05

Coesão e o Single Responsibility Principle

Olá! Bem-vindo ao curso de Orientação a objetos avançado e SOLID do Alura.

Nossa primeira aula será sobre **coesão**.

Neste curso, vamos discutir as ideias de Orientação a objetos - como coesão, acoplamento, encapsulamento, etc -, com um ponto de vista mais avançado e mais aprofundado sobre esses assuntos. Vamos discutir como conseguir fazer classes que são bastante coesas e que são pouco acopladas, aprender como evitar classes e métodos que não estão bem encapsulados e etc. Portanto, para esse curso, eu espero que você conheça um pouco de orientação a objetos e já tenha certa prática com alguma linguagem que dá suporte. Nesse curso eu vou usar C#, mas tudo que eu discutir vai funcionar para todas as linguagens que são OO, ok?

Todo mundo já ouviu falar de coesão, certo? Como na máxima: “Olha, todas as suas classes, elas tem que ser sempre muito coesas.". Mas a pergunta é: como fazer isso? Como criar classes que são coesas o tempo inteiro? Como evitar criar classes que tenham baixa coesão?

Para discutir isso, eu vou mostrar um código que parece código do mundo real. Eu tenho aqui a classe `CalculadoraDeSalario`. Essa classe, como dá para ver pelo método `calcula`, tem por objetivo calcular o salário de um funcionário. Para cada funcionário, existe uma regra diferente de cálculo de salário.

Exemplo:

```
public class CalculadoraDeSalario
    {
        public double calcula(Funcionario funcionario)
        {
            if (funcionario.Cargo is Desenvolvedor)
            {
                return dezOuVintePorcento(funcionario);
            }

            if (funcionario.Cargo is Dba || funcionario.Cargo is Tester)
            {
                return quinzeOuVinteCincoPorcento(funcionario);
            }

            throw new Exception("funcionario invalido");
        }

        private double dezOuVintePorcento(Funcionario funcionario)
        {
            if (funcionario.SalarioBase > 3000.0)
            {
                return funcionario.SalarioBase * 0.8;
            }
            else
            {
                return funcionario.SalarioBase * 0.9;
            }
        }
```

```
        private double quinzeOuVinteCincoPorcento(Funcionario funcionario)
        {
            if (funcionario.SalarioBase > 2000.0)
            {
                return funcionario.SalarioBase * 0.75;
            }
            else
            {
                return funcionario.SalarioBase * 0.85;
            }
        }
    }
```

Se o funcionário for um desenvolvedor, ele cairá nesse método 10% ou 20%. O método verifica e, se o salário do funcionário é maior do que o valor estipulado, ele dá um desconto. Caso contrário (`else`), ele dá um outro tipo de desconto. A mesma coisa acontece para os cargos `DBA` e `TESTER` - observe que ambos compartilham a mesma regra, a regra de 15% ou 25% e, ali, a implementação é mais ou menos a mesma.

Essa é uma classe que parece bastante com código do mundo real, certo? Essas classes olham o estado de um objeto e, a partir daí, decidem qual algoritmo executar, fazendo isso por meio de `if`s, `for`s, `while`s e etc.

E qual é o problema desse código? Será que ele é um código coeso? Será que ele está muito acoplado?

Para responder essas perguntas, voltemos a ele.

Esse código não é muito coeso, certo? Mas como que eu sei disso?

A minha primeira dica é sempre parar e pensar: “Poxa, será que essa classe vai parar de crescer?". Lembre-se de que se a classe é coesa, ela tem uma única responsabilidade, ela cuida de apenas uma única parte do meu sistema, ela representa uma única entidade. Portanto, se a classe é coesa, ela para de crescer um dia, por que as responsabilidades de uma entidade param de aparecer. Assim, no caso da classe `CalculadoraDeSalario` , por exemplo, a resposta para as perguntas formuladas anteriormente é não, por que sempre que eu criar um cargo novo no meu sistema, eu vou ter que alterar essa classe.

Agora imagine no mundo real, que é muito mais complexo do que esse código, onde eu tenho 30, 40, 50 cargos, ou seja, 30, 40, 50 regras diferentes. O que vai acontecer com essa classe? Essa classe vai ficar gigante, e por esse motivo, ela não vai ser coesa e resultará em um código complicado. E todo mundo sabe as consequências de um código complicado, certo? Ele é mais difícil de manter, muito mais suscetível a um bug - por que o código é enorme -, alguma coisa pode passar despercebida, o reuso é muito menor - se a classe faz muita coisa, é difícil levá-la para um outro sistema por que o outro sistema raramente vai precisar de tudo o que ela faz -, e assim por diante.

Vejam só, que complexidade!

Um código complicado é bastante discutível no mundo OO por que, às vezes, eu posso ter um código complicado, mas dentro de uma classe coesa. E esse tipo de código incomoda menos, por que se eu tenho um método público cujo código está feio, é fácil: eu jogo a implementação dele fora e escrevo de novo. Existem várias técnicas de refatoração pra isso - escrever variáveis com bons nomes, criar métodos privados, extrair para variáveis locais que explicam melhor o que o algoritmo está fazendo e etc. Eu nem vou discutir muito sobre refatoração, porque entraríamos em uma outra discussão e existe um curso disponível sobre o assunto (o curso de refatoração).

O que me incomoda nessa classe é que, além de complicada, ela não é coesa. Ela vai crescer pra sempre e, dessa forma, vai dificultar muito a vida do desenvolvedor. Veja o sistema que estamos analisando. Eu tenho uma `CalculadoraDeSalario`, que possui várias regras de cálculo. Toda vez que surgir uma regra nova, eu vou ter que colocá-la dentro da classe. Além disso, esse tipo de código dificulta também a testabilidade. Se eu estou pensando em teste automatizado, por exemplo, o teste dessa classe vai ser muito complicado. Eu vou ter que escrever uma bateria imensa pra ela. Difícil, certo?

Então qual é a ideia? O que eu estou tentando fazer é pegar cada uma das regras que eu tinha de cálculo - `DezOuVintePorCento`, `QuinzeOuVintePorCento` e `TrintaOuQuarentaPorCento` -, e colocá-la em um único lugar.

Para saber o que aconteceria na prática, voltemos mais uma vez para o meu código. Nele, eu tenho os dois métodos privados. Agora imagine que cada um desses métodos privados seja levado para uma classe em particular, ou seja, imagine que extrairemos um método privado e o colocaremos em uma classe. Dessa forma, cada uma dessas regras, cada uma dessas classes, será coesa - por que eu vou ter uma classe cuja única responsabilidade é entender da regra de `DezOuVintePorCento` ou de `QuinzeOuVinteCincoPorCento`, ou seja, cada classe será responsável por uma única regra. Se a regra mudar, tudo bem, é só ir lá e mexer na classe. Porém, se uma regra nova aparecer, eu não vou precisar abrir a classe `DezOuQuinzePorCento` ou `QuinzeOuVinteCincoPorCento`; eu posso simplesmente criar uma nova classe. Assim, eu estou criando classes coesas.

E como eu optei por tratar a coesão nesse código? Eu, com a minha experiência, percebi que o que muda de um código para o outro é basicamente a regra que é aplicada em cada um dos casos. Sabendo disso, eu peguei a regra e a coloquei em classes menores, tornando-as mais coesas.

Eu vou mostrar isso em um código, fazendo essa refatoração pra vocês verem como isso vai ficar mais legal. A primeira coisa que eu vou fazer é pegar cada um desses métodos privados e extrair pra uma classe. Para isso, eu vou criar uma classe que chamaremos de `DezOuVintePorCento` usando esse método privado:

```
private double dezOuVintePorcento(Funcionario funcionario)
        {
            if (funcionario.SalarioBase > 3000.0)
            {
                return funcionario.SalarioBase * 0.8;
            }
            else
            {
                return funcionario.SalarioBase * 0.9;
            }
        }
```

E vou jogar pra lá. Por enquanto, deixa assim, ele está meio estranho, mas tudo bem.

Vou fazer a mesma coisa com o `QuinzeOuVinteECincoPorCento`:

```
private double quinzeOuVinteCincoPorcento(Funcionario funcionario)
        {
            if (funcionario.SalarioBase > 2000.0)
            {
                return funcionario.SalarioBase * 0.75;
            }
            else
```

```
            {
                return funcionario.SalarioBase * 0.85;
            }
        }
```

Legal.

A primeira coisa que eu já percebi é que só existem duas coisas em comum entre essas duas classes que eu acabei de criar: ambas devolvem um `double` e recebem um `Funcionario`, e ambas são uma regra de cálculo. Lembre-se de que no mundo orientado à objetos, a gente cria interfaces, certo? É sempre legal programar pra interfaces. No próximo capítulo sobre acoplamento, discutiremos a importância de interfaces - elas são estáveis e etc; chegaremos lá.

Aqui, por enquanto, eu vou só criar uma interface, uma `IRegraDeCalculo`, método `public double calcula` que vai receber um `funcionario`:

```
public double Calcula(Funcionario funcionario);
```

Agora, eu vou nessas duas regras de negócio que nós temos e vou implementar a `IRegraDeCalculo`. Vou mudar o método para `Calcula`, e assim, o método fica `Public`:

```
public class DezOuVintePorCento : IRegraDeCalculo {

    public double Calcula(Funcionario funcionario);
            if(funcionario.SalarioBase() > 3000.0) {
                return funcionario.SalarioBase() * 0.8;
        }
        else {
            return funcionario.SalarioBase() * 0.9;
        }
 }
```

Vou fazer a mesma coisa no `QuinzeOuVinteECincoPorCento`:

```
public class QuinzeOuVinteECincoPorCento: IRegraDeCalculo  {
    public double Calcula(Funcionario funcionario) {
        if(funcionario.SalarioBase() > 2000.0) {
            return funcionario.SalarioBase() * 0.75;
        }
        else {
            return funcionario.SalarioBase() * 0.85;
        }
    }

}
```

Excelente.

Veja só! As duas classes que eu criei com as regras de negócio estão mais coesas! Uma só tem a regra de `QuinzeOuVinteECincoPorCento` e a outra só tem a de `DezOuVintePorCento`.

Note que essas classes não sofrem com o problema da classe antiga, a `CalculadoraDeSalario`, por que essa nova classe não vai crescer pra sempre. Sua única responsabilidade é cuidar da regra de `DezOuVintePorCento` e da regra de `QuinzeOuVinteECincoPorCento`, e assim por diante.

Agora vamos voltar para a `CalculadoraDeSalario` e fazer a refatoração mais simples - que é dar um `new` na regra `DezOuVintePorCento()`. `calcula` e passar o `funcionario`. Faremos a mesma coisa no código abaixo, `return new QuinzeOuVinteECincoPorCento().Calcula(funcionario);`:

```
public class CalculadoraDeSalario
    {
        public double Calcula(Funcionario funcionario)
        {
            if (funcionario.Cargo is Desenvolvedor)
            {
                return new DezOuVintePorcento().Calcula(funcionario);
            }

            if (funcionario.Cargo is Dba || funcionario.Cargo is Tester)
            {
                return new QuinzeOuVinteCincoPorcento().Calcula(funcionario);
            }

            throw new Exception("funcionario invalido");
        }
}
```

Ótimo! Já está melhor, certo?

Todas as classes do novo código são mais ou menos coesas - mais ou menos por que as duas primeiras estão bem coesas, mas a `CalculadoraDeSalario` não, por que ela não para de crescer; sempre que um cargo novo aparecer, será necessário adicionar um `if`.

Resolveremos o problema da `CalculadoraDeSalario` da seguinte forma: todo cargo tem uma regra de negócio associada, certo? Então é isso que iremos fazer aqui. Quando eu criar um cargo, eu vou passar pra ele o tipo de regra de negócio que ele vai usar - `DESENVOLVEDOR(new DezOuVintePorCento())`, ou `DBA(new QuinzeOuVinteECincoPorCento())`, e farei a mesma coisa para o `TESTER(new QuinzeOuVinteECincoPorCento());`, portanto:

```
public abstract class Cargo
    {
        public Cargo(IRegraDeCalcula regra)
        {

        }
    }
```

Ou seja, eu vou criar o construtor do `Cargo` que vai receber uma `RegraDeCalculo` - chamada de `regra` e que será guardada em um atributo.

Mas por que eu fiz isso?

Veja bem, se eu criar um cargo novo, como por exemplo `SECRETARIO`, o código não compila. Ele vai, obrigatoriamente, me pedir uma regra de cálculo.

Então vamos lá! Nosso código está funcionando e eu vou voltar para a `CalculadoraDeSalario`. Aqui, observe como ficou mais fácil! Podemos simplesmente jogar toda essa classe fora e fazer `return funcionario.Cargo.Regra().Calcula(funcionario);`:

```
public double Calcula(Funcionario funcionario)  {
    return funcionario.Cargo.Regra.Calcula(funcionario);
```

Na verdade, eu poderia esconder isso aqui dentro do próprio `funcionario`, de alguma forma como:

```
public double  Calcula(Funcionario funcionario)  {
     return funcionario.CalculaSalario();
}
```

E esse método, lá dentro, vai fazer:

```
public double CalculaSalario()  {
    return  Cargo.Regra.Calcula(this);
}
```

Calculando para ele mesmo (`this`), ou seja, tudo isso continua funcionando.

Nessa minha implementação, talvez a classe `CalculadoraDeSalario` passe até a ser inútil, porque agora ela só tem uma única linha de código: `CalculaSalario()`. Para fazer isso, o meu grande segredo foi usar as classes `DezOuVintePorCento` e `QuinzeOuVinteECincoPorCento`. Além disso, observe a classe `Funcionario`: ela é bem simples e possui diversos atributos - um deles é o `cargo`, e aqui o `CalculaSalario`. Essa classe é coesa, até porque não tem como ela não ser, certo? Ela só tem regras de `Funcionario`.

Assim, o problema de coesão abordado durante toda a explicação estava na classe `CalculadoraDeSalario`, que fazia muita coisa. A solução encontrada foi espalhar as regras em classes diferentes e tornar o `Cargo` mais inteligente. Se observarmos o novo código, perceberemos que ele ficou muito melhor.

Mas agora daremos uma olhada no código antigo de novo, porque na nossa refatoração, nós resolvemos dois problemas que essa classe tinha. O primeiro era de coesão e já foi discutido, mas o segundo, ainda não. Nós não discutimos o segundo problema por que eu não queria entrar em detalhes agora, mas olha só, um outro grande problema de códigos orientados a objetos é a quantidade de pontos que eu tenho que mudar dada a alteração do meu usuário. No código antigo, ou seja, da maneira com ele estava programado no começo, se eu criasse um cargo novo - por exemplo, o cargo de `SECRETARIO` -, o que eu ia ter que fazer? Eu ia ter que criar uma classe nova, certo? Eu ia ter que adicionar a regra do novo cargo à classe `CalculadoraDeSalario`.

Mas e se eu esquecesse?

O ponto pode nem sempre estar ligado à esquecer. O ponto é que às vezes eu sou um desenvolvedor e ainda não estou familiarizado com o projeto, e como essa mudança é indireta, ela não é clara no meu código - pois toda vez que um cargo novo aparece eu tenho que mexer na `CalculadoraDeSalario`.

Esse é um grande problema de código OO: a **propagação de mudança**.

Idealmente, eu tenho que mexer num único ponto. Dessa forma, se o cliente pediu uma mudança, eu vou em um único lugar, altero o que for necessário e a mudança se propagará para o resto. Eu não vou precisar programar usando `Ctrl + F` - esse é um ótimo indício de que o código não está bem orientado a objetos.

Esses pontos de mudança devem estar explícitos no seu design!

No nosso código novo, por exemplo, nós resolvemos esse problema. Para esse tipo de código, dá-se o nome de **encapsulamento**, por que o `Cargo` deixou vazar para o mundo de fora aonde a regra de cálculo está, ou seja, o `Cargo` sabe a regra que ele deve escolher, e portanto, esse código tem que estar dentro dele.

Antes, o nosso código, além de não ser coeso, estava encapsulado.

Mais pra frente, tem um capítulo em que eu só vou falar de encapsulamento, mas eu já queria explicar pra vocês desde o começo o que é encapsulamento - e olha só como ele é perigoso!

Voltando ao nosso código, percebemos que o encapsulamento nele estava problemático, pois a regra de cálculo estava saindo da classe `Cargo` e seria necessário mexer em outro lugar para colocar a regra que é relativa ao cargo. E agora que esse problema foi resolvido, sempre que eu criar um cargo novo, o compilador vai pedir pra eu passar a regra de cálculo - não tem como eu criar um cargo sem passar a regra de cálculo. Com tudo ligado no meu sistema, eu não vou ter o problema de esquecer de passar uma regra pra um cargo.

Observe que esse problema no mundo real pode ser pior. Por que aqui, eu posso ter 10 classes diferentes que possuem uma regra de negócio relacionada ao cargo. Tanto que eu fiz a brincadeira com o salário, mas eu poderia ter 3, 4, 5 outras regras espalhadas em outras 4 ou 5 classes diferentes.

Em suma, sempre buscaremos ter classes coesas por que uma classe coesa é mais fácil de ser lida, possui mais reuso, é mais fácil pegá-la e levá-la para um outro sistema, ela provavelmente vai ser mais simples - porque ela vai ter menos código-, e eu não vou precisar abri-la o tempo inteiro. Além disso, elas sempre virão fechadas no Visual Studio - se só abrirmos no caso particular quando for preciso mudar alguma regra ou quando houver algum problema naquela regra.

Portanto, a grande vantagem de uma classe coesa é que ela é pequenininha, bem focada e é possível saber quando será necessário mexer nela.

No meu slide, eu coloquei a sigla **SRP**. No começo do curso, eu falei que esse era um curso de Orientação a objetos avançado e SOLID. Mas o que é SOLID?

SOLID é um acrônimo, um conjunto de 5 boas práticas em relação a Orientação a objetos em que cada letra representa uma prática em particular. O **S** nos remete ao **SRP**, o *Single Responsibility Principle* - em português: Princípio da Responsabilidade Única. E esse princípio está estritamente relacionado com a coesão.

Nesse curso, discutimos uma maneira de conseguir alcançar o *Single Responsibility Principle* e, mais do que isso, discutimos uma maneira de detectar classes que não são coesas. Comece a prestar atenção nisso no seu dia a dia! Quando você está escrevendo uma classe que não para de crescer, esse é um indício de que ela não é coesa. Quando você tiver uma classe com 40, 50, 60 métodos, pare e pense: “Será que a minha classe tem mesmo 60 comportamentos diferentes? Será que eu não consigo separar isso em classes menores, mais coesas?", e assim, use os conhecimentos adquiridos nessa aula para aproximar a sua classe do **SRP**.

Obrigado.