02

A grande variedade de impostos e o padrão Strategy

Design Patterns

Classes e métodos gigantes? Cinco minutos para entender o que aquele método faz ou onde está o código que faz uma alteração simples? Diversas variáveis e diversos ifs e fors no mesmo método? Código complexo e obscuro? Toda vez que uma alteração aparece, você precisa mudar em 20 classes diferentes? Sim, problemas muito comuns do nosso dia-a-dia. Mas por que isso acontece?

Um fato que é conhecido sobre todo software é que ele, mais cedo ou mais tarde, vai mudar: novas funcionalidades aparecerão, outras deverão ser alteradas etc. O problema é que, geralmente, essas mudanças não são feitas de forma muito planejada. Por esse motivo, durante o desenvolvimento de um projeto de software, é bem comum a criação de código onde as responsabilidades se misturam e espalham por várias classes, fazendo com que a manutenção do código fique cada vez mais difícil, já que uma simples mudança pode obrigar o desenvolvedor a alterar diversas classes. Nesse cenário, temos uma situação onde o 'Design' das classes não está bom e é possível melhorá-lo.

Para atingir o objetivo da melhora do 'Design' e diminuir o custo de manutenção, existem algumas técnicas, onde podemos aplicar a orientação a objetos de uma maneira a simplificar o código escrito.

Resolveremos diversos dos problemas que vemos em códigos, como classes e métodos gigantes que fazem muita coisa, classes que precisam conhecer mais 10 outras classes para fazer seu trabalho, métodos complicados com muitos ifs e condições, mudanças que se propagam por várias classes e assim por diante.

Algumas das principais técnicas para atingir um bom 'Design' são tão comuns que foram catalogadas em um conjunto de alternativas para solucionar problemas de Design de código, chamados de Design Patterns, mais conhecido em português como os Padrões de Projetos, os quais, durante esse curso, aprenderemos a utilizar em um projeto. Um padrão de projeto nada mais é do que uma solução elegante para um problema que é muito recorrente em nosso dia-a-dia.

Mais importante do que entender como é a implementação de um padrão de projeto, é entender a motivação do padrão: em quais casos ele faz sentido e deve ser aplicado. Durante os próximos capítulos, observe o cenário do exemplo dado com atenção e o leve para o seu mundo e para o contexto de sua aplicação. Aquele cenário acontece? Se sim, então talvez o padrão de projeto ensinado naquele capítulo seja uma boa saída para controlar a crescente complexidade daquele código.

Durante todos os outros capítulos, faremos muitos exercícios. Todas as explicações serão baseadas em problemas do mundo real e você, ao final, utilizará os conhecimentos adquiridos para resolver os exercícios, que são desafiadores! Não se esqueça de prestar muita atenção ao cenário e ao problema proposto. Entenda a motivação de cada padrão para que você consiga levá-lo para os problemas do seu mundo!

Muitas regras e código complexo

Tomando como exemplo uma aplicação cujo objetivo é a criação de orçamentos, temos uma regra de negócio na qual os valores dos orçamentos podem ser submetidos à alguns impostos, como ISS, ICMS e assim por diante. Com isso, temos a simples classe que representa o orçamento, recebendo via construtor o seu valor: `public class Orcamento { private double valor; public Orcamento(double valor) { this.valor = valor; } public double getValor() { return valor; } }` Com isso, podemos criar novos orçamentos, instanciando objetos do respectivo tipo e caso queiramos calcular um imposto sobre seu valor, basta utilizarmos o atributo `valor` para isso. Assim, podemos estipular que o ICMS valha 10% e precisamos calculá-lo, baseado no valor do orçamento. Para isso, podemos ter a seguinte classe com um simples

método para realizar o cálculo: `public class CalculadorDeImpostos { public void realizaCalculo(Orcamento orcamento) { double icms = orcamento.getValor() * 0.1; System.out.println(icms); // imprimirá 50.0 } }` Podemos ainda querer calcular outro imposto, como o ISS, que é 6% do valor do orçamento. Com isso, adicionamos a nova regra ao código anterior. Mas devemos escolher qual o imposto que será calculado. Portanto, o método realizaCalculo deverá receber uma informação, indicando qual o imposto terá o cálculo realizado: `public class CalculadorDeImpostos { public void realizaCalculo(Orcamento orcamento, String imposto) { if("ICMS".equals(imposto)) { double icms = orcamento.getValor() * 0.1; System.out.println(icms); // imprimirá 50.0 } else if("ISS".equals(imposto)) { double iss = orcamento.getValor() * 0.06; System.out.println(iss); // imprimirá 30.0 } } }` Note que uma das consequências do código que acabamos de criar, é que espalhamos os cálculos e nossas regras de negócio. Dessa maneira, não temos nenhum encapsulamento de nossas regras de negócio e elas se tornam bastante suscetíveis a serem replicadas em outros pontos do código da aplicação. Por que não encapsulamos as regras dos cálculos em uma classe especializada para cada imposto?

Encapsulando o comportamento

Ao invés de mantermos as regras espalhadas pela nossa aplicação, podemos encapsulá-las em classes cujas responsabilidades sejam realizar os cálculos. Para isso, podemos criar as classes ICMS e ISS cada um com seu respectivo método para calcular o valor do imposto de acordo com o orçamento.

```
public class ICMS {

    public double calculaICMS(Orcamento orcamento) {
        return orcamento.getValor() * 0.1;
    }

}


public class ISS {

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

}
```

Agora temos as duas classes que separam a responsabilidade dos cálculos de impostos, com isso, podemos utilizá-las na classe `CalculadorDeImpostos` da seguinte maneira:

```
public class CalculadorDeImpostos {

    public void realizaCalculo(Orcamento orcamento, String imposto) {

    if( "ICMS".equals(imposto) ) {

      double icms = new ICMS().calculaICMS(orcamento);
      System.out.println(icms); // imprimirá 50.0

    } else if( "ISS".equals(imposto) ) {

      double iss = new ISS().calculaISS(orcamento);
      System.out.println(iss); // imprimirá 30.0
```

```
        }
    }
}
```

Agora o código está melhor, mas não significa que esteja bom. Um ponto extremamente crítico desse código é o fato de que quando quisermos adicionar mais um tipo diferente de cálculo de imposto em nosso calculador, teremos que alterar essa classe adicionando mais um bloco de `if`, além de criarmos a classe que encapsulará o cálculo do novo imposto. Parece bastante trabalho.

Eliminando os condicionais com polimorfismo e o pattern Strategy

O que queremos em nosso código é não realizar nenhum condicional, ou seja, não termos mais que fazer `if`s dentro do `CalculadorDeImpostos`. Dessa forma, não devemos mais receber a `String` com o nome do imposto, no qual realizamos os `if`s. Mas como escolheremos qual o imposto que deve ser calculado?

Uma primeira possibilidade é criar dois métodos separados na classe `CalculadorDeImpostos`. Um para o ICMS e outro para o ISS, dessa forma teremos:

```
public class CalculadorDeImpostos {

    public void realizaCalculoICMS(Orcamento orcamento) {

      double icms = new ICMS().calculaICMS(orcamento);
      System.out.println(icms);

    }

    public void realizaCalculoISS(Orcamento orcamento) {

      double iss = new ISS().calculaISS(orcamento);
      System.out.println(iss);

    }

}
```

No entanto, agora só transferimos o problema dos vários `if`s para vários métodos. O que não resolve o problema. O próximo passo para conseguirmos melhorar essa solução é termos um único método, genérico, que consegue realizar o cálculo para qualquer imposto, sem fazer nenhum `if` dentro dele.

```
public class CalculadorDeImpostos {

    public void realizaCalculo(Orcamento orcamento) {

      double icms = new ICMS().calculaICMS(orcamento);
      // Mas e se quisermos outro imposto?

      System.out.println(icms);

    }
```

```
}
```

Agora estamos presos ao ICMS. Precisamos que nosso código fique flexível o bastante para utilizarmos diferentes impostos na realização do cálculo. Uma possibilidade para resolvermos esse problema é, ao invés de instanciarmos o imposto que desejamos dentro do método, recebermos uma instância do Imposto que queremos utilizar, como no código seguinte:

```
public class CalculadorDeImpostos {

    public void realizaCalculo(Orcamento orcamento, Imposto imposto) {

      double valor = imposto.calcula(orcamento);

      System.out.println(valor);

    }

}
```

No entanto, não temos o tipo `Imposto` em nossa aplicação e além disso, nesse tipo precisamos passar uma instância de `ISS` e `ICMS`. Para isso, podemos criar uma interface chamada `Imposto` e fazermos as classes `ISS` e `ICMS` a implementar.

```
public interface Imposto {
    double calcula(Orcamento orcamento);
}

public class ICMS implements Imposto {

    public double calcula(Orcamento orcamento) {
        return orcamento.getValor() * 0.1;
    }

}

public class ISS implements Imposto {

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

}
```

E agora o nosso `CalculadorDeImpostos` está pronto para ser utilizado e flexível o bastante para receber diferentes tipos (ou "estratégias") de impostos. Um código que demonstra essa flexibilidade é o seguinte:

```
public class TesteDeImpostos {

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

```
        Orcamento orcamento = new Orcamento(500.0);

    CalculadorDeImpostos calculador = new CalculadorDeImpostos();

    // Calculando o ISS
    calculador.realizaCalculo(orcamento, iss);

    // Calculando o ICMS
    calculador.realizaCalculo(orcamento, icms);
    }
}
```

Agora, com um único método em nosso `CalculadorDeImpostos`, podemos realizar o cálculo de diferentes tipos de impostos, apenas recebendo a estratégia do tipo do imposto que desejamos utilizar no cálculo.

Quando utilizamos uma hierarquia, como fizemos com a interface `Imposto` e as implementações `ICMS` e `ISS`, e recebemos o tipo mais genérico como parâmetro, para ganharmos o polimorfismo na regra que será executada, simplificando o código e sua evolução, estamos usando o Design Pattern chamado Strategy.

Repare que a criação de uma nova estratégia de cálculo de imposto não implica em mudanças no código escrito acima! Basta criarmos uma nova classe que implementa a interface `Imposto`, que nosso `CalculadorDeImpostos` conseguirá calculá-lo sem precisar de nenhuma alteração!