01

Introdução

Transcrição

[00:00] Todos nós já nos deparamos com um código que é difícil de manter. O código fica todo misturado, todo espalhado, os dados em um canto, o comportamento espalhado por toda a aplicação.

[00:17] Pra fazer uma alteração temos que abrir todos os arquivos e ver como eles influenciam aquilo que queremos atualizar.

[00:30] Código gigante, métodos que fazem muitas coisas, código com muita responsabilidade, tudo isso são características que podem ser melhoradas usando Boas Práticas de Orientação ao Objeto.

[00:40] Para começar, neste primeiro capítulo vamos ter uma visão geral do que podemos fazer. Então nós começamos criando um projeto aqui, “mkdir” ou “ooruby”.

[00:50] Nesse projeto, vamos criar um arquivo chamado “livro.rb”. Nós vamos trabalhar com uma livraria, então vamos definir o básico, a classe livro: “class livro”.

[01:05] E o nosso livro precisa de um título, um preço, e um ano de lançamento.

[01:15] Então nós criamos accessors para todos esses valores e logo depois criamos dois livros: por exemplo, um livro de “rails”, que o preço é 70, o nome dele é “ Agile Web Development with Rails”, de 2011, e o outro livro de ruby, 60 “Programming Ruby 1.9”, de 2010.

[01:38] Criamos uma array com dois livros: livro rail e livro ruby. Nós criamos um objeto, um outro objeto, preenchemos eles e criamos um array com os dois.

[01:53] Por fim, eu quero imprimir uma nota fiscal para todos esses livros. Então vou utilizar um método chamado “imprime_nota_fiscal” e passar a minha array de livros, os que eu quero imprimir a nota fiscal.

[02:03] Vamos definir esse método “def_imprime_nota_fiscal(livros)”, e pra cada livro, isto é, “livros.each do livro”, eu vou imprimir tanto o título, quanto o preço desse livro.

[02:22] Quando nós voltamos para o terminal, e executamos livro.rb, a saída é o nome e o preço dos dois livros. Voltando ao nosso código, o que acontece se eu alterar o livro.preco para o valor 1?

[02:40] Eu volto para o meu console, rodo de novo o programa e o valor do livro é alterado.

[02:47] Será que faz sentido, qualquer lugar da minha aplicação alterar o valor de um livro?

[02:55] Vamos até a nossa classe livro, e ao invés de disponibilizar o accessor, que é tanto a leitura, quanto a escrita, vamos usar só o reader, que é só uma leitura.

[03:03] Executamos de novo o programa e o resultado é que o método “preco =”, não existe. Claro, não temos mais como atribuir preço, título e nem ano de lançamento.

[03:15] O que queremos dizer é que, quando você cria um livro, esses três campos são obrigatórios, senão o livro não faz sentido.

[03:21] Então no construtor, isso é, no “initialize” do nosso livro, nós vamos receber o título, preço e ano de lançamento e atribuir esses três valores as nossas variáveis membro, aos nossos atributos.

[03:33] Todo esse código cria o livro e seta as variáveis, vira simplesmente uma chamada ao construtor. Construiu o objeto, ele tem os valores. Se executarmos de novo, ele reclama de novo que o método preço não existe.

[03:48] Isso porque, no método “imprime_nota_fiscal”, nós tentamos fazer alguma coisa que não faz sentido algum. Não disponibilizando um método, para aceitar o valor dessa variável, eu consigo controlar o acesso a ela.

[04:04] E a partir daí, eu tenho esse meu valor encapsulado. Executando nosso código, nós temos o resultado que esperávamos. Vamos remover todo esse código, manter só a classe livro e criar outro livro, um livro de algoritmos.

[04:20] Então “algoritmos= Livro.new”, e os parâmetros de sempre: o nome do livro, preço e ano de lançamento. (“Algoritmos”, 100, 1998). Sem segredos.

[04:31] Eu quero enviar um newsletter de todos os livros que são mais antigos que 1999. Por isso, eu vou chamar um método “livro_para_newsletter(algoritmos)”, passar o livro e se ele pertencer a esse período, será impresso na newsletter.

[04:47] Então vamos criar esse método: “def livro_para_newsletter(livro)”. Ele recebe o livro, verifica se o ano de lançamento for menor que 1999, imprime newsletter de liquidação, o título e o preço do livro.

[05:03] Rodamos o programa e imprime o livro de Algoritmos. Afinal, pois ele é de 1998. Voltando ao nosso código, agora queremos aplicar uma promoção para os livros anteriores a 2000.

[05:20] Então de novo, vamos criar uma chamada para “aplica_promocoes” e se o livro for anterior ao ano 2000, vamos dar 30% de desconto. Isso é, vamos pegar o preço do livro e multiplicar por 0.7. “livro.preco*=0.7”.

[05:34] Quando rodamos o programa ele fala: o método preço não existe. Lembram que tínhamos controlado o escopo, nós fechamos o preço? Então vamos disponibilizar de novo: vamos colocar, não só o reader, mas também o writer, o método preço igual.

[05:51] Colocamos, rodamos de novo o programa e o resultado é 100. Então ele não aplicou a promoção de 30%? Reparem, se mudarmos a ordem das invocações e chamar o nosso programa de novo, ele imprime com 30% de desconto.

[06:09] A ordem das coisas que eu faço está afetando o meu preço, sendo que o livro deveria ter só um preço, independente da ordem que eu chamo.

[06:18] Ao invés de forçar uma chamada de “aplica_promocoes” em determinado momento, vamos aplicar as promoções quando você cria o livro. Logo de cara, essa promoção é aplicada, para que o desenvolvedor não possa esquecer de chamar esse método.

[06:35] Então se colocarmos esse método no construtor, o código de mudar o preço dentro do “intialize”, forçamos: toda a vez que você cria um livro, colocamos o preço correto.

[06:44] Se o ano de lançamento for menor do que 2000, multiplicamos por 0.7. Roda o programa, perfeito: 70.

[06:54] Repara que esse problema de ter que lembrar de chamar o método de promoções só aconteceu porque tínhamos quebrado o encapsulamento, liberado o método preço igual, para qualquer um acessar de qualquer ponto da minha aplicado.

[07:08] Agora que nós protegemos de novo, vamos tirar fora esse accessor e colocar só o reader. Pronto, está encapsulado e controlado. Podemos refatorar também. Podemos tirar e colocar de uma maneira mais comum, o preço é *= 0.7 <2000.

[07:27] Em mais uma refatoração, nós extraímos essa lógica do preço para um método chamado “calcula_preco”. Passo o preço que foi passado no construtor, e nesse método nós fazemos a verificação.

[07:38] Se o ano for menor do que 2000, é ele *0.7. Caso o contrário, é o próprio preço. Lembrar do “@” porque é uma variável membro. “@ano_lancamento < 2000”. Vamos executar o programa, só para conferir que está tudo certo.

[07:55] O cálculo do preço não estava jogado em toda a aplicação, imagina se vários lugares dessa aplicação fizessem cálculos com esse preço e mudassem o valor da variável de preço do livro?

[08:03] Se não tivéssemos controlado o escopo da varável preço do nosso livro, iamos correr atrás de cada um dos pontos que faziam alteração, pra ver se influenciava ou não a nova promoção.

[08:14] No momento que agrupamos os valores da classe livro, junto com o comportamento da classe livro, sabemos se precisa mudar alguma coisa relacionada ao livro, é só ir na classe livro e mudar lá.

[08:27] É o único ponto da aplicação. O código fica espalhado, está centralizado ao redor dos dados que ele pertence.