02

Acoplamento e a estabilidade

Bem-vindo ao curso de Orientação a objetos avançado do Alura.

Nesta aula, falaremos sobre **acoplamento**.

Acoplamento é um termo muito comum entre os desenvolvedores, em especial entre aqueles que programam usando linguagens OO - lembrando da máxima da Orientação a objetos: "Tenha sempre classes que são muito coesas e pouco acopladas.". Neste capítulo, em particular, deixaremos a parte da coesão um pouco de lado e discutiremos mais sobre acoplamento.

Eu tenho um código acoplado quando uma classe depende da outra. Imagine um sistema grande em que as classes não dependem umas das outras. Isso quase nunca acontece, não é? Então por que será que o acoplamento é ruim?

Observe o código `GeradorDeNotaFiscal`. Se você olhar o método `gera`, que é o principal método dessa classe, você saberia dizer o que ele faz?

O método dessa classe pega uma fatura, descobre seu valor mensal, faz uma conta para descobrir o valor do imposto e gera uma nota fiscal. Em seguida, ele manda um e-mail, `email.EnviaEmail`, persiste no `dao`, `dao.Persiste`, e retorna a nota fiscal. Dessa forma, tanto o enviador de e-mail quanto o dao estão sendo enviados pelo construtor dessa classe. Observe:

```
class GeradorDeNotaFiscal
    {
         private EnviadorDeEmail email;
         private NotaFiscalDao dao;

         public GeradorDeNotaFiscal(EnviadorDeEmail email, NotaFiscalDao dao) {
            this.email = email;
            this.dao = dao;
         }

        public NotaFiscal Gera(Fatura fatura) {

            double valor = fatura.ValorMensal;

            NotaFiscal nf = new NotaFiscal(valor, ImpostoSimplesSobreO(valor));

            email.EnviaEmail(nf);
            dao.Persiste(nf);

            return nf;
        }

        private double ImpostoSimplesSobreO(double valor) {
            return valor * 0.06;
        }
    }
```

Mas qual é o problema desse código? Qual é o problema relacionado ao acoplamento desse código?

Pense no seguinte: hoje, esse código manda um e-mail e o salva no banco de dados usando um dao. Imagine que amanhã ele também vai ter que mandar para o SAP, ele vai ter que mandar um SMS, ele vai ter que disparar um outro sistema da empresa e etc. Essa classe, `GeradorDeNotaFiscal`, vai crescer, vai passar a depender de muitas outras classes.

Com a nossa experiência, nós aprendemos que acoplamento é uma coisa ruim - "nunca acople o seu sistema", "faça suas classes não serem acopladas" -, mas a pergunta é: por quê? Qual é o real problema do acoplamento? Por que ele é tão ruim assim?

Temos aqui, por exemplo, um `GeradorDeNotaFiscal` que depende do `EnviadorDeEmail`, que depende de um `NFDAO`, e que depende de um `SAP`. O grande problema do acoplamento é que uma mudança em qualquer uma das classes das quais eu dependo pode impactar a minha classe principal. Ou seja, se o `EnviadorDeEmail` parar de funcionar, esse problema pode ser propagado para o `GeradorDeNotaFiscal`. Da mesma forma, se o `NFDAO` parar de funcionar, o problema também será propagado para o gerador, e assim por diante.

Podemos pensar em exemplos de código. Se a interface da classe `SAP` mudar, essa mudança vai ser propagada para o `GeradorDeNotaFiscal`. E o problema é que a partir do momento em que eu tenho muitas dependências, essas dependências podem propagar problemas pra minha classe principal. E é exatamente por isso que o acoplamento é ruim, pois minha classe geradora fica muito dependente, muito frágil, muito fácil de parar de funcionar.

Esse é o problema do acoplamento. E ele é bem fácil de ser enxergado, certo? É por isso que nós temos que tentar diminuí-lo.

Mas será que eu consigo resolver o problema do acoplamento, ou seja, impossibilitar a interdependência gerada pelo acoplamento entre as classes?

É impossível. Nós sabemos que, na prática, quando estamos fazendo sistemas de médio e grande porte, dependências sempre existirão.

O grande ponto é começar a diferenciar os tipos de acoplamento, ou seja, entender quando o acoplamento é realmente problemático ou quando ele é problemático, mas não tanto assim; porque se conseguirmos catalogar, começaremos a evitar os acoplamentos que são realmente perigosos e a acoplar somente as classes que são menos perigosas. Esse é o objetivo - que será alcançado até o final dessa aula.

Não é sempre que o acoplamento causa problemas no meu código. Às vezes, fazemos alguns acoplamentos sem perceber. Por exemplo: em Java, quando eu quero lidar com vários elementos, eu uso uma lista, certo? Quando eu estou escrevendo uma string, eu uso a classe `String - String` e `List` são classes como qualquer outra classe, e quando eu as uso, eu estou me acoplando com elas.

Mas aí é que está.

Quando eu, por exemplo, me acoplo com `IList`, eu não enfrento tantos problemas quanto quando eu me acoplo com um DAO, com um `EnviadorDeEmail` ou com qualquer outra classe que tenha uma regra de negócio associada - a mesma coisa acontece com `String`.

O ponto é: por quê? Qual é a característica de `IList` ou de `String` que torna seu acoplamento menos problemático?

Perceba que essa é a questão chave da nossa discussão, por que se descobrirmos o segredo da interface `IList`, conseguiremos replicar esse segredo para as outras classes, superando o problema do acoplamento.

Você pode pensar assim: "Poxa, acoplar com `IList` não gera problemas porque `IList` é uma interface que o C# fez. Vem na linguagem C#". A mesma coisa com a classe `String`, " `String` vem com o C#". Mas não é isso que os torna

menos suscetíveis às consequências do acoplamento.

A resposta, na verdade, é uma característica que ambas possuem e que eu preciso replicar nas minhas classes.

A interface `IList` possui várias implementações - `List`, `LinkedList`, resumindo, qualquer coisa `List`. No desenho eu coloquei a `GoogleList` - o Google possui diversas bibliotecas que fazem uso da interface `IList`. Além disso, existem muitas classes que usam e que dependem da `IList`.

Suponha que o nosso código, o `GeradorDeNotaFiscal`, por exemplo, depende de `IList`. Isso seria um acoplamento também.

Agora, imagine que você está programando o C#, você está criando a linguagem C#, você tem acesso ao código-fonte de `List`, `LinkedList` e etc, e eu peço para você uma mudança na interface `IList`. Você vai fazer essa mudança?

É claro que não! Por que você sabe que essa mudança é difícil. Mudar a interface `IList` implica em mudar a classe `List`, a classe `LinkedList` e assim por diante. `IList` é uma interface muito importante do sistema. Não podemos mexer nela por que essa mudança quebrará muitas outras classes. Isso faz com que a interface `IList` seja o que chamamos de **estável**. Ou seja, ela tende a mudar muito pouco. E se ela tende a mudar muito pouco, quer dizer que a chance dela propagar um erro ou uma mudança para classe que a está usando é menor, entendeu?

Ou seja, se a minha classe depende da `IList`, isso não é um problema por que a `IList` não muda. E, se ela não muda, não haverão impactos propagados por ela nas outras classes. Esse é o ponto. Queremos acoplar classes, interfaces e módulos que sejam estáveis, que tendem a mudar muito pouco.

Essa é a diferença da `IList`: ela muda muito pouco. O nome dado para o acoplamento realizado com esse tipo de interface é **acoplamento aferente**, ou seja, o acoplamento aferente mostra as classes que dependem dessa outra que possui uma propagação de mudança pequena - nesse caso, a `IList`. Para facilitar, vamos nos colocar no lugar das classes. No caso do acoplamento aferente, a lógica é essa: "Eu sou uma classe, e o acoplamento aferente mostra quem depende de mim.". Já o **acoplamento eferente** funciona de modo contrário, na medida em que nos mostra as classes das quais dependemos - exemplo: a classe `GeradorDeNotaFiscal` depende de `NFDAO`, de `SAP`, de `EnviadorDeEmail` e etc. Ou seja: "Eu, classe, dependo de outras.". Olha só a diferença!

E o que isso nos mostra?

Quando temos muitas classes que dependem de uma classe em específico, essa classe se torna estável, ou seja, seu módulo se torna estável - esse é o acoplamento aferente. Então, o acoplamento eferente se torna importante por que nos possibilita enxergar com clareza classes que são estáveis.

Voltemos ao nosso código. Nós temos um `GeradorDeNotaFiscal` que depende da `String` e da `IList`; esse acoplamento é mais estável. Temos também `EnviadorDeEmail`, `NFDAO`, `SAP` e etc; esses acoplamentos são mais perigosos, eles podem mudar.

Como podemos redesenhar para fazer com que o `GeradorDeNotaFiscal` dependa majoritariamente de classes estáveis? Como criar alguma coisa no sistema que seja estável?

Faremos isso da mesma forma que o pessoal lá da Microsoft fez com `IList`: subjugaremos as implementações às nossas classes.

Na classe `GeradorDeNotaFiscal`, temos uma interface `AcaoAposGerarNota` e essa interface é implementada por `SAP`, por `EnviadorDeEmail` e por `NFDAO`. Agora imagine que existam outras 10 implementações embaixo - que são ações que eu executo depois de gerar a nota. Essa interface que acabamos de criar, `AcaoAposGerarNota`, virou estável. A chance dela

mudar vai ser menor, por que você, programador, vai ter medo de mexer nela, pois se você criar um método a mais, você mudará uma assinatura de algum método e você vai ter que mudá-la em todas as implementações abaixo.

E se eu fizer o meu `GeradorDeNotaFiscal` parar de depender do `SAP`, do `EnviadorDeEmail`, e do `NFDAO`, e passar a depender agora de um monte de `AcaoAposGerarNota`, nós resolveríamos o problema do acoplamento?

Sim! Por que passaríamos a depender de uma classe que é bastante estável.

É por esse motivo que interfaces possuem um papel essencial em sistemas orientados a objetos. É sempre legal lembrar da ideia de programar voltado para interfaces.

Por quê? Porque além de ganhar flexibilidade - por ter várias implementações embaixo daquela interface -, a interface tende a ser estável - e, se ela é estável, acoplá-la nos trará problemas menores.

Essa é a grande ideia pra reduzir o problema do acoplamento: não é deixar de acoplar, é começar a acoplar com coisas estáveis, ou seja, que tendem a mudar menos. Interface é um bom exemplo disso. Interfaces tendem a mudar menos por que elas possuem diversas implementações embaixo, geralmente só possuem um contrato e não possuem um código dentro.

Vamos ver isso no código. Eu tenho aqui o `GeradorDeNotaFiscal`, e ele depende do `EnviadorDeEmail` e do `NotaFiscalDao`. E agora eu sei que pra resolver o problema do acoplamento, eu preciso criar uma interface, essa que, mais pra frente, vai ser estável:

```
class GeradorDeNotaFiscal
    {
         private EnviadorDeEmail email;
         private NotaFiscalDao dao;

         public GeradorDeNotaFiscal(EnviadorDeEmail email, NotaFiscalDao dao) {
            this.email = email;
            this.dao = dao;
         }

        public NotaFiscal Gera(Fatura fatura) {

            double valor = fatura.ValorMensal;

            NotaFiscal nf = new NotaFiscal(valor, ImpostoSimplesSobreO(valor));

            email.EnviaEmail(nf);
            dao.Persiste(nf);

            return nf;
        }

        private double ImpostoSimplesSobreO(double valor) {
            return valor * 0.06;
        }
    }
```

Vamos lá!

O que eu vou fazer aqui é criar uma interface, `Ctrl + N`, `Interface`; vou chamar de `IAcaoAposGerarNota`. Essa interface vai ter um método que eu vou chamar de `executa()`, e ela vai receber uma nota fiscal - veja só, todos eles recebem uma nota fiscal; então, vou receber uma nota fiscal aqui também:

```
void Executa(NotaFiscal nf);
```

O gerador vai parar de receber cada um deles em particular - `EnviadorDeEmail email, NotaFiscalDao dao` -, e vai começar a receber uma lista de `AcaoAposGerarNota` - vou chamá-la de `acoes`:

```
public GeradorDeNotaFiscal(IList<IAcaoAposGerarNota> acoes)  {
```

Excelente.

Agora, vamos tirar do código o que nós não precisamos mais - `private EnviadorDeEmail email; private NotaFiscalDao dao;` - porque agora nós temos uma lista de ações. E aqui, no lugar de fazer `email.EnviaEmail(nf); dao.Persiste(nf);`, vamos fazer um loop. Para cada `IAcaoAposGerarNota acao` na lista de `acoes`, faremos um `acao.Executa(nf);`:

```
public class GeradorDeNotaFiscal  {
    private List<AcaoAposGerarNota>  acoes;
    public GeradorDeNotaFiscal(List<IAcaoAposGerarNota>  acoes)  {
        this.acoes = acoes;
    }
    public NotaFiscal gera(Fatura fatura)  {
        double valor = fatura.ValorMensal;
        NotaFiscal nf = new NotaFiscal(valor , impostoSimplesSobre0(valor));
        foreach(AcaoAposGerarNota acao  in  acoes)  {`
            acao.Executa(nf);
        }
        return nf;
}
```

Quem já conhece padrão de projeto (Design Patterns) - ou fez o nosso curso online sobre o assunto - percebeu que o que eu fiz aqui foi usar o padrão chamado **observer**. E veja só como ele resolve bem o problema do acoplamento! Passamos a depender de uma lista de ações, `IAcoesAposGerarNota`, e assim que a ação acontece nós notificamos todas as ações para que cada uma delas faça o seu trabalho.

Precisamos agora implementar essas ações. Começaremos com o dao, implementando o `IAcaoAposGerarNota`:

```
public class NotaFiscalDao : IAcaoAposGerarNota  {
```

Está certo?

Não! Ainda falta adicionar o método `executa` e para isso, substituiremos o nome:

```
public void Executa(NotaFiscal nf)  {
```

A implementação de exemplo aqui é o `Console.WriteLine`, mas na prática, é o código que acessa o banco de dados e etc.

Faremos a mesma coisa no `EnviadorDeEmail`:

```
public class EnviadorDeEmail : IAcaoAposGerarNota  {

    public void Executa(NotaFiscal nf)  {
```

Agora, na hora de instanciar o `GeradorDeNotaFiscal`, basta passarmos todas as ações que queremos - `new NotaFiscalDao`, `new EnviadorDeEmail` -, e assim por diante.

E assim, resolvemos o problema do `GeradorDeNotaFiscal` usando interfaces e, mais do que isso, dependendo de um módulo estável que nós criamos, o `IAcaoAposGerarNota`.

Olha só que simples!

Nesse capítulo nós discutimos sobre o que é acoplamento e por que ele gera problemas. O acoplamento será problemático quando ele propagar mudanças em uma classe que depende de outra, e essas mudanças afetarão a classe principal.

E como podemos evitar isso?

Devemos acoplar classes, interfaces e módulos que sejam estáveis. Lembre-se: um módulo estável é aquele que tenta mudar pouco, ou seja, ele possui alguma coisa ao redor dele que o faz mudar muito pouco. E eu mostrei que, no caso da interface, o número de implementações embaixo, o número de pessoas usando aquela interface, são uma força contra mudança.

Então, acople-se com coisas que são estáveis e evite ao máximo acoplamento com coisas instáveis no seu sistema!

Por hoje é isso!

Obrigado.