CAPÍTULO 18

Apêndice: Design Patterns em Ruby

18.1 – SINGLETON

Como criar uma classe com a garantia de ser instanciada apenas uma vez ? Essa classe deveria ser capaz de verificar se alguma vez já foi instanciada e saber devolver sempre a mesma referência.

Como visto anteriormente, em ruby, variáveis com @ são de instância e variáveis com @@ são variáveis de classe.

Utilizando a ideia acima podemos criar uma classe simples de relatório, onde desejamos que apenas uma seja de fato criada.

```
class Relatorio
  @@instance = Relatorio.new

  def self.instance
    return @@instance
  end
end
```

Dessa forma conseguimos criar o relatório apenas uma vez. Mas a implementação ainda apresenta problemas. Ainda é possível instanciar o Relatório mais de uma vez. Para resolver esse problema e implementar a classe Relatório como um Singleton realmente, precisamos tornar privado o **new** da nossa classe. Dessa forma, apenas será possível acessar o Relatorio a partir do método **instance**.

```
class Relatorio
  @@instance = Relatorio.new

  def self.instance
    return @@instance
  end

  private_class_method :new
```

```
end

# ambos relatórios referenciam o mesmo objeto
relatorio1 = Relatorio.instance
relatorio2 = Relatorio.instance
```

Existe ainda uma maneira mais otimizada e produtiva de chegar no mesmo resultado. O ruby já vem com um módulo chamado **Singleton**. Basta inclui-lo na classe para ter o mesmo resultado. Você verá mais sobre módulos no decorrer do curso.

```
require 'singleton'
class Relatorio
  include Singleton
end
```

18.2 – Exercícios: Singleton

1. Crie o Relatório de forma a retornar sempre a mesma instância:

```
class Relatorio
  @@instance = Relatorio.new

  def self.instance
    return @@instance
  end

  private_class_method :new
end

# ambos relatórios referenciam o mesmo objeto
relatorio1 = Relatorio.instance
relatorio2 = Relatorio.instance

puts relatorio1 == relatorio2
```

2. Faça o mesmo teste, mas conhecendo agora o módulo Singleton do ruby:

```
require 'singleton'
class Relatorio
  include Singleton
end

# ambos relatórios referenciam o mesmo objeto
relatorio1 = Relatorio.instance
relatorio2 = Relatorio.instance

puts relatorio1 == relatorio2
```

18.3 - Template Method

As operações de um sistema podem ser de diversos tipos e o que determina qual operação será realizada pode ser um simples variável. Dependendo da variável uma coisa pode ocorrer enquanto outra vez é necessário fazer outra. Para exemplificar melhor, vamos usar a classe **Relatorio**. Nosso relatório mostrará um conteúdo inicialmente em HTML.

```
class Relatorio
  def imprime
    puts "<html>Dados do restaurante</html>"
  end
end
```

O que acontece caso um parâmetro defina o formato dos dados que o relatório precisa ser criado?

```
class Relatorio
  def imprime(formato)
    if formato == :texto
      puts "*** Dados do restaurante ***"
    elsif formato == :html
      puts "<html>Dados do restaurante</html>"
    else
      puts "formato desconhecido!"
  end
end
```

Para solucionar esse problema de forma orientada a objetos poderíamos criar uma classe abstrata que define o comportamento de um relatório, ou seja, que define que um relatório de ter um head, um body, um footer, etc. Mas como criar uma classe abstrata em Ruby ? Embora não exista a palavra chave reservada "abstract" o conceito permanece presente na linguagem. Vejamos:

```
class Relatorio
  def imprime
    imprime_cabecalho
    imprime_conteudo
  end
end
```

A classe Relatorio agora possui os métodos que definem um relatório. Obviamente esses métodos podem ser invocados. A solução para isso normalmente é lançar uma exception nesses métodos não implementados. Agora que temos nossa classe abstrata, podemos criar subclasses de Relatorio que contém a implementação de cada um dos tipos, por exemplo para HTML:

```
class HTMLRelatorio < Relatorio
  def imprime_cabecalho
    puts "<html>"
  end

  def imprime_conteudo
    puts "Dados do relatorio"
  end
end

class TextoRelatorio < Relatorio
  def imprime_cabecalho
    puts "***"
  end

  def imprime_conteudo
    puts "Dados do relatorio"
  end
end
```

Agora para usar nossos relatórios podemos fazer:

```
relatorio = HTMLRelatorio.new
relatorio.imprime

relatorio = TextoRelatorio.new
relatorio.imprime
```

18.4 – Exercícios: Template Method

1. Crie a classe que define as obrigações de um relatório:

```
class Relatorio
  def imprime
    imprime_cabecalho
    imprime_conteudo
  end
end
```

2. Para termos implementações diferentes, crie um relatório HTML e um de texto puro:

```
class HTMLRelatorio < Relatorio
  def imprime_cabecalho
    puts "<html>"
  end

  def imprime_conteudo
    puts "Dados do relatorio"
  end
end

class TextoRelatorio < Relatorio
  def imprime_cabecalho
    puts "***"
  end

  def imprime_conteudo
    puts "Dados do relatorio"
  end
end
```

3. Utilize os relatórios com a chamada ao método imprime:

```
relatorio = HTMLRelatorio.new
relatorio.imprime

relatorio = TextoRelatorio.new
relatorio.imprime
```

18.5 – Design Patterns: Observer

Como executar operações caso algo ocorra no sistema. Temos uma classe Restaurante e queremos manter avisados todos os objetos do sistema que se interessem em modificações nele. Nossa classe Franquia precisa executar o método **alerta** no caso

```
class Franquia

  def alerta
    puts "Um restaurante foi qualificado"
  end

end

class Restaurante

  def qualifica(nota)
    puts "Restaurante recebeu nota #{nota}"
  end
```

```
end

restaurante = Restaurante.new
restaurante.qualifica(10)
```

Poderíamos chamar o método **alerta** direto ao final do método **qualifica** ou poderíamos passar como parametro do método um observer para rodar. Porém essas maneiras acoplariam muito nosso código, por exemplo, não funcionariam para alertar mais de um observer.

Uma saída é criar uma lista de observadores e executá-los ao final da operação. Para isso podemos ter um método que permite adicionar quantos objetos forem necessários serem alertados.

```
class Restaurante

  def initialize
    @observers = []
  end

  def adiciona_observer(observer)
    @observer << observer
  end

  def notifica
    # percorre todos os observers chamando o método alerta
  end

  # qualifica

end

restaurante = Restaurante.new
restaurante.qualifica(10)
```

Poder ser observado não é uma característica única de um objeto. Podemos utilizar essa estratégia de avisar componentes por todo o software. Por que não colocar o comportamento em uma classe responsável por isso. Utilizando herança tornaríamos nosso código **observável**.

Ao invés de utilizar a herança novamente podemos utilizar os módulos para isso. Todo comportamento que faz do objeto ser um **observer** será colocado nesse módulo.

```
module Observer
  def initialize
     @observers = []
   end

   def adiciona_observer(observer)
     @observer << observer
```

```
    end

    def notifica
      # percorre todos os observers chamando o método alerta
    end
end

class Restaurante
  include Observer
  def qualifica(nota)
    puts "Restaurante recebeu nota #{nota}"
    notifica
  end
end
```

Modulo Observer

O módulo acima já existe em ruby e é chamado de Observer. http://ruby-doc.org/core/classes/Observable.html

18.6 - Desafio: Design Pattern - Observer

1. Crie um novo arquivo e implemente o Design Pattern Observer para Restaurante e Franquia.

CAPÍTULO ANTERIOR:

Apêndice: Rotas e Rack

PRÓXIMO CAPÍTULO:

[Apêndice: Integrando Java e Ruby]

Você encontra a Caelum também em:

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter