02

Entendendo o encapsulamento

Olá! Bem-vindo à aula cujo tema é encapsulamento. Nós já discutimos até agora acoplamento e coesão. Falta o 3o. pilar aí, que é esse tal de encapsulamento, certo? Vamos começar do jeito que estamos começando todos os nossos capítulos até então, que é com código. Dá uma olhada nessa classe:

```
public class ProcessadorDeBoletos {

    public void processa(List<Boleto> boletos, Fatura fatura) {
        double total = 0;
        for(Boleto boleto : boletos) {
            Pagamento pagamento = new Pagamento(boleto.getValor(),
        MeioDePagamento.BOLETO);
            fatura.getPagamentos().add(pagamento);

            total += fatura.getValor();
        }

        if(total >= fatura.getValor()) {
            fatura.setPago(true);
        }
    }
}
```

Eu tenho um `ProcessadorDeBoletos`. O método `processa` deixa bem claro que, dadas uma lista de boletos e uma fatura, ele passeia por cada boleto, gera um pagamento, associa esse pagamento à fatura. E aí ele tem essa variável de total, onde ele vai somando o valor de todos os boletos. E 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. E aí ele marca a fatura como paga. Tá certo? Um código também do mundo real, certo, isso parece bastante com aqueles Batch Jobs que estamos acostumados a escrever. Aqueles processos que rodam por trás dos panos pra fazer verificação.

Tenho certeza de que todo mundo ali que lida com pagamento e recebe pagamento em boleto, se fez a implementação na unha, ele tem um códigozinho por trás que faz mais ou menos a mesma coisa.

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

De volta a ele, vamos lá. Acoplamento. Acoplamento não parece tão ruim, certo? Porque eu estou dependendo de `Boleto`, de `Fatura`, não tenho muito como fugir disso, eu estou numa regra de negócio que envolve essas duas entidades. Acoplamento com `List` já discutimos que não é problemático, certo, `List` é uma interface muito estável etc. e tal. Então, não é acoplamento.

Coesão. Vamos olhar esse código. Esse código também faz uma única coisa, ele só toma conta da regra de processar um boleto. Ele não está lá tão bonito, eventualmente eu podia tentar extrair algum método privado e tudo mais, mas também não é problema de coesão.

O problema que está acontecendo aqui é um problema de **encapsulamento**. Veja só. Essa classe se chama `ProcessadorDeBoletos`. A ideia é ela processar boletos, os *n* boletos porque 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`?

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

```
        total += boleto.getValor();

    }
    if(fatura.getValor()<= total)  {
       fatura.setPago(true);
    }
```

Essa é uma regra que não deveria estar no `ProcessadorDeBoletos`. Essa é uma regra da fatura. Isso devia estar escondido na classe `Fatura`. Qual que é o problema disso? Qual que é o problema de esta regra estar jogada aí?

O problema é que amanhã, se aparecer o `ProcessadorDeCartaoDeCredito`, igual eu falei, você vai ter que repetir esse código. Agora imagina que eu tenha 3, 4 processadores diferentes. No momento em que eu tiver que mudar alguma regra dessas - sei lá, mudei a regra de quando eu marco uma fatura como paga - eu vou ter que sair buscando no meu código onde que eu reescrevi essa regra de negócio. Tudo isso por quê? Porque essa regra não está escondida. Veja só, a fatura é que deve ser a responsável por se marcar como paga. Ela que sabe o momento de ela estar paga e o momento de ela não estar paga. Até porque, veja só, ela tem a lista de pagamentos. Essa regra podia estar lá dentro, sem qualquer dor de cabeça do ponto de vista de implementação.

Esse é um problema de encapsulamento. Encapsular é conseguir esconder o comportamento dentro da classe. Quando eu quebro o encapsulamento ou quando eu vazo o encapsulamento, o que acontece é que eu coloco uma regra de negócio fora do lugar em que ela deveria estar. E o problema é esse que vocês estão vendo. Eu tenho que usar do Ctrl + C e Ctrl + V se eu quiser ter essa regra em vários lugares diferentes. Não é tão simples quanto invocar um método.

Vamos lá, próximo código.

```
NotaFiscal nf = new NotaFiscal();
double valor;
if (nf.getValorSemImposto() > 10000)  {
    valor = 0.06 * nf.getValor();
}
else {
    valor = 0.12 * nf.getValor();
}
```

Dá uma olhada. É um problema parecido: eu tenho uma `NotaFiscal`, e tenho um `if` aí "Olha, se o valor da nota sem imposto for maior que 10mil, eu vou calcular o valor da nota de um jeito. Caso contrário, eu vou calcular de outro jeito".

Veja só, esse é um outro exemplo de código que está mal encapsulado. Tá certo? O cara que tem esse código – imagina que é uma outra classe qualquer, com esse trecho de código – ele sabe demais de como funciona uma nota fiscal. E quem que deve saber como que funciona uma nota fiscal? A própria classe nota fiscal.

Nós até chamamos códigos como esse, que entendem demais de como a outra classe funciona, nós chamamos esse mau cheiro de código de **intimidade inapropriada**. Por quê? Por que esse código conhece demais a nota fiscal, ela sabe que se

o valor for maior que 10mil ela tem que fazer assim, se não, ela tem que fazer assado. 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();
```

E aí a regra de calcular o imposto está escondido na `NotaFiscal`. Muito melhor, muito mais fácil. Em todo lugar que eu precisar saber o valor do imposto, basta invocar esse método. Se um dia eu tiver que mudar a regra de negócio, eu vou mudar num único ponto.

Um princípio de Orientação a objetos, galera, agora que eu já mostrei pra vocês esses dois lados, do código que está íntimo demais, e do código encapsulado, um desses princípios 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.getValorSemImposto() > 10000)  {
    valor = 0.06 * nf.getValor();
}
else {
    valor = 0.12 * nf.getValor();
}
```

E aí, 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. Está certo? Quando eu tenho um código que pergunta uma coisa pra um objeto, para aí tomar uma decisão, é um código que não está seguindo o **Tell, Don't Ask**. Ou seja, eu tenho que dizer pro objeto o que ele tem que fazer. Eu não tenho que perguntar, para depois tomar a decisão.

O segundo código aí já faz isso direito:

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

Eu estou mandando o objeto calcular o valor do imposto. Lá dentro, óbvio, a implementação vai ser esse `if` aí, 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 – e isso é bem característica de código procedural, certo, porque quando você está programando lá em C você não tem muito como fugir disso. Isso é programação procedural.

Aqui no mundo OO, eu tenho que mandar nas coisas. Eu mando meu objeto fazer alguma coisa, eu não pergunto. A partir do momento em que eu pergunto para tomar uma decisão, muito provavelmente eu estou furando meu encapsulamento aí. Está legal? 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();
```

Eu quero saber se isso está bem encapsulado. Eu faço pra mim 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, calcula o valor do imposto. É um nome bem semântico, deixa claro pra mim que ele está calculando o valor do imposto. Ótimo, consegui responder essa pergunta, então está legal.

A próxima pergunta é: Como que ele faz isso? Ou seja, como que 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. 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, se ele esconde esse "como" ele está fazendo, o que eu ganho? Eu ganho que eu posso trocar essa implementação sem nenhum problema. Se eu entrar no código `calculaValorImposto()` e mudar a regra, as classes clientes não serão afetadas, todas elas continuarão a funcionar com a regra nova.

Lembra do código que eu dei no começo? Onde eu tinha lá aquele `if` primeiro, para depois tomar a decisão? Se eu mudar a regra de negócio ali, ela não vai se propagar para todo lugar do meu sistema. Está certo? Isso é um código encapsulado. O código encapsulado é o código que esconde como aquele método faz a tarefa dele.

Um exemplo bem comum de código bem encapsulado – 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 lá `new NotaFiscalDao`, por exemplo. E aí você faz `nf.pegaTodos()`. O que esse método faz? 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()`.

Pessoal, uma coisa bastante interessante de você pensar quando você programa OO é você não só pensar na implementação que você está escrevendo naquele momento, mas também pensar 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 difícil de manter. Sistemas difíceis de manter são aqueles que não pensam na propagação de mudança. É um sistema em que você, para fazer uma mudança, tem que mudar em 10 pontos diferentes. Códigos bem encapsulados geralmente resolvem esse tipo de problema, porque você muda num lugar e a mudança se propaga.

Se você pensar 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 está feio por dentro, não tem problema: você pode apagar o desenho dessa peça e melhorar, deixa-lo mais bonito. O problema são os encaixes dessa peça. Se tem muita peça ao redor usando a sua peça, mudar o desenho é 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 – e aquela 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, e eu uso essa nota fiscal. Pra eu 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.

Eu, 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 tem muita relação entre o código que é bastante orientado a objetos e um código fácil de ser testado. Então eu aproveito e mato 2 coelhos com uma única cajadada. Porque eu acabo escrevendo um teste; esse teste me ajuda a verificar se meu código está coeso ou se não está coeso; e aí meu sistema acaba saindo testado.

Mas teste também não é o ponto desse meu curso corrente. Dá uma olhada lá na nossa formação de teste, que tem bastante coisa.

Isso é encapsulamento. É quando eu consigo encapsular e esconder como as classes fazem as tarefas delas. Escondendo, isso quer dizer que as classes clientes não conhecem da implementação, e, por consequência, eu posso mudar a implementação à vontade, que os clientes não vão nem perceber que isso aconteceu. Tá certo? Isso é um código encapsulado. Encapsule o tempo inteiro.

Outro exemplo, que é bastante importante. Dá uma olhada nesse código aqui:

```
public void algumMetodo()  {
    Fatura fatura = pegaFaturaDeAlgumLugar();
    fatura.getCliente().marcaComoInadimplente();
}
```

Eu tenho um método qualquer, eu tenho uma fatura, e eu faço `fatura.getCliente().marcaComoInadimplente();` . Eu tenho certeza de que você já escreveu um código parecido com esse. A.getB.getC.getD.metodoQualquer();. Você tem aí uma cadeia de invocações. Qual que é o problema disso? O problema disso é que eu também estou furando o encapsulamento nesse caso.

Imagina só que, por algum motivo, a classe cliente muda. Mudou o [TODO 12'25 API] dela, ela não tem mais esse método `marcaComoInadimplente()` . Onde que meu código vai quebrar? Meu código vai quebrar em todo lugar que usa um cliente e em todo lugar que usa uma fatura, que usa um cliente ao mesmo tempo. Tá certo? Que usa um cliente, como nesse código em particular, de maneira indireta, porque `fatura.getCliente().marcaComoInadimplente();`

Esse é o problema de invocações em cadeia, A.getB.getC.getD. Se B mudar, ou se C mudar, ou se D mudar, o seu código vai quebrar. A ideia é que, se você precisa marcar o cliente, uma fatura como inadimplente, você faça alguma coisa parecida com isso aqui:

```
public void algumMetodo() {
    Fatura fatura = pegaFaturaDeAlgumLugar();
    fatura.marcaClienteComoInadimplente();
}
```

Dentro lá da `fatura` , você vai fazer `Cliente.marcaComoInadimplente` , certo, 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.

- Ah, 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` . Lembra 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` . Eu diminuo ao máximo pontos de mudança.

No mundo OO, tem até essa **Lei de Demeter**, que é mais ou menos famosa, e ela diz mais ou menos isso: Olha, evita ao máximo a ideia de você fazer essas invocações em cadeia. A.getB.getC etc., invoca 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? Diminuo propagação de mudanças. Certo? Legal.

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

Então vamos lá, vamos corrigir esse código. Sabemos que o problema do encapsulamento está aqui, certo?

```
if(total >= fatura.getValor()) {
        fatura.setPago(true);
}
```

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

A primeira coisa que eu vou fazer é começar tirando esse `setPago` :

```
public void setPago(boolean pago) {
    this.pago = pago;
}
```

Getters e setters, pessoal, são uma coisa bastante perigosa no mundo Java, 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 eu quero isso. Por exemplo, às vezes eu quero, sim, um método [TODO 15'05 `setRua()`] na minha classe `Endereco` , porque mudar rua não tem segredo, é basicamente mudar uma informação pela outra.

Agora, se está numa fatura como paga ou não, nesse meu problema em particular, tem uma regra de negócio associada. Então, não posso querer que qualquer um do mundo de fora consiga simplesmente falar: está pago, ou não está pago. Essa regra tem que estar escondida em algum lugar, aqui na `Fatura` . Getters e setters, pessoal, setters mais em particular, muito cuidado na hora de criá-los. Porque você pode eventualmente estar furando seu encapsulamento.

Sabe aquela coisa que você fez no primeiro dia de Java, que você criou getters e setters pra tudo? Cuidado, não é bem assim. Getters são menos problemáticos, porque você simplesmente está dando uma informação pro cliente, pro usuário de fora, e, a não ser que essa informação seja uma outra classe, que não esteja encapsulada, ele vai conseguir mudar alguma coisa. Mais setters, não, você está dando a chance de ele fazer qualquer coisa na sua classe.

Imagina só se eu desse aqui, por exemplo, um `setPagamentos(List<Pagamento> pagamentos)` . Estou dando a chance pra ele passar uma lista de pagamentos pra mim, jogar a antiga fora, passar uma nova – não sei se essa é a melhor alternativa. Tá certo? 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 era pra eu 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: se a fatura está paga ou não. Voltando aqui pro meu código, deixa eu jogar isso aqui fora, que eu não preciso mais,

```
}
if(fatura.getValor()<= total)  {
   fatura.setPago(true);
}
```

O `total += boleto.getValor();` também não vai estar aqui, e essa regra `double total = 0` não vai estar aqui.

Veja só, `fatura.getPagamentos().add(pagamento);` . Discutimos já a Lei de Demeter, certo, isso aqui fura um pouco. Eu estou enfiando um pagamento, que está associado a uma fatura, e a fatura nem percebeu que isso aconteceu. Veja só, não parece uma boa ideia, até porque a regra já diz isso. Adicionou um pagamento, esse pagamento que foi efetuado pode fazer com que a fatura mude de estado. Não é assim que funciona? O cara fez um pagamento. Se o pagamento for maior do que o valor da fatura, ela está paga.

Veja só como eu vou resolver isso. Em vez de fazer `fatura.getPagamentos().add(pagamento);` , eu vou fazer `fatura.adicionaPagamento(pagamento);` . E vai passar esse pagamento aqui.

Eu vou criar esse 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 a fatura sabe que ela tem uma lista de pagamentos lá dentro, 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;
    }
```

O `valorTotalDosPagamentos` , vou implementar aqui só pra gente ver como fica. Fazer uma implementação simples mesmo:

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

    for(Pagamento p : pagamentos)  {
    total += p.getValor();
    }
    return total;
}
```

Nem é a melhor implementação do mundo, porque eu tenho um loop aqui, e dentro, um outro loop, então poderia cachear esse valor, e coisa e tal. Mas lembra daquilo que nós discutimos: implementação, código problemático, o algoritmo mais complicado que deveria ser, é um problema fácil de resolver. Depois é só vir aqui e mudar o comportamento, e tudo vai continuar funcionando.

O importante nessa aula aqui é perceber que agora esse código está encapsulado. Veja só, eu crio um pagamento, e eu faço `fatura.adicionaPagamento(pagamento);` . O que esse método faz? Adiciona um pagamento. Como ele faz, quais são as regras dentro dele? Daqui eu não sei.

```
public class ProcessadorDeBoletos {
```

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

            for(Boleto boleto : boletos) {
                      Pagamento pagamento = new Pagamento(boleto.getValor(),
                       MeioDePagamento.BOLETO);
            fatura.adicionaPagamento(pagamento);
            }
           }
}
```

Aqui dentro do `Fatura`, veja só, que a gente implementou aquela regra do estar pago, do não estar pago. Está certo?

```
public void adicionaPagamento(Pagamento pagamento)  {
    this.pagamentos.add(pagamento);
    if(valorTotalDosPagamentos()>this.valor)  {
    }
}
```

Isso é um código encapsulado. Quando eu falei pra vocês de getters e setters, aliás, eu falei pra vocês que setters são perigosos. Getters são menos problemáticos, certo, porque se você devolver esse getCliente aqui:

```
    public String getCliente() {
        return cliente;
    }
```

No `ProcessadorDeBoletos`, do lado de fora, se eu fizer, por exemplo:

```
public class ProcessadorDeBoletos {

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

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 aqui, isso não vai afetar minha fatura. Não é o que acontece com a lista. Tanto é que o código antigo era `fatura.getPagamentos().add()`, certo? Se eu não quero que isso aconteça, eu posso bloquear essa classe `Fatura`.

Eu vou dar o `getPagamentos`, mas aqui eu vou usar e abusar da API do Java, vou fazer assim:

```
public List<Pagamento> getPagamentos()  {
    return Collections.unmodifiableList(pagamentos);
}
```

O que isso vai fazer pra mim? Ele vai me devolver uma lista de pagamentos, então aqui eu posso continuar usando `fatura.getPagamentos()` sem problema. Ele vai deixar eu ler essa lista, mas se eu tentar escrever na lista, esse código vai me lançar uma exceção.

Então, essa é outra dicazinha que eu dou pra vocês, que eu sempre faço. Se eu quero que a minha classe seja bastante encapsulada, e eu não quero que os clientes saiam pegando as minhas listas e enfiando coisas dentro dela, sem eu saber o que está acontecendo, eu dou o getter pra lista mas não deixo ele modificar. Agora, toda mudança na lista de pagamentos tem que ser feita pela classe `Fatura`. Eu escondi toda a ideia de você manipular essa lista de pagamentos na classe `Fatura`.

Está claro agora, 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 – pega uma classe que o está usando – e tenta responder as duas perguntas: O quê? E como?

O "O quê?" você tem que ser capaz de responder, porque o nome do método tem que te dizer isso. O "Como?" você não tem que conseguir responder. Está certo? 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 qualquer coisa do tipo, 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". Não está encapsulado. Resolvo o problema de encapsulamento, e isso vai lhe garantir aí depois umas boas horas de sono, porque vai lhe dar menos trabalho para fazer uma mudança do seu usuário final.

Então isso é encapsulamento. Eu espero que essa aula o tenha 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!