02

Revisitando a Orientação a Objetos

Introdução

Nem sempre utilizamos as melhores práticas para escrever nosso código. Por mais que algumas vezes pareçam enxutos e bem estruturados, nossos programas podem ser melhorados em alguns aspectos fazendo uso de boas práticas de Orientação a Objetos. Alguns jargões ou recursos amplamente difundidos em comunidades de algumas linguagens ou frameworks podem ser melhor utilizados após uma análise crítica sobre o código que produzimos.

Certamente você já se deparou com códigos muito confusos e difíceis de se manterem devido a problemas como repetição excessiva de código, muitas linhas para poder produzir uma tarefa que em tese seria simples, métodos muito grandes que fazem muitas tarefas e às vezes em lugares pouco óbvios de serem localizados; todos esses, entre outros problemas, causam certo transtorno para qualquer desenvolvedor.

Em Ruby não é diferente: podemos e devemos escrever programas em Ruby usufruindo das boas práticas de orientação a objetos. É muito importante combinar os recursos que a linguagem nos oferece com as boas práticas de OO para termos um código simples, claro e de fácil manutenção, agilizando o trabalho dos desenvolvedores.

Para exemplificar algumas situações comuns, vamos construir um pequeno sistema que nos permita catalogar os produtos de uma livraria, puramente em Ruby, sem frameworks adicionais. Primeiramente, cadastraremos apenas livros, e, posteriormente, revistas, ebooks e outros. Utilizando orientação a objetos, podemos criar, para representar um livro, a classe `Livro`:

```
class Livro

end
```

O problema de getters e setters para todos os atributos

Nossos livros precisam ter algumas características indispensáveis: título, ano de lançamento e preço de venda. Convertemos essas informações em atributos:

```
class Livro
  attr_accessor :titulo, :preco, :ano_lancamento
end
```

Agora podemos instanciar objetos do tipo `Livro` e atribuir alguns valores para eles:

```
livro_rails = Livro.new
livro_rails.preco = 70.00
livro_rails.titulo = "Agile Web Development with Rails"
livro_rails.ano_lancamento = 2011

livro_ruby = Livro.new
livro_ruby.preco = 60.00
livro_ruby.titulo = "Programming Ruby 1.9"
livro_ruby.ano_lancamento = 2010
```

```
livros = [livro_rails, livro_ruby]
```

Suponha que os dois livros acima foram comprados por algum cliente. Em algum lugar de nosso sistema poderíamos ter um método para listar os itens que foram comprados. Esse método recebe uma array de livros:

```
def imprime_nota_fiscal(livros)
  livros.each do |livro|
    puts "Titulo: #{livro.titulo} - #{livro.preco}"
  end
end
```

Tudo parece funcionar bem:

```
Titulo: Agile Web Development with Rails - 70.0

Titulo: Programming Ruby 1.9 - 60.0
```

Até que alguém tem a ideia de alterar o preço dos livros no método `imprime_nota_fiscal` :

```
def imprime_nota_fiscal(livros)
  livros.each do |livro|
    livro.preco = 1.00
    puts "Titulo: #{livro.titulo} - #{livro.preco}"
  end
end
```

Claro que ser capaz de alterar o preço do livro na hora de imprimir a nota fiscal é um caso díficil de acontecer. Entretanto, não há nada que impeça o acesso ao preço do livro no método `imprime_nota_fiscal` ! Dessa forma, ao rodar o código

```
livros = [livro_rails, livro_ruby]
imprime_nota_fiscal livros
```

obtemos como resposta:

```
Titulo: Agile Web Development with Rails - 1.0

Titulo: Programming Ruby 1.9 - 1.0
```

Note que os preços serão emitidos na nota fiscal com valores diferentes do que esperávamos! É uma questão simples, mas possivelmente talvez você já tenha se deparado com uma oportunidade de modificar um valor que não deveria ser alterado naquele trecho de código. Ou encontrou um código que acessava variáveis, alterando o valor delas, quando elas não deveriam ser alteradas diretamente, sem passar por algum processamento.

Por que isso ocorre? Como poderíamos resolver o problema? Note que, na maioria dos lugares, provavelmente não desejaremos alterar o preço do livro, apenas ler seu conteúdo. Até agora, o único momento em que necessitamos

estabelecer um preço foi logo após a criação de algum objeto do tipo `Livro`. Repare também que a mesma analogia é válida para o `titulo` e `ano_lancamento`: na maioria das situações precisaremos apenas ler seus valores.

Encapsulando variáveis membro

Veja que manter um attr_acessor para os atributos da classe `Livro` não é uma ideia tão boa assim. E se, em vez disso, usássemos apenas um `attr_reader`?

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

Rodando a aplicação, o Ruby reclama que não podemos setar nenhum valor:

```
livro.rb:6: undefined method 'preco=' for #<Livro:0x100000> (NoMethodError)
```

Como só estabelecemos valores para os atributos no momento da criação do objeto, poderíamos realizar essa tarefa via construtor:

```
class Livro
  attr_reader :titulo, :preco, :ano_lancamento
  def initialize(titulo, preco, ano_lancamento)
    @titulo = titulo
    @preco = preco
    @ano_lancamento = ano_lancamento
  end
end
```

Agora instanciamos nossos livros passando esses argumentos:

```
livro_rails = Livro.new("Agile Web Development with Rails",70,2011)
livro_ruby = Livro.new("Programming Ruby 1.9",60,2010)
```

Mas o erro persiste, uma vez que o método `imprime_nota_fiscal` tenta chamar um accessor que não existe mais. Ele tenta quebrar o encapsulamento de nosso preço:

```
livro.rb:6: undefined method 'preco=' for #<Livro:0x100000> (NoMethodError)
```

Construção de objetos e lógica

Considere agora uma outra situação: a gerência deciciu dar um desconto para os livros antigos que fossem cadastrados no estoque. Qualquer livro cadastrado a partir de hoje cujo ano de lançamento for inferior a 2000 terá 30% de desconto. Note que livros anteriormente cadastrados no estoque não receberão esse desconto. Portanto, foi feito um método que, no momento em que um livro é cadastrado no estoque, checa seu ano de lançamento e altera seu preço se necessário:

```
def aplica_promocoes(livro)
  if livro.ano_lancamento < 2000
     livro.preco *= 0.7
end
```

Repare que, devido à nova regra para a liquidação de títulos antigos, teremos que alterar o preço do livro; contudo o `preco` é apenas um atributo que pode ser lido. Aparentemente, precisaremos torná-lo um `attr_accessor` novamente:

```
  class Livro
    attr_reader :titulo, :ano_lancamento
    attr_accessor :preco
    def initialize(titulo, preco, ano_lancamento)
      @titulo = titulo
      @preco = preco
      @ano_lancamento = ano_lancamento
    end
end
```

Também foi solicitado que os títulos da liquidação fossem anunciados juntamente ao preço promocional no newsletter semanal de nossa livraria, que incluirá todos os livros cujos lançamentos forem anteriores a 1999. O código da newsletter verifica se o livro está no período de lançamento adequado e adiciona o título e o preço na newsletter:

```
def livro_para_newsletter(livro)
  if livro.ano_lancamento < 1999
     puts "Newsletter/Liquidacao"
     puts livro.titulo
     puts livro.preco
  end
end
```

Acabamos de cadastrar um novo livro e invocamos os métodos `livro_para_newsletter(livro)` e `aplica_promocoes(livro)`:

```
algoritmos = Livro.new("Algoritmos", 100, 1998)
livro_para_newsletter algoritmos
aplica_promocoes algoritmos
```

O código acima deveria nos retornar algo como:

Newsletter

Algoritmos

70.0

Entretanto, se executarmos o código acima, obteremos uma saída diferente como resposta:

Newsletter

```
Algoritmos

100.0
```

Note que o preço do livro no newsletter possui um valor incorreto! Isso acontece porque invocamos o método do newsletter antes de invocar o método para aplicar o preço promocional! E repare que o método do newsletter não efetua reajuste do preço do livro! Podemos resolver esse problema aplicando nossa lógica de desconto no momento de instanciarmos um novo livro:

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

  def initialize(titulo, preco, ano_lancamento)
    @titulo = titulo
    @preco = preco
    @ano_lancamento = ano_lancamento
    if ano_lancamento < 2000
      @preco *= 0.7
    end
  end
end
```

Com isso, podemos remover quaisquer reajustes de preço existentes em nosso código, como o método `aplica_promocoes(livro)`:

```
def aplica_promocoes(livro)
  if livro.ano_lancamento < 2000
     livro.preco *= 0.7
end
```

Extraindo métodos

Podemos ainda refatorar nosso método `initialize` da classe `Livro` para executar nossa lógica de desconto de uma forma que deixe nosso código mais legível:

```
class Livro
  attr_reader :titulo, :ano_lancamento
  attr_accessor :preco
  def initialize(titulo, preco, ano_lancamento)
    @titulo = titulo
    @ano_lancamento = ano_lancamento
    @preco = calcula_preco(preco)
  end

private

def calcula_preco(base)
    if @ano_lancamento < 2000
      base * 0.7
```

```
    else
      base
    end
  end
end
```

Apenas isolamos nossa regra de negócios para um método e o invocamos no construtor. Extraímos um método e o mantemos como privado para que não seja acessado por fora do objeto: encapsulamos seu comportamento.

Note que agora não precisamos mais que o atributo `preco` seja um `attr_accessor`, pois seu valor, de acordo com nossa regra de negócios, será estipulado no momento da criação do objeto. Podemos então alterá-lo para um `attr_reader`:

```
  class Livro
    attr_reader :titulo, :ano_lancamento, :preco
    def initialize(titulo, preco, ano_lancamento)
      @titulo = titulo
      @ano_lancamento = ano_lancamento
      @preco = calcula_preco(preco)
    end

private

 def calcula_preco(base)
    if @ano_lancamento < 2000
      base * 0.7
    else
      base
    end
  end
end
```

Note que isso resolve diversos dos nossos problemas. O primeiro deles é que não precisamos mais expor métodos de atribuição de valor às variáveis membro, como o `preco`. Também unificamos nossa regra de negócios em um único ponto: na classe `Livro`. Os dados de um livro estão juntos com o comportamento ligado a ele. Essa é a base de Orientação a Objetos: dados e comportamento que são relacionados ficam no mesmo lugar.

Por fim, isso não nos obriga a decorar a ordem de métodos que devem ser chamados para reajustar o preço dos livros, uma vez que o próprio objeto saberá como deve recalcular seu valor.

Evoluindo um código encapsulado

Imagine agora que precisamos saber se nossos livros possuem alguma reimpressão ou não. Sempre que instanciarmos um objeto do tipo `Livro`, precisamos passar essa informação. Para tanto, seguindo a lógica que usamos anteriormente, criamos um novo atributo `possui_reimpressao` sem fazer uso de `attr_accessor`, considerando que isso poderia causar acessos indesejados ao valor do atributo, e inicializamos seu valor no construtor (`initialize`):

```
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
```

Criamos também, como é convencionado em Ruby, um método chamado `possui_reimpressao?` , que nos devolverá o valor de `@possui_reimpressao` :

```
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 possui_reimpressao?
    @possui_reimpressao
  end
```

E mudamos a criação de nossos livros para passar esse argumento:

```
algoritmos = Livro.new("Algoritmos",100,1998,true)
```

Como escrever código Ruby utilizando recursos da linguagem e boas práticas de Orientação a Objetos será o foco dos próximos capítulos.

Mãos à obra! Boa sorte nos exercícios!