve ser acrescentado em todos os métodos do objeto.

Para eliminarmos esses vários `ifs` , precisamos primeiro separar cada ação de acordo com o estado em uma classe diferente. Por exemplo:

```
class EmAprovacao {
  // ...
}
class Aprovado {
  // ...
}
class Reprovado {
  // ...
}
class Finalizado {
  // ...
}
```

Todas elas são possíveis estados de um orçamento, o que nos leva a crer que poderíamos ter uma interface comum para todas elas. Todos esses estados devem também dar o desconto extra para o orçamento, de acordo com sua regra de negócio.

Vamos criar a interface e uma primeira implementação de estado, o estado `EmAprovacao` :

```
interface EstadoDeUmOrcamento {
  void aplicaDescontoExtra(Orcamento orcamento);
}

class EmAprovacao implements EstadoDeUmOrcamento {
  public void aplicaDescontoExtra(Orcamento orcamento) {
    orcamento.valor -= orcamento.valor * 0.05;
  }
}
```

Veja que ele dá um desconto extra de 5%, já que o estado atual do objeto é EM APROVAÇÃO. Veja agora a implementação do estado APROVADO:

```
class Aprovado implements EstadoDeUmOrcamento {
  public void aplicaDescontoExtra(Orcamento orcamento) {
    orcamento.valor -= orcamento.valor * 0.02;
  }
}
```

Veja que ele dá um desconto de 2% já que ele representa o desconto que pode ser dado quando o objeto estiver no estado APROVADO.

Os outros estados são parecidos:

```
class Reprovado implements EstadoDeUmOrcamento {
  public void aplicaDescontoExtra(Orcamento orcamento) {
    throw new RuntimeException("Orçamentos reprovados não recebem desconto extra!");
  }
}

class Finalizado implements EstadoDeUmOrcamento {
  public void aplicaDescontoExtra(Orcamento orcamento) {
    throw new RuntimeException("Orçamentos finalizados não recebem desconto extra!");
  }
}
```

Repare que todas as implementações recebem um `Orcamento`. Isso é necessário já que precisamos alterar o orçamento, e as classes de estado precisam enxergá-lo. Repare também que acessamos "diretamente" o atributo `valor`. Para que isso seja possível, precisamos colocá-lo como `protected`, e fazer com que todos esses estados estejam no mesmo pacote da classe `Orcamento`.

Precisamos agora fazer com que o `Orcamento` use essas classes para representar seu estado interno, e não mais as constantes. Além disso, precisamos fazer com que a classe que representa o estado do orçamento, como as classes `Aprovado`, `Reprovado` e `EmAprovacao`, respondam pelo comportamento de descontoExtra:

```
class Orcamento {
  protected double valor;
  protected EstadoDeUmOrcamento estadoAtual; // veja a mudança aqui

  public Orcamento() {
    this.estadoAtual = new EmAprovacao();
  }

  public void aplicaDescontoExtra() {
    estadoAtual.aplicaDescontoExtra(this);
  }
}
```

Podemos incrementar ainda mais nossa classe `Orcamento`, implementando a troca de estados. Por exemplo, se o orçamento está no estado EM APROVAÇÃO, ele pode ir apenas para os estados APROVADO e REPROVADO. Dos estados APROVADO e REPROVADO, podemos ir apenas para o estado FINALIZADO.

Representar isso em código procedural é difícil. Precisaríamos de várias condições (leia-se `ifs`), para alcançar o resultado esperado. O State nos ajuda nesse problema também. Basta representarmos as possíveis trocas em todas as classes que representam o estado. Para representar em todas as classes, precisamos alterar a interface:

```
interface EstadoDeUmOrcamento {
  void aplicaDescontoExtra(Orcamento orcamento);
  void aprova(Orcamento orcamento);
  void reprova(Orcamento orcamento);
  void finaliza(Orcamento orcamento);
}
```

Cada estado, por sua vez toma a decisão correta, e muda o estado do orçamento. Veja o estado `EmAprovacao` abaixo. Observe o método `aprova()` : do estado EM APROVAÇÃO, pode-se ir para o estado APROVADO. É isso que a classe implementa. Ela muda o estado do orçamento para APROVADO. Agora repare no método `finaliza()` . Não podemos ir para o estado FINALIZADO daqui, e por isso o método lança a exceção.

```
class EmAprovacao implements EstadoDeUmOrcamento {
  public void aplicaDescontoExtra(Orcamento orcamento) {
    orcamento.valor -= orcamento.valor * 0.05;
  }

  public void aprova(Orcamento orcamento) {
    // desse estado posso ir para o estado de aprovado
    orcamento.estadoAtual = new Aprovado();
  }

  public void reprova(Orcamento orcamento) {
    // desse estado posso ir para o estado de reprovado tambem
    orcamento.estadoAtual = new Reprovado();
  }

  public void finaliza(Orcamento orcamento) {
    // daqui não posso ir direto para finalizado
    throw new RuntimeException("Orcamento em aprovação não podem ir para finalizado diretame
  }
}
```

Veja o estado Aprovado, por exemplo:

```
class Aprovado implements EstadoDeUmOrcamento {
  public void aplicaDescontoExtra(Orcamento orcamento) {
    orcamento.valor -= orcamento.valor * 0.02;
  }

  public void aprova(Orcamento orcamento) {
    // jah estou em aprovacao
    throw new RuntimeException("Orçamento já está em estado de aprovação");
  }

  public void reprova(Orcamento orcamento) {
    // nao pode ser reprovado agora!
    throw new RuntimeException("Orçamento está em estado de aprovação e não pode ser reprova
  }

  public void finaliza(Orcamento orcamento) {
    // daqui posso ir para o estado de finalizado
    orcamento.estadoAtual = new Finalizado();
  }
}
```

O `Orcamento` por sua vez, sempre que recebe uma ação que depende do seu estado, repassa a chamada para o seu estado atual:

```
class Orcamento {
  protected double valor;
  protected EstadoDeUmOrcamento estadoAtual; // veja a mudança aqui

  public Orcamento() {
    this.estadoAtual = new EmAprovacao();
  }

  public void aplicaDescontoExtra() {
    estadoAtual.aplicaDescontoExtra(this);
  }

  public void aprova() {
    estadoAtual.aprova(this);
  }

  public void reprova() {
    estadoAtual.reprova(this);
  }

  public void finaliza() {
    estadoAtual.finaliza(this);
  }

}
```

Repare que eliminamos todos os `ifs` dessa classe e separamos as responsabilidades. Uma classe para cada possível estado do objeto. Além disso, repare que a criação de um novo "Estado" é fácil: basta criar uma nova classe que implementa `EstadoDeUmOrcamento` e ela funcionará sem muito esforço!

Olhe uma classe que testa essa implementação:

```
class TestesDoDescontoExtra {

  public static void main(String[] args) {
    Orcamento reforma = new Orcamento(500.0);

    reforma.aplicaDescontoExtra();
    System.out.println(reforma.getValor()); // imprime 475,00 pois descontou 5%
    reforma.aprova(); // aprova nota!

    reforma.aplicaDescontoExtra();
    System.out.println(reforma.getValor()); // imprime 465,50 pois descontou 2%

    reforma.finaliza();

    // reforma.aplicaDescontoExtra(); lancaria excecao, pois não pode dar desconto nesse est
    // reforma.aprova(); lança exceção, pois não pode ir para aprovado
  }
}
```

Essa é a grande graça da orientação a objetos! Classes pequenas e com responsabilidades bem definidas. E, com a ajuda do polimorfismo, podemos juntar esses comportamentos e formar um sistema maior.