02

Entendendo o encapsulamento

Olá! Bem-vindo à aula cujo tema é encapsulamento.

Nós já discutimos acoplamento e coesão. Agora, falaremos sobre o encapsulamento, o 3º pilar, ok?

Vamos começar do jeito que estamos começando todos os nossos capítulos até então, com um código. Dá uma olhada nessa classe:

```
public class ProcessadorDeBoletos
    {
         public void Processa(List<Boleto> boletos, Fatura fatura) {

            double total = 0;

            foreach(Boleto boleto in boletos) {
                Pagamento pagamento = new Pagamento(boleto.Valor, MeioDePagamento.BOLETO);
                fatura.Pagamentos.Add(pagamento);

                total += boleto.Valor;
            }

            if(total >= fatura.Valor) {
                fatura.Pago = true;
            }
        }
    }
```

Temos aqui um `ProcessadorDeBoletos`. O método `processa` deixa bem claro que, dada uma lista de boletos e uma fatura, ele passeia por cada boleto, gera um pagamento e associa esse pagamento à fatura. Além disso, ele possui essa variável de total, onde ele vai somando o valor de todos os boletos. No fim, ele faz uma verificação: se o valor da fatura for menor do que o total pago, quer dizer que a fatura já foi paga por completo. Nesse caso, ele marca a fatura como paga. Entendeu?

Um código também do mundo real, isso parece bastante com aqueles Batch Jobs que estamos acostumados a escrever. Aqueles processos que existem pra fazer verificação.

Tenho certeza de que todo mundo que lida com pagamento e recebe pagamento em boleto, mesmo quando não usa algum código como esse que estamos analisando, usa um códigozinho por trás que faz mais ou menos a mesma coisa.

Agora, a pergunta de sempre: qual é o grande problema desse código?

De volta a ele, vamos lá!

Acoplamento: o acoplamento não parece tão ruim, certo? Por que eu estou dependendo de `Boleto` e de `Fatura`, ou seja, eu estou numa regra de negócio que envolve essas duas entidades. Acoplamento com `List` - como nós já discutimos - não é problemático, pois `List` é uma interface muito estável. Então, o problema não é acoplamento.

Coesão: vamos olhar novamente para o nosso código. Esse código faz uma única coisa, ele cuida da regra de processar um boleto. Ele não está muito bonito, eventualmente eu podia tentar extrair algum método privado e etc, mas o problema nele não é um problema de coesão.

O problema desse código é um problema de **encapsulamento**.

Veja só: essa classe se chama `ProcessadorDeBoletos`. A ideia dela é processar boletos - os *n* boletos, por que o usuário pode pagar com mais de um boleto, né? É por isso que a gente está recebendo uma `List`.

Mas a pergunta é: e amanhã? E se eu fizer o `ProcessadorDeCartaoDeCredito`?

Repare que, nesse código em particular, uma regra de negócio está no lugar errado. Está vendo o `total` que pega o valor do boleto, e lá embaixo o `if` que é usado pra ver se o valor é maior ou igual ao da fatura?

```
        total += boleto.Valor;

    }
    if(fatura.Valor<= total)  {
       fatura.Pago = true;
    }
```

Essa é uma regra que não deveria estar no `ProcessadorDeBoletos`, pois ela é uma regra da fatura. Portanto, isso deveria estar escondido na classe `Fatura`.

E qual é o problema disso? Qual é o problema desta regra estar no `ProcessadorDeBoletos`?

O problema é que amanhã, se aparecer o `ProcessadorDeCartaoDeCredito`, como eu falei, teremos que repetir esse código. Agora imagina que eu tenha 3, 4 processadores diferentes. No momento em que eu tiver que mudar alguma regra dessas - a regra de quando eu marco uma fatura como paga, por exemplo -, eu vou ter que buscar no código onde eu reescrevi essa regra de negócio.

Tudo isso por quê?

Por que essa regra não está escondida.

Veja só: a fatura deve ser a responsável por se marcar como paga. Ela sabe o momento de estar paga e o momento de não estar paga. Até por que, veja só, ela tem a lista de pagamentos. Essa regra podia estar lá dentro, sem qualquer problema do ponto de vista de implementação.

Esse é um problema de encapsulamento.

Encapsular é conseguir esconder o comportamento dentro da classe.

Quando o encapsulamento quebra ou quando ele vaza, o que acontece é que existe uma regra de negócio fora do lugar em que ela deveria estar. E o problema é esse que vocês estão vendo no nosso código. Eu tenho que usar Ctrl + C e Ctrl + V se eu quiser ter essa regra em vários lugares diferentes, portanto, não é tão simples quanto invocar um método.

Vamos lá, próximo código:

```
NotaFiscal nf = new NotaFiscal();
```

```
double valor;
if (nf.ValorSemImposto > 10000)  {
    valor = 0.06 * nf.Valor;
}
else {
    valor = 0.12 * nf.Valor;
}
```

Dá uma olhada.

Temos um problema parecido: eu tenho uma `NotaFiscal` e tenho um `if` que calcula o valor da nota fiscal. Se o valor da nota sem imposto for maior que 10 mil, ele vai calcular o valor da nota de um jeito. Caso contrário, ele vai calcular de outro jeito.

Veja só! esse é um outro exemplo de código que está mal encapsulado.

A classe que tem esse código – imagina que é uma outra classe qualquer, com esse trecho de código – sabe demais sobre como funciona uma nota fiscal. E quem deve saber como funciona uma nota fiscal? A própria classe nota fiscal.

Nós até chamamos códigos como esse, que entendem demais de como a outra classe funciona, de **intimidade inapropriada**.

Por quê?

Por que esse código conhece demais a nota fiscal, ele sabe que se o valor for maior que 10 mil ele tem que fazer o cálculo de uma forma, se não, ele tem que fazer de outra. Não é legal.

Veja só como que eu resolveria isso.

Nesse outro trecho de código, eu tenho isso aqui:

```
NotaFiscal nf = new NotaFiscal();
double valor = nf.CalculaValorImposto();
```

Nesse caso, a regra de calcular o imposto está escondida na `NotaFiscal` .

Muito melhor, muito mais fácil.

Em todo lugar que eu precisar saber o valor do imposto, bastará invocar esse método. Se um dia eu tiver que mudar a regra de negócio, eu vou mudar em um único ponto.

Esse princípio de Orientação a objetos - agora que eu já mostrei pra vocês esses dois lados, do código que está íntimo demais, e do código encapsulado - é o que nós chamamos de **Tell, Don't Ask**, ou seja, "Diga, não pergunte".

Mas como assim, diga e não pergunte?

Dá uma olhada: no código de cima, a primeira coisa que eu estou fazendo é uma pergunta:

```
NotaFiscal nf = new NotaFiscal();
double valor;
if (nf.ValorSemImposto > 10000)  {
```

```
    valor = 0.06 * nf.Valor;
}
else {
    valor = 0.12 * nf.Valor;
}
```

De acordo com a resposta dessa pergunta, eu tomo uma ação: eu calculo o valor de um jeito ou calculo o valor de outro. Isso é perguntar, entendeu? Quando eu tenho um código que pergunta uma coisa pra um objeto, para depois tomar uma decisão, ele é um código que não está seguindo o **Tell, Don't Ask**, ou seja. eu preciso dizer para o objeto o que ele tem que fazer e não perguntar para depois tomar a decisão.

Esse segundo código já faz isso direito:

```
NotaFiscal nf = new NotaFiscal();
double valor = nf.CalculaValorImposto();
```

Nesse código, eu digo para o objeto calcular o valor do imposto. Lá dentro, a implementação vai ser esse `if`, não tem como fugir disso; mas agora, eu estou mandando o objeto fazer alguma coisa. Sempre que você tiver códigos como esse, que perguntam para tomar uma decisão, será uma programação procedural – e isso é bem característica de código procedural, porque quando você está programando em C você não tem como fugir disso.

No mundo OO, nós temos que mandar nas coisas. Devemos mandar o objeto fazer alguma coisa, não perguntar. A partir do momento em que eu pergunto antes de tomar uma decisão, eu estou furando meu encapsulamento, ok?

Agora, vamos lá, vamos tentar descrever bem o que é um código encapsulado.

Quando eu olho para um código, por exemplo este:

```
NotaFiscal nf = new NotaFiscal();
double valor = nf.CalculaValorImposto();
```

Para saber se ele está bem encapsulado, eu preciso fazer duas perguntas.

A primeira é: O que esse método faz? E como que eu sei isso?

Eu sei o que o método faz pelo nome dele. Dessa forma, ele calcula o valor do imposto. É um nome semântico que deixa claro que ele está calculando o valor do imposto.

Ótimo.

A próxima pergunta é: Como ele faz isso? Como ele calcula o valor do imposto?

Se eu olhar só pra esse código, eu não sei responder. Eu não sei dizer qual que é a regra que ele está usando por debaixo dos panos. Eu não sei qual é a implementação do `CalculaValorImposto()`, e isso, na verdade, é uma coisa boa, pois eu nunca posso saber como que um método faz alguma coisa. Eu tenho que deixar isso escondido. Eu tenho que deixar isso encapsulado nele.

Se o método esconde bem, se ele esconde como ele faz o que ele deixa bem claro pelo nome, ou seja, se ele esconde o "como", eu posso trocar essa implementação sem nenhum problema. Se eu entrar no código `CalculaValorImposto()` e mudar a regra, as classes clientes não serão afetadas, elas continuarão a funcionar com a regra nova.

Lembra do código que eu dei no começo? Onde eu tinha primeiro um `if`, para depois tomar a decisão?

Se eu mudar a regra de negócio ali, ela não vai se propagar para todo lugar do meu sistema, entende? Isso é um código encapsulado. O código encapsulado é o código que esconde como o método faz a tarefa dele.

Um exemplo bem comum de código bem encapsulado – e isso as pessoas acertam na maioria das vezes nos códigos OO – são os DAOs. O DAO é aquela classe que acessa um banco de dados, acessa uma fonte de dados qualquer. Geralmente, você faz um `new NotaFiscalDao` e um `nf.PegaTodos()`, por exemplo. Após isso, esse método pega todas as notas fiscais.

Como ele faz?

Não sei!

Não sei se vem de um banco de dados, se ele está consumindo um serviço web, se ele está lendo um arquivo texto. Tudo isso está escondido dentro da implementação desse `PegaTodos()`.

Uma coisa bastante interessante de pensar quando se programa OO é não só pensar na implementação que você está escrevendo naquele momento, mas também nas classes clientes, nas classes que vão usar seu código. Em sistemas OO, é isso que geralmente complica um código, é isso que geralmente faz um sistema se tornar difícil de manter. Os sistemas difíceis de manter são aqueles que não pensam na propagação da mudança. É um sistema em que para fazer uma mudança, você deverá mudar 10 pontos diferentes, por exemplo. Dessa forma, os códigos bem encapsulados resolvem esse tipo de problema, por que você muda em um único lugar e a mudança se propaga.

Pense no sistema OO como um quebra-cabeça: você tem uma peça e as outras peças se encaixam nesse quebra-cabeça. Se o desenho da sua peça estiver feio por dentro, não tem problema: você pode apagar o desenho dessa peça e melhorar, deixá-lo mais bonito. O problema estará nos encaixes dessa peça. Se tem muita peça ao redor usando a sua peça, mudar o desenho será difícil, certo? Esse é o ponto.

Quando você programar orientado a objetos, não pense só no código que você está escrevendo, pense no código que você está usando.

Sempre que eu programo, eu tenho duas classes abertas: a classe que eu estou programando que é a classe principal, por exemplo, a classe `NotaFiscal`, e eu tenho também uma classe que pode ser um método main qualquer, onde eu experimento a minha nota fiscal. Então, eu escrevo a `NotaFiscal` e escrevo uma main, onde eu uso essa nota fiscal pra ver se a interface que eu estou programando está bonita - se está coesa, se está encapsulada, se meu código não está muito acoplado e assim por diante. Dessa forma, na prática, não uso um método main; eu escrevo um teste automatizado pra isso. Se você não sabe o que é um teste, faça o nosso curso de Teste de unidade e TDD, você vai ver que existe muita relação entre um código que é bastante orientado a objetos e um código fácil de ser testado. Dessa forma, pode-se fazer duas coisas ao mesmo tempo: você escreve um teste e esse teste te ajuda a verificar se o seu código está coeso ou se não está coeso, e assim, o sistema acaba sendo testado.

Mas teste também não é o ponto desse meu curso corrente. Se você tiver dúvidas a respeito, dá uma olhada na nossa formação de teste que tem bastante coisa.

Encapsulamento é quando eu consigo encapsular e esconder como as classes fazem as tarefas delas. Esconder implica em não deixar as classes clientes conhecerem a implementação, dessa forma, eu posso mudar a implementação à vontade, pois os clientes não vão perceber que isso aconteceu, ok?

Isso é um código encapsulado. Encapsule o tempo inteiro.

Vamos ver outro exemplo, que é bastante importante:

```
public void AlgumMetodo()  {
    Fatura fatura = PegaFaturaDeAlgumLugar();
    fatura.Cliente.MarcaComoInadimplente();
}
```

Eu tenho uma fatura e eu faço `fatura.Cliente.MarcaComoInadimplente();` - tenho certeza de que você já escreveu um código parecido com esse, `A.GetB.GetC.GetD.MetodoQualquer();` . Você possui aqui uma cadeia de invocações.

E qual é o problema disso?

O problema disso é que também estamos furando o encapsulamento nesse caso.

Imagine que, por algum motivo, a classe cliente muda, ela passa a não ter mais o método `MarcaComoInadimplente()` .

Onde o código vai quebrar?

O código vai quebrar em todos os lugares que usam um cliente, que usa uma fatura e um cliente ao mesmo tempo ou que usa um cliente, como nesse código em particular, de maneira indireta: `fatura.Cliente.MarcaComoInadimplente();` .

Esse é o problema de invocações em cadeia A.GetB.GetC.GetD. Se B, C ou D mudar, o seu código vai quebrar. A ideia é: se você precisar marcar o cliente, uma fatura como inadimplente, você deve fazer alguma coisa parecida com isso aqui:

```
public void AlgumMetodo() {
    Fatura fatura = PegaFaturaDeAlgumLugar();
    fatura.MarcaClienteComoInadimplente();
}
```

Dentro lá da `fatura` , você vai fazer `Cliente.MarcaComoInadimplente` e vai repassar a invocação, não tem problema. Mas o ponto é que você encapsulou a maneira com que a fatura faz pra marcar um cliente como inadimplente.

Mas e se meu cliente mudar?

Tudo bem! Se ele mudar, a classe `Fatura` vai parar de funcionar, mas eu vou mexer num único lugar, que é na classe `Fatura` .

Lembre-se que a ideia é diminuir pontos de mudança.

Eu prefiro ter que mexer no cliente e na fatura, do que ter que mexer no cliente e na fatura em todo mundo que mexe em cliente e em todo mundo que mexe em cliente de maneira indireta através da `Fatura` . Assim, eu diminuo ao máximo os pontos de mudança.

No mundo OO, existe a **Lei de Demeter**, e ela diz mais ou menos isso: "Olha, evite ao máximo fazer essas invocações em cadeia e invoque um método qualquer" - como é o exemplo desse código.

Por quê? O que eu estou ganhando quando eu sigo a Lei de Demeter na maioria dos casos?

Eu estou ganhando encapsulamento. Estou escondendo meu código.

E o que eu ganhei com encapsulamento?

Eu diminui a propagação da mudança.

Vamos então agora refatorar aquele código do nosso `ProcessadorDeBoletos` para um código mais encapsulado, pra você ver como é fácil.

Vamos lá! Vamos corrigir esse código!

Sabemos que o problema do encapsulamento está aqui, certo?

```
if(total >= fatura.Valor) {
        fatura.Pago = true;
}
```

A fatura não pode ser marcada como paga por esse código `ProcessadorDeBoletos` . Essa regra de negócios deve estar dentro do `Fatura` , ou seja, eu tenho que achar um bom lugar pra colocar isso aqui dentro.

Para isso, a primeira coisa que eu vou fazer é começar mudando a visibilidade do setter do Pago:

```
public bool Pago {get; private set;}
```

Getters e setters são bastante perigosos no mundo C# porque a partir do momento em que você dá um setter pra um atributo da sua classe, você está dando a oportunidade de qualquer cliente mudar aquele valor de qualquer jeito. E nem sempre nós queremos isso.

Agora, se está em uma fatura marcada como paga ou não, nesse meu problema em particular, existe uma regra de negócio associada. Então, não podemos esperar que qualquer pessoa do mundo de fora consiga simplesmente falar se está pago ou se não está. Essa regra tem que estar escondida em algum lugar dentro da `Fatura` através de Getters e setters. Mas tenha muito cuidado em criá-los! Especialmente setters, por que você pode eventualmente estar furando seu encapsulamento.

Sabe o que fizemos no primeiro dia de C#, criando tudo como public {get;set;}?

Cuidado, não é bem assim.

Getters publicos são menos problemáticos por que você simplesmente está dando uma informação pra o cliente, para o usuário de fora, e, a não ser que essa informação seja uma outra classe que não esteja encapsulada, ele vai conseguir mudar alguma coisa. Mas setters não. Nos setters, você dá a chance dele fazer qualquer coisa na sua classe.

Imagine se eu desse aqui, por exemplo, um `SetPagamentos(IList<Pagamento> pagamentos)` .

Eu estou dando a chance para o cliente passar uma lista de pagamentos pra mim, jogar a antiga fora, passar uma nova - e eu não sei se essa é a melhor alternativa, ok?

Então eu sempre penso bastante antes de sair criando setters. Veja mesmo que essa classe `Fatura` não tem nenhum agora. O único era aquele `setPago` que foi usado para motivar a discussão com vocês, mas agora eu joguei fora.

Então aqui dentro eu preciso de um método, e esse método vai tomar uma decisão, ou seja, ele vai dizer se a fatura está paga ou não.

Voltando para o código, vamos jogar o que nós não precisamos mais fora:

```
    }
    if(fatura.Valor<= total)  {
       fatura.Pago = true;
    }
```

O `total += boleto.Valor;` não estará mais aqui, e essa regra `double total = 0` também.

Observe: `fatura.Pagamentos.Add(pagamento);` . Já discutimos a Lei de Demeter, e essa fatura de pagamento fura um pouco. Estamos colocando um pagamento que está associado a uma fatura, e a fatura nem percebeu que isso aconteceu! Veja só! Isso não parece uma boa ideia, por que a regra já diz isso. Ao adicionar um pagamento, esse pagamento que foi efetuado pode fazer com que a fatura mude de estado, certo? Um pagamento foi feito; se ele for maior do que o valor da fatura, ela está paga.

Observe como eu vou resolver isso: em vez de fazer `fatura.Pagamentos.Add(pagamento);` , eu vou fazer `fatura.AdicionaPagamento(pagamento);` . E vai passar esse pagamento aqui.

Eu vou criar um método, e veja só a implementação:

```
public void AdicionaPagamento(Pagamento pagamento)  {
    this.pagamentos.Add(pagamento);
```

Ótimo!

A fatura está tomando conta da estrutura dela, então ela sabe que existe uma lista de pagamentos lá dentro, que eu estou adicionando.

Legal.

E, aqui dentro, vou verificar se o valor total dos pagamentos é maior do que o valor da fatura. Se isso for verdade, eu vou setar a fatura como paga:

```
    if(ValorTotalDosPagamentos() >this.valor)  {
        this.Pago = true;
    }
```

Vou implementar o `valorTotalDosPagamentos` só pra gente ver como fica. Vou fazer uma implementação simples mesmo:

```
private double ValorTotalDosPagamentos()  {
    double total = 0;

    foreach(Pagamento p in Pagamentos)  {
    total += p.Valor;
    }
    return total;
}
```

Essa não é a melhor implementação do mundo - por que eu tenho um loop aqui, e dentro um outro loop, então poderia cachear esse valor e etc -, mas lembre-se daquilo que nós discutimos: implementação, código problemático, o algoritmo mais complicado que deveria ser, são problemas fáceis de resolver. Só é preciso vir aqui e mudar o comportamento e tudo continuará funcionando.

O importante nessa aula é perceber que agora esse código está encapsulado.

Veja só: eu crio um pagamento e faço `fatura.AdicionaPagamento(pagamento);` .

O que esse método faz?

Adiciona um pagamento.

Como ele faz? Quais são as regras dentro dele?

Não dá pra saber.

```
public class ProcessadorDeBoletos {

        public void processa(List<Boleto> boletos, Fatura fatura) {

            foreach(Boleto boleto in boletos) {
                      Pagamento pagamento = new Pagamento(boleto.Valor,
                       MeioDePagamento.BOLETO);
            fatura.AdicionaPagamento(pagamento);
             }
           }
}
```

Dentro do `Fatura` implementamos a regra do estar pago ou não estar pago, certo?

```
public void AdicionaPagamento(Pagamento pagamento)  {
    this.Pagamentos.Add(pagamento);
    if(ValorTotalDosPagamentos()>this.valor)  {
    }
}
```

Isso é um código encapsulado!

Quando eu falei pra você sobre getters e setters, eu disse que setters são perigosos. Getters são menos problemáticos por que se você devolver esse getCliente aqui, por exemplo:

```
    public string Cliente {get;set;}
```

E no `ProcessadorDeBoletos` do lado de fora, eu fizer:

```
public class ProcessadorDeBoletos {

    public void processa(List<Boleto> boletos, Fatura fatura) {
    String nomeDoCliente = fatura.Cliente;
```

Ele vai me devolver uma string com o nome do cliente. Mas isso não tem problema, porque se eu mudar o nome do cliente, essa variável não vai afetar minha fatura. Não é o que acontece com a lista.

Agora está claro, está escondido, está encapsulado. Muito melhor.

Legal! Nosso código está bem mais encapsulado agora!

Então, vamos lá: o que é encapsulamento?

Encapsulamento é esconder como a classe implementa as suas tarefas.

Como que eu sei que as minhas classes e métodos estão encapsulados?

Fácil! Basta eu olhar pra ele, ver o nome dele – pegue uma classe que o está usando – e tente responder as duas perguntas: "O quê?" e "Como?".

O “O quê?” você deve ser capaz de responder, por que o nome do método tem que te dizer isso. O “Como?” você não deve conseguir responder, ok? Se você conseguiu responder como a classe faz aquela tarefa - “Ah, ela faz, ela acessa o banco de dados, que eu sei que é um banco de dados porque eu estou tendo que passar a connection do banco de dados pra ela” ou “Ah, eu sei que ela marca a fatura como paga, porque o código está aqui, eu estou vendo um if na minha frente, if nota fiscal maior do que tanto, marca como, sei lá, pago, inativo, eu não sei. Eu estou vendo o código ali” ou qualquer coisa do tipo - seu código não está encapsulado.

Resolva o problema de encapsulamento e isso vai te dar menos trabalho para fazer uma mudança do seu usuário final.

Isso é encapsulamento.

Espero que essa aula tenha te ajudado a perceber quando seu código não está encapsulado e mostrar como resolver esse problema.

Esconda o código, encapsule o código sempre nos lugares certos.

Obrigado.