02

Generalizando comportamentos através do method_missing

Observe os métodos `livro_que_mais_vendeu_por_titulo` , `livro_que_mais_vendeu_por_ano` e `livro_que_mais_vendeu_por_editora` são muito parecidos, tendo apenas o campo como diferença. Podemos então refatorar esses nossos métodos criando um unico método nomeado `livro_que_mais_vendeu_por (campo)` , dessa forma ao invés de passarmos o titulo, ano ou editora, passaremos unicamente o campo que gostaríamos de consultar:

```
def livro_que_mais_vendeu_por(&campo)
   livro_que_mais_vendeu_por("livro", &campo)
end
```

Podemos fazer o mesmo para o método `revista_que_mais_vendeu_por` :

```
def revista_que_mais_vendeu_por(&campo)
   revista_que_mais_vendeu_por("revista", &campo)
end
```

Dessa forma, precisamos alterar nosso arquivo `sistema.rb` , para passarmos o campo como parâmetro no momento em que invocamos esses métodos que refatoramos:

```
puts estoque.livro_que_mais_vendeu_por(&:titulo).titulo
puts estoque.revista_que_mais_vendeu_por(&:titulo).titulo
```

Ao executarmos a nossa aplicação, tudo continua funcionando. Obtemos como resposta:

Algoritmos

Revista de Ruby

Já melhoramos nosso código, mas note que os métodos `livro_que_mais_vendeu_por` e `revista_que_mais_vendeu_por` ainda são muito parecidos. Imagine que agora queremos que nossa livraria passe a vender também uma versão eletrônica de nossos livros. Vamos criar o livro `online_arquitetura` , adicioná-lo em nosso estoque e imprimir o ranking de ebooks mais vendidos por titulo, assim como estamos fazendo com nossos livros e revistas:

```
algoritmos = Livro.new("Algoritmos", 100, 1998, true, "editora", "livro")
arquitetura = Livro.new("Introdução À Arquitetura e Design de Software", 70, 2011, true, "editor
programmer = Livro.new("The Pragmatic Programmer", 100, 1999, true, "editora", "livro")
ruby = Livro.new("Programming Ruby", 100, 2004, true, "editora", "livro")
revistona = Livro.new("Revista de Ruby", 10, 2012, true, "Revistas", "revista")
online_arquitetura = Livro.new("Introdução a Arquitetura e Design de Software", 50, 2012, true,

estoque = Estoque.new
estoque << algoritmos << algoritmos << ruby << programmer << arquitetura << ruby << ruby << revi
estoque.vende ruby
estoque.vende algoritmos
```

```
estoque.vende algoritmos
estoque.vende revistona
estoque.vende online_arquitetura

puts estoque.livro_que_mais_vendeu_por(&:titulo).titulo
puts estoque.revista_que_mais_vendeu_por(&:titulo).titulo
puts estoque.ebook_que_mais_vendeu_por(&:titulo).titulo
```

Obteremos como resposta o seguinte erro:

```
sistema.rb:30: undefined method 'ebook_que_mais_vendeu_por' for#<Estoque:0xb740704c>(NoMethodEr
```

Isso ocorreu, pois não criamos o método `ebook_que_mais_vendeu_por`. Quando o método não existe em nosso objeto, o Ruby chama um método chamado `method_missing` e nos passa o nome desse método. Então vamos imprimir neste caso o nome do método que ele está nos passando. Em nossa classe Estoque, sobrescrevemos o método `method_missing`:

```
class Estoque
attr_reader :livros

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

   def method_missing(name)
      puts "Nao encontrei: #{name}"
   end

# demais métodos
```

Agora, quando executamos nossa aplicação novamente, obtemos o seguinte erro como resposta:

```
Nao encontrei: ebook_que_mais_vendeu_por
sistema.rb:30: undefined method `titulo' for nil:NilClass (NoMethodError)
```

Como o Ruby não encontrou o método `ebook_que_mais_vendeu_por` ele invocou o método `method_missing`. Dessa forma, temos oportunidade de implementar nesse método uma verificação, para que caso ele tenha um padrão determinado, extraimos seu tipo e campo e fazemos a invocação. Caso contrário delega ao método pai (`object`).

O primeiro passo é transformar o simbolo (`name`) em uma string com o método `to_s` e verificar se ele é uma chamada ao método `que_mais_vendeu_por` titulo, editora ou ano de lançamento. Então podemos fazer algo como:

```
   def method_missing(name)
      matcher = name.to_s.match "(.+)_que_mais_vendeu_por_(.+)"

      puts "Nao encontrei: #{name}"
   end
```

Dessa forma, com o uso de `(.+)` estamos dizendo que queremos qualquer coisa que mais vendeu por qualquer campo. E podemos agora fazer um `if` para verificar se este método que está sendo invocado é deste tipo, caso seja faremos algo, pelo contrario simplesmente delegamos a chamada ao método pai:

```
def method_missing(name)
   matcher = name.to_s.match "(.+)_que_mais_vendeu_por_(.+)"

   if matcher

   else
      super
   end
end
```

Agora dentro de nossa lógica, caso seja um método do tipo `que_mais_vendeu_por` precisamos saber qual o seu tipo (se é uma revista, livro ou ebook) e também seu campo (titulo, ano_lancamento, preço) e chamamos o método `que_mais_vendeu_por` passando o título e campo recebido:

```
def method_missing(name)
   matcher = name.to_s.match "(.+)_que_mais_vendeu_por_(.+)"

   if matcher
      tipo = matcher[1]
      campo = matcher[2].to_sym  #pois precisamos converter para simbolo
      que_mais_vendeu_por(tipo, &campo)
   else
      super
   end
end
```

Podemos apagar os métodos `livro_que_mais_vendeu_por` e `revista_que_mais_vendeu_por` e mudar nossas invocações no arquivo `sistema.rb` para deixarmos de passar o campo como parâmetro e simplesmente chamarmos o método:

```
estoque = Estoque.new

estoque << algoritmos << algoritmos << ruby << programmer << arquitetura
<< ruby << ruby << revistona << revistona << online_arquitetura

estoque.vende ruby
estoque.vende algoritmos
estoque.vende algoritmos
estoque.vende revistona
estoque.vende online_arquitetura

puts estoque.livro_que_mais_vendeu_por_titulo.titulo
puts estoque.revista_que_mais_vendeu_por_titulo.titulo
puts estoque.ebook_que_mais_vendeu_por_titulo.titulo
```

Aos executarmos nosso programa novamente, tudo está funcionando. Obtemos como saída:

```
Algoritmos
```

Revista de Ruby

Introdução a Arquitetura e Design de Software

Observe agora que os métodos `que_mais_vendeu_por` e `quantidade_de_vendas_por` não são chamados de fora de nossa classe, então vamos passa-los pra um escopo privado, para que não sejam invocados por outras classes:

```
private
   def quantidade_de_vendas_por(produto, &campo)
    @vendas.count { |venda| campo.call(venda) == campo.call(produto) }
   end

   def que_mais_vendeu_por(tipo, &campo)
  @vendas.select{ | l |  l.tipo == tipo}.sort {|v1,v2| quantidade_de_vendas_por(v1, &campo) <=>
   end
```

Executamos a aplicação e tudo continua funcionando.

Algoritmos

Revista de Ruby

Introdução a Arquitetura e Design de Software

Observe que todo objeto também tem um método pra saber se ele responde ou não pra uma mensagem, esse método se chama `respond_to?` , como por exemplo:

```
puts estoque.respond_to?(:ebook_que_mais_vendeu_por_titulo)
```

Ao executar nosso código, note que ele responde como `false` , como se ele não fosse capaz de responder por este método.

Algoritmos

Revista de Ruby

Introdução a Arquitetura e Design de Software

false

Isso ocorre porque toda vez que implementamos o `method_missing` para mudar o comportamento de métodos, precisamos sobrescrever também o método `respond_to?` . Então, em nosso caso, precisamos verificar se a chamada de método for equivalente ao método que esperamos ou se nossa classe pai responde por este método, e então devolvemos o nome do método.

```
def respond_to?(name)
   name.to_s.match ("(.+)_que_mais_vendeu_por_(.+)") || super
end
```

Executamos nosso código novamente, e obtemos a resposta esperada:

> Algoritmos
>
> Revista de Ruby
>
> Introdução a Arquitetura e Design de Software
>
> ebook_que_mais_vendeu_por_titulo

Note que podemos sobrescrever o método `method_missing` sempre que queremos agir de forma diferente de acordo com o nome dos nossos métodos, mas não sabemos o nome de todos eles, ou queremos suportar infinitos nomes de métodos possíveis, como fizemos no caso do nosso exemplo `que_mais_vendeu_por` . Lembrando que precisamos também sobrescever o método `respond_to` sempre que sobrescrevermos o `method_missing` .