02

Enxugando seu código

Observe que todas as nossas classes estão em um único arquivo. Vamos começar separando nossos arquivos. Em nosso arquivo `livro.rb` mantemos a nossa classe livro, e criamos o arquivo `sistema.rb` inserindo o restante de nosso código. Precisamos adicionar ao arquivo `sistema.rb` o arquivo `livro.rb`, portanto inserimos `require "livro.rb"` no início do código, porém, ao executarmos a nossa aplicação recebemos o erro:

```
/home/guilherme/.rvm/rubies/ruby-1.9.3-p125/lib/ruby/site_ruby/1.9.1/rubygems/custom_require.rb
```

Ele não encontrou nosso arquivo. Isso ocorreu pois estamos utilizando a versão 1.9 do Ruby, e como a partir da versão 1.8 ele não faz mais a procura a partir do diretório fonte, precisamos dizer exatamente onde está o arquivo livro, ou setar um path de procura com `require_relative "livro"`.

```
require_relative "livro"

def livro_para_newsletter(livro)
     if livro.ano_lancamento < 2000
     puts "Newsletter/Liquidacao"
     puts livro.titulo
     puts livro.preco
     puts livro.possui_reimpressao?
     end
end

# restante do código
```

Ao executarmos nossa aplicação novamente vemos que tudo está funcionando.

Agora vamos extrair nosso contador. Inserimos `require_relative "contador"` no arquivo `sistema.rb` e copiamos todo o código do contador para o arquivo `contador.rb`.

Faremos o mesmo com nossa classe Estoque. Extraimos seu código para o arquivo `estoque.rb` e inserimos `require_relative "estoque"` no `sistema.rb`. Mas observe que dentro do arquivo estoque, ele precisa de uma referencia para o Contador, então seria arriscado alguém adicionar o estoque sem adicionar o contador, por isso dentro do arquivo `estoque.rb` inserimos um `require_relative "contador"`, e dessa forma podemos remover o `require_relative "contador"` do arquivo `sistema.rb`, pois nosso estoque vai requerer automaticamente tudo que ele precisa, que no nosso caso é o contador. Executamos o código novamente, e tudo continua funcionando.

Ficamos com o arquivo `sistema.rb` da seguinte forma:

```
require_relative "livro"
require_relative "estoque"

def livro_para_newsletter(livro)
     if livro.ano_lancamento < 2000
     puts "Newsletter/Liquidacao"
     puts livro.titulo
```

```
      puts livro.preco
      puts livro.possui_reimpressao?
      end
  end

  # ...
```

Temos também nosso arquivo `estoque.rb` :

```
  require_relative "contador"

  class Estoque
  attr_reader :livros

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

E por fim nosso arquivo `livro.rb` :

```
  class Livro
  attr_reader :titulo, :preco, :ano_lancamento

    def initialize(titulo, preco, ano_lancamento, possui_reimpressao)
      @titulo = titulo
      @ano_lancamento = ano_lancamento
      @preco = calcula_preco(preco)
      @possui_reimpressao = possui_reimpressao
    end

    def to_csv
     "#{@titulo},#{@ano_lancamento},#{@preco}"
    end

    def possui_reimpressao?
      @possui_reimpressao
    end

  # ...
```

Nosso sistema precisa suportar também quem é a editora de um livro, portanto adicionamos um `reader` para editora, e adicionamos também no construtor a editora, atribuindo o valor ao seu atributo :

```
  class Livro
  attr_reader :titulo, :preco, :ano_lancamento, :editora

    def initialize(titulo, preco, ano_lancamento, possui_reimpressao, editora)
      @titulo = titulo
      @ano_lancamento = ano_lancamento
```

```
    @preco = calcula_preco(preco)
    @possui_reimpressao = possui_reimpressao
    @editora = editora
  end

# continuação do código
}
```

Desejamos agora fazer um ranking em nosso sistema de acordo com vendas de livros, revistas e ebooks que temos em nosso estoque. Para isso, nosso método remove do arquivo `estoque.rb` é renomeado para vende, pois afinal estamos vendendo e não removendo o livro. E adicionamos este livro vendido a um array de vendas:

```
def vende(livro)
   @livros.delete livro
   @vendas << livro
end
```

Criamos essa array de vendas em nosso Initialize:

```
class Estoque
attr_reader :livros

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

# demais métodos de nossa classe
```

Vamos criar agora um método que calcule a quantidade de vendas de um título específico, recebendo o produto e contando quantas vendas têm o mesmo título desse produto, ou seja, quantas vezes ele foi vendido:

```
    def quantidade_de_vendas_de_titulo(produto)
     @vendas.count { |venda| venda.titulo == produto.titulo }
    end
```

Além disso, para saber qual foi o título de livro mais vendido, ordenamos nossas vendas comparando a quantidade de vendas de cada um dos livros:

```
    def livro_que_mais_vendeu_por_titulo
     @vendas.sort {|v1,v2| quantidade_de_vendas_de_titulo(v1) <=> quantidade_de_vendas_de_titulo
    end
```

Temos agora um ranking de quantidade de vendas por título!

Gostaríamos agora de rankear a quantidade de vendas por editora, portanto precisamos adicionar uma editora aos nossos livros já criados no arquivo `sistema.rb` :

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

Retiramos as impressões que usamos em nosso estoque nos exercicios anteriores, pois não precisaremos mais delas. Adicionamos alguns livros ao nosso estoque, vendemos alguns livros e imprimimos o livro que mais vendeu por título:

```
estoque = Estoque.new
estoque << algoritmos << algoritmos << ruby << programmer << arquitetura << ruby << ruby
estoque.vende ruby
estoque.vende ruby
estoque.vende ruby
estoque.vende algoritmos
estoque.vende algoritmos
puts estoque.livro_que_mais_vendeu_por_titulo.titulo
```

Ao executarmos nossa aplicação utilizando os exemplos acima teremos como retorno o livo de Ruby, pois foi nosso livro mais vendido por título:

```
Programming Ruby
```

Podemos testar também que se retirarmos a venda de dois dos livros de ruby, nossa aplicação devolverá como livro mais vendido o livro de algoritmos, pois passa a ser o livro mais vendido por titulo:

```
Algoritmos
```

Tudo está funcionando bem! Agora queremos saber qual foi o ano em que foram vendidos mais livros, portanto podemos criar o método `livro_que_mais_vendeu_por_ano` da seguinte forma:

```
def livro_que_mais_vendeu_por_ano
 @vendas.sort {|v1,v2| quantidade_de_vendas_por_ano(v1) <=> quantidade_de_vendas_por_ano(v2)
end
```

Mas note que a única diferença deste novo método para o método que devolve o livro mais vendido por título é a troca da expressão "titulo" por "ano". Logo, existe uma estrutura que pode ser reaproveitada `quantidade_de_vendas_por` . Dessa forma podemos extrair esse método:

```
def livro_que_mais_vendeu_por_ano
 @vendas.sort {|v1,v2| quantidade_de_vendas_por(v1, &:ano_lancamento) <=> quantidade_de_venda
end
```

E o método `livro_que_mais_vendeu_por_titulo` , pode usar a estrutura generalizada de quantidade de vendas como:

```
def livro_que_mais_vendeu_por_titulo
 @vendas.sort {|v1,v2| quantidade_de_vendas_por(v1, &:titulo) <=> quantidade_de_vendas_por(v
end
```

Agora, o método que efetivamente calcula a quantidade de vendas baseado num determinado campo, deve ser generalizado da seguinte forma:

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

Perceba que o campo passado como parâmetro para esse método é chamado com o método call, o que dá muita flexibilidade para o uso deste método.

Executamos o programa novamente e tudo continua funcionando.

Ainda usando a mesma estratégia, como queremos também ordenar os livros que mais venderam pela editora, criamos o método:

```
def livro_que_mais_vendeu_por_editora
 @vendas.sort {|v1,v2| quantidade_de_vendas_por(v1, &:editora) <=> quantidade_de_vendas_por(
end
```

Porém repare que este método é sempre igual. Fazemos o `@vendas.sort` alterando apenas o campo utilizado para a ordenação. Podemos então refatorar nosso código, criando o método `livro_que_mais_vendeu_por` que recebe como parâmetro a lambda do campo.

```
def livro_que_mais_vendeu_por(&campo)
 @vendas.sort {|v1,v2| quantidade_de_vendas_por(v1, &campo) <=> quantidade_de_vendas_por(v2, &
 end
```

E agora, alteramos nossos outros métodos para chamar `livro_que_mais_vendeu_por` :

```
def livro_que_mais_vendeu_por_titulo
   livro_que_mais_vendeu_por(&:titulo)
end

def livro_que_mais_vendeu_por_ano
   livro_que_mais_vendeu_por(&:ano_lancamento)
end

def livro_que_mais_vendeu_por_editora
   livro_que_mais_vendeu_por(&:editora)
end
```

Código bem mais enxuto. Executando novamente obtemos como retorno:

```
Algoritmos
```

Agora, além de livros, queremos vender ebooks e revistas, então criamos um campo dentro da classe livro pra representar que tipo de produto é esse que estamos vendendo.

```
class Livro
attr_reader :titulo, :preco, :ano_lancamento, :editora, :tipo

  def initialize(titulo, preco, ano_lancamento, possui_reimpressao, editora, tipo)
    @titulo = titulo
    @ano_lancamento = ano_lancamento
    @preco = calcula_preco(preco)
    @possui_reimpressao = possui_reimpressao
    @editora = editora
    @tipo = tipo
  end
# demais métodos do código
end
```

E da mesma forma que temos um ranking de livros, queremos ter um ranking de ebooks e de revistas. Poderiamos copiar todos os métodos já criados e alterar seus nomes, como por exemplo para revistas, fariamos da seguinte forma:

```
  def revista_que_mais_vendeu_por_titulo
     revista_que_mais_vendeu_por(&:titulo)
  end

  def revista_que_mais_vendeu_por_ano
     revista_que_mais_vendeu_por(&:ano_lancamento)
  end

  def revista_que_mais_vendeu_por_editora
     revista_que_mais_vendeu_por(&:editora)
  end
```

E no método `livro_que_mais_vendeu_por` filtramos as vendas do tipo livro, e o mesmo para revista, filtrando apenas as vendas do tipo revista:

```
  def livro_que_mais_vendeu_por(&campo)
   @vendas.select { | l | l.tipo == "livro"}.sort {|v1,v2| quantidade_de_vendas_por(v1, &campo)
  end

def revista_que_mais_vendeu_por(&campo)
   @vendas.select { | l | l.tipo == "revista"}.sort {|v1,v2| quantidade_de_vendas_por(v1, &campo
  end
```

Vamos incluir o tipo livro nos quatro livros já existentes em nosso código:

```
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")
```

E vamos criar uma revista, que neste exemplo chamamos de revistona:

```
revistona = Livro.new("Revista de Ruby", 10, 2012, true, "Revistas", "revista")
```

Agora para testar o funcionamento do código, adicionamos a revistona duas vezes ao nosso estoque, efetuamos uma venda e imprimimos a revista que mais vendeu por titulo:

```
estoque = Estoque.new
estoque << algoritmos << algoritmos << ruby << programmer << arquitetura << ruby << ruby << rev
estoque.vende ruby
estoque.vende algoritmos
estoque.vende algoritmos
estoque.vende revistona
puts estoque.livro_que_mais_vendeu_por_titulo.titulo
puts estoque.revista_que_mais_vendeu_por_titulo.titulo
```

Ao executar o programa, ele imprimirá a revista de Ruby:

> Algoritmos
>
> Revista de Ruby

Mas observe que os métodos `livro_que_mais_vendeu_por` e `revista_que_mais_vendeu_por` são quase iguais, só muda uma string. Podemos então generalizar a parte comum aos dois métodos com um método chamado `que_mais_vendeu_por` que receba um tipo como parâmetro:

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

E agora alteramos nossos métodos efetivamente devolvem o produto mais vendido daquele tipo para chamar o método genérico passando o tipo como argumento:

```
def livro_que_mais_vendeu_por_titulo
   que_mais_vendeu_por("livro", &:titulo)
end

def livro_que_mais_vendeu_por_ano
   que_mais_vendeu_por("livro", &:ano_lancamento)
end

def livro_que_mais_vendeu_por_editora
   que_mais_vendeu_por("livro", &:editora)
end

def revista_que_mais_vendeu_por_titulo
   que_mais_vendeu_por("revista", &:titulo)
end

def revista_que_mais_vendeu_por_ano
```

```
    que_mais_vendeu_por("revista", &:ano_lancamento)
end

def revista_que_mais_vendeu_por_editora
    que_mais_vendeu_por("revista", &:editora)
end
```

Execute o programa novamente e observe que o comportamento é o mesmo, mas observe a quantidade de métodos similares que temos em nosso código. Será que podemos alterá-lo de alguma maneira para evitarmos essa continua replicação de código? E não seria possível representarmos revistas e livros de outra maneira que não utilizando uma string? Veremos a seguir soluções para estas questões.