02

Modelos anêmicos e encapsulamento

Nossa livraria deseja mudar a regra de descontos, baseando-se em um novo processo, de acordo com a raridade do livro.

1. Se o livro foi lançado em um ano anterior a 2006 e não possui reimpressões, dê 5% de desconto.
2. Se o livro foi lançado em um ano anterior a 2006 e possui reimpressões, dê 10% de desconto.
3. Se o livro foi lançado entre 2006 e 2010 - inclusive, dê 4% de desconto.

Vamos refatorar o método `calcula_preco` e adicionar nele as nossas novas regras:

```
def calcula_preco(base)
    if @ano_lancamento < 2006
        if @possui_reimpressao
            return base * 0.9
        else
            return base * 0.95
        end
    elsif @ano_lancamento <=2010
        return base * 0.96
    else
        return base
    end
  end
end
```

Repare que, como já havíamos encapsulado esse código, podemos alterá-lo sem medo de afetar outras partes do sistema. Todos que precisam saber o preço de um livro continuam sabendo da mesma maneira, e todas as partes do sistema que deveriam ser afetadas (todos que querem saber o preço) serão afetadas.

Podemos refatorar o código e remover a palavra chave "return" uma vez que cada condição só possui uma linha e é válida como return:

```
def calcula_preco(base)
    if @ano_lancamento < 2006
        if @possui_reimpressao
            base * 0.9
        else
            base * 0.95
        end
    elsif @ano_lancamento <=2010
        base * 0.96
    else
        base
    end
  end
end
```

Porém, observe como o método ficou com diversos `ifs` e tornou-se relativamente longo. Como faríamos para melhorar esse trecho de código?

Essas e outras perguntas serão respondidas ao longo desse curso.

Um estoque de livros pode ser representado por várias instâncias de Livro. Uma forma de representar um estoque seria armazenar essas instâncias em um array. Vamos criar um "estoque" com o livro de algoritmos que já temos e com um novo livro de arquitetura que vamos criar, conforme abaixo:

```
arquitetura = Livro.new("Introdução À Arquitetura e Design de Software", 70, 2011, true)
estoque = [algoritmos, arquitetura]
```

Para cadastrar mais livros, basta adicionar novos objetos ao array. Vamos cadastrar mais dois livros usando o método `<<` da classe `Array`:

```
estoque << Livro.new("The Pragmatic Programmer", 100, 1999, true)
estoque << Livro.new("Programming Ruby", 100, 2004, true)
```

É necessária uma nova funcionalidade para que outros sistemas possam obter informações sobre o nosso estoque. Para isso, vamos exportar o estoque em formato CSV que significa "Comma-Separated Values" (http://pt.wikipedia.org/wiki/Comma-separated_values (http://pt.wikipedia.org/wiki/Comma-separated_values)): cada livro será representado pelo seu título e ano de lançamento, separados por vírgula.

Cada linha tem um único livro. Exportar a lista atual seguindo essas regras deve gerar o seguinte resultado:

> Algoritmos, 1998
>
> Introdução À Arquitetura e Design de Software,2011
>
> The Pragmatic Programming, 1999
>
> Programming Ruby, 2004

Uma possível implementação dessa funcionalidade seria iterar pelos elementos do array, imprimindo cada um deles no formato esperado:

```
estoque.each do |livro|
  puts "#{livro.titulo},#{livro.ano_lancamento}"
end
```

Há um problema com essa abordagem. Em qualquer lugar do sistema que for preciso utilizar a exportação, será necessário copiar e colar esse código. Então vamos isolar esse comportamento em um método. Assim, sempre que for preciso exportar os livros, basta chamar esse método passando o estoque. Vamos chamá-lo de `exporta_csv`:

```
def exporta_csv(estoque)
  estoque.each do |livro|
    puts "#{livro.titulo},#{livro.ano_lancamento}"
```

```
  end
end
```

Portanto, chamando o método `exporta_csv(estoque)`, obtemos como resposta:

```
Algoritmos,1998
Introdução À Arquitetura e Design de Software,2011
The Pragmatic Programmer,1999
Programming Ruby,2004
```

Uma outra funcionalidade importante é saber qual o total de livros disponíveis em estoque. Como temos um array que representa o estoque, usamos em geral a chamada

```
estoque.size
```

Também será preciso informar a quantidade de livros com valor maior que o preço que for passado como argumento. Portanto, desejamos selecionar os elementos cujo preço é menor que um valor determinado, algo como:

```
livro.preco <= valor
```

O método de `Array` que faz isso é o `select`. Esse método recebe um bloco que recebe um parâmetro e deve retornar `true` ou `false`.

```
estoque.select do |livro|
  livro.preco <= valor
end
```

O resultado é um novo array somente com os elementos que, passados para o bloco, fazem ele devolver `true`.

Vamos colocar essa funcionalidade num método e chamá-lo de `mais_barato_que`. Assim, quando for necessário saber o total de livros com valor abaixo de R$ 80,00 em estoque, basta invocar: `mais_barato_que(estoque, 80)`

```
def mais_barato_que(estoque, valor)
  estoque.select do |livro|
    livro.preco <= valor
  end
end

baratos = mais_barato_que(estoque, 80)
baratos.each do |livro|
  puts livro.titulo
end
```

Dessa forma, obteremos como resposta:

Introdução À Arquitetura e Design de Software

Repare que estamos criando vários métodos com funcionalidades relacionadas ao estoque. O conceito de estoque é essencial para cada um desses métodos e trechos de código, tanto que, para que eles sejam úteis, é preciso passar o estoque como argumento quando invocamos cada um deles.

O estoque, assim como um livro, é uma parte importante do domínio no qual estamos trabalhando. Todos esses métodos que criamos têm responsabilidades relacionadas ao estoque. Isso é uma pista de que todos esses métodos poderiam estar agrupados em uma classe que contém determinados comportamentos e responsabilidades: a classe `Estoque`.

Um estoque é formado por várias instâncias de livros. Podemos representar isso com uma propriedade `livros` na classe `Estoque`:

```
class Estoque
  attr_reader :livros
end
```

Agora podemos criar um estoque mas, se tentarmos adicionar um novo livro, teremos problema porque não inicializamos o atributo `livros` com um array. Vamos adicionar o método initialize em `Estoque` para que toda instância dessa classe tenha um atributo `livros` que é um array:

```
class Estoque
  attr_reader :livros
  def initialize
    @livros = []
  end
end
```

Agora podemos criar um estoque e adicionar livros tranquilamente:

```
estoque = Estoque.new
estoque.livros << algoritmos << arquitetura
estoque.livros << Livro.new("The Pragmatic Programmer", 100, 1999, true)
estoque.livros << Livro.new("Programming Ruby", 100, 2004, true)
```

Vamos mover o método `exporta_csv` para a classe `Estoque`. Atualmente ele recebe um array como argumento, mas agora podemos usar a variável de instância `livros` da classe. Após essa alteração será possível exportar um estoque simplesmente invocando o método `estoque.exporta_csv`. Vamos mover o método:

```
class Estoque
  attr_reader :livros

  def initialize
    @livros = []
  end

  def exporta_csv
    @livros.each do |livro|
```

```
        puts "#{livro.titulo},#{livro.ano_lancamento}"
      end
    end
  end
```

Para ver o estoque em funcionamento, comente o código do método `mais_barato_que` :

```
#baratos = mais_barato_que(estoque, 80)
#baratos.each do |livro|
#puts livro.titulo
#end
```

Para usar a funcionalidade de exportação, é preciso invocar o método `exporta_csv` em um objeto estoque:

```
estoque.exporta_csv
```

E o resultado será:

```
Algoritmos,1998

Introdução À Arquitetura e Design de Software,2011

The Pragmatic Programmer,1999

Programming Ruby,2004
```

O método `mais_barato_que` também será movido para a classe `Estoque` :

```
class Estoque
  # código já existente na classe

  def mais_barato_que(valor)
    @livros.select do |livro|
      livro.preco <= valor
    end
  end
end
```

Dessa forma, modificamos a invocação de seu método para:

```
baratos = estoque.mais_barato_que 80
baratos.each do |livro|
  puts livro.titulo
end
```

E, quando executarmos o código novamente, teremos como resposta:

```
Algoritmos,1998
```

Introdução À Arquitetura e Design de Software,2011

The Pragmatic Programmer,1999

Programming Ruby,2004

Introdução À Arquitetura e Design de Software,2011

Mas como descobrir a quantidade de livros dentro do estoque? Em vez de acessar diretamente a variável `livros`, vamos criar um método que represente nossa necessidade:

```
class Estoque
  # código já existente na classe

  def total
    @livros.size
  end
end
```

Para descobrir o total de livros atuais em um estoque, podemos invocar o método `total`:

```
estoque.total
```

Em nosso estoque atual, o resultado seria `4`.

Para criar uma instância de `Livro` é preciso passar os argumentos: título, preço e ano. Se adicionarmos ao estoque um livro nulo (`estoque.livros << nil`), teremos a seguinte saída:

```
in 'exporta_csv':undefined method 'titulo' for nil: NilClass (NoMethodError)
```

Para resolver isso, podemos criar um método na classe `Estoque` que recebe um instância de `Livro` e adiciona esse livro apenas se ele for um livro válido. Vamos criar o método `adiciona`:

```
class Estoque
  # código já existente na classe

  def adiciona(livro)
    @livros << livro if livro
  end
end
```

Adicionar um livro ao estoque agora é feito invocando o método `adiciona`. Vamos criar um novo estoque e inserir os livros que estavam sendo inseridos no anterior:

```
estoque = Estoque.new
estoque.adiciona  algoritmos
estoque.adiciona  arquitetura
estoque.adiciona  Livro.new("The Pragmatic Programmer", 100, 1999, true)
```

```
estoque.adiciona  Livro.new("Programming Ruby", 100, 2004, true)
estoque.adiciona nil
```

Agora basta avisar todos os desenvolvedores para nunca acessarem o atributo `livros` de um estoque pois, se ele for usado para adicionar um livro ao estoque, essa lógica que acabamos de criar não será executada.

Precisamos pendurar esse aviso por todo o ambiente de trabalho para que ninguém nunca se esqueça disso. Quantas vezes não vimos alguém acessando um valor de uma maneira que não deveria simplesmente porque aquele valor estava disponível para ser acessado?

Nossa classe `Estoque` deixa visível parte de seu comportamento interno para quem a instancia; a forma como ela armazena os livros, para ser mais exato. Um programador que conheça os métodos da classe `Array` (usada internamente pelo estoque) pode adicionar livros à vontade, mesmo sem um preço. Ou pior, além de adicionar alguém poderia retirar, alterar...

A classe `Estoque` não possui nenhum controle sobre o que acontece com seus valores internos, isto é, a variável membro é pública para ser acessada e escrita.

Não desejamos que seja possível alterar a lista sem a autorização do estoque. Não deveria ser possível fazer nada com algo interno ao estoque sem antes passar pelo estoque. Essa "proteção" do estado interno de um objeto é o que chamamos de encapsulamento.

Quando foi usado o método `attr_reader :livros` , foi criado um método de instância na classe livro que retorna uma variável de instância com o mesmo nome (`@livros`). O método que foi gerado é um "getter" (http://blog.caelum.com.br/nao-aprender-oo-getters-e-setters/ (http://blog.caelum.com.br/nao-aprender-oo-getters-e-setters/)), e é esse método que está expondo o estado interno do nosso estoque. Vamos corrigir isso!

Precisamos remover a linha que cria o "getter". Para isso é só apagar a linha `attr_reader :livros` da classe `Estoque` .

Vejamos como ficou o código completo da classe `Estoque` após todas as alterações:

```
class Estoque
  def initialize
    @livros = []
  end

  def exporta_csv
    @livros.each do |livro|
      puts "#{livro.titulo},#{livro.ano_lancamento}"
    end
  end

  def mais_barato_que(valor)
    @livros.select do |livro|
      livro.preco <= valor
    end
  end

  def total
    @livros.size
  end

  def adiciona(livro)
```

```
      @livros << livro if livro
    end
  end
```

Foi feita uma solicitação enquanto a classe `Estoque` era criada e aperfeiçoada. A exportação precisa incluir também o preço de cada livro. Esse comportamento está no método `exporta_csv` de `Estoque`. Basta adicionar uma chamada ao método de `Livro` que retorna o preço e incluir o retorno na exportação:

```
class Estoque
  # código já existente na classe

  def exporta_csv
    @livros.each do |livro|
      puts "#{livro.titulo},#{livro.ano_lancamento},#{livro.preco}"
    end
  end
end
```

Vamos checar se tudo está correto. Execute o código e verifique a saída:

```
Algoritmos,1998,90.0

Introdução À Arquitetura e Design de Software,2011,70

The Pragmatic Programmer,1999,90.0

Programming Ruby,2004,90.0
```

Mas repare bem no código do método `exporta_csv`: boa parte da funcionalidade consiste em fazer chamadas para um outro objeto. Esse objeto é uma instância de `Livro` que é passada como parâmetro para o bloco de `each`:

```
@livros.each do |livro|
  puts "#{livro.titulo},#{livro.ano_lancamento},#{livro.preco}"
end
```

Outra coisa que podemos notar é que os métodos que estão sendo invocados em `livro` estão apenas retornando o estado atual do objeto `livro`. O método `exporta_csv` está preocupado apenas com o estado interno de um livro. A cada iteração do loop o método pergunta para `livro` qual o valor de suas propriedades.

Durante o processo de construção da classe `Estoque`, vimos os problemas que aparecem quando o estado interno de um objeto é conhecido por outros objetos externos. Poderíamos encapsular o acesso aos dados de um livro mas, nesse caso, pode ser interessante poder resgatar somente o preço ou o título de um livro qualquer.

E se `Livro` possuir a habilidade de se apresentar no formato CSV? Nesse caso o método `exporta_csv` de `Estoque` não precisa se preocupar em perguntar nada para um livro, mas apenas dizer para o livro se formatar em CSV! Vamos fazer essa alteração em `Livro`:

```
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
end
```

Antes um livro era apenas uma forma de armazenar dados. Agora, os objetos desse tipo oferecem uma forma de acessar esses dados internos, e existe uma lógica envolvida nesse acesso: um formato específico é retornado. Podemos alterar o método `exporta_csv` de `Estoque`, para que passe a fazer uso desse comportamento:

```
class Estoque
  # código já existente na classe

  def exporta_csv
    @livros.each do |livro|
      puts livro.to_csv
    end
  end
end
```

Para garantir que tudo está funcionando, vamos invocar o método `exporta_csv`:

```
estoque.exporta_csv
```

O resultado será:

```
Algoritmos,1998,90.0

Introdução À Arquitetura e Design de Software,2011,70

The Pragmatic Programmer,1999,90.0

Programming Ruby,2004,90.0
```

Ótimo! Agora nossa classe `Estoque` não sabe mais quais são as propriedade de um livro, ela apenas diz para o livro que precisa usar a forma "CSV" de apresentação. E o livro é o verdadeiro responsável por criar esse formato e o retornar. Como `Estoque` não pergunta mais para `Livro` e apenas diz para ele fazer algo, podemos dizer que agora o método `exporta_csv` usa o princípio Tell, Don't Ask.

Por fim, nossa classe `Livro` ganhou algum comportamento. Orientação a objetos é sobre aliar dados e comportamentos para acessar e alterar esses dados. Inclusive há um termo usado para definir objetos que são utilizados apenas para carregar dados: modelo anêmico ou anemic domain model (http://www.arquiteturajava.com.br/livro/cuidado-com-o-modelo-anemico.pdf (http://www.arquiteturajava.com.br/livro/cuidado-com-o-modelo-anemico.pdf)).

É importante observarmos a forma como nossos objetos interagem na medida em que adicionamos funcionalidade a um sistema. Se um objeto depende muito do estado interno de algum outro, estamos quebrando o encapsulamento.

O encapsulamento é importante não só para que fique claro qual objeto tem responsabilidade sobre qual informação, mas também para tornar os objetos menos dependentes ou acoplados uns aos outros. Objetos muito acoplados dificultam a evolução de um sistema já que alterar uma única classe pode ter muitos efeitos colaterais em qualquer outro método que usa instâncias daquele tipo.