02

Composição, herança e os cuidados com mixins

Extraindo comportamento comum no construtor

Dando uma olhada no construtor da classe `Livro`, vemos que ele ainda faz muita coisa: ele atribui valores a diversas variáveis membro. Esse construtor está muito parecido nos outros produtos, então vamos tentar extraí-lo para o nosso módulo `Produto`.

```
module Produto

  def initialize(titulo, preco, ano_lancamento, possui_reimpressao, possui_sobrecapa, editora)
     # atribui variaveis
  end

  # metodos
end
```

No entanto, nem todas essas variáveis fazem sentido para todos os produtos. As variáveis `possui_reimpressao` e `possui_sobrecapa` são exclusivas do produto Livro. Vamos removê-las do construtor do módulo `Produto`:

```
module Produto

  def initialize(titulo, preco, ano_lancamento, editora)
     # atribui variaveis
  end

  # metodos
end
```

e deixá-las apenas no construtor da classe `Livro`. Nesse construtor, ainda recebemos todas as variáveis, mas fazemos apenas a atribuição das variáveis membro exclusivas dessa classe; a atribuição das outras, deixamos a cargo do construtor do módulo `Produto`. Para chamá-lo, usamos a palavra-chave `super`:

```
class Livro
  include Produto

  def initialize(titulo, preco, ano_lancamento, possui_reimpressao, possui_sobrecapa, editora)
    super(titulo, preco, ano_lancamento, editora)
    @possui_reimpressao = possui_reimpressao
    @possui_sobrecapa = possui_sobrecapa
  end

  # metodos
end
```

Podemos fazer a mesma coisa na classe `Revista`. Mantemos apenas a atribuição das variáveis membro `possui_reimpressao` e `numero`, delegando o resto para o construtor de `Produto`:

```
class Revista
  include Produto

  def initialize(titulo, preco, ano_lancamento, possui_reimpressao, numero, editora)
    super(titulo, preco, ano_lancamento, editora)
    @possui_reimpressao = possui_reimpressao
    @numero = numero
  end

  # metodos
end
```

Por fim, fazemos o mesmo na classe `EBook`. Nela, não há nenhuma variável membro exclusiva dela, então podemos delegar a chamada diretamente ao construtor de `Produto`:

```
class EBook
  include Produto

  def initialize(titulo, preco, ano_lancamento, editora)
    super(titulo, preco, ano_lancamento, editora)
  end

  # metodos
end
```

Nesse caso, como estamos simplesmente chamando o construtor de `Produto`, podemos simplesmente remover o construtor da classe `EBook`, deixando apenas o construtor que `Produto` coloca para nós:

```
class EBook
  include Produto

  # o construtor vem de Produto

  # metodos
end
```

Extraindo mais comportamento comum

Olhando para nossas classes `Livro` e `Revista`, vemos que ainda existe código repetido entre elas: o método `possui_reimpressao?`, pois ambas as classes representam produtos que são impressos, além de ambas receberem um parâmetro `possui_reimpressao` no construtor. Vamos, então, tentar extrair esse comportamento comum para um novo módulo, chamado `Impresso`:

```
modulo Impresso
  def initialize(possui_reimpressao)
    @possui_reimpressao = possui_reimpressao
  end

  def possui_reimpressao?
    @possui_reimpressao
```

```
  end
end
```

Agora, na classe `Livro`, basta incluir esse módulo e chamar o construtor dele. Mas veja só: como o Ruby vai saber qual construtor queremos chamar quando usamos `super` nesse caso?

```
class Livro
  include Produto
  include Impresso

  def initialize(titulo, preco, ano_lancamento, possui_reimpressao, possui_sobrecapa, editora)
    super(titulo, preco, ano_lancamento, editora) # de quem é esse construtor?
    super(possui_reimpressao) # de quem é esse construtor?
    @possui_sobrecapa = possui_sobrecapa
  end

  # metodos
end
```

Na verdade, é possível saber quem é o `super` nesse caso. O Ruby define que o `super` é sempre relativo ao último módulo que incluímos. Ou seja, no nosso caso, o `super` se refere ao construtor do módulo `Impresso`.

Precisamos corrigir nosso código, então, já que não temos acesso ao construtor de `Produto` na classe `Livro`. Mas todo `Impresso` também é um `Produto`, então podemos fazer o construtor do módulo `Impresso` receber todos os parâmetros do `Produto` além do parâmetro que ele já recebia (`possui_reimpressao`):

```
module Impresso

  def initialize(titulo, preco, ano_lancamento, editora, possui_reimpressao)
    super(titulo, preco, ano_lancamento, editora) # aqui o super vai ser o construtor de Produto
    @possui_reimpressao = possui_reimpressao
  end

  # metodos
end
```

E, agora, na classe `Livro`, chamamos o construtor de `Impresso`:

```
class Livro
  include Produto
  include Impresso

  def initialize(titulo, preco, ano_lancamento, possui_reimpressao, possui_sobrecapa, editora)
    super(titulo, preco, ano_lancamento, editora, possui_reimpressao) # construtor de Impresso
    @possui_sobrecapa = possui_sobrecapa
  end

  # metodos
end
```

Pronto, agora a classe `Livro` funciona. Mas veja que, se invertermos as linhas `include Produto` e `include Impresso`, nosso código para de funcionar! Isso porque o módulo `Impresso` está altamente acoplado ao módulo `Produto`!

Estamos incluindo o módulo `Produto` na classe `Livro` para indicar que um livro é um produto. E estamos incluindo também o módulo `Impresso` para indicar que um livro se comporta como um produto impresso. Veja que são dois casos diferentes!

Como queremos indicar que um livro é um produto, devemos usar herança, e não composição. Sempre que temos uma classe que é uma especialização de outra (um livro é um produto específico), a herança é uma solução mais adequada que a composição de módulos.

Então vamos alterar nosso código, transformando `Produto` numa classe e fazendo `Livro`, `Revista` e `EBook` estenderem ela. Além disso, vamos remover o construtor do módulo `Impresso`.

```
class Produto
  def initialize(titulo, preco, ano_lancamento, editora)
     # atribui variaveis
  end

  # metodos
end



module Impresso
  def possui_reimpressao?
    @possui_reimpressao
  end
end



class Livro < Produto
  include Impresso

  def initialize(titulo, preco, ano_lancamento, possui_reimpressao, possui_sobrecapa, editora)
    super(titulo, preco, ano_lancamento, editora) # construtor de Produto
    @possui_reimpressao = possui_reimpressao
    @possui_sobrecapa = possui_sobrecapa
  end

  # metodos
end
```

```
class Revista < Produto
  include Impresso

  def initialize(titulo, preco, ano_lancamento, possui_reimpressao, numero, editora)
    super(titulo, preco, ano_lancamento, editora) # construtor de Produto
    @possui_reimpressao = possui_reimpressao
    @numero = numero
  end
```

```
  # metodos
end


class EBook < Produto
  # o construtor vem de Produto

  # metodos
end
```

Veja que construtores nos módulos podem ser perigosos. Quando colocamos um construtor num módulo, assumimos muito sobre a forma como esse módulo será utilizado. No entanto, na maioria das vezes, não temos como saber como esse módulo será usado e quais módulos serão combinados com ele. Dependendo do modo como os módulos são incluídos numa classe, o comportamento final pode ser bem estranho! Ao usar herança para definir um construtor comum a diversas classes, temos mais controle sobre essa combinação.

Um outro perigo que deve ser evitado ao usarmos módulos é escrever módulos que alteram o estado do classe na qual são incluídos. No nosso caso, os construtores dos módulos alteravam o valor de variáveis membro das outras classes. Novamente, isso é perigoso pois não controlamos a forma como os módulos são combinados. Por isso, deve-se evitar ao máximo alterar variáveis membro de uma classe a partir de um módulo. O ideal é que um módulo acesse somente métodos públicos da classe na qual ele vai ser incluído.