02

Extraindo comportamento e duck typing

Com a expansão da empresa, nosso modelo Livro passa a receber também uma variável que define se o livro possui sobrecapa. Como de costume, definimos a variável no construtor:

```
class Livro

  def initialize(titulo, preco, ano_lancamento, possui_reimpressao, possui_sobrecapa, editora, t
    @titulo = titulo
    @preco = preco
    @ano_lancamento = ano_lancamento
    @possui_reimpressao = possui_reimpressao
    @possui_sobrecapa = possui_sobrecapa
    @editora = editora
    @tipo = tipo
  end

  # metodos
end
```

Da mesma maneira, cada revista deve armazenar qual o seu número. No caso de uma revista de Jardinagem que está em seu trigésimo mês, seu número seria 30. Uma revista de arquitetura em seu quarto mês teria numero 4. Com o mesmo padrão modificamos novamente o construtor:

```
class Livro

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

  # metodos
end
```

No nosso sistema verificamos a criação atual de um livro e de uma revista:

```
livro = Livro.new("Programming Ruby", 100, 2004, true, "", "livro")
revistona = Livro.new("Revista de Ruby", 10, 2012, true, "Revistas", "revista")
```

Mas o que passar como valor numero para o Livro? E o que passar como possui sobrecapa em uma Revista? Não faz sentido colocar valores. Pior ainda, não faz sentido ter essas variáveis no objeto, seja como null ou qualquer outra coisa. Uma revista não deveria ter campos relativos a livro, mesmo que opcionais. E vice-versa.

Além disso, já está visivelmente estranho que uma revista é um livro quando instanciamos ela. Estamos utilizando uma classe chamada Livro para representar revistas?!

Uma primeira solução é copiar a classe Livro contida no arquivo livro.rb e colar todo o código em revista.rb, somente mudando o nome da classe.

Na classe Livro mantemos somente a questão de possuir sobrecapa:

```
class Livro

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

  # metodos
end
```

Na classe Revista podemos adicionar somente o numero:

```
class Revista

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

  # metodos
end
```

Olhando agora cada uma dessas classes, o uso da string tipo fica ainda mais grosseiro do que já era. Usar uma string para definir de que tipo é um objeto soa cada vez mais estranho já que a classe deste objeto já determina isso. Então, vamos remover completamente essa variável tipo, seu attr_reader e a atribuição da variável, tanto no Livro quanto na Revista. O objeto é do tipo que ele é e não de um tipo definido por alguma string.

Por fim, criamos a classe EBook com a mesma estratégia e sem sobrecapa:

```
class EBook

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

  # metodos
end
```

Mas EBooks não possuem a questão de reimpressão portanto removemos tal atributo do construtor, da atribuição e o método `possui_reimpressao` ?.

No nosso sistema, atualizamos o require para adicionar a revista e o ebook:

```
require_relative 'livro'
require_relative 'revista'
require_relative 'ebook'
require_relative 'estoque'
```

Na criação dos objetos agora definimos cada objeto como ele realmente deve ser, EBook, Livro, Revista:

```
livro = Livro.new("Programming Ruby", 100, 2004, true, true, "", "livro")
revistona = Livro.new("Revista de Ruby 3", 10, 2012, true, 3, "Revistas", "revista")
```

Ao rodar a aplicação ocorre um erro:

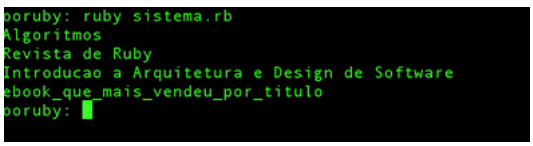
Note que ainda utilizamos o antigo método tipo no método que_mais_vendeu_por:

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

Uma solução bem simples seria implementar em cada classe um método tipo:

```
class Livro

  def tipo
    "livro"
  end

  # todo o codigo aqui
end
class EBook

  def tipo
    "ebook"
  end

  # todo o codigo aqui
end
class Revista

  def tipo
    "revista"
  end

  # todo o codigo aqui
end
```

Apesar do sistema voltar a funcionar podemos notar algo de estranho com a saída. A nossa chamada ao método respond_to devolve o nome do método que ele encontrou, ao invés de devolver true ou false.



Enquanto no Ruby fazer um if nesse valor funcionaria como o caso verdadeiro, é muito estranho que o método devolva algo diferente de verdadeiro ou falso, uma vez que na impressão o resultado deixa de fazer sentido. Vamos relembrar o método respond_to? de nosso estoque:

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

Note que se o método resultar com um match perfeito, ele devolve uma string. Vamos alterá-lo para devolver true ou false:

```
matched = name.to_s.match("(.+)_que_mais_vendeu_por_(.+)")
matched || super
```

O que podemos fazer após extrair essa variável é uma dupla transformação: inverter o valor para inverter novamente. Isto é, se ele é vazio, ele vira true, que vira false. Se ele é uma string, vira false, que vira true. Para isso basta usarmos dois "!".

```
matched = name.to_s.match("(.+)_que_mais_vendeu_por_(.+)")
!!matched || super
```

A prática de usar !! tem sua vantagem e desvantagem: ela funciona bem ao converter um objeto para true ou false de acordo com sua existência, mas confunde um pouco o leitor. O importante é lembrar que se há um objeto, retorna true, se não, false.

Rodamos a aplicação e pecebemos que o respond_to? passou a devolver true ou false. Mas ainda há algo de estranho que desejamos refatorar. Cada classe retorna exatamente o que ela é. É estranho ter que implementar um método que diz o que sou, como me comporto.

Olhando o método que_mais_vendeu_por novamente, vemos que ele chama os produtos vendidos de l e não de produto. Vamos mudar:

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

Agora note como filtramos através do tipo:

```
  @vendas.select {|produto| produto.tipo == tipo}
```

Ao invés de buscar por um tipo específico, perguntando que tipo ele é (ask), iremos dizer a ele um tipo, e o próprio produto verifica se ele é do mesmo tipo que passamos, isto é, se tem um match entre os tipos. Dessa maneira, não quebramos o encapsulamento da classe.

```
  @vendas.select {|produto| produto.matches?(tipo) }
```

Implementar o método matches? em Livro fica bem fácil:

```
class Livro

  def matches?(query)
    query=="livro"
  end

end
```

E note que só de mudar essa maneira de trabalhar passamos a suportar queries mais complexas:

```
class Livro

  def matches?(query)
    query=="livro" || query=="impresso"
  end

end
class Revista

  def matches?(query)
    query=="revista" || query=="impresso"
  end

end
class EBook

  def matches?(query)
    query=="ebook" || query=="digital"
  end

end
```

Agora podemos não só classificar pelo tipo do produto vendido como ebook, revista ou livro mas também por características ou categorias. Uma outra maneira de escrever a comparação seria:

```
class EBook

  def matches?(query)
    ["ebook", "digital"].include?(query)
  end

end
```

Rodando a aplicação novamente, verificamos que tudo funciona. Mas, como é possível chamarmos o método `matches?` em todos os objetos da mesma maneira?

Em Ruby, podemos chamar qualquer método em qualquer objeto, independente de ele ser de um tipo específico. Se o objeto responde (respond_to?) por aquele método (:matches?), ele será invocado normalmente. O nome dessa característica da linguagem é duck typing: não importa qual o tipo do objeto e sim se ele possui o método ou não.

Uma das vantagens dessa abordagem é não precisar de uma fase de compilação para garantir que o método exista. Contudo, uma desvantagem é que se tal método existe mas executa uma tarefa diferente da imaginada (possui semântica diferente), o método será chamado erroneamente.