02

Coesão e o Single Responsibility Principle

Olá! Bem-vindo ao curso de Orientação a objetos avançado e SOLID do Alura. Neste curso eu vou discutir com vocês todas aquelas ideias de Orientação a objetos, como coesão, acoplamento, encapsulamento etc., só que dessa vez com um ponto de vista um pouquinho mais avançado, um pouquinho mais profundo nesses assuntos.

Vou discutir com vocês como conseguir fazer classes que são bastante coesas, que são pouco acopladas, como evitar fazer classes e métodos que não estão bem encapsulados etc. e tal. Pra esse curso, eu espero que você conheça um pouquinho de Orientação a objetos, já tenha alguma prática com alguma linguagem que dá suporte. Aqui no curso, vou usar Java, mas tudo o que eu discutir vai funcionar para todas as linguagens que são OO. Tá legal?

A primeira aula vai ser sobre coesão. Todo mundo já ouviu falar de coesão, certo? E tem aquela máxima lá, "Olha, todas as suas classes, elas têm que ser sempre muito coesas." Agora a pergunta é: como fazer isso? Como criar classes que são coesas o tempo inteiro? Como evitar criar classes que tenham baixa coesão?

E para discutir isso, eu vou mostrar um código que parece código do mundo real. Dá uma olhada. Eu tenho aqui a minha classe `CalculadoraDeSalario`. Essa classe, dá para ver pelo método `calcula`, tem por objetivo calcular o salário de um funcionário. Dado um funcionário, e um cargo que esse funcionário tem, eu tenho uma regra diferente do cálculo do salário dele. Dá uma olhada:

```
public class CalculadoraDeSalario {


    public double calcula(Funcionario funcionario) {
        if(DESENVOLVEDOR.equals(funcionario.getCargo())) {
            return dezOuVintePorcento(funcionario);
        }

        if(DBA.equals(funcionario.getCargo()) || TESTER.equals(funcionario.getCargo())) {
            return quinzeOuVinteCincoPorcento(funcionario);
        }

        throw new RuntimeException("funcionario invalido");
    }

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

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

```
            }
        }

    }
```

Se o cara for desenvolvedor, ele cai nesse método aí 10% ou 20%. Esse método lá dentro verifica se o salário dele é maior do que tanto, e aí ele dá um desconto. Caso contrário (`else`), ele dá um outro desconto. A mesma coisa acontece para `DBA` e `TESTER` . Veja que ambos compartilham a mesma regra, essa 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? Onde eu tenho classes que olham o estado de um objeto e, a partir daí, decidem qual algoritmo executar, e fazem isso por meio de `if` s e `for` s e `while` s etc. e tal.

Agora a pergunta é: qual é o problema desse código? Será que ele é um código coeso? Será que ele está muito acoplado? Vamos lá.

Voltando aqui para ele. Esse código, ele não é lá muito coeso, está certo? Como que eu sei disso? A minha primeira dica é o seguinte, eu sempre paro e penso: "Puxa, será que essa classe vai parar de crescer? Porque veja só: se a classe é coesa, ou seja, ela tem uma única responsabilidade, ela cuida de apenas uma única parte do meu sistema, ela representa uma única entidade. Se essa classe é coesa, ela tem que parar de crescer um dia. Porque as responsabilidades de uma entidade um dia param de aparecer.

Nesse caso, em particular, não. Veja só. Sempre que eu criar um cargo novo no meu sistema, eu vou ter que alterar essa classe. Sempre, sempre, sempre. Agora imagina no mundo real, muito mais complexo do que esse código, onde eu tenho, sei lá, 30, 40, 50 cargos. E imagina que eu tenha 30 regras diferentes. O que vai acontecer com essa classe? Essa classe vai ficar gigante, ela vai ter 30 regras diferentes. Tá certo? Então, ela não vai ser nada coesa.

Qual é a consequência disso? Um código complicado. Todo mundo sabe a consequência de um código complicado, certo? É mais difícil de manter, é muito mais suscetível a um bug, porque o código é enorme, você vai deixar passar alguma coisa ali, o reuso é muito menor, porque se a classe faz muita coisa é difícil levá-la pra um outro sistema. Porque o outro sistema raramente vai precisar de tudo o que ela faz. E assim por diante.

Veja só, pessoal, que complexidade, um código complicado, ele é bastante discutível no mundo OO. Porque às vezes eu posso ter um código complicado, mas dentro de uma classe coesa. E esse tipo de código me incomoda menos. Porque se eu tenho um método público, cujo código está feio, é fácil: eu jogo a implementação dele fora e escrevo de novo. Tem um monte de técnica 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 etc. e tal. Eu nem vou entrar nesse mérito de refatoração, porque é uma outra discussão e você pode até fazer o nosso curso sobre o assunto (o curso de refatoração).

Mas aqui, o que me incomoda é que essa classe, além de complicada, ela não é coesa. Ela vai crescer pra sempre. Então, ela vai dificultar e muito a vida do desenvolvedor.

Veja o sistema hoje. Eu tenho uma `CalculadoraDeSalario` , que tem as várias regras de cálculo. E toda vez que surgir uma regra nova, eu vou ter que enfiá-la aí dentro. Esse tipo de código, galera, ainda dificulta a testabilidade. Se eu estou pensando em teste automatizado, o teste dessa classe vai ser muito complicado. Eu vou ter que escrever uma bateria imensa pra ela. Difícil, certo? Qual é a ideia?

O que eu estou tentando fazer agora é pegar cada uma das regras que eu tinha de cálculo ? `DezOuVintePorCento` , `QuinzeOuVintePorCento` , `TrintaOuQuarentaPorCento` , sei lá, as regras que forem aparecendo -, e colocar cada uma dessas regras num único lugar.

Veja só o que ia acontecer na prática. Vou voltar aqui pro meu código. Eu tenho os dois métodos privados, e imagina que cada um desses métodos privados eu vá levar para uma classe em particular. Eu vou extrair um método privado e colocar numa classe. O que eu vou ganhar? Veja só que cada uma dessas regras, cada uma dessas classes, será coesa. Porque eu vou ter uma classe cuja única responsabilidade é entender da regra de `DezOuVintePorCento`, a outra, de `QuinzeOuVinteCincoPorCento`.

Ou seja, cada classe vai entender de uma única regra. Se a regra mudar, tudo bem, eu vou lá e mexo na classe. Mas veja só que ela vai mudar por um único motivo. O motivo é se essa regra em particular mudar. Se uma regra nova aparecer, eu não vou nem precisar abrir a classe `DezOuQuinzePorCento` ou `QuinzeOuVinteCincoPorCento`, não vou. Vou simplesmente criar uma nova classe.

Aqui 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 pro outro é basicamente a regra que é aplicada em cada um dos casos. Tá certo?

Eu peguei a regra e coloquei em classes menores. Essas classes serão mais coesas. Excelente. Eu vou mostrar isso em código agora pra vocês, fazendo essa refatoração, pra vocês verem como isso vai ficar muito mais legal.

Agora no Eclipse, vamos lá. A primeira coisa que eu vou fazer é pegar cada um desses métodos privados e extrair pra uma classe. Então veja, vou criar a classe que eu vou chamar de `DezOuVintePorCento`.

E vou pegar esse método privado aqui:

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

E vou jogar pra lá. Por enquanto deixa assim, está meio estranho mas tudo bem. Vou fazer a mesma coisa com o `QuinzeOuVinteECincoPorCento`. Nova classe:

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

}
```

Legal. A primeira coisa que eu já percebi. É que veja só que tem duas coisas em comum entre essas duas classes que eu acabei de criar, certo? Ambas devolvem um `double` e recebem um `Funcionario`, e ambas são uma regra de cálculo. No mundo orientado a objetos, a gente vai criar interfaces, certo? É sempre legal programar pra interfaces.

No próximo capítulo sobre acoplamento, eu vou discutir a importância de interfaces. Elas são estáveis, etc. e tal. Mas eu chego lá.

Aqui por enquanto eu vou só criar a interface, então uma `RegraDeCalculo`, método `public double calcula` que vai receber um funcionário aqui, que eu vou colocar `funcionario`:

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

Agora eu vou nessas duas regras de negócio que eu tenho, e vou implementar, `implements RegraDeCalculo`. Vou mudar o método aqui para `calcula`, o método fica `public`, certo? Legal.

```
public class DezOuVintePorCento implements RegraDeCalculo {

    public double calcula(Funcionario funcionário) {
            if(funcionario.getSalarioBase() > 3000.0) {
                return funcionario.getSalarioBase() * 0.8;
        }
        else {
            return funcionario.getSalarioBase() * 0.9;
        }
 }
}
```

A mesma coisa no `QuinzeOuVinteECincoPorCento`:

```
public class QuinzeOuVinteECincoPorCento implements RegraDeCalculo  {
    public double calcula(Funcionario funcionario) {
        if(funcionario.getSalarioBase() > 2000.0) {
            return funcionario.getSalarioBase() * 0.75;
        }
        else {
            return funcionario.getSalarioBase() * 0.85;
        }
    }

}
```

Excelente. Veja só. As duas classes que eu criei com as regras de negócio são muito mais coesas. Essa aqui só tem a regra de `QuinzeOuVinteECincoPorCento`, e essa aqui só tem a de `DezOuVintePorCento`. Ótimo!

Veja só que essa classe – as duas, na verdade – não sofrem o problema da classe antiga da `CalculadoraDeSalario`, porque essa classe não vai crescer pra sempre. A única responsabilidade dela é cuidar dessa regra de `DezOuVintePorCento`, e dessa outra, de `QuinzeOuVinteECincoPorCento`, e assim por diante.

Agora vou voltar aqui para a `CalculadoraDeSalario` e vamos fazer a refatoração mais simples, que é dar um `new` aqui na regra. `DezOuVintePorCento(). calcula` e passo o `funcionario`. A mesma coisa aqui no debaixo, `return new QuinzeOuVinteECincoPorCento().calcula(funcionario);`:

```
public class CalculadoraDeSalario {
```

```
    public double calcula(Funcionario funcionario) {
        if(DESENVOLVEDOR.equals(funcionario.getCargo())) {
            return new DezOuVintePorCento(). calcula(funcionario);
        }

        if(DBA.equals(funcionario.getCargo()) || TESTER.equals(funcionario.getCargo())) {
            return new QuinzeOuVinteECincoPorCento().calcula(funcionario);
        }

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

Ótimo, já está melhor, certo? Esse nosso código já está bem melhor. Todas as classes são mais ou menos coesas, mais ou menos por que essas duas estão bem coesas, mas a `CalculadoraDeSalario` está meio estranha ainda. Essa aqui ainda não para de crescer, certo, sempre que um cargo novo aparecer, eu vou ter que lembrar de colocar um `if` a mais aqui.

Vamos resolver isso aqui. Como que eu vou fazer? Fácil. Todo cargo tem uma regra de negócio associada, certo? Então é isso que eu vou 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()),` , o `DBA(new QuinzeOuVinteECincoPorCento()),` , e a mesma coisa para o meu `TESTER(new QuinzeOuVinteECincoPorCento());` :

```
public enum Cargo {
    DESENVOLVEDOR(new DezOuVintePorCento()),
    DBA(new DezOuVintePorCento()),
    TESTER(new QuinzeOuVinteECincoPorCento());
}
```

Está certo? As pessoas esquecem – esse é um detalhe do Java – que o enum é quase uma classe, eu posso ter código nele.

O que eu vou fazer aqui, vou criar o construtor do `Cargo` , que vai receber uma `RegraDeCalculo` , que vou chamar de `regra` , vou guardar essa `regra` num atributo do enum, e vou criar aqui o `getRegra` :

```
private RegraDeCalculo regra;

Cargo(RegraDeCalculo regra)  {
    this.regra = regra;
}
public RegraDeCalculo getRegra()  {
    return regra;
}
```

Por que que eu fiz isso? Porque, veja só, com o enum desse jeito, se eu criar um cargo novo, igual, por exemplo `SECRETARIO` , se eu fizer isso aqui, o código não compila. Ele vai, obrigatoriamente, me pedir o quê? Uma regra de cálculo. Está certo?

Então, vamos lá. Nosso código está funcionando e eu vou voltar aqui para a `CalculadoraDeSalario` . Aqui, veja só como ficou mais fácil, dá pra jogar tudo isso aqui fora e fazer simplesmente `return funcionario.getCargo().getRegra().calcula(funcionario);` :

```
public double calcula(Funcionario funcionario)  {
    return funcionario.getCargo().getRegra().calcula(funcionario);`
```

Eu poderia até, na verdade, esconder isso aqui dentro do próprio `funcionario`, alguma coisa como:

```
public double  calcula(Funcionario funcionario)  {
     return funcionario.calculaSalario();
}
```

E esse método lá dentro vai fazer simplesmente:

```
public double calculaSalario()  {
    return  cargo.getRegra().calcula(this);
}
```

Calcula para ele mesmo (`this`). Tudo isso aqui continua funcionando. Apaguei os imports ali em cima e o código ficou menor.

Nessa minha implementação, talvez essa classe `CalculadoraDeSalario` passe até a ser inútil, porque agora ela só tem uma única linha de código, `calculaSalario()`. O meu grande segredo aqui foi as classes `DezOuVintePorCento`, `QuinzeOuVinteECincoPorCento`.

Dá uma olhada também na classe `Funcionario`. A minha classe `Funcionario` é bem simples, ela tem um monte de atributos, um deles é o `cargo`, e aqui o `calculaSalario`. Essa classe é coesa, até porque não tem muito por que ela não ser, certo? Ela só tem regras de `Funcionario`. Não tem problema.

O problema de coesão que eu dei pra vocês nesse exercício foi justamente naquela `CalculadoraDeSalario` que fazia muita coisa, e a solução foi espalhar as regras em classes diferentes, e fazer o `Cargo` ser mais inteligente. Certo?

Já vou explicar também a vantagem dessa refatoração aqui do `Cargo`, que não envolve só coesão. Já chego lá.

Ótimo, nosso código ficou muito melhor. Mas agora eu quero dar uma olhada no código antigo de novo, porque na nossa refatoração, nós resolvemos dois problemas, na verdade, que essa classe tinha. Um deles era de coesão, que eu discuti com vocês, mas o segundo eu passei batido, porque eu não queria entrar em detalhe agora. Mas olha só, um outro grande problema de códigos orientados a objeto é a quantidade de pontos que eu tenho que mudar, dada uma alteração do meu usuário.

Veja só. Nesse código antigo, da maneira com que estava programado antes, se eu criasse um cargo novo, por exemplo, o cargo de `SECRETARIO`, o que eu ia ter que fazer? Eu ia ter que mexer no meu enum, certo? Ia ter que adicionar uma linha a mais ali no meu enum, só que, mais do que isso, eu ia ter que abrir essa classe `CalculadoraDeSalario` e colocar a regra dele aí.

Agora, a pergunta é: e se eu esquecesse? O ponto é às vezes nem esquecer, o ponto é que às vezes eu sou um desenvolvedor que caí de paraquedas no projeto e não conheço tudo. E como essa mudança é indireta, não é clara no meu código – toda vez que um cargo novo aparece eu tenho que mexer na `CalculadoraDeSalario` eu não vou lembra. Esse é um grande problema de código OO. É a **propagação de mudança**.

Eu tenho que mexer num único ponto, idealmente. O cliente pediu uma mudança, eu vou num único lugar, e nesse lugar eu mexo, e a mudança propaga pro resto. Eu não tenho que programar usando `Ctrl + F`. Esse é um ótimo indício

de que seu código não está bem orientado a objetos: é quando você tem que, o tempo inteiro, apelar para `Ctrl + F`, ou pra grep no Linux, alguma forma pra sair buscando código pra descobrir onde tem que mudar.

Idealmente, esses pontos de mudança, eles são sempre explícitos no seu design. No nosso código novo, nós resolvemos esse problema. No próprio enum agora eu tenho lá, eu passo pra ele a regra de cálculo que eu vou usar.

Esse tipo de código, galera, a gente fala que é um problema de **encapsulamento**. Porque o enum ali, `Cargo`, deixou vazar pro mundo de fora aonde vai ficar essa regra de cálculo, não estava escondida nele. O `Cargo` sabe a regra que tem que escolher. Então esse código tem que estar dentro dele.

Está certo? Antes, o nosso código, além de não ser coeso, ele não estava encapsulado. Mais pra frente, tem um capítulo em que eu só vou falar de encapsulamento. Mas já queria alertar pra vocês desde o começo o que é encapsulamento, e olha só como isso é perigoso. Tá bem?

De novo, encapsulamento aqui estava problemático, a regra de cálculo estava saindo da classe `Cargo`, do enum `Cargo`. Eu tinha que mexer em outro lugar para colocar a regra que é relativa ao cargo. E agora eu resolvi, certo, sempre que eu criar um enum novo, o compilador vai me encher o saco e vai pedir pra eu passar ali a regra de cálculo. Não tem como eu criar um cargo sem passar a regra de cálculo.

"Ah, mas estou criando um cargo que não tem regra", ótimo. Você vai criar a regra e vai colocar no enum. Tudo ligadinho no meu sistema, eu não vou ter o problema de esquecer de passar uma regra pra um cargo.

Veja que esse problema, pessoal, no mundo real pode ser pior. Porque eu posso ter 10 classes diferentes que tenham uma regra de negócio relacionada ao cargo. Aqui eu fiz a brincadeira com o salário, mas eu poderia ter 3, 4, 5 outras regras espalhadas em 4, 5 outras classes diferentes. Tá bem?

Voltando aqui para o meu slide, classes coesas, eu as quero o tempo inteiro. Por quê? Porque uma classe coesa é mais fácil de ser lida, eu tenho mais reuso, eu posso pegá-la e levar 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. Essa é uma coisa com a qual me preocupo bastante, uma classe coesa, ela geralmente vem fechada no meu Eclipse. Porque eu só a abro no caso particular quando eu preciso mudar aquela regra ou quando eu encontrei um problema naquela regra. Mas eu não fico mexendo nela o tempo inteiro.

Essa é a vantagem de uma classe coesa, ela é pequenininha, bem focada, eu sei quando eu tenho que mexer nela, e eu não tenho que mexer nela o tempo inteiro.

No meu slide, eu coloquei a sigla **SRP**. No começo do curso, eu falei que esse era um curso de Orientação a objetos avançada e SOLID. O que é SOLID? SOLID é o acrônimo, é o conjunto de 5 boas práticas em relação a Orientação a objetos, cada letra fala de uma prática em particular.

O **S** nos remete ao **SRP**, o *Single Responsibility Principle* ? em português, Princípio da Responsabilidade Única. A tradução disso, de maneira simples, é coesão. Tenha classes coesas. E aqui, eu discuti com vocês uma maneira de eu conseguir isso, e mais, eu discuti uma maneira de observar que as classes não são coesas.

Comece a prestar atenção nisso no seu dia a dia, quando você está escrevendo uma classe que não para de crescer nunca, esse é um indício de que ela não é coesa. Quando você tem uma classe com 40, 50, 60 métodos, pare e pense "Será que a minha classe, ela tem que ter mesmo 60 comportamentos diferentes? Será que eu não consigo separar isso em classes menores, mais coesas? Então é isso: nesta aula, a lição é coesão. Obrigado.