HORUS FACULDADES
CURSO DE BACHAREL EM SISTEMAS DE INFORMAÇÃO

CLAIRTON RODRIGO HEINZEN

PROTÓTIPO DE UM SISTEMA DE GERAÇÃO DE RELATÓRIO ALIMENTADO POR XML DE NFE DESENVOLVIDO COM O FRAMEWORK RUBY ON RAILS

Pinhalzinho/SC
2011

CLAIRTON RODRIGO HEINZEN

PROTÓTIPO DE UM SISTEMA DE GERAÇÃO DE RELATÓRIO ALIMENTADO POR XML DE NFE DESENVOLVIDO COM O FRAMEWORK RUBY ON RAILS

Monografia de Trabalho de Graduação de Curso apresentado à Horus Faculdades, como parte dos requisitos à obtenção do grau de bacharel em Sistemas de Informação.
Orientador: Claudinei Hoss

Pinhalzinho/SC
2011

CLAIRTON RODRIGO HEINZEN

PROTÓTIPO DE UM SISTEMA DE GERAÇÃO DE RELATÓRIO ALIMENTADO POR XML DE NFE DESENVOLVIDO COM O FRAMEWORK RUBY ON RAILS

Esta monografia foi julgada adequada para a obtenção do título de Bacharel em Sistemas de Informação, e aprovada em sua forma final pelo curso de Sistemas de Informação da Horus Faculdades.

_____

Eduardo Urnau
Coordenador do curso de Sistemas de Informação

Apresenta à comissão examinadora integrada pelos seguintes professores:

_____

Claudinei Hoss
Orientador e membro banca – Horus Faculdades

_____

Jean Carlos Hennrichs
Presidente banca – Horus Faculdades

_____

Paulo Augusto Bastian
Membro banca – Horus Faculdades

_____

Revisão ortográfica realizada em 02/11/2011 por Salete Savegnago
Graduada em Letras - Unoesc/Chapecó-SC
Pós-graduada em Metodologia do Ensino – Faculdade de Ciência e Letras Plínio Augusto do Amaral/ Amparo-SP.

AGRADECIMENTOS

Agradecer primeiramente a Deus, pois por mais que o homem programe e analise, nunca fará obra tão perfeita quanto a sua criação. Aos professores, pelo apoio e conhecimentos oferecidos, especialmente ao Professor Orientador Claudinei Hoss pela dedicação.

Aos colegas do curso, pela amizade, pelas boas gargalhadas e pelos conhecimentos compartilhados.

A minha família, que me apoiou. Ao Ministério da Educação, pelo PROUNI (Programa Universidade Para Todos), que me possibilitou iniciar esse curso de graduação.

RESUMO

Em nível nacional, pode-se dizer que o Brasil passa por uma grande transformação de cunho tecnológico. O projeto SPED (Sistema Público de Escrituração Digital), marca a transição do armazenamento das informações sobre as movimentações fiscais para as empresas brasileiras, para a era digital, onde as informações em meio eletrônico passam a ter o mesmo valor dos documentos tradicionais gravados em papel. Diante da obrigação da informatização, necessária para a emissão de NFe (Nota Fiscal Eletrônica), muitas empresas que não têm condições de desenvolver, ou terceirizar o desenvolvimento de um software ERP (*Enterprise Resource Planning*) e acabam utilizando-se do Emissor Gratuito de NFe, disponibilizado pela Receita Federal. Contudo este emissor não gera nenhum dado útil para a administração da empresa. A partir dessa deficiência surge a necessidade de uma ferramenta que através do arquivo XML gerado pelo emissor gratuito de NFe, forneça uma forma de visualizar informações facilitando o dia a dia da empresa. Tal ferramenta fora implementada fazendo-se uso do padrão de projetos MVC (*Model-View-Controller*) bem como da linguagem *Ruby* e do *Framework* de desenvolvimento rápido para a WEB, *Ruby on Rails*. O protótipo da ferramenta oferece as empresas, que a utilizarem, a obtenção de dados totalizadores de impostos, fretes, descontos e adicionais, bem como também do valor individual e valor total de cada item, com seu respectivo percentual de participação no totalizador geral.

Palavras-chave: NFe; XML NFe; Ruby on Rails; Ruby; *Model-View-Controller*.

ABSTRACT

In national level, one can say that Brazil is going through a major transformation of a technological nature. The project SPED, marks the transition from storing information about the changes to tax Brazilian companies for the digital age, where informations in electronic form have the same validity of the papers stored in paper . Given the obligation of the computerization necessary to issue Invoice, many companies are unable to develop or outsource the development of an ERP software and end up using Issuer Free Invoices, provided by the Receita Federal. However, this issuer does not generate any useful data for management. From this deficiency arises the need for a tool that through the XML invoice file generated by the issuer free invoice, provide a way to view information facilitating the day to day business. This tool was implemented making use of the MVC design pattern (Model-View-Controller) and the Ruby language and the rapid development framework for WEB, Ruby on Rails. The prototype of the tool offers companies, that are using it, to obtain data totalizators taxes, freight, and additional discounts, and also of individual value and total value of each item, with their respective percentage of participation in the general totalizator.

Keywords: Invoice, XML Invoice, Ruby on Rails, Ruby, *Model-View-Controller*.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE SIGLAS

.NET	-*Framework* de Programação
AJAX	-*Assincronos JavaScript And XML*
API	-*Application Programming Interface*
B2B	-*Bussines to Bussines*
BeOS	-Nome de Sistema Operacional
C	-Linguagem de Programação
C++	-Linguagem de Programação
CoC	-*Convention over Configuration*
CRUD	-*Create, Retrieve, Update, Destroy*
CSS	-*Cascading Style Sheets*
DANFE	-Documento Auxiliar da NFe
DELETE	-Exclui um recurso existente no protocolo HTTP
DOS	-*Disk Operating System*
DRY	-*Don't Repeat Yourself*
ERP	-*Enterprise Resource Planning*
GET	-Criar um recurso no protocolo HTTP
GUI	-*Graphical User Interface*
HEAD	-Busca informações sobre um recurso já existente no protocolo HTTP
HTML	-*Hypertext Markup Language*
HTTP	-*Hypertext Transfer Protocol*
IPI	-Imposto sobre Produtos Industrializados
ICMS	-Imposto sobre Circulação de Mercadorias e Prestação de Serviços
Java	-Linguagem de Programação
JSON	-*JavaScript Object Notation*
MacOS	-*Macintosh Operating System*
MS-DOS	-*Microsoft-Disk Operating System*
MVC	-*Model-View-Controller*
NFe	-Nota Fiscal Eletrônica
ORM	-*Object Relational Mapper*
OS/2	-*Operating System/2*
PHP/FI	-*Personal Home Page/Form Interpreter*
POO	-Programação Orientação a Objetos

POST	-Busca um recurso no protocolo HTTP
PUT	-Altera um recurso existente no protocolo HTTP
REST	-*Representation State Transfer*
RoR	-*Ruby on Rails*
SEFAZ	-Secretaria da Fazenda
SmallTalk	-Linguagem de Programação
SPED	-Sistema Público de Escrituração Digital
SQL	-*Structure Query Language*
UML	-*Unified Modeling Language*
UNIX	-Sistema Operacional
URL	-*Uniform Resource Locator*
UTC	-*Universal Time Coordinated*
XML	-*Extensible Markup Language*
YAML	-*YAML Aint't Markup Language*
WHATWG	-*Web Hypertext Application Technology Working Group*
Windows 98	-Sistema Operacional Família Windows
Windows Me	-Sistema Operacional Família Windows
Windows NT	-Sistema Operacional Família Windows
Windows XP	-Sistema Operacional Família Windows
W3C	-*World Wide Web Consortium*

SUMÁRIO

1 INTRODUÇÃO

A integração e cooperação entre administrações tributárias têm sido temas muito debatidos em países federativos, especialmente naqueles que, como o Brasil, possuem forte grau de descentralização fiscal. Nesses países, a autonomia tributária tem gerado, tradicionalmente, multiplicidade de rotinas de trabalho, burocracia, baixa incidência de troca de informações e falta de compatibilidade entre os dados econômico fiscais dos contribuintes. Para os cidadãos, o Estado mostra-se multifacetado, ineficiente e moroso. Para o governo, o controle apresenta-se difícil porque falta a visão integrada das ações dos contribuintes. Para o País, o custo público e privado do cumprimento das obrigações tributárias torna-se alto, criando um claro empecilho ao investimento e geração de empregos.

Há alguns anos, iniciou-se uma grande operação de automação dos sistemas alfandegários com a implantação do Projeto Sistema Público de Escrituração Digital (SPED). Dentre os três subprojetos do SPED, encontra-se o projeto NFe (Nota Fiscal Eletrônica), ao qual este trabalho está relacionado.

O projeto NFe prevê a substituição da emissão de notas fiscais modelo 1 e 1A que são emitidas em papel, para um arquivo eletrônico, assinado digitalmente que tem a mesma validade que um arquivo físico.

Para poder emitir a NFe é necessário estar em dia com as obrigações fiscais, possuir um certificado digital e também um sistema que possa gerar o arquivo XML (*Extensible Markup Language*), que contém os dados referentes a operação e assiná-lo com o certificado. Para isso o governo disponibiliza um software gratuito, que também possui um cadastro básico de produtos e pessoas envolvidas (clientes e transportadoras).

Os arquivos XML são fontes ricas em dados, são estruturados e seguem um padrão estabelecido pela Receita Federal. A geração de relatórios através do dados contidos nesses arquivos, poderia facilitar a administração da empresa emissora, uma vez que o sistema gratuito oferecido pela receita não tem suporte a relatórios.

Um sistema que extraísse esses dados dos arquivos das notas fiscais poderia suprir essa necessidade, e dar mais competitividades para essas empresas, pois elas teriam informações disponíveis através de relatórios. Extraindo as informações dos arquivos XML e colocando-as em um Sistema Gerenciador de Banco de Dados (SGBD) a disponibilidade e rapidez para geração da informação no momento desejado é aumentada significativamente, sendo possível que uma empresa emissora de NFe (usuário do emissor gratuito), depois de importar no

sistema os arquivos XML, pudesse saber quanto vendeu em um determinado período, quanto imposto pagou, quanto vendeu de determinado produto, entre outros.

Disponibilizando este sistema na web a empresa poderia acessá-lo de qualquer lugar em que fosse ter acesso a internet, suprindo a necessidade do armazenamento obrigatório, determinado pela Receita Federal, de 5 anos mais 1, dos arquivos das NFe's, que ficariam junto ao sistema aonde ele estaria instalado. A escolha de um data center ou infraestrutura própria ficaria a critério do usuário.

Para a pesquisa foi utilizado o método de pesquisa aplicada, sendo coletados informações bibliográficas para o posterior desenvolvimento do protótipo. Os métodos para a abordagem e procedimento da pesquisa foram respectivamente qualitativa pois não necessitou técnicas estatísticas e exploratório, pois se baseou em materiais bibliográficos.

O *Framework Ruby on Rails*, que foi usado no projeto, tem se destacado para o desenvolvimento de sistemas web, pois oferece um desenvolvimento rápido com soluções prontas para problemas corriqueiros como Ajax[1], CRUD[2], ORM[3], versionamento e suporte a banco de dados, onde a troca de SGBD é feita apenas alterando apenas o arquivo de configuração de conexão. O acréscimo de funcionalidades pode, muitas vezes, ser feitos por códigos já prontos e testados, desenvolvidos por terceiros e disponíveis em um gerenciador de pacotes da Linguagem *Ruby*, que é usada para a construção do *Framework*.

Para a modelagem e análise foi utilizado alguns dos diagramas UML **(***Unified Modeling Language*), sendo especificado apenas os diagramas mais importantes e indispensáveis para o bom entendimento do sistema.

Através da realização desse trabalho, cumprindo o objetivo de gerar um protótipo de sistema de geração de relatório, alimentado pelo arquivos XML de NFe, pude compreender a sintaxe e a Orientação a Objetos da linguagem *Ruby* e também sobre o Padrão MVC (*Model View Controller*), o Padrão de Projetos *Active Record*, os comandos do *framework Ruby on Rails*, assim como o seus principais componentes.

1 Tecnologia capaz de efetuar pedidos ao servidor sem ter que recarregar a página (LEME, 2009, p. 134)
2 Métodos *Create, Read, Update e Delete*, de uma entidade usados para manter os registros em um SGBD (TATE e HIBBS, 2006, p. 19-21)
3 ORM (*Object-Relational Mapping*) é um conjunto de técnicas para a transformação entre os modelos orientado a objetos e relacional (CAELUM, 2011, p. 56)

1.1 ORGANIZAÇÃO DO TRABALHO

No capítulo 2, será feita uma breve explanação sobre o que é projeto NFe, seu objetivos e a estrutura do XML da NFe.

No capítulo 3, expõe-se sobre a linguagem *Ruby*, desde um pouco da história até sintaxe básica e sua Orientação a Objetos.

Mais adiante no capítulo 4, teremos uma pesquisa sobre o *framework Ruby On Rails*, seus padrões e sobre os outros *Frameworks* que o compõe, suas facilidades e abstrações.

No capítulo 5, uma compilação sobre importantes pontos do desenvolvimento só protótipo.

2 NFE

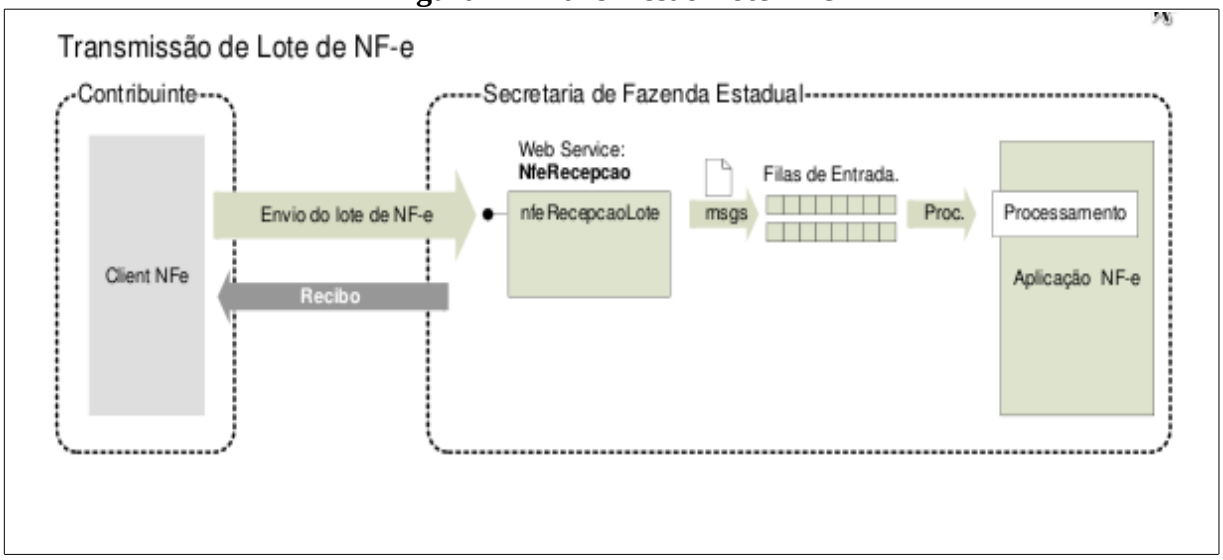
Segundo Duarte (2011, p. 74, 75)

> A NFe é um documento eletrônico que contém dados do contribuinte remetente, do destinatário e da operação a ser realizada. Este documento é assinado com certificado digital do remetente e enviado à Secretaria da Fazenda (SEFAZ) de sua unidade federativa, para validação e autorização.

Também segundo Duarte (2011), considera-se Nota Fiscal Eletrônica (NFe), o documento emitido e armazenado eletronicamente, de existência apenas digital, com o intuito de documentar operações e prestações, cuja validade jurídica é garantida pela assinatura digital do emitente e autorização de uso pela administração tributária da unidade federada do contribuinte, antes da ocorrência do fato gerador .

Ainda segundo Duarte (2011), o processo da NFe pode ser descrito da seguinte maneira, o remetente gera um arquivo no formato XML seguindo o padrão da NFe, assina-o digitalmente, e envia-o para a Secretaria da Fazenda (SEFAZ). Alguns minutos após, ele deve consultar através da Internet, a situação quanto à autorização de uso da NFe. Somente após a consulta e autorização do SEFAZ a mercadoria poderá circular.

Figura 1 - Transmissão Lote NFe



Fonte: ENCAT (2011, p. 28)

Também segundo Duarte (2011), tanto quem emite quanto quem recebe a NFe deve armazená-la pelo prazo legal. O arquivo eletrônico deve ser enviado ao destinatário de alguma forma. O destinatário, recebendo NFe deve verificar a validade e autenticidade do documento

fiscal através dos serviços disponíveis na Internet, essa verificação poderá ser feita através de *web services* quando o receptor possuir um sistema de informação, ou de maneira manual, indo até o site https://www.nfe.fazenda.gov.br e informando a chave de acesso da NFe. Independente da forma de verificação executada, a NFe só poderá ser escriturada do ponto de vista contábil e fiscal se os dados e a situação da nota estiverem corretos.

2.1 EMISSOR GRATUITO

Segundo Duarte (2011), A SEFAZ desenvolveu um programa para emissão de NF-e, que apresenta funcionalidades como: importação, digitação, validação, assinatura, transmissão, cancelamento e exportação de notas fiscais, além da impressão do Documento Auxiliar da NFe (DANFE) e manutenção de cadastros como: produtos, clientes, etc . Esse programa é indicado para pequenas e médias empresa cujo volume de notas seja pequeno.

2.2 OBJETIVOS DA NFE

Pode-se definir como objetivo da NFe (RECEITA FEDERAL DO BRASIL, 2011):

> a implantação de um modelo nacional de documento fiscal eletrônico que venha substituir a sistemática atual de emissão do documento fiscal em papel, com validade jurídica garantida pela assinatura digital do remetente, simplificando as obrigações acessórias dos contribuintes e permitindo, ao mesmo tempo, o acompanhamento em tempo real das operações comerciais pelo Fisco.

Segundo Duarte (2011), a NFe é um documento fiscal instituído pelo Ajuste SINIEF 07/2005 e modificado pelos Ajustes SINIEF 04/2006, SINIEF 05/2007 e SINIEF 08/2007 que a estabelece como substituta da Nota Fiscal modelo 1 ou 1-A emitida pelos contribuintes do Imposto sobre Produtos Industrializados (IPI) ou Imposto sobre Operações Relativas à Circulação de Mercadorias e sobre a Prestação de Serviços de Transporte Interestadual e Intermunicipal e de Comunicação (ICMS) .

Para Receita Federal Do Brasil (2011), o projeto da NFe pretende tornar ainda mais simples as informações tributárias, substituindo os modelos 1 e 1A emitidos em papel, oferecendo benefícios ao emitente, remetente, a sociedade e ao fisco. Entre os benefícios ao

emitente:

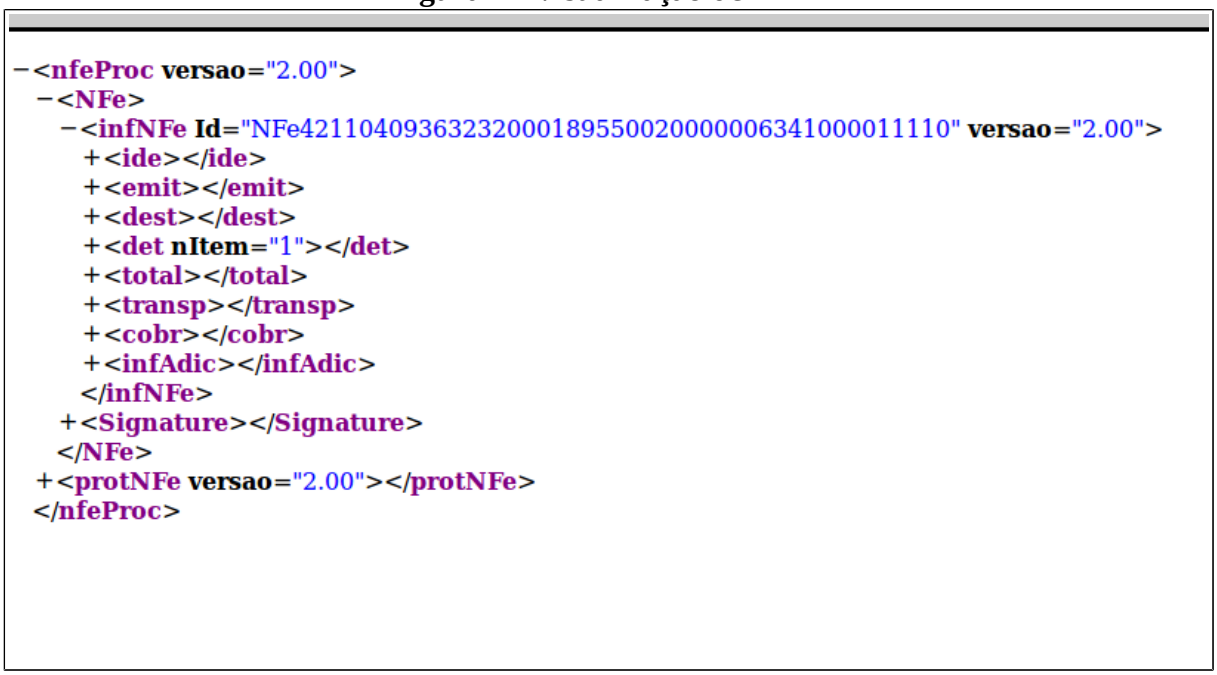
- Redução de custos de impressão;
- Redução de custos de aquisição de papel;
- Redução de custos de envio do documento fiscal;
- Redução de custos de armazenagem de documentos fiscais. Ainda segundo o órgão governamental, entre os benefícios ao remente podemos citar:
- Eliminação de digitação de notas fiscais na recepção de mercadorias;
- Redução de erros de escrituração devido a erros de digitação de notas fiscais.

Também segundo a Receita Federal Do Brasil (2011), existe um dos benefícios mútuos é o incentivo a uso de relacionamentos eletrônicos com clientes *Business to Business* (B2B).

2.3 XML

> A XML é um padrão aprovado pela *World Wide Web Consortium* (W3C) que deve se tornar um formato universal para a troca informações na Web […] da mesma forma que o *Hypertext Markup Language* (HTML) é uma linguagem de marcação nas quais seus documentos são compostos de uma mistura de dados e marcações [..] cada documento XML contém um ou mais elementos, sendo que os elementos são delimitados por uma *tag* de início e outra *tag* de fim (ex. <nome>e</nome>) [..] o texto entre as marcas é o conteúdo (AMARAL, 2003 p. 42, 43 e 44).

Figura 2 - Visualização de XML



Segundo Amaral (2003, p. 43), a diferença básica entre XML e HTML é que:

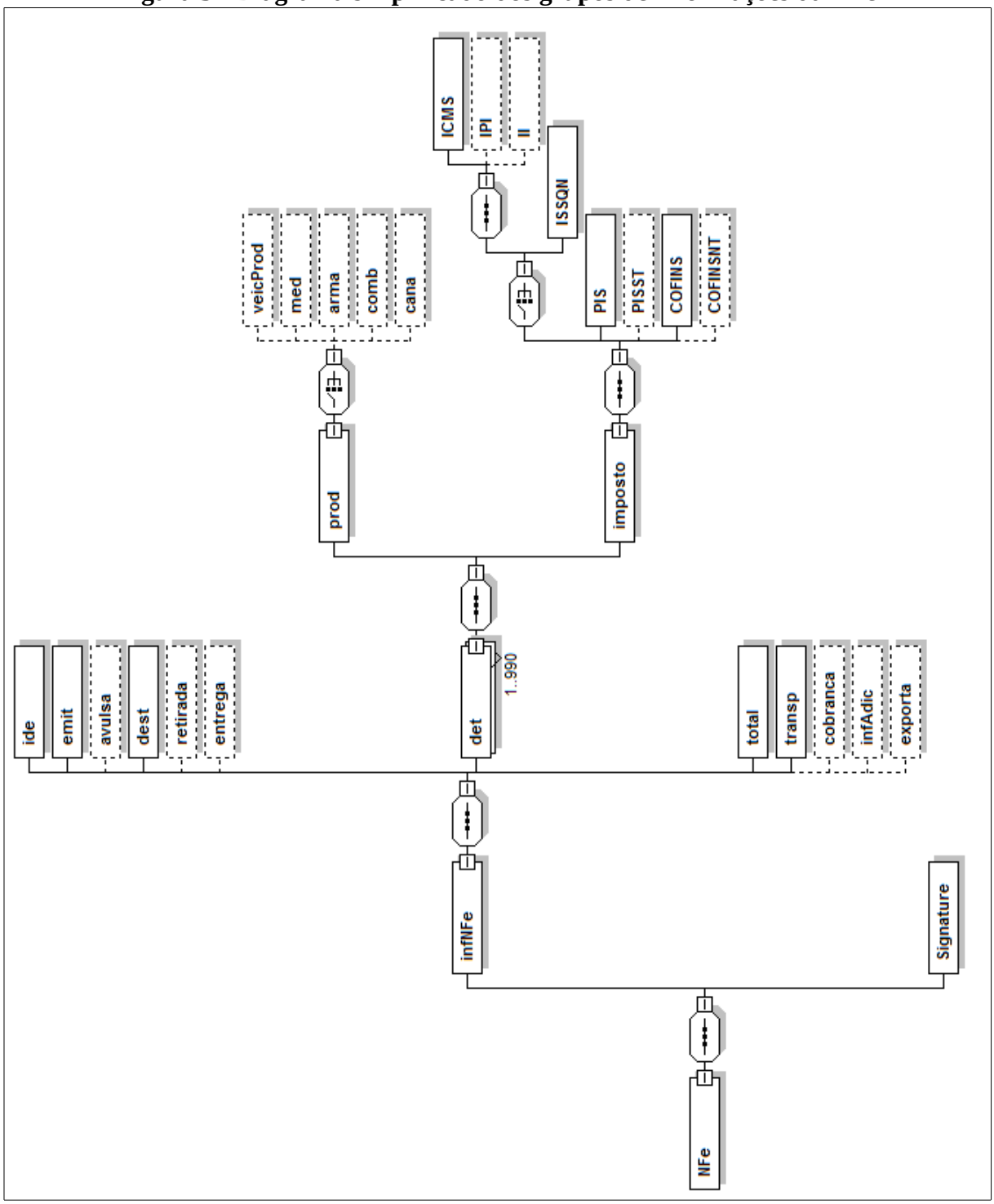
> A XML permite a troca eletrônica de dados na Web legível para o computador, enquanto o HTML permite a troca de documentos seja legível apenas para o homem […] com a XML há um ganho semântico pois os programas passam a entender a estrutura dos documentos disponíveis na Web, possibilitando assim pesquisas inteligentes das informações armazenadas.
> A necessidade de recuperação de dados armazenados na Web impulsiona constantemente o surgimento de novas tecnologias para tratar esses dados. XML pode ser vista como uma destas tecnologias. A XML pode ser uma opção para representar um modelo de dados contidos na Web(AMARAL, 2003, p. 50).

As principais tags do XML da NFe são ENCAT(2011):

- det: Grupo das informações de identificação da NFe;
- emit: Grupo de identificação do emitente da NFe ;
- avulsa: Identificação do Fisco Emitente da NFe ;
- dest: Grupo de identificação do Destinatário da NFe;
- retirada: Identificação do Local de Retirada ;
- entrega: Identificação do Local de Entrega ;
- det: Detalhamento de Produtos e Serviços da NFe ;
- veicProd: Detalhamento Específico de Veículos novos ;
- med: Detalhamento Específico de Medicamento e de matérias-primas farmacêuticas ;
- arma: Detalhamento Específico de Armamentos ;
- comb: Detalhamento Específico de Combustíveis ;
- imposto : Tributos incidentes no Produto ou Serviço ;
- ICMS : imposto ICMS;
- IPI: Imposto sobre Produtos Industrializados ;
- II: Imposto de Importação ;
- PIS : Imposto PIS;
- COFINS : Imposto Cofins.

Como demonstra o Diagrama da Estrutura do XML da NFe:

Figura 3 - Diagrama simplificado dos grupos de informações da NFe



Fonte: ENCAT (2011, p. 107)

3 A LINGUAGEM RUBY

Ruby é uma linguagem dinâmica, *open source* com foco na simplicidade e na produtividade, é uma linguagem com um equilíbrio cuidadoso (COMUNIDADE *RUBY*, 2011).

O seu criador, *Yukihiro Matsumoto* ou Matz, como é conhecido mundialmente (CAELUM, 2011, p. 4), uniu partes das suas linguagens favoritas (*Perl*, *Smalltalk*, *Eiffel*, *Ada*, e *Lisp*) para construir uma nova linguagem unindo a programação funcional[4] com a programação imperativa[5] (Comunidade *Ruby*, 2011).

Ainda segundo a Comunidade *Ruby* (2011), Matz diz que está a tentar tornar o *Ruby* natural, e não simples e também que o *Ruby* é simples na aparência, mas muito complexo no interior, tal como o corpo humano.

Segundo Oliveira Junior (2011, p. 1), "*Ruby* […] tem vários recursos para processar arquivos de texto e para fazer tarefas de gerenciamento de sistema (assim como o *Perl*). Ela é simples, direta ao ponto, extensível e portável."

Para Leme (2006), "O nome *Ruby* é o mesmo da pedra preciosa, já que foi inspirada em *Python* e *Perl* (pérola em inglês)." Ela é "altamente portável: é uma linguagem desenvolvida em sua maioria no Linux, mas funciona muito bem em muitos tipos de UNIX, DOS, Windows 98/Me/NT/XP, *MacOS*, *BeOS*, OS/2 etc" (OLIVEIRA JUNIOR, 2006, p. 3).

Segundo Caelum (2011, p. 5), o *Ruby* possui um gerenciador de pacotes e dependências bastante avançado, flexível e eficiente chamado *Ruby-Gem*, as *gems* podem ser definidas como bibliotecas reutilizáveis de código *Ruby*, podendo conter algum código nativo (em C, *Java*, .Net), parecidas aos *jars* do mundo *Java*, ou os *assemblies* do mundo .Net. Também podemos dizer que o *Ruby-Gem* é um sistema gerenciador de pacotes comparável a qualquer um do mundo *NIX, como os arquivo com extensão *.deb* do *apt-get*, o *MacPorts* ou *.rpm* do *yum*, entre outros, através dele podemos instalar e utilizar centenas de *gems* disponíveis.

Quantos aos interpretadores a linguagem *Ruby* tem várias implementações, entre elas temos o *Jruby*, que é escrita totalmente em *Java*, nela pode-se rodar códigos *Java* e *Ruby* juntos (CAELUM, 2011, p. 164).

---

4 Linguagens que não possuem o conceito de sequências de comandos e que são baseadas em funções matemáticas (GOULART, 2010)

5 Linguagens expressam sequências de comandos que realizam transformações sobre dados (GOULART, 2010)

3.1 HISTÓRIA DA LINGUAGEM

A linguagem *Ruby* foi anunciada à comunidade pela primeira vez em 1995 pelo seu criador Matz (CAELUM,2011, p. 4).

Ela foi construída sobre conhecimentos obtidos, como afirma a Comunidade *Ruby* (2011)

> Inicialmente, Matz estudou outras linguagens[...] *Perl, Smalltalk, Eiffel, Ada, Lisp* e *Python* [..] em busca de encontrar uma sintaxe ideal [..] ele queria uma linguagem interpretada que fosse mais poderosa que o *Perl* e mais orientada aos objetos do que o *Python*.

Ruby "é uma linguagem orientada a objetos, com tipagem forte e dinâmica. Curiosamente é uma das únicas linguagens nascidas fora do eixo EUA - Europa que atingiram enorme sucesso comercial." (CAELUM, 2011, p. 4).

Ela é realmente, totalmente orientada de objetos, não como o *Java*, onde há tipos primitivos que não são objetos, pois em *Ruby* até uma classe é um objeto (CAELUM, 2011, p. 4).

De ano em ano

> O *Ruby* arrastou consigo programadores devotos em todo o mundo. Em 2006, o *Ruby* atingiu aceitação massiva, com a formação de grupos de utilizadores em todas as principais cidades mundiais e com as conferências sobre *Ruby* com lotação esgotada (COMUNIDADE RUBY, 2011).

Figura 4 - Índice Tiobe de Uso das Linguagens

Position Mar 2011	Position Mar 2010	Delta in Position	Programming Language	Ratings Mar 2011	Delta Mar 2010	Status
1	1	=	Java	19.711%	+2.20%	A
2	2	=	C	15.262%	-2.02%	A
3	4	↑	C++	8.754%	-0.86%	A
4	6	↑↑	C#	7.210%	+2.95%	A
5	3	↓↓	PHP	6.566%	-3.34%	A
6	7	↑	Python	5.737%	+1.51%	A
7	5	↓↓	(Visual) Basic	4.710%	-1.86%	A
8	12	↑↑↑↑	Objective-C	3.518%	+1.55%	A
9	8	↓	Perl	1.969%	-1.85%	A
10	10	=	JavaScript	1.866%	-0.78%	A
11	11	=	Ruby	1.498%	-0.83%	A
12	-	=	Assembly*	1.345%	-	A
13	9	↓↓↓↓	Delphi	0.997%	-1.69%	A
14	13	↓	Go	0.958%	+0.04%	A-
15	21	↑↑↑↑↑↑	Lisp	0.934%	+0.38%	A
16	24	↑↑↑↑↑↑↑↑	Lua	0.812%	+0.30%	A-
17	32	↑↑↑↑↑↑↑↑↑↑	Ada	0.726%	+0.35%	A-
18	18	=	Pascal	0.706%	+0.10%	A
19	38	↑↑↑↑↑↑↑↑↑↑	NXT-G	0.640%	+0.34%	B
20	-	=	Scheme*	0.634%	-	B

Fonte: TIOBE(2011)

O índice TIOBE, que mede o crescimento das linguagens de programação, posiciona o *Ruby* em 11ª entre as linguagens de programação mais utilizadas no mundo segundo o resultado divulgado em 27 de março de 2011. “Muito deste crescimento é atribuído à popularidade do software escrito em *Ruby*, em particular a *Framework* de desenvolvimento *web Ruby on Rails*” (COMUNIDADE RUBY, 2011).

3.2 SINTAXE DA LINGUAGEM

Ruby “tem uma sintaxe elegante de leitura natural e fácil escrita.” (COMUNIDADE RUBY, 2011). Onde o final de um sentença é demarcada pela quebra de linha.

Segundo Augusto (2006),

> Algumas linguagens são pontuadas, utilizando frequentemente o ponto e vírgula ';'

> para terminar cada instrução. No *Ruby* segue-se a convenção utilizada nas *shells*, como a *sh* e a *csh*: múltiplas instruções numa mesma linha são separadas por ';' mas este sinal não é necessário no fim da linha para a terminar; o carácter de fim de linha ('\n') é um terminador de instrução. Se uma linha terminar com uma *backslash* ('\'), o '\n' que se lhe segue é ignorado, o que permite estender uma única instrução lógica por várias linhas de programa.

3.2.1 Variáveis

Segundo Leme (2006, p. 19):

> Em programação imperativa, uma variável é um objeto que contém um valor que pode ser avaliado e alterado. Estas variáveis são diferentes das variáveis da matemática (programação funcional), que possuem um único valor dependente do contexto onde foram declaradas.

Ainda segundo Leme (2006, p. 20), "uma variável no *Ruby* pode armazenar valores de qualquer tipo ao longo do tempo. Esta é a chamada tipagem dinâmica ou *duck typing*. *Ruby* não precisa que a variável seja declarada".

Os tipos de dados existentes no *Ruby*, segundo Oliveira Junior (2006), são: *String*(textos em geral), números(conversão automática dependendo do tamanho para *FixNum* ou *BigNum*), *Array*(*array* com o índice composto por números inteiros), *Hashes*(*array* com índice composto por *string*), Símbolos(espécie de *String* leve que representam nomes e algumas Strings dentro do interpretador, os símbolos são criados usando : na frente do seu nome. Ex.: :variavel_simbolo), Ranges(intervalos de valores Ex: x = 2 .. 5) e Booleanos.

Uma variável não é um objeto, mas uma referência a um objeto. "Isso significa que o mais próximo que chegamos dos objetos são as referências armazenadas pelas variáveis, sendo que as referências são valores binários que nos permitem acessar um objeto, atuando assim como um intermediário" (LEME, 2006, p. 20).

Como o *Ruby* é um linguagem de *script* e apesar da tipagem dinâmica e de não ser necessário a sua declaração, vale lembrar, que o *Ruby* também tem tipagem forte, isso é, receberemos um erro se tentarmos somar um valor *float* com uma *string* (OLIVEIRA JUNIOR, 2006).

3.2.2 Constantes

Na prática *Ruby* não tem constantes, mas tem algo parecido, são variáveis que iniciam com letra maiúscula, na qual o Ruby emite um aviso quando seu valor for alterado (OLIVEIRA JUNIOR, 2006, p. 17 ,18).

> O Ruby realmente não tem algo que seja realmente constante, mas a linguagem tem um padrão que diz que variáveis declaradas com a primeira letra maiúscula são constantes. A linguagem não impede de alterar o valor dessa constante se você realmente quiser: o Ruby vai apenas dizer que você realmente não deveria estar fazendo isso (URUBATAN, 2009 , p. 27).

3.2.3 Comentários

> Um comentário irá ocultar uma ou mais linha do interpretador do *Ruby;* sendo assim, essas linhas serão ignoradas durante a execução do programa. Isto permite ao programador inserir informações no código que poderá vir a facilitar futuras alterações e até mesmo gerar a documentação (LEME, 2009, p. 22).

Ainda para Leme (2009, p. 22), existem dois tipos de comentários, o comentário simples é declarado usando o carácter #, o que estiver depois dele até o final da linha será ignorado pelo compilador. Exemplo:

```
#o que estiver depois vai ser ignorado pelo interpretador
```

Para iniciar o comentário composto usa-se a tag =begin e terminado usando =end. Exemplo:

Quadro 1 - Comentários Longos

```
=begin
      o que estiver
      aqui em varias linhas
      vai ser ignorado
=end
```

3.2.4 Módulos

"Módulos são um jeito de absorver vários métodos e não ter que definir uma classe" (OLIVEIRA JUNIOR, 2006, p. 122).

Eles "são repositórios de coisas. Essa foi a melhor explicação que consegui, falando de forma bem genérica" (URUBATAN, 2009, p. 38).

"São úteis para criar *name spaces* para evitar conflito de nomes e um meio de simular algo do tipo de herança múltipla, disponível em outras linguagens" (OLIVEIRA JUNIOR, 2006, p. 122).

3.2.5 Blocos de Código

"Blocos de código consistem em um dos recursos mais versáteis e flexíveis do *Ruby*." (URUBATAN, 2009 p. 18).

"São muito utilizados em *Ruby*, são uma das peças chave da linguagem", são "pedaços de código delimitados por { e } […] para blocos de uma linha ou do e *end* para blocos maiores" (OLIVEIRA JUNIOR, 2006 p. 10).

> *Ruby* tem suporte a *closures* reais. Isso quer dizer que, se em um bloco de código forem utilizados variáveis visíveis no contexto da criação do bloco, qualquer alteração nessas variáveis vai ser refletida no contexto original, ou seja, os bloco de código têm um link com o contexto de origem, o que os torna mais úteis ainda e os diferencia muito de ponteiros para métodos de C/C++ (URUBATAN, 2009, p. 19).

Quadro 2 - Bloco de Código

```
bloco = Proc.new { puts "bloco de codigo" }
bloco.call
```

3.2.6 Estrutura de Controle

"*Ruby* é uma linguagem dinâmica mas também imperativa, e todas as linguagens imperativas têm estruturas de controle de fluxo e *loop*"(URUBATAN, 2009, p. 41).

"O interessante é que, em *Ruby*, o *if* não termina com *endif*, ou é limitado por chaves, ou está delimitado por tabulações […] termina em *end* […] a forma é *if...elsif..else...end*" (OLIVEIRA JUNIOR, 2006, p. 62).

Quadro 3 - Uso do if

```
i = 10
if i > 10
      puts "maior que 10"
elsif i == 10
      puts "igual a 10"
else
      puts "menor que 10"
end
```

Também pode ser usado o case, a estrutura segue abaixo:

Quadro 4 - Uso do Case

```
i=10
case i
      when 5
            puts "cinco"
      when 10
            puts "dez"
      else
            puts "??"
end
```

"A forma do case é case..*when...else...end*" (OLIVEIRA JUNIOR, 2006, p. 62).

"Case é um atalho mais organizado e semântico para uma sequência de elsif" (URUBATAN, 2009, p. 41).

3.2.7 Estruturas de Repetição

Segundo Oliveira Junior (2006, p. 64):

> Temos quatro maneiras de interagir com o conteúdo do *loop*, por dentro:
> - break sai do *loop* completamente
> - *next* vai para a próxima iteração
> - *return* sai do *loop* e do método onde o *loop* está contido
> - *redo* reinicia o *loop*

Urubatan (2009, p. 42) afirma, "os operadores […] [acima] podem ser utilizadas com qualquer um dos *loops*".

Ainda segundo Urubatan (2009, p. 42), “o *while* é um *loop* bastante flexível, pois permite o controle da condição do *loop*, podendo ser utilizado com qualquer condição booleana, derivada da comparação de qualquer tipo de objeto”.

Quadro 5 - Loop Usando While

```
a= 0
while a <= 10
      puts a
end
```

O autor também explana sobre o laço for, que é utilizado é utilizado para repetir um bloco de código, quando se conhece o números de vezes que ele deve ser executado.

Quadro 6 - Uso do For

```
for i in 1..5
      puts i
end
```

Segundo Oliveira Junior (2006, p. 64), existe também o comando *loop*, que é o mais flexível e terminado quando encontrado um break.

Quadro 7 - Uso Operador de loop Loop

```
loop
      puts “a”
      break if true
end
```

Também segundo Oliveira Junior (2006), apesar das interações de *arrays* poderem ser executados de várias maneiras, a mais indicada é usar o método *each* da classe *Array*.

3.2.8 Palavras Reservadas

O *Ruby,* com a intenção de ser simples, possui poucas palavras reservadas, são elas: *alias, and, BEGIN, begin, break, case, class, def, defined, do, eles, elsif, END, end, ensure, false, for, if, in, module, next, nil, not, or, redo, rescue, retry, return, self, super, then, true, undef, unless, until, when, while e yield* (CAELUM, 2011, p. 10).

3.3 THREADS

"*Ruby* tem um sistema de *threading* independente do sistema operacional […] para cada plataforma […] que roda *Ruby* […] tem *multi threading*, até no MS-DOS" (OLIVEIRA JUNIOR, 2006, p. 3).

Ainda segundo Oliveira Junior (2006, p. 129) para usar uma *thread* usa-se a sintaxe
t = Thread.new … end.

3.4 ORIENTAÇÃO A OBJETOS NO RUBY

Para Nassu e Setzer (1999, p. 25), a POO (Programação Orientação a Objetos) é um paradigma de programação popularizado na década de 80 no qual os programas são compostos por várias classes que contêm métodos e propriedades, os objetivos principais da POO são:

- Modularização: divisão dos programas em módulos independentes, facilitando o desenvolvimento, depuração e manutenção do software;
- Projeto de Módulos independentes da aplicação: os módulos não devem conter características particulares da aplicação para a qual está sendo desenvolvida, isso provê o reaproveitamento desses módulos em próximos projetos de sistema;
- Generalidade e Flexibilidade: projeto do módulo independente do tipo e estrutura de dados a serem armazenados;
- Reutilização: é uma consequência dos anteriores, prevê o máximo de reaproveitamento de código feito em projetos já realizados para novos.

Para Goulart (2010), a primeira linguagem que apresentou conceitos de orientação a objetos foi a simula67 no ano de 1966 e então em meados de 1970 foi desenvolvida a linguagem *Smalltalk* que foi a primeira totalmente orientada a objetos. Na década de 80, surge a linguagem C++ uma evolução da linguagem de programação C em direção à orientação a objetos em objetos.

A Orientação a Objetos, também conhecida como Programação Orientada a Objetos, é um paradigma que não causa nenhuma mudança perceptível ao usuário final, mas prove uma melhor organização do código. Nesse Modelo de programação o software é conjunto de

modelos de objetos que também são chamados de classes que trocam informações entre si e se comportam com base na implementação de sua estrutura interna (MILANI, 2010, 237-239).

Segundo Nassu e Setzer (1999, p. 25), os principais conceito de POO são herança, polimorfismo, encapsulamento, e mensagens.

Para a reutilização de código o *Ruby* também dispõe de módulos que são úteis para criar *name spaces* evitando conflito de nomes, sendo um meio de simular algo do tipo de herança múltipla, disponível em outras linguagens (OLIVEIRA JUNIOR, 2006, p. 122).

O conceito de Orientação a Objetos tem vários recursos que oferecem vantagens a esses modelos de programação que veremos a seguir.

3.4.1 Classes

Para Milani (2010, p. 239), "uma classe é um modelo criado para representar um conjunto de objetos com características semelhantes, como informações e comportamentos. Qualquer ação dentro de Orientação a Objetos parte de uma classe".

Leme (2009, p. 24) afirma com a mesma ideia:

> Uma classe é uma estrutura que abstrai um conjunto de objetos com características similares. Uma classe define um comportamento de seus objetos através de métodos e os estados possíveis destes objetos através de atributos. Em outros termos, uma classe descreve os serviços providos por seus objetos através de atributos. Em outros termos, uma classe os serviços providos por seus e quais informações ele pode armazenar. No *Ruby*, a definição de uma classe se inicia com o uso da palavra *class* e termina com a palavra *end*.

Exemplo:

Quadro 8 - Declaração de Classe

```
Class MinhaClasse
      #aqui dentro vai o conteúdo da classe
      #conforme a utilidade dela
end
```

"Classes representam uma das bases da orientação a objetos no *Ruby*" (URUBATAN, 2009, p. 34).

"Nos nomes das classes é utilizado o *camelCase*, da mesma maneira que em *Java*, com maiúsculas separando duas ou mais palavras no nome da classe" (OLIVEIRA JUNIOR, 2006, p. 72).

3.4.2 Atributos

Para Milani (2010, p. 240), “atributo faz menção as variáveis contidas na classe. São as informações que a mesma pode armazenar”. Os atributos podem ser de vários tipos, esse tipos definem o seu escopo:

- *private*: indica que a variável só pode ser acessada de dentro da classe onde está contida;
- *public*: a variável pode ser acessada a partir de códigos externos;
- *protected*: a variável só pode ser acessada de forma interna pela classe onde está contida uma subclasse;
- *static*: são variáveis cujos valores são compartilhados por todos os objeto de uma mesma classe, ao seu valor ser alterado por um objeto, ele valor será alterado para todos os outros objetos.

Para Urubatan (2009 p.18), “O *Ruby* não tem palavras-chave para definir o escopo das variáveis: isso é feito por meio de símbolos”.

Segundo Leme (2009, p. 20) isso ajuda “de forma que somente pelo nome [..] possa saber o escopo da variável […] quanto ao escopo, as variáveis podem ser variáveis locais, variáveis globais, variáveis de instância e variáveis de classe”.

> As variáveis de instância e globais possuem automaticamente o valor nil como valor padrão, podendo ser usadas […] sem que sejam atribuídos valores a elas […] as variáveis locais e as variáveis de classe precisam ser explicitamente inicializadas caso contrário ocorrerá um erro (LEME, 2009, p. 21).

3.4.2.1 Variáveis de Escopo Local

Segundo Leme (2009, p. 20)

> Variáveis locais são definidas dentro de um método, só existindo dentro dos limites daquele método especifico Os seus nomes devem começar com letras minúsculas ou com underline (_), embora o mais comum seja que eles comecem apenas com letras minúsculas. Exemplos de variáveis locais: razaosocial, _nascimento.

3.4.2.2 Variáveis de Escopo Global

Para Leme (2009), uma variável global, depois de criada, estará disponível para qualquer parte do programa, pela convenção do Ruby, as variáveis globais iniciam com o carácter $ e devem ser usadas com cuidado e somente quando forem realmente necessárias apesar da própria linguagem utilizar em alguma vezes. Um exemplo de variável global seria $carinho.

Segundo Oliveira Junior (2006, p.10), “declarar a variável com um $ na frente do nome já a torna pública”.

3.4.2.3 Variáveis de Instância

Para Leme (2009), as variáveis de instância iniciam com o carácter @, assim sendo @produto seria uma variável de instância.

> As variáveis de instância implementam os atributos de uma classe e coletivamente representam o estado de cada objeto. Cada objeto possui uma cópia de cada variável de instância definida na classe de forma independente dos outros objetos da mesma classe (LEME, 2009, p. 21).

As variáveis de instância “seriam uma analogia às variáveis privadas do *Java*” (OLIVEIRA JUNIOR, 2006, p. 70).

3.4.2.4 Variáveis de Classe

Segundo Oliveira Junior (2006) as variáveis de classe seriam semelhantes às variáveis estáticas do *Java*, onde estão no contexto da classe e não da instância.

> Uma variável de classe é compartilhada entre todos os objetos dessa classe, portanto podem ser usadas por cada um deles e também pelos métodos da classe [..] mesmo que existissem dezenas de objetos dessa classe […] só existiria uma cópia […] da variável (LEME, 2009, p. 21).

O seu nome da variável deve começar com @@, e ela deve ser inicializada antes de sua

utilização (OLIVEIRA JUNIOR, 2006).

3.4.3 Métodos

Segundo Milani (2010, p. 242), métodos de uma classe são as funções que a mesma tem a sua disposição para executar definindo o seu comportamento. Os métodos, assim como os atributos, podem ser de vários tipos. A lógica e os tipos que definem o escopo dos métodos são os mesmos dos atributos, com exceção do *static*, no qual, a forma de comportamento muda, pois um método estático é único para toda as referências da classe e sua execução não depende de dados concretos do objeto, pois não há relação com sua variáveis não estáticas.

Leme (2009, p. 23) diz:

> Em orientação a objeto, um método é um sub-rotina que é executada por um objeto ao receber uma mensagem […] eles determinam o comportamento dos objetos de uma classe e são análogos a funções ou procedimentos da programação estrutura […] chamadas de método podem alterar o estado do objeto.

Para o *Ruby* as definições de métodos iniciam com a palavra *def* e terminam com a palavra *end*, sendo que a declaração de argumentos pode ou não ser feita entre parênteses, assim como a chamada, sendo que a declaração pode ter expressões padrão, que são avaliadas da direita para a esquerda no momento em que o método é chamado, que são especificados com o símbolo = seguido do valor padrão, o valor retornado é o valor da última expressão dentro do bloco de código do método, se o comando *return* não for encontrado(LEME, 2009, p. 23,24).

Quadro 9 - Declaração de um Método

```
def meu_metodo(parmatro1, parametro2='valor_padrão')
      #código do método
end
```

A nomenclatura de métodos, assim como a nomenclatura das variáveis, segue algumas convenções:

- “nos nomes do métodos deve-se usar letras minúsculas separando as palavras com sublinhado” (OLIVEIRA JUNIOR, 2006, p. 70);
- “métodos que retornam *true* ou *false* tem seu nome terminado por '?'” (LEME, 2009, p. 24), eles são chamados métodos predicados, seriam os métodos

is_<condição> do *Java* (OLIVEIRA JUNIOR, 2009, p. 51);

- "métodos que modificam seu objeto […] são terminados por '!'" (LEME, 2009, p. 24):

> Métodos que alteram o valor e estado de um objeto geralmente tem um método com o mesmo nome, porém com um ! no final. Esses métodos são os chamados métodos destrutivos. A versão sem ! pega uma cópia de referência, faz a mudança, e retorna um novo objeto. Os métodos destrutivos alteram o próprio objeto (OLIVEIRA JUNIOR, 2006, p. 44);

- métodos que setam valores "a um atributo do mesmo nome do método do objeto são terminados por '=' e chamados automaticamente em uma expressão de atribuição, tendo como argumento o elemento à direita da atribuição" (LEME, 2009, p. 24);
- método de "leitura de uma variável de instância tem o mesmo nome da variável, sem o carácter @ no início" (URUBATAN, 2009, p. 45).

Urubatan (2009, p. 37) afirma que os métodos no *Ruby* não exitem:

> Não há métodos em *Ruby*! A diferença é bastante sutil, mas ajuda a entender melhor como as coisas funcionam. No *Ruby* não se chama um método de objeto: envia-se uma mensagem para um objeto, e esta pode ter parâmetros, mas sempre tem um retorno […] como se cada objeto tivesse uma caixa de correio interna, que só aceita mensagens para destinatários conhecidos, e todos os destinatários desconhecidos vão para o mesmo lugar: uma caixa com o nome de *method_missing*.
> Isso permite que métodos sejam adicionados apenas no momento em que se tornam necessários […] isso acontece bastante no *Framework Rails*.

Para Nassu e Setzer (1999, p. 31), dois métodos especiais são definidos na POO o método construtor e o destrutor. O método construtor é chamado no momento que o objeto é criado permitindo que seu estado interno seja inicializado com valores adequados, enquanto o destrutor é chamado antes de objeto deixar de existir. No *Ruby* "O método *initialize* é o construtor das classes em *Ruby*" (OLIVEIRA JUNIOR, 2009, p. 69) e o destrutor ainda segundo Oliveira Júnior (2011) é o *teardown.*

3.4.4 Objeto

Segundo Milani (2010, p. 242,243), "um objeto é representado pela ação de preencher os atributos de uma classe com dados reais, afim de obter uma referência de um item real (atingível) daquele tipo em questão. É a diferença entre algo e o resultado de sua criação".

Para Urubatan (2009, p. 34), “Tudo no *Ruby* é um objeto, e todo objetos é instância de uma classe. Por exemplo, 1 é uma instância da classes FixNum, e todos os métodos dessa classe podem ser chamados nessa instância”.

3.4.5 Herança

Segundo Milani (2010, p. 243,245), herança significa a possibilidade de uma classe herdar características de outra classe, tornando-se filha dessa classe da qual herda as características, essas características em questão são os atributos e métodos se assim o escopos deles permitir. O método da classe pai também podem ser sobrescritos e implementados na classe filha da maneira como for necessário.

Quanto a herança, Nassu e Setzer (1999, p. 25), definem como sendo um mecanismo onde uma classe é definida usando outra classe já existente, herdando suas variáveis de instância e seu métodos, além de possuir propriedades e métodos próprios. Para Urubatan (2009, p. 35), “o *Ruby* suporta herança utilizando o operador < na definição de uma classe” e Leme (2009, p. 24) confirma exemplificando “Herança é expressa pelo símbolo de menor ('<') ao lado da classe seguido do nome da superclasse. Exemplo class Automovel < Veiculo”.

Ainda, Urubatan (2009, p. 34), diz que “métodos de classe são herdados por classes descendentes da classe onde foram definidos e podem saber a qual objeto pertencem, pois nessa situação a palavra *self* vai apontar a classe e não uma de suas instâncias”. Ou, segundo Leme (2009, p. 25), o operador de escopo :: pode ser usado onde o *self* não resolveria o problema, :: “é usado para acessar membros internos de uma classe, ou para acessar explicitamente uma constante a quaisquer classes/módulos, prefixando a constante”. Isso também identifica o uso de mensagens da POO, que segundo Nassu e Setzer (1999, p. 29-30), é um ponto que diferencia a POO da programação tradicional, pois os métodos podem agir sobre seus próprios dados (propriedades declaradas fora do método e dentro da classe).

3.4.6 Encapsulamento

“Encapsulamento é o termo designado para representar que o funcionamento interno de

cada objeto é próprio, podendo ter comportamentos particulares (privados) que somente a própria classe pode invocar e utilizar, a partir de estímulos externos" (MILANI, 2010, p. 245).

No caso do conceito de POO de encapsulamento que segundo Nassu e Setzer (1999, p. 29), podemos definir como proteger a estrutura interna do objeto, no *Ruby* para Leme (2009, p. 25), é possível limitar a visibilidade de métodos e constantes usando os operadores de escopo: *public* (acesso público), *protected* (acesso apenas a objetos da classe e subclasses) e *private* (só podem ser chamados sem receptor explícito).

3.4.7 Interface

> Para permitir que classes distintas possam ter determinados comportamentos iguais, mas programados em um único lugar, foram criadas as interfaces […] o conceito de interface está atrelado a um contrato […] se determinada classe deseja se adequar a uma interface ela deve implementar todos os métodos dessa interface […] já que apenas as assinaturas dos métodos são definidas. A assinatura indica o que é feito, mas não como é feito (MILANI, 2010, p. 245,246).

Assim podemos ter duas classes que implementam a mesma interface, e duas formas de implementar o mesmo método, um para cada classe.

No *Ruby* para Leme (2009, p. 25), chamados sem argumentos alteram a visibilidade padrão enquanto chamados com argumentos alteram a visibilidade dos métodos e constantes especificados, porém devido à natureza dinâmica de *Ruby*, isto dá apenas uma certa formalidade à interface das Classes.

3.4.8 Abstração

A abstração na Programação Orientada a Objetos é a forma de criar classes com as características principais. O que diferencia a abstração das interfaces é que a abstração utiliza a definição de métodos na classes pai para serem implementados nas classes filhas, enquanto a interface define método que podem ser implementado em classes que tenham determinado comportamento igual sendo classes diferentes (MILANI, 2010, p. 249).

Ainda para Milani (2010, p. 249), "sempre que métodos abstratos forem usados, é necessário que a classe que o declara também seja abstrata, pois somente seus herdeiros

poderão implementar esse tipo de método”, também é útil lembrar que classes abstratas não podem ser instanciadas diretamente, é necessário que elas sejam herdadas e que seus métodos sejam implementados e então seus filhos podem ser instanciados.

3.4.9 Polimorfismo

Polimorfismo é a capacidade que o objeto tem de se comportar de formas diferentes, essa propriedade da Orientação a Objetos está ligada ao conceito de herança e interface, onde as classes filhas podem retroceder ás classes pais e se comportar como elas usando seus próprios métodos e atributos (MILANI, 2010, p. 250).

Nassu e Setzer (1999, p. 27), definem o polimorfismo como a forma que permite que um mesmo método seja definido em várias classes com implementações diferentes.

Quanto ao *Ruby*, ele suporta esse conceito e permite até sobrescrever métodos da própria linguagem, ele tem um recurso bastante interessante, que o torna uma linguagem muito flexível, esse recurso se chama *open class*, ou seja, todas as classes do *Ruby* são sempre abertas, o que possibilita que elas sejam alteradas a qualquer momento. Isso é bastante útil, mas também bastante perigoso pois podemos fazer com que o método + realize uma operação de subtração (URUBATAN, 2009, p. 17,18).

4 O FRAMEWORK RUBY ON RAILS

Ruby on Rails, RoR, ou simplesmente *Rails*, é um *Framework*. Para Leme (2009, p. 1), um *Framework* é um "conjunto de classes que colaboram para realizar uma responsabilidade para um domínio de um subsistema da aplicação".

O *Ruby on Rails* é escrito na linguagem *Ruby* e foi criado

> originalmente por David Heinemeier Hansson, um hacker dinamarquês, durante o desenvolvimento de um projeto de colaboração online […] tudo começou quando David não estava satisfeito com os *Frameworks* web disponíveis […], dessa forma ele decidiu escrever o seu próprio. Durante esse desenvolvimento ele deu ênfase em algumas coisas como convenção em vez de configuração e principalmente maior produtividade (LEME, 2009, p. 1).

O *Ruby on Rails* permite que você crie rapidamente aplicações organizadas, por isso é um *Framework* especial, segundo Tate e Hibbs (2006, p. 1):

> O problema de programação mais comum dos tipos de projeto de desenvolvimento de hoje envolve a criação de uma interface de usuário baseada em web para o gerenciamento de um banco de dados relacional. Para esse tipo de problema, o *Rails* é muito mais produtivo do que qualquer outro *Framework* de desenvolvimento para web que qualquer um de nós tenha utilizado. As forças não estão ligadas a uma única invenção fantástica; melhor que isso, o *Rails* está repleto de recursos para torná-lo mais produtivo e muitos deles criados usando uns aos outros.

O *Ruby on Rails* também usa o conceito de meta programação:

> As técnicas de meta programação utilizam programas para escrever outros programas. Outros *Frameworks* usam extensa geração de código, o que oferece aos usuários um único impulso de produtividade, mas pouca coisa além disso, e *scripts* de personalização permitem que o usuário adiciona código de personalização, em somente um pequeno número de pontos cuidadosamente selecionados. A meta programação substitui essas duas técnicas primitivas e elimina suas desvantagens. O *Ruby* é uma das melhores linguagens para meta programação e o *Rails* se também utiliza muito bem essa capacidade (TATE e HIBBS, 2006, p. 2).

Sua simples e poderosa abordagem de desenvolvimento adota o termo *Convention over Configuration* (CoC), em português convenção ao invés de configuração:

> A maioria dos *Frameworks* de desenvolvimento para .NET e *Java* força […] escrever páginas de código de configuração. Se […] seguir as convenções de nomenclatura sugeridas, o *Rails* não precisará de muita configuração. Na verdade, com frequência […] poderá cortar todo o seu código de configuração por um fator de cinco ou mais *Frameworks Java* similares simplesmente por seguir as configurações comuns (TATE e HIBBS, 2006, p. 1).

O *Ruby on Rails* também usa a filosofia do *Don't Repeat Yourself* (DRY), que segundo Caelum (2011, p. 39), significa que não se deve repetir código e sim modularizá-lo escrevendo

trechos de código que serão reutilizáveis e evitar o *Copy And Paste* de código.

Além disso o *Ruby on Rails* também utiliza Padrões de Projeto como o MVC, Gama et.al (2005, p.19), conceitua que os padrões de projeto de software, do inglês *Design Patterns*, descrevem soluções e consequências para problemas recorrentes no desenvolvimento de sistemas de software orientados a objetos. O conceito de padrão de projeto foi criado na década de 70 por Christopher Alexander. Segundo ele “cada padrão descreve um problema no nosso ambiente e o cerne da sua solução, de tal forma que você possa usar essa solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira.”.

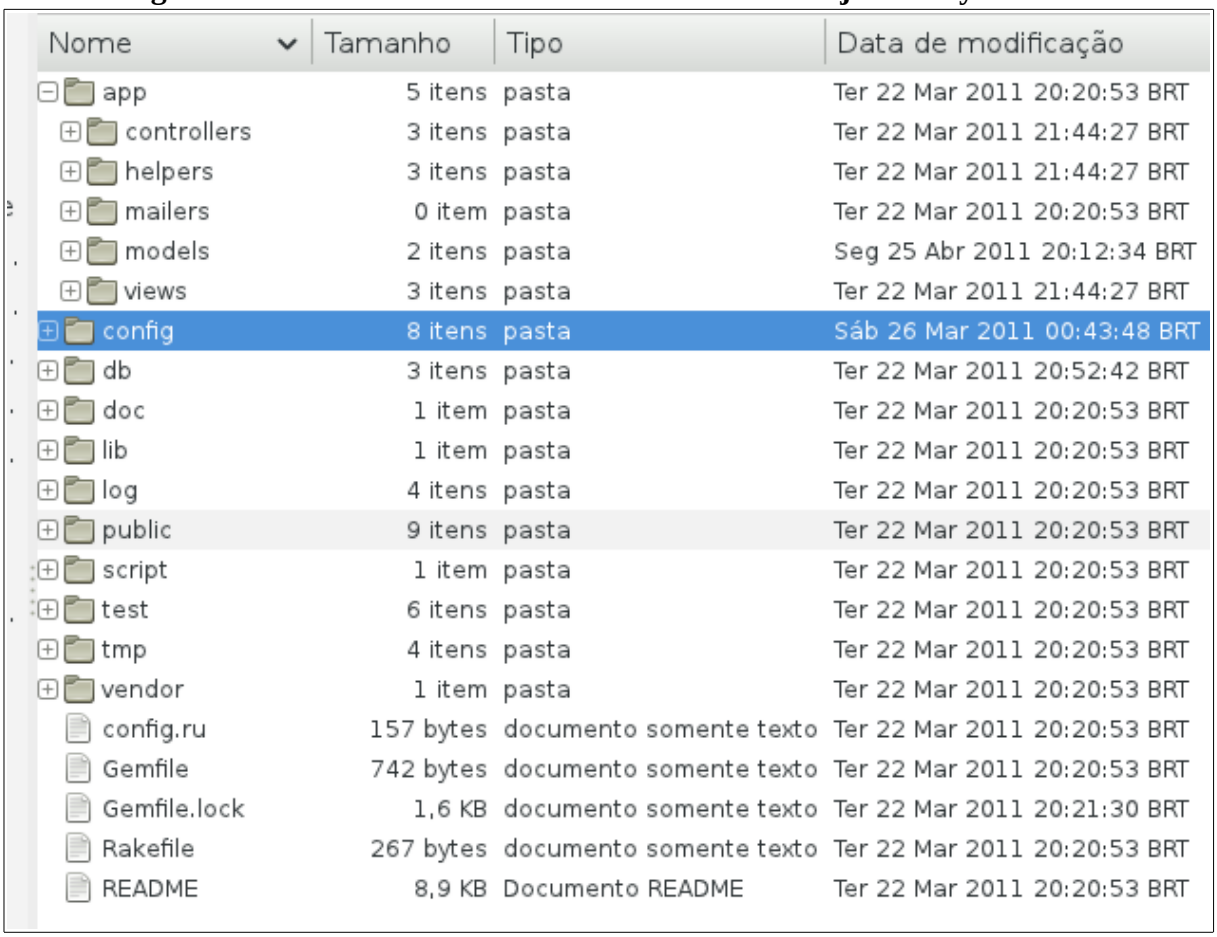
4.1 ESTRUTURA DE DIRETÓRIOS

> Quase sempre você cria código temporário nos seus primeiros estágios de desenvolvimento para a ajudá-lo a colocar a aplicação em execução com rapidez e ver como os componentes mais importantes funcionam em conjunto. O *Rails* cria automaticamente grande parte da estrutura necessária (TATE e HIBBS, 2006, p. 3).

Quando se cria um projeto, o *Ruby on Rails* já cria dentro dele vários diretórios, Leme (2009, p. 32, 33):

- *app*: um dois mais usados, neles ficam os arquivos referente aos *models, views, controlers e helpers;*
- *config*: nesse diretório ficam os arquivos que configuram a aplicação(banco de dados, roteamento de solicitações, estrutura e inicialização);
- *db*: contém os *scripts* para a geração e migração do banco de dados;
- *doc*: documentação;
- *lib*: bibliotecas do *Rails;*
- *log*: *logs* da aplicação;
- *script*: *scripts* para gerenciamento e execução;
- *public*: contém imagens, *Cascading Style Sheets* (CSS), *JavaScript;*
- *test*: teste criados para a aplicação;
- *tmp*: arquivos temporários do *Rails;*
- *vendor*: bibliotecas de terceiros.

Figura 5 - Estrutura de Diretórios Padrão de um Projeto Ruby on Rails



4.2 SERVIDOR HTTP

Segundo Elias (2011), “o HTTP (*Hypertext Transfer Protocol*) é um protocolo que atua na camada de aplicação de acordo com o modelo OSI e estabelece um conjunto de regras, padrão para troca de mensagens entre recursos na *WEB*” ele tem uma característica que é de não manter uma conexão, ele sempre conecta, envia uma mensagem, recebe uma resposta e desconecta.

Há vários programas que podem ser usados para implementar o protocolo HTTP e o *Ruby on Rails* pode ser executado em vários servidores *web* diferentes, apache, *webrick*, *lighttp*, *mongrel*, ou qualquer servidor web que suporte CGI, mas o CGI não é indicado para o *Ruby on Rails* pois é muito lento (TATE e HIBBS, 2006, p. 8-10).

Entre as várias opções podemos destacar o *mongrel*, segundo Caelum (2011, p. 166-167), ele foi escrito por Zed Shaw, em *Ruby*, é bastante performático e foi planejado

especificamente para servir aplicações *Rails*, apesar de hoje já servir outras aplicações web em *Ruby*, ele é a principal alternativa para *deployment* destas aplicações.

Ainda segundo Caelum, o problema com o *Mongrel* é que uma instância do *Rails* não pode servir mais de uma requisição ao mesmo tempo. Em outras palavras, o *Rails* não é *thread-safe*, ele possui um *lock* que não permite a execução de seu código apenas por uma *thread* de cada vez. Por causa disso, para cada requisição simultânea que precisamos tratar, é necessário um novo processo do servidor *Mongrel* rodando em uma porta diferente, sendo que é comum adicionar um balanceador de carga na frente de todos os *Mongrels* para receber as requisições, geralmente na porta 80 e despachar para as instâncias de *Mongrel* .

4.3 CONSOLE

O *Ruby on Rails* possui comando que servem para várias coisas, desde criar um projeto até a geração automática de um CRUD para um cadastro. Veremos os principais comandos agora.

Segundo Caelum (2011, p.42):

> Quando instalamos o *Rails* em nossa máquina, ganhamos o comando '*rails*'. Através desse comando podemos executar rotinas importantes nos nossos projetos. Uma dessas rotinas é a de criação de um projeto *rails*, onde precisamos usar o parâmetro *new* mais o nome do projeto […] *rails new* meu_projeto .

Após criar o projeto, temos alguns comandos para nos ajudar na sua administração e manutenção do projeto (URUBATAN, 2009, p. 60,61):

- *rails about*: lista a versão do *Ruby on Rails* com as dependências utilizadas;
- *rails console*: abre uma console com o ambiente da aplicação inicializado;
- *rails dbconsole*: abre um console com o banco para o qual a aplicação está configurada;
- *rails destroy*: deleta um código gerada, por exemplo *script*/*rails destroy model* nome_do_model, serve para deletar um *model;*
- *rails generate*: serve para criar um componente (*model, view, controller, test*, *migrate*), usando o *scaffold* como parâmetro ele gera todo CRUD para o cadastro;
- *rails plugin*: instalar ou remover *plug-ins;*
- *rails runner*: executar arquivo de código *Ruby;*

- *rails server*: iniciar o servido HTTP;
- *rake* é uma ferramenta de *build* semelhante ao *make* e ao *ant*, em escopo e propósito (CAELUM, 2011, p. 56).

4.3.1 Scaffold

No *Ruby on Rails*:

> temos a opção de utilizar uma tarefa chamada *scaffold*, cuja tradução se equivale aos andaimes de um prédio em construção: criar a base para a construção de sua aplicação. Essa tarefa gera toda a parte de MVC (*Model View Controller*) relativa ao modelo dado, conectando tudo o que é necessário para, por exemplo, lógicas de listagem, inserção, edição e busca, junto com suas respectivas visualizações […] *helpers*, testes e o *layout* do site […] existe também uma opção para passar como parâmetro o modelo a ser criado. Com isso, o *Rails* se encarrega de gerar o modelo e o *script* de criação da tabela no banco de dados para você (CAELUM, 2011, p. 45,46).

Segundo Leme (2009, p. 48,85), *scaffold* cria um estrutura básica que facilita a prototipação da aplicação criando a estrutura básica (CRUD) para a manipulação de dados. O comando para o *scaffold* é *rails generate scaffold* nome_do_model nome_do_campo:tipo nome_do_outro_campo:tipo. Os tipo usados podem ser *string*, *text*, *integer*, *float*, *decimal*, *datetime*, *timestamp*, *time*, *date*, *binary*, e *boolean*.

4.4 TESTES UNITÁRIOS

No *Ruby on Rails* a estrutura é criada de maneira que os teste são internos e aparecem desde o primeiro momento em que é criado o projeto, isso supre a necessidade que há de debuggar o código como é feito quando compila um programa escrito em uma linguagem como *Java* ou C++. Para automatizar os testes ao máximo possível o *Ruby on Rails* utiliza o *Framework Test::Unit*. Existem três tipos de teste, os de unidade que servem para testar os modelos, os funcionais que servem para testar os controladores e os de integração que servem para testar situações de alto nível simulando, por exemplo, interações entre controladores (TATE e HIBBS, 2006 p. 115-118).

4.5 MVC

> Quando inicia o contato com o *Rails* é inevitável que também conheça o padrão de arquitetura e de design de software MVC-*Model-View-Controller*, ou Modelo-Visão-Controlador, em uma tradução livre […] ele foi descrito pela primeira vez em 1979 pelo cientista norueguês Trygve Mikkjel Heyerdhal Reenskaug quando ele estava envolvido com o linguagem *SmallTalk*, na empresa Xerox Park (LEME, 2009, p. 29).

Segundo Tate e Hibbs (2006, p. 4), “na metade dos anos 70, a estratégia modelo-visualização-controlador (MVC) evolui na comunidade *Smalltalk* para reduzir a ligação entre a lógica de negócio e a lógica da apresentação”.

Para Leme (2009, p. 29), Reenskaug percebeu que o aumento da complexidade das aplicações desenvolvidas era fundamental que alterações feitas no *layout* não deveriam afetar a manipulação de dados e estes poderiam ser organizados sem alterar o *layout.* O MVC procura resolver esse problema com a separação da lógica de negócio, lógica de apresentação e interação com o usuário com o componente *Controller* entre os dois.

Os *Frameworks web* como o *Rails*, escrito em *Ruby* e o *Struts*,projeto escrito em *Java*, utilizam um padrão derivado do MVC, que possui uma pequena diferença e por isso é denominado Model2, o Model2 que usa os mesmos padrões do MVC, mas os ajusta as aplicações web sem estado, ou seja, no Model2 usa um *browser* que chama um controlador por meio de padrões web, o controlador chama o modelo para obter dados e validar as entradas, disponibilizando para a *View* correspondente, então chama para o gerador de visualizações, baseando-se nos resultados da validação ou nos dados recuperados, a camada de visualização então gera uma página web que exibe esses dados (TATE e HIBBS, 2009, p. 4,5).

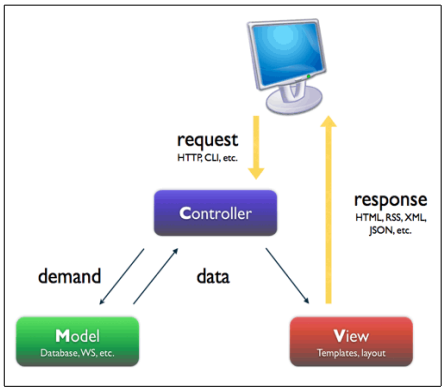
Para Leme (2009 p. 30), esse padrão chamado Model2 foi criado pela Sun, mas com controvérsias.

Para Lisboa (2009, p. 216), o MVC é útil para criar aplicações *Graphical User Interface* (GUI), sendo desenvolvido originalmente para o lado do cliente, mas tem evoluído opções para se ajustar a WEB. É importante entender que através da URL que é mapeado o padrão MVC, por exemplo http://www.exemplo.com.br/controlador/ação/parametro1/parametro2 temos ideia do controlador que será requisitado juntamente com a ação .

Os componentes VC, ou seja *view* e *controller* descendem do *Framework ActionPack* (TATE e HIBBS, 2006), enquanto o M, ou seja, o *model* herda sua funções do *Framework ActiveRe*cord (CAELUM, 2011, p. 55). Quando um evento é disparado, o *Framework Ruby*

*on Rai*ls, através do método HTTP utilizado sabe o que deve fazer. Na URL há o nome do controlador que está ligado ao *model*, ele automaticamente executa o método no *model*, sendo isso executado com sucesso, por padrão, o controlador chama a *view* com o nome do método executado no controlador (TATE e HIBBS, 2006).

Figura 6 – Comunicação Padrão MVC



Fonte: Valente (2001)

4.5.1 *Model*

Para Lisboa (2009, p. 217,218), o modelo possui a estrutura de dados e classe associada, ele provê dados para o controlador e notifica a visão sobre as mudanças. Ele é usado para implementar a lógica do domínio, como por exemplo, executar cálculos, o que frequentemente envolve o acesso ao banco de dados.

Segundo Urubatan (2009, p. 67), para o *Framework Ruby on Rail*s "o *model* é uma entidade *Ruby* que representa uma tabela do banco de dados. Todos os *models* descendem da classe *ActiveRecord::Base*". O *Framework* que gerencia o *model* é o *ActiveRecord*.

4.5.1.1 *ActiveRecord*

> O *Rails* apresenta o *Framework Active Reco*rd, que salva os objetos no banco de dados baseado em um padrão de projeto catalogado por Martin Fowler, a versão do *Rails* do *Active Record* descobre as colunas em um esquema de banco de dados, e automaticamente as anexas em seus objetos de domínio, usando a meta programação. Essa abordagem de agrupamento de banco de dados é simples, elegante e poderosa (TATE e HIBBS, 2006, p. 1).

Tate e Hibbs (2006, p. 19-21), também mencionam que o Padrão *Active Record* foi publicado no livro chamado *Patterns Enterprise Architecture* no qual mencionava que manipulação dos dados é feito por meio de objetos de registro, onde cada objeto de registro representa uma linha da tabela do banco de dados e possui métodos *Create, Read, Update e Delete* (CRUD), ou seja, as operações para manter os registros. O *Framework Active Recor*d do *Ruby on Rails*, tem algumas vantagens:

- os atributos são adicionados às classes automaticamente com base nas colunas do banco de dados;
- possui gerenciamento e validações de relacionamento por meio de uma linguagem personalizada;
- as convenções de nomenclatura do *Ruby on Rail*s permitem que o banco de dados descubra campos específicos;

Para Leme (2009, p. 45), "o controle de iteração entre a aplicação e o banco de dados é feito por um dos *Frameworks* do *Rails* chamado *ActiveRecord* [..] o *ActiveRecord* é um dos melhores exemplos de aplicação da filosofia difundida pelo *Rails*: Convenção em vez de Configuração".

Ainda segundo Leme (2009), o *ActiveRecord* efetua o mapeamento entre as tabelas do banco de dados e as classes da aplicação, o que faz com que o desenvolvedor não precise se preocupar com comando SQL.

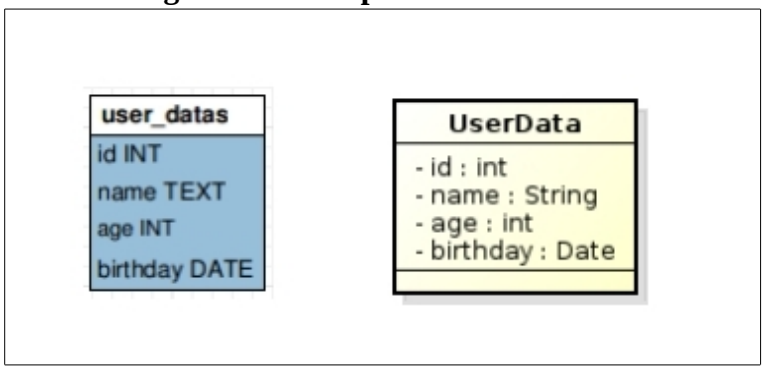
As classes do *ActiveRecord* devem seguir algumas convenções (TATE e HIBBS, 2006, p. 27):

- O nome da classe deve ser no singular, enquanto o nome da tabelas, no plural, é interessante também usar os nomes em idioma inglês, pois o próprio *Frameworks* se encarregará de buscar a tabela correta quando a classe for usada;
- Em cada tabela deve haver uma coluna com um valor único, por padrão, mesmo que não especificado, o *Ruby on Rai*ls cria uma coluna chamada id, cujo o valor é atribuído por uma sequência automática do banco de dados, essa coluna que é criada automaticamente também é determinada com chave primária, que pode ser alterada, mas o mesmo não é indicado;
- As chaves estrangeiras devem ser nomeadas como <classe>_id, onde classe é o nome da outra classe com qual a classe atual estabelece relacionamento;
- Nomes das tabelas compostos por várias palavras, assim como nome de métodos, atributos, colunas, símbolos, devem ser separadas por sublinhado,

enquanto o nome das Classes devem ter a primeira letra maiúscula.

Segue abaixo um exemplo de uma tabela com sua respectiva classe com o padrão *Active Record*:

Figura 7 – Exemplo do *ActiveRecord*



Fonte: URUBATAN (2009).

Nesse caso o classe UserData ficaria assim:

Quadro 10 - Classe do ActiveRecord

```
class UserData < ActiveRecord::Base
      validates_presence_of :name
end
```

Através da herança a classe UserData passa a ter vários métodos para pesquisa, além do atributos baseados nas colunas da tabela, que também são gerados em tempo de execução. Um dele é o find_by_<nome_da_coluna>, que em nosso exemplo seria o find_by_name, esse método é criado para cada coluna, o outro é gerado para uma pesquisa por mais de um atributo no formato find_by_<nome_da_coluna>_and_<nome_da_outra_coluna>, exemplificando find_by_name_and_age (URUBATAN, 2009, p. 199,200).

Ainda Segundo Urubatan (2009, p. 200,201), o principal método para a busca de dados é o método *find*, que por ser flexível consegue resolver todas as situações que os buscadores dinâmicos não conseguem resolver, a assinatura básica do método é *find* (o que buscar, opções), o que buscar pode ser o ID de um objeto, como também podem ser utilizados os símbolos :*first*, :*last* ou :*all*, que significam respectivamente primeiro, último ou todos. Esse primeiro parâmetro pode ser substituído pelos métodos *first*, *last*, *all*, que recebem apenas o *hash* de opções, as opções que podem ser usadas estão especificadas abaixo:

- :*conditions* pode receber uma *string*, um *hash* ou um *array;*
- :*order* um fragmento SQL com “created_at DESC, name”;
- :*limit* o número máximo de registros a serem retornados;

- :*offset* o número de registro a serem ignorados;
- :*joins* fragmento de SQL com *joins* “LEFT JOIN comments on comments.post_id=id”;
- :*include* nome de associações que devem ser carregadas junto com a consulta;
- :*select* por padrão ele usa “*” que traz todas as colunas, mas aqui pode ser especificadas as colunas que deseja trazer;
- :*from* por padrão é utilizado o nome da tabela do modelo, mas pode ser alterado para uma tabela ou *view;*
- :*lock* um fragmento SQL como “FOR UPDATE”.

4.5.1.1.1 Validações

O *ActiveRecord* também pode ser usado para fazer validações, como o exemplo no quadro 10, que como o método validates_presence_of, torna o campo referido *name*, obrigatório, também há como validar o tamanho de uma *string*(validates_length_of : width => 6..10) , ou um certo formato. Pode-se usar também expressões regulares e escrever seus próprios métodos de validação (TATE e HIBBS, 2006, p. 36).

4.5.1.1.2 Relacionamentos

Para Tate e Hibbs (2009, p. 40):

> Banco de dados relacionais são fundamentalmente baseados em diferentes tipos de relacionamento entre tabelas. Uma coleção de colunas de tabelas chamadas chaves oferece a estrutura para todos os relacionamentos. Uma chave primária e uma coleção de coluna de uma tabela que identifica exclusivamente uma linha da mesma tabela. Um gerente de banco de dados pode associar duas tabelas fazendo a correspondência entre chaves primária de uma tabela e chaves estrangeiras de outra. O *ActiveRecord* também usa chaves primárias e estrangeiras para gerenciar relacionamentos. Ao contrários dos bancos de dados relacionais, o *ActiveRecord* limita seus identificadores a uma única coluna de banco de dados.

O *ActiveRecord* dispões de vários métodos para fazer os relacionamentos, cada um para um caso especifico:

- *belongs_to* usado na relação de muitos para um;

- *has_many* usado na relação de um para muitos, ou seja o outro lado do belongs_to, através do parâmetro *dependent*, setado como *true*, como fazer uma exclusão em cascata;
- *has_one*, o relacionamento mais simples, um para um.;
- *has_and_belongs_to_many*, método para o relacionamento de muitos para muitos, o relacionamento mais complexo, onde é criado um terceira tabela que serve de intermédio ;

4.5.1.1.3 Transações

Segundo Tate e Hibbs (2006, p. 37), para código que devem ser executados em uma única unidade, pode-se usar o controle de transações do *ActiveRecord.* Um exemplo poderia ser uma transferência bancária, onde a operação só deve ser concluída inteiramente, e em caso de erro, as informações retornam ao estado anterior ao inicio da transação. No *Ruby on Rails* isso poderia ser feito circuncidando o bloco de código desejado com NomeDoModelo.transaction e o end.

4.5.2 *View*

Para Lisboa (2009, p. 217 e 218), a visão é um grupo de classes relacionadas a elementos GUI e gera o modelo para o usuário final provendo interfaces para aceitar a entrada para o controlador. É basicamente sinônimo de HTML, para muitas aplicações é uma camada de *template*, onde não deve conter lógica de aplicação. Em aplicações mais modernas com AJAX, a visão pode gerar ou conter XML, *JavaScript Object Notation* (JSON) ou qualquer formato requerido.

Por padrão o *Ruby on Rails* utiliza a extensão *.erb* para as *views*, ele são arquivo que misturam texto puro com código *Ruby* que estão contidos entre <% e %>, para imprimir textos há um demarcador de código mais simples é <%= e %>. Para cada métodos do *controller* geralmente temos um *view*, que é renderizada por esse método do controlador. Para não repetir códigos, as *views* podem usar o recurso de *partials*, arquivo que usam a mesma

extensão .erb que tem os nomes iniciados por _(*underline*), e são renderizados da seguinte maneira: render :*partial* => <nome_da_partial>. Para enxugar ainda mais o código pode-se usar os *helpers* (URUBATAN, 2009, p. 74-80).

4.5.2.1 *Helpers*

Enquanto a lógica está no *model* e a visualização nas *views*, mas muitas vezes duas *views* possuem uma parte com o mesmo código, nesse caso há possibilidade de reaproveitar uma porção de código separado, que pode ser colocado em um *helper*, ele normalmente armazena o código mais maçante, deixa o código da *view* mais limpo. Quando criamos uma *controller*, um *helper* é gerado automaticamente no diretório app/helper/ em um arquivo nomedocontroller_helper.rb, esse *controller* estará disponível apenas para as *views* correspondentes, mas existe um *helper* global, é o arquivo *application_helper*, que está disponível para qualquer *view* da aplicação (LEME, 2009, p. 79).

4.5.2.2 HTML

Para Alvarez (2004), o HTML é uma linguagem de marcação de texto que permite aglutinar textos, imagens e áudios e também a introdução de referências a outras páginas por meio dos links hipertextos. Quando foi criado, não se pensou na possibilidade da web chegar a ser utilizada da forma como é utilizada hoje, por isso foram se incorporando modificações, as quais são os padrões do HTML. Numerosos padrões já se apresentaram. O HTML 4.01 é o mais utilizado atualmente e foi lançado em 2004. Também foram embutidas mais outras várias tecnologias como o CSS e a linguagem *JavaScript*.

Para Eis, desde 2004, logo após um *workshop* sobre o XHTML feito pelo W3C, organização responsável pela manutenção e criação dos padrões para o HTML, empresas como Apple, Mozilla e Opera notaram que a direção do W3C estava se distanciando do que os desenvolvedores atuais necessitavam, em resposta elas se aliaram à eles e fundaram a WHATWG (*Web Hypertext Application Technology Working Group*) para criar a próxima versão do HTML denominada HTML5.

4.5.3 *Controller*

Para Lisboa (2009, p. 217 e 218), os controladores são Classes conectando modelos e visões, aceitando a entrada do usuário, executando a lógica da aplicação, sendo a única que recebe os requerimentos.

Segundo Urubatan (2009, p. 69):

> Os controladores em um aplicação *Rails* consistem em uma forma de publicar seus modelos na internet. Não se deve ter muitos código no controladores, já que servem apenas para passar os parâmetros recebidos para o *model*. De preferência devem chamar um ou dois métodos do model, encaminhar o resultado desses métodos para uma *view*, podendo mandar para uma *view* diferente do resultado recebido do *model* dependendo do tipo de *request* recebido.

Também segundo Urubatan (2009, p. 69,212-219), a partir da versão 2.0 o *Ruby on Rails* favorece a abordagem *Representation State Transfer* (REST) para o desenvolvimento das aplicações, através da utilização deste e do *ActiveRecord*, o desenvolvimento de aplicações se tornar mais rápido e fácil. O padrão REST visa utilizar os outros métodos do protocolo HTTP além do GET e POST usados respectivamente para buscar um recurso e criar um recurso, são eles, PUT que serve para alterar um recurso existente, DELETE para excluir um recurso existente e HEAD que tem utilidade de buscar informações sobre um recurso já existente. Um recurso neste cenário seria qualquer coisa que possa interessar a um ser humano ou computador sendo acessível por URL.

Ainda segundo segundo Urubatan (2009), ao criar um controlador no *Ruby on Rail*s, o *ActionPack* (*Framework* que no *Ruby on Rails* é responsável pelo VC do MVC), define sete métodos padrão:

- *index*: responderá requisições GET para o endereço de controlador, a implementação busca uma lista de registros do modelo e passa esse resultado para a *view* correspondente;
- *show*: que reponderá à requisições GET para o endereço controlador/id do registro. Na implementação padrão busca pelo registro do modelo com o ID recebido como parâmetro e encaminha para a *view* correspondente;
- *new*: responderá à requisições no endereço controlador/*new*. Na implementação padrão cria um novo registro do modelo correspondente e encaminha o resultado para uma *view;*
- *edit*: responderá à requisições para o endereço do controlador controlador/id do modelo/*edit*. Na implementação padrão ele é semelhante ao show, mas

encaminha o resultado para uma *view* diferente;

- *create*: responderá à requisições do tipo POST para o endereço no controlador. Na implementação padrão cria um novo registro utilizando os parâmetros recebidos, se tudo ocorrer bem, redireciona o *request* para o show do novo registro, se houver algum erro, retorna o *request* para a pagina *new;*
- *update*: responderá requisições do método HTTP PUT para o endereço controlador/ID do registro. Na implementação padrão busca um registro no banco de dados e atualiza os dados com o valores do parâmetro recebidos. Se houver algum erro retorna para a página de edição, se não, redireciona para a pagina show do registro alterado;
- *destroy*: responde às requisições do tipo DELETE para o endereço controlador/id do registro. Por padrão busca pelo registro referenciado pelo id recebido como parâmetro, que se for deletado com sucesso, redireciona para a página de lista de registros, ou se houver algum erro, mostra uma mensagem.

4.6 LAYOUTS

Segundo Tate e Hibbs (2006, p. 75-78), o *Ruby on Rails* usa um recurso denominado *layouts*, que serve para especificar um conjunto de elementos comuns para as páginas renderizadas pelos controladores. Esses elementos geralmente são cabeçalhos, rodapés e barras laterais. Um *layout* pode ser usado para alguns *controllers* específicos, sendo que assim em cada um deles deve ser informado o *layout* que vai usar, ou então, pode-se usar um mesmo *layout* para toda a aplicação, definindo na superclasse que se encontra no diretório app/*controllers*/ no arquivo application.rb, se usarmos um *layout* chamado vermelho a declaração ficaria como no quadro abaixo.

Quadro 11 - Layout Padrão para a Aplicação

```
Class ApplicationController < ActionController::Base
      layout 'vermelho'
end
```

4.7 BANCO DE DADOS

Os bancos de dados surgiram em meados de 1960 por causa da necessidade de computadores armazenarem dados estruturados permanentemente com rotinas padronizadas disponíveis para sua manipulação. Um SGBD(Sistema Gerenciador de Banco de Dados) é um conjunto de softwares destinados a controlar todos os aspectos do banco de dados, como a estrutura, leitura, gravação e recuperação(se houver falhas), além de controles de concorrência, de segurança e de acesso (NASSU e SETZER, 1999, p.1).

Segundo Beaulieu (2010, p. 30), o MySQL é uma opção aos bancos de dados comerciais mais consolidados e difundidos, como Oracle *Database* e SQL *Server*. Junto com o PostgreSQL, o MySQL é uma alternativa *open source* para os bancos de dados pagos que inclusive implementam o padrão ANSI SQL, que permite que a mesma instrução SQL seja executada como sucesso independente do tipo de banco de dados.

Por sua vez o PostgreSQL também é muito simples de instalar inclusive na plataforma Windows, depois a partir da versão 8. Apesar do PostgreSQL ter sido iniciado como um projeto fechado, ele teve seu código aberto em meados de 1996, um pouco depois de receber o primeiro interpretador SQL baseado no SQL ANSI 92, hoje ele implementa também o SQL ANSI 96 e o 99, esse último abrange algumas definições da Orientação a Objetos (PEREIRA NETO, 2007, p. 24-30).

4.7.1 Bancos de Dados no Ruby on Rails

Uma das características do *Rails* é a facilidade de se comunicar com diversos SGBD de modo transparente ao programador, sem necessitar de alterações além da configuração de conexão. Ao criar o projeto, o *Ruby on Rails* cria também um arquivo de configuração do banco que está localizado em *config/database.yml* (CAELUM, 2011, p.45). Segundo Leme (2009, p. 33), o arquivo possuí a extensão .yml, o que indica que utiliza a linguagem de domínio YAML (*YAML Ain't Markup Language*). YAML é um formato de codificação de dados legíveis por humanos, inspirados em linguagens como XML, C, *Python*, *Perl*.

Segundo Tate e Hibbs os parâmetros de configuração de conexão com o banco de dados são:

- *adapter*: nome do adaptador do banco de dados;
- *database*: nome da data-base;
- *username*: nome do usuário;
- *password*: senha do usuário;
- *host*: *host* aonde está o banco de dados.

Ainda segundo Tate e Hibbs (2006), o *Ruby on Rails* tem três ambientes para o banco de dados: *development* (desenvolvimento), *test* (testes) e *production* (produção). No ambiente de desenvolvimento, as Classes são recarregadas a cada ação, o que faz que se perda no desempenho mas se ganhe entre a alteração e exibição, enquanto no ambiente de produção carrega as classes apenas uma vez. No ambiente de testes, o banco de dados é gerado e removido a cada teste, por isso pede-se atenção na hora de configurar os três ambientes.

Quadro 12 - Exemplo de Arquivo de Configuração Banco de Dados

```
# MySQL.  Versions 4.1 and 5.0 are recommended.
#
# Install the MySQL driver:
#   gem install mysql2
#
# And be sure to use new-style password hashing:
#   http://dev.mysql.com/doc/refman/5.0/en/old-client.html
development:
  adapter: mysql2
  encoding: utf8
  reconnect: false
  database: florarails_development
  pool: 5
  username: root
  password: cafe
  socket: /var/run/mysqld/mysqld.sock
```

Fonte: LEME (2009).

4.8 MIGRATIONS

> Para manter a portabilidade e facilitar o controle de versionamento do banco de dados, o *Rails* utiliza um conceito chamado *migrations*(migrações). As migrações são *scripts* em *Ruby* descrevendo modificações de qualquer natureza no banco de dados e são a maneira mais fácil de acrescentar tabelas e classes de dados ao *Rails* (LEME, 2009, p. 45)

Ainda segundo Leme (2009, p. 44-47), ao criar um *model* pelo *Framework*

automaticamente é criado uma *migration*, ou também através do comando *generate migrati*on nome_da_migration, as *migrations* ficam no diretório db/migrate, e o seu nome é composto pela hora *Universal Time Coordinated* (UTC), seguido do nome dado no comando ou o nome do *model*, com a extensão .rb. O arquivo que é gerado contém uma classe com dois métodos precedidos pelo *self*, o que indica que são estáticos, os mesmos são self.up e self.down, servem respectivamente para armazenar uma sequência de códigos para fazer algo no banco de dados, como criar uma tabela, por exemplo , e para desfazer, deixando igual ao estado anterior.

Quadro 13 - Exemplo de Migration

```
class CreateGrupos < ActiveRecord::Migration
  def self.up
    create_table :grupos do |t|
      t.string :descricao

      t.timestamps
    end
  end

  def self.down
    drop_table :grupos
  end
end
```

5 DESENVOLVIMENTO

Nesse capítulo serão abordados os fatos concernentes ao desenvolvimento do protótipo.

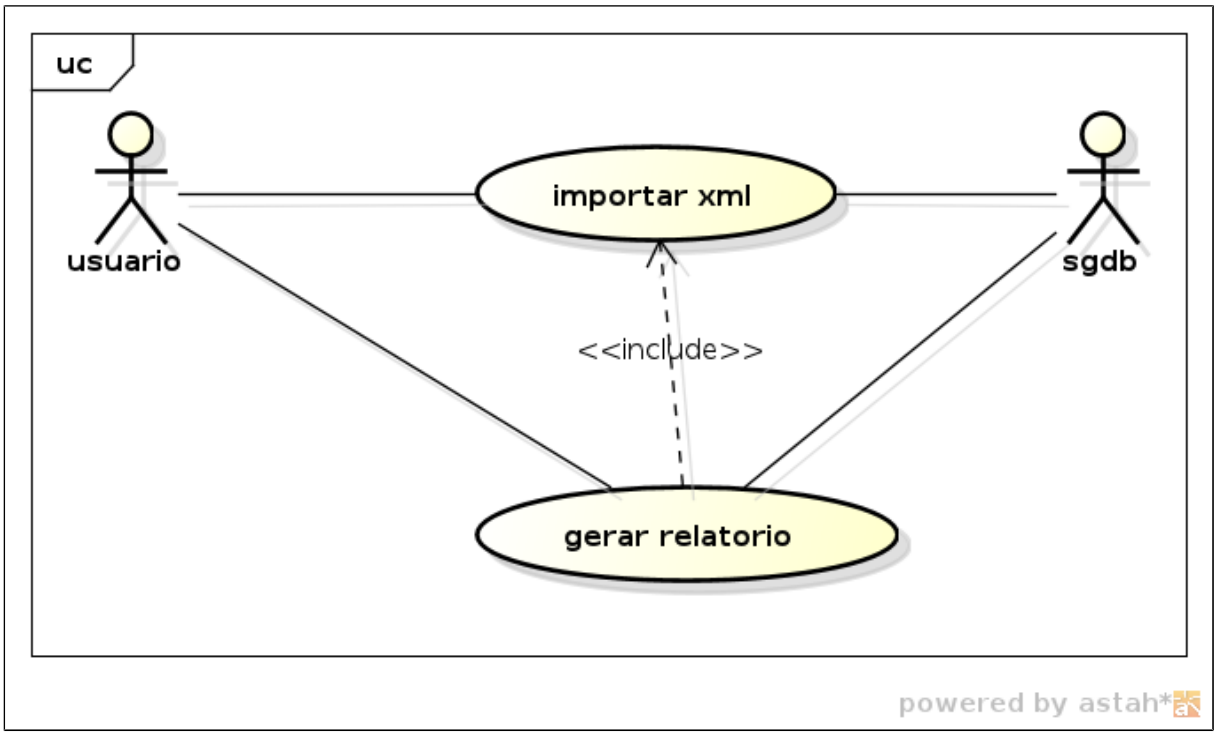
O projeto recebeu o nome CRH, iniciais do nome do autor desse trabalho, e foi versionado pelo GIT.

Os arquivos XML para o teste e desenvolvimento do protótipo, foram encontrados na internet nos links <http://nf-eletronica.com/blog/?p=77#more-77> e <http://www.robertodiasduarte.com.br/nf-e-exemplos-de-xml-e-danfe/>, e encontram-se na pasta *samples* do projeto.

5.1 DIAGRAMAS

Os diagramas criados na analise, fase inicial do desenvolvimento, demonstram as funcionalidades e funcionamento do sistema e serão demonstrados a seguir, iniciando pelo Diagrama de Caso de Uso, na Figura 8, que demonstra de forma sintética o funcionamento do sistema, onde o ator usuário irá importar o XML da NFe para o ator SGBD, fato obrigatório para posteriormente o ator usuário solicitar o relatório para o qual o ator SGBD fornecerá os dados.

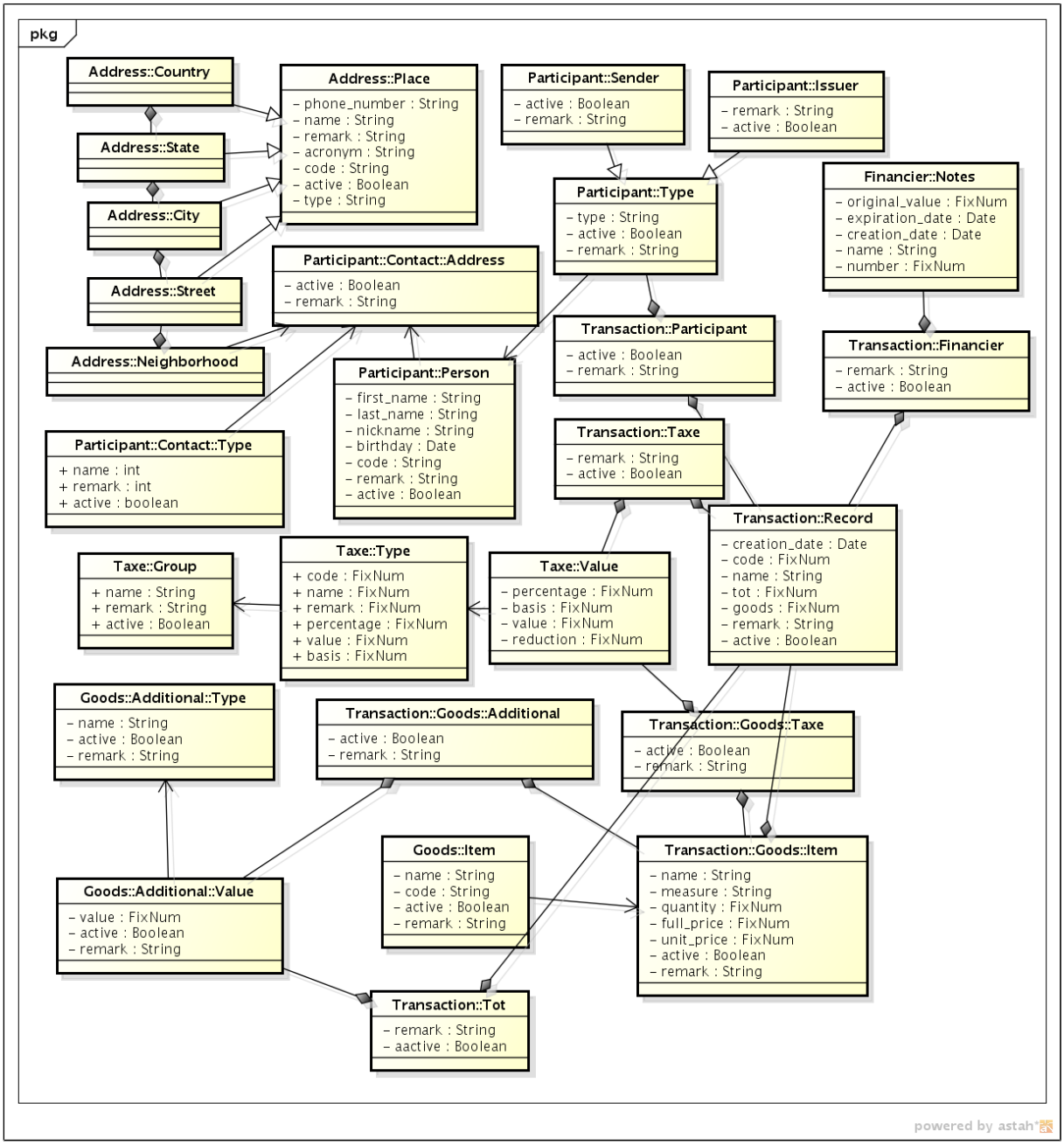
Figura 8 - Diagrama de Caso de Uso



Um dos objetivos do projeto foi seguir os conceitos propostos pelo *Framework Ruby on R*ails, por isso os nomes das propriedades, classes e métodos foram designados em idioma inglês, como pode ser visto na Figura 9, assim evitando problemas, como no caso da classe *Person*, que tem o nome da tabela a qual ela se refere no plural, que gramaticamente correto seria *people* e não *persons*. Lembrando que o não uso dessa convenção traria configurações que atrasariam o desenvolvimento.

Outro ponto interessante é que relacionamento entre as tabelas no banco de dados é feito somente através da aplicação, isso pode até parece ser estranho, mas avaliando o encapsulamento que o *ActiveRecord* proporciona, podendo usar a mesma aplicação para vários SGDB's, o fato da chave estrangeira, excluiria da lista de bancos de dados alguns bancos como o SQLite ou algumas *engines* do MySQL. Dessa forma que o relacionamento entre as tabelas funciona, possibilita usar SQLite para ambiente de desenvolvimento, MySQL para testes, e PostgreSQL para produção, por exemplo, alterando apenas o arquivo de configuração como visto no capítulo 4.7.1 deste trabalho.

Figura 9 - Diagrama de Classes

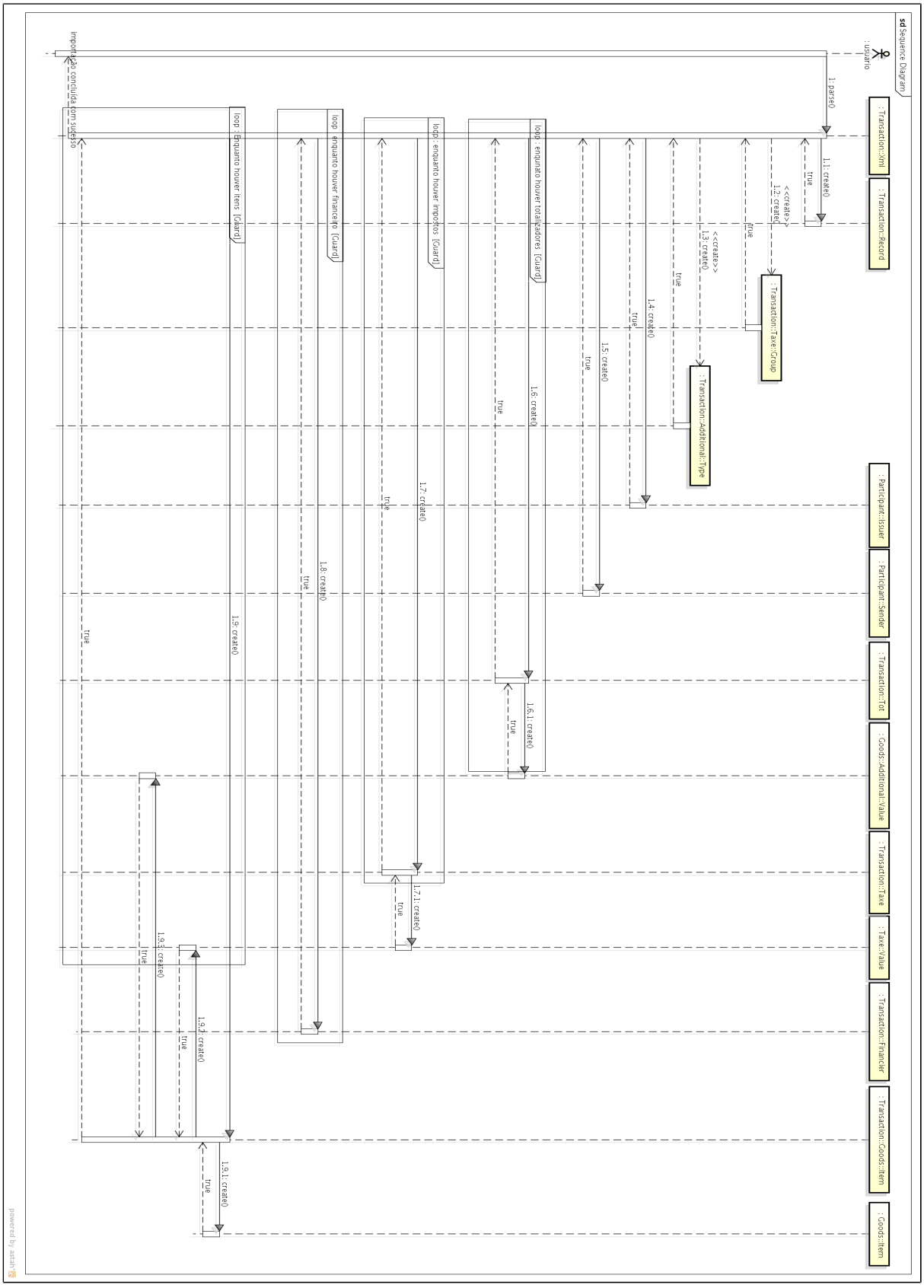


O Diagrama de Classes visto, se refere a camada *Model* do MVC, a maioria das classes não possui métodos, porque seus métodos são herdados de sua classe pai, a ActiveRecord::Base, isso inclui os métodos para criar, excluir, atualizar e pesquisar. A classe Transaction::Xml é uma exceção, pois além dos métodos herdados possui vários métodos próprios.

Por sua vez, o Diagrama de Sequência, Figura 10, demonstra a troca de mensagem entre métodos de várias classes, onde é realizado o parser do XML da NFe, sendo lido as

informações e usando as facilidades do *framework*, os dados são persistidos no SGBD.

Figura 10 - Diagrama de Sequência



5.2 PESQUISA DE INFORMAÇÕES PERSISTIDAS NO SGBD

Para recuperar os dados do banco de dados, foram utilizadas duas formas, uma onde os dados foram recuperados através dos métodos que o *Framework ActiveRecord* disponibiliza, Quadro 14, encapsulando o SQL, e um outro, Quadro 15, com uma *string* com o comando SQL a ser executado.

Quadro 14 - Consulta de Dados *ActiveRecord* sem SQL

```
@taxe = Taxe::Group.all(:order => :id)
@taxe.each do |taxe|
      @taxeTot[taxe.id] = 0.00
      @taxeDet[taxe.id] = 0.00
      @taxeCab[taxe.id] = taxe.remark
end
```

Quadro 15 - Consulta de Dados *ActiveRecord* com SQL

```
@additional = Goods::Additional::Type.find_by_sql(
      "SELECT
            goods_additional_types.id,
            goods_additional_types.remark
      FROM
            goods_additional_types
            INNER         JOIN         goods_additional_values         on
goods_additional_types.id                                              =
goods_additional_values.goods_additional_type_id
            INNER           JOIN           transaction_tots           on
transaction_tots.goods_additional_value_id=goods_additional_values.id
      GROUP BY
            goods_additional_types.id,goods_additional_types.id
      ORDER BY
            goods_additional_types.id"
)
@additional.each do |additional|
      @additionalTot[additional.id] = 0.00
      @additionalDet[additional.id] = 0.00
      @additionalCab[additional.id] = additional.remark
end
```

5.3 INTERNACIONALIZAÇÃO

No momento em que as *Views* são criadas automaticamente através do uso do comando *scaffold*, os *labels* levam automaticamente o nome das respectivas colunas que representam.

Para uma maior manutenibilidade do sistema, no momento da troca desses nomes por algo mais humanizado, foi utilizado a API (*Application Programming Interface*) de Internacionalização que o *Ruby on Rails* disponibiliza, denominada I18n.

Inicialmente criou-se um arquivo, que no caso foi batizado de pt-BR.rb, dentro do diretório config/locales, cujo uma parte do código se encontra no Quadro 16, atribuindo aos símbolos a sua tradução humanizada.

Quadro 16 - Código Arquivo de Internacionalização

```
{
      :'pt-BR' => {
            :new => "Novo",
            :back => "Voltar",
            :destroy => "Excluir",
            :show => "Mostar",
            :edit => "Editar",
            :create => "Criar",
            :save => "Salvar",
            :listing => "Listando",
      }
}
```

Para que o idioma seja aplicado, é preciso defini-lo nas classe AplicationController presente em app/controllers/aplication_controller.rb com um método chamado *localizate*, como exemplificado no Quadro 17.

Quadro 17 – Definição do Idioma

```
#método responável pela internacionalização
def localizate
      I18n.locale = params[:locale] || 'pt-BR'
end
```

No local onde é necessário a tradução, é usado o método I18n.t, passando com parâmetro o símbolo definido no arquivo pt-BR.rb, como pode ser observado no Quadro 18.

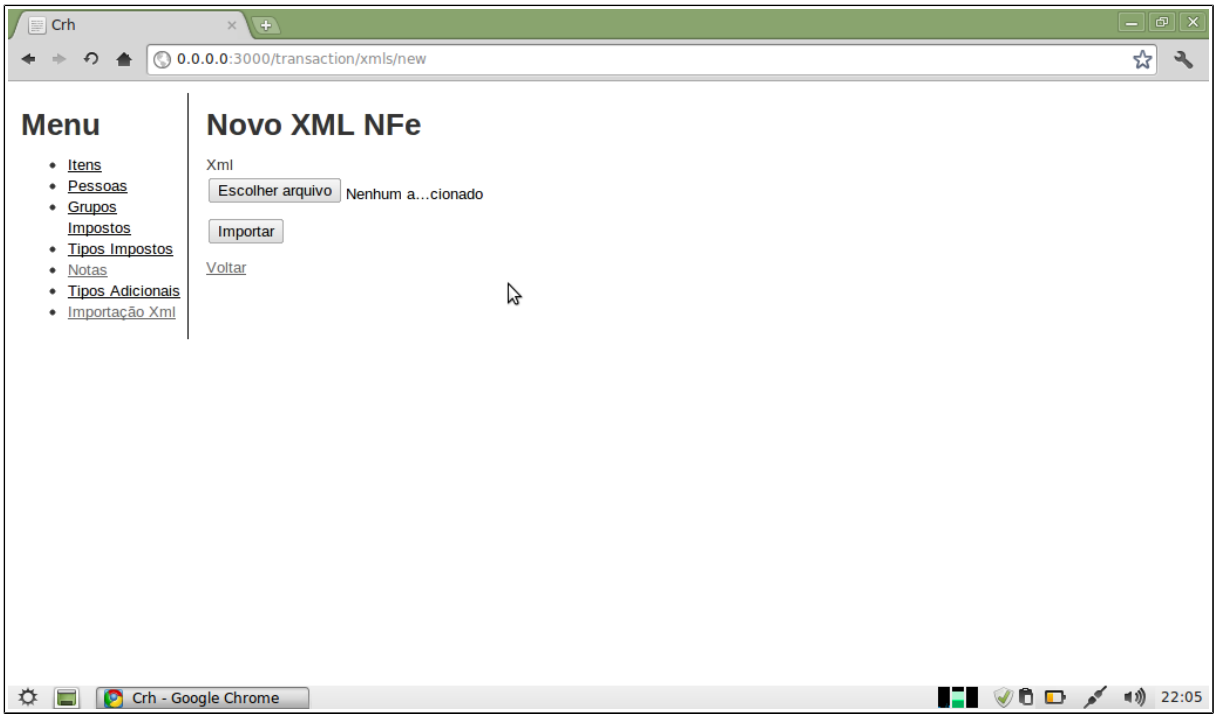
Quadro 18 – Exemplo de Chamado da API de Internacionalização

```
I18n.t(:edit)
```

5.4 PRINCIPAIS TELAS

O formulário que o usuário utilizará para importar o XML de NFe, conta com apenas um campo e se encontra na *view* chamada *new,* da classe Transaction::Xml.

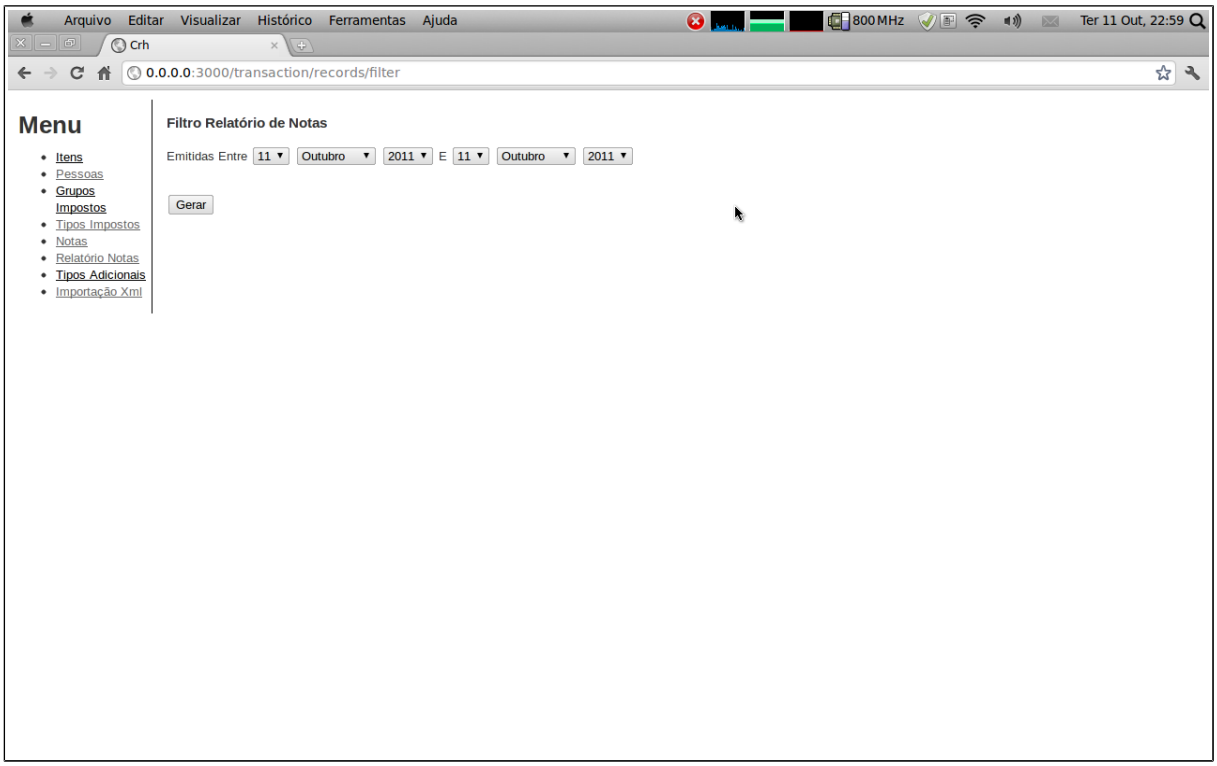
Figura 11 – Formulário de Importação de XML



Clicando no botão escolher arquivo, se abrirá uma janela, na qual o usuário escolherá em seu computador o arquivo que deseja importar, e depois clicará em importar. O sistema executara a rotina descrita no Diagrama de Sequência da Figura 10.

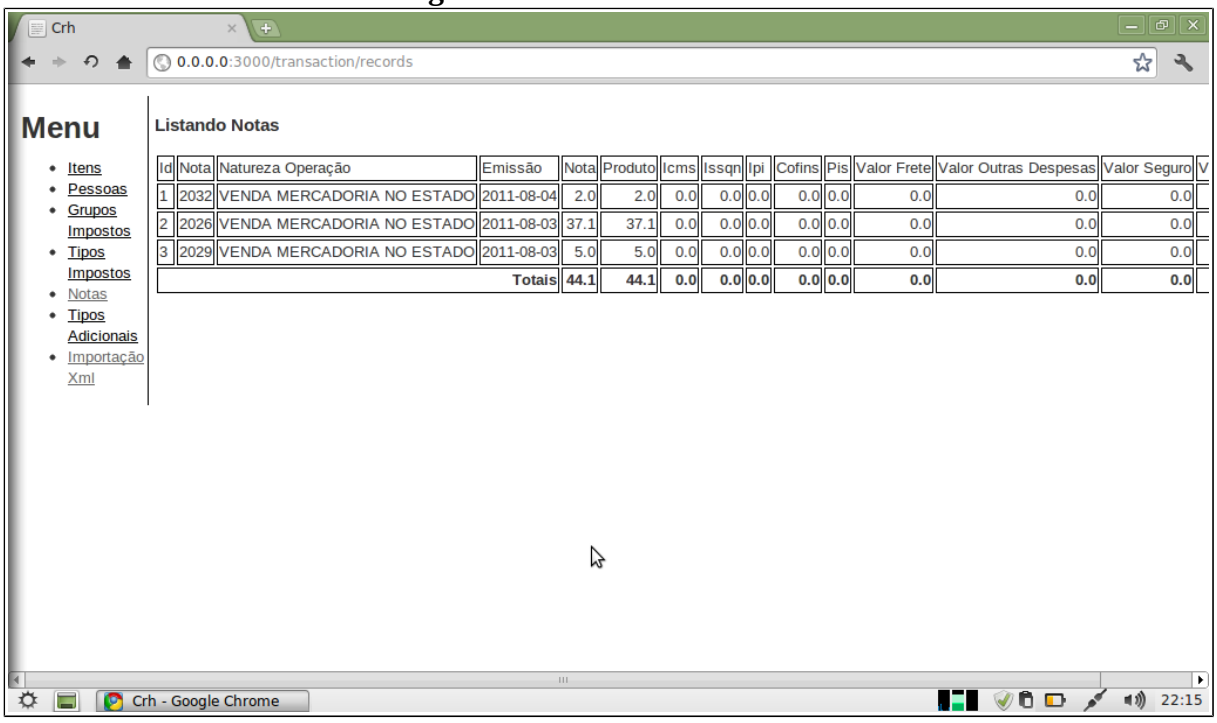
Com os arquivos XML importados para o SGBD, já será possível visualizar o relatório de notas, que tem a tela exibida na Figura 12, nessa tela ele poderá selecionar o período desejado e em seguida clicará em gerar.

Figura 12 – Tela de Filtro Relatório de Notas



Em seguida o sistema será direcionado para a *view report*, que está encarregada de renderizar as informações conforme o período escolhido anteriormente pelo usuário. O relatório mostra algumas informações com os totalizadores de campos numéricos:

Figura 13 – Relatório de Notas



Id	Nota	Natureza Operação	Emissão	Nota	Produto	Icms	Issqn	Ipi	Cofins	Pis	Valor Frete	Valor Outras Despesas	Valor Seguro
1	2032	VENDA MERCADORIA NO ESTADO	2011-08-04	2.0	2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	2026	VENDA MERCADORIA NO ESTADO	2011-08-03	37.1	37.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	2029	VENDA MERCADORIA NO ESTADO	2011-08-03	5.0	5.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
			Totais	**44.1**	**44.1**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

6 CONCLUSÃO

No decorrer da efetivação desse trabalho, grandes obstáculos foram ultrapassados. O principal deles foi a falta de alguém que conhecesse o *Framework Ruby on Rails* e que pudesse auxiliar no desenvolvimento. Por muitas vezes perdi horas em problemas simples, mas sobretudo pude comprovar, mesmo com essas dificuldades, que essa ferramenta é realmente fantástica e confesso que consegui apenas conhecer superficialmente seu poder.

Em dias onde há pluralidade de Sistemas Operacionais, várias opções de SGBD, sistemas legados ou ferramentas já consolidadas para se manter, softwares escritos com a ajuda do dinamismo do *Ruby on Rails* podem tornar as tarefas menos complicados e doloridas.

A forma como o *Ruby* trabalha, disponibilizando gerenciador de pacotes próprio, induz ao reaproveitamento de código. O próprio *Ruby on Rails* também, seguindo a mesma ideia do *Ruby*, disponibiliza vários pacotes de códigos que podem ser usados como *plugins* ou como uma *gem*. Isso é de grande ajuda, mesmo para quem não queira disponibilizar o fonte, mas quer automatizar alguns procedimentos repetitivos e reutilizar pacotes de código em projetos diferentes.

O desenvolvimento do protótipo assim como dessa monografia, são o produto dos objetivos propostos desde o inicio, o cumprimento destes levou há uma grande bagagem de conhecimento que será muito valioso profissionalmente.

Aos que desejarem usar o sistema, os fontes estão disponíveis no GitHub, mais precisamente no endereço <https://github.com/clairton/crh>, sobre a licença GNU *General Public License*. Quem desejar participar do projeto pode entrar em contato comigo no mesmo endereço ou pelo e-mail clairon.rodrigo@gmail.com.

6.1 TRABALHOS FUTUROS

Para trabalhos futuros há várias coisas a fazer, pois a complexidade das informações contidas em um XML de NFe é bem grande, assim seria necessário implementar alguns impostos que estão faltando como o ISS.

Algo de grande importância também seria escrever testes unitários e funcionais que dão mais confiabilidade a segurança para desenvolver novas funcionalidades sem que as já

existentes sejam perdidas ou danificadas acidentalmente.

Outro fator é a integridade da XML que se está tentando importar. A partir do momento em que ele é assinado e posteriormente autorizado pelo SEFAZ, não pode mais ser alterado, essa validação de adulteração do conteúdo pode ser feita através da assinatura presente no próprio arquivo. Ainda, fiscalmente falando, seria interessante consultar o status de NFe naquele momento no SEFAZ, afirmando se ela está ou não autorizada.

Poderia também ser incorporado a visualização de do DANFE, através dos dados do XML.

A interface atual do protótipo se encontrado bem simples, recursos como CSS, Ajax podem ser aplicados para melhorar a usabilidade do sistema.

Considerando os dados disponíveis no SGBD, podem ser gerados uma infinidade de gráficos e relatórios, até mesmo pode ser usada uma API ou *gem*, já existente ou então fazer uma, para o próprio usuário poder personalizar os relatórios e gráficos da forma que achar necessário sem precisar entender de programação.

REFERÊNCIAS

ALVAREZ, Miguel Angel. **Introdução ao HTML.** Disponível em <http://www.criarweb.com/artigos/10.php> Criar Web. Acesso em: 16 mai. 2011.

AMARAL, Juliana Alves. **Engenharia de software orientada para a Web: intercâmbio de Frameworks através de XML.** Belo Horizonte: C/ Arte, 2003.

AUGUSTO, José Soares. **Guia do Utilizador de Ruby**. Disponível em: <http://calypso.inesc-id.pt/jasa/scripts/uguide/uguide00.hhtml>. Acesso em: 28 mar. 2011.

BEAULIEU, Alan. Uma Breve Introdução. In:___.**Apredendo SQL**. São Paulo. Novatec, 2010.

CAELUM. **Desenvolvimento Ágil para Web 2.0 com Ruby on Rails** . Disponível em <http://www.caelum.com.br/download/caelum-ruby-on-rails-rr71.pdf>. Caelum. Acesso em: 6 mar. 2011.

COMUNIDADE RUBY. **Ruby a programer's best friend**. Disponível em: <http://www.ruby-lang.org/pt/>. Acesso em: 27 mar. 2011.

DUARTE, Roberto Dias. **Big Brother Fiscal – III O Brasil Na Era Do Conhecimento**. Disponível em <http://www.robertodiasduarte.com.br/files/bbf3-v1.01s.pdf>. Acesso em: 25 mar. 2011.

DUARTE, Roberto Dias. **NF-e: Exemplos de XML e DANFE**. Disponível em <http://www.robertodiasduarte.com.br/nf-e-exemplos-de-xml-e-danfe/>. Acesso em: 25 mar. 2011.

EIS, Diedo. **Uma pequena introdução sobre HTML5 e sua história.** . Disponível em <http://www.tableless.com.br/html5-brevissima-introducao>. Acesso em: 16 mai. 2011.

ELIAS, Wagner. **HTTP Essentials**. Disponível em <http://wagnerelias.com/2009/02/06/httpessentials/>. Wagner Elias - Think Security First. Acesso em: 02 mai. 2011.

ENCAT. **Manual de Integração – Contribuinte Padrões Técnicos de Comunicação** .

Disponível em: <http://www.nfe.fazenda.gov.br/portal/exibirArquivo.aspx?conteudo=zxlLdxB/oYA=>. Acesso em: 09 mai. 2011.

GAMA, Erich et al. **Padrões de projeto: soluções reutilizáveis de software orientado a objetos**. Porto Alegre: Bookman, 2005.

GOULART, Gardiz. **Programação Orientada a Objetos** . Pinhalzinho, 2010. (Apostila da disciplina Programação Orientada a Objetos, Curso de Sistemas de Informação Horus Faculdades).

GOULART, Gardiz. **Paradigmas de Linguagem de Programação** . Pinhalzinho, 2010. (Apostila da disciplina Programação Orientada a Objetos, Curso de Sistemas de Informação Horus Faculdades).

LEME, Ricardo Roberto. **Desenvolvendo Aplicações Web com Ruby on Rails 2.3 e Posgresql.** Rio de Janeiro: Brasport, 2009.

LISBOA, Flavio Gomes da Silva. **Zend Framework componentes poderosos para PHP**. São Paulo, Novatec, 2009.

MILANI, André. **Construindo Aplicações Web com PHP e MySQL**. São Paulo, Novatec, 2010.

NASSU, Eugênio A., SETZER, Valdemar W.. **Bancos de dados orientado a Objetos**. São Paulo: Edgar Blucher, 1999.

NF-eletrônica Nacional, **Nota Fiscal eletrônica - exemplos de arquivo XML**. BRASIL. Disponível em <http://nf-eletronica.com/blog/?p=77#more-77>. Acesso em: 01 jun. 2011.

OLIVEIRA JUNIOR, Eustaquio Rangel De. **Ruby: Conhecendo a Linguagem**. Rio de Janeiro: Brasport, 2006.

PEREIRA NETO, Alvaro. Introdução ao PostgreSQL e a Linguagem SQL. In:___.**PostgreSQL Técnicas Avançadas Open Source 7.x e 8.x Soluções para Desenvolvedores e Administradores de Bancos de Dados**. 4 ed. São Paulo. Érica, 2010.

OLIVEIRA JUNIOR, Eustaquio Rangel De. **Tutorial de Ruby** . BRASIL. Disponível em: <http://eustaquiorangel.com/downloads/tutorialruby.pdf>. Acesso em: 01 mai. 2011.

RECEITA FEDERAL DO BRASIL. **Portal da Nota Fiscal Eletrônica**. BRASIL. Disponível

em: <http://www.nfe.fazenda.gov.br>. Acesso em: 02 fev. 2011.

RECEITA FEDERAL DO BRASIL. **Sistema Público de Escrituração Digital**: BRASIL. Disponível em: <http://www1.receita.fazenda.gov.br>. Acesso em: 02 fev. 2011.

SILVA, Ricardo Pereira. **UML2 em Modelagem Orientada a Objetos**. Florianópolis. Visual Books, 2007.

TATE, Bruce A.,HIBBS, Curt, **Ruby on Rails Executando**. Rio de Janeiro: Alta Books, 2006.

TIOBE, Software. **Tiobe software**. Disponível em <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>. Acesso em: 28 mar. 2011.

URUBATAN, Rodrigo. **Ruby on Rails: desenvolvimento fácil e rápido de aplicações Web.** São Paulo: Novatec, 2009.

VALENTE, Fernando. **Fernando Valente Programação e Tecnologia**. Disponivel em <http://www.fernandovalente.com.br/wordpress/tag/mvc/>. Acesso em: 18 abr. 2011.

WAZLAWICK, Raul Sidnei. **Análise e Projeto de Sistemas de Informação Orientados a Objetos**. Rio de Janeiro, Elsevier, 2004.