02

Responsabilidades e coesão dos objetos

Nossa empresa precisa agora listar o número máximo de livros que já estiveram em estoque para prever um futuro aumento no tamanho de nosso depósito. Por exemplo, se entram 10 livros, saem 2 (total: 8), entram 4 (total: 12) e saem 5 (total 7), o tamanho máximo necessário foi 12.

Imagine que vendemos o livro de algoritmos que criamos no capítulo anterior, ou seja, deletamos de nosso estoque. Podemos usar o método `delete` do array:

```
estoque.livros.delete algoritmos
```

Mas o método `livros` não existe, portanto precisaremos criar o atributo `attr_reader` de livros na classe `Estoque` :

```
class Estoque
   attr_reader :livros
   #outros métodos da classe
end
```

Então, uma vez que liberamos o acesso à variável `livros` , podemos acessá-la e utilizar o método `<<` da classe `Array` para adicionar os livros ao nosso estoque:

```
algoritmos = Livro.new("Algoritmos", 100, 1998, true)
arquitetura = Livro.new("Introdução À Arquitetura e Design de Software", 70, 2011, true)
programmer = Livro.new("The Pragmatic Programmer", 100, 1999, true)
ruby = Livro.new("Programming Ruby", 100, 2004, true)

estoque = Estoque.new
estoque.livros <<  algoritmos
estoque.livros <<  arquitetura
estoque.livros <<  programmer << ruby
```

Agora que podemos adicionar e remover livros de nosso estoque, precisamos saber o número máximo de livros que já estiveram em estoque, ou seja, qual o tamanho máximo de meu estoque até o momento. Então queremos imprimir este valor da seguinte forma:

```
estoque = Estoque.new
estoque.livros <<  algoritmos
puts estoque.livros.maximo_necessario # imprimir 1
estoque.livros <<  arquitetura
puts estoque.livros.maximo_necessario # imprimir 2
estoque.livros <<  programmer << ruby
puts estoque.livros.maximo_necessario # imprimir 4

estoque.livros.delete algoritmos
puts estoque.livros.maximo_necessario # imprimir 4
```

Note que, para adicionar e remover um item do estoque, usamos os métodos tradicionais de `Array`. Seria possível redefinir o método `<<` de um Array? Algo como recriar a classe `Array` e sobrescrever seu método:

```
class Array
  attr_reader :maximo_necessario
  def << (livro)
    push (livro)
    if @maximo_necessario.nil? || @maximo_necessario < size
      @maximo_necessario = size
    end
  end
end
```

Dessa forma, toda vez que adicionamos um novo livro, verificamos se ainda não existe um `maximo_necessario` ou se o `maximo_necessario` é menor que a quantidade de livros atual. Se estivermos em uma dessas situações, o `maximo_necessario` passa a ser o tamanho atual. Ou seja, toda vez que excedemos o `maximo_necessario`, este novo número passa a ser nosso `maximo_necessario`.

Note que, como Ruby permite a reabertura das classes (open classes), o código acima é aceito pelo interpretador e, ao rodarmos nosso programa, obtemos o resultado que desejávamos. Ou melhor, parcialmente. Ele imprime 1, mas devolve um erro na terceira tentativa de adicionar elementos no array.

```
livro.rb:89: ln '<<': can't convert Livro into Integer (TypeError)
```

O que aconteceu? Ele indica que o método `<<` não existe. Isso ocorreu pois quebramos o comportamento do método `<<` da classe `Array` ao sobrescrevê-lo. O método original retornava o próprio array e nós não fazemos isso, mas sim retornamos o `maximo_necessario` (última linha do método).

Ao reabrir a classe `Array`, precisamos ter certeza de que todo método alterado mantém a compatibilidade com o método original. Portanto, no método `<<`, precisamos devolver `self` para retornar o próprio array:

```
class Array
  attr_reader :maximo_necessario
  def << (livro)
    push (livro)
    if @maximo_necessario.nil? || @maximo_necessario < size
      @maximo_necessario = size
    end
    self
  end
end
```

Agora sim o resultado é o esperado.

```
1
2
4
4
```

Note que, ao abrir uma classe, estamos nos acoplando fortemente a tudo o que ela contém, conteve e conterá. Um desenvolvedor pode usar nosso código numa versão antiga ou futura do Ruby e nossas mudanças têm que manter compatibilidade com todas elas, o que é muito difícil.

Portanto, é importante estarmos atentos sempre que abrimos alguma classe e alteramos seu comportamento. Devemos sempre prestar atenção no retorno dos métodos, uma vez que podemos introduzir bugs no sistema. Em nosso exemplo, quebramos o contrato do método `<<` .

Outro ponto é que devemos pensar na classe como um todo. Se o desenvolvedor invocar o método push em vez de `<<` , o resultado será `0` e nosso código não funcionará como esperávamos! Isso pois, ao abrir uma classe, temos que conhecer exatamente como ela funciona por completo.

Vemos aqui a clara importância dos testes: para garantir que o contrato não seja quebrado, toda vez que um classe é aberta, temos que rodar todos os testes existentes na classe que alteramos. Essa é uma prática extremamente incomum, porém necessária para o bom funcionamento do código. Toda vez que uma classe é reaberta, um projeto que se preocupa com testes deve executar os testes originais nela para garantir compatibilidade.

Além de todo esse acoplamento, estamos alterando uma classe que todo o sistema utiliza e podemos não saber como ela é utilizada em outros pontos. Seria possível alterar somente o nosso array de livros? Algo como definir o método e o atributo somente no caso do array que é criado dentro de nosso estoque? Para isso, poderíamos fazer algo como:

```
def @livros.<< (livro)
  push (livro)
  if @maximo_necessario.nil? || @maximo_necessario < size
    @maximo_necessario = size
  end
  self
end

def @livros.maximo_necessario
  @maximo_necessario
end
```

E pronto. Somente o nosso array de livros possui esses dois métodos e afetamos de uma maneira mais leve o sistema: mudamos somente aqueles arrays que fazem sentido.

Mas o código fica ainda mais confuso do que antes. Poderíamos isolar esse código em um módulo, que só define parte do comportamento de um objeto:

```
module Contador
  def << (livro)
    push (livro)
    if @maximo_necessario.nil? || @maximo_necessario < size
      @maximo_necessario = size
    end
    self
  end

  def maximo_necessario
    @maximo_necessario
  end
end
```

E então podemos simplesmente dizer ao objeto que criamos que queremos que ele estenda desse módulo:

```
class Estoque
  attr_reader :livros

  def initialize
    @livros = []
    @livros.extend Contador
  end

  # ...
end
```

Note que o método `maximo_necessario` poderia também ser um atributo `attr_reader` :

```
module Contador
  def << (livro)
    push (livro)
    if @maximo_necessario.nil? || @maximo_necessario < size
      @maximo_necessario = size
    end
    self
  end

  attr_reader :maximo_necessario
end
```

Agora tudo está funcionando, mas ainda existe algo que precisa ser melhorado. Conforme vimos nos capítulos anteriores no conceito de 'Tell don't Ask', não devemos ficar expondo nossas variáveis membro. O que queremos é dizer ao objeto para fazer alguma coisa, não ficar pedindo dados dele. Então, em vez de pedir aos livros do estoque para que adicione algo (ex: `estoque.livros << algoritmos`), simplesmente queremos que o estoque adicione algo (ex: `estoque << algoritmos`). Sempre mandando o estoque fazer alguma coisa em vez de pedir.

Dessa forma, nosso código deve ficar:

```
estoque = Estoque.new
estoque <<  algoritmos
puts estoque.maximo_necessario
estoque <<  arquitetura
puts estoque.maximo_necessario
estoque <<  programmer << ruby
puts estoque.maximo_necessario

estoque.delete algoritmos
puts estoque.maximo_necessario
```

Portanto para que isso funcione precisamos definir estes métodos:

```
class Estoque
  attr_reader :livros
```

```
  # outros métodos já existentes na classe

  def << (livro)
    @livros << livro if livro
    self
  end

  def remove(livro)
    @livros.delete livro
  end

  def maximo_necessario
    @livros.maximo_necessario
  end
end
```

Observe que refatoramos o método `delete` para `remove`, portanto sua chamada deve ser alterada também:

```
estoque.remove algoritmos
```

Agora, executando nossa aplicação novamente, tudo está funcionando.

Note que devemos usufruir das open classes e da alteração dinâmica da estrutura de objetos de forma cautelosa. Com cuidado, podemos obter o mesmo resultado evitando o alto acoplamento com a criação de novas classes. É importante analisar eventuais consequências que o uso das open classes pode trazer e lembrar sempre de rodar todos os testes que envolvem a classe que alteramos, bem como os testes já existentes em nosso sistema.