02

Classes abertas, Open Closed e Dependency Inversion Principles

Olá, pessoal!

Nas aulas passadas discutimos sobre acoplamento e coesão. Quando discutimos 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.

E qual é a vantagem disso?

A vantagem é que se ela muda muito pouco, é melhor que eu me acople à ela, afinal, ela não vai propagar a mudança. Sempre que pensarmos em acoplamento, ou precisarmos nos acoplar com alguma outra classe ou módulo, a ideia é que nos acoplemos com módulos que são estáveis.

Com essa ideia na cabeça, vamos dar uma olhada nesse código:

```
public class CalculadoraDePrecos
    {
        public double Calcula(Compra produto)
        {
            TabelaDePrecoPadrao tabela = new TabelaDePrecoPadrao();
            Frete correios = new Frete();

            double desconto = tabela.DescontoPara(produto.Valor);
            double frete = correios.Para(produto.Cidade);

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

class TabelaDePrecoPadrao
    {
        public double DescontoPara(double valor)
        {
            if(valor>5000) return 0.03;
            if(valor>1000) return 0.05;
            return 0;
        }
    }
class Frete
    {
        public double Para(string cidade)
        {
            if ("SAO PAULO".Equals(cidade.ToUpper()))
            {
                return 15;
            }
            return 30;
```

```
        }
    }
```

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

Perceba que aqui eu tenho várias classes, por que eu estou pensando bastante em coesão. Então, idealmente, eu tenho classes pequenas, bastante coesas e com pouca responsabilidade.

Temos essa `TabelaDePrecoPadrao` , que possui uma regra de negócio - que está em uma classe. Eu tenho a classe `Frete` que calcula o frete também, mas em uma outra classe. E, na calculadora, essa classe depende das outras duas.

Ok, tudo está funcionando perfeitamente. Só que agora pense no seguinte: imagine 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 de entrega de produtos. Dessa forma, imagine que a regra cresceu, ok? De acordo com o produto e com o cliente, eu calculo o preço de maneira diferente.

Como eu vou implementar isso?

Eu tenho duas maneiras. A primeira delas seria colocar um `if` na `CalculadoraDePrecos` . Da seguinte forma: `if(REGRA 1)` , uma regra qualquer - eu não especifiquei uma regra, mas imagine que eu tenho 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.Valor);

        }
        if (REGRA 2){
            TabelaDePrecoDiferenciada tabela = new TabelaDePrecoDiferenciada();
            desconto = tabela.DescontoPara(produto.Valor);
        }
        double frete = correios.Para(produto.Cidade);
        return produto.Valor * (1 - desconto) + frete;
    }
}
```

Faremos a mesma coisa com o frete. Eu não coloquei nesse código, mas imagine a mesma coisa. Se cair na regra 1, use os correios; se cair na regra 2, use a empresa XPTO - também de entrega.

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

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

```
public class Frete {

    public double Para(String cidade) {
         if(REGRA 1)  {
              if("SAO PAULO".equals(cidade.ToUpper())) {
                           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 a complexidade também vai crescer, certo?

Imagine que se eu tiver 10 regras, eu vou ter 10 ifs e, portanto, o código vai ficar difícil de manter. Veja só, no primeiro código que eu dei pra você, o acoplamento ia crescer, porque a classe `CalculadoraDePrecos` ia começar a depender de muitas outras classes. No segundo código, a coesão dessas classes `Frete` e `TabelaDePrecoPadrao` também complicaria.

Então, acoplamento e coesão... Já discutimos anteriormente que a grande graça de programar orientado a objetos é ter um balanço entre essas duas coisas. Eu nunca vou conseguir ter máxima coesão e zero acoplamento. A ideia é

encontrar esse equilíbrio, lembra?

Portanto, o primeiro conceito que eu quero passar pra vocês é a ideia de que as classes devem ser **abertas**.

Mas como assim "aberta"? O que é uma classe aberta?

Eu coloquei a sigla **OCP** (*Open Closed Principle*), que é o princípio que fala disso.

Mas o que é esse princípio de 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, mudar o comportamento dela de maneira fácil; mas ela deve estar fechada para alteração, ou seja, eu não tenho que ficar o tempo inteiro indo nela pra incluir um if, para fazer uma modificação ou alguma coisa do tipo. Então, ela deve ser fechada para modificação para não precisarmos entrar o tempo inteiro nela e escrever códigos e, além disso, ela deve ser aberta para extensão, ou seja, eu tenho que conseguir mudar a execução dela ao longo do tempo.

Complicado, né? Como eu faço isso?

Eu vou mostrar isso 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, por exemplo, `if(regra1)` calcula desse jeito, caso contrário, e assim por diante. Preciso evitar esse if tanto aqui quanto dentro das implementações, da tabela de preço, do frete e etc. Afinal, está tudo perfeito - como eu mostrei pra vocês. Esse código é simples, super coeso, assim como os códigos do `Frete` e da `CalculadoraDePrecos`. 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 `ITabelaDePreco`. O único método vai ser `double descontoPara`:

```
public interface ITabelaDePreco  {
    double DescontoPara(double valor);
}
```

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

```
public class TabelaDePrecoPadrao : ITabelaDePreco  {
```

Vou fazer a mesma coisa pro `Frete`. Vou chamar a nova interface de `IServicoDeEntrega`. O método que ele vai ter é o `double Para`, que recebe uma cidade:

```
public interface IServicoDeEntrega  {
    double Para(String cidade);
}
```

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

```
public class Frete : IServicoDeEntrega  {
```

Ótimo, está tudo perfeito.

Eu sei que essas interfaces tenderão a ser estáveis. Agora, o que eu vou fazer é o seguinte:

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

Esse `new` está me incomodando nesse código. Eu preciso fazer com que seja possível trocar a implementação da `TabelaDePreco`.

Como que eu vou fazer isso?

Eu vou receber pelo construtor. Então, `CalculadoraDePrecos` vai ser uma `ITabelaDePreco` no construtor, vou chamar a interface de `tabela` e ela vai receber um `IServicoDeEntrega` - que eu vou chamar aqui de `entrega`:

```
public class CalculadoraDePrecos  {

    public CalculadoraDePrecos(ITabelaDePreco tabela, IServicoDeEntrega 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.Cidade);
```

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

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

Olha só, eu dei new nessa classe. Eu preciso passar pra ela uma tabela e uma entrega. Vou criar duas variáveis locais 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 observe que 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, observe que muda para `TabelaDePrecoDiferenciada`.

Para deixar mais compilado, eu vou criar a classe já implementando a interfaceMas.

Veja só! A minha `CalculadoraDePrecos` continua funcionando! Só que o comportamento dela vai ser diferente, por que quando ela for usar a `ITabelaDePreco`, ela vai usar a tabela de preço diferenciada.

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

Por que isso deu certo?

Por que eu pensei bem nesse meu código! Eu criei uma abstração `ITabelaDePreco`, uma interface. Se é uma interface, logo eu vou ter *n* implementações. E qualquer implementação vai entrar nessa porta da `ITabelaDePreco`, o polimorfismo vai fazer a mágica pra mim.

Veja só como está muito melhor! Essa classe agora evolui facilmente!

Eu consigo mudar a `ITabelaDePreco`, consigo mudar `IServicoDeEntrega`, e esse código `CalculadoraDePrecos` está fechado - por que eu não vou precisar mexer nele. Então, está aberto para extensão, mas está fechado para modificação. Isso é o **OCP**. Olha só que código bacana!

Legal!

Viu o que a gente fez?

Eu criei abstrações que eu chamei de `ITabelaDePreco` e de `IServicoDeEntrega` e fiz a minha classe `CalculadoraDePrecos` depender dessas interfaces. Observe que essas interfaces são estáveis, elas tendem a mudar muito pouco. Está tudo certo com o acoplamento, e perceba que agora minha classe é aberta, por que eu consigo mudar o comportamento dela. Então, dependendo da `ITabelaDePreco` que eu passar, a minha calculadora vai funcionar de uma maneira diferente. Dependendo da empresa de frete que eu passar, a minha calculadora também vai funcionar de uma maneira diferente. Ela está aberta para extensão. E veja só como eu estendi mudando a implementação que eu passo para as dependências no construtor. E ela está fechada para modificação - eu não preciso ir nela para mudar o

comportamento da `ITabelaDePreco`, 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.

Esse é o **OCP**, o Princípio do Aberto e Fechado.

E veja só como eu usei, como eu lidei com ele, como eu resolvi o problema do acoplamento e da coesão.

Eu criei uma interface, que é estável, recebi pelo construtor, e isso possibilitou a mudança no comportamento da minha classe principal - bastou a implementação da `CalculadoraDePrecos`.

Isso é programar orientado a objetos! É pensar em abstração.

Quando eu tenho uma boa abstração, eu consigo evoluir o meu sistema criando novas implementações para as abstrações que eu já pensei antes. Agora meu sistema evolui facilmente. Basta eu criar novas implementações,pois as classes são todas coesas, são simples, são fáceis de serem testadas de maneira automatizada.

Mas e o **DIP**, o Princípio da Inversão de Dependências?

Isso você já sabe o que é, eu só não tinha dado o nome ainda. É a ideia de depender sempre de classes que são estáveis - generalizando esse conceito.

Portanto, sempre que você for depender, dependa de alguém que seja 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 é que *C* seja mais estável que *B*. Portanto, *C* deve ser mais estável do que *B*, que por sua vez deve ser mais estável do que *A*.

Resumindo, a ideia é que você sempre passe a depender de modos mais estáveis que você.

Mas esse princípio vai mais longe do que isso. Ele fala o seguinte: “Olha, se você estiver em uma classe, tente depender de abstração. Você não pode depender de implementação. Dependa sempre de abstrações.”

Se você está em uma abstração, a ideia é que a abstração não conheça a implementação, entendeu? Dependa sempre de abstração, por que abstração é estável. Nunca dependa de implementação.

E a abstração, por sua vez, 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.

Mas cuidado! Não confunda isso com “injeção” de dependência!

Injeção de dependência é ter os parâmetros no construtor e, assim, ter essa dependências magicamente injetadas pra você. O nome é parecido. Aqui é o princípio da **inversão** de dependência. A ideia é que você está invertendo a maneira como você depende das coisas, pois você passa a depender de abstrações.

Isso é **OCP** e isso é **DIP**.

Eu deixei pra falar dele nesse momento por que agora você entende bem o que é coesão, entende bem o que é acoplamento e entende estabilidade. Agora você tem ferramentas que são suficientes para entender e usá-las segundo o OCP.

Sempre que eu programo, eu penso muito em abstração, por que a abstração vai me dar diversas vantagens. Ela vai deixar minha classe ser 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.

Programar orientado a objetos é pensar em abstração.

Quando eu estou dando aula de Orientação a objetos básica, e os alunos estão vendo polimorfismo, herança, encapsulamento e etc pela primeira vez, eu costumo fazer uma brincadeira. No meio da aula eu falo “Gato, cachorro e pássaro”, esperando que eles associem essas palavras à palavra "animal". Ou seja, eu estou tentando fazer o meu aluno pensar em abstrações. 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. Por que no mundo procedural, você está muito preocupado com a implementação. E isso é natural. Mas 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 que vai fazer a coisa funcionar - o if ou o for - é importante. Mas no sistema OO, pensar no projeto de classes é fundamental. É isso que vai garantir a facilidade da manutenção.

Eu tinha a minha `CalculadoraDePrecos` e agora eu dependo da interface `Frete` e da interface `ITabelaDePrecos`. Dessa forma, basta eu passar as implementações concretas de cada um delas e a minha `CalculadoraDePrecos` vai mudar.

Resumindo, nesse capítulo discutimos **classes abertas** e o **OCP** (o Princípio do aberto e fechado) – a ideia é que suas classes sejam abertas para evolução, mas fechadas pra mudança. Além disso, falamos sobre o **DIP** (Dependency Inversion Principle), cuja ideia é inverter a dependência e sempre depender de abstrações.

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

Até a próxima aula!