01

Responsabilidades e coesão dos objetos

Transcrição

[00:00] Voltando ao nosso código, imagina que precisamos agora vender os nossos livros, vender os livros que estão em estoque. Então vamos apagar esse pedaço de código que não vamos utilizar, adiciona aqueles nossos quatro livros que conhecíamos e vamos vender.

[00:17] Isso é, vamos pegar os livros do nosso estoque e remover. “livros.delete Algoritmos”. Mas o método livros não existe, então vamos criar um atributo reader, do livro: “attr_reader:livros”.

[00:37] Já que eu liberei o acesso a variável livros, eu vou acessar aqui também a variável livros e usar a setinha bonitinha pra adicionar na minha array.

[00:47] Posso usar também, com concatenação, adiciono um e depois adiciono outro livro, vou extrair essa variável, com uma variável chamada programmer, e o livro de “ruby” que também vou extrair para uma outra variável. Fica mais fácil de trabalhar, de novo livros e setinha.

[00:58] Então rodamos agora a nossa aplicação, “ruby livro.rb”, e ele não imprime nada, está tudo funcionando. Queremos imprimir o estoque de livros que foi necessário até agora. Qual é o tamanho máximo do meu estoque até agora?

[01:17] Quantos livros eu precisei no máximo, pra manter o meu estoque em algum lugar do mundo? Se eu coloco dois livros no meu estoque, e tiro um livro, e depois coloco mais três livros, então nesse instante eu preciso de quatro livros no meu estoque. Um estoque que caiba quatro livros.

[01:37] Se depois eu vendo dois livros de novo, apesar de ter dois livros no meu estoque, eu precisei de um estoque tamanho quatro, pra caber os quatro livros quando tinha os quatro.

[01:47] Então é nesse número que eu estou interessado: Qual é o tamanho do estoque que eu precisei até hoje? Repara que livros é uma “Array”, então o que eu poderia fazer é pegar a classe “array” do “ruby” e definir um atributo chamado “maximo_necessario”.

[02:06] Toda a vez que eu adicionar um objeto, um livro nessa array, eu vou adicionar o livro no método “push”, que é o método de adicionar objetos na array, e vou verificar.

[02:20] Se, eu ainda não defini o máximo de objetos necessários, ou se o máximo até agora é menor do que o tamanho atual, o novo máximo é o tamanho atual.

[02:33] Toda a vez que eu coloco um novo livro, eu verifico. Seu eu não tinha um máximo até então, ou se o máximo até então é menor que a quantidade de livros atuais, o máximo de livros que eu preciso é o tamanho atual.

[02:44] Se eu estou estourando o tamanho máximo do meu estoque, armazeno esse número novo. Vamos rodar a nossa aplicação, ele imprimiu um. Mas, deu um problema logo depois.

[02:59] Repara, vamos na linha 89 e ele fala que o método de duas setinhas não existe. Mas como assim, não existe? Quando eu chamava esse método em uma “array”, eu podia chamar ele logo depois de novo. Por quê?

[03:17] Porque o método de duas setinhas de “push” em uma “Array”, devolve a própria “Array. Mas no nosso caso, não. Reescrevemos esse método e abrimos a classe “Array” de uma maneira que quebrava compatibilidade com a classe

"Array".

[03:29] Quebramos o código da classe "Array" em todo o nosso sistema. Ao reabrir a classe "array", eu preciso ter certeza que todo o método que eu mexo, eu mantenha a compatibilidade. Então no método de adicionar objetos, eu preciso devolver eu mesmo: self na última linha desse código. Rodo novamente a nossa aplicação, e agora funciona.

[03:53] Vamos adicionar a mesma linha de impressão, pra depois adicionar o livro de arquitetura e mais um livro. Três; vamos adicionar o livro de ruby. Quatro: e vamos adicionar a mesma impressão depois remover, depois de vender um livro.

[04:07] Quatro de novo. Apesar de ter três livros no estoque, um dia eu precisei de espaço para quatro livros. E foi esse número que ele imprimiu pra mim. Então repara que sempre que eu abro uma classe, eu tenho a responsabilidade de manter a compatibilidade com tudo que ela fazia anteriormente.

[04:24] E lembrando que as "Array" só vão funcionar dessa maneira depois do interpretador do ruby passar por essa linha. Enquanto ele não passar por essa linha de código, nenhuma array vai ter esse tipo de funcionamento. Será que é isso que queremos para a nossa aplicação inteira?

[04:44] Sempre temos que tomar muito cuidado quando reabrimos uma classe. E se eu quisesse, ao invés de adicionar esse método, esse comportamento, na classe array, adicionar ele só nessa array de livros? Só nos objetos que forem referenciados através da variável livros.

[05:02] Eu poderia definir esse método, ao invés de definir ele na classe aberta, eu podia definir nos meus livros: "def@.livros." o nome do método. Então eu estou definindo esse método somente nesses livros, e não em todas as arrays do meu sistema. A minha array de livros possui esse método dessa maneira.

[05:24] E o nosso atributo? Vamos definir também: "def@livros.maximo_necessario". Devolve o valor de máximo necessário. Então eu estou definindo esses dois métodos só pela nossa array de livros, nenhuma outra array do meu sistema.

[05:41] Rodo a minha aplicação novamente, e está tudo funcionando. Isto é, os meus objetos estavam abertos pra alterar o comportamento deles, de uma maneira dinâmica.

[05:50] Mas repara que esse código é muito feio, seria quase impossível de testar, muito nojento, vamos extrair ele e colocar em outro lugar. Vamos colocar esse código em um módulo.

[06:03] Lembra? Da definição de módulo em ruby? Módulo contador, que conta o máximo necessário, e ele tem um método de push tradicional, e um método de máximo necessário. O módulo define esses dois métodos que estamos interessados.

[06:22] E o que queremos fazer é: que o objeto referenciado pela variável livros, estenda esse módulo: "@livros.extend Contador". Voltamos para o terminal, executamos o programa, e está tudo certo. O módulo foi incluído nos nossos livros dinamicamente, em tempos de execução.

[06:36] Aquele método do máximo necessário, podia ser um "attr_reader", não tem problema. Em mais um exemplo, eu posso criar uma array de números, isto é, "numbers = []", mandar estender de contador, "numbers.extend Contador", adicionar alguns números, por exemplo: 13, 15, 17, 19, e imprimir o máximo necessário.

[07:02] Como incluímos o contador, tudo funciona. E se eu tirar o contador completamente? Ele fala que o método não existe. Mas nós já vimos no capítulo anterior, que não é bom ficar expondo todas as nossas variáveis membro, todas as nossas internals.

[07:24] Queremos mandar o objeto fazer alguma coisa, e não ficar pedindo dados dele. Então ao invés de pedir os livros do estoque pra adicionar algo, simplesmente: “Estoque << algoritmos”, “Estoque << arquitetura << programmer”, “Estoque <<ruby”, “Estoque.delete algoritmos”.

[07:46] E por fim, “estoque.maximo_necessario”. Sempre mandando o estoque fazer alguma coisa. Sempre mandando fazer, ao invés de pedir. O próximo passo é definir esses métodos.

[07:53] Vamos definir o método “delete”, que recebe um livro e delega essa chamada para o @livros.delete, para a nossa array. O método máximo_necessario faz a mesma coisa: delega a chamada para @livros.maximo_necessario.

[08:13] Por fim, repare o nome do método adicione e delete tem algo estranho. Se eu adiciono eu removo, então adiciona, vamos colocar aquela setinha mais padrão, e lembrar que o estoque devolve ele mesmo.

[08:31] Além disso, o nome do método “delete”, vira “remove”. Vamos rodar aplicação de novo, e pronto, tudo funcionando. Recapitulando, nós vimos que é possível reabrir uma classe qualquer no ruby, e adicionar alterar comportamento dela.

[08:47] E o perigo disso também: é importantíssimo manter a compatibilidade, não quebrar o que já existe. E isso pode ser muito delicado em uma aplicação grande.

[08:56] Depois vimos também, que é possível alterar o comportamento de um único objeto. Pegar um objeto e falar, olha: adiciona um método, altera o comportamento de um método seu. E isso também é possível de fazer, através de uma composição de módulos.

[09:13] Eu pego um módulo e incluo ele, finalizando nós retomamos aquela ideia de que evitamos liberar acesso aos valores internos, ao funcionamento da nossa classe, do nosso objeto, mas sim, mandamos objeto fazer alguma coisa: “Tell don’t Ask.”