UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO

ESCOLA DE INFORMÁTICA APLICADA

CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

RAILS VERSUS STRUTS: UM COMPARATIVO DE FRAMEWORKS

SYLVESTRE MERGULHÃO DA CUNHA NETO

ORIENTADOR: MÁRCIO BARROS

MARÇO / 2007

RAILS VERSUS STRUTS: UM COMPARATIVO DE FRAMEWORKS

Projeto de Graduação apresentado à Escola
de Informática Aplicada da Universidade
Federal do Estado do Rio de Janeiro
(UNIRIO) para obtenção do título de
Bacharel em Sistemas de Informação

SYLVESTRE MERGULHÃO DA CUNHA NETO

ORIENTADOR: MÁRCIO BARROS

Sumário

1.Introdução

1.1.Motivação

Ruby é uma linguagem de programação que a princípio desperta pouco interesse. Afinal, para que aprender mais uma linguagem de programação se ela não apresenta nenhum bom atrativo?

Na pesquisa constante por linguagens, ferramentas e métodos de desenvolvimento deparei-me com um novo framework de desenvolvimento de aplicações para web chamado Ruby on Rails. Framework escrito na linguagem que não oferecia antes atrativo qualquer. O framework se dizia o "amigo do programador", afinal ele, assim como os demais, deveria existir para minimizar o trabalho do desenvolvedor e não o contrário. Utilizando conceitos antigos e já consolidados, mas muitas vezes esquecidos pelos desenvolvedores, o Rails mostrou que valeria a pena perder algumas horas para estudá-lo e verificar se cumpria o que prometia.

Um desses antigos conceitos é chamado "Don't Repeat Yourself", DRY na sigla em inglês, que quer dizer algo como "Não se repita". A idéia principal do DRY é de que informações não devem estar duplicadas. As duplicações dificultam mudanças, diminuem a clareza e dão margem a inconsistência. O conceito é anterior, mas ganhou este nome após a publicação do livro "The Pragmatic Programmer" (HUNT e THOMAS, 1999). Quando o princípio DRY é aplicado, a modificação de um elemento em um sistema não afeta outro logicamente separado e, adicionalmente, a modificação de elementos que estão relacionados implica na mudança para todos os envolvidos, mantendo-os em sincronia.

Outro conceito importante é chamado KISS, que pode ser "Keep it sweet and simple" ou no modo mais conhecido "Keep it simple, stupid!". A idéia que esse conceito passa é de sempre fazer as coisas do modo mais simples possível, tanto em relação às funcionalidades do que se está desenvolvendo, quanto em relação ao modo como está sendo desenvolvido. As aplicações de console dos Unix no geral usam esse conceito. São pequenas aplicações que servem para fazer pequenas coisas, mas fazer bem. É a separação das tarefas complexas em tarefas menores, com escopo reduzido, que podem ser reutizadas em diversas situações, ou seja: uma modularização, que nos leva de volta ao conceito DRY. Numa citação supostamente atribuída à Einstein é dito: "Tudo deve ser feito o mais simples possível, mas não simploriamente".

Por último, o Rails trata de um conceito que prega que o desenvolvedor apenas precisa definir a configuração daquilo que não é convencional. O Rails possui uma série de convenções. Por exemplo, uma classe de modelo *Client* possui sua tabela no banco nomeada *clients*. Se uma nova aplicação for desenvolvida, pode-se fazê-la sem precisar de nenhuma linha de configuração – excluindo-se a configuração de senhas do banco de dados. Caso necessite, é possível customizar o nome da tabela da qual a classe se refere, útil para bancos de dados de aplicações legadas.

Após leitura sobre a teoria e os conceitos por trás do Ruby on Rails, partir para algo mais prático era o objetivo. Neste sentido, foi encontrado um tutorial que ensinava os passos para desenvolver uma simples aplicação CRUD (create, retrieve, update and delete) já utilizando um sistema de validação de dados (FERRAZ, 2006). Em menos de uma hora foi instalado tudo que era necessário – a linguagem, as bibliotecas, o conector para o banco de dados Postgresql e o framework em si. Pouco mais de uma hora seguindo os passos do tutorial, desenvolvendo a aplicação e ela estava pronta. Antes disso, nunca havia tido contato com a linguagem Ruby, mostrando-se como, de fato, a linguagem é intuitiva e, no caso do Rails, como o framework facilitou o trabalho.

É inevitável para um novo framework que ele mostre o que tem de melhor ou de inovador em comparação aos que já existem disponíveis. Para que a migração ou utilização de algo diferente se este não traz melhoria alguma? Um dos grandes estudos encontrados em diversos sites e blogs na internet trata da comparação com a linguagem e o framework mais utilizados atualmente para desenvolver aplicações corporativas: a linguagem Java e o consolidado framework Struts. O ponto mais discutido na maioria desses estudos trata do número de linhas de código necessárias para desenvolver uma aplicação.

Por fim, não foram encontrados estudos que mostrassem avaliações mais profundas, comparando também a nível de funcionalidade e de interpretação dos *design patterns*, que confrontassem tanto as linguagens quanto os frameworks a fim de se conseguir uma visão macro das reais diferenças.

1.2.Objetivo do Trabalho

Após conhecer mais profundamente a linguagem Ruby, entende-se porque ela foi escolhida para o desenvolvimento do Rails. É uma linguagem dinâmica que possui a soma de funcionalidades presentes separadamente em diversas linguagens.

Ao longo do trabalho serão avaliados a linguagem Ruby e o framework Rails comparando-os com a linguagem Java e com o framework Struts. A escolha foi feita devido a linguagem Java e o framework Struts serem muito utilizados atualmente no meio comercial para desenvolvimento de aplicações. Será comparado desde os conceitos básicos de cada linguagem até a implementação dos *design patterns* em cada um dos frameworks.

1.3.Estrutura do Texto

Primeiramente, serão apresentadas as principais características e funcionalidades da linguagem Ruby. Características estas que, por serem dinâmicas, permitiram o desenvolvimento de um framework também dinâmico, sendo facilmente extensível através de plugins e mostrando que agilidade no desenvolvimento não significa pouca qualidade ou falta de documentação. Em algumas situações serão comparadas características da linguagem Java com linguagem Ruby para orientação quanto a forma de desenvolvimento.

Na segunda parte do capítulo dois, será apresentado um estudo mais profundo do framework Rails, abordando cada um dos seus três principais módulos. O Active Record é o módulo que trata do acesso ao banco de dados, possuindo uma camada ORM – *Object-relational mapping*, ou mapeamento objeto-relacional, em português – que trata de transformar os dados das tabelas do banco em objetos Ruby. O Action Controller é o centro nervoso da aplicação Rails e orquestra todos os componentes trabalhando as informações enviadas pelo Active Record e enviando-as para a visão: o Active View. Esse é o responsável por apresentar as informações para o requisitante.

Para este trabalho foram desenvolvidas duas aplicações que utilizam a mesma base de dados, uma implementada em Java, com frameworks Struts e Hibernate, e outra em Ruby, com framework Rails. Trata-se de uma aplicação de comércio eletrônico simplificada, desenvolvida apenas para avaliação e comparação entre as linguagens e os frameworks. Ao longo do capítulo três há comparações de diversos níveis em relação as duas aplicações que foram desenvolvidas. São apresentadas as pequenas diferenças na interface entre as duas versões do sistema e as grandes diferenças no modo em como foram implementadas. No fim do capítulo temos um comparativo numérico de linhas de código de programação e configuração e de quantidade de arquivos em cada aplicação.

2. Estudo do framework Rails

Nesse capítulo será apresentada a linguagem Ruby (MATSUMOTO, 2007), um pouco de sua história, filosofia e estilo de desenvolvimento. Serão apresentadas informações introdutórias sobre o modo como foi idealizado, seu modelo de orientação a objetos, atributos e classes.

Na segunda parte trataremos do framework Rails (HANSSON, 2007). Serão apresentados os principais módulos que compõem o framework, assim como sua forma de utilização, acesso a banco de dados, controladores, uso de sessão e cookies e o modelo de visão.

Na última parte expõe-se o básico sobre os métodos de *deployment*, um comparativo de servidores para uso em produção e escalabilidade.

2.1. Linguagem Ruby

2.1.1. História

A linguagem Ruby foi criada por Yukihiro "Matz" Matsumoto, tendo iniciado seu trabalho em fevereiro de 1993, mas somente liberando sua primeira versão pública no ano de 1995. O nome Ruby foi dado devido ao presente de aniversário de um amigo.

Yukihiro Matsumoto, nascido no japão em 14 de abril de 1965, é um cientista da computação e desenvolvedor de software livre, mundo em que ficou conhecido após a popularização do Ruby. Matsumoto foi um programador autodidata durante todo o colégio. Graduou-se em ciência da informação pela Universidade de Tsukuba, onde se associou a departamentos encarregados de pesquisas com linguagens de programação e compiladores.

Atualmente, Matsumoto é o chefe do departamento de pesquisas e desenvolvimento no Network Applied Communication Laboratory, na prefeitura de Shimane.

2.1.2. Filosofia

O primeiro desejo de Matz com o Ruby era de tornar os programadores mais felizes, reduzindo o trabalho manual que precisasse ser feito. Segundo ele, o desenvolvimento de

sistemas deveria enfatizar as necessidades do homem e não da máquina:

"Muitas pessoas, especialmente engenheiros de computação, focam nas máquinas. Eles pensam, "Fazendo isso, a máquina será mais rápida. Fazendo isso, a máquina será mais eficiente. Fazendo isso, a máquina irá fazer determinada coisa melhor". Eles estão focando nas máquinas. Mas de fato nós precisamos focar nos humanos, em como os humanos lidam com programação ou operação das aplicações das máquinas. Nós somos os mestres. Elas são as escravas." , disse Matz (VENNERS, 2003).

Ruby segue o princípio da mínima ou menor surpresa (POLS, na sigla em inglês), significando que a linguagem se comporta intuitivamente ou como o programador imagina que deva ser. Essa caracterização não é originária de Matz e, em geral, Ruby seria algo mais próximo "da menor surpresa na cabeça Matz". Contudo, muitos programadores também acharam-na similar ao seu próprio modelo mental.

Matz afirmou (VENNERS, 2003): "Todos têm antecedentes individuais. Alguém pode vir do Python, outro pode vir do Perl, e eles podem se surpreender por diferentes aspectos da linguagem. Eles vêm até mim e dizem: "Eu estou surpreso com esse aspecto da linguagem, então o Ruby viola o princípio da mínima surpresa". Mas espere. Espere. O princípio da mínima surpresa não é somente para você. O princípio da mínima surpresa quer dizer da minha mínima surpresa. E isso significa o princípio da mínima surpresa depois de você programar em Ruby muito bem. Por exemplo, eu fui um programador C++ antes de iniciar a desenvolver o Ruby. Eu programei em C++ exclusivamente por dois ou três anos. E, mesmo depois desses anos programando em C++, ela ainda me surpreendia."

2.1.3.Semântica

Ruby é uma linguagem orientada a objetos, ou seja, qualquer variável é um objeto, mesmo classes e tipos que em muitas outras linguagens são designadas como primitivos. Toda função é um método. Variáveis sempre são referências para os objetos e não os objetos propriamente ditos. Ruby suporta herança, mixins (ver seção 2.1.6) e singletons. Embora Ruby não suporte herança múltipla, classes podem importar módulos como *mixins*. É possível usar sintaxe procedural, mas tudo que seja feito proceduralmente é de fato realizado por uma instância de um objeto chamado *main*. Como esta classe é mãe de qualquer outra classe, as mudanças ficam visíveis a todas as classes e objetos.

Ruby tem sido descrita como uma linguagem de diversos paradigmas: ela lhe

permite programar proceduralmente (definindo funções e variáveis fora de classes faz delas parte de Object), bem como programar com orientação a objetos (tudo é um objeto) ou funcionalmente (suporta funções anônimas e blocos de código, além das funções retornarem o último valor computado). Suporta threads e possui tipificação forte já que qualquer variável é uma instância de uma classe, porém ainda assim é dinâmica. Isto é, uma variável inicia o programa pertencendo a uma determinada classe, por exemplo *Float*, mas ao longo do código, pode sofrer um *cast* para outra classe e terminar sendo uma instância da classe *String* (ver exemplo na seção 2.1.4).

De acordo com o Ruby FAQ (BLACK e FOWLER, 2007), "Se você gosta de Perl, você irá gostar de Ruby e se sentirá em casa com sua sintaxe. Se você gosta de Smalltalk, você irá gostar de Ruby e se sentirá em casa com sua semântica. Se você gosta de Python, você pode ou não gostar devido a grande diferença na filosofia de desenvolvimento entre Python e Ruby/Perl."

2.1.4.Orientação a objetos

Tudo que é manipulado em Ruby é um objeto. Não há tipos primitivos como na maioria das linguagens. Essa abordagem se assemelha a da linguagem Smalltalk. Pode-se fazer:

```
puts "alguma coisa".upcase # imprimirá "ALGUMA COISA"
```

Estar-se-á aplicando a função `upcase()` da classe String no objeto criado por "alguma coisa". Outras classes comuns do Ruby são:

- Fixnum – representa inteiros com até o tamanho da palavra binária do processador menos 1 bit. Exemplos: 81, 6589, 2, 99, 100, ...
- Bignum – representa inteiros maiores que os de Fixnum: 46546545788974432, 1853020188851841, 34336838202925124846578490892 81;
- Float – representa números decimais: 1.29, 5.0, entre outros;
- Range – representa intervalo de valores como 1..10 ou b..e;
- Expressão regular – representa uma expressão regular como: /a/ ou /^\s*[a-z]/ .

Isso prova que qualquer variavel é de fato uma instância de uma classe, ou seja, um objeto. Os operadores de adição, subtração e multiplicação, por exemplo, também são

métodos em Ruby. Então Ruby não possui "operadores", no sentido estrito do termo, mas apenas métodos. Um método de soma, pode ser escrito:

```
class Numero < Fixnum
  def +(numero)
    71
  end
end
```

Nesta implementação, o método `+()` da classe `Numero` que herdou da classe `Fixnum` foi sobrescrito e retornará sempre 71.

Como dito anteriormente, Ruby possui tipificação dinâmica. Isso quer dizer que uma instância de um objeto pode sofrer um *cast* para outro objeto em tempo de execução, de forma transparente para o programador. Abaixo, apresenta-se um exemplo de código que provoca essa conversão.

```
num = 81
6.times do
  puts "#{num.class}: #{num}"
  num *= num
end
```

O resultado da execução esse código é apresentado a seguir.

```
Fixnum: 81
Fixnum: 6561
Fixnum: 43046721
Bignum: 1853020188851841
Bignum: 3433683820292512484657849089281
Bignum: 1179018457773858317152087286141251866567821159227584110909696l
```

Nesse exemplo a variável `num`, inicializada com valor 81, foi multiplicada por si mesmo 6 vezes. Quando o valor de `num` atingiu o valor máximo suportado por um objeto do tipo `Fixnum`, automaticamente ocorreu um *cast* para `Bignum`, que suporta valores de maior grandeza.

2.1.4.1.Atributos

Em Ruby não é necessário escrever métodos *getters* e *setters*. Para isso existem os *attribute helpers*, que deixam claro o conceito de facilidade do Ruby. Sem o uso desses atalhos seria necessário o seguinte código para uma classe Carro, por exemplo:

```
class Carro
  def modelo # getter
    @modelo
  end

  def modelo=(valor) # setter - operador de atribuição
    @modelo = valor
  end

  def fabricante
    @fabricante
  end

  def fabricante=(valor)
    @fabricante = valor
  end
end
```

Com uso dos atalhos, o código fica reduzido, conforme apresentado a seguir.

```
class Carro
  attr_accessor :modelo, :fabricante
end
```

Em termos de escopo das variáveis, temos as seguintes situações:

- Variáveis locais – somente existirão dentro do contexto em que foram declaradas, como um método ou um *while*, por exemplo.

```
local_var = 1
```

- Variáveis de instância – são as variáveis de instância de uma classe.

```
@instan_var = 1
```

- Variáveis de classe – manterão o mesmo valor para qualquer instância da classe em que estão definidas.

```
@@class_var = 1
```

- Variáveis globais – estarão disponíveis para toda a aplicação em qualquer situação.

```
$global_var = 1
```

2.1.4.2.Controle de acesso

Em Ruby, assim como em Java, pode-se usar três tipos de acesso aos métodos: públicos, privados e protegidos. Os métodos públicos podem ser acessados de qualquer classe e por qualquer instância de objeto. Os métodos privados podem ser acessados somente por objetos da mesma classe. Métodos protegidos são métodos que podem ser acessados por qualquer objeto dentro da mesma estrutura hierárquica da classe. Por exemplo, uma classe `Pessoa` que herda de `SerHumano` pode acessar métodos protegidos de `SerHumano`.

Na maioria das linguagens o controle de acesso aos métodos é especificado no cabeçalho de cada método. Em Ruby essa especificação se dá por meio de blocos, como exemplificado abaixo.

```
class Carro
  attr_accessor :modelo, :fabricante

  def metodoPublico
  end

  protected
  def metodoProtegido
  end
```

```
  def outroMetodoProtegido
  end

  private
  def metodoPrivado
  end

  public
  def deNovoMetodoPublico
  end
end
```

Também pode-se usar um atalho para definir a visibilidade dos método, como exemplificado abaixo.

```
class Carro
  attr_accessor :modelo, :fabricante

  def metodoPublico
  end

  def metodoProtegido
  end

  def outroMetodoProtegido
  end

  def metodoPrivado
  end

  def deNovoMetodoPublico
  end

  public :metodoPublico, :deNovoMetodoPublico
  protected :metodoProtegido, :outroMetodoProtegido
  private :metodoPrivado
end
```

2.1.5.Módulos

Módulos são semelhantes a classes, possuindo uma coleção de métodos, constantes, outros módulos e definições de classes. Mas, diferentemente das classes, não se pode criar uma instância de um módulo.

Módulos têm dois propósitos. Primeiro de fazer o papel de *namespace*, permitindo criar métodos que não colidirão devido a nomes iguais. Segundo, de permitir compartilhar funcionalidades entre classes. Uma classe permite incluir um módulo, deixando todos os métodos, constantes e variáveis do módulo disponíveis para a classe como se eles tivessem sido definidos na mesma. Diversas classes podem incluir o mesmo módulo, compartilhando as funcionalidades do módulo sem usar herença. Também é possível uma mesma classe incluir diversos módulos. A utilização dessas inclusões é denominada *mixin* e é descrita na próxima seção.

2.1.6.Mixins

Ruby, assim como outras linguagens OO, não suporta herança múltipla, mas permite adicionar comportamentos a uma classe utilizando os *mixins*. Eles funcionam como os *includes*: é declarada a classe ou módulo que deseja herdar o comportamento, conforme indicado abaixo.

```
class MeuObjeto < ObjetoPai
  include Debug
  include Log
end
```

A classe `MeuObjeto` está herdando de `ObjetoPai`, mas "incluindo" o comportamento das classes `Debug` e `Log`. Isso significa que todos os métodos, contantes e variáveis definidos em `Debug` e `Log` estarão disponíveis para as instâncias da classe `MeuObjeto`. Os conflitos nesse nível são tratados da seguinte maneira: se dois includes possuem métodos com mesmo nome, prevalece aquele que foi incluído por último.

2.1.7.Blocos de código

Qualquer código entre chaves é um bloco. Blocos também podem estar entre as tags `do` e `end`. A seguir, dois exemplos, um de cada tipo de blobo.

```
3.times { puts "Ruby rulez!" } # Irá imprimir 3 vezes
                               # "Ruby rulez!"
```

ou

```
3.times do # Irá imprimir 6 vezes "Ruby rulez!"
  puts "Ruby rulez!"
  puts "Ruby rulez!"
end
```

Por convenção, são usadas as chaves para códigos com apenas uma linha e as tags `do` e `end` naqueles em que exista mais de uma linha. Blocos de código podem ser passados através de parâmetro para funções, servindo como callbacks, por exemplo.

2.1.8.Tratamento de exceções

Ruby possui tratamento de exceções, mas diferentemente do Java, ele não exige que toda situação que pode gerar uma exceção seja tratada. As palavras chaves são: begin (equivalente ao try do Java), rescue (catch), ensure (finally) e retry - que permite a re-execução do código a partir do begin. Como exemplo, segue trecho de código da biblioteca "net/smtp", escrito por Minero Aoki:

```
@esmtp = true
begin
  # First try an extended login. If it fails because the
  # server doesn't support it, fall back to a normal
login
  if @esmtp then
    @command.ehlo(helodom)
  else
    @command.helo(helodom)
  end
rescue ProtocolError
  if @esmtp then
    @esmtp = false
    retry
  else
    raise
  end
end
```

2.2.O Framework Rails

Rails trata de indivíduos e interações. Não existem conjuntos de ferramentas, configurações complexas ou processos elaborados. Isto leva a uma transparência em que o que os desenvolvedores fazem se reflete imediatamente no que o cliente vê. Trata-se de um processo intrinsecamente interativo.

O Rails não renega a documentação: ele a torna trivial e facilita a criação de documentação em HTML para todo o seu código. No entanto, o processo de desenvolvimento em Rails não é direcionado por documentos. Não se encontrará especificações de 500 páginas em um projeto em Rails.

Rails oferece a capacidade de resposta rápida a mudanças. O sólido, quase obsessivo, caminho que norteia o Rails e honra o princípio DRY faz com que as mudanças em aplicativos Rails impactem pouco em alteração de código ao contrário do que acontece em diversos outros frameworks. E como os aplicativos Rails são escritos em Ruby, onde o conceito pode ser expresso de forma concisa e precisa, as mudanças tendem a ser localizadas e fáceis de serem escritas. A principal ênfase na unidade e teste funcional, juntamente com o suporte para testes, fixtures (ver seção 2.2.1.1) e mock objects, dão aos desenvolvedores a rede de segurança que precisam ao realizarem tais mudanças. Com uma boa bateria de testes realizada, as mudanças são menos desgastantes e estressantes.

Rails é um framework MVC. Rails disponibiliza ao desenvolvedor uma estrutura onde serão desenvolvidos os modelos, visões e controladores. Ele faz a conexão entre todos estes componentes conforme o programa é executado. Uma das vantagens do Rails é de que esse processo de conectar “as partes” é baseado na padronização dos componentes, de forma que tipicamente não se necessite escrever qualquer meta-dado de configuração externa a fim de que o sistema funcione corretamente. Este é um exemplo da filosofia do Rails em relação ao favorecimento da convenção em detrimento da configuração.

2.2.1.Rails por dentro

Um dos aspectos interessantes do Rails é a sua componentização. O desenvolvedor gasta a maior parte do tempo tratando de elementos de alto nível como Active Record e Action View, mas existe um componente importante, abaixo de tudo, chamado Rails, que orquestra o que todos os outros componentes fazem e trata para que trabalhem harmoniosamente. Ao mesmo tempo, somente uma pequena porção dessa infraestrutura é

relevante para os desenvolvedores no seu dia-a-dia.

2.2.1.1.Estrutura de diretórios

O Rails assume um padrão para estrutura de diretórios da aplicação. Abaixo segue a estrutura do primeiro nível de diretórios criados ao rodar o comando `rails minha_aplicacao`.

```
minha_aplicação/
  /app         - arquivos dos modelos, visões e controles
  /components  - componentes reutilizáveis
  /config      - arquivos de configuração, como de banco
                 de dados, por exemplo
  /db          - arquivos do schema do banco de dados
  /doc         - Documentação gerada automaticamente
  /lib         - código compartilhado
  /log         - arquivos de log gerados pela aplicação
  /public      - o diretório que é acessível pela web. Da
                 visão do navegador parece que toda a
                 aplicação roda daqui
  /scripts     - conjunto de scripts utilitários
  /test        - local para os testes de unidade,
                 funcionais, mock objects e fixtures
  /vendor      - códigos de terceiros, como plugins
```

A maior parte do trabalho fica no diretorio app, onde fica o código principal da aplicação. É apresentado abaixo sua estrutura interna.

```
app/
  /controllers - onde são colocados os controladores
  /helpers     - onde são colocados os helpers
                 ver seção 2.2.2.3
  /models      - onde são colocados os modelos
  /views       - onde são colocados os arquivos de visão
                 e templates
```

O diretório *doc* é usado para a documentação da aplicação, produzida usando RDoc, o sistema de documentação de código do Ruby.

Os diretórios *lib* e *vendor* tem propósitos similares. Os dois são repositórios de código que são usados pela aplicação, mas que não pertencem exclusivamente a ela. O diretório *lib* é usado para colocar código próprio, enquanto que o diretório *vendor* é usado

para colocar código de terceiros.

O Rails gera seus logs de tempo de execução e coloca-os dentro do diretório *log*. Dentro desse diretório se encontrará um arquivo de log para cada ambiente que o Rails roda: desenvolvimento, teste e produção. Os logs gerados possuem estatísticas de tempo, informações sobre cache e acesso ao banco de dados.

O diretório *public* é a interface externa da aplicação. É ele que o servidor web usa como diretório base da aplicação. Lá se localizam as imagens estáticas da aplicação, além das blibliotecas de javascript e arquivos de *cascading style sheets*.

O diretório *scripts* fica com programas úteis ao desenvolvedor, como o `generate`, que serve para geração de controladores e modelos; o `server` que é um servidor web para desenvolvimento; entre outros.

No diretório *test* um destaque para os *fixtures*. Fixture é uma funcionalidade do Rails que permite a população da base de desenvolvimento com dados de exemplo antes dos testes serem executados. Os dados são pré-definidos em arquivos com padrão YAML (INGERSON, EVANS e BEN-KIKI, 2007) ou CSV e são independentes do banco de dados que está sendo utilizado.

2.2.2.Os módulos básicos

2.2.2.1.Active Record

O Active Record conecta objetos de negócio com tabelas do banco de dados para criar um modelo de domínio onde lógica e dados se encontram presentes em um conjunto. Trata-se de uma implementação de um padrão de mapeamento objeto-relacional (ORM).

Ele se diferencia da maioria de outros frameworks ORM no modo como é configurado, utilizando a convenção. Com isto, o Active Record minimiza a quantidade de configuração que os desenvolvedores precisam fazer.

Frameworks ORM mapeiam das tabelas do banco de dados às classes. Se um banco de dados possui uma tabela chamada `orders`, nosso progama terá uma classe chamada `Order`. Cada linha desta tabela corresponde a um objeto da classe `Order`. Atributos da classe representarão as colunas da tabela `orders`, com seus mesmos nomes, por padrão.

Somado a isso, as classes Rails que envolvem as tabelas de banco de dados provêem

uma disposição de métodos de classe que realizam operações de tabela. Por exemplo, para buscar um `order` com um id em particular. Implementando-se isto como um método de classe que retorne o objeto `Order` correspondente, em um código Ruby, ficaria como a seguir.

```
order = Order.find(1)
```

Adicionalmente pode ser feito:

```
order = Order.find(1)
order.discount = 0.5
order.save
```

Assim, uma camada ORM mapeará desde as tabelas até as classes, sendo as linhas os objetos e as colunas os atributos de tais objetos. Os métodos de classe são usados a fim de se realizar uma operação de tabela, enquanto os métodos de instância realizam operações nas linhas individuais.

2.2.2.2.Action Controller

O controlador no Rails é o centro nervoso da aplicação. Ele coordena a interação entre o usuário, as visões e o modelo. O Rails Action Controller faz a maior parte do trabalho e o código que precisa ser escrito para cada aplicação se concentra nas funcionalidades da mesma. Isso faz com que os controladores fiquem limpos, fáceis de desenvolver e manter. Algumas das funcionalidades do Action Controller são citadas a seguir.

- Rotear URLs para ações internas dos controladores: ele gerencia URLs de fácil leitura para as pessoas.

  Em sua simplicidade, um aplicativo web aceita um pedido vindo de um navegador, processa-o e envia uma resposta. Como um aplicativo sabe o que fazer com um pedido vindo do navegador? Um aplicativo de compras receberá pedidos de exibição de um catálogo, adicionar itens ao carrinho, e assim por diante. Como direcionar esses pedidos ao código apropriado?

  O Rails codifica esta informação na URL pedida e usa um subsistema de roteamento

a fim de determinar o que deve ser feito com o pedido. O processo em si é bastante flexível: o Rails deve determinar o nome do controlador que cuidará desse pedido em particular, juntamente com uma lista de quaisquer outros parâmetros pedidos. Geralmente um desses parâmetros adicionais identifica a ação a ser invocada no controlador de destino.

Por exemplo, um pedido feito em nosso aplicativo de compras pode parecer com http://gulashop.com/product/show/123. Isto é interpretado pelo aplicativo como um pedido invocando a ação `show()` do `ProductController`, tendo requisitado isso, a aplicação mostra os detalhes do produto em questão com a id 123.

Não é necessário que se utilize o padrão "controller/action/id" de URL. Um aplicativo de blog pode ser configurado para que as datas dos artigos possam fazer parte das URLs. Requisitando-o com http://gulablog.com/blog/2006/11/08, por exemplo, e poderá requisitar a ação `display()` do `ArticleController` para mostrar os artigos de 8 de novembro de 2006.

- Definir os Action Methods: quando um controlador processa um pedido, ele procura um método público com o mesmo nome da ação de entrada. Se encontrar, este método é requisitado. Caso não encontre, mas o controlador implemente um método `method_missing()`, este método é chamado, passando o nome da ação como o primeiro parâmetro e uma lista vazia de argumentos como o segundo. Se nenhum método puder ser chamado, o controlador procura um template que tenha o mesmo nome da ação requisitada. Se este template for encontrado, ele será diretamente renderizado. No caso de nada disto ocorrer um erro `Unknown Action` será gerado.

  Por padrão, qualquer método público em um controlador pode ser chamado como uma ação. Pode-se previnir que métodos em particular sejam acessíveis tornando-os privados ou ainda usando o método `hide_action()`. O código a seguir exemplifica a proteção de um método.

```
class MeuBlogController < ActiveRecord::Base
  def create_post
    post = Post.new(params[:post])
    if some_verify(post)
      post.save
    end
  end
```

```
  hide_action :some_verify
  def some_verify(post)
    # código
  end
end
```

- Responder ao usuário: parte do trabalho dos controladores é responder ao usuário. Existem basicamente três modos de fazer esse trabalho:
    - A mais comum é renderizar um template, o "V" do MVC, usando informações providas pelo controlador;
    - O controlador pode retornar diretamente uma resposta ao navegador, sem invocar um template, o que raramente é usado ;
    - O controlador pode enviar outro tipo de dado qualquer que não HTML, como um arquivo executável, uma imagem ou um PDF.

  Um template é um arquivo que define o conteúdo da resposta para a aplicação. Rails suporta dois tipos de templates por padrão: rhtml, que é HTML embutido com código Ruby, e o chamado "builder", uma maneira de escrever conteúdo em forma de programação, útil para geração de XML, por exemplo, onde não é necessário se preocupar em como é formatado o XML, mas com o conteúdo em si. Por convenção, o template para uma ação `action` de um controlador `control` deverá se localizar em app/views/control/action.rhtml ou app/views/control/action.rxml.

- Gerenciar os cookies e sessões dos usuários: Rails abstrai o uso dos cookies através de uma interface simples. O controlador possui um atributo chamado `cookies`, que é um hash com os elementos dos cookies. Quando uma requisição é recebida, o objeto do cookie é inicializado com os nomes e valores enviados pelo browser para a aplicação. A qualquer momento, a aplicação pode adicionar e remover chave/valor do objeto de cookies. Esse novo objeto `cookies` será enviado de volta ao navegador na resposta. Esses novos valores dos cookies estarão disponíveis nas requisições seguintes.

  Em seguida, o exemplo de um controlador Rails que armazena em um `cookie` a data e hora atuais e em seguida redireciona para outra ação. Isso faz o browser trocar de URL, chamando a outra ação que faz a captura do valor do `cookie` e exibe.

```
class HoraComCookiesController < ApplicationController
  def gravar_cookie
    cookies[:hora] = Time.now.to_s
    redirect_to :action => "ler_cookie"
  end
  def ler_cookie
    cookie_value = cookies[:hora]
    render(:text => "O cookie contém:#{cookie_value}")
  end
end
```

Uma sessão em Rails é um hash que persiste pelos *requests*. Diferentemente dos cookies, as sessões podem gravar qualquer tipo de objeto que pode ser serializado. Para reconhecer as sessões, o Rails gera uma chave de 32 caracteres hexadecimais, que é salva em um cookie e referenciada no servidor com os objetos salvos. O uso da sessão se dá utilizando o objeto `session`, como mostrado abaixo:

```
class HoraCertaSessaoController <
ApplicationController
  def gravar_na_sessao
    session[:hora] = Time.now
    redirect_to :action => "ler_da_sessao"
  end
  def ler_da_sessao
    session_value = session[:hora]
    render(:text => "O session[:hora] contém:
                                    #{session_value}")
  end
end
```

Repare que na linha onde é gravada a hora no objeto da sessão não há o uso da função `to_s()` já que a sessão permite gravar diretamente o objeto do tipo `Time` que é serializável, diferentemente dos cookies que somente aceitam strings.

Um outro uso da sessão no Ruby se dá no objeto especial `flash`. Esse objeto é usado para comunicação entre ações. Quando é usada a função `redirect_to()` para transferir o controle de uma para outra ação, o navegador gera uma nova requisição. Essa nova requisição é iniciada em uma nova instância do controlador que não possui as mesmas instâncias de variáveis que estavam disponíveis na ação original. Mas algumas vezes é interessante a comunicação entre duas instâncias diferentes. O objeto flash permite facilmente essa comunicação. Ele também é

organizado como um hash e permite a gravação de qualquer tipo de objeto serializável, já que também é gravado na sessão, mas possui a característica especial de somente permanecer gravado nela por um *request*. O exemplo abaixo ilustra com clareza.

```
class Admin::CategoryController <
Admin::AdminController
  def list
    @categories = Category.find(:all)
  end

  def create
    @category = Category.new(params[:category])
    if @category.save
      flash[:notice] = "Salvo com sucesso!"
    else
      flash[:notice] = "Ocorreu erro ao salvar!"
    end
    redirect_to :action => 'list'
  end
end
```

O controlador `CategoryController` acima possui dois métodos: um para salvar e outro para listar. Ao salvar uma nova categoria na ação `create`, uma string será salva na chave `notice` da flash. Essa string estará disponível para uso dentro da ação `list` e da view relativa que, em seguida exibirá a mensagem na tela. Num próximo request a mensagem já não estará mais disponível.

- Gerenciar os filtros e verificações: filtros permitem executar um método qualquer antes ou depois de uma ação de um controlador. Esta é uma funcionalidade poderosa, que pode ser usada para fazer sistemas de autenticação, logger, entre outros. Em seguida um simples exemplo da implementação de um `before_filter`.

```
class Admin::CategoryController < Admin::AdminController
  before_filter :authorize

  def list
    @categories = Category.find(:all)
  end


  def create
    @category = Category.new(params[:category])
```

```
    if @category.save
      flash[:notice] = "Salvo com sucesso!"
    else
      flash[:notice] = "Ocorreu erro ao salvar!"
    end
    redirect_to :action => 'list'
  end



  private
  def authorize
    unless session[:admin_id]
      flash[:notice] = "Faça login, por favor."
      redirect_to(:controller => "login",:action => "login")
    end
  end
end
```

No código acima, o acesso a qualquer dos métodos disponíveis (`list()` e `create()`) estará sujeito a execução anterior do método privado `authorize()`, que faz a verificação do usuário logado. Caso o usuário não esteja logado ele será redicirecionado para a ação `login` do controlador `login` com uma mensagem informativa na flash.

Da mesma forma que o `before_filter`, também pode ser gerado o `after_filter`, mas sua execução ocorre após a ação.

O uso mais comum do `before_filter` é a verificação de que algumas condições estão de acordo antes da execução de uma ação. O `verify` do Rails é uma abstração do `before_filter`, que ajuda nesses mecanismos de verificação tornando as pré-condições mais claras que num código explícito.

No código acima poderíamos substituir o `before_filter` e o método `authorize()` pelo código abaixo.

```
  verify :session => :admin_id,
       :add_flash => { :notice => "Faça login, por favor."},
       :redirect_to => { :controller => "login",
                         :action => "login" }
```

Este código verifica na sessão se existe a variável `admin_id`. Caso não exista, duas ações serão executadas. A primeira adicionará na flash uma nota informativa e a

segunda redirecionará o navegador para o controlador `login` e ação `login`, como no exemplo anterior.

2.2.2.3.Action View

O módulo Action View encapsula toda a funcionalidade necessária para renderizar templates, mais comumente gerando código HTML e XML para o usuário. Pelo poder e simplicidade da linguagem Ruby não é difícil de imaginar que ela é utilizada nos templates em sua forma embutida, ou seja, não é necessário aprender outra linguagem para os templates, pois ela é a mesma utilizada para desenvolver todo o resto da aplicação.

Dentro do diretório `app/views` é criado um subdiretório para cada controlador da aplicação. Dentro de cada subdiretório estarão os templates para cada uma das ações do controlador.

O ambiente que é disponibilizado pelo controlador para a Action View possui:

- Acesso a todas as variáveis de instância do controlador. É desse modo que os controladores enviam dados de apresentação para os templates;
- Os objetos `headers`, `params`, `request`, `response` e `session` possuem métodos de acesso disponibilizados pelo controlador, permindo a visão ler atributos da sessão ou verificar que tipo de header foi enviado, por exemplo;
- O objeto do controlador fica disponível através do atributo `controller`. Dessa maneira, o template pode invocar qualquer método público do controlador;
- O caminho para o diretório base dos templates fica disponível através do atributo `base_path`.

O mais simples template RHTML é um código html puro. Se um template não possui conteúdo dinâmico, ele simplesmente é enviado diretamente ao navegador. Um exemplo:

```
<h1>GulaShop!</h1>
<p>Bom dia, bem vindo ao GulaShop.</p>
```

Mas o objetivo dos templates é justamente mostrar conteúdos dinâmicos, como a seguir.

```
<h1>GulaShop!</h1>
<p>Bem vindo visitante, hoje é <%= Time.now %></p>
```

Automaticamente a função `to_s` é chamada no objeto `Time` retornado pela função `Time.now`, convertendo o resultado em uma string e colocando-o na resposta que irá ao navegador.

Qualquer código Ruby pode ser incluído nos templates, como no exemplo a seguir.

```
<% 3.times do %>
Rails is cool!<br/>
<% end %>
```

Este código é equivalente ao código Ruby abaixo:

```
3.times do
  puts "Rails is cool!<br/>"
end
```

Ambos gerarão três linhas informando que o Rails é muito legal, cada uma sendo seguida de uma quebra de linha.

Um importante item de segurança para as aplicações Rails é a função que faz a codificação dos caracteres HTML válidos. Supondo que o desenvolvedor pegue campos de um formulário e na próxima tela exiba, por exemplo, o nome que foi digitado na tela anterior do form:

```
Obrigado por se cadastrar, <%= params[:name] %>.
```

Se o usuário inseriu algum código HTML no campo `name` do formulário, ele será apresentado agora, o que pode descaracterizar o site ou incluir código como javascript. Uma maneira mais segura de fazer seria utilizar o código a seguir.

```
Obrigado por se cadastrar, <%= h(params[:name]) %>.
```

Funcionalidades que ganham destaque são os Helpers e o suporte a layouts. Serão discutidas a seguir.

Um helper é um módulo que contém funções para auxiliar na visão, retirando lógicas complexas do código de apresentação. São equivalentes às `taglibs` no desenvolvimento utilizando JSP.

Por default, cada controlador possui seu módulo exclusivo de helpers, além de existir o módulo da aplicação que cobre todo o seu escopo e não só os métodos de um controlador em específico.

Um exemplo da implementação de um Helper é o título de uma página. Pode ser feito, sem o uso de um Helper, assim:

```
<h3><%= @page_title || "GulaShop, the complete store" %></h3>
```

Caso a variável de instância `@page_title` não tenha sido definida no controlador, o texto que está em seguida será apresentado. Uma melhor forma de se obter o mesmo efeito, mas retirando lógica de dentro da visão, seria com a criação do Helper:

```
module StoreHelper
  def page_title
    @page_title || "GulaShop, the complete store"
  end
end
```

E a visão ficaria apenas com o código abaixo.

```
<h3><%= page_title %></h3>
```

O Rails já vem com Helpers para diversas funções, incluindo formatação de datas, moeda, números, strings, tags e formulários html, javascript, debug entre outros. A seguir, alguns exemplos.

```
<%= distance_of_time_in_words(Time.now, Time.now + 33, false) %>
     1 minute

<%= human_size(123_456) %>
     120.6 KB

<%= number_to_currency(123.45) %>
     $123.45

<%= number_to_currency(234.56, :unit => "R$ ", :precision => 2) %>
     R$ 235.56

<%= number_to_percentage(66.66666) %>
     66.667%


<%= number_to_phone(2125551212, :area_code => true,
                                :delimiter => " ") %>
     (212) 555 1212

<%= debug(params) %>
      --- !ruby/hash:HashWithIndifferentAccess
      name: Dave
      language: Ruby
      action: objects
      controller: test

<%= truncate(@trees, 20) %>
      I think that I sh...

<%= pluralize(1, "person") %> but <%= pluralize(2, "person") %>
     1 person but 2 people
```

Um destaque especial dos Helpers built-in vai para os geradores de html e formulários. Pode-se criar links assim:

```
<%= link_to "Add Comment", :action => "add_comment" %>
```

O código acima gera tag HTML inserindo a URL relativa a ação `add_comment` utilizando como texto “Add Comment”. Ou um mais completo pode ser assim:

```
<%= link_to "Delete", { :controller => "admin",
                        :action     => "delete",
                        :id         => @product },
                      { :class      => "redlink",
                        :confirm    => "Are you sure?" }
%>
```

Gerando um link HTML com texto “Delete”, passando controlador, ação e

parâmetro, escolhendo qual a classe do *cascading style sheet* usar e gerando javascript que abrirá uma janela de confirmação ao clicar no link. O Rails também possui o helper `image_tag`, que é usado como exemplificado a seguir.

```
<%= image_tag("dave.png", :class => "bevel",
                          :size => "80x120") %>
```

A junção do helper `image_tag` com o `link_to` é visto a seguir.

```
<%= link_to(image_tag("delete.png", :size="50x22"),
            { :controller => "admin",
              :action     => "delete",
              :id         => @product
            },
            { :confirm    => "Are you sure?" })
%>
```

Um formulário pode ser definido assim:

```
<%= form_tag :action => :save %>
Nome: <%= text_field(:name, :product) %><br/>
Preço: <%= text_field(:price, :product) %><br/>
Categoria: <%= select(:category, :product, %w{ Cars Boats
Bikes Airplanes }) %><br/>
<%= submit_tag %>
<%= end_form_tag %>
```

Do mesmo modo existem helpers para todos os tipos de campos formulários HTML: text fields, text areas, radio buttons, checkboxes, selection lists, grouped selection lists e file upload.

Todos os códigos mostrados até agora apresentaram exemplos isolados, mas sem o contexto do layout a que pertenciam. A idéia dos layouts é parte integrante do DRY, ou seja, não escreva seu título duas vezes. Escreva uma vez só e inclua ele quantas vezes precisar em seus templates. Um template em Rails se parece com o mostrado a seguir.

```
<html>
  <head>
    <title>GulaShop: the complete store! \o/</title>
    <%= stylesheet_link_tag 'scaffold' %>
  </head>
  <body>
    <%= @content_for_layout %>
  </body>
</html>
```

Repare no método `stylesheet_link_tag()` que é um helper para inclusão do link de folha de estilos. A variável `@content_for_layout` será substituída pelo template renderizado no controlador.

Os layouts se localizam em `app/views/layouts/`. Por padrão, cada controlador possui um layout, que possui o mesmo nome do controlador seguido da extensão `.rhtml`. Na falta desse, o layout `application.rhtml` é utilizado. O layout padrão do controlador pode também ser trocado, como exemplificado abaixo.

```
class StoreController < ApplicationController
  layout "standard"
  # ...
end
```

São funcionalidades comuns a outros sistemas de layout, com as vantagens da não necessidade de arquivos de configuração e da possibilidade de utilização dos Helpers.

3. Comparativo

Nas páginas seguintes serão apresentados os dois sistemas desenvolvidos para este trabalho. Um sistema foi escrito utilizando Java, seus *design patterns* e os frameworks Struts (THE APACHE SOFTWARE FOUNDATION, 2007) - conhecido framework MVC para aplicações web - e Hibernate (RED HAT MIDDLEWARE, 2007) - framework para mapeamento objeto-relacional. O outro, utilizou Ruby on Rails.

3.1. As aplicações desenvolvidas

As aplicações foram desenvolvidas utilizando os mesmos layouts para todas as páginas. Notam-se diferenças apenas em algumas mensagens de notificação, erros e pequenos detalhes de funcionalidade, mas de modo geral um usuário se sente como num mesmo sistema ao comparar as duas versões. A seguir, apresentamos um resumo dos principais casos de uso das aplicações.

Nome do Caso de Uso: Ver lista de produtos	**UC#: 01**
Descrição: É exibida a lista de produtos para o Cliente. A lista pode exibir todos os produtos ou o resultado de uma pesquisa realizada pelo Cliente.	**Ator:** Cliente

Nome do Caso de Uso: Adicionar produto ao carrinho de compras	**UC #: 02**
Descrição: Cliente adiciona um produto ao seu carrinho de compras e é redirecionado para o UC #03.	**Ator:** Cliente

Nome do Caso de Uso: Ver carrinho de compras	**UC#: 03**
Descrição: São listados os itens do carrinho de compras do Cliente com suas respectivas quantidades.	**Ator:** Cliente

Nome do Caso de Uso: Cadastrar Cliente	**UC#: 04**
Descrição: Cliente insere seus dados para cadastro no sistema.	**Ator:** Cliente

Nome do Caso de Uso: Finalizar compra	**UC#: 05**
Descrição: Caso o Cliente esteja logado, é finalizada sua compra com os itens e quantidades presentes no carrinho de compras. Caso contrário é enviado para o UC #06.	**Ator:** Cliente

Nome do Caso de Uso: Efetuar login de Cliente	**UC#: 06**
Descrição: Cliente faz seu login no sistema utilizando seu nome de usuário e senha cadastrados no UC #04.	**Ator:** Cliente

Nome do Caso de Uso: Inserir comentário sobre produto	**UC#: 07**
Descrição: Depois de logado, o sistema permite o Cliente adicionar comentários aos produtos.	**Ator:** Cliente

Nome do Caso de Uso: Cadastrar categoria	**UC#: 08**
Descrição: Administrador cadastra nova categoria de produtos.	**Ator:** Administrador

Nome do Caso de Uso: Cadastrar fabricante	**UC#: 09**
Descrição: Administrador cadastra novo fabricante.	**Ator:** Administrador

Nome do Caso de Uso: Cadastrar produto	**UC#: 10**
Descrição: Administrador cadastra novo produto.	**Ator:** Administrador

Nome do Caso de Uso: Cadastrar promoção	**UC#: 11**
Descrição: Administrador cadastra nova promoção, seu tempo de duração, produtos afetados e percentual de desconto.	**Ator:** Administrador

Nome do Caso de Uso: Alterar categoria	**UC#: 12**
Descrição: Administrador altera categoria de produtos.	**Ator:** Administrador

Nome do Caso de Uso: Alterar fabricante	**UC#: 13**
Descrição: Administrador altera fabricante.	**Ator:** Administrador

Nome do Caso de Uso: Alterar produto	**UC#: 14**
Descrição: Administrador altera produto.	**Ator:** Administrador

Nome do Caso de Uso: Alterar promoção	**UC#: 15**
Descrição: Administrador altera promoção.	**Ator:** Administrador

Nome do Caso de Uso: Efetuar login de Administrador	**UC#: 16**
Descrição: Administrador faz login no sistema utilizando login e senha previamente cadastrados no banco de dados.	**Ator:** Administrador

Para a versão desenvolvida em Rails foram realizadas modificações no esquema do banco de dados apenas para garantir o *convention over configuration*. Fora isso, todos os campos são iguais, assim como os relacionamentos. A tela de abertura das aplicações são idênticas, conforme indicado abaixo.

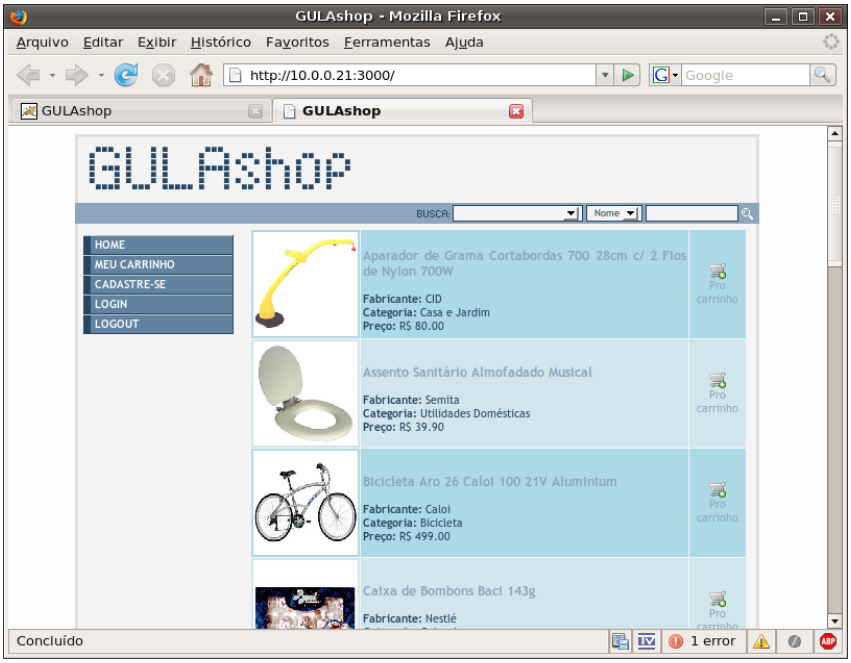

Fig.1 – Tela de abertura

A seguir, exemplo das diferenças em mensagens de erros.

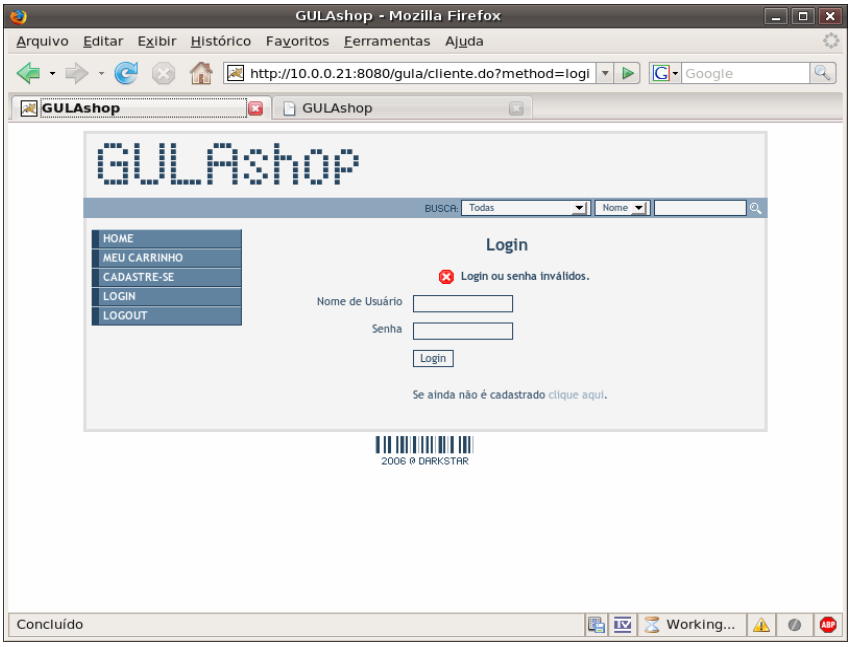

Fig.2 - Mensagem de erro no login de usuário - Versão Java

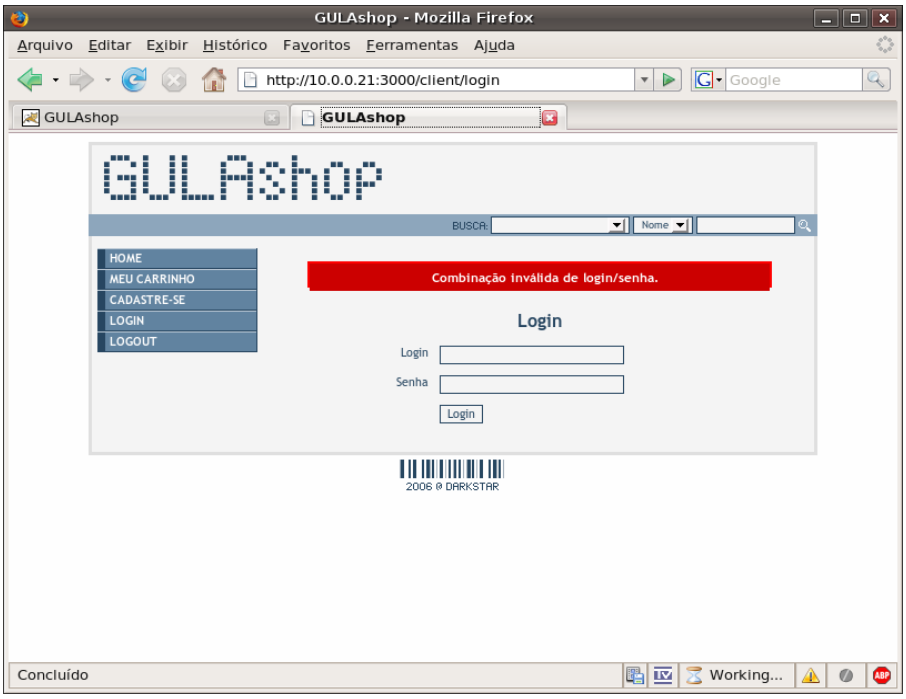

Fig.3 - Mensagem de erro no login de usuário - Versão Ruby

3.1.1.Especificidades da aplicação em Rails

Algumas funcionalidades são mais simples de implementar utilizando o framework Rails. Na versão Java, ao fazer login o usuário é enviado à página de abertura, caso seu

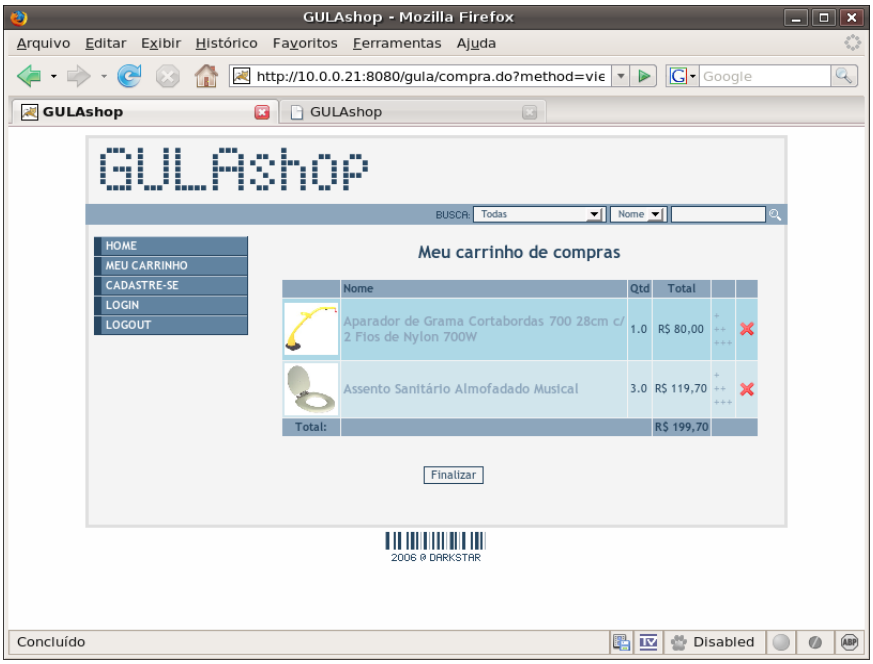

Fig.4 - Carrinho de compras - Versão Java

carrinho esteja vazio, ou para a página do carrinho, em caso contrário. Na versão em Rails, o usuário é enviado à página de abertura. Contudo, existe uma ocasião especial onde a versão Rails foge a regra. Ao tentar finalizar uma compra sem estar logado, o usuário é enviado a tela de login nas duas versões do sistema.

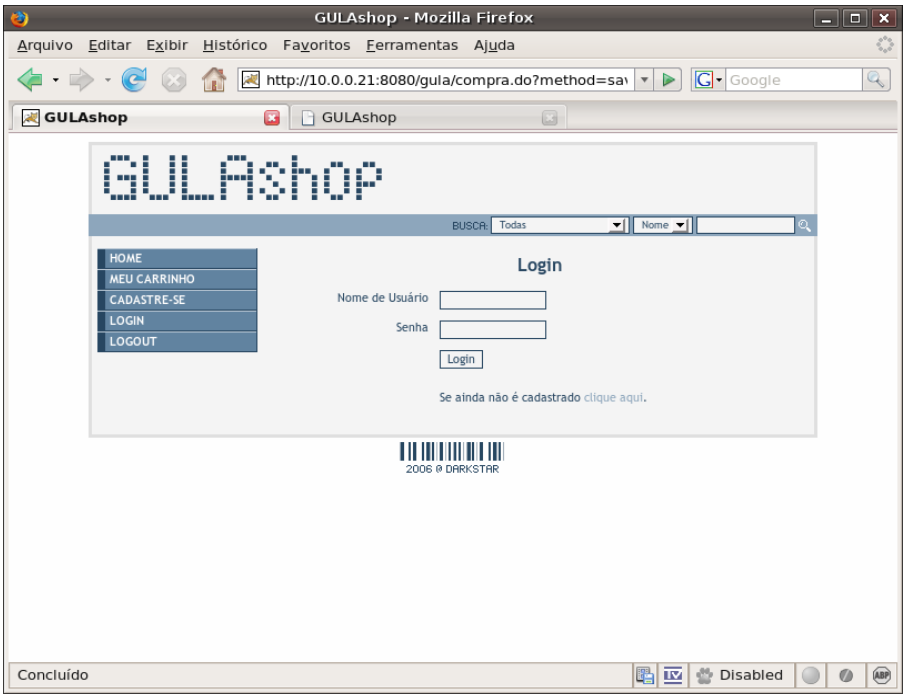

Fig.5 - Ao clicar em *Finalizar Compra* - Versão Java

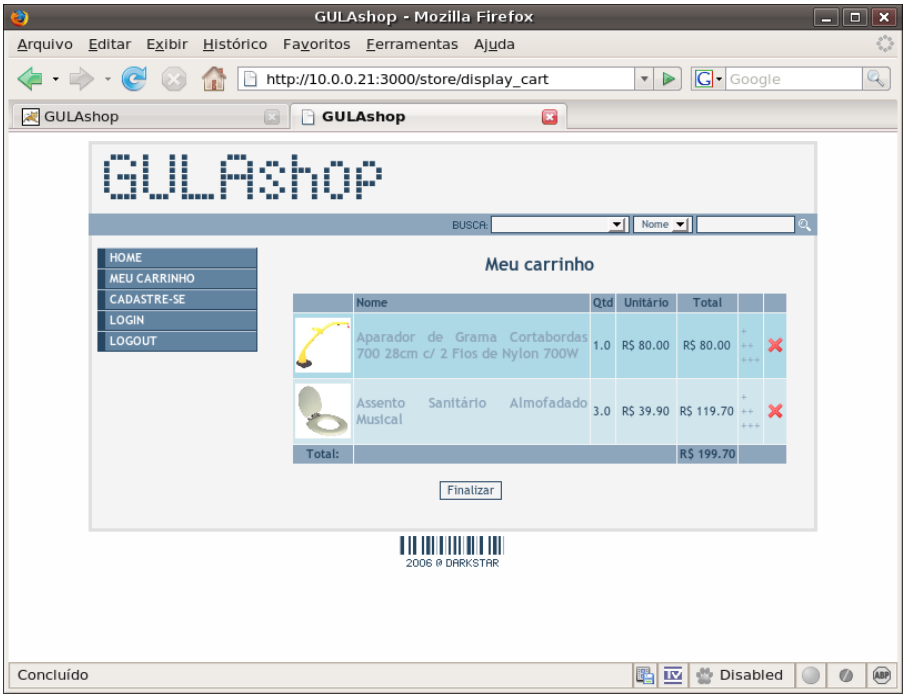

Fig.6 - Carrinho de compras - Versão Ruby

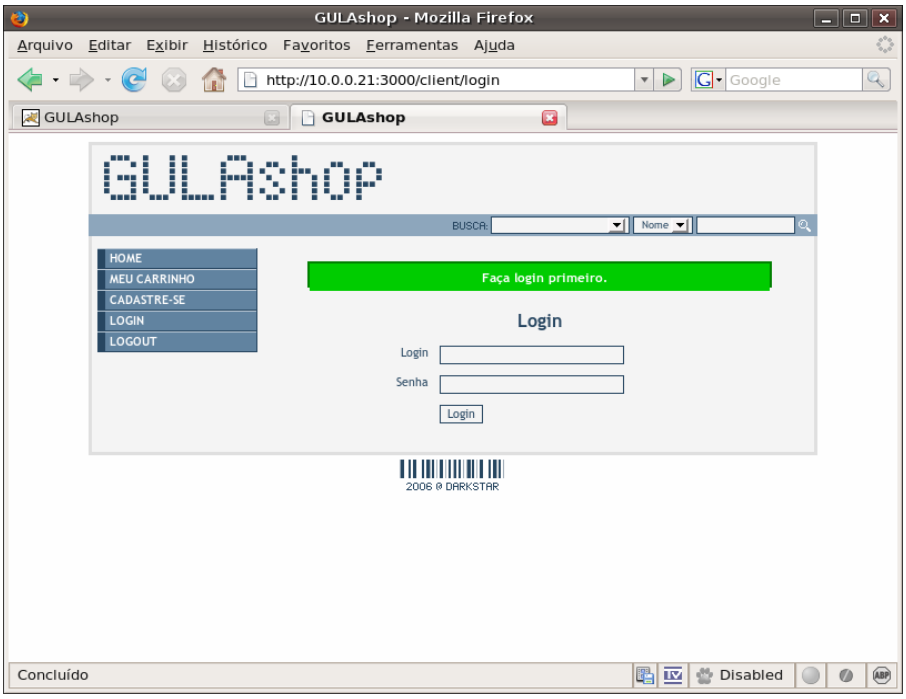

Fig.7 - Ao clicar em *Finalizar Compra* - Versão Ruby

Na versão em Java, ao efetuar o login, o usuário é direcionado para a página do carrinho, seguindo a regra definida. Já na versão Rails, o sistema verifica em que situação foi originada a requisição de login e, após o login, redireciona o navegador do usuário para esta situação, que faz a finalização da compra.

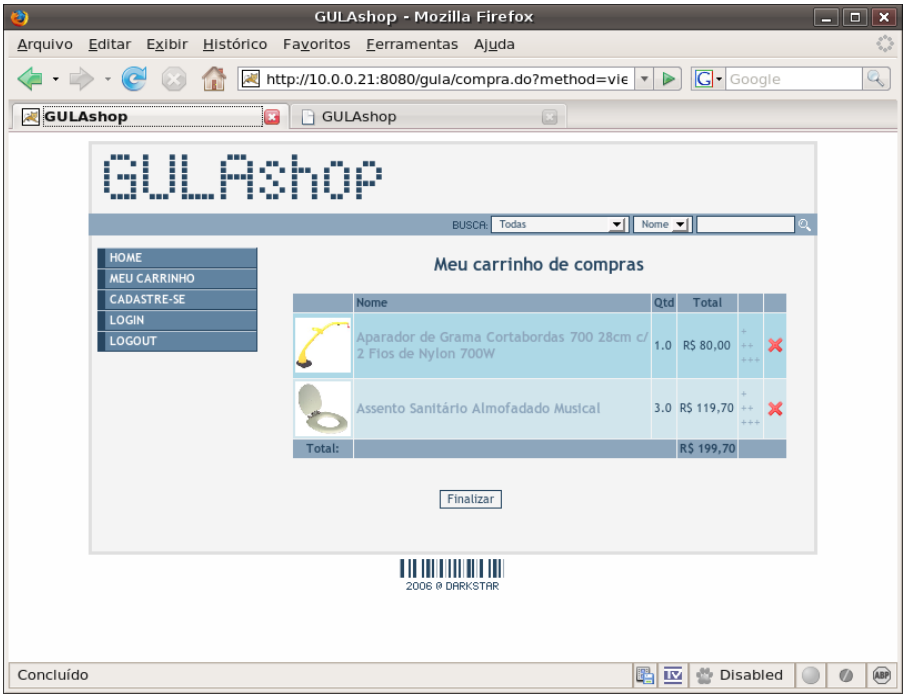

Fig.8 - Versão Java redireciona usuário para página do carrinho

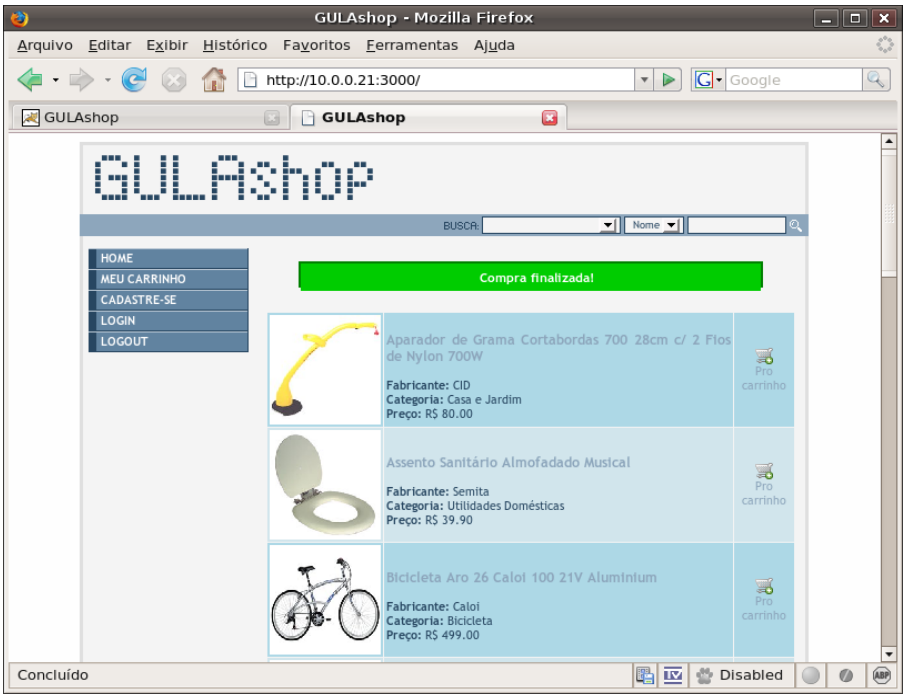

Fig.9 - Versão Rails finaliza a compra após o login

Uma funcionalidade como essa também pode ser implementada utilizando Java. O destaque está para o custo de desenvolvimento que implicaria nessa linguagem e na abordagem como foi solucionada com Rails.

Quando o usuário clica no botão *Finalizar Compra* é invocado o método `checkout` do controlador `StoreController`. Antes da execução do checkout é necessário verificar se o usuário está logado no sistema para que a compra seja computada em seu nome. Por isso, para a execução do método `checkout` é invocado o `before_filter` chamado `authorize`. A seguir, o código como foi implementado.

```
class StoreController < ApplicationController
  before_filter :authorize, :only => :checkout
  ...

  def checkout
   @items = @cart.items
   if @items.empty?
        flash[:notice] = l(:empty_cart)
        redirect_to(:action => 'index')
   else
        @order = Order.new
        @order.client = Client.find(session[:client_id])
        @order.date = Time.now
```

```
        @order.order_items << @items
        if @order.checkout
                @cart.empty!
                flash[:notice] = l(:end_cart)
        else
                flash[:notice] = l(:cart_error)
        end
        redirect_to(:controller => "store",
                    :action => "index")
    end
  end
  ...
end
```

Pode-se verificar que em nenhuma parte do código do método `checkout` existe qualquer verificação ou redirecionamento caso o usuário não esteja logado. Isso é tratado pelo `before_filter`. O método `authorize` é compartilhado por diversos controladores. Para manter o conceito DRY, ele foi colocado dentro do `ApplicationController`, controlador principal que é herdado por todos os outros controladores. Segue trecho do `ApplicationController` que contém o método `authorize`.

```
class ApplicationController < ActionController::Base
  ...
  private
  def authorize
        unless session[:client_id]
                flash[:notice] = l(:do_login)
                session[:jumpto] = request.parameters
                redirect_to(:controller => "client",
                            :action => "login")
        end
  end
end
```

O método `authorize` apenas verifica se a variável `client_id` está na sessão. Se ela estiver, o usuário já está logado na sessão e nada precisa ser feito. Caso contrário, é salva na sessão a variável `jumpto`, que guarda todos os parâmetros do request, sejam eles via `POST` ou `GET`. Em seguida, o navegador é redirecionado para a ação `login` do `ClientController`. O método `login` no `ClientController` vemos a seguir.

```
class ClientController < ApplicationController
  ...
  def login
   if request.get?
        session[:client_id] = nil
        @client = Client.new
   else
        @client = Client.new(params[:client])
        logged_in_client = @client.try_to_login
        if logged_in_client
                session[:client_id] = logged_in_client.id
                jumpto = session[:jumpto] ||
                                { :controller => "store",
                                  :action => "index" }
                session[:jumpto] = nil
                redirect_to(jumpto)
        else
                flash[:error] = l(:invalid_login)
                redirect_to :action => 'login'
        end
   end
  end
  ...
end
```

A primeira parte do método serve apenas para garantir que qualquer login seja aceito apenas via requisição `POST`. A parte mais importante está após o primeiro `else`, quando é realizada a tentativa de login. Em caso positivo, a variável `client_id` é gravada na sessão. Em seguida, são recuperados, caso existam, os parâmetros do último request, que estão na sessão, na variável `jumpto`; caso contrário, são assumidos parâmetros default - que enviam o usuário para a página inicial. Por fim, é apagado o conteúdo de sessão `jumpto` e o usuário é redirecionado.

A pequena quantidade de linhas de código necessárias para fazer uma funcionalidade como esta é notável. Totalizam 3 linhas de código. Isso demonstra a importância dada no Rails para funcionalidades que são relevantes do ponto de vista do usuário, pois levam mais usabilidade aos sistemas, e exemplifica como podem ser implementadas de forma fácil. Rails mostra como um framework deve ser um facilitador e não um complicador, principalmente para as tarefas mais comuns de qualquer sistema de informação.

3.2.Classes do Modelo em Struts x Rails ou "Beans? Pra que feijões?"

Em Java utilizam-se classes básicas para representar entidades do mundo real. Essas classes comumente apenas possuem métodos getters e setters, que em Java, ao contrário do Ruby, devem ser expressamente programados. Por exemplo, na versão do sistema desenvolvida em Java temos a classe Cliente que pertence ao pacote org.gula.entity e que representa um cliente do mundo real. Abaixo, o código da classe com a exclusão apenas dos comentários.

```
package org.gula.entity;

import java.io.Serializable;

public class Cliente implements Serializable{
   private int id;
   private String nome;
   private String login;
   private String senha;

   public Cliente() {
      this.nome = "";
      this.login = "";
      this.senha = "";
   }

   public Cliente(int id) {
      this.id = id;
   }

   public Cliente(String nome, String login, String
senha) {
      this.nome = nome;
      this.login = login;
      this.senha = senha;
   }

   public boolean equals(Cliente cliente) {
      if (this.getLogin().equals(cliente.getLogin()) &&
(this.getSenha().equals(cliente.getSenha()))) {
         return true;
      }
      return false;
   }
```

```
    public int getId() {
       return id;
    }
    public void setId(int id) {
       this.id = id;
    }
    public String getLogin() {
       return login;
    }
    public void setLogin(String login) {
       this.login = login;
    }
    public String getNome() {
       return nome;
    }
    public void setNome(String nome) {
       this.nome = nome;
    }
    public String getSenha() {
       return senha;
    }
    public void setSenha(String senha) {
       this.senha = senha;
    }
}
```

Sendo uma simples classe, com métodos construtores, verificador de igualdade, setters e getters, não há muito que se possa fazer com ela a não ser transportar dados da entidade Cliente. Essa é a funcionalidade de um Bean em Java. Para que objetos dessa classe possam refletir algum dado que existe no banco de dados foi utilizado o framework ORM Hibernate do Java. Para fazer o mapeamento entre os dados do banco e os atributos de cada classe, o Hibernate precisa de um arquivo XML de configuração. A seguir, a porção que representa o mapeamento da classe Cliente na tabela `Clientes` do banco de dados.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-
3.0.dtd">
<hibernate-mapping>
...
   <class name="org.gula.entity.Cliente"
table="Clientes">
       <id name="id" column="ID">
           <generator class="native"/>
       </id>
```

```
        <property name="nome" column="Nome"/>
        <property name="login" column="Login"/>
        <property name="senha" column="Senha"/>
    </class>
...
</hibernate-mapping>
```

A partir desse mapeamento pode-se então, por exemplo, fazer a recuperação de um item do banco. Segue abaixo trecho de código da classe `ClienteDAO` - pacote org.gula.persistence.dao - que recupera um cliente baseando na igualdade do atributo `login`.

```
package org.gula.persistence.dao;

import java.util.ArrayList;
import java.util.Collection;

import org.gula.entity.Cliente;
import org.hibernate.*;
import org.hibernate.criterion.Expression;

import org.gula.persistence.dao.DAOException;
import org.gula.persistence.HibernateUtil;

public class ClienteDAO {
...
   public Cliente retrieve(Cliente clientePk) throws
DAOException
   {
      Cliente cliente;
      Session session = null;
      try
      {
         session = HibernateUtil.currentSession();
         cliente = (Cliente) session
     .createCriteria(Cliente.class)
     .add( Expression.eq( "login", clientePk.getLogin()))
     .uniqueResult();
         HibernateUtil.closeSession ();
      }
      catch (Exception e)
      {
         throw new DAOException(e);
      }
      return cliente;
   }
}
```

É chamada a sessão do Hibernate e adiciona-se à busca alguns critérios: o objeto

retornado deve ser da classe `Cliente` - internamente ele usa o arquivo xml de mapeamento para saber em qual tabela do banco procurar; o atributo `login` deve ser igual ao do objeto `clientePk` que é passado como parâmetro; e a busca deve retornar um único resultado. Em caso de exceção, é lançado um DAOException, caso contrário é retornado o objeto `cliente` recuperado do banco.

Em Rails, temos uma abordagem diferente dessa apresentada para o Java. As classes que representam as entidades do mundo real fazem mais do que apenas serem transportadoras de objetos. Uma classe de modelo representa uma tabela do banco de dados e deve herdar da classe `Base` do pacote `ActiveRecord`. Na herança, o Rails verifica o nome da classe que está sendo criada, nesse caso `Client`, e automaticamente a relaciona com uma tabela no banco de dados que possui o nome da classe pluralizado (em inglês) e sem a capitulação, ou seja, o nome da tabela que o Rails irá procurar no banco será `clients`. Caso ele não encontre esta tabela, é lançada uma exceção. São criados métodos getters e setters, em tempo de execução, que representam cada uma das colunas da tabela no banco. Para descobrir como o Rails faz o mapeamento automático das classes basta olhar no log do banco. Temos entre outras querys:

```
SHOW FIELDS FROM clients
```

Essa query retorna as colunas de uma tabela do banco de dados juntamente com seus tipo e propriedades. Se inserido no console do Mysql, o comando retorna como apresentado a seguir.

```
mysql> SHOW FIELDS FROM clients;
+----------+-------------+------+-----+---------+----------------+
| Field    | Type        | Null | Key | Default | Extra          |
+----------+-------------+------+-----+---------+----------------+
| id       | int(11)     | NO   | PRI | NULL    | auto_increment |
| name     | varchar(80) | NO   | UNI | NULL    |                |
| login    | varchar(80) | NO   | UNI | NULL    |                |
| password | varchar(80) | NO   |     | NULL    |                |
+----------+-------------+------+-----+---------+----------------+
4 rows in set (0.00 sec)
```

A partir dessa informação, o Rails sabe quais são os atributos que devem ser gerados, incluindo seus tipos, apesar de não serem necessários, pois o Ruby faz conversão automática dos mesmos. O Rails faz isso não somente com o Mysql, mas com qualquer dos bancos de dados compatíveis, que inclui os mais conhecidos do mercado: Mysql, Postgresql, Oracle,

Mssql, BD2 e SQLite.

Como conclusão de tudo que o Rails faz automaticamente para a aplicação, temos abaixo o código da classe de modelo `Client`:

```
class Client < ActiveRecord::Base
        validates_uniqueness_of :login, :name
        validates_presence_of :login, :name, :password
        validates_confirmation_of :password

        def self.login(login, password)
          find(:first,:conditions => ["login = ? and
                        password = ?",login,password])
        end

        def try_to_login
          Client.login(self.login, self.password)
        end
end
```

Apenas dois métodos auxiliares para o login dos clientes foram definidos. Esses métodos poderiam ter sido implementados diretamente nos controladores, mas, como são genéricos, é mais interessante mantê-los na classe de modelo e assim podem ser reutilizados e mantêm-se o conceito DRY. Os métodos invocados no início da definição da classe serão explicados no próximo capítulo.

Como a classe `Client` herdou da classe `Base` do `ActiveRecord`, herdou também os seus métodos. Para fazermos a mesma busca realizada na versão em Java com Hibernate, temos como a seguir.

```
client = Client.find(:first, :conditions => ["login = ?",
"login a ser buscado"])
```

Para evitar problemas de *sql injection* (WIKIPEDIA, 2007), todas as strings passadas por parâmetro sofrem automaticamente conversão – *escape.* Será retornado o cliente encontrado ou `nil`, caso não haja nenhum com login igual ao informado. Alternativamente, o Rails cria um método automático de busca para cada atributo. Pode-se fazer como abaixo.

```
client = Client.find_by_login("login a ser buscado")
```

E o resultado será o mesmo do anterior. Para uma busca por `id`, temos:

```
client = Client.find(268)
```

E será retornado o objeto client com id igual a 268. Para retornar todos os clientes num array basta:

```
client = Client.find(:all)
```

Pode-se perceber a flexibilidade e agilidade ganha num desenvolvimento orientado a banco de dados utilizando-se o Active Record, que é o ORM do Rails. Construções mais complexas com diversos JOINS também são possíveis e, em parte, transparentes. Pode-se também fazer um SQL diretamente utilizando o método `find_by_sql`. No sistema desenvolvido em Rails não foi necessária nenhuma consulta diretamente com sql.

3.3. Validação dos formulários ou dos modelos?

Em Struts existem os ActionForms que são uma representação dos formulários que são submetidos. A validação dos ActionForms pode ser realizada tanto diretamente nas Actions, pelo método `validate` do ActionForm, ou ainda utilizando o `Validator framework` (THE APACHE SOFTWARE FOUNDATION, 2007). Nos dois casos, o que é validado são os dados do ActionForm, ou seja, os dados do formulário. Em Rails, utiliza-se o conceito da validação dos modelos. O código abaixo mostra o código necessário para criar um objeto da classe `Client` e depois salvá-lo no banco.

```
client = Client.new(:name => 'nome de teste',
                    :login => 'login_de_teste',
                    :password => 'senha_de_teste')
client.save
```

Ao acionar o método de instância `save` é executada toda rotina de validação da classe `Client`. Os métodos `validates_*` chamados no início da definição da classe `Client` recebem como parâmetros os atributos que devem ser validados.

```
class Client < ActiveRecord::Base
        validates_uniqueness_of :login, :name
        validates_presence_of :login, :name, :password
        validates_confirmation_of :password
  ...
```

Os métodos possuem nomes auto-explicativos. No exemplo acima, são validados: a unicidade do login e name, a presença no objeto do login, name e password e a validação especial que verifica igualdade entre dois atributos, nesse caso a confirmação do password. Caso alguma das validações falhe, o retorno do método `save` é falso e junto ao objeto `client` é adicionado um atributo chamado `errors` - herança da classe `Base` -, que contém as mensagens de erro geradas para cada item que falhou. As mensagens posteriormente são usadas para exibição na tela. O Rails insere mensagens padrão ou pode-se passá-las como parâmetro para os métodos `validates`.

Os métodos `validates_*` do Rails resolvem a maioria dos problemas de validação dos modelos de forma simples. Contudo, existem situações onde é necessário um tratamento mais sofisticado. Foi o que aconteceu com o modelo `product` do sistema. Parte do código da classe `Product` está reproduzida para análise do sistema de validação.

```
class Product < ActiveRecord::Base
  belongs_to :category
  belongs_to :manufacturer

  validates_uniqueness_of :nome,
              :message => [l(:error_unique,l(:name))]
  validates_presence_of :nome,
              :message => [l(:error_presence,l(:name))]
  validates_presence_of :category_id,
              :message => [l(:error_presence,l(:category))]
  validates_presence_of :manufacturer_id,
              :message => [l(:error_presence,l(:manufacturer))]
  validates_numericality_of :preco,
              :message => [l(:error_numericality,l(:price))]
  validates_numericality_of :estoque,
              :message => [l(:error_numericality,l(:stock))]

  attr_accessor :content_type
...
  def picture=(picture_field)
    self.imagem = picture_field.read
    self.content_type = picture_field.content_type.chomp
  end
...
```

```
  protected
  def validate
    errors.add(:preco, l(:error_positive,l(:price))) unless
                              preco.nil? || preco > 0.0
    errors.add(:estoque, l(:error_minimum_1,l(:stock))) unless
                              estoque.nil? || estoque > 1.0
  end

  def validate_on_create
    if self.imagem.empty? || validate_content
      errors.add(:picture, l(:error_jpeg))
    end
  end

  def validate_on_update
    if self.imagem.empty?
      @product = Product.find(self.id)
      self.imagem = @product.imagem
      return
    end
    if validate_content
      errors.add(:picture, l(:error_jpeg))
    end
  end

  def validate_content
    return true if (self.content_type =~ /image\/jpeg/).nil?
    return false
  end
end
```

Primeiramente tem-se as chamadas para as validações comuns que puderam ser utilizadas. Pode-se notar a utilização de três métodos que chamam atenção pelo nome que possuem. São eles: `validate`, `validate_on_create` e `validate_on_update`. Esses métodos são chamados automaticamente pelo Rails, cada um em uma situação.

O método `validate` é chamado sempre que é feita uma tentativa de salvar ou atualizar um objeto no banco. No método `validade` da classe `Product` tem-se a verificação do preço ser um valor positivo e do estoque ser superior a 1.

O método `validate_on_create` é chamado sempre que é feita uma tentativa de salvar um objeto no banco. O método `validade_on_create` da classe `Product` verifica se uma imagem válida foi carregada ao servidor. Nesse caso, apenas imagens com *content-type* de jpeg são aceitas.

O método `validate_on_update` é chamado sempre que é feita uma tentativa de atualizar um objeto no banco. O método `validade_on_update` da classe `Product` verifica se uma imagem válida foi carregada ao servidor. Se sim, a imagem é substituída pela nova.

Caso contrário, permanece a imagem antiga.

Das doze classes de modelo criadas para o sistema apenas três precisaram utilizar um dos métodos `validate` definidos pelo usuário. Todas as outras utilizaram apenas os métodos padrão do Rails.

3.4.Comparativo numérico

Importante para uma comparação entre o esforço gasto para desenvolver um software entre duas linguagens diferentes é a quantidade de linhas de código. Nesse caso, será comparada a quantidade de linhas de código de programação efetivamente, a quantidade de linhas em arquivos de template - que possuem apenas código de apresentação - e também a quantidade de linhas gastas em arquivos de configuração.

Foram utilizados os softwares livres SLOCCount (WHEELER, 2007) e CLOC (DANIAL, 2007) para contagem automatizada das linhas de código. Onde os dois aplicativos apresentaram diferenças na contagem foi aplicada média aritmética entre os valores e, caso necessário, arredondamento para baixo para deixar o valor inteiro. Os dois softwares não consideram linhas em branco e comentários como linha de código.

3.4.1.Número de linhas de código

Na aplicação Java obteve-se o total de 3854 linhas de código enquanto na aplicação Ruby obteve-se 681 linhas de código. Isso representa uma proporção de 5,66 vezes mais código escrito em linguagem Java para representar as mesmas funcionalidades que foram desenvolvidas em linguagem Ruby. Como explicar uma diferença tão significativa? Uma das explicações está na linguagem Ruby, que consegue ser mais concisa e realizar mais funcionalidades com menos código. A outra, está nas funcionalidades oferecidas pelo Rails, como o sistema de validação de modelos utilizando os métodos `validades`, que possibilitam o desenvolvedor escrever menos código, para preocupar-se mais com os problemas de sua aplicação e deixar com o framework os problemas de implementação. A seguir um gráfico comparativo.

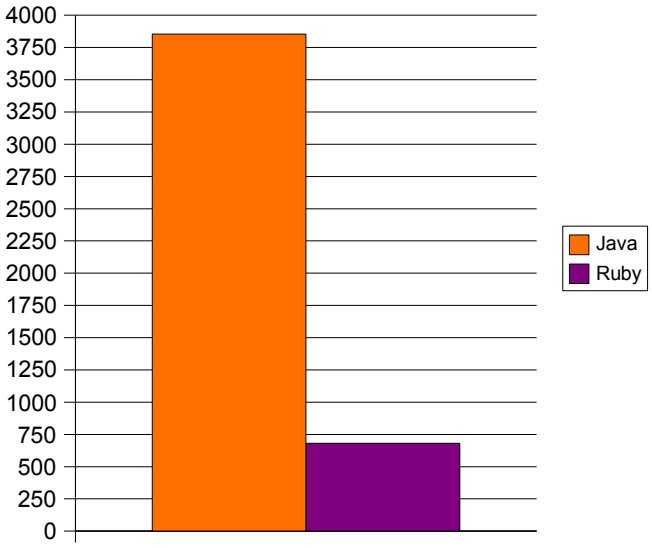

Fig.10 - Gráfico comparativo linhas de código

3.4.2.Número de linhas de código nos templates

Em JSP foram contadas 875 linhas de código. Em RHTML, a contagem ficou em 500 linhas. Uma relação de 1/1,75. Em termos percentuais temos que o total de linhas de RHTML foi 57% do total de linhas de JSP.

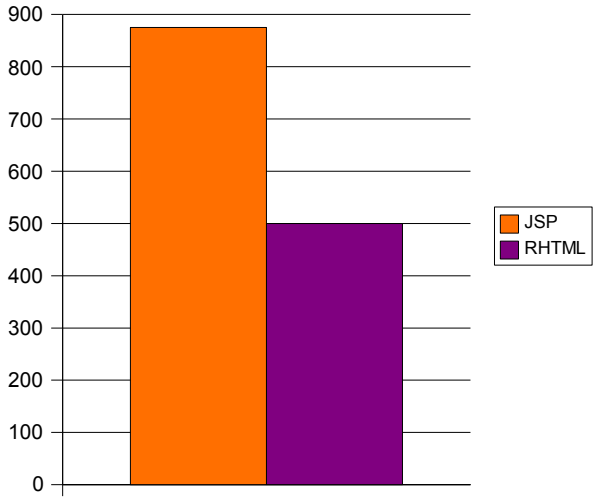

Fig 11 - Gráfico comparativo linhas de template

3.4.3.Número de linhas em arquivos de configuração

Nesse quesito o Rails mostra como leva ao extremo o conceito *convention over configuration*. Apenas 18 linhas no único arquivo YAML que foi configurado. Para a versão em Java, total de 273 linhas em arquivos XML.

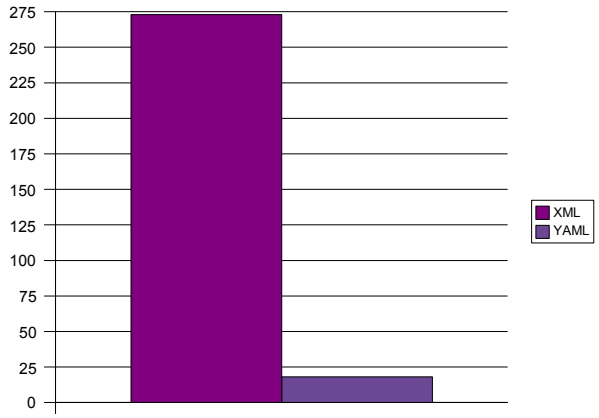

Fig 12 - Gráfico comparativo linhas em arquivos de configuração

3.4.4.Número de arquivos

Em quantidade de arquivos novamente temos um número mais equilibrado quando trata-se de templates. Foram 25 arquivos de template RHTML e 26 arquivos JSP. Em quantidade de arquivos de código, foram 24 arquivos Ruby e 44 arquivos Java. A relação não é tão grande quanto no comparativo de linhas de código, mas ainda assim é expressiva: 1,833 mais arquivos de linguagem Java.

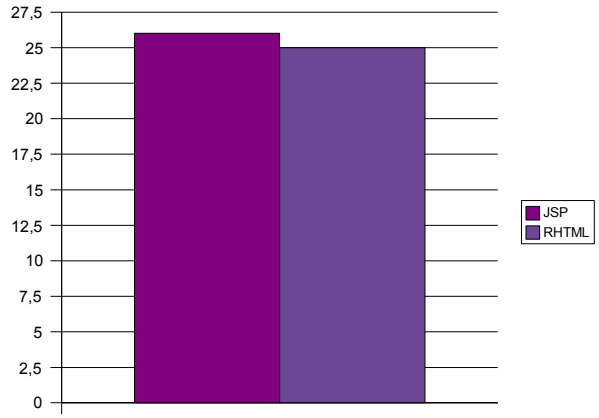

Fig.13 - Gráfico comparativo de número de arquivos de templates

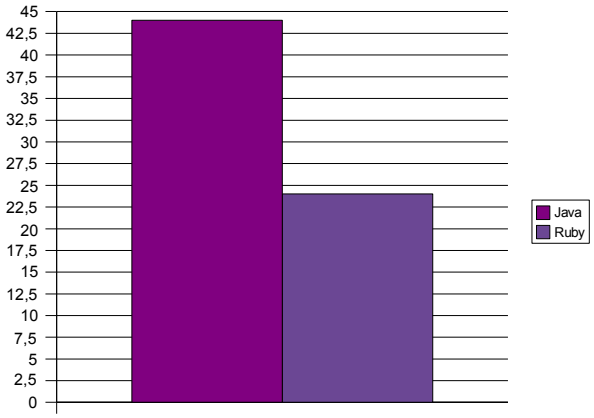

Fig.14 - Gráfico comparativo de número de arquivos de código

4. Conclusões

Ao longo do trabalho verificaram-se diversos aspectos das linguagens Java e Ruby, utilizadas no desenvolvimento dos sistemas expostos, que receberam uma abordagem aprofundada dando ênfase não apenas no mero comparativo de linhas de código, como também em suas funcionalidades, conceitos e paradigmas tendo como base um mesmo sistema, desenvolvido por uma mesma pessoa com as mesmas capacidades.

A linguagem Ruby mostrou-se mais concisa, permitindo desenvolver mais funcionalidades com menos código sem perder legibilidade, ao contrário do Java, que tem como uma de suas principais características ter um código por demais prolixo. Além disso, observou-se que a linguagem, ao seguir uma lógica comum as idéias dos programadores, permite um rápido entendimento do código já escrito e a fácil recordação de seus métodos e sintaxe. Sua tipificação dinâmica permite a geração de funções que podem reagir diferentemente a cada tipo. Apesar de não suportar herança múltipla, pode-se adicionar comportamento através de *mixins*. Considerando o exposto anteriormente, a linguagem não poderia ser de outra forma senão dinâmica, permitindo a sua boa adaptação a inúmeras situações.

Por sua vez, o Rails, ao aplicar o conceito DRY, favorece o desenvolvedor na medida que as alterações nos requisitos impactam em menos alteração de código. Percebe-se ainda que, por ter uma estrutura bem definida, o desenvolvedor, ao se deparar com uma aplicação para sua manutenção, sabe previamente onde encontrar cada porção de código. Tal estrutura foi apresentada como sendo um dos pilares de outro conceito do Rails, o *convention over configuration*.

O Active Record do Rails conseguiu, de fato, libertar a aplicação utilizada como exemplo de SQLs e de linguagens de consulta a objetos, permitindo realizar buscas no banco com apenas uma linha de código programado. Foi constatado que, pelo fato dos operadores em Ruby também serem métodos, a implementação dos cookies e da sessão no Rails se apresentou de forma simples, como na utilização de um hash. Temos ainda o aspecto da linguagem Ruby ser poderosa no tratamento de strings, não sendo, necessária a criação de um sistema de templates: a linguagem é utilizada embutida na forma de scriptlet. Foi apresentado um sistema semelhante às taglibs do desenvolvimento JSP, denominado Helper, que auxilia no desenvolvimento dos templates, porém com uma característica importante que é a geração de código HTML em concordância com os padrões da W3C.

Os filtros dos Action Methods encontrados no Rails permitem a verificação de condições específicas antes da realização das ações, possibilitando de forma simples a geração de sistemas de login, ACLs ou semelhantes. Não foi encontrada nenhuma funcionalidade correspondente no Struts. Foram estudados os métodos `validates` do Rails responsáveis pelas validações básicas antes de salvar um modelo para o banco de dados. O Struts, por sua vez, não apresenta nenhum método similar, sendo a validação realizada nos formulários. Por fim, a característica do Ruby de geração de atributos, getters e setters em tempo de execução, proporciona no desenvolvimento em Rails a não geração de Beans.

Todo o acima exposto contribui para um resultado favorável a dupla Ruby e Rails ao se realizar um comparativo de linhas de código com o trio Java, Struts e Hibernate. Este resultado é dito favorável por apresentar uma quantidade bastante considerável de linhas de código a menos do que utilizando o Java. Sabemos que a questão da quantidade de linhas de código deve ser avaliada com cautela quando considerada como principal fator em uma estimativa de custo de desenvolvimento de um projeto. No entanto, neste caso específico, a discrepância entre a quantidade de linhas de código em ambas as linguagens é tamanha que torna imperativa a sua consideração quando da elaboração de uma estimativa de custo de projeto envolvendo uma ou outra linguagem, valendo o mesmo para o custo de manutenção.

O Rails apresenta-se como um promissor framework de desenvolvimento para a WEB que, em pouco mais de dois anos de existência, alcançou um bom nível de maturidade e muitos admiradores ao redor do mundo. Em decorrência, vem ajudando na popularização da linguagem Ruby, tão pouco difundida anteriormente.

Referências Bibliográficas

BLACK, D.; FOWLER, C., 2007. **Ruby FAQ**.

Disponível em: <http://faq.rubygarden.org/>. Acesso em: 14/03/2007.

DANIAL, A., 2007. **CLOC**.

Disponível em: <http://cloc.sourceforge.net/>, Acesso em: 14/03/2007.

FERRAZ, R., 2007. **Rails para sua Diversão e Lucro**.

Disponível em: <http://kb.reflectivesurface.com/br/tutoriais/railsDiversaoLucro/>. Acesso em: 14/03/2007.

HANSSON, D., 2007. **Ruby on Rails**.

Disponível em: <http://www.rubyonrails.org/>. Acesso em: 14/03/2007.

HUNT, A.; THOMAS, D., 1999. **The Pragmatic Programmer**. 1.ed. Estados Unidos: Addison-Wesley Professional, 1999.

INGERSON, B.; EVANS, C.; BEN-KIKI, O., 2007. **YAML Ain't Markup Language**.

Disponível em: <http://www.yaml.org/>. Acesso em: 14/03/2007.

MATSUMOTO, Y., 2007. **Ruby Programming Language**.

Disponível em: <http://www.ruby-lang.org/en/>. Acesso em: 14/03/2007.

RED HAT MIDDLEWARE, 2007. **Hibernate.org**.

Disponível em: <http://www.hibernate.org/>. Acesso em: 14/03/2007.

THE APACHE SOFTWARE FOUNDATION, 2007. **Commons Validator**.

Disponível em: <http://jakarta.apache.org/commons/validator/>. Acesso em: 14/03/2007.

THE APACHE SOFTWARE FOUNDATION, 2007. **Struts**.

Disponível em: <http://struts.apache.org/>. Acesso em: 14/03/2007.

VENNERS, B., 2003. **The Philosophy of Ruby**.

Disponível em: <http://www.artima.com/intv/ruby4.html>. Acesso em: 14/03/2007.

WHEELER, D., 2007. **SLOCCount**.

Disponível em: <http://www.dwheeler.com/sloccount/>. Acesso em: 14/03/2007.

WIKIPEDIA, the free encyclopedia: **SQL Injection**, 2007.

Disponível em: <http://en.wikipedia.org/wiki/SQL_injection>. Acesso em: 14/03/2007.

WIKIPEDIA, the free encyclopedia: **KISS Principle**, 2007.

Disponível em: <http://en.wikipedia.org/wiki/KISS_principle>. Acesso em: 14/03/2007.

Anexo A – Diagrama de modelo de Banco de Dados

A seguir o diagrama de modelo de banco de dados utilizado na aplicação desenvolvida em Rails. Atenção para o nome das tabelas que são o plural (em inglês) das classes que representam os modelos da aplicação e para a padronização das chaves estrangeiras que são o nome da classe seguida de "_id".