02

Classes abertas, Open Closed e Dependency Inversion Principles

Olá pessoal! Nas aulas passadas, discutimos sobre acoplamento e sobre coesão. Sobre acoplamento, em particular, eu falei bastante pra vocês sobre classes estáveis. Você lembra disso? A classe estável é aquela que tende a mudar muito pouco. Qual que é a vantagem disso? A vantagem é que se ela muda muito pouco é melhor que eu me acople a ela, afinal, ela não vai propagar a mudança pra mim.

Sempre que eu quiser pensar em acoplamento, ou precisar me acoplar com alguma outra classe ou módulo, a ideia é que eu me acople com módulos que são estáveis. Ótimo.

Com essa ideia na cabeça, dá uma olhada nesse código aí que eu vou dar pra vocês:

```
public class CalculadoraDePrecos {

    public double calcula(Compra produto) {
        TabelaDePrecoPadrao tabela = new TabelaDePrecoPadrao();
        Frete correios = new Frete();

        double desconto = tabela.descontoPara(produto.getValor());
        double frete = correios.para(produto.getCidade());

        return produto.getValor() * (1-desconto) + frete;
    }
}

public class TabelaDePrecoPadrao {
    public double descontoPara(double valor) {
        if(valor>5000) return 0.03;
        if(valor>1000) return 0.05;
        return 0;
    }
}

public class Frete {
    public double para(String cidade) {
        if("SAO PAULO".equals(cidade.toUpperCase())) {
            return 15;
        }
        return 30;
    }
}
```

Eu tenho uma calculadora de preço. A ideia dela é basicamente pegar um produto da minha loja e tentar descobrir o preço desse produto. Ele vai primeiro pegar o preço do produto bruto, aí vai usar essa tabela de preços padrão (`TabelaDePrecoPadrao`) pra calcular o preço, porque pode ter um eventual desconto, em seguida, ele vai descobrir também o valor do frete, porque eu tenho que mandar esse produto pelos correios, e no final, ele faz a conta ali. Ele pega o produto, multiplica pelo valor do desconto, mais o frete, uma regra convencional.

Percebe também que aqui eu tenho várias classes, porque eu estou pensando bastante em coesão, então, idealmente, eu tenho classes pequenas, bastante coesas, com pouca responsabilidade.

Eu tenho essa `TabelaDePrecoPadrao`, que tem uma regra de negócio, está numa classe. Eu tenho a classe `Frete` que toma conta ali de calcular o frete também, numa outra classe. E, na calculadora, essa classe depende das outras duas.

Tá legal, está tudo funcionando, e está perfeito. Só que agora pensa no seguinte: imagina que o meu software vá crescer. Então, eu não tenho só a tabela de preços padrão. Eu tenho a tabela de preços padrão e a tabela de preços diferenciados. Pra entrega, eu não uso só os correios. Eu uso os correios, ou estou usando uma outra empresa particular também de entrega de produtos. Imagina que a regra cresceu. Está certo? Eu tenho, de acordo com meu produto, de acordo com o cliente eu calculo o preço de maneira diferente. Como que eu vou implementar isso?

Eu tenho duas maneiras. Vamos lá. A primeira delas seria colocar um `if` na `CalculadoraDePrecos`. Dá uma olhada: `if(REGRA 1)`, uma regra qualquer, eu não especifiquei uma regra, mas imagina que eu tenha uma condição qualquer. Se a regra não acontecer, eu vou fazer `TabelaDePrecoPadrao` e vou usá-la. Caso contrário, se for a `REGRA 2`, ele vai usar a `TabelaDePrecoDiferenciada`.

```
public class CalculadoraDePrecos {

    public double calcula(Compra produto) {

        Frete correios = new Frete();

        double desconto;
        if (REGRA 1){
            TabelaDePrecoPadrao tabela = new TabelaDePrecoPadrao();
            desconto = tabela.descontoPara(produto.getValor());

        }
        if (REGRA 2){
            TabelaDePrecoDiferenciada tabela = new TabelaDePrecoDiferenciada();
            desconto = tabela.descontoPara(produto.getValor());
        }
        double frete = correios.para(produto.getCidade());
        return produto.getValor() * (1 - desconto) + frete;
    }
}
```

A mesma coisa lá pro frete. Aqui eu não coloquei nesse código, mas imagina a mesma coisa. Se cair na regra 1, usa os correios, se cair na regra 2, usa empresa XPTO também de entrega.

Não parece uma boa ideia, certo, porque eu vou começar a encher esse código de if, esse código vai ficar complicado; essa classe começa a perder coesão porque ela começa a saber de muita coisa; o acoplamento vai crescer, porque ela vai depender da `TabelaDePrecoPadrao`, da diferenciada, dos correios, da empresa XPTO e assim por diante. Está complicado.

Segunda alternativa: seria fazer separado. Eu pego a minha classe `Frete` e eu coloco esse if na classe de frete. Se eu estou na REGRA 1, faz desse jeito, se eu estou na REGRA 2, faz daquele jeito.

```
public class Frete {

    public double para(String cidade) {
         if(REGRA 1)  {
```

```
            if("SAO PAULO".equals(cidade.toUpperCase())) {
                        return 15;
            }
          return 30;
          }

        if(REGRA 2) { ... }
        if(REGRA 3) { ...}
        if(REGRA 4) { ...}
    }
}
```

A mesma coisa para a `TabelaDePrecoPadrao`. Se eu estou na REGRA 1, dá esse desconto, se eu estou na REGRA 2, dá aquele outro desconto, e assim por diante.

```
public class TabelaDePrecoPadrao {

    public double descontoPara(double valor) {
            if(REGRA 1) {
                if(valor>5000) return 0.03;
                if(valor>1000) return 0.05;
                return 0;
        }

        if(REGRA 2) { ... }
        if(REGRA 3) { ...}
        if(REGRA 4) { ...}
    }
}
```

O problema é que também a complexidade vai crescer, certo? Imagina que eu tenho 10 regras, eu vou ter 10 ifs aí, o código vai ficar difícil de manter. Veja só, nos dois códigos que eu dei pra vocês, no primeiro deles o acoplamento ia crescer, porque a classe `CalculadoraDePrecos` ia começar a depender de muitas outras classes. No meu segundo caso, aqui, a coesão dessas classes `Frete` e `TabelaDePrecoPadrao` também ia complicar.

Então, acoplamento, coesão... Eu estou falando pra vocês o tempo inteiro que a grande graça de programar orientado a objetos é balancear entre essas duas coisas. Eu nunca vou conseguir ter máxima coesão e zero acoplamento. A ideia é encontrar esse equilíbrio. Tá legal? Vamos lá.

O primeiro conceito que eu quero passar pra vocês é a ideia de que as classes têm que ser **abertas**. Mas como assim "aberta", o que que é uma classe aberta? Eu coloquei aí até a sigla **OCP** (*Open Closed Principle*), que é o princípio que fala disso. Mas que raio que é esse negócio de princípio do aberto e fechado, o que são classes abertas?

A ideia é que as suas classes sejam abertas para extensão. Ou seja, eu tenho que conseguir estendê-la, ou seja, mudar o comportamento dela, de maneira fácil. Mas ela tem que estar fechada para alteração. Ou seja, eu não tenho que ficar o tempo inteiro indo nela pra mexer um if a mais, para fazer uma modificação ou coisa do tipo. Então, de novo, fechada para modificação, ou seja, eu não quero ter que o tempo inteiro entrar nela e sair escrevendo código, mas ela tem que estar aberta para extensão, ou seja, eu tenho que conseguir mudar a execução dela ao longo do tempo.

Meio maluco isso, né? Como que eu faço isso? Eu vou mostrar pra vocês em código, e aí vai ficar muito mais claro. Vamos lá. Esse é o código que eu tenho agora. E eu sei que eu quero evitar qualquer tipo de if, sei lá, `if(regra1)` calcula

desse jeito, caso contrário, e assim por diante. Preciso evitar esse if tanto aqui quanto dentro das implementações, de tabela de preço, de frete etc. e tal.

Afinal, está tudo bonitinho, como eu mostrei pra vocês. Esse código é simples, super coeso, esse código aqui do `Frete` também é simples, super coeso, a `CalculadoraDePrecos` também.

Mas a gente precisa mudar o comportamento, e é isso que vai acontecer no mundo real. Então, a primeira coisa que eu vou fazer é pensar numa abstração. Já que eu tenho diferentes tabelas de preço, eu preciso pensar numa abstração comum entre todas elas. E, por enquanto, vai ser o próprio método que eu tenho aqui, esse `double descontoPara(double valor)` .

A primeira coisa que eu vou fazer é criar uma interface que vai representar essa abstração pra mim. Eu vou chamar de `TabelaDePreco` . O único método vai ser `double descontoPara` .

```
public interface TabelaDePreco  {
    double descontoPara(double valor);
}
```

Essa classe aqui, `TabelaDePrecoPadrao` implementa a interface `TabelaDePreco` :

```
public class TabelaDePrecoPadrao implements TabelaDePreco  {
```

Vou fazer a mesma coisa pro `Frete` . Vou chamar aqui (a nova interface) de `ServicoDeEntrega` . O método que ele vai ter lá é esse `double para` que recebe uma cidade:

```
public interface ServicoDeEntrega  {
    double para(String cidade);
}
```

E aqui o `Frete` vai implementar `ServicoDeEntrega` :

```
public class Frete implements ServicoDeEntrega  {
```

Ótimo, está tudo perfeito. E eu sei que essas interfaces, elas tenderão a ser estáveis. Agora, o que eu vou fazer é o seguinte:

```
TabelaDePrecoPadrao tabela = new TabelaDePrecoPadrao();
Frete correios = new Frete();
```

Esse `new` é o que me incomoda aqui. Os dois. O ponto é: eu preciso fazer com que seja possível eu trocar a implementação da `TabelaDePreco` . Como que eu vou fazer isso? Eu vou receber pelo construtor. Então,

`CalculadoraDePrecos` vai ser uma `TabelaDePreco` no construtor, a interface vou chamar de `tabela`, e vai receber um `ServicoDeEntrega`, vou chamar aqui de `entrega`.

```
public class CalculadoraDePrecos  {

    public CalculadoraDePrecos(TabelaDePreco tabela, ServicoDeEntrega entrega) {
    }
```

Vou guardar esses dois parâmetros que eu recebi no construtor como atributos da classe, afinal eu vou precisar usar nesses métodos aqui. Vou jogar os dois `new`s fora, e aqui não é mais `correios`, eu mudei o nome para `entrega`:

```
double frete = entrega.para(produto.getCidade());
```

Dá uma olhada nesse código, como está muito melhor. Eu vou até escrever aqui um método de teste para você entender. Eu vou escrever uma classe que se chama `Teste`, que vai ter uma `main` qualquer, e aqui vou fazer o seguinte:

```
public class Teste  {
    public static void main(String[] args)  {
        new CalculadoraDePrecos(tabela, entrega)
    }
}
```

Olha só, dei new nessa classe. Eu preciso passar pra ela uma tabela e uma entrega. Vou criar duas variáveis locais aqui pra ficar mais claro ainda. E agora, qual tabela de preço que eu passo? A que eu quiser! Então, por exemplo, `TabelaDePrecoPadrao`. Qual serviço de entrega eu passo? `new Frete()`:

```
public class Teste  {
    public static void main(String[] args)  {
        TabelaDePreco tabela = new TabelaDePrecoPadrao();
        ServicoDeEntrega entrega = new Frete();
```

E olha só, aqui eu tenho a minha calculadora que funciona de um jeito, com a `TabelaDePrecoPadrao` e com o `Frete`.

```
    CalculadoraDePrecos calculadora = new CalculadoraDePrecos(tabela, entrega);
```

Quando eu tiver uma outra implementação, então muda para `TabelaDePrecoDiferenciada`, dá uma olhada. Esse código, vamos fazê-lo compilar rapidinho.

Vou criar a classe, já implementando a interface, certo, o Eclipse é inteligente. Mas veja só! A minha `CalculadoraDePrecos` continua funcionando! Só que o comportamento dela vai ser diferente, porque, quando ela for usar a `TabelaDePreco`, ele vai usar qual tabela de preço? A diferenciada.

Veja só que eu consegui mudar o comportamento da `CalculadoraDePrecos` sem mexer no código dela. Simplesmente porque eu mudei a ferramenta de trabalho, a dependência que ela recebe. Isso que é uma classe aberta para extensão. Eu consigo mudar como que ela vai funcionar, passando, por exemplo, uma dependência pelo construtor.

Por que que isso deu certo? Porque eu pensei bem esse meu código! Eu criei uma abstração `TabelaDePreco`. É uma interface. Se é uma interface, logo, eu vou ter *n* implementações. E qualquer implementação vai entrar nessa porta da `TabelaDePreco`, o polimorfismo vai fazer a mágica pra mim.

Veja só como está muito melhor. Essa classe evolui agora facilmente. Eu consigo mudar a `TabelaDePreco`, consigo mudar `ServicoDeEntrega`, e esse código `CalculadoraDePrecos` está fechado. Porque eu não vou precisar mexer nele. Então, está aberto para extensão, mas está fechado para modificação. Isso é o tal do **OCP**. Olha só que código bacana, né?!

Legal! Viu o que a gente fez? Eu criei uma abstração, que eu chamei de `TabelaDePreco`, outra de `ServicoDeEntrega`, fiz a minha classe `CalculadoraDePrecos` depender dessas interfaces, e veja bem, essas interfaces são estáveis, elas tendem a mudar muito pouco. Está tudo beleza com o acoplamento, e veja que agora minha classe é aberta! Porque eu consigo mudar o comportamento dela. Então, dependendo da `TabelaDePreco` que eu passar, a minha calculadora vai funcionar de uma maneira diferente. Dependendo da empresa de frete que eu passar, a minha calculadora vai funcionar também de uma maneira diferente.

Ela está aberta para extensão. E veja só como eu estendi: mudando a implementação que eu passo pras dependências no construtor. E ela está fechada para modificação. Eu não preciso ir nela para mudar o comportamento da `TabelaDePreco`, eu não preciso ir nela para mudar o comportamento do `Frete`. Se aparecer um frete novo, eu crio uma nova classe, e a classe `CalculadoraDePrecos` vai continuar funcionando pra isso.

Isso é o tal do **OCP**, o Princípio do Aberto e Fechado. E veja só como eu usei, como eu lidei com ele, como que eu joguei com esse problema aí do acoplamento/ coesão. Criei uma interface, que é estável, recebi pelo construtor, e isso fez agora com que eu possa mudar o comportamento da minha classe principal simplesmente mudando a implementação que eu estou passando pra essa classe `CalculadoraDePrecos`.

Isso é programar orientado a objetos. É pensar em abstração. Quando eu tenho uma boa abstração, eu consigo simplesmente evoluir o meu sistema criando novas implementações das abstrações em que eu já pensei antes. Agora meu sistema está lindo e maravilhoso. Ele evolui facilmente, basta eu criar novas implementações, as classes são todas coesas, são simples, são fáceis de serem testadas de maneira automatizada.

Mas e esse tal do **DIP**, o Princípio da Inversão de Dependências? Isso você já sabe o que é, eu só não tinha dado o nome. Sabe essa ideia maluca de você sempre depender de classes que são estáveis? Dá pra generalizar esse conceito.

A ideia é: sempre que você for depender, depende de alguém mais estável. Então, *A* depende de *B*, a ideia é que *B* seja mais estável que *A*. Mas *B* depende de *C*. Então, a ideia e que *C* seja mais estável que *B*. A ideia é que você sempre passe a depender de modos mais estáveis que você.

E mais do que isso, esse princípio vai mais longe. Ele fala o seguinte: “Olha, se você está numa classe, tenta depender de abstração. Você não pode depender de implementação. Dependa sempre de abstrações.”

Se você está numa abstração, a ideia é de que a abstração não conheça a implementação. Consegue ver o caminho? Dependa sempre de abstração, porque abstração é estável. Nunca dependa de implementação.

E a abstração, por sua vez, ela só pode conhecer outras abstrações. A ideia é que ela não conheça detalhes de implementação. Isso é o que nós chamamos de *Dependency Inversion Principle*, o Princípio de Inversão de Dependência. Não confunda isso com “injeção” de dependência. Injeção de dependência é aquela ideia de você ter os

parâmetros no construtor, e alguém magicamente injetar essas dependências pra você. O nome é parecido. Aqui é o princípio da **inversão** de dependência. A ideia é que você está invertendo a maneira de você depender das coisas. Passa a depender agora de abstrações.

Isso é **OCP** e isso é **DIP**. Eu deixei pra falar dele só agora, porque agora você entende bem o que é coesão, entende bem o que é acoplamento, e entende estabilidade. Agora você tem ferramenta suficiente pra jogar e entender essa ideia aí do OCP.

Sempre que eu programo, eu penso muito em abstração. O tempo inteiro. Porque a abstração vai me dar um monte de vantagem. Ela vai deixar que minha classe seja aberta o tempo inteiro, então eu posso mudar, criar uma nova implementação, e minha classe que depende da abstração vai funcionar com ela. A abstração é estável, então ela não vai propagar mudança problemática pra classe principal, e assim por diante.

Programar orientado a objetos é pensar em abstração. Quando eu estou dando aula de Orientação a objetos básica, e aí o cara está vendo pela primeira vez todas aquelas ideias malucas de polimorfismo, herança, encapsulamento etc. e tal, uma brincadeira que eu sempre faço com eles é: no meio da aula eu falo "Gato, cachorro e pássaro". Eu espero que eles me respondam "animal". Eu viro e falo "ISS, INXPTO e sei-lá-o-que-das-quantas", outros nomes de imposto, e eu espero que a pessoa me fale "imposto". Eu faço o tempo inteiro o meu aluno pensar em abstração. Isso é programar orientado a objetos. É pensar primeiro na abstração, e depois, na implementação.

Essa é uma mudança de pensamento com quem programa procedural. Porque no mundo procedural, você está muito preocupado com a implementação. E é natural. No mundo OO, você tem que inverter: a sua preocupação maior tem que ser com a abstração, com o projeto de classes.

Pense no seu projeto. A implementação é importante, o código ali que vai fazer a coisa funcionar, o if, o for, é importante. Mas no sistema OO, pensar no projeto de classes é fundamental. É isso que vai garantir a facilidade de manutenção.

Eu tinha lá a minha `CalculadoraDePrecos` e agora eu dependo de uma interface `Frete`, dependo de uma interface `TabelaDePrecos`. E aí basta eu passar implementações concretas de cada um deles, que a minha `CalculadoraDePrecos` vai mudar.

Resumindo, falei nesse capítulo pra vocês de **classes abertas**, e o tal do **OCP** (o Princípio do aberto e fechado) – a ideia é que suas classes sejam abertas para evolução, mas fechadas pra mudança. E eu falei pra vocês do **DIP** (Dependency Inversion Principle), cuja ideia é você inverter a dependência, e sempre depender de abstrações. Porque abstrações são legais, são estáveis etc. e tal.

Esse foi o conteúdo dessa aula, e isso mostra pra gente as outras duas letrinhas aí do SOLID, o **O** do OCP, e o **D** do DIP.