UNIVERSIDADE FEDERAL DE SANTA CATARINA

CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

INTEGRAÇÃO DE BIBLIOTECAS JAVA EM APLICAÇÃO DESENVOLVIDA COM O FRAMEWORK RUBY ON RAILS

AUTORES

DANNY PEREIRA MATTOS
FRANCIS NOVELLO SANTOS

Orientador(a):
FRANK AUGUSTO SIQUEIRA, DR.

Florianópolis, 5 de julho de 2009.

INTEGRAÇÃO DE BIBLIOTECAS JAVA EM APLICAÇÃO DESENVOLVIDA COM O FRAMEWORK RUBY ON RAILS

Trabalho de conclusão de curso apresentado como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação pela Universidade Federal de Santa Catarina

Prof. Frank Augusto Siqueira - Orientador

DANNY PEREIRA MATTOS

FRANCIS NOVELLO SANTOS

INTEGRAÇÃO DE BIBLIOTECAS JAVA EM APLICAÇÃO DESENVOLVIDA COM O FRAMEWORK RUBY ON RAILS

Trabalho de conclusão de curso apresentado como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação pela Universidade Federal de Santa Catarina

---

Orientador: Prof. Frank Augusto Siqueira, Dr.
Universidade Federal de Santa Catarina
frank@inf.ufsc.br

Banca examinadora

Prof. Antônio Carlos Mariani
Universidade Federal de Santa Catarina
a.c.mariani@inf.ufsc.br

Prof. Leandro José Komosinski, Dr.
Universidade Federal de Santa Catarina
leandro@inf.ufsc.br

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A crescente demanda por novas aplicações fez com que a agilidade e a simplicidade se tornassem peças fundamentais no processo de desenvolvimento de software.

Ruby surgiu como uma linguagem de programação extremamente prática e, juntamente com o framework Ruby on Rails, formou uma das mais poderosas e ágeis tecnologias de desenvolvimento, despertando o interesse de um grande número de desenvolvedores que passaram a estudá-la e acompanhar sua evolução.

Recentemente, a sua capacidade de integração com a linguagem Java através do interpretador JRuby (uma das linguagens mais utilizadas atualmente) fez com que grande parte dos recursos e bibliotecas desta linguagem pudessem ser adicionados no código Ruby, objeto deste trabalho.

ABSTRACT

The raising demand for new applications turns agility and simplicity as fundamental pieces in the process of software development.

Ruby appears as a programming language extremely practice and, with its framework ‘Ruby on Rails’, become one of the most powerful and agile development technologies, raising the interest of a great number of developers that started to study and see its evolution.

Recently, its integration capacity with the Java Language through his interpreter JRuby (one of the most used programming languages in the last years) makes that great part of the resources and libraries of this language could be added to the Ruby code, object of this work.

1 INTRODUÇÃO

1.1 Motivação

A linguagem de programação Java tem se apresentado, nos últimos anos, como uma das principais linguagens de programação utilizadas pelos desenvolvedores em todo o mundo, segundo *TIOBE Programming Community Index* (JANSEN, 2008). Com isso, também foi crescente o desenvolvimento de bibliotecas úteis na linguagem que facilitaram o desenvolvimento de sistemas.

A linguagem Ruby, diferentemente do Java, é uma linguagem de programação interpretada (MATSUMOTO, 1995), com tipagem dinâmica e forte, totalmente orientada a objetos (enquanto Java é uma linguagem mista que aproveita conceitos da programação estruturada e orientada a objetos). Foi criada pelo japonês Yukihiro Matsumoto, que aproveitou as melhores idéias das outras linguagens da época, procurando criar uma linguagem intuitiva, de codificação rápida.

Com o surgimento do framework Ruby on Rails, a linguagem Ruby ganhou grande repercussão e muitos especialistas começaram a apontá-la como concorrente ou sucessora da linguagem Java na preferência entre os programadores (SALVADOR, 2008). Ciente disso, a Sun Microsystems contratou os dois desenvolvedores que estavam trabalhando no interpretador do Ruby para a linguagem Java – o JRuby, que ainda era muito instável (o interpretador clássico, e ainda mais utilizado, é escrito em C) - para trabalharem em tempo integral neste projeto dentro da própria Sun, e anunciou o Ruby como linguagem de primeira classe (linguagem com as mesmas facilidades da própria linguagem Java) para desenvolvimento na plataforma Java a partir da versão 7.

Por essas características, podemos supor que a linguagem Ruby passará a ser cada vez mais utilizada para o desenvolvimento de aplicações Web, e a sua integração com as bibliotecas do Java, sua sintaxe amigável e simples e sua agilidade, adquirida com o framework Rails, trará um mundo de novas possibilidades, onde a simplicidade é a palavra de ordem.

1.2 Objetivos do Trabalho

O escopo do trabalho engloba as linguagens de programação Ruby e Java, o Framework Ruby on Rails e a integração de bibliotecas da especificação Java à linguagem Ruby, utilizando o framework Ruby on Rails e o JRuby (interpretador Ruby para o *bytecode* Java). Exclui-se do escopo o estudo da integração em outros ambientes e arquiteturas.

O objetivo principal desse trabalho é integrar, a um sistema desenvolvido em Ruby e gerado pelo framework Rails, bibliotecas da especificação Java que contenham funcionalidades que não existam na linguagem Ruby ou que sejam mais robustas ou interessantes de serem utilizadas em uma arquitetura Java (em sistema rodando sob a Máquina Virtual Java - JVM).

Para a realização desse objetivo, devermos tratar, mais especificamente, cada um dos seguintes pontos:

- Realizar um estudo comparativo entre as linguagens Ruby e Java;
- Realizar um estudo do *framework* Ruby on Rails;
- Desenvolver um sistema web em linguagem Ruby gerado a partir do Rails, utilizando o JRuby (interpretador escrito em Java), e implantá-lo em um servidor Java;
- Integrar algumas bibliotecas da especificação Java a este sistema.

1.3 Justificativa

Ao longo dos últimos anos as equipes de desenvolvimento que trabalham com a linguagem Ruby têm encontrado dificuldades na integração de algumas bibliotecas da lingua-

gem Java (tanto das especificações JavaSE – *standard edition* - como JavaEE – *enterprise edition*) à linguagem Ruby em uma arquitetura Java (o interpretador JRuby levou sete anos, de 2001 a 2008 para ter uma versão estável, sendo que os principais desenvolvedores têm trabalhado dentro da própria Sun desde 2006).

Diante deste cenário apresentado, justifica-se um trabalho de pesquisa que procure compreender melhor a integração entre as duas linguagens dentro de um ambiente Java e que procure soluções para alguns dos problemas ainda enfrentados pelos desenvolvedores Ruby na utilização de bibliotecas do Java.

1.4 Metodologia

Utilizaremos o ambiente de desenvolvimento NetBeans que, a partir da versão 6, oferece extenso suporte ao desenvolvimento com Ruby on Rails. Neste ambiente desenvolveremos uma aplicação utilizando a linguagem Ruby utilizando o Ruby on Rails e demonstraremos a inserção de funcionalidades de bibliotecas com código Java, a esta aplicação, culminado com a implantação desta aplicação em um servidor Java.

1.5 Organização do texto

Ao longo deste trabalho abordaremos, primeiramente, cada uma das tecnologias envolvidas no desenvolvimento de uma aplicação que utilize as linguagens Java e Ruby de maneira integrada, as próprias linguagens, os *frameworks* e bibliotecas selecionadas, exemplos de integração bem sucedidos de algumas bibliotecas e a aplicação com a demonstração de integração de outras bibliotecas do Java à aplicação desenvolvida em Ruby.

No capitulo 2 será mostrado um estudo conceitual e comparativo entre as linguagens Java e Ruby, abordando suas características, semelhanças e principais diferenças. Já no capítulo 3 será abordado o *Framework* Ruby on Rails, descrevendo sua importância e capacidade no desenvolvimento de aplicações web com Ruby. No capítulo 4 será apresentado o JRuby, ferramenta que possibilita a integração dessas duas linguagens abordadas neste trabalho, exemplificando essa integração com trechos de códigos e casos de sucesso na integração das bibliotecas do JMS (*Java Message Service*) e *Spring Framework*. Nos demais capítulos será apresen-

tada a aplicação que foi desenvolvida ao longo deste trabalho, sua arquitetura, desenvolvimento e como foi realizada a integração das bibliotecas Java ao código Ruby .

2 COMPARAÇÃO ENTRE AS LINGUAGENS DE PROGRAMAÇÃO JAVA E RUBY

2.1 Ruby

Ruby é uma linguagem de programação que foi criada em 1993 por Yukihiro “Matz” Matsumoto. Sua implementação clássica foi escrita na linguagem de programação *C* e ficou conhecida como *Matz Ruby*. O objetivo de *Matz* foi produzir uma linguagem produtiva e agradável para ser utilizada seguindo o princípio da “least surprise”, o que significa que ela tenta minimizar a complexidade, deixando as coisas mais simples para o desenvolvedor através da eliminação de caracteres especiais e inclusão de facilidades de programação.

Ruby tem outras características fundamentais como as de ser interpretada e altamente reflexiva, sendo todas as variáveis, inclusive os tipos, objetos primitivos. Daí a classificação como totalmente orientada a objetos (apesar desta classificação não ser formal pela falta de uma definição clara do que é uma linguagem totalmente orientada a objetos, o que também pode ser questionado por Ruby oferecer suporte para o paradigma funcional, por exemplo). Ruby é também dinamicamente e fortemente tipada, o que significa que todas as variáveis devem ter um tipo (fazer parte de uma classe), mas esta classe pode ser alterada dinamicamente. Ruby tem recursos de tratamento de exceções, assim como as linguagens *Java* e *Python*, o que facilita o tratamento de erros. A linguagem está disponível para diversas plataformas, como Microsoft Windows, .NET, Linux, Solaris e Mac OS X, além de também ser executável sobre a máquina virtual do Java (através do JRuby). Diferentemente de outras tecnologias de código aberto, não existe uma empresa por trás de suas operações, porém, a *Sun*

Microsystems tem empenhado bastante tempo à pesquisa e desenvolvimento do interpretador JRuby (que será visto em um capítulo mais a frente). Além disso, o projeto conta com doações feitas pelos usuários e por empresas que conseguiram aumentar sua produtividade utilizando Ruby.

Ruby tem um *garbage collector* (coletor de lixo, para limpar referências a objetos não mais utilizados) que realmente é do tipo “marca-e-limpa”. Ele atua em todos os objetos do Ruby, não precisando se preocupar em manter contagem de referências em bibliotecas externas. Ruby não precisa de declaração de variáveis, usando apenas a convenção de nomenclatura para delimitar o escopo das variáveis (por exemplo: ’var’ = variável local, ’@var’ = variável de instância e ’$var’= variável global). Ruby não exige o uso do ’self’ (ou ‘this’) em cada membro da instância, tem herança única, porém entende o conceito de módulos que são coleções de métodos. Toda classe pode importar um módulo e incorporar seus métodos. Alguns consideram que isso é uma maneira mais limpa do que a utilização de herança múltipla, que é complexa e não é tão freqüentemente usada quanto a herança simples (ou única).

Inteiros em Ruby podem (e devem) ser usados sem levar em conta a sua representação interna. Existem inteiros pequenos (instâncias da classe *Fixnum*) e grandes (*Bignum*), mas o desenvolvedor não precisa se preocupar com qual está sendo utilizado atualmente. Se um valor é pequeno o bastante, um inteiro é um *Fixnum*, do contrário é um *Bignum*. A conversão ocorre automaticamente no Ruby. Além disso, a linguagem é altamente portável, desenvolvida principalmente no *Linux*, mas funcionando em muitos tipos de *UNIX, DOS, Windows 95/98/Me/NT/2000/XP, MacOS, BeOS, OS/2*, etc. Ela tem vários recursos para processar arquivos de texto e para fazer tarefas de gerenciamento de sistema (assim como o Perl).

2.2 Java

Java é uma linguagem de programação orientada a objetos desenvolvida na década de 90. Diferentemente da maioria das linguagens convencionais, que são compiladas para código nativo, a linguagem Java é compilada para um "bytecode" (código intermediário) que é executado por uma máquina virtual (JVM), daí a sua classificação como híbrida.

Foi criada como uma ferramenta de programação em computação, fazendo parte de um pequeno trabalho anônimo e secreto chamado "the Green Project" da Sun Microsystems em 1991 e, em 1995, foi rebatizada para Java. Desde seu lançamento, a plataforma Java foi adotada mais rapidamente do que qualquer outra linguagem de programação na história da computação, atingindo mais de 4 milhões de desenvolvedores em todo mundo (WIKIPEDIA, 2009). Java continuou crescendo e hoje é uma referência no mercado de desenvolvimento de *software*, tornou-se popular pelo seu uso na Internet e hoje possui seu ambiente de execução presente em navegadores web, *mainframes*, SOs, celulares, *palmtops* e cartões inteligentes, entre outros.

2.3 Diferenças básicas entre as linguagens

As linguagens de programação Java e Ruby surgiram na mesma época (início dos anos 90), porém enquanto o Java começou a sua ascensão em 1995 com uma adoção muito rápida pelos desenvolvedores, Ruby permaneceu sem grande repercussão até o surgimento da biblioteca Rails em 2003. Assim como Java, Ruby segue o paradigma de programação orientado a objetos, porém existem muitas diferenças entre elas. A primeira é que Ruby é dinamicamente tipada e seu código fonte é executado por um interpretador (escrito em *C*, ou a alternativa JRuby explorada neste trabalho, escrito em Java). A vantagem da tipagem dinâmica é a eliminação da repetição de código e diminuição no número de exceções que podem ser causadas, enquanto o Java possui tipagem estática. Isso não significa que a tipagem de Ruby seja mais fraca, e sim, que ela força a passagem correta dos objetos, visto que em Ruby, tudo é objeto.

Ruby é executado em um interpretador, permitindo que o desenvolvedor possa testar um código sem compilação. É possível até mesmo executar Ruby interativamente, cada linha como foi digitado. Além do rápido retorno, linguagens interpretadas têm uma vantagem em lidar com *bugs* (pequenos erros ocorridos ao longo do desenvolvimento). Com linguagens compiladas como Java, para analisar o código o programador deve verificar qual a versão e-xata da aplicação está com problemas e, em seguida, procurar o código com *bugs* no sistema

de gerenciamento de código-fonte. Na vida real, muitas vezes, essa é uma tarefa difícil e cansativa, enquanto com linguagens interpretadas, por outro lado, o código fonte é imediatamente disponível para a análise.

A linguagem Ruby foi influenciada por inúmeras linguagens, dentre elas podemos destacar *Perl* (linguagem prática), *Smalltalk* (orientação a objetos – tudo é um objeto, multiprogramação – execução de tarefas concorrentemente e sintaxe semelhante), *Eiffel e Ada* (sintaxe) e *Java e Python* (tratamento de exceções).

Ruby suporta múltiplos paradigmas de programação. Além do seu estilo orientado a objeto puro, ela suporta, também, o modelo procedural, onde é possível escrever código fora de qualquer classe ou método, e o paradigma funcional.

Em Ruby, você pode empacotar qualquer função ou bloco de código, sem a necessidade de classes anônimas ou definições de métodos..

Ruby tem muito para ensinar aos programadores Java. O *framework* Ruby on Rails (RoR) mostra como pode ser simples desenvolver uma aplicação Web. Com JRuby, RoR em breve será capaz de reutilizar ainda mais funcionalidades existentes em Java. JRuby já incorpora *Jython*, *JavaScript* e outras linguagens dinâmicas para script de aplicações integradas a Java.

Mesmo quando não estão usando linguagens dinâmicas, desenvolvedores em Java podem beneficiar-se dos conceitos de Ruby integrados a ambientes já desenvolvidos em Java.

3 O FRAMEWORK RUBY ON RAILS

3.1 Principais Características

Ruby on Rails (RoR) é um poderoso conjunto de ferramentas de software que permite a construção rápida de sofisticadas aplicações web. Feito em Ruby, RoR é um *framework* que proporciona um amplo conjunto de capacidades como, por exemplo, lidar com toda a comunicação com o banco de dados. Desta forma, o programador pode lidar simplesmente com os objetos, deixando para o framework a geração das consultas em *SQL*.

O Ruby on Rails fornece um modelo de sistema para tratamento de *layouts* e seções de página, acrescentando facilidades para o processamento da tela e das atualizações em Ajax.

O *framework* foi criado para fazer o melhor uso do tempo de desenvolvimento, eliminando gargalos e permitindo a criação de soluções, em uma abordagem interativa, com mais agilidade. RoR foi projetado para, entre outros ideais, ser uma solução de desenvolvimento completa, onde suas camadas se comunicam da forma mais transparente possível, ser uniforme (escrito totalmente em Ruby), seguir a arquitetura MVC (*Model-View-Controller*) e ser orientado a banco de dados, uma vez que é possível criar aplicações com base em estruturas pré-definidas.

As aplicações em Rails seguem dois padrões principais. O DRY – *Don´t Repeat Yourself* - diz que cada trecho de conhecimento de um sistema deve ser expresso em um único lugar; e o "*Convention Over Configuration*", que define padrões de nomenclatura e localização de arquivos que, quando seguidos, fazem com que a aplicação funcione sem a necessidade de configurações adicionais. O Rails é um *framework* altamente integrado, não só com *XP* (*Ex-*

treme Programming), mas com todas as metodologias de desenvolvimento ágil. Isso porque o Rails foi desenvolvido seguindo os valores do manifesto ágil, mesmos valores que formam a base de todas as metodologias de desenvolvimento ágil.

O Rails, através de um conjunto de bibliotecas, gera uma estrutura onde os desenvolvedores criam os modelos, controladores e interfaces, seguindo o padrão MVC e integrando-as através das convenções de nomenclatura. Para a camada de persistência utiliza-se o *ActiveRecord*, que através de convenções (e não configurações, como na API de persistência do Java) realiza o mapeamento objeto-relacional para o Rails.

A camada de visão é responsável pela criação das páginas exibidas pela aplicação e a camada de controle é o centro lógico da aplicação que faz a interação entre as camadas.

3.2 Origem

Rails foi criado por David Hansson, desenvolvedor dinamarquês de uma empresa americana chamada *37signals* e teve sua primeira versão (v0.5) disponibilizada em 25 de Julho de 2004.

3.3 Componentes

RoR possui em sua estrutura outros 5 *frameworks*: *Active Record*, *Action Pack*, *Action Mailer*, *Active Support* e *Active WebServices*.

O *Active Record* é uma camada de mapeamento objeto-relacional (ob*ject-relational mapping layer*), responsável pela interoperabilidade entre a aplicação e o banco de dados e pela abstração dos dados (WIKIPEDIA,2009). A extensa utilização de introspecção, tanto do lado da estrutura relacional, quanto da parte das classes e objetos, torna praticamente desnecessário escrever mais do que meia dúzia de linhas de código para se fazer o mapeamento de uma classe/tabela.

Action Pack compreende o *Action View* (geração de visualização de usuário, como HTML, XML, JavaScript, entre outros) e o *Action Controller* (controle de fluxo de negócio). Da

mesma forma que o módulo *Active Record* é responsável pela implementação dos modelos, *Action View* é o módulo responsável pela implementação das visões. O diferencial está no fato de este módulo ser fortemente influenciado pela metodologia ágil, pois quase todas as funcionalidades tipicamente utilizadas para construção de uma interface web podem ser adicionadas através de uma API concisa e intuitiva. Mesmo as tendências mais recentes das aplicações Web modernas — como Javascript, DHTML e AJAX — estão fortemente integradas ao *framework*.

O *Action Mailer* é um *framework* responsável pelo serviço de entrega e recebimento de *e-mails*. É relativamente pequeno e simples, porém poderoso e capaz de realizar diversas operações apenas com chamadas de entrega de correspondência.

O Active Support é uma coleção de várias classes úteis e extensões de bibliotecas padrões, que foram consideradas úteis para aplicações em Ruby on Rails.

O Action WebServices provê uma maneira de publicar APIs interoperáveis com o Rails, sem a necessidade de perder tempo com especificações de protocolo. Implementa *WSDL* e *SOAP*.

O *Action WebServices* não estará mais presente na versão 2.0 do Rails, visto que o mesmo está voltando-se para a utilização do modelo REST. Mesmo assim, aos ainda interessados em utilizá-lo, será possível fazê-lo através da instalação de um *plugin*.

3.4 Conceitos

Os conceitos: DRY e *Convention over Configuration* estão implementados por todo o Rails, mas podem ser mais notados nos "pacotes" do *Active Record* e *Action Pack*. Seu uso visa aumentar a produtividade do desenvolvedor.

DRY (*Don't Repeat Yourself*, ou em português “Não se repita”) é o conceito por trás da técnica de definir nomes, propriedades e códigos em somente um lugar e reaproveitar essas informações em outros. Por exemplo, ao invés de ter uma tabela *Pessoas* e uma classe *Pessoa*, com uma propriedade, um método "acessador" (*getter*) e um "armazenador" (*setter*) para cada campo na tabela, têm-se apenas a tabela no banco de dados. As propriedades e métodos

necessários são "injetados" na classe através de funcionalidades da linguagem Ruby. Com isso, economiza-se tempo, já que não é necessário alterar a tabela e uma série de classes relacionadas a ela. No Rails, é preciso alterar apenas o banco de dados, sendo que tudo o que se baseia nessas informações são atualizadas automaticamente.

Na maioria dos casos, é preciso utilizar convenções no dia-a-dia da programação, em geral, para facilitar o entendimento e manutenção por parte de outros desenvolvedores. Sabendo disso, e sabendo que o tempo gasto para configurar XML em alguns *frameworks* de outras linguagens é extremamente alto, decidiu-se adotar o conceito de *Convention over Configuration*, que permite integrar as diferentes partes do *framework* sem praticamente nenhuma intervenção direta do desenvolvedor (WIKIPEDIA, 2009). As convenções englobam todo tipo de aspecto de uma aplicação web: o nome dos modelos e tabelas relacionais associadas; o nome da colunas das tabelas que constituem chaves primárias e estrangeiras; o nome dos arquivos e das classes que implementam cada camada da estrutura MVC, o nome e identificador dos campos HTML, e diversas outras (GAMA,2008).

3.5 Escalabilidade

Uma questão freqüente é sobre a escalabilidade de aplicações escritas em Rails. Ao contrário de outras tecnologias, não é preciso fazer um código específico para que o sistema esteja preparado para "escalar". Quando necessário, pode-se adotar uma das táticas disponíveis para escalabilidade em Rails. Vale notar que o único problema da escalabilidade é a manutenção de sessões entre servidores. Portanto, a saída mais óbvia é guardar estas sessões em volumes NFS (*Network File System*), acessíveis por todos os servidores de aplicação. Outra tática é usar o armazenamento de sessões diretamente no banco de dados. Uma terceira seria salvar a sessão em um *cookie* na máquina do usuário. Como se pode ver, uma aplicação Rails já nasce com todo o suporte necessário para crescer sem traumas (WIKIPEDIA, 2009).

Porém, a maioria dos sites não necessita de esquemas sofisticados de escalabilidade. Em sites menores ou normais, uma configuração padrão do servidor web consegue suportar uma boa quantidade de carga, principalmente se forem usados o *lighttp* – que é um servidor web projetado para otimizar ambientes de alta performance, possuindo um baixo consumo de memória e um bom gerenciamento de carga da CPU – e *FastCGI* (Uma variante da interface

CGI que provê um melhor desempenho e escalabilidade (WIKIPEDIA,2009). Ao invés de criar um processo para cada requisição, como é feito com um programa CGI, ele utiliza um processo persistente que manipula várias requisições durante o "tempo de vida" do processo. Existem casos de sites feitos em Rails que suportaram 5 milhões de visitas em um mês, ou seja, aproximadamente 115 por minuto, uma performance considerada suficiente para 90% das aplicações atuais.

3.6 Uso do Padrão MVC (*Model-View-Controller*) no Rails

Como já dito, o Ruby é construído em torno do padrão MVC, separando os dados, lógica e telas de apresentação.

O *Model* é representado pelo *Active Record* (mantém a relação entre a Objetos e banco de dados, como mencionado anteriormente) .

O *View* é representado pelo *Action View* (apresentação dos dados em formato particular).

Já o *Controller* é representado pelo *Action Controller* que, por um lado, examina os modelos para um dado específico e, por outro, organiza os dados (pesquisa, ordenação, ajuste) para que o dado se adapte às necessidades de um determinado ponto de vista. Esse subsistema é um mediador (*broker*) de dados e está situado entre o *Active Record* e o *Action View*.

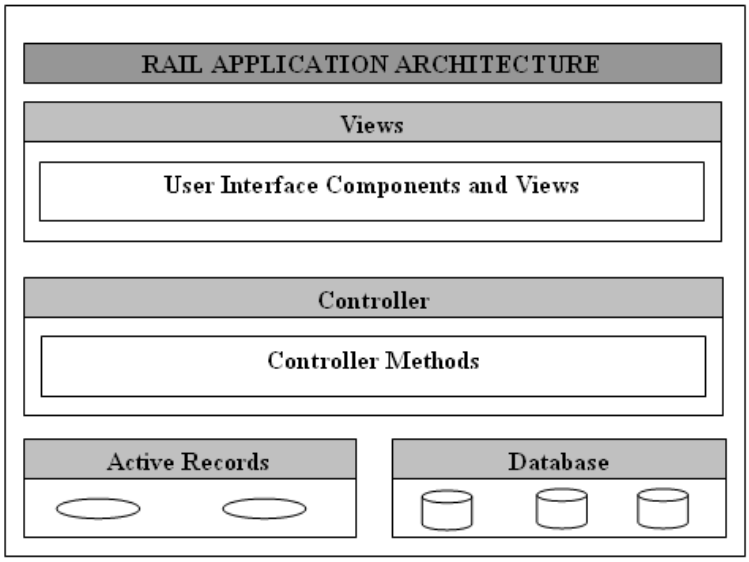

Figura 1: Modelo MVC do Ruby on Rails.

4 INTEGRAÇÃO DE CÓDIGO FONTE RUBY E JAVA UTILIZANDO O INTERPRETADOR JRUBY

O interpretador JRuby é a implementação em Java do interpretador clássico desenvolvido para o Ruby, escrito na linguagem C. É um software liberado através das licenças CPL/GPL/LGPL (open source), e que permite a integração do código escrito em Ruby com qualquer aplicação em Java ou código escrito em Java em uma aplicação Ruby, como demonstrado neste projeto.

A versão original do interpretador foi desenvolvida por Jan Arne Petersen em 2001, e durante muito tempo foi uma parte direta do código do interpretador Ruby 1.6, desenvolvido na linguagem C. Com o lançamento da versão 1.8.6, iniciou-se um esforço para a atualização do interpretador com as características e semânticas dessa nova versão da linguagem.

Desde 2001, vários contribuidores têm participado do projeto, que possui quatro membros considerados como líderes do time de desenvolvimento: Charles Nutter, Thomas Enebo, Ola Bini e Nick Sieger. Em setembro de 2006, a Sun Microsystems contratou Enebo e Nutter para trabalhar no projeto JRuby em tempo integral, com o intuito de prestar dedicação total ao aperfeiçoamento do interpretador. Em junho de 2007, *a ThoughtWorks* contratou Ola Bini para trabalhar com Ruby e JRuby.

JRuby e a plataforma Java são uma atraente combinação para ser aplicada a diversas situações de desenvolvimento de software. Por exemplo, a partir de um script em Ruby pode-se chamar a biblioteca “Math”, do Java, e ter acesso a suas poderosas capacidades computacionais ou, ainda, chamar a biblioteca “Swing” e exibir uma caixa de diálogo ao usuário, para que ele entre com um valor, permitindo, após isso, que o script prossiga.

Além das características citadas acima, o JRuby apresenta melhor escalabilidade com threads nativas do Ruby, suportando o padrão Unicode nativamente e permitindo também ser compilado em tempo de execução.

O suporte ao *framework* Rails surgiu a partir da versão 0.9, incluindo a possibilidade de executar RubyGems (bibliotecas nativas do Ruby) e o servidor WEBrick. Tecnicamente, o JRuby suporta quase 100% a linguagem Ruby, mas com a vantagem de rodar sobre a máquina virtual do Java (JVM), acessando nativamente aplicações escritas em Java, e podendo rodar sob servidores de aplicação Java como o Glassfish e o JBoss.

A possibilidade da implantação de aplicações Ruby nestes servidores fez com que os holofotes do desenvolvimento Web se voltassem para a linguagem e com a facilidade de estruturação do modelo MVC pelo *framework* Rails, além do poder da maioria das bibliotecas do Java, que já estão em uma fase de desenvolvimento mais avançada que as do Ruby, tornaram possível a utilização de código Ruby em uma nova aplicação de médio ou grande porte, ou ainda a integração de módulos Ruby em uma aplicação pré-existente, o que sofria uma forte resistência em virtude da necessidade de utilização de outros servidores de menor confiabilidade e desempenho.

Ainda não foi possível igualar o tempo de execução de um script Ruby executado diretamente com o interpretador clássico escrito em C, e o mesmo script executado em um ambiente Java através do interpretador JRuby. A equipe que desenvolve o JRuby porém trabalha com a previsão de que os scripts serão executados mais rapidamente do que no próprio ambiente Ruby.

Para exemplificar a invocação do Java dentro do código Ruby, seguem dois trechos de código:

```
1.  require ‘java’

    include_class “javax.swing.JFrame”
    include_class “javax.swing.JButton”

    frm = JFrame.new(“My frame”)
    btn = JButton.new(“My button”)

    frm.set_size(300, 300)
    frm.content_pane.add(btn)
    frm.show
```

Nesse trecho é exemplificada a inclusão da biblioteca swing dentro do código Ruby.

```
2.  include_class
    “java.awt.event.ActionListener”

    class MyListener < ActionListener
      def actionPerformed(event)
        event.source.text = “New text”
      end
    end

    btn.add_action_listener(MyListener.new)
```

No exemplo, há a inclusão da biblioteca responsável pelo tratamento de eventos do Java dentro de um código escrito em Ruby. Nos itens a seguir destacaremos casos de sucesso na utilização do JRuby para a integração de algumas bibliotecas da linguagem Java.

4.1 Spring Framework

O *Spring Framework* é um conjunto de bibliotecas utilizado para injeção de dependências, orientação a aspectos e controle de fluxo, entre outras utilidades, em uma arquitetura Java empresarial (JavaEE sem EJB) considerada alternativa à arquitetura JavaEE tradicional onde são utilizados os *Enterprise Java Beans* (EJB).

A integração do Spring com o Ruby através do JRuby é possível com o *plugin* Goldspike, desenvolvido por Chris Nelson, que possibilita a implantação (*deploy*) de uma aplicação desenvolvida em Ruby em um servidor Java.

Código para instalação do plugin:

```
Jruby script/plugin install svn://rubyforge.org/var/svn/jruby-extras/trunk/rails-integration/plugins/goldspike
```

Chris Nelson também desenvolveu um módulo para a integração do Spring.

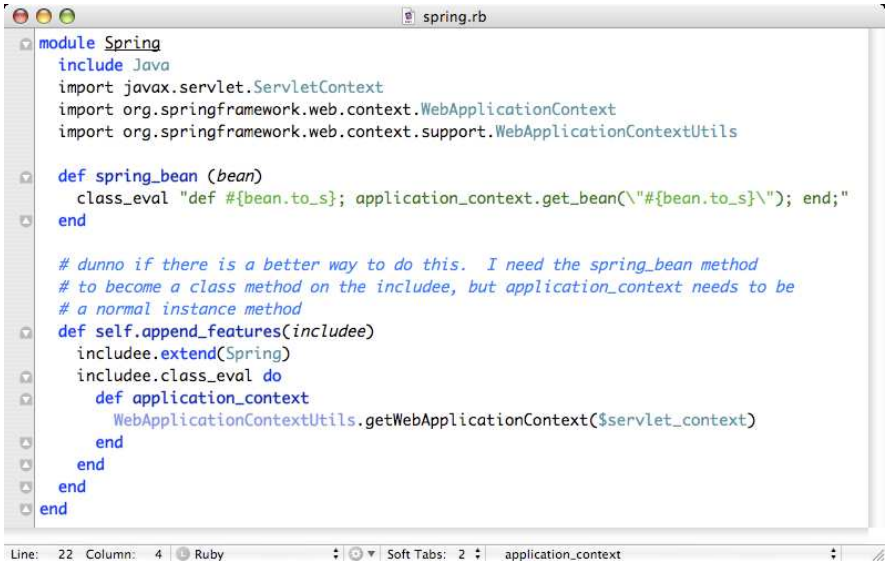

Figura 2: Módulo para integração do Spring.

Os *beans* do Spring podem ser acessados através de um controlador do Ruby:

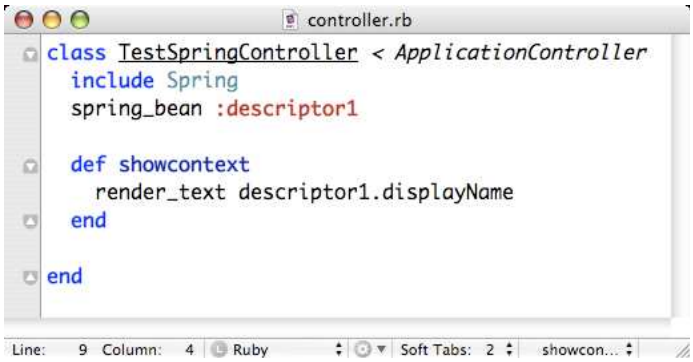

Figura 3: Controlador do Ruby acessando um bean do Spring.

4.2 Java Message Service (JMS)

A integração do JMS com o Ruby foi possível com a utilização da biblioteca do Apache ActiveMQ. Com isto, mensagens de clientes podem ser processadas com esta implementação do JMS através do JRuby. Abaixo segue o código utilizado na integração.

```
require "java"

include_class "org.apache.activemq.ActiveMQConnectionFactory"
include_class "org.apache.activemq.util.ByteSequence"
include_class "org.apache.activemq.command.ActiveMQBytesMessage"
include_class "javax.jms.MessageListener"

ENV['RAILS_ENV'] = 'development'
```

```
RAILS_ROOT=File.expand_path(File.join(File.dirname(__FILE__), '..','..'))
load File.join(RAILS_ROOT, 'config', 'environment.rb')

class MessageHandler include javax.jms.MessageListener

  def onMessage(serialized_message)
    message_body = serialized_messageed_message.get_content.get_data.inject("") { |body, byte|
body << byte }
    customer_payload = YAML.load(message_body)
    customer = Customer.new(:name => customer_payload.name)
    customer.id = customer_payload.id
    customer.save!
  end

  def run
    factory = ActiveMQConnectionFactory.new("tcp://localhost:61616")
    connection = factory.create_connection();
    session = connection.create_session(false, Session::AUTO_ACKNOWLEDGE);
    queue = session.create_queue("Customer");
    consumer = session.create_consumer(queue);
    consumer.set_message_listener(self);
    connection.start();
    puts "Listening..."
  end
end

handler = MessageHandler.new
handler.run
```

5 ARQUITETURA DA APLICAÇÃO

A aplicação da loja virtual foi desenvolvida seguindo a arquitetura clássica do *framework* Ruby on Rails, exatamente para demonstrar a capacidade de integração de uma aplicação que segue esta arquitetura a bibliotecas da linguagem Java através do interpretador JRuby.

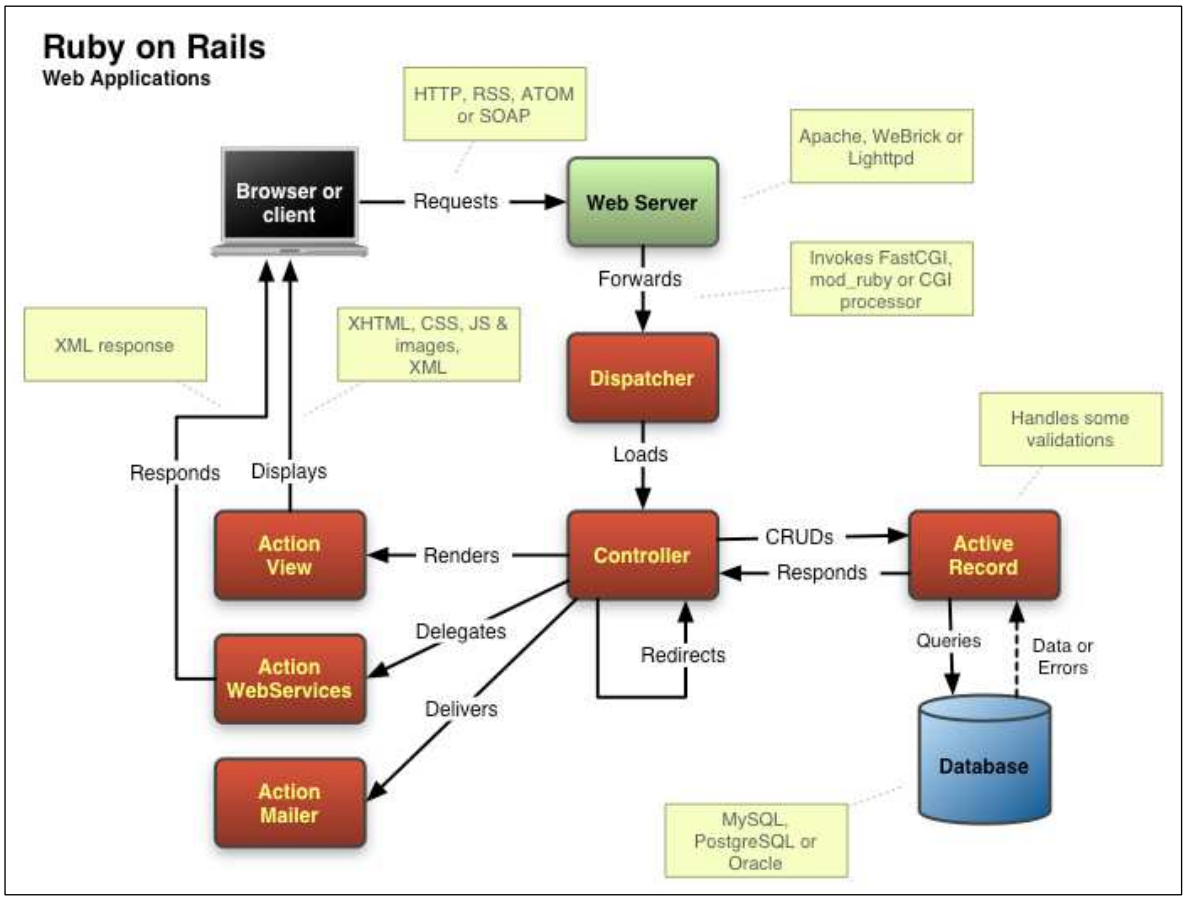

Figura 4: Arquitetura do Ruby on Rails

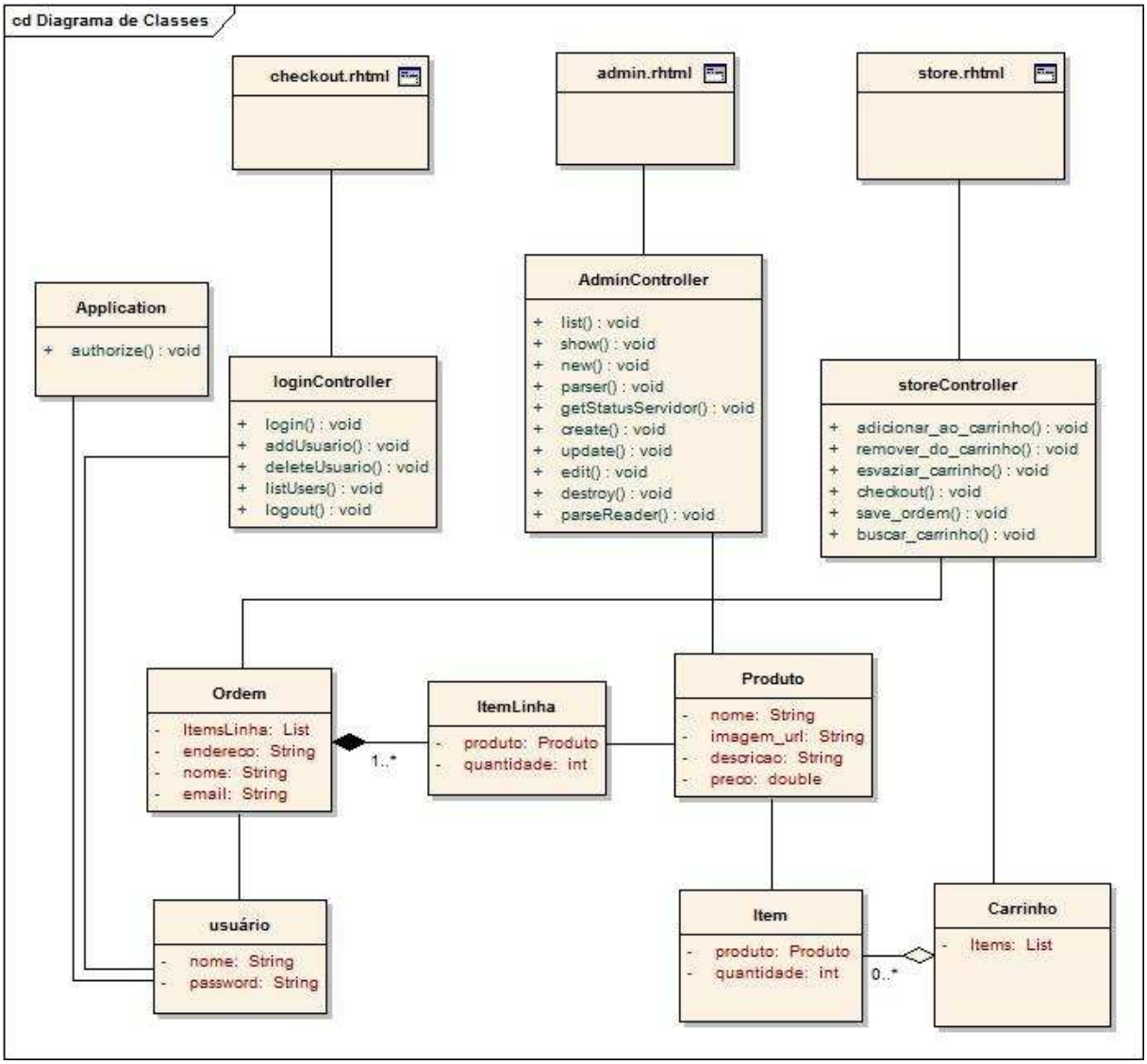

Figura 5: Diagrama de Classes da aplicação

Seguindo o padrão MVC gerado pelo framework, a aplicação apresenta portanto as camadas *model*, *view* e *controller*. Descrevemos brevemente nos tópicos seguintes o que representam as classes do diagrama acima:

5.1 Camada Model (Modelos)

Representa o domínio da aplicação e possui as seguintes classes:

Classe Produto – Representa um produto do site, com preço e descrição disponíveis na loja virtual.

Classe Carrinho – Carrinho de compras do usuário que possui uma lista de itens.

Classe Item – Representa um item do carrinho, vinculando um Produto a uma determinada quantidade.

Classe Ordem – Ordem de Compra.

Classe ItemLinha – Item da Ordem de Compra.

Classe Usuário – Usuário do sistema.

5.2 Camada Controller (Controladores)

Representa a camada onde são executadas as regras de negócio da aplicação. Possui as classes a seguir:

Classe storeController – Aqui são executadas as operações referentes ao carrinho de compras, adição ou remoção de um ou de todos os itens do carrinho e *checkout*.

Classe adminController – Aqui estão as operações de *parser* e geração de arquivos XML com as listas de produtos e as operações de adição e remoção de produtos na loja virtual.

Classe loginController – Aqui estão as operações referentes aos usuários do sistema.

5.3 Camada View (Visualização)

Possui as seguintes páginas HTML com a visualização das informações processadas pelos controladores:

Store.rhtml – Visualização das operações da classe storeController.

Admin.rhtml – Visualização das operações da classe adminController.

Checkout.rhtml - Visualização das operações da classe *loginController*.

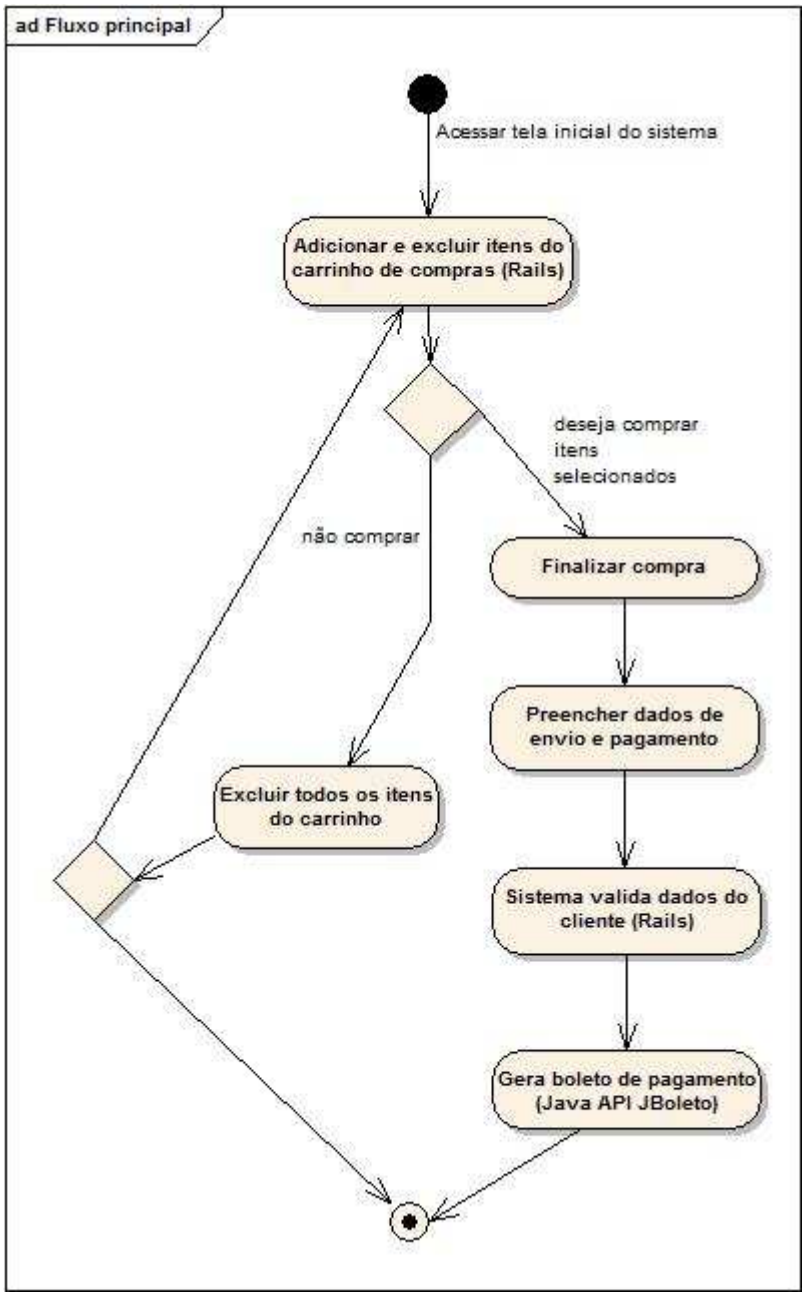

Figura 6: Fluxo Principal da Aplicação. Mostra o caminho percorrido pelo usuário ao adquirir um produto em nossa loja virtual e quais linguagens e APIs estão sendo utilizadas nas principais etapas processo.

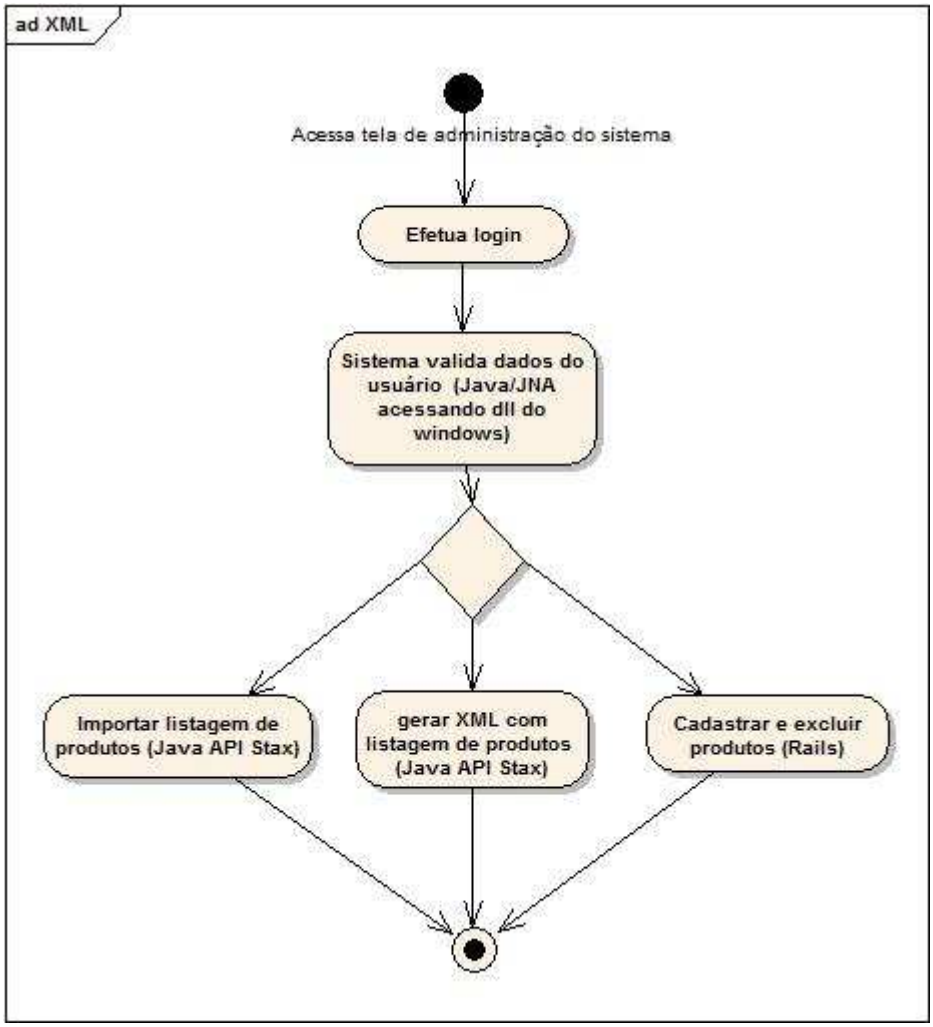

Figura 7: Fluxo da parte administrativa da aplicação. Demonstra as ações possíveis para um administrador do sistema e quais as linguagens e APIs utilizadas em cada funcionalidade.

6 A BIBLIOTECAS DA LINGUAGEM JAVA UTILIZADAS NA APLICAÇÃO

6.1 APIS para processamento de XML em Java

A linguagem XML, segundo definição da W3C (*The World Wide Web Consortium*) foi desenvolvida para '*transportar e armazenar dados, com foco no que os dados são (representam)*' e não pode ser confundida com a conhecida linguagem HTML (*Hypertext Markup Language*) que é destinada à '*visualização dos dados, com foco em como os dados são exibidos*', segundo definição também da W3C, que é a organização responsável pelos padrões da *World Wide Web*.

O XML é hoje uma linguagem utilizada para o transporte de dados entre os mais diversos tipos de aplicação e com isso o processamento de arquivos no formato XML se tornou uma questão crítica para grande parte das aplicações WEB.

Na linguagem Java a manipulação de arquivos XML (geração e "*parsing*", ou extração do conteúdo de um XML) pode ser tratada com diversas bibliotecas (ou APIs - *Application Programming Interface*, que é a nomenclatura utilizada neste trabalho), como o SAX (*Simple API for XML*), a DOM (*Document Object Model*) e a StAX (*Streaming API for XML*).

A API DOM é um modelo de *parser* de XML baseado em árvore, pois gera um modelo em memória de toda a estrutura do arquivo XML e, portanto consome muitos recursos do sis-

tema, porém permitindo maior liberdade para manipulação dos elementos e atributos e visualização da estrutura do arquivo.

A API SAX é um modelo de *parser* de XML baseado em eventos, sendo indicada para situações em que os objetivos são um "*parsing*" rápido do XML, verificar se ele é bem formado (se ele cumpre as normas estabelecidas para o padrão XML), se é válido (segue os padrões definidos por um DTD – *Document Type Definition* ou por um XML Schema) e em seguida extrair as informações do documento enquanto o processo é executado. Apesar de ser bem mais eficiente que o DOM, a geração automática de eventos enquanto o arquivo é processado obriga a aplicação a processá-los, o que também gera algum custo em termos de recursos da aplicação.

Tanto a API DOM como a SAX processam o arquivo XML como um todo, fornecendo muito pouco controle ao programador. Chegamos então ao foco do nosso trabalho que será a API StAX, que apresenta um ótimo desempenho para o "*parsing*" de XML, por não gerar eventos automaticamente como o SAX ou montar uma estrutura do XML em memória como acontece com o DOM. Os pacotes utilizados para isto são o *javax.xml.stream* e o *javax.xml.stream.events*.

Característica	**StAX**	**SAX**	**DOM**
Tipo	Captura de eventos; Streaming	Eventos	XML em memória baseado em árvore
Facilidade de Uso	Alta	Média	Alta
Utilização de Memória e CPU	Boa	Boa	Depende da situação

Tabela 1 – Tabela comparativa entre as APIs de processamento de XML em Java

6.2 API StAX

A API StAX foi introduzida na especificação da linguagem Java através da norma JSR 173 de março de 2004 e pela sua atualidade é menos abordada em livros sobre XML do que as bibliotecas mais antigas, DOM e SAX.

A StAX possui duas maneiras diferentes de executar o "*parsing*" de um documento XML que seriam a baseada em cursor utilizando a interface *XMLStreamReader* e a baseada em um iterador com a interface *XMLEventReader*.

A maneira como o "*parsing*" é executado pelo StAX garante ao programador um total controle sobre como os eventos são processados e é mais indicado assim para arquivos que são transmitidos por uma rede, notadamente os '*web services*', também pelo fato de consumir menos memória.

Para efetuar o "*parsing*", ou seja, a leitura do XML, na metodologia baseada em cursor é utilizado um objeto *FileInputStream* para localizar o arquivo XML e um *XMLStreamReader* para a leitura deste arquivo.

```
FileInputStream fileInputStream = new FileInputStream(fileLocation);
XMLStreamReader xmlStreamReader = XMLInputFactory.newInstance().
createXMLStreamReader(fileInputStream);
```

Após é utilizada uma iteração para obter os elementos do XML enquanto existirem.

```
while (xmlStreamReader.hasNext()) {
printEventInfo(xmlStreamReader);
 }
```

```
xmlStreamReader.close();
```

Aqui foi demonstrado como a StAX fornece ao programador a possibilidade de processar somente os eventos que forem desejados e extrair as informações dos atributos pertinentes. O *parser* só prossegue no momento que o próximo evento é chamado, ao contrário das demais APIs mencionadas que processam o XML como um todo. Abaixo um exemplo de como o processamento pode ser feito para cada nó do XML.

```
int eventCode = reader.next();
    switch (eventCode) {

    case XMLStreamConstants.START_ELEMENT :
           System.out.println("event = START_ELEMENT");
           System.out.println("Localname = "+reader.getLocalName());
           break;

    case XMLStreamConstants.END_ELEMENT :
           System.out.println("event = END_ELEMENT");
           System.out.println("Localname = "+reader.getLocalName());
           break;

    case XMLStreamConstants.PROCESSING_INSTRUCTION :
           System.out.println("event = PROCESSING_INSTRUCTION");
           System.out.println("PIData = " + reader.getPIData());
           break;
```

Para a geração de XML a API utiliza o objeto *XMLStreamWriter*, como exemplo abaixo.

```
XMLStreamWriter writer = XMLOutputFactory.newInstance().
        createXMLStreamWriter(outStream);
```

6.3 API JNA

A API JNA é destinada ao acesso a bibliotecas nativas do sistema operacional, como as DLLs do Windows, sem a necessidade de conhecimento de outras linguagens de programação além do Java. A API simplesmente carrega a biblioteca e usa uma invocação de método em Java para acessar as funções fornecidas.

Após a biblioteca ser importada pode-se usar a chamada estática "*Native*" para carregá-la em memória, conforme o exemplo seguinte.

```
Kernel32 INSTANCE = (Kernel32)
Native.loadLibrary("kernel32", Kernel32.class);
```

Depois se pode criar um objeto que faça chamadas para funções da biblioteca nativa, neste caso uma DLL do Sistema Windows para retornar valores para a aplicação Java.

```
Kernel32 lib = Kernel32.INSTANCE;
SYSTEMTIME time = new SYSTEMTIME();
lib.GetSystemTime(time);
System.out.println("Today's integer value is " + time.wDay);
```

6.4 APIS JBoleto e JBarCodeBean

O JBoleto é uma biblioteca desenvolvida na linguagem Java, por brasileiros, e que permite a geração de boletos bancários para diversos bancos (conforme tabela abaixo, extraída da documentação da API). O JBoleto é de código aberto (*open source*), licenciado sob a licença GNU LGPL. Para a utilização da biblioteca numa aplicação Java basta incluí-la no *classpath* do projeto.

BancoBrasil	Classe responsável em criar os campos do Banco do Brasil
BancoModelo	Classe modelo para a criação de outros. Copie este arquivo para o nome do banco que pretende criar, seguindo a mesma nomenclatura de nomes das outras classes. Caso o banco tenha algum método diferente de calcular os seus campos, pode-se criar os seus métodos particulares dentro desta classe, pois os métodos dessa classe são padrões.
BancoReal	Classe responsável por gerar o boleto do Banco Real
Bradesco	Classe responsável em criar os campos do Banco Bradesco.
CaixaEconomica	Classe responsável em criar os campos da Caixa Econômica.
Hsbc	Classe para gerar o boleto do HSBC.
Itau	Classe responsavel em criar os campos do Banco Itaú.
NossaCaixa	Classe responsavel em criar os campos do Banco Nossa Caixa.
Santander	Classe para gerar o boleto do Banco Santander
Unibanco	Classe para gerar o boleto do Banco Unibanco

Tabela 2 – Tabela com algumas classes do JBoleto.

A API JBarcodeBean é destinada à geração de código de barras em boletos de cobrança também para a linguagem Java, podendo ser utilizadas em aplicativos desktop ou web, gerando códigos de barra nos padrões demonstrados abaixo, consideradas as mais populares segundo os próprios desenvolvedores da biblioteca.

Codabar

Code128

Code11

Interleaved 2 of 5

Code39

Demonstraremos em outros capítulos deste trabalho, a utilização do JBoleto e JBarCodeBean em uma aplicação JRuby.

6.5 API iText

O iText é uma biblioteca (API) para a geração de arquivos PDF (*Portable Document Format*), especialmente recomendada para aplicações web. Segundo seus desenvolvedores pode ser utilizada nas seguintes situações:

- Devido ao tempo ou tamanho, o arquivo não pode ser produzido manualmente.
- O conteúdo do documento deve ser calculado ou baseado na entrada do usuário.
- O conteúdo precisa ser personalizado
- O PDF precisa ser disponibilizado em um ambiente web
- Documentos precisam ser criados por um processo ‘*batch’* (em lote)
- Geração de arquivos PDFs a partir de XMLs ou bancos de dados

7 DESENVOLVIMENTO DA APLICAÇÃO

7.1 Aplicação

Para a realização prática deste trabalho de conclusão de curso, foi desenvolvida uma aplicação que exemplifica a integração das linguagens Ruby e Java, que possibilitando agregar a facilidade do código Ruby com as importantes funcionalidades e bibliotecas do Java.

O intuito não foi de apenas adicionar bibliotecas para demonstrar a integração entre as duas linguagens, e sim, adicionar bibliotecas que ainda não houvessem sido integradas por nenhum outro desenvolvedor Ruby nem pela equipe de desenvolvimento do JRuby; além disso, optou-se pela integração de bibliotecas que resultariam em uma funcionalidade interessante à aplicação.

Outro ponto importante que merece ser ressaltado é que todas as vantagens e facilidades oferecidas pela linguagem Ruby somadas ao seu framework Ruby on Rails foram mantidas. Por isso, optou-se por não integrar nenhuma API da especificação JavaEE e mostrar o quão eficiente e prático é esse *framework*, que apresenta um conjunto de funcionalidades que supera, em alguns aspectos, o uso dos principais *frameworks* e bibliotecas da linguagem Java utilizados na maioria das aplicações Web nos dias atuais, como Spring, Hibernate e Struts.

A aplicação escolhida consiste em um Web site de comercio eletrônico, totalmente escrito em linguagem Ruby, implantada em um servidor Glassfish, com chamadas de códigos Java. O intuito disso é mostrar a possibilidade de comunicação entre as duas linguagens utilizadas.

As principais integrações desenvolvidas, que serão mais detalhadas nas próximas seções, serviram não apenas para agregar maior funcionalidade e segurança ao sistema, como, principalmente, demonstrar que é possível utilizar qualquer biblioteca desenvolvida para a linguagem Java em código Ruby através do interpretador JRuby.

Dentre as bibliotecas integradas destacam-se:

- Stax (*Streaming API for XML*) – criação de um *parser* XML para a importação e exportação dos produtos do site de comércio eletrônico;
- JNA (*Java Native Access*) – chamada de DLLs nativas do sistema operacional para realização de algumas funcionalidades;
- Jboleto, JBarcodeBean e Itext – responsáveis pela geração de um boleto bancário ao fim do processo de compra;

A aplicação apresenta duas etapas principais. A primeira é o comercio eletrônico em si, onde são apresentados na tela todos os produtos da loja e, à medida que o usuário vai adicionando os produtos no carrinho é atualizado o valor total da compra e a quantidade de cada produto. Nesta parte, o usuário também tem a possibilidade de esvaziar o carrinho, remover um item ou finalizar a compra.

Ao finalizar a compra, o usuário deverá informar alguns dados, como nome e e-mail e confirmar o pedido. Com isso, caso esta confirmação seja efetuada com sucesso, o sistema deverá gerar um boleto e informar que a compra foi finalizada. Após isso o carrinho de compras é esvaziado, possibilitando a compra de novos produtos.

A outra etapa é de uso administrativo, onde para entrar é necessário informar o *login* de um usuário que possua acesso a essa funcionalidade. Neste ponto, é possível ver a lista de usuários cadastrados, assim como criar, alterar ou excluir um usuário. É possível, também, manter a lista de produtos (alterar, adicionar ou excluir) e ver a quantidade de ordens de compras do dia.

É nessa parte que se encontra a maioria das funcionalidades que usam recursos de algumas bibliotecas específicas da linguagem Java. Dentre essas, podemos destacar a importação

e exportação de produtos para o formato XML, a validação de usuário e verificação do espaço livre do disco rígido do servidor.

7.2 Geração da estrutura inicial da aplicação no padrão MVC utilizando o Rails

A geração da estrutura básica da aplicação foi realizada através do comando “Scaffold” do *framework* Rails com parte do código e funcionalidades essenciais baseado em exemplos disponíveis na Internet e mencionados nas referências ao final do trabalho e no código fonte.

7.3 Integração com a API StAX

Duas funcionalidades desenvolvidas para essa aplicação foram a de geração e importação de um arquivo XML, onde o administrador do sistema poderá gerar um arquivo com todos os produtos da base e importar uma lista com novos produtos, eventualmente recebida de um fornecedor.

Essas funcionalidades facilitariam a troca de informações para adição e/ou atualização dos produtos de maneira mais dinâmica e rápida.

A ferramenta utilizada para realização desse *parser* XML foi a API Stax (http://stax.codehaus.org), que será vista mais adiante. Para isso, foi necessário adicionar essa biblioteca no *classpath* da aplicação e importar algumas classes das APIs do próprio Java, como mostrado a seguir:

```
require 'java'

import javax.xml.stream.XMLOutputFactory;

import javax.xml.stream.XMLInputFactory;

import javax.xml.stream.XMLStreamWriter;
```

```
import javax.xml.stream.XMLStreamReader;

import javax.xml.stream.XMLEventReader;

import javax.xml.stream.XMLStreamException;

import java.io.fileOutputStream;

import java.io.fileInputStream;

import java.io.fileWriter;
```

Para a criação de um arquivo XML, contendo todos os produtos da base para exportação, foi criada uma função que pega todos os produtos do banco de dados, cria um arquivo de saída e instancia o elemento responsável pela criação de um arquivo XML, como pode ser visto a seguir:

```
def criarArquivo

      produtos = Produto.find(:all)

      out = java.io.FileOutputStream.new "C:/catalog.xml"

      outFactory = XMLOutputFactory.newInstance()

      writer = outFactory.createXMLStreamWriter out

      i=0
```

Nesse trecho, obtém-se do banco a lista com todos os produtos cadastrados, define-se o diretório e o nome do arquivo que será criado e cria-se o objeto responsável pela montagem do XML.

```
      writer.writeStartDocument("ISO-8859-1", "1.0")

      writer.writeStartElement("Produtos")

      while i < produtos.length

            produto = produtos[i]

            writer.writeStartElement("Produto")

            writer.writeStartElement("Descricao")

            writer.writeCharacters(produto.descricao)

            writer.writeEndElement()

            writer.writeStartElement("Preco")
```

```
            writer.writeCharacters(JavaLang::String.valueOf(produto.preco))
            writer.writeEndElement()
            writer.writeStartElement("Titulo")
            writer.writeCharacters(produto.titulo)
            writer.writeEndElement()
             writer.writeStartElement("URL_Imagem")
            writer.writeCharacters(produto.imagem_url)
            writer.writeEndElement()
            writer.writeEndElement()
            i+=1
      end
      writer.writeEndElement()
      writer.writeEndDocument()
      writer.flush()
      writer.close()
      out.close()
         flash[:notice] = 'PRODUTO(S) EXPORTADO(S) COM SUCESSO!!'
    end
```

No trecho acima, é criado um conjunto de *tags* no arquivo XML, contendo a descrição, título, preço e caminho da imagem de cada produto. Além disso, é colocado o padrão utilizado para codificação de caracteres (ISO-8859-1) e, ao final, o arquivo de saída é fechado e o processo é terminado.

Já para a importação de novos produtos foi criada uma função que carrega um arquivo de entrada e instancia os elementos responsáveis pela leitura do arquivo XML. Após todos os dados de cada produto serem lidos, o mesmo é criado no banco de dados, como é mostrado a seguir:

```
def importarArquivo
     fileInputStream = java.io.FileInputStream.new "C:/import.xml"
```

```
inputFactory = XMLInputFactory.newInstance()

reader = inputFactory.createXMLEventReader fileInputStream

while (reader.hasNext)

  evento = reader.nextEvent()

  if  (evento.isStartElement())

     if evento.asStartElement().getName().getLocalPart()=="Descricao"

        evento = reader.nextEvent();

        descricao = evento.asCharacters().getData()

        redo

     end

     if evento.asStartElement().getName().getLocalPart()=="Preco"

        evento = reader.nextEvent();

        preco = evento.asCharacters().getData()



        redo

      end



     if evento.asStartElement().getName().getLocalPart()=="Titulo"

        evento = reader.nextEvent();

        titulo = evento.asCharacters().getData()

        redo

     end



     if evento.asStartElement().getName().getLocalPart()=="URL_Imagem"

        evento = reader.nextEvent();

        url_imagem = evento.asCharacters().getData()

        redo
```

```
            end

        end
```

Nesta parte de código mostrada, é criado um objeto que representa o arquivo de entrada (um XML que contém produtos a serem importados) e um objeto responsável pela leitura desse mesmo arquivo que verifica cada *tag* do arquivo e armazena em variáveis aquelas que farão parte da construção de um novo produto da aplicação de comércio eletrônico.

Já no trecho que virá a seguir, é feita uma verificação pra ver se todas as variáveis possuem algum valor e, então, um novo produto é criado na base de dados e as variáveis recebem valores nulos para que continue a repetição e leitura de todo o arquivo.

```
        if descricao != nil && preco != nil && titulo != nil && url_imagem!=nil

            produto = Produto.new(:descricao => descricao, :preco => preco,    :titulo
=>titulo, :imagem_url =>url_imagem)

            produto.save(false)

            descricao = nil



            preco = nil

            titulo = nil

            url_imagem = nil

            end

           reader.next

        end



          flash[:notice] = 'PRODUTO(S) IMPORTADO(S) COM SUCESSO!!'

      end
```

7.4 Integração com a API JNA

Outra API integrada à aplicação foi a JNA – *Java Native Access*, que tem como característica a chamada de DLLs dos sistemas operacionais, como será mostrado mais adiante.

Na nossa aplicação essa API é utilizada em dois momentos, que serão tratados nos próximos tópicos.

7.5 Autenticação com JNA

A API foi utilizada no *login*, para acesso à parte administrativa do sitema por um usuário responsável pela manutenção e cadastro de produtos e usuários. Neste caso, é validado se este usuário é o mesmo que está logado também no sistema operacional.

Para essa funcionalidade, foi necessário o uso da DLL advapi32, do sistema operacional Windows. Advapi32.dll é uma parte de uma API de serviços avançados de apoio à segurança e registros do sistema operacional.

Segue trecho do código:

```
advapi32 = com.sun.jna.NativeLibrary.getInstance('advapi32')
 getUser = advapi32.getFunction('GetUserNameA')
 num = getUser.invokeInt([byte,ref].to_java)
 nmUsuarioWindows = JavaLang::String.new(byte, 0, ref.getValue()-1)
 if nmUsuarioWindows != usuario.nome
    flash[:notice] = "Usuário deve ser o mesmo que está logado neste computador"
 else
    session[:usuario_id] = usuario.id
 redirect_to(:action => "index")
 end
```

No trecho mostrado, é instanciado um elemento responsável pelo acesso às funções da DLL, acessando a função “GetUserNameA”, que retorna o nome do usuário logado no SO. Após isso é feita uma comparação entre este nome e o *login* informado pelo usuário.

7.6 Obtenção do Status do Servidor com JNA

Após um usuário administrador realizar o *login* do sistema, o mesmo terá acesso a uma funcionalidade que informa o espaço livre do disco rígido da máquina servidora da aplicação. Essa informação é capturada através da DLL do Windows Kernel32, que é chamada através da API JNA.

O Kernel32.dll é a biblioteca que lida com as principais tarefas do Windows. Entre elas, encontram-se o gerenciamento de memória e disco rígido, as interrupções e operações de entrada/saída do sistema.

A seguir um trecho relevante do código:

```
raiz = JavaLang::String.new("C:\\")

kernel32 = com.sun.jna.NativeLibrary.getInstance('kernel32')

getDiskFreeSpace =  kernel32.getFunction('GetDiskFreeSpaceExA')

utilizado = LongByReference.new

total = LongByReference.new

livre = LongByReference.new

num = getDiskFreeSpace.invokeInt([raiz, utilizado, total, livre].to_java)

utilizado_valor =  utilizado.value

total _valor = total.value

livre_valor = livre.value
```

Assim como na função demonstrada anteriormente, um objeto responsável pelo acesso às funções da DLL é criado para que, o mesmo, chame a função “GetDiskFreeSpaceExA”, que retorna os totais de espaço livre, utilizado e total do disco rígido do servidor.

7.7 Integração com as APIS JBoleto, JbarCodeBean e iText

Outra funcionalidade disponibilizada na aplicação foi a de geração de um boleto bancário ao final do processo de fechamento de um pedido, contendo o valor total do mesmo.

O boleto contém dados fictícios (o único dado dinâmico é o valor), servindo apenas para demonstrar a possibilidade de uso das bibliotecas JBoleto, JBarcodeBean e Itext, que são muito utilizadas pelos desenvolvedores Java e que serão tratadas mais adiante.

Assim como nas integrações anteriores, foi necessário adicionar essas bibliotecas no *classpath* da aplicação e importar algumas classes do Java no código Ruby, como mostrado a seguir:

```
import org.jboleto.JBoleto

import org.jboleto.JBoletoBean

import org.jboleto.control.Generator

import org.jboleto.control.PDFGenerator
```

Já no método responsável pelo fechamento da compra, foi adicionado o seguinte trecho de código, que irá gerar um boleto do Banco do Brasil contendo o valor total do carrinho de compras:

```
jBoletoBean =org.jboleto.JBoletoBean.new

 jBoletoBean.setDataDocumento("21/04/2009")

 jBoletoBean.setDataProcessamento("21/04/2009")

 jBoletoBean.setCedente("Ruby")



 jBoletoBean.setNomeSacado("Ruby ")

 jBoletoBean.setEnderecoSacado("Rua Teste")

 jBoletoBean.setBairroSacado("Kobrasol")
```

```
                    jBoletoBean.setCidadeSacado("Sõ José")

                    jBoletoBean.setUfSacado("SC")

                    jBoletoBean.setCepSacado("88102-090");

                    jBoletoBean.setCpfSacado("0000000000000")

                    jBoletoBean.setLocalPagamento("ATE O VENCIMENTO, PREFERENCIALMENTE NO BAN-
CO DO BRASIL")

                    jBoletoBean.setLocalPagamento2("APOS O VENCIMENTO, SOMENTE NO BANCO DO
BRASIL")

                    jBoletoBean.setDataVencimento("10/06/2006")

                    jBoletoBean.setInstrucao1("APOS O VENCIMENTO COBRAR MULTA DE 2%")

                    jBoletoBean.setInstrucao2("APOS O VENCIMENTO COBRAR R$ 0,50 POR DIA DE A-
TRASO")

              jBoletoBean.setInstrucao3("")

                    jBoletoBean.setInstrucao4("")

                    jBoletoBean.setCarteira("XX")

                    jBoletoBean.setAgencia("XXXX")

                    jBoletoBean.setContaCorrente("XXXXXX")

                    jBoletoBean.setNumConvenio("XXXXXXXX");



                    jBoletoBean.setNossoNumero("XXXXXXXXXX",10)

                     @carrinho= session[:carrinho]

                    jBoletoBe-
an.setValorBoleto(JavaLang::String.valueOf(@carrinho.preco_total))

                     generator = org.jboleto.control.PDFGenerator.new(jBoletoBean, 001)

                     jBoleto = org.jboleto.JBoleto.new(generator, jBoletoBean, 001)

                    jBoleto.addBoleto();

                    jBoleto.closeBoleto("banco_brasil_teste.pdf");
```

No trecho acima, um objeto responsável pela criação do boleto é instanciado e são passadas a ele todas as informações necessárias para a geração, incluindo o valor total da com-

pra. Este objeto cria, no diretório do sistema, um arquivo no formato PDF contendo a imagem de um boleto - neste caso, do Banco do Brasil.

Ao longo de praticamente todas as integrações foi necessário o uso de algumas funcionalidades das próprias bibliotecas do Java, que vêm como padrão na sua JDK. Dentre os pacotes adicionados destacam-se o *java.lang*, que contém as classes que constituem recursos básicos da linguagem, necessários à execução de qualquer programa Java; e *java.io*, responsável pela entrada e saída de dados em uma aplicação Java.

7.8 Bibliotecas com dificuldades de integração

Ao desenvolver a aplicação, algumas bibliotecas foram testadas, como a API de Segurança do Java, o JAAS. Ao tentarmos promover a utilização destas bibliotecas nos deparamos com diversas dificuldades, como exceções indeterminadas causadas por problemas ainda conhecidos no próprio interpretador JRuby, além de uma infinidade de arquivos XML para configurar.

Estas dificuldades nos levaram a abandonar a idéia da autenticação da aplicação com JAAS não só pelos problemas apresentados pelo interpretador como pela própria configuração destes arquivos XML ir contra o próprio princípio do trabalho que busca a simplicidade do Ruby on Rails. A solução do login com autenticação pelo Sistema Operacional com JNA foi a alternativa encontrada.

8 CONCLUSÃO

Nos últimos anos, a agilidade tornou-se fundamental em qualquer negócio, devido principalmente ao mercado globalizado, com sua infinidade de empresas competindo entre si de todos os cantos do planeta, as quais geralmente se utilizam de alguma aplicação para gerar, executar, armazenar, organizar, enfim, fazer com que seus objetivos sejam atingidos da melhor maneira possível, a fim de manter o funcionamento da sua estrutura, otimizar processos e aumentar seus lucros.

Além da agilidade, outros fatores como robustez, segurança e praticidade formam as palavras-chave no desenvolvimento de uma aplicação qualificada a exercer qualquer função essencial para estas organizações.

Foi com o intuito de diminuir as dificuldades e agilizar o processo de desenvolvimento de software que surgiu a linguagem Ruby e, posteriormente, seu mais conhecido *framework*, o Ruby on Rails. Juntos, eles formam uma poderosa estrutura para o desenvolvimento ágil de qualquer aplicação.

Com o amadurecimento do interpretador JRuby, tornou-se possível a integração entre o Ruby e a linguagem Java, com sua ampla comunidade de desenvolvedores, uma enorme fonte de referências e inúmeras bibliotecas. Estas últimas exercem fundamental importância em várias aplicações, reduzindo o tempo de desenvolvimento e o grau de dificuldade na implementação de quaisquer funcionalidades.

Ao desenvolver este trabalho chegamos à conclusão de que, com a integração destas linguagens e a capacidade de utilizar estes aplicativos como módulos, partes integrantes de sistemas já desenvolvidos em Java sem a necessidade de retrabalho, ou o desenvolvimento de

um sistema novo em Ruby com bibliotecas Java agregadas, além da possibilidade de implantação destes aplicativos em servidores que seguem a especificação Java, que possui ampla aceitação e utilização pelo mercado, são fatores que podem ser levados em consideração na utilização destas linguagens da forma apresentada aqui, através do interpretador JRuby.

Em contrapartida também constatamos que a utilização de algumas bibliotecas ainda depende de uma evolução do interpretador, o que também deve ser considerado e pode ser um fator decisivo na opção pelo uso da linguagem Ruby dentro da plataforma nova nesse momento inicial.

8.1 Objetivos alcançados

Podemos concluir com base em nossos estudos que os objetivos principais foram alcançados, pois, conseguimos compreender a linguagem, integrar algumas bibliotecas Java na aplicação desenvolvida e fazer com que esta última se tornasse uma aplicação que poderá servir como material raro de estudo aos iniciantes na programação Ruby.

Cabe ressaltar, no entanto, as dificuldades enfrentadas para execução do trabalho. Dentre as maiores dificuldades encontradas podemos destacar a falta de material de apoio, devido, principalmente, à pequena, porém crescente, comunidade Ruby e a não tão fácil adaptação a sua sintaxe.

Entender os princípios da linguagem e sua sintaxe, assim como desenvolver a aplicação base em Ruby foi a etapa mais crítica de todo o desenvolvimento deste trabalho de conclusão de curso. Etapa esta que necessitou de um bom tempo de dedicação e aprendizagem, e que era fundamental para a continuidade e o sucesso do trabalho proposto.

8.2 Trabalhos futuros

Pretendemos acompanhar a evolução e melhorias do interpretador JRuby a fim de determinar o ponto em que este será estável o suficiente para a utilização das principais bibliotecas da linguagem Java sem problemas e exceções.

REFERÊNCIAS BIBLIOGRÁFICAS

GAMA, Gustavo. *Ruby on Rails.* Abril 2008. Disponível em: <http://blog.vettalabs.com/2008/04/07/ruby-on-rails/>. Acesso em: 16 de novembro de 2009.

AKITA, Fabio. *Ruby: Sucesso pela arrogância?* . Dez 2006. Disponível em: <http://www.rubyonbr.org/articles/2006/12/06/rails-sucesso-pela-arrogncia/>, Acesso em: 16 de novembro de 2008.

OFICINA DA NET. *Linguagem Ruby on Rails Ganha Forca na Web 2.0. Março 2007*. Disponível em: <http://www.oficinadanet.com.br/noticias_web/593/linguagem_ruby_on_rails_ganha_forca_na_web_2.0>. Acesso em: 14 de maio de 2008.

LOCAEWB. *MVC e Ruby on Rails, uma visão simplificada.* Jun 2008. Disponível em: <http://tecblog.locaweb.com.br/2008/06/23/mvc-e-ruby-on-rails-uma-visao-simplificada/>. Acesso em: 11 de outubro de 2008.

WIKIPEDIA. *Ruby on Rails*. Ago 2008. Disponível em: <http://pt.wikipedia.org/wiki/Ruby_on_Rails>. Aceso em: 07 de setembro de 2008.

BETTEREXPLAINED. *Starting ruby on rails what I wish*. Jun 2007. Disponível em: <http://betterexplained.com/articles/starting-ruby-on-rails-what-i-wish-i-knew/>. Acesso em: 19 de setembro de 2008.

MENEZES, José. *Tudo o que webmasters devem saber sobre RoR*. Março 2007. Disponível em: <http://www.plugmasters.com.br/sys/materias/628/1/Tudo-o-que-webmasters-devem-saber-sobre-RoR>. Acesso em: 16 de novembro de 2008.

SALVADOR, Fábio. *PHP e Ruby ameaçam liderança da linguagem Java, mas esta continua na frente*. Maio 2008. Disponível em: <http://www.melhordetodos.com.br/index.php?codigo=340>. Acesso em: 02 de maio julho de 2008.

BODDEM, Brian. *Rails on Spring*. Disponível em <www.integrallis.com/downloads/rails_on_spring.pdf>. Acesso em: 13 de julho de 2008.

W3C. *Extensible Markup Language (XML)*. Disponível em: <www.w3.org/XML/>. Acesso em: 16 de abril de 2009.

FEDERIZZI, Gustavo. *APIs para XML*. Disponível em: <http://www.inf.ufrgs.br/procpar/disc/inf01008/trabalhos/sem01-1/t2/apis_xml_java/>. Acesso em: 16 de abril de 2009.

CHINTHAKA, Eran. *Introduction to StAX*. Jun 2007. Disponível em: <http://wso2.org/library/1844>. Acesso em 15 de abril de 2009.

JNA. *Java Native Access*. Disponível em: https://jna.dev.java.net/. Acesso em: 18 de abril de 2009..

JBARCODEBEAN. *About JbarCodeBean*. Disponível em: <http://jbarcodebean.sourceforge.net/intro.html>. Acesso em: 18 de abril de 2009.

JBOLETO. *JBoleto*. Disponível em: <http://www.jboleto.org>. Acesso em: 18 de abril de2009.

ITEXT. *Project Description*. Disponível em: <http://www.lowagie.com/iText/>. Acesso em: 18 de abril de2009.

FOX, Joshua. Ruby for the Java World. Jul 2006. Disponível em: <http://www.javaworld.com/javaworld/jw-07-2006/jw-0717-ruby.html>. Acesso em: 02 de junho de 2008.

WIKIJRUBY. Calling Java from Ruby. Maio 2009. Disponível em: <http://wiki.jruby.org/wiki/Calling_Java_from_JRuby>. Acesso em: 25 de janeiro de 2009.

INTEGRAÇÃO DE BIBLIOTECAS JAVA EM APLICAÇÃO DESENVOLVIDA COM O FRAMEWORK RUBY ON RAILS

Danny Pereira Mattos
Francis Novello Santos

Bacharelado em Sistemas de Informação, 2009
Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC), Brasil, 88040-900
{danny}{francisn}@inf.ufsc.br

Resumo

Ruby surgiu como uma linguagem de programação extremamente prática e, juntamente com o framework Ruby on Rails, formou uma das mais poderosas e ágeis tecnologias de desenvolvimento, despertando o interesse de um grande número de desenvolvedores que passaram a estudá-la e acompanhar sua evolução. A integração do Rails às amplamente utilizadas bibliotecas da linguagem Java é nosso objeto de estudo.

Palavras-chave: Ruby, Ruby on Rails, Integração, Java, JRuby

Abstract

Ruby appears as a programming language extremely practice and, with its framework Ruby on Rails, become one of the most powerful and agile development technologies, raising the interest of a great number of developers that started to study and see its evolution. The integration of Rails with the widely used Java libraries are our object of study here.

Keywords: Ruby, Ruby on Rails, Integration, Java, JRuby

INTRODUÇÃO

A linguagem de programação Java tem se apresentado, nos últimos anos, como uma das principais linguagens de programação utilizadas pelos desenvolvedores em todo o mundo, segundo *TIOBE Programming Community Index* (JANSEN, 2008). Com isso, também foi crescente o desenvolvimento de bibliotecas úteis na linguagem que facilitaram as necessidades básicas e avançadas no desenvolvimento de sistemas.

A linguagem Ruby, diferentemente do Java, é uma linguagem de programação interpretada (MATSUMOTO, 1995), com tipagem dinâmica e forte, totalmente orientada a objetos (enquanto Java é uma linguagem mista que aproveita conceitos da programação estruturada e orientada a objetos). Foi criada pelo japonês Yukihiro Matsumoto, que aproveitou as melhores idéias das outras linguagens da época, procurando criar uma linguagem intuitiva, de codificação rápida.

Com o surgimento do framework Ruby on Rails, a linguagem Ruby ganhou grande repercussão e muitos especialistas começaram a apontá-la como concorrente ou sucessora da linguagem Java na preferência entre os programadores (SALVADOR, 2008).

COMPARAÇÃO ENTRE AS LINGUAGENS JAVA E RUBY

As linguagens de programação Java e Ruby surgiram na mesma época (início dos anos 90), porém enquanto o Java começou a sua ascensão em 1995 com uma adoção muito rápida pelos desenvolvedores, Ruby permaneceu sem grande repercussão até o surgimento da biblioteca Rails em 2003. Assim como Java, Ruby segue o paradigma de programação orientado a objetos, porém existem muitas diferenças entre elas. A primeira é que Ruby é dinamicamente tipada e seu código fonte é executado por um interpretador (escrito em *C*, ou a alternativa JRuby explorada neste trabalho, escrito em Java). A vantagem da tipagem dinâmica é a eliminação da repetição de código e diminuição no número de exceções que podem ser causadas, enquanto o Java possui tipagem estática. Isso não significa que a tipagem de Ruby seja mais fraca, e sim, que ela força a passagem correta dos objetos, visto que em Ruby, tudo é objeto.

Ruby é executado em um interpretador, permitindo que o desenvolvedor

possa testar um código sem compilação. É possível até mesmo executar Ruby interativamente, cada linha como foi digitado. Além do rápido retorno, linguagens interpretadas têm uma vantagem em lidar com *bugs* (pequenos erros ocorridos ao longo do desenvolvimento). Com linguagens compiladas como Java, para analisar o código o programador deve verificar qual a versão exata da aplicação está com problemas e, em seguida, procurar o código com *bugs* no sistema de gerenciamento de código-fonte. Na vida real, muitas vezes, essa é uma tarefa difícil e cansativa, enquanto com linguagens interpretadas, por outro lado, o código fonte é imediatamente disponível para a análise.

A linguagem Ruby foi influenciada por inúmeras linguagens, dentre elas podemos destacar *Perl* (linguagem prática), *Smalltalk* (orientação a objetos – tudo é um objeto, multiprogramação – execução de tarefas concorrentemente e sintaxe semelhante), *Eiffel e Ada* (sintaxe) e *Java e Python* (tratamento de exceções).

Ruby suporta múltiplos paradigmas de programação. Além do seu estilo orientado a objeto puro, ela suporta, também, o modelo procedural, onde é possível escrever código fora de qualquer classe ou método, e o paradigma funcional.

Em Ruby, você pode empacotar qualquer função ou bloco de código, sem a necessidade de classes anônimas ou definições de métodos. Já em Java, é preciso circundar códigos com try{} e finally{}, por exemplo, para garantir que a lógica seja executada.

RUBY ON RAILS

Ruby on Rails (RoR) é um poderoso conjunto de ferramentas de software que permite a construção rápida de sofisticadas aplicações web. Feito em Ruby, RoR é um *framework* que proporciona um amplo conjunto de capacidades como, por exemplo, lidar com toda a comunicação com o banco de dados. Desta forma, o programador pode lidar simplesmente com os objetos, deixando para o framework a geração das consultas em *SQL*.

O *framework* foi criado para fazer o melhor uso do tempo de desenvolvimento, eliminando gargalos e permitindo a criação de soluções, em uma abordagem interativa, com mais agilidade. RoR foi projetado para, entre outros ideais, ser uma solução de desenvolvimento completa, onde suas camadas se comunicam da forma mais transparente possível, ser uniforme (escrito totalmente em Ruby), seguir a arquitetura MVC (*Model-View-Controller*) e ser ori-

entado a banco de dados, uma vez que é possível criar aplicações com base em estruturas pré-definidas.

As aplicações em Rails seguem dois padrões principais. O DRY – *Don´t Repeat Yourself* - diz que cada trecho de conhecimento de um sistema deve ser expresso em um único lugar; e o "*Convention Over Configuration*", que define padrões de nomenclatura e localização de arquivos que, quando seguidos, fazem com que a aplicação funcione sem a necessidade de configurações adicionais. O Rails é um *framework* altamente integrado, não só com *XP* (*Extreme Programming*), mas com todas as metodologias de desenvolvimento ágil. Isso porque o Rails foi desenvolvido seguindo os valores do manifesto ágil, mesmos valores que formam a base de todas as metodologias de desenvolvimento ágil.

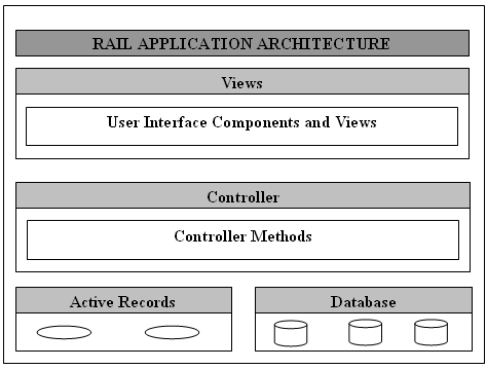

Figura 1: Modelo MVC do Ruby on Rails.

JRUBY

O interpretador JRuby é a implementação em Java do interpretador clássico desenvolvido para o Ruby, escrito na linguagem C. É um software liberado através das licenças CPL/GPL/LGPL (open source), e que permite a integração do código escrito em Ruby com qualquer aplicação em Java ou código escrito em Java em uma aplicação Ruby, como demonstrado neste projeto.

A versão original do interpretador foi desenvolvida por Jan Arne Petersen em 2001, e durante muito tempo foi uma parte direta do código do interpretador Ruby 1.6, desenvolvido na linguagem C. Com o lançamento da versão 1.8.6, iniciou-se um esforço para a atualização do interpretador com as características e semânticas dessa nova versão da linguagem.

Desde 2001, vários contribuidores têm participado do projeto, que possui quatro membros considerados como líderes do time de desenvolvimento: Charles Nutter, Thomas Enebo, Ola Bini e Nick Sieger. Em setembro de 2006, a Sun Microsystems contratou Enebo e Nutter para trabalhar no projeto JRuby em tempo integral, com o intuito de prestar dedicação total ao aperfeiçoamento do interpretador. Em junho de 2007, *a ThoughtWorks* con-

tratou Ola Bini para trabalhar com Ruby e JRuby.

DESENVOLVIMENTO DA APLICAÇÃO

A aplicação consiste em um Web site de comercio eletrônico, totalmente escrito em linguagem Ruby, implantada em um servidor Glassfish, com chamadas de códigos Java. O intuito disso é mostrar a possibilidade de comunicação entre as duas linguagens utilizadas.

As principais integrações desenvolvidas serviram não apenas para agregar maior funcionalidade e segurança ao sistema, como, principalmente, demonstrar que é possível utilizar qualquer biblioteca desenvolvida para a linguagem Java em código Ruby através do interpretador JRuby.

Dentre as bibliotecas integradas destacam-se:

- Stax (*Streaming API for XML*) – criação de um *parser* XML para a importação e exportação dos produtos do site de comércio eletrônico;
- JNA (*Java Native Access*) – chamada de DLLs nativas do sistema operacional para realização de algumas funcionalidades;
- Jboleto, JBarcodeBean e I-text – responsáveis pela geração de um boleto bancário ao fim do processo de compra;

A aplicação apresenta duas etapas principais. A primeira é o comercio eletrônico em si, onde são apresentados na tela todos os produtos da loja e, à medida que o usuário vai adicionando os produtos no carrinho é atualizado o valor total da compra e a quantidade de cada produto. Nesta parte, o usuário também tem a possibilidade de esvaziar o carrinho, remover um item ou finalizar a compra.

Ao finalizar a compra, o usuário deverá informar alguns dados, como nome e e-mail e confirmar o pedido. Com isso, caso esta confirmação seja efetuada com sucesso, o sistema deverá gerar um boleto e informar que a compra foi finalizada. Após isso o carrinho de compras é esvaziado, possibilitando a compra de novos produtos.

A outra etapa é de uso administrativo, onde para entrar é necessário informar o *login* de um usuário que possua acesso a essa funcionalidade. Neste ponto, é possível ver a lista de usuários cadastrados, assim como criar, alterar ou excluir um usuário. É possível, também, manter a lista de produtos (alterar, adicionar ou excluir) e

ver a quantidade de ordens de compras do dia.

CONCLUSÃO

Foi com o intuito de agilizar o processo de desenvolvimento de software que surgiu a linguagem Ruby e, posteriormente, seu mais conhecido *framework*, o Ruby on Rails. Juntos, eles formam uma poderosa estrutura para o desenvolvimento ágil de qualquer aplicação.

Com o amadurecimento do interpretador JRuby, tornou-se possível a integração entre o Ruby e a linguagem Java, com sua ampla comunidade de desenvolvedores, uma enorme fonte de referências e inúmeras bibliotecas. Estas últimas exercem fundamental importância em várias aplicações, reduzindo o tempo de desenvolvimento e o grau de dificuldade na implementação de quaisquer funcionalidades.

REFERENCIAS

GAMA, Gustavo. *Ruby on Rails.* Abril 2008. Disponível em: <http://blog.vettalabs.com/2008/04/07/ruby-on-rails/>.

AKITA, Fabio. *Ruby: Sucesso pela arrogância?* . Dez 2006. Disponível em: <http://www.rubyonbr.org/articles/2006/12/06/rails-sucesso-pela-arrogncia/>.

OFICINA DA NET. *Linguagem Ruby on Rails Ganha Forca na Web 2.0. Março 2007*. Disponível em: <http://www.oficinadanet.com.br/noticias_web/593/linguagem_ruby_on_rails_ganha_forca_na_web_2.0>.

8.3 LOCAWEB. *MVC e Ruby on Rails, uma visão simplificada.* Jun 2008. Disponível em: <http://tecblog.locaweb.com.br/2008/06/23/mvc-e-ruby-on-rails-uma-visao-simplificada/>.

WIKIPEDIA. *Ruby on Rails*. Ago 2008. Disponível em: <http://pt.wikipedia.org/wiki/Ruby_on_Rails

BETTEREXPLAINED. *Starting ruby on rails what I wish*. Jun 2007. Disponível em: <http://betterexplained.com/articles/starting-ruby-on-rails-what-i-wish-i-knew/>.

MENEZES, José. *Tudo o que webmasters devem saber sobre RoR*. Março 2007. Disponivel em: <http://www.plugmasters.com.br/sys/materias/628/1/Tudo-o-que-webmasters-devem-saber-sobre-RoR>.