02

Acoplamento e a estabilidade

Bem-vindo à aula de acoplamento do curso de Orientação a objetos avançada do Alura. Nesta aula, eu vou discutir um pouquinho sobre acoplamento.

Acoplamento é um termo muito comum entre os desenvolvedores, em especial entre aqueles que programam usando linguagens OO. Até porque tem aquela grande frase, a máxima da Orientação a objetos, que é "Tenha classes que são muito coesas e pouco acopladas."

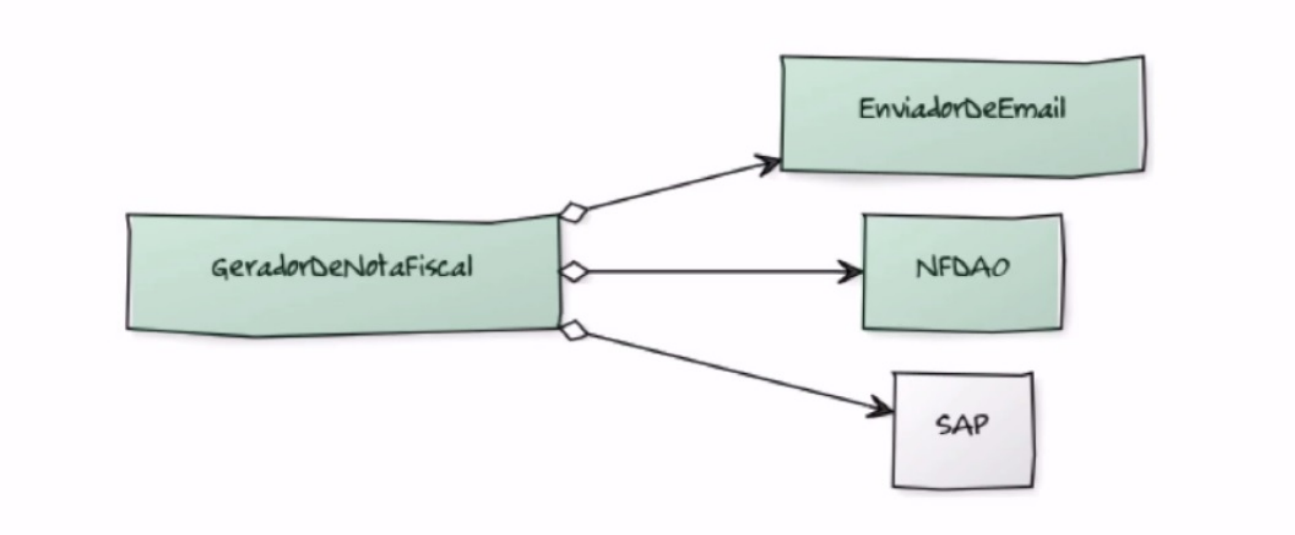
Neste capítulo, em particular, a parte de coesão vou dar uma deixada de lado, e vou discutir um pouquinho mais sobre acoplamento. Mas vamos lá. Dá uma olhada nesse código. Eu tenho um `GeradorDeNotaFiscal` e se você olhar o método `gera`, que é o principal método dessa classe, o que ele faz? Ele pega uma fatura, descobre o valor mensal da fatura, gera uma nota fiscal, certo, faz uma conta lá qualquer com o valor da fatura, isso não vem ao caso. Em seguida, ele manda um e-mail, olha lá `email.enviaEmail`, e depois ela persiste no `dao`, `dao.persiste`. E aí retorna a nota fiscal.

Tanto o enviador de e-mail, quanto o dao, eu estou recebendo ali no construtor da classe `GeradorDeNotaFiscal`. Excelente. Qual que é o problema desse código? Qual que é o problema do ponto de vista de acoplamento desse código?

Pensa o seguinte: hoje, esse código aqui em particular, ele manda e-mail e ele salva no banco de dados usando um dao. Imagina que amanhã ele também vai ter que mandar pro SAP, ele vai ter que mandar um SMS, ele vai ter que disparar um outro sistema da empresa etc.

Essa classe `GeradorDeNotaFiscal`, ela vai crescer, ela vai passar a depender de muitas outras classes.

A gente acaba sempre aprendendo que acoplamento é uma coisa muita ruim, "nunca acople o seu sistema", "faça suas classes não serem acopladas", mas a pergunta é: por quê? Qual que é o real problema do acoplamento? Por que ele é tão ruim assim? Dê uma olhada no diagrama abaixo:

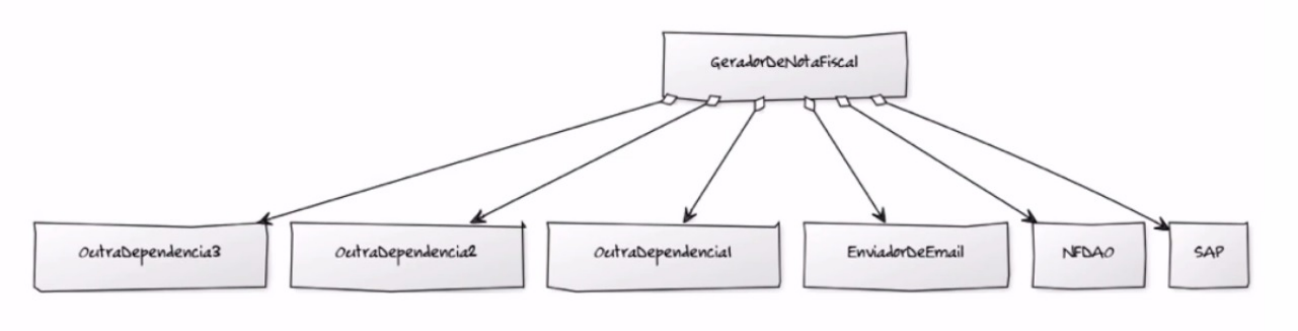


Hoje, eu tenho um `GeradorDeNotaFiscal` que depende do `EnviadorDeEmail`, que depende de um `NFDAO`, e que depende de um `SAP`.

O grande problema do acoplamento é que uma mudança em qualquer uma das classes de que eu dependo pode impactar na minha classe principal. Ou seja, se o `EnviadorDeEmail` parar de funcionar, esse problema pode ser

propagado pro `GeradorDeNotaFiscal`. Se o `NFDAO` parar de funcionar, o problema vai ser propagado pro gerador. E assim por diante.

Posso até pensar em exemplos de código. Se a interface da classe `SAP` mudar, essa mudança vai ser propagada para o `GeradorDeNotaFiscal`. Então, o problema é: a partir do momento em que eu tenho muitas dependências, a minha classe depende de várias.



Quer dizer que eu tenho muitas outras classes que podem propagar problemas pra minha classe principal. E é exatamente por isso que o acoplamento é ruim. Minha classe geradora, ela fica muito dependente, muito frágil, muito fácil de ela parar de funcionar.

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

Mas agora a grande pergunta é: será que eu consigo zerar o acoplamento? Ou seja, resolver o problema do acoplamento, nenhuma classe vai se acoplar com ninguém. É impossível. Nós sabemos que, na prática, quando estamos fazendo sistemas de médio, grande porte, dependências existirão. O acoplamento vai existir. Eu vou depender de uma classe, minha classe vai depender de outra, e assim por diante.

O grande ponto aqui é começar a diferenciar os tipos de acoplamento. Quando que o acoplamento é realmente problemático, e quando que ele é problemático, mas não tanto assim? Porque se eu conseguir catalogar, eu vou começar a evitar acoplamentos que são realmente perigosos e me acoplar com coisas que são menos perigosas. Essa é a charada, é aonde eu quero chegar nesta aula.

O ponto é: não é sempre que eu fico incomodado com acoplamento. Alguns acoplamentos eu nem lembro que eu estou fazendo. Por exemplo, em Java, quando eu quero lidar com um monte de elementos, eu normalmente uso uma lista, certo? Quando eu estou escrevendo uma string, eu uso a classe `String`. `String` e `List` são classes do mesmo jeito que qualquer outra classe sua. E quando eu faço uso delas, eu estou me acoplando com elas.

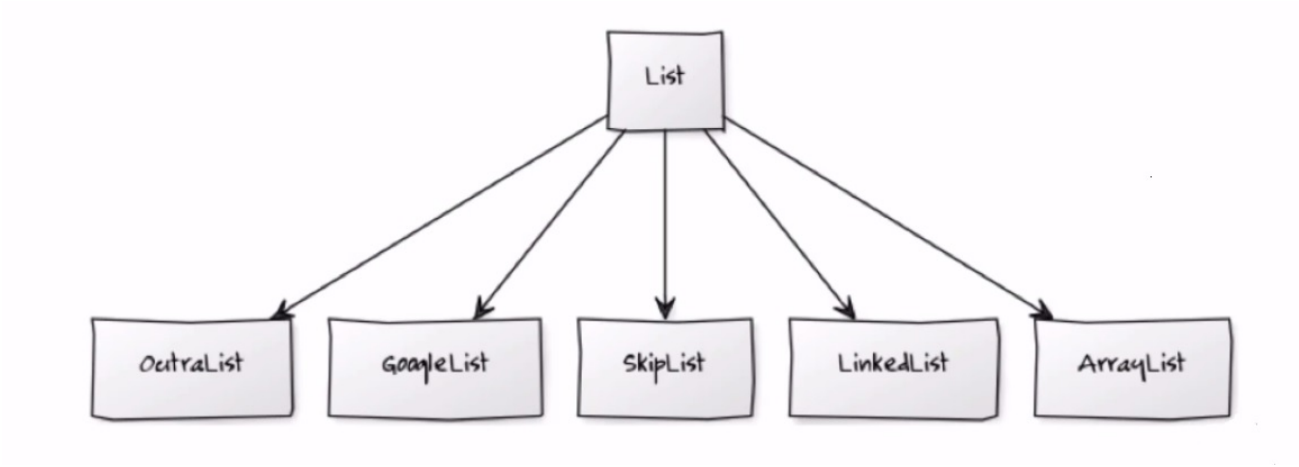
Mas aí que está. Quando eu me acoplo com `List`, eu não sinto tanta dor no coração quanto eu sinto quando eu me acoplo com um DAO por exemplo. Ou com um `EnviadorDeEmail`, ou com qualquer outra classe que tenha uma regra de negócio associada. A mesma coisa com `String`. Eu me acoplo a ela, mas me incomoda menos. É um acoplamento que não me dói.

O ponto é: por quê? Qual é a característica de `List` e qual é a característica de `String` que faz com que eu tenha menos medo de me acoplar do que com as outras classes? E veja só que essa é a questão chave, porque se eu descobrir o segredo da interface `List`, eu vou conseguir replicar esse segredo pras minhas classes. E aí, do mesmo jeito que eu não me importo em me acoplar com `List`, eu não vou me importar em me acoplar com alguma outra coisa do meu sistema.

Quando que é bom, e quando que é ruim? As pessoas geralmente falam assim: “Puxa, acoplar com `List` não é problema porque `List` é uma interface que o Java fez. Vem na linguagem Java”. A mesma coisa com a classe `String`,

“ `String` vem com o Java”. Mas não é bem essa resposta que eu procuro.

A resposta, na verdade, é uma característica que `List` tem, que `String` tem, que eu preciso replicar nas minhas classes. Dê uma olhada:



A interface `List`, quantas implementações ela tem embaixo dela? `ArrayList`, `LinkedList`, qualquer outra coisa `List`. Aqui no desenho eu coloquei `GoogleList` - o Google tem um monte de bibliotecas que fazem uso da interface `List` - etc. e tal.

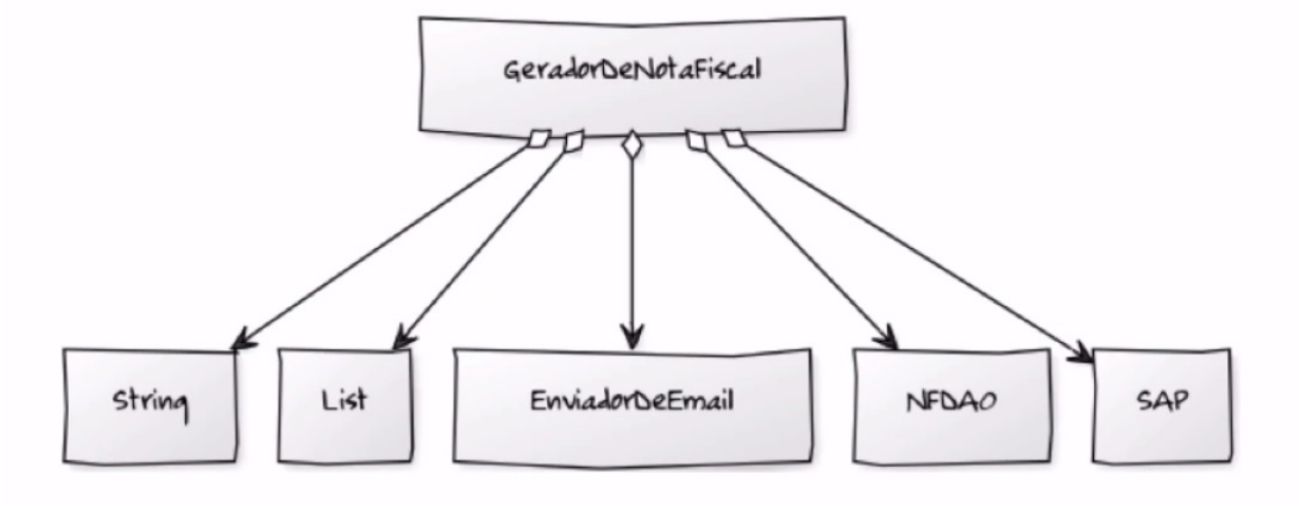
Eu tenho um monte de implementações de `List`. Além disso, eu tenho muitas classes que usam `List`, que dependem de `List`. O seu código, por exemplo, o meu `GeradorDeNotaFiscal`, suponha que ele depende de `List`. É um acoplamento também. Agora, imagina que você está nesse cenário. Você programa lá o Java, você está criando a linguagem Java, você tem acesso ao código-fonte de `List`, `ArrayList`, `LinkedList` etc. e eu peço pra você uma mudança na interface `List`. Você vai fazer essa mudança?

É claro que não! Porque você sabe que essa mudança é difícil. Mudar a interface `List` implica em mudar a classe `ArrayList`, a classe `LinkedList` e assim por diante. `List` é uma interface muito importante do meu sistema. Eu não posso mexer nela porque eu sei que essa mudança vai quebrar muitas outras classes. Isso faz com que a interface `List` 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 de ela propagar um erro, uma mudança, pra classe que a está usando é menor. Consegue ver isso?

Ou seja, se a minha classe depende de `List`, isso não é um problema porque `List` não muda. Se ela não muda, eu não vou sofrer impacto com a mudança dela. Esse é o ponto. Eu quero me acoplar com classes, interfaces, módulos, que sejam estáveis. Que tendem a mudar muito pouco.

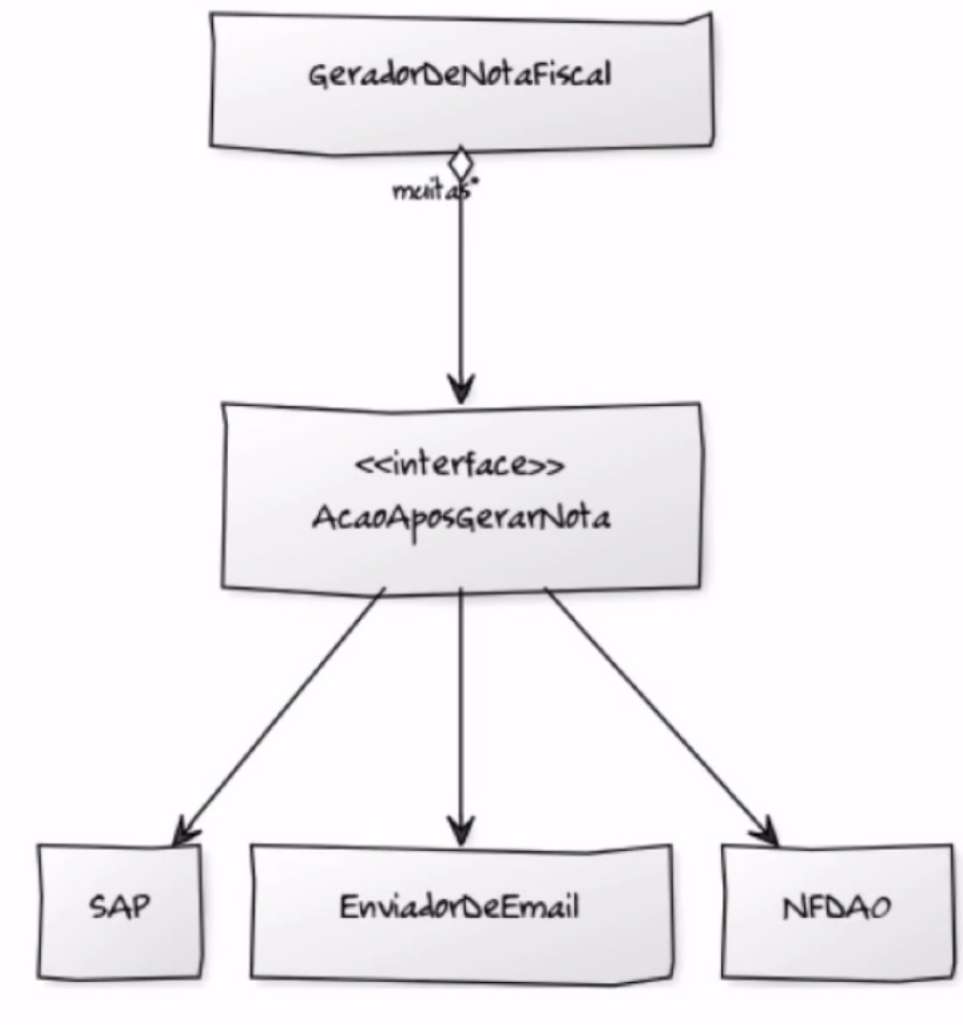
Essa é a diferença de `List` pro resto. `List` muda muito pouco. O nome disso em particular – a gente está sempre acostumado a ver o acoplamento daquele ponto de vista onde eu tenho uma classe e eu dependo de várias outras. Classe `GeradorDeNotaFiscal` depende de `NFDAO`, de `SAP`, de `EnviadorDeEmail` etc. Isso é o que nós chamamos de **acoplamento eferente**. Eu, classe, dependo de outras. Mas o outro lado do acoplamento, que é o que eu estou mostrando pra vocês na interface `List` é também importante, e nós chamamos isso de **acoplamento aferente**. Eu sou uma classe, e o acoplamento aferente mostra quem depende de mim. Olha só a diferença.

E o que isso mostra pra mim? Quando eu tenho muitas outras classes que dependem de uma classe em específico, isso faz com que essa classe seja estável, com que esse módulo seja estável. Então, o acoplamento do outro lado é importante pra dar essa visão pra gente, de coisas que são estáveis. E por que eu quero isso? Porque eu quero me acoplar com coisas estáveis. Dê uma olhada:



Isso é o que nós temos hoje. `GeradorDeNotaFiscal` depende de `String`, depende de `List`, esse acoplamento me incomoda menos, e aí eu tenho `EnviadorDeEmail`, `NFDAO`, `SAP` e assim por diante. Esses acoplamentos são mais perigosos, pois podem mudar.

O ponto é: como que eu consigo redesenhar isso de maneira a fazer com que o `GeradorDeNotaFiscal` dependa agora de coisas que são estáveis? Como que eu crio alguma coisa no meu sistema que é estável? Do mesmo jeito que o pessoal lá da *Sun* (ou da *Oracle* hoje), fez com `List`. Eu tenho uma interface `List` e eu tenho várias implementações embaixo.



Eu tenho agora o meu `GeradorDeNotaFiscal`, eu tenho uma interface `AcaoAposGerarNota`, e essa interface é implementada por `SAP`, por `EnviadorDeEmail` e por `NFDAO`. Imagina só que eu tivesse mais 10 outras implementações embaixo, que são ações que eu executo depois de gerar a nota.

Essa interface que eu acabei de criar, ela acabou de virar estável. A chance de ela mudar vai ser menor. Porque você, programador, vai ter medo de mexer nela. Mexeu nela, criou um método a mais, mudou uma assinatura de algum método, você vai ter que mudar em todas as implementações abaixo. Isso vai fazer com que ela seja estável, naturalmente.

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`, eu resolvi o problema do acoplamento. Porque agora eu dependo de algo que é bastante estável. É por isso, pessoal, que interfaces têm um papel muito importante em sistemas orientados a objetos. É sempre legal aquela ideia de "Programe voltado pra interface".

Por quê? Porque além de eu ganhar flexibilidade, certo - porque eu posso ter várias implementações embaixo daquela interface -, aquela interface, ela tende a ser estável. E se ela é estável, me acoplar com ela é um problema menor. Tá certo?

Essa é a grande ideia pra reduzir o problema do acoplamento. Não é deixar de acoplar. É começar a acoplar com coisas estáveis, coisas que tendem a mudar menos. Interface é um bom exemplo disso. Interfaces tendem a mudar menos, porque têm um monte de implementação embaixo, porque interface geralmente só tem um contrato, não tem um código ali dentro, isso faz com que ela seja estável.

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

```
public class GeradorDeNotaFiscal {

    private final EnviadorDeEmail email;
    private final NotaFiscalDao dao;

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

    public NotaFiscal gera(Fatura fatura) {

        double valor = fatura.getValorMensal();

        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 `AcaoAposGerarNota`. Essa interface vai ter um método que eu vou chamar de `executa()`, e ela vai receber – veja só, todos eles recebem uma nota fiscal, certo, todas as dependências recebem uma nota fiscal – então, vou receber também uma nota fiscal aqui.

```
void executa(NotaFiscal nf);
```

Aqui no meu gerador, eu vou parar de receber cada um deles em particular (`EnviadorDeEmail email, NotaFiscalDao dao`) e vou começar a receber uma lista de `AcaoAposGerarNota` e vou chamar de `acoes` .

```
public GeradorDeNotaFiscal(List<AcaoAposGerarNota> acoes)  {

}
```

`List` , ele vai importar pra mim, é uma lista convencional do `java.util` . Vou tirar esses caras fora (`private final EnviadorDeEmail email; private final NotaFiscalDao dao;`), certo, agora eu não preciso mais, porque agora eu tenho uma lista de ações. E aqui, em vez de fazer `email.enviaEmail(nf); dao.persiste(nf);` , eu vou fazer um loop. Para cada `AcaoAposGerarNota acao` na lista de `acoes` , faco `acao.executa(nf);` .

```
public class GeradorDeNotaFiscal  {
    private List<AcaoAposGerarNota>  acoes;
    public GeradorDeNotaFiscal(List<AcaoAposGerarNota>  acoes)  {
        this.acoes = acoes;
    }
    public NotaFiscal gera(Fatura fatura)  {
        double valor = fatura.getValorMensal();
        NotaFiscal nf = new NotaFiscal(valor , impostoSimplesSobre0(valor));
        for(AcaoAposGerarNota acao  :  acoes)  {`
            acao.executa(nf);
        }
        return nnf;
}
```

Pra quem já conhece padrão de projeto, e fez até o nosso curso aqui online sobre o assunto, percebeu que o que eu fiz aqui foi usar o observer. Esse padrão aqui chama `observer` . E veja só como ele resolve bem o problema do acoplamento. Eu passo a depender de uma lista de ações, `AcoesAposGerarNota` , e assim que a ação acontece, eu notifico todas as ações pra que cada uma delas faça seu trabalho.

O que eu preciso agora é implementar essas ações. No `dao` , tenho que implementar `AcaoAposGerarNota` .

```
public class NotaFiscalDao implements AcaoAposGerarNota  {

}
```

Tá legal? Erro de compilação, claro, porque o método `executa` não está escrito, eu vou substituir o nome:

```
public class NotaFiscalDao implements AcaoAposGerarNota  {
    public void executa(NotaFiscal nf)  {
        System.out.println("salva nf no banco");
    }
}
```

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

A mesma coisa no `EnviadorDeEmail`. Eu vou implementar `AcaoAposGerarNota`. Vou mudar o método aqui de `enviaEmail` para `executa`, que é o método da interface, e tudo funcionando.

```
public class EnviadorDeEmail implements AcaoAposGerarNota  {

    public void executa(NotaFiscal nf)  {
```

Agora, basta eu, na hora de instanciar o `GeradorDeNotaFiscal`, passar todas as ações que eu quero, certo? `new NotaFiscalDao`, `new EnviadorDeEmail`, e assim por diante.

Eu resolvi o problema de acoplamento agora do `GeradorDeNotaFiscal` usando interfaces e, mais do que isso, dependendo de um módulo estável que eu criei, que é o `AcaoAposGerarNota`. Olha só que simples.

Bem, então o que nós vimos neste capítulo? Nós discutimos um pouquinho do que é acoplamento, e por que ele é ruim. O acoplamento é ruim porque quando eu tenho uma classe que depende de outra classe, mudanças nas classes de que eu dependo podem afetar a classe principal. Isso é problemático. Eu preciso diminuir isso.

Como eu faço isso? Eu tento me acoplar com classes, interfaces, módulos, que sejam estáveis. Um módulo estável é aquele que tenta mudar muito pouco. Ele tem alguma coisa ao redor dele que faz ele 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 nela. Então, acople-se com coisas que são estáveis. E evite ao máximo acoplamento com coisas instáveis no seu sistema. E por hoje é isso, obrigado!