UNIVERSIDADE FEDERAL DO TOCANTINS
CAMPUS UNIVERSITÁRIO DE PALMAS
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

GETNA: GERADOR DE CÓDIGO RUBY ON RAILS A PARTIR DE BANCO DE DADOS

Palmas
2008

LUIZ ARÃO ARAÚJO CARVALHO

GETNA: GERADOR DE CÓDIGO RUBY ON RAILS A PARTIR DE BANCO DE DADOS

Monografia apresentada como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação pela Universidade Federal do Tocantins.

Orientador: MSc. Thereza Patrícia Pereira Padilha.

Palmas

2008

DEDICATÓRIA

Dedico este trabalho a Deus e a todos meus familiares, em especial, minha mãe e meu pai.

AGRADECIMENTOS

Primeiramente, gostaria de agradecer a Deus, por ter me concedido a vida e todas as oportunidades e conquistas que me levaram a estar aonde estou hoje.

Aos meus pais, Cristóvão Rodigues de Carvalho e Maria José Araújo Carvalho, as pessoas mais importantes pra mim. Deram-me estudo, conforto, cultura e lições que formaram o homem que sou hoje. Foram a base de todo o empenho que realizei e o abrigo nas horas difíceis que enfrentei.

Aos meus irmãos, Andréia Araújo Carvalho e Cristovão Junior (o pica pau) que foram companheiros durante toda minha vida, nos bons e maus momentos que vivi. Um agradecimento especial ao meu irmão, pois liberou o computador de seu *Dotinha* para que pudesse desenvolver essa monografia.

A todos os professores da UFT pelos ensinamentos. Principalmente, a minha orientadora, Thereza Patrícia Pereira Padilha, pelos conselhos e puxões de orelha. Uma pessoa que tem todo o crédito nessa monografia, pois chegou a sentar ao meu lado e reescrever parte desta. Dedicou o seu tão precioso tempo em horas de revisões.

Às minhas namoradas espalhadas pelo Brasil, pois não pude dar atenção a nenhuma delas.

Agradeço a todos os meus amigos que, diretamente ou indiretamente, contribuíram com esse trabalho. Em especial, Aéccio, Brunno, Charles, Daiene, Fernando, Paulo Roberto, Ribeiro, Robério, Rosana e Sofia, além de toda a turma que entrou no curso junto comigo.

E a toda a comunidade de desenvolvedores *Ruby* e *Rails* que contribuíram para esse projeto, em especial, a alguns membros da rails-br, Silvio Fernandes, Vinicius Luiz, Junio Gonsalvez, Ricardo Yasuda, Cairo Noleto, Leandro Camargo, Felipe Diesel e Davis Zanetti.

“*Para quem só sabe usar martelo*,

todo o *problema é um prego.*”

(Abraham Maslow)

RESUMO

Na literatura, existem vários geradores de código disponíveis que tem como principal propósito automatizar o processo de desenvolvimento de aplicações em diversas linguagens de programação. Este trabalho apresenta o desenvolvimento de um gerador de código, denominado GEtna, com a função de criar as classes a partir de um determinado banco de dados. Vale a pena ressaltar que o gerador GEtna difere dos existentes pelas suas funcionalidades disponibilizadas.

O gerador proposto aumenta a velocidade do processo de desenvolvimento de uma aplicação Rails, pois recebe como entrada tabelas de um banco de dados relacional (MySQL, PostgreSQL ou SQLite) e fornece como saída toda a estrutura de arquivos (classes). Vale a pena ressaltar que o GEtna proporciona uma padronização das classes geradas e, conseqüentemente, sustenta sua legibilidade, que é um fator muito importante quando se trabalha com equipes de desenvolvedores. Assim, neste trabalho será apresentada uma descrição sobre geradores de código, padrões de projeto, métodos de mapeamento e definições de dados que são fundamentais para o desenvolvimento de um gerador de código.

Palavras-chave: Banco de Dados Relacional, Desenvolvimento de Sistema, Gerador de Código, Rails.

ABSTRACT

In literature, there are several available code generators that have the main goal to automatize the application development process for some programming languages. This work presents the development of a code generator, named GEtna, that creates classes from a specific database. It's important to note that GEtna generator differs of other generators by your available functionalities.

The proposed generator increases the velocity of a Rails application development process because it receives as input relational database tables (MySQL, PostgreSQL or SQLite), and provides as output the complete file structure (class). The GEtna provides a padronization of generated classes and, consequently, maintains the legibility that is a very important factor when there are developers' groups. So, in this work a description about code generators, design patterns, mapping methods and data definition is presented because they are essential for a code generator development.

Keywords: Relational Database, System Development, Code Generator, Rails.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE SIGLAS

BSD	*Berkeley Software Distribution*
CMS	*Content Management Systems*
CPU	*Central Processing Unit*
CRUD	*Create, Retrieve, Update* e *Delete*
CSS	*Cascading Style Sheets*
DRY	*Don't repeat yourself*
GPL	*GNU General Public License*
IDE	*Integrated Development Environment*
LGPL	*Lesser General Public License*
MIT	*Massachusetts Institute of Technology.*
ORM	*Object-Relational Mapping*
PHP	*PHP: Hypertext Preprocessor*
POLS	*Principle of Least Surprise*
REST	*Representational State Transfer*
SGBD	Sistema Gerenciador de Banco de Dados
SQL	*Structured Query Language*
SSH	*Secure Shell*
TAR	*Tape ARchive*
URL	*Uniform Resource Locator*
WWW	*World Wide Web*

SUMÁRIO

1 INTRODUÇÃO

Atualmente, está sendo vivenciada uma era digital em que o tempo é algo que se tem tornado uma variável bastante importante em diversas áreas de atuação. Dentre estas áreas, destaca-se o desenvolvimento de software, pois há constantes e intensas imposições de prazos e cronogramas que, em sua maioria, não são cumpridos. Mesmo existindo diversas tecnologias para melhorar o desempenho de atividades e, assim, diminuir a perda de tempo, ainda têm-se empecilhos que atrasam a busca por um custo temporal aceitável.

Diante desse contexto, na literatura, é possível encontrar ferramentas para otimizar estas atividades, tal como *Rails*. O *Rails* é um excelente *framework* para agilizar diversos procedimentos na construção de uma aplicação, sendo de fácil legibilidade e de programação, porém ainda existem alguns procedimentos repetitivos que podem ser automatizados por meio de um gerador de código.

Os geradores de código podem construir código em várias linguagens de programação, bem como efetuar a geração de uma única vez ou em etapas. Podem ser manipulados utilizando interfaces gráficas ou por meio de linhas de comando. As entradas e saídas são definidas conforme a necessidade, sendo que o desenvolvedor especifica parâmetros de saída de forma manual. Portanto, é de extrema relevância que o programador saiba a priori o que se deseja obter como resultado.

Diante deste contexto, este projeto de graduação propõe a criação de um gerador de código, denominado GEtna, que a partir de um banco de dados padronizado de acordo com as convenções do *Rails*, crie toda a estrutura de arquivos e classes necessárias, além de validações de dados, migrações, personalizações e internacionalização. O código gerado possui as funcionalidades de inserção, deleção, alteração e listagem. Segue ainda o padrão de nomeação de tabelas e de atributos estabelecido pelo *Ruby On Rails* e a organização de arquivos e responsabilidades utilizados no MVC (*Model* – Modelo, *View* – Interface, *Controller* - Controlador) por meio do uso de templates .

Este projeto está organizado em seis capítulos. O segundo capítulo aborda todo o material e as tecnologias utilizados como referência no desenvolvimento do projeto, como geradores, ORM, a linguagem, o *framework* e as licenças. O terceiro capítulo contém métodos utilizados durante todo o processo de criação e execução do projeto. As funcionalidade e particularidades do GEtna são mostradas no capítulo quatro. O quinto capítulo apresenta os resultados obtidos com a utilização do gerador. Por fim, a conclusão e as possíveis melhorias para este projeto serão apresentadas no capítulo seis.

2 REVISÃO DE LITERATURA

Este capítulo apresenta, inicialmente, uma visão geral sobre geradores de código. Em seguida, será apresentada uma descrição sobre dois geradores *Ruby* on *Rails* disponíveis na literatura, que são: *Scaffold* e *Goldberg*. Na seqüência, serão abordados conceitos sobre banco de dados relacional e *object-relacional mapping* como forma de manipulação de informações. A linguagem *Ruby* e o *framework Ruby on Rails* complementam também o escopo desse estudo. E, por fim, importantes características de projetos *open source* são apresentadas.

2.1 Geradores de Código

Geradores de código são basicamente programas que geram outros programas. Os geradores podem ser definidos como ferramentas tanto para formatar códigos simples quanto para gerar aplicações complexas a partir de modelos abstratos (*templates*). São muito utilizados para agilizar o processo de desenvolvimento, pois aumentam a produtividade e diminuem o tempo gasto na codificação da aplicação e, conseqüentemente, o custo final (AMBLER, 2004; KLUG, 2007).

Segundo Herrington (2003), a geração de código é a técnica pela qual se constrói código utilizando programas. Os geradores de código podem trabalhar por meio de linhas de comando ou de interfaces gráficas, estas últimas são as formas mais interativas. Estes podem construir código em várias linguagens de programação, bem como efetuar a geração dos códigos de uma única vez ou em etapas. As entradas e saídas são definidas conforme a necessidade, sendo que o desenvolvedor especifica parâmetros de saída de forma manual. Portanto, é de extrema relevância que o programador saiba a priori o que se deseja obter como resultado.

De acordo com Herrington (2003), a utilização de geradores para desenvolvimento de softwares traz enormes vantagens aos códigos gerados, tais como:

- **qualidade**: a escrita de grande quantidade de código vai gradualmente diminuindo a qualidade final do software. Na geração automática, a qualidade do software está diretamente ligada à qualidade dos *templates*. Esse, por sua vez, não possui qualidade associada à quantidade de código a ser desenvolvida, visto que os templates possuem uma estrutura fixa que não é alterada pelo tamanho do projeto;

- **consistência**: o código gerado tende a ser consistente, pois possui uma padronização de nomenclatura de classes, métodos, variáveis e localização (pastas e subpastas), o que torna o código de fácil entendimento e modificação;

- **único ponto de conhecimento**: caso seja necessária uma alteração de alguma estrutura de entrada do projeto, necessita-se apenas saber o local onde essa modificação deve ser realizada. Após isso, o código gerado se propagará para todos os processos seguintes à alteração, gerando sempre um código compatível com as entradas;

- **maior tempo para o projeto**: considerando-se o uso de uma biblioteca externa (API), podem acontecer alterações nessa fonte obrigando assim que modificações sejam feitas em todas as partes do código em que aquela modificação irá influenciar. Com a utilização do gerador, esse processo poderá ser feito apenas nos *templates* do gerador e uma nova geração efetuará as mudanças necessárias no código. Essa característica faz toda a diferença em cronogramas de projetos que utilizam ou não geradores (forma manual);

- **abstração**: como, normalmente, os geradores utilizam os modelos abstratos de dados (*templates*) do código alvo, fica fácil migrar de uma plataforma/linguagem para outra, necessitando apenas uma revisão no

escopo de abstração, ou seja, modificar os *templates* de acordo com as necessidades da plataforma/linguagem alvo;

- **agilidade**: uma característica chave da geração de código é a maleabilidade do código de saída. No nível de negócio, o software será de mais fácil manutenção e implementação de novas funcionalidades. Mesmo considerando-se o tempo que se economiza no ato da geração, podemos afirmar que essa técnica dá um grande percentual de agilidade ao desenvolvedor.

E ainda, segundo Herrington (2003), a construção de um gerador de código pode seguir as seguintes etapas de desenvolvimento:

- a definição de qual a saída se deseja obter;
- a identificação de qual entrada será utilizada e como essa será analisada;
- a definição de padrões na entrada, de como eles serão interpretados para obtenção e geração de uma saída formatada de acordo com as expectativas predefinidas; e
- a geração da saída a partir das informações extraídas da entrada.

Geradores de código são uma boa forma de agilizar o desenvolvimento e melhorar o nível de qualidade no início do projeto. Fatores, esses, que podem decidir o futuro de uma aplicação.

2.2 Geradores para Ruby on Rails

O *Ruby on Rails* não possui somente geradores de estruturas baseadas em entrada do usuário, mas também possui soluções prontas para aplicações em casos específicos. Para isso, nesta seção, dois geradores disponíveis na literatura são

descritos, sobretudo apresentando suas funcionalidades, vantagens e desvantagens.

2.2.1 Scaffold

O *Scaffold* (andaime em português) é uma ferramenta de geração de código que acompanha o *Rails*. Este é eficiente no que se propõe a fazer de um modo rápido para se iniciar o desenvolvimento de uma aplicação. É, também, uma das técnicas de geração de código mais utilizadas junto ao *framework*. Possui ferramentas similares em outras linguagens de programação, como php Scaffold (para PHP) e o *Scaffold* do *Grails* (*framework* para Java inspirado no *Rails*).

Segundo Hansson e Thomas (2008), o Scaffold do *Rails* é uma estrutura auto-gerada para manipular um modelo. Baseia-se na utilização de templates estruturados para a geração de um esqueleto de manipulação de objetos. Cria arquivos a partir de uma entrada do usuário, permitindo que funcionalidades básicas como criar, ler, atualizar e apagar (CRUD) entidades em seu recém-criado *controller* possam ser efetuadas imediatamente com seu aplicativo (WILLIAMS, 2007).

Para efetuar a geração utilizando essa ferramenta, é necessário passar como parâmetro (entrada) o nome da entidade e todos os seus atributos acompanhados com os seus respectivos tipos de dados. Um exemplo das informações necessárias para efetivar o procedimento pode ser encontrado na Figura 1.

```
C:\>script/generate usuario nome:string endereco:string situacao:boolean login:string
senha:string email:string descricao:text
```

Figura 1: Exemplo de comando para geração de código com Scaffold

Neste exemplo, o script, realizado via console, cria o CRUD para a entidade. É informado como parte do parâmetro, atributos seguidos de seus respectivos tipos. Neste caso, por exemplo, tem-se o tipo String para o atributo nome.

Esse processo se torna custoso tanto em relação a tempo quanto a gasto de esforço, ainda por cima sujeito a falhas se não tratado com atenção. Suponha um caso de dezenas de entidades em que o desenvolvedor terá que realizar essa tarefa para cada uma dessas entidades. Depois de gerada, ainda, necessita-se validar todos os campos (como o tamanho da senha do usuário), definir relacionamentos entre entidades e tratar migrações para que espelhem a nova base de dados a ser gerada. Quando se define as validações, como tamanho e presença, e uma nova base de dados for gerada a partir dessa, essa terá que possuir todas essas atribuições da base de dados anterior, esse é um processo que demanda muito tempo e cuidado.

2.2.2 Goldberg

O Goldberg é um gerador de sistemas para gerência de conteúdo, ou seja, este cria um site pré-montado em que é possível adicionar diversos conteúdos como notícias, *blogues*, sites comerciais e entre outros. Goldberg é um gerador *Rails* que permite a criação de uma estrutura para o projeto. Este tem autenticação de usuários, navegação do sistema de menu completamente gerenciável, segurança e todo menu é integrado necessitando apenas *linkarmos* as ações dos *controllers* ou *content view*. Um sistema de CMS (*Content Management Systems*) está integrado dentro dele, sua implementação é extremamente trivial (STRAIOTO, 2007). Esse CMS é um gerenciador de conteúdo que realiza tarefas como o que deve ser mostrado na página inicial e em que ordem essas informações serão apresentadas.

Para instalação desse gerador necessita-se que na máquina possua o *Ruby*, *RubyGem* e o próprio *Rails*, além do *Mysql* como banco de dados. A partir daí deve-se instalar as dependências (*plugins* que o gerador utiliza) para que ele funcione corretamente, que são: *RedCloth* (criação de textos no formato *Textile*) e *Plugin Migrations* (auxílio de migrações de banco).

Após essa parte de instalação, é necessária a configuração do projeto *Rails* para o funcionamento do gerador que consiste em criar um diretório chamado .rails e

um subdiretório chamado *generators*. Em seguida, por meio de linha de comando, é possível realizar o processo de geração, sendo que com o uso de parâmetros, pode-se definir o layout que o site possuirá (quatro estilos diferentes). Um exemplo de um projeto gerado por esse gerador pode ser observado na Figura 2. Uma interface limpa é o que apresenta a página inicial do Goldberg, com uma barra com menus de acesso e um formulário de *login* lateral.

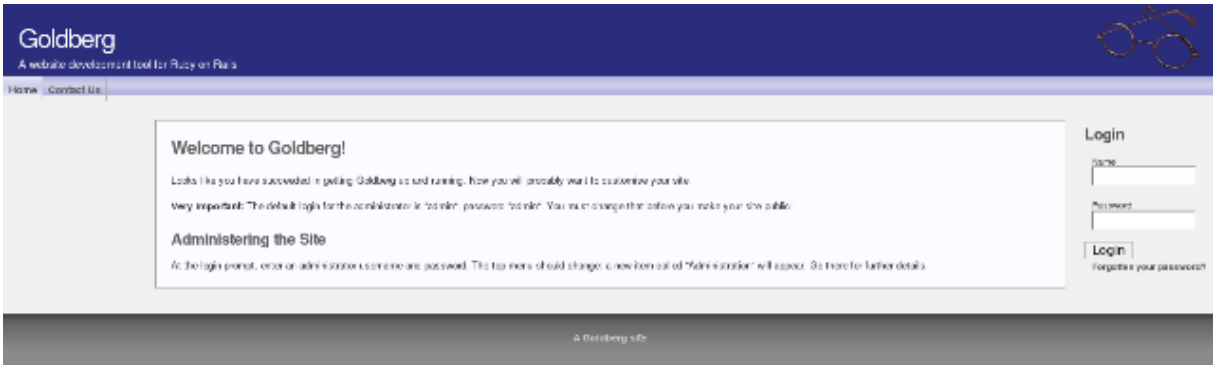


Figura 2: Exemplo de projeto gerador pelo Goldberg

Esse gerador fornece além dos recursos citados anteriormente, uma interface de administração para gerenciamento do conteúdo do projeto, em que se pode adicionar e remover itens do menu, além de modificar a seqüência desses itens, criar usuários, adicionar permissões a eles e realizar a configuração interna do gerador. A interface de gerenciamento do gerador Goldberg pode ser analisada na Figura 3.

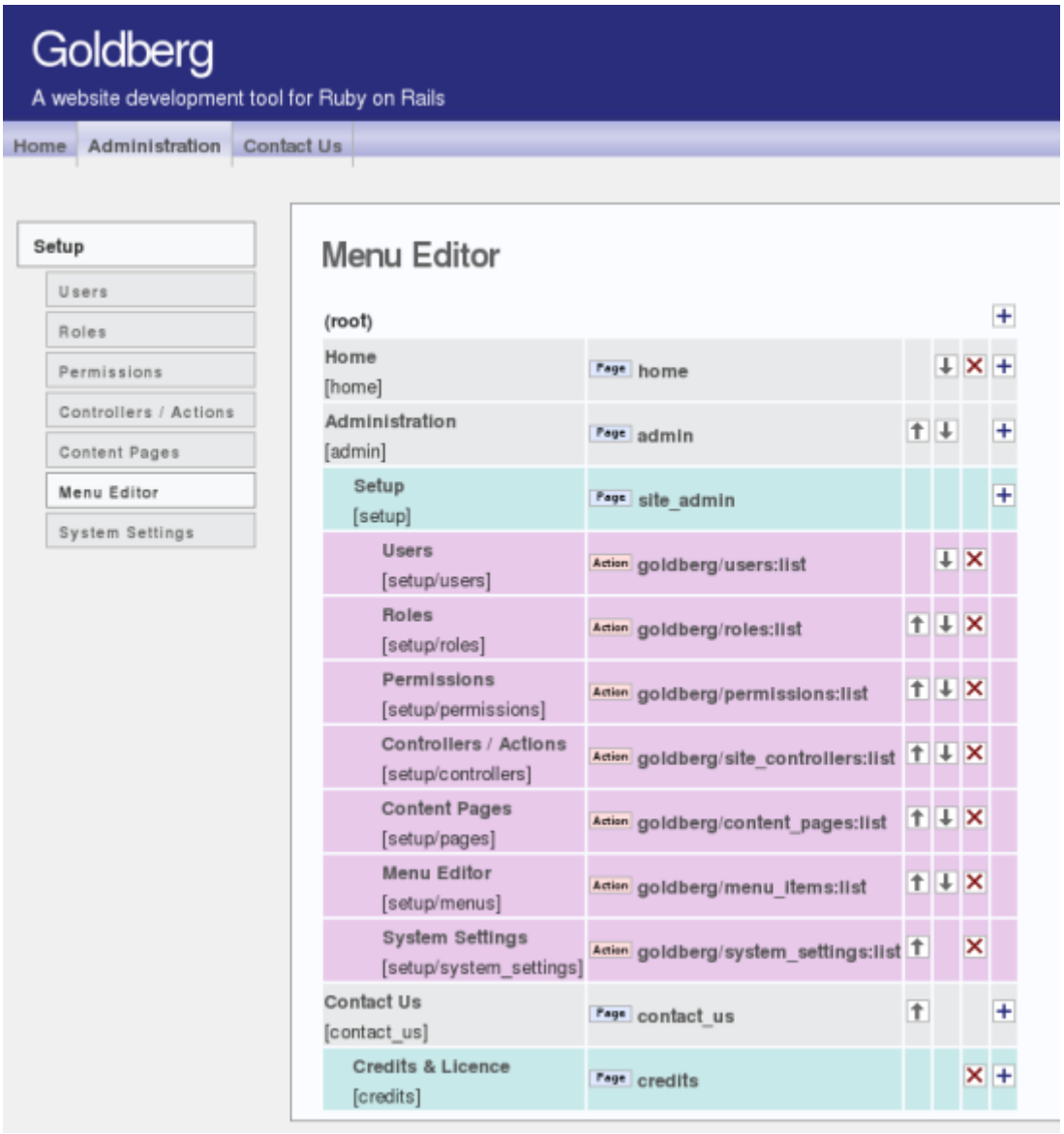


Figura 3: Interface de Gerenciamento do Goldberg

O menu (situado à esquerda) possibilita a criação de permissões e atribuição de autorias, assim como organização do conteúdo e geração de novas entidades. Esse gerador é bastante útil quando se trata de geração de aplicações para necessidades genéricas, mas quando há necessidade de um aplicativo para controle interno de uma empresa com detalhes particulares, esse gerador não possui flexibilidade para implementar tais detalhes facilmente. Outro fator é a contínua necessidade de construir entidade por entidade, seguindo na contramão de tecnologias ágeis.

2.3 Banco de dados Relacional

Banco de dados relacional é uma coleção de dados persistentes que pode ser acessada via Sistema Gerenciador de Banco de Dados (SGBD) baseado no modelo relacional. Para eventuais manipulações nessas coleções utiliza-se a Linguagem de Consulta Estruturada (*Structured Query Language* - SQL).

2.3.1 Sistema de Banco de Dados Relacional

Segundo Korth, Silberschatz e Sudarshan (2006), um Sistema Gerenciador de Banco de Dados Relacional é composto por:

- usuários e sua interação: usuários navegantes (usuários comuns que integrarem com o banco de dados através que aplicações). Os programadores de aplicação (profissionais que desenvolvem as aplicações que utilizam o banco como forma de armazenamento de dados). E, por fim, os usuários especialistas (escrevem aplicações especializadas para manipulação do banco);
- Sistema Gerenciador de Banco de Dados: é uma coleção de dados relacionados e uma coleção de programas para acesso a estes dados. O SGBD proporciona um ambiente conveniente e eficiente para armazenamento e recuperação de informações. É dividido em componentes de processamento de consultas e gerenciador de memória;

- armazenamento em disco: estrutura de dados que são necessárias como parte da implementação física do sistema (arquivos, dicionários, índices e estatísticas de dados).

Essa composição é necessária para o funcionamento do sistema de banco de dados como um todo, incluindo desde o pessoal que desenvolve softwares que utilizam banco quanto os utilizadores desse tipo de softwares.

2.3.2 Modelo Entidade-Relacionamento

São modelos semânticos ou conceituais. Se prestam à etapa de projeto conceitual, pois nessa etapa a base de dados é concebida de maneira bastante preliminar (a base não está associada a nenhum SGBD ou plataforma). Cada construção tem semântica bem definida, pois possuem notações diagramáticas de fácil entendimento.

O modelo entidade-relacionamento tem por base a percepção de que o mundo real é formado por um conjunto de objetos chamado de entidades e pelos conjuntos de relacionamentos entre esses objetos. A extensão deste modelo permitiu a inclusão de conceitos, tais como: classes e heranças (KORTH; SILBERSCHATZ; SUDARSHAN, 2006).

Suas construções básicas são entidade (retângulos), relacionamentos (losangos) e atributos (ovais). Podem ser mais bem compreendida no exemplo da Figura 4.

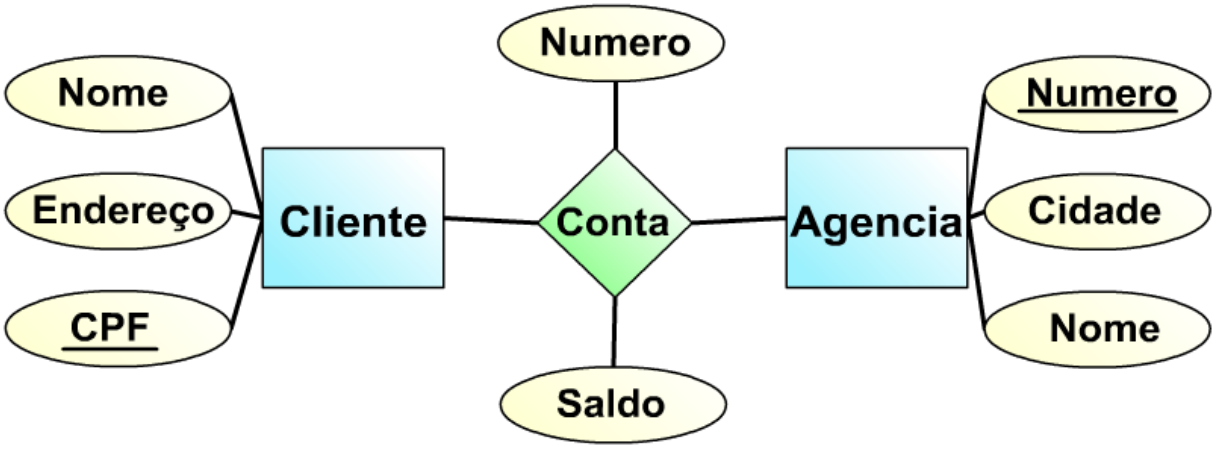


Figura 4: Exemplo de Diagrama Entidade Relacionamento

Nesse exemplo, temos como entidades Cliente e Agencia, representadas pelos retângulos azuis. A conta do cliente na agência representa o relacionamento entre as duas tabelas (losango verde). Todas as elipses amarelas representam atributos de cada entidade e relacionamento na qual estão ligadas.

2.4 Object-Relational Mapping

Mapeamento de Objeto-Relacional (ORM) é uma abordagem que permite a construção de sistemas utilizando o paradigma orientado a objetos com a persistência destes objetos em bancos de dados relacionais. Utilizando-se de técnicas e estratégias específicas, é possível mapear classes com seus atributos e associações para o modelo relacional (SILVA et al.; 2006).

Segundo (AMBLER, 1999), “o mapeamento de classes pode ser feito mediante a paridade entre classe e tabela, ou seja, uma classe é mapeada para uma tabela”. Este mapeamento direto de classes para tabelas representa a forma mais simples de mapeamento, tornando mais fácil o entendimento e a manutenção de uma aplicação. A idéia deste mapeamento pode ser visualizada na Figura 5.

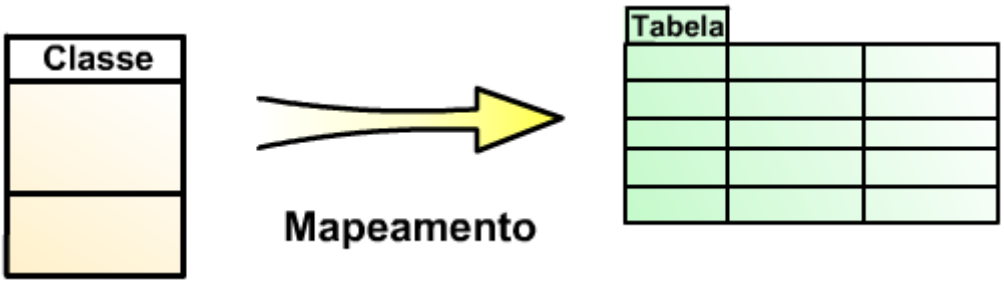


Figura 5: Mapeamento de Tabelas - Simples

Porém, nem sempre o mapeamento é tão simples assim. No caso de uma estrutura hierárquica, várias classes podem ser mapeadas para uma tabela, bem como uma classe pode ser mapeada para várias tabelas. Esse mapeamento mais complexo de classes e tabelas pode ser observado na Figura 6.

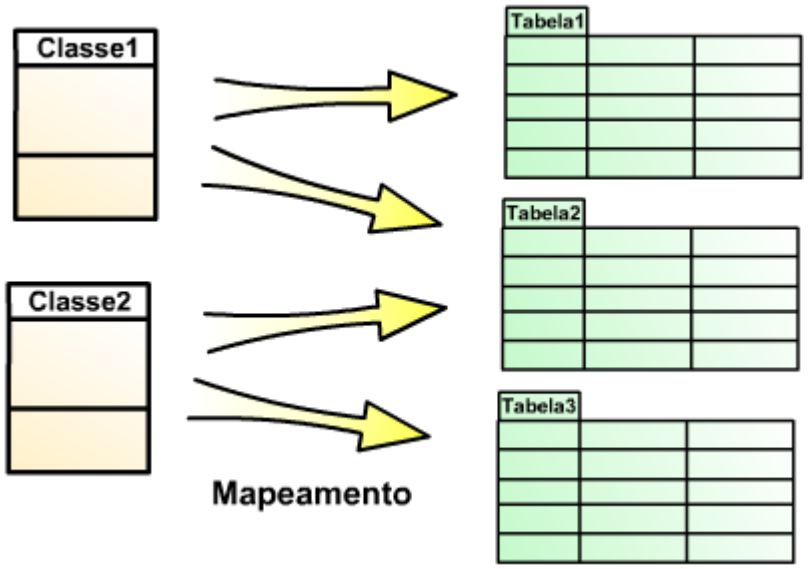


Figura 6: Mapeamento de Tabelas - Complexo

As tabelas Classe1 e Classe2 possuem, respectivamente, relacionamento com as tabelas Tabela1 e Tabela3. Esse mapeamento seria de certa forma trivial, e poderia ser tratado como relacionamento simples. Um terceiro relacionamento feito pelas duas tabelas impossibilita esse tratamento, pois ambas possuem atributos na Tabela2. Dessa forma é necessário que haja uma especificação de quais atributos pertencem a cada classe na Tabela2.

Ao tratar do mapeamento de atributos de uma classe para colunas em tabelas de um banco de dados relacional, deve-se levar em conta que os atributos podem ser de tipos de dados primitivos como inteiros, pontos flutuantes, caracteres, booleanos e binários, bem como ser de tipos de dados complexos como tipos baseados criados pelo usuário. Os atributos podem ser ainda multivalorados, o que viola as regras de normalização do modelo relacional.

Além disso, podem existir atributos de controle ou utilizados em cálculos, que geralmente não necessitam serem mapeados (AMBLER, 1999). Desta forma, os atributos simples podem ser mapeados diretamente para colunas em uma tabela, já os atributos complexos e multivalorados podem necessitar de tabelas adicionais para seu armazenamento. Estes atributos complexos, geralmente, possuem características recursivas, ou seja, são classes que possuem outros atributos e, assim, sucessivamente.

O mapeamento segue o seguinte conceito: as classes mapeiam cada uma das tabelas do banco de dados de modo que as linhas dessas tabelas se tornam objetos e as colunas referem-se aos atributos dessa classe. Esse conceito pode ser visualizado na Figura 7.

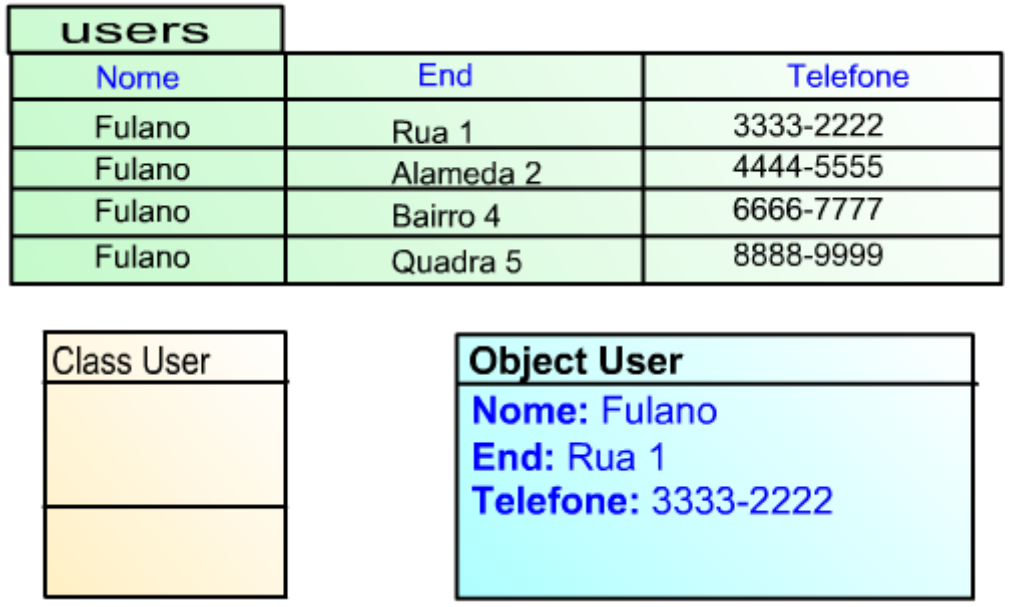


Figura 7: Mapeamento de Tabelas, Objetos e Atributos

Essa técnica possibilita mais do que códigos limpos, permite que persistências de objetos no banco de dados sejam feitas sem simples e transparente.

O ORM se comporta como uma camada que possui uma gama de métodos que cuidam dessa tarefa, tais como: *create* (cria um novo objeto da classe a partir dos dados da tabela), *find* (busca um determinado registro no banco e o torna um objeto da classe), *delete* (exclui registros do banco) e *update* (atualiza registros de uma tabela de acordo com as solicitações). Esses métodos variam de acordo com a linguagem utilizada.

Na Tabela 1, é possível observar um pouco das facilidades e de como a legibilidade do código fica melhor ao se utilizar o ORM ao invés de se injetar dentro de seu código o SQL puro, o que fere as boas práticas de programação, que sugere que códigos SQL estejam separados dos códigos de desenvolvimento (RODRIGES; COSTA; SILVEIR, 2001).

ORM	SQL
`Usuario.create(:nome=>'fulano', :endereço=>'rua 1', :telefone=>'3333 2222')`	`INSERT INTO usuarios ("nome", "endereco", "telefone") VALUES("fulano", "rua 1", "3333 2222")`
`Usuario.find(1)`	`SELECT * FROM usuarios WHERE usuarios."id" = 1)`
`Usuario.destroy(3)`	`DELETE FROM usuarios WHERE "id" = 2`
`Usuario.update(1, { :nome => 'Beltrano'})`	`UPDATE usuarios SET "nome" = 'beltrano' WHERE "id" = 1`

Tabela 1: Comparação entre ORM e SQL

Essa camada, chamada de persistência, encontra-se entre a camada de negócio (onde estão as classes de domínio da aplicação, ou seja, classes que definem as regras de negócio) e o banco de dados. Esta camada permite que o impacto das modificações em uma delas seja atenuado em relação à outra. Isto

diminui o grau de dependência do banco de dados e aumenta a facilidade de manutenção do código. Na Figura 8 pode-se analisar a disposição dessas camadas.

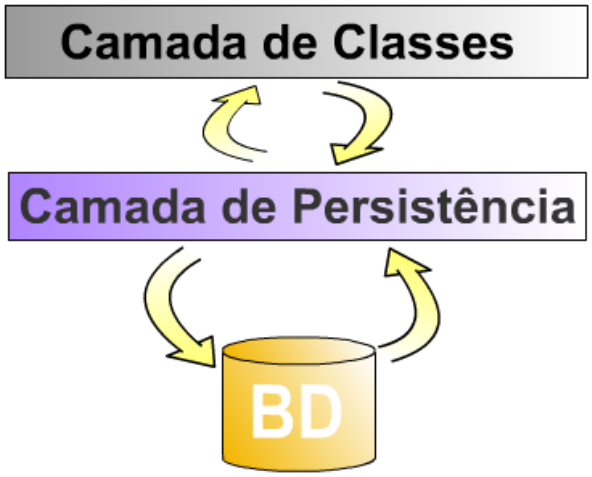


Figura 8: Camada de Persistência

Toda responsabilidade por persistir objetos fica a cargo da camada de persistência, liberando a aplicação destas tarefas, e assim aumentando a produtividade no desenvolvimento (YODER; JOHNSON; WILSON,1998). Na camada de persistência está a definição das estratégias de mapeamento do modelo orientado a objetos para o modelo relacional, apresentadas anteriormente.

A camada de classes representa todo o conjunto de classes (*controllers*, *views* dentre outras) que podem utilizar métodos da camada de persistência. Assim, as transações com o banco de dados ficam transparentes.

2.5 Ruby

Nesta seção será apresentada uma breve descrição da linguagem *Ruby*, incluindo a filosofia e o estilo de desenvolvimento. Serão apresentadas ainda informações introdutórias sobre o modo como foi idealizado, seu modelo de orientação a objetos, atributos e classes.

2.5.1 Filosofia

Ruby segue o *princípio da mínima surpresa* (*principle of least surprise* - POLS), significando que a linguagem se comporta intuitivamente ou como o programador imagina que deva ser. Essa caracterização não é originária de Matz (MATSUMOTO, 2008) e, em geral, *Ruby* seria algo mais próximo “da menor surpresa na cabeça Matz”. Contudo, muitos programadores também a acharam similar ao seu próprio modelo de pensamento.

Matz, de acordo com Venners (2003), explica que lhe motivou à criação do *Ruby* foi que:

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

O *Ruby*, de acordo com suas predefinições, proporciona ao programador uma codificação mais agradável e intuitiva, diminuindo a dificuldade de entendimento de código e agilizando o processo de desenvolvimento, com características como simplicidade, elegância e praticidade.

2.5.2 Semântica

Uma linguagem totalmente orientada a objeto, assim como *Ruby*, é definida por seu criador, isso significa que qualquer variável é um objeto, mesmo classes e tipos que em muitas outras linguagens são designadas como primitivos, ou seja, são

estruturas estáticas que servem como base para construção de outros tipos de dados (CUNHA NETO, 2007).

Toda função é um método, pois como tudo é classe, funções sempre serão métodos de alguma classe. Já as variáveis são referências para os objetos e não os objetos propriamente ditos. *Ruby* suporta herança, embora, não suporte herança múltipla, mas as classes podem importar módulos como *mixins* (módulos que possuem métodos que ao ser importados por uma classe, são incorporados ao seu escopo).

2.5.3 Orientação a Objeto

Como já foi mencionado, *Ruby* é totalmente orientado a objeto, similar ao *Smalltalk*, de modo que qualquer tipo de dado é uma classe e possui métodos. No Quadro 1 é apresentado um exemplo em que formata uma string

```
1 puts "carvalho".capitalize #Imprime Carvalho
```

Quadro 1: Capitalização de Strings

Nesse trecho de código, o método de capitalização (tornar a primeira letra da palavra maiúscula) de uma string é chamado, ou seja, não é necessário chamar algum método de uma classe e passar essa string como parâmetro. Logo, após mostramos o retorno do método com a função `puts`, que é o método de saída padrão do *Ruby*.

Isso significa que até mesmo uma string, que não foi instanciada explicitamente, em uma variável possui diversos métodos disponíveis a sua constituição. Em outras palavras, uma string já é um objeto de uma classe e, na Tabela 2, é possível verificar alguns métodos que instâncias da classe *String* possuem.

scan	slice	capitalize!	to_i	Each	Grep	instance_variable_get
reverse!	reverse	chop!	eql?	downcase!	Select	instance_of?
==	sub	length	tr_s	Downcase	find_all	instance_exec
each_byte	insert	each_line	unpack	Zip	any?	taint
=~	upcase	swapcase	match	Reject	between?	each_with_index
tr!	tr_s!	[]	index	Sort	Methods	instance_variables
squeeze!	>	include?	rstrip!	to_a	Freeze	__send__
inspect	next	chomp!	*	Find	Extend	protected_methods
chop	strip!	gsub	delete	Entries	nil?	singleton_methods
next!	squeeze	intern	gsub!	Map	object_id	frozen?
rstrip	dump	<<	to_s	Id	tainted?	respond_to?
casecmp	count	succ	rjust	member?	Method	instance_eval
split	sub!	capitalize	%	Max	is_a?	public_methods
succ!	hash	oct	>=	Min	__id__	untaint
clone	===	kind_of?	dup	Class	Type	

Tabela 2: Métodos da Classe String

2.5.4 Módulos

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

De um modo geral, existem duas principais funções dos módulos. A primeira função é de realizar o papel de *namespace* para outros métodos ou classes de modo a impedir a colisão de nomes, ou seja, faz com que nomes de módulos e classes criadas pelo programador entrem em conflito com nome de classes e módulos da linguagem.

No quadro 2, tem-se um módulo `Utilidades` com uma classe `String` e outro módulo chamado `Socket` que são palavras reservadas utilizadas no *Ruby*. Ao definir essas classes dentro de um módulo, o acesso a elas é, então, feito por intermédio do *namespace Utilidades* (`Utilidades::String.new()` e `Utilidades:Socket.new()`), assim distinguindo das classes *Ruby*, que são acessadas diretamente (`String.new()` e `Socket.new()`).

```ruby
module Utilidades

  class String
    def initialize
    end
  end

  module Socket
  def open; end
  def close; end
  end

end

```

Quadro 2: Módulos Ruby

A segunda função atribuída aos módulos é de simular heranças, em que uma classe ao invés de herdar métodos de uma classe mãe ela importa os módulos, que, nesse caso, se comportam como se fossem pacotes de métodos, constantes e variáveis disponíveis em seu escopo, de modo que eles pareçam ter sido definidos na própria classe importadora.

Esse módulo pode ser incluído por diversas classes, como pode ser visualizada a Figura 9. As classes `FileConv` e `FileReader` simulam, respectivamente, classes que convertem e lêem determinado texto, e para essa tarefa de manipulação de arquivos, incluem o modulo `FileManipulator` que possuem métodos de leitura, escrita e informações sobre esse tipo de dado.

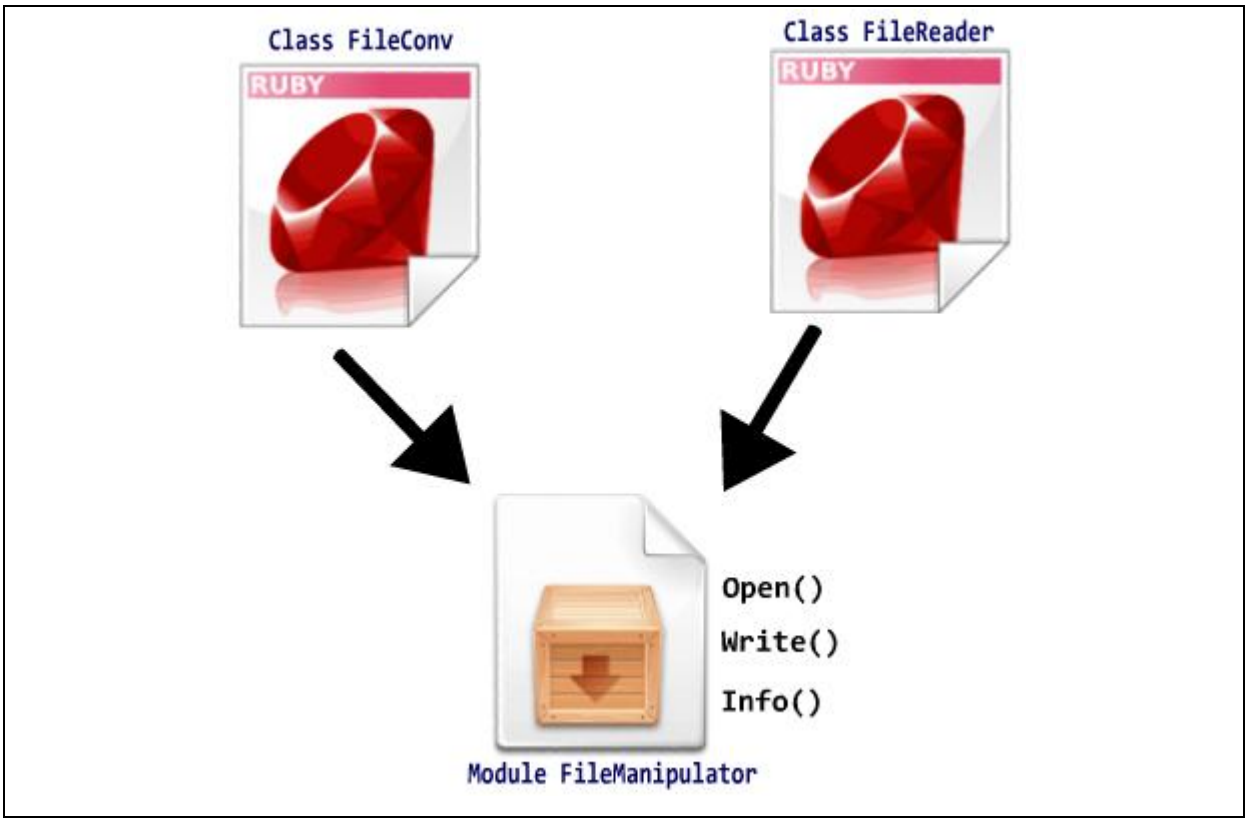


Figura 9: Importação de Módulos

2.6 Ruby on Rails

Esta seção aborda o *framework Ruby on Rails* ou, simplesmente, *Rails*. Serão apresentados os principais módulos que compõem o *framework*, assim como sua forma de utilização, acesso a banco de dados, estrutura de arquivos, configuração e convenções.

2.6.1 Rails, um framework ágil

O *Rails* é um *framework* que preza a agilidade no desenvolvimento do software, em que a programação deve ser simples e agradável, não precisando, necessariamente, ser dependente de algum tipo de metodologia para se tornar ágil.

Pelo simples fato dessa agilidade fazer parte da estrutura do *framework*, segundo Thomas e Hansson (2008), existem quatro princípios do manifesto ágil (BECK, 2001) que o *Rails* segue fielmente:

- indivíduos e iterações em vez de processos e ferramentas;
- softwares em funcionamento em vez de uma documentação abrangente;
- colaboração do cliente em vez de negociação de um contato; e
- respostas às mudanças em vez de seguir um plano.

O *Rails* foca indivíduos e iterações, não existem configurações complexas e processos demorados. Existem somente programadores, seus editores de texto favoritos e códigos *Ruby*. Essa simplicidade proporciona, em projetos que utilizam *Rails*, uma transparência no desenvolvimento, onde aquilo que o cliente vê reflete no que os desenvolvedores se propuseram a fazer.

Quando se fala em software em vez de documentação abrangente não significa que a documentação é discriminada. Ao contrário, o *Ruby* proporciona métodos simples de geração de documentação a partir da codificação realizada (a documentação é extraída principalmente de comentários). O desenvolvedor, então, pode se focar na funcionalidade do software e, então, gerar suas documentações em um documento HTML. *Rails* não se baseia em documentação, mas sim na comunidade, a sua principal fonte de informação.

Com esse ganho de agilidade, em pouco tempo, já é possível ter um projeto criado com funcionalidades básicas. Um protótipo do projeto, então pode ser apresentada ao cliente, sendo possível assim, dele ter uma idéia inicial do que está sendo desenvolvido. Dessa forma, o *Rails* encoraja a participação do cliente, que opina no restante do desenvolvimento, oferrecendo mudanças e melhorias.

Essa facilidade em responder a mudança é resultado de duas características do *Rails*. A primeira é sua maneira quase obsessiva de honrar o DRY (*Don't repeat yourself,* ato de não repetir código ou ações). Essa filosofia DRY prega que o

desenvolvedor repita o minimo de código possível em sua aplicação. Desse modo, mudanças em seus projetos têm impacto bem menor se comparado com outras estruturas. Por exemplo, se for necessário alterar a forma da busca de uma informação em um determinado método e este se encontra replicado em diversos arquivos, então o trabalho será bem maior que efetuar essa mudança em apenas um.

A segunda característica é que o desenvolvimento *Rails* é feito em *Ruby* em que conceitos podem ser expressos precisamente e concisamente, e as mudanças tendem a se restringir a um determinado local. Sendo assim, as mudanças são fáceis de localizar e modificar.

2.6.2 MVC

O MVC (*Model*, *View* e *Controller*) é um padrão de arquitetura de software, geralmente, usado em padrões de projeto de software, embora este conceito compreenda mais da arquitetura de uma aplicação do que um típico padrão de projeto (KLUNG, 2007).

Esse padrão foi, originalmente, projetado para atuar em aplicativos GUI (*Graphical User Interface ou Interface Gráfica do Usuário*) convencionais, em que os desenvolvedores descobriram que a separação de interesse resultava em muito menos acoplamento, o que, por sua vez, tornava o código mais fácil de escrever e de manter (THOMAS; HANSSON, 2008).

O *Ruby on Rails* é um *framework* MVC, o qual seus projetos são divididos nos três tipos de componentes: modelo, visão e controlador. Esses componentes são a base de toda a aplicação e onde é focado a maior parte do código e do esforço em um projeto.

O componente modelo impõe todas as regras de negócio que se aplicam aos dados e objetos da aplicação. Por exemplo, se um campo no banco de dados tiver um limite de 30 caracteres, o modelo impõe essa restrição, de modo que somente

dados com até este limite sejam persistidos no banco de dados. Esse tratamento faz sentido, pois garante que nenhum dado inválido proveniente de outro lugar da aplicação seja guardado.

A visão é responsável por gerar uma interface com o usuário, normalmente baseada em dados do modelo. Por exemplo, em uma loja os dados de todos os produtos cadastrados estão em um modelo, mas esses dados são apresentados ao usuário por meio de uma interface gerada pela visão. Para ações de persistência, a visão simplesmente se encarrega de enviar os dados capturados em sua interface e enviá-los para o modelo que, por sua vez, o guarda no banco caso seja válido.

O componente controlador, por sua vez, funciona como um gerenciador de requisições, recebendo eventos na maioria das vezes vindas do usuário, coordenando a criação de objetos e indicando as visões apropriadas para a requisição.

O sistema de MVC do *Rails* funciona de maneira organizada, em que cada componente espera ser requisitado por outro para entrar em ação. Na Figura 10, é possível analisar um esquema de como esse processo de requisição funciona.

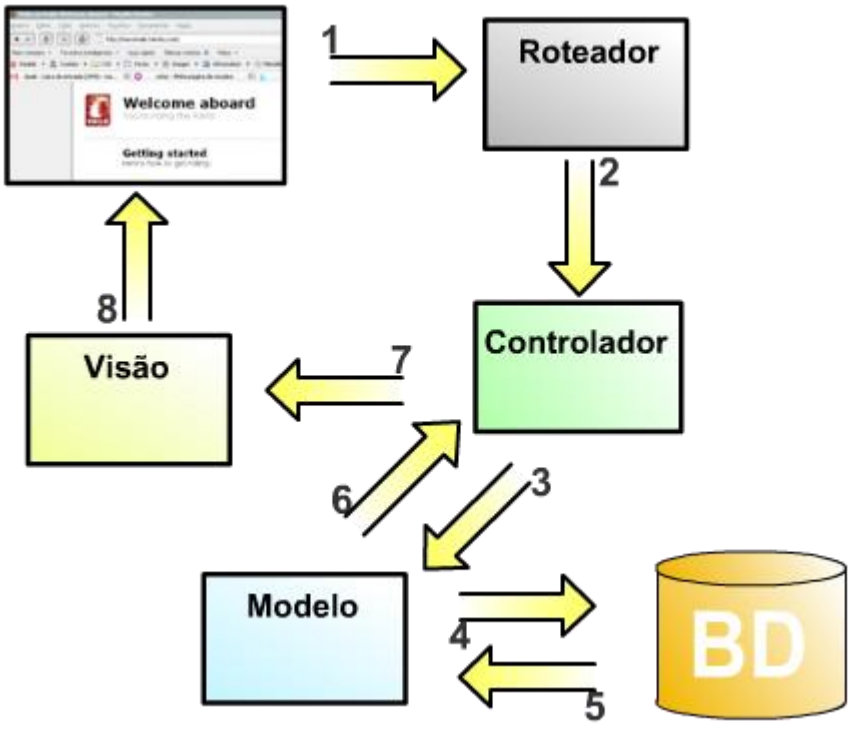


Figura 10: Fluxo de requisição no MVC do Rails

Ao executar um projeto *Rails*, as requisições enviadas ao aplicativo (1), primeiramente, passam por um roteador (2) que, então, determina para qual parte da aplicação a requisição deve ser encaminhada. A própria requisição deve ser analisada por meio de métodos existentes no controlador (3), chamado, na terminologia do *Rails*, de *ação*.

Essa *ação* então requisita de um modelo (4) algum dado necessário para completar sua função. Esse modelo, então, faz uma busca no banco de dados (5), cria um objeto com esses dados e devolve ao controlador (6). Esse objeto, por sua vez, é enviado à visão (7) que é então transformado de modo a ser apresentado ao usuário pelo navegador (8). Esse ciclo se repete a cada iteração (requisição) do usuário com a aplicação.

2.6.3 Active Record, o ORM do Rails

O *Active Record* é a camada de ORM que acompanha o *framework Ruby on Rails*. Este segue aproximadamente o modelo ORM padrão, em que tabelas são mapeadas para classes, linhas para objetos e colunas para atributos de objeto. O *Active Record* difere da maioria das outras bibliotecas ORM na maneira como é configurada. Contando com a convenção e iniciando com padrões sensatos, o Active Record minimiza o volume de configuração que os desenvolvedores realizam (THOMAS; HANSSON, 2008).

No Quadro 3, pode-se observar um exemplo da utilização desse ORM em uma classe *Ruby*.

```
1  require "active_record"
2
3  class Usuario < ActiveRecord
4
5  end
6
7  user = Usuario.new
8  user.nome = "fulano"
9  user.save
10
```

Quadro 3: Uso do ActiveRecord

Primeiramente, importamos a classe ActiveRecord com o método `require`. Logo em seguida, uma classe `Usuario` é criada a qual herda as funções do ORM. Isso é necessário para o uso de todas as facilidades presentes no Active Record. Na linha 7, um objeto `user` da classe `Usuario` é instanciado. Na seqüência, o nome “fulano” é atribuído a este objeto e salvo. É importante ressaltar que com somente este passos um novo registro na tabela `usuarios` é persistido.

O Active Record faz bem mais que isso, pois se integra, transparentemente, a todo o resto do *framework*. Active Record possui ainda um sofisticado sistema de validação de dados no modelo, de modo que se os dados não passarem por essa validação, as visões do *Rails* podem extrair e formatar esses erros de validação com apenas uma linha de código.

2.6.4 Análise do Framework

O *Rails*, além de ser o nome dado ao *framework*, é o nome dado ao componente responsável pelo gerenciamento de todos os outros módulos de alto nível que compõe o *framework*. Esses módulos são utilizados freqüentemente pelos programadores e constituem em *ActiveRecord* (componente de persistência de banco de dados), *ActiveController* (componente de controle de fluxo de requisições) e ActiveView (componente de interface).

Cada componente realiza uma função específica e independente dentro do *framework* e o *Rails* se responsabiliza que todos trabalhem de uma maneira conjunta e transparente para o desenvolvedor. Portanto, sem a presença do componente *Rails* quase nada aconteceria. Essa infra-estrutura será mais detalhada nos capítulos a seguir.

2.6.4.1 Estrutura de diretórios

O *Rails* exige certo padrão de layout em relação aos diretórios em seus projetos, para que funcione corretamente. A estrutura inicial desses diretórios pode ser encontrada na Figura 11.

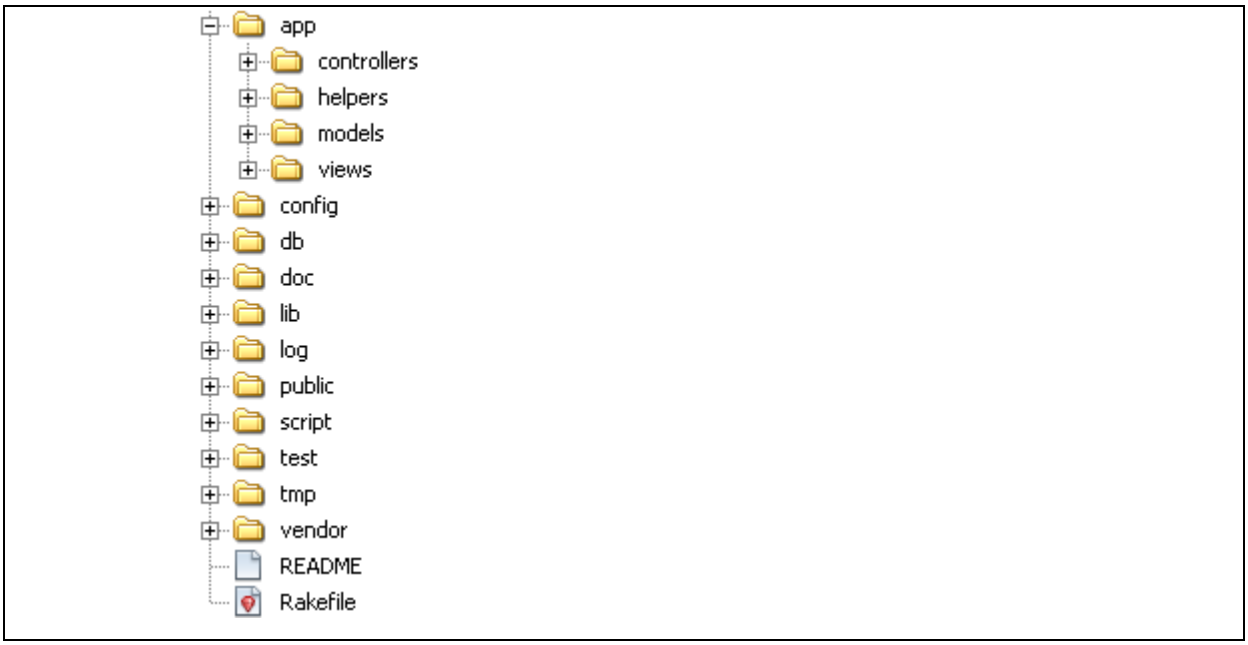


Figura 11: Estrutura de Diretórios de uma Aplicação Rails

No primeiro nível de uma aplicação, além dos diretórios, existem dois arquivos, que são: *README* e *Rakefile*. O arquivo *README* é responsável por informações sobre o projeto. O *Rakefile*, por sua vez, é usado para executar testes, criar documentos e extrair a estrutura do seu esquema de banco de dados. *Rakefile*, inclusive, é um arquivo de script, em *Ruby*, bastante utilizado para automatizações.

Com relação aos diretórios, tem-se o *app/,* que comporta a maior parte do código escrito da aplicação. Esses códigos são distribuídos em quatro outras pastas. A pasta *models/* contém classes de persistência de dados. A pasta *views/* tem arquivos que definem as interfaces da aplicação. A pasta *controllers/* possui arquivos de controle de requisições, que fazem a ligação entre as transações entre as v*iews/* e os *models/*. Além disso, tem-se a pasta *helpers/* que contém arquivos de suporte

para as *views* em que podem ser escritos códigos de formatação de dados ou métodos compartilhados por muitas views/.

Voltando a raiz da aplicação, existe a pasta *config*/ que possui grande importância na aplicação, pois nela se encontram o arquivo de configuração da conexão ao banco de dados, arquivo de rotas (possui uma lógica que gera todas as possíveis rotas (*urls*) da aplicação) e arquivo de inicialização, com métodos e variáveis que serão utilizados no desenvolvimento.

No diretório *db/*, as migrações, arquivos utilizados para controle de versão do banco de dados são encontradas. O diretório *doc/* é destinado às documentações geradas para a aplicação, que são produzidas por um módulo chamado *Rdoc* (*Ruby Document*) que gera diversos arquivos HTML com base na aplicação. Os códigos com métodos e classes para fins diversos (como métodos utilizados por várias classes ou que não se enquadram na estrutura de um controller ou model) são colocados na pasta *lib/,* assim como a pasta *vendor/*, mas essa última é reservada para *plugins* e *gems*.

Durante a execução de uma aplicação *Rails*, uma série de informações como SQLs, tempos de execução e erros, por exemplo, são armazenadas em arquivos de logs, que se concentram na pasta *log/* do projeto. O diretório *public/* da aplicação é a face externa de modo que o servidor WEB vê esse diretório como a base do aplicativo. Esse diretório possui, inclusive, arquivos que são acessados diretamente como arquivos de CSS, JavaScripts e imagens. Em *script/* existem programas de auxílio e automatização de tarefas. Dentre estes programas, destacam-se:

- about – exibe a versão do *Ruby* do *Rails* além de outras informações sobre a configuração da aplicação;
- console – inicia um console que pode ser utilizado para realização de testes na aplicação;
- generate – suporte a geração de código. Por padrão, cria controllers, models, mailers, scaffolds e web services;
- destroy – remove arquivos auto-gerados criados pelo *generate*; e

- server – executa sua aplicação *Rails* em um servidor integrado.

Arquivos como cachê de páginas e outros arquivos temporários são depositados na pasta *tmp/*.

2.6.4.2 Convenção para nomenclatura

A primeira vista o *Rails* causa confusão por ele tratar automaticamente atribuições de nome. Por exemplo, ao definir um nome de uma classe, essa é automaticamente interligada a uma tabela no banco sem necessitar a realização de nenhum tipo de configuração manual.

No *Rails*, por convenção, nomes de variáveis possuem letras minúsculas e palavras separadas por sublinhado (ex: *preco_do_produto*). Nomes de classes e módulos não possuem sublinhado, ao invés disso cada palavra, incluindo a primeira, inicia-se com uma letra maiúscula (ex: *GeradorRelatorio*). A nomenclatura de tabelas do banco de dados e de arquivos segue de maneira similar às variáveis, sendo letras minúsculas e palavras separadas por sublinhado (ex: *usuário_model.rb*). Porém, no caso das tabelas, os nomes precisam estar no plural (ex: *grupo_usuarios*) (THOMAS; HANSSON, 2008).

O *Rails* utiliza essas convenções para diminuir o processo de configuração convertendo nomes automaticamente. Por exemplo, se uma aplicação possui uma classe modelo (*Model*) que cuide de grupos de usuários, pela convenção a classe poderia ser chamada de *GrupoUsuario*. A partir disso, o *Rails* automaticamente deduziria que:

- A tabela correspondente a essa classe no banco de dados deve se chamar grupo_usuarios. Esse é o nome da classe convertido em letras minúsculas, com palavras separadas por sublinhado e no plural;
- E o arquivo modelo referente a essa classe deve se chamar grupo_usuario.rb, seguindo as normas de nomenclatura de arquivos.

Essas nomenclaturas podem ser mais bem visualizadas na Tabela 3.

Atribuição de nomes a modelos	
Tabela	grupo_usuarios
Arquivo	App/models/grupo_usuarios.rb
Classe	GrupoUsuario

Tabela 3: Atribuição de nomes a modelos

2.7 Código Aberto

Software com Código Livre, ou Software *Open Source*, são softwares que possui seu código fonte livre para que possa ser utilizado por qualquer um. Ás vezes há uma confusão entre software livre e software *open source*, pois os dois existem basicamente para o mesmo propósito, o compartilhamento de conhecimento pela distribuição de código (SANT'ANA, 2008).

Aplicações que se intitulam Softwares livres utilizam, comumente, licenças como GPL - *GNU General Public License* e LGPL - *GNU Lasser General Public License.* Por outro lado, softwares livres utilizam, entre outras, a MIT – Licença do *Massachusetts Institute of Technology* e BSD - *Berkeley Software Distribution.*

A diferença básica entre Softwares livres e *open source* é que no caso de códigos que utilizam licenças para softwares livres se agregam a essa licença de modo que futuras versões devem manter a mesma licensa do código ogininário. Da mesma forma, outros softwares que utilizarem parcialmente ou totalmente esse código, são obrigados a também utilizar a mesma licença. De acordo com seus idealizadores isso proporciona, ao software, sua contínua liberdade.

Diferentemente, projetos *open source* garantem a total liberdade do código, de modo que esse código pode ser utilizado para qualquer fim (livre ou proprietário), mudar de licença e até ser fechado em um projeto não *open source*. *Open source*

não significa apenas o acesso livre ao código fonte. Os termos de distribuição de software open-source, segundo Coar (2006), devem obedecer aos seguintes critérios:

- **redistribuição livre** - A licença não deve restringir qualquer das partes de vender ou retirar algum trecho e adicioná-lo a outro projeto. A licença não deve exigir nenhum tipo pagamento para sua utilização.
- **código fonte** - O programa deve incluir código fonte, e deve permitir a distribuição tanto do código fonte como forma compilada. Quando, de alguma forma, um produto não é distribuído com o código fonte, deve haver uma forma bem explicita de como o obter.
- **obras derivadas** - A licença deve permitir modificações e trabalhos derivados, e devem permitir-lhes ser distribuído sob os mesmos termos da licença do software original ou não;
- **integridade do autor do código fonte** - A licença pode restringir o código fonte de ser distribuído em uma forma modificada, apenas se a licença permitir a distribuição de arquivos patch (atualizações) com o código fonte para o propósito de modificar o programa no momento de sua construção.
- **nenhuma discriminação contra pessoas ou grupos** - A licença não deve discriminar qualquer pessoa ou grupo de pessoas;
- **nenhuma discriminação contra os campos de atividade** - A licença não deve restringir ninguém de fazer uso do programa em uma área de atividade específica. Por exemplo, ela não pode restringir o programa para que seja utilizado apenas para pesquisas genéticas;
- **distribuição de licença** - Os direitos ligados ao programa devem aplicar-se a todos a quem o programa é redistribuído sem a necessidade de realização de uma licença adicional por essas partes;
- **licenças não devem ser específicas de um produto** - Os direitos associados ao programa não devem depender que

- **Software Licença não deve restringir outro software** - A licença não deve colocar restrições quanto a outro software que é distribuído junto com o software licenciado. Por exemplo, a licença não deve insistir em que todos os outros programas distribuídos na mesma mídia devem ser de código livre;
- **licença deve ser tecnologicamente neutra** - Nenhuma das disposições da licença pode ser subordinada a qualquer indivíduo, tecnologia ou estilo de interface.

Todas essas exigências devem ser seguidas, além da disponibilização do código em um servidor, possibilitando o acesso de todos. Algo importante é a utilização de alguma ferramenta para controle de versão e apoio ao desenvolvimento comunitário simultâneo.

2.7.1 GIT

Git é um software gratuito para controle de versão distribuído. Em outras palavras, Git é uma ferramenta que auxilia no desenvolvimento conjunto um projeto, em que vários desenvolvedores podem fazer alterações em determinada parte do código e unir esses códigos em um lugar de maneira harmônica. Git foi criado por Linus Torvalds para o desenvolvimento do Kernel do Linux.

O Git foi desenvolvido como uma infra-estrutura para que outras ferramentas fossem criadas sobre ela. Ao invés de realizar o controle de versão de código por meio da indexação de arquivos, Git observa o conteúdo, independente do arquivo original.

Simplesmente fazer *branches* (Braços, no Git são cópias de um projeto que seguem um caminho diferente no desenvolvimento) é inútil se não for possível mesclá-los trivialmente. Com isso seria possível fazer uma cópia do projeto, realizar

alterações em sua estrutura e, facilmente, unir as modificações desse *branch* com as modificações do projeto original (AKITA, 2008).

Uma dos maiores atrativos do GIT é o fato de ser distribuído, ou seja, o projeto não, necessariamente, precisa estar centralizado em um único local. Esse repositório central é onde todos devem manipular os códigos, como é feito em outros softwares de controle de versão como, por exemplo, o *subversion*, ver Figura 12. cada estação de trabalho pega o código de um repositório central realiza suas alterações e devolve a sua origem, tornando essas modificações válidas.

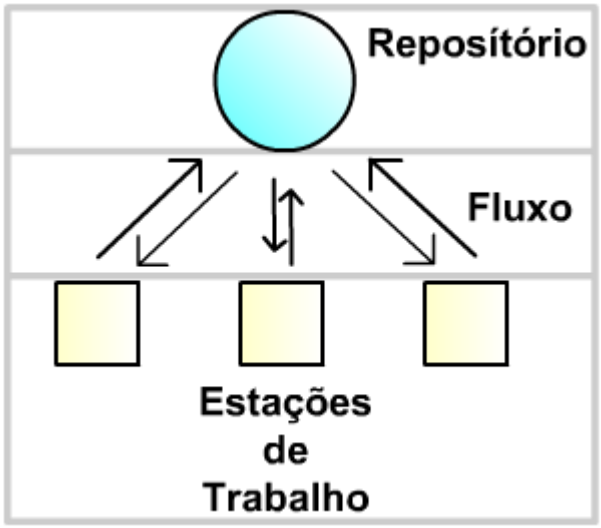


Figura 12: Controle de Versão com Software centralizado

O Git utiliza um sistema distribuído de colaboração, sua estrutura de distribuição pode ser vista na Figura 13, onde cada projeto que é clonado (“efetuar cópia de um projeto”) se torna um repositório. Permitindo assim que outras estações possam clonar, modificar e devolver os códigos sem necessitar acessar o repositório pai. Essa técnica permite ainda a realização modificações de código e ao invés de enviá-los ao repositório pai realizar o *versionamento* (controle de versão) localmente.

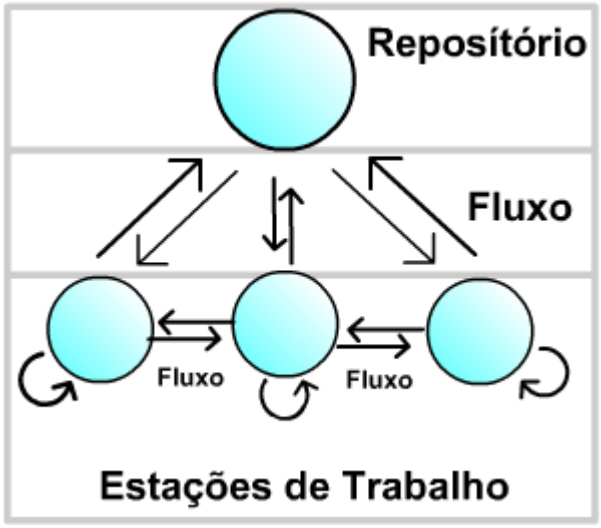


Figura 13: Versionamento Distribuído do Git

Esse tipo de técnica facilita a disseminação de código, sem a necessidade de permissões de escrita no repositório, eleito como principal. Pode se realizar todas as alterações, manter a cópia local atualizada e informar ao desenvolvedor do projeto principal das modificações realizadas. O desenvolvedor que coordena o desenvolvimento do projeto então pode clonar o projeto de algum desenvolvedor da rede e mesclar com o projeto principal.

2.7.1.1 GitHub

O GitHub é uma forma rápida e eficiente de controle de versão distribuído ideal para desenvolvimento colaborativo de software e não apenas um repositório on-line. GitHub proporciona um meio mais fácil de participar e colaborar com projetos, tendo como atrativos gráficos do projeto, envio e recebimento de solicitações de colaboração, acompanhamento do desenvolvimento, com notificações de maneira cômoda (GITHUB, 2008).

O GitHub proporciona de maneira simples a criação e distribuição de códigos. Disponibiliza hospedagem tanto para repositórios privados quanto públicos (apenas

para esse último existem planos sem custos). Possui sistema de autenticação via SSH (Secure Shell) e chaves criptografadas, o que propicia segurança aos projetos.

Sua interface é intuitiva, contendo a lista dos arquivos de seu projeto seguido de data de modificação e mensagem do último *commit* (ação de concretização das últimas ações realizadas no código). As mensagens sempre são acompanhadas pelo nome do desenvolvedor que enviou essas informações. Podemos visualizar a página inicial de um projeto hospedado no GitHub na Figura 14.

Figura 14: Página inicial de um projeto no GitHub

É possível também navegar pelos códigos, direto pelo navegador até algum arquivo. Um exemplo de um arquivo hospedado pode ser visto na Figura 15, onde é possível a visualização de informações como: o diretório atual (1), seu código com sintaxe destacada (2), o modo de escrita e leitura do arquivo (3), quantidade de linhas totais (4), quantidade de linhas, excluindo linhas em branco (5), tamanho em Kb (Kilo bytes) (6). Há ainda possibilidade de editar no próprio navegador trechos do código (7), visualizá-lo em modo texto puro (8) e ver todo o histórico desse arquivo desde sua criação (9).

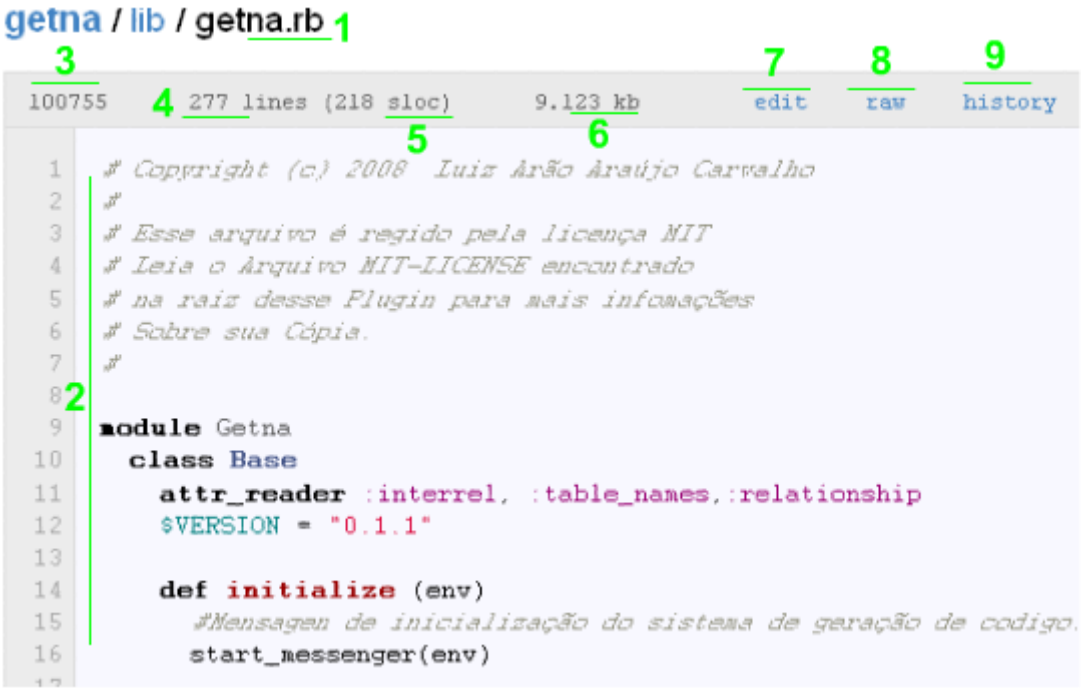
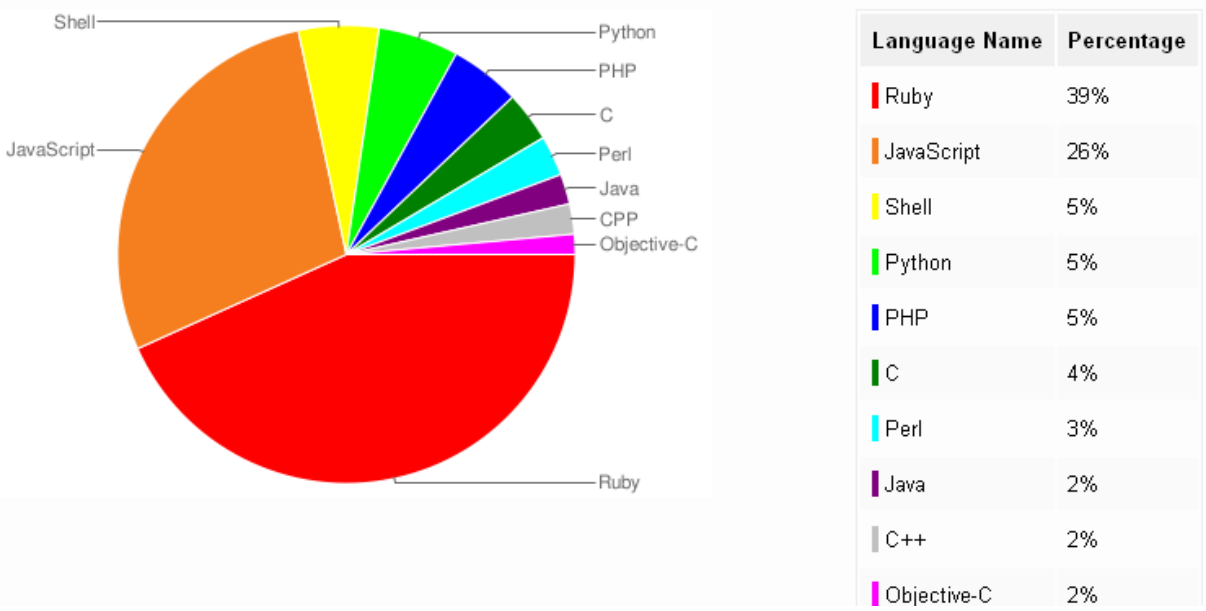


Figura 15: Opções e informações de arquivos no GitHub

O GitHub é um site de hospedagem de repositórios de códigos *Open Source* muito utilizado por diversas linguagens. É, atualmente, o repositório de código mais utilizado pelos *Rubystas* (desenvolvedores *Ruby*). Para se ter uma noção mais exata de tal dimensão, observe as porcentagens apresentadas no gráfico da Figura 16.



Language Name	Percentage
Ruby	39%
JavaScript	26%
Shell	5%
Python	5%
PHP	5%
C	4%
Perl	3%
Java	2%
C++	2%
Objective-C	2%

Figura 16: Linguagens mais utilizadas no GitHub (GITHUB LANGUAGES, 2008)

Esse repositório é tão bem aceito pelos desenvolvedores *Ruby* e *Ruby on Rails*, que o próprio *framework* tem seu núcleo de desenvolvimento no GitHub, o que mostra o grande potencial e poder do Git e do GitHub. O *Rails*, por exemplo, possui cerca de 350 (trezentos e cinqüenta) desenvolvedores trabalhando em seu projeto e mais de 2300 (dois mil e trezentos) desenvolvedores como espectadores do projeto. Esses espectadores comumente reportam falhas aos desenvolvedores do núcleo do *Rails* (conhecido como *Rails* Core).

2.7.2 Licenças

Ao se construir um código software deve-se atribuir algum tipo de licença a ele para garantir seus direitos sobre a obra. Há uma gama de opções quando se tratam de licenças para projetos *open source*, cada uma com suas características. Assim é importante o conhecimento de suas características para efetuar a escolha mais adequada para cada tipo de projeto. Realiza-se então uma abordagem sobre as mais utilizadas no mundo do código livre.

2.7.2.1 GNU General Public License

As licenças de software são normalmente desenvolvidas para restringir a liberdade de compartilhá-lo e modificá-lo. Pelo contrário, a *GNU General Public License* ou Licença Pública Geral GNU pretende garantir a liberdade de compartilhar e modificar o software livre, garantindo que o software será livre para os seus utilizadores. (GNU, 2007). É a licença mais utilizada em softwares livres.

Uma confusão comum, feita quando tratamos de softwares livre (*free software*) é que esse deve obrigatoriamente ser grátis. O termo, inglês, “free” acaba gerando essa confusão por possuir os dois significados, grátis e livre. Essa licença

se refere à liberdade e não ao preço. A GPL foi desenvolvida para garantir que cópias do software podem ser distribuídas livremente (cobrar por isso é opção do desenvolvedor), receber códigos fontes ou utilizá-lo como quiser.

2.7.2.2 GNU Lasser General Public License – LGPL

A GNU Lesser General Public License (LGPL, antes conhecida como GNU Library General Public License) é uma licença de software livre similar a GPL, mas, menos restritiva. A principal diferença entre as duas é que a LGPL permite que seus softwares sejam ligados com programas que não sejam GPL ou LGPL, que podem ser software livre ou Software proprietário. Geralmente acompanham bibliotecas utilizadas pela FSF (Free Software Fundation ou Fundação para o Software Livre, organização que gerencia licenças de software livre) em seus softwares.

2.7.2.3 BSD

A licença BSD (Berkeley Software Distribution) foi criada pela Universidade de Berkeley que desenvolveu o seu próprio sistema operacional baseado no Unix. Esta licença impõe poucas limitações, tendo como objetivo disponibilizar o desenvolvimento do software para a sociedade e ao mesmo tempo permitir que um financiador privado faça uso da pesquisa para fins proprietários. Qualquer pessoa pode alterar o programa e comercializá-lo como se fosse de sua autoria (BACIC, 2003).

2.7.2.4 MIT

A licença MIT criada pelo Instituto de Tecnologia de Massachusetts (Massachusetts Institute of Technology). É uma das licenças mais liberais que existe podendo ser modificada livremente pelo autor do software. Permite que o código seja utilizado em softwares licenciados em programas livres ou proprietários. Licença amplamente utilizada em projetos de código aberto como *Rails*, *PuTTy* e *X Windows*.

3 METODOLOGIA

Para a realização deste trabalho, várias etapas foram percorridas, iniciando com a definição do escopo e finalizando com os testes de desempenho do gerador GEtna. Assim, a seguir será apresentada uma breve descrição de cada uma das etapas.

3.1 Definição do Escopo

Esta fase foi sem dúvida a primeira etapa efetuada. Neste caso, foram definidas quais funcionalidades deveriam estar presentes no gerador. Foi analisado ainda onde e como deveriam ser aplicadas tais funcionalidades. Para isso, foram analisados alguns processos que deveriam ser otimizados em várias linguagens, sobretudo no *Rails*, que são:

- criação de arquivos

A cada nova entidade na aplicação, necessitamos criar toda a estrutura de arquivos e diretórios que o comportará. Se o sistema for relativamente grande, processos como esse tem um grande custo temporal para o projeto.

- replicação de dados entre classes e banco de dados

Ao se definir a estrutura de um banco de dados, atribuímos, paralelamente, nomes de variáveis, tipos de dados desse banco. Na criação de classes temos que novamente criar variáveis e definir os tipos de cada uma dessas. Se for considerado ainda que mais de uma classe utilize esses atributos, pode-se observar que haverá grande quantidade de replicação, um processo trabalhoso e dispensável.

3.2 Linguagem de Programação Escolhida

Com a especificação dos problemas a serem resolvidos, precisou-se definir como o gerador proposto deveria ser desenvolvido, bem como o foco da aplicação. A linguagem escolhida para a implementação foi o *Ruby*, por ser uma linguagem fácil de manipular e possuir uma vasta fonte de documentação. Vale a pena ressaltar o uso da IDE Netbeans para o desenvolvimento da aplicação.

Com a determinação da linguagem, definiu-se então que o gerador seria utilizado em projetos junto ao framework *Ruby on Rails*. Esse *framework* possui entre dezena de outras características que auxiliam o desenvolvimento do gerador, as seguintes:

- suporte a geração de código - O *Rails* facilita o desenvolvimento de geradores de código, por trabalhar amplamente com esse tipo de ferramentas. Sendo assim, possível utilizar métodos do *framework* para executar esse tipo de operação;
- componente ORM - Esse tipo de ferramenta auxilia, em alto grau, o desenvolvimento de um gerador que tem como entrada um banco de dados. Possui métodos de captura de dados avançados e de simples utilização;
- padrões - Por utilizar padrões a saída da geração se torna mais simples, necessitando apenas efetuar mudanças de acordo com a entidade a ser gerada para que esta se mantenha nos padrões exigidos; e
- suporte a módulos adicionais - Esses módulos podem estar em forma de *gems* ou de *plugins.* São módulos que adicionam funcionalidades extras ao *framework*. Sendo possível, assim, a criação do gerador como um *plugin* para o *Rails*.

Uma última escolha nessa fase foi a definição entre criar uma gem ou um plugin. Uma *gem* é um módulo que adiciona novas funcionalidades ao coração do *Ruby*, sendo expansível a qualquer outro módulo, ou *framework* nele existente,

conseqüentemente, disponíveis às aplicações *Rails*. A segunda possibilidade era um *plugin* (módulo de caráter expansível a uma aplicação *Rails*), que acoplado ao um projeto do *framework* pode ser utilizado em conjunto com a aplicação. Dentre as duas opções apresentadas, o plugin fixou-se mais apropriado verificando questões como facilidade de instalação na aplicação, utilização e desenvolvimento.

3.3 Mapeamento a partir do Banco de Dados

Uma das tarefas mais importantes foi encontrar uma forma de mapeamento do banco de dados, pois dependendo da forma de captura, utilizando (um *plugin*, comandos SQL ou um recurso do próprio *Rails*) influenciaria diretamente na qualidade final gerador.

O uso de comandos SQL manuais podem deixar o processo de geração mais lento e mais pesado. Com a utilização de *plugin* de terceiros perderíamos a portabilidade e simplicidade do gerador, pois haveria dependência direta de um agente externo, que necessitaria sempre ser instalado junto com o gerador. Uma possibilidade ainda pior, na utilização de outro *plugin* é na hipótese desse ser descontinuado fazendo assim com que nossa ferramenta se tornasse legada em uma possível atualização do *Rails*.

A melhor saída seria utilizar componentes já disponibilizados pelo *Ruby* ou pelo *Rails*. A solução encontrada foi a utilização do *ActiveRecord*, um das *Gems* que compõem o *framework* é responsável pela manipulação da base de dados.

Foram realizados estudos em relação à geração de código em outras linguagens e no *Rails*. Uma preocupação nessa etapa foi deixar o código gerado limpo, funcional e re-aproveitável, a resposta estava na utilização de padrões do próprio *Rails*. Com essa definição realizada, efetuaram-se a criação de *templates*, arquivos que contêm toda a lógica dos códigos a serem gerados.

3.4 Disponibilização do Gerador

Foi realizado um estudo sobre licenças de código livre para formalizar o gerador como um projeto oficialmente livre e disponível para toda a comunidade. A licença MIT (*Massachusetts Institute of Technology*), por exemplo, foi a escolhida por proporcionar maior flexibilidade em questão de mudanças na licença e no próprio código, e uma das mais utilizadas em projetos *Ruby on Rails*.

Uma questão levantada durante o desenvolvimento do gerador foi como o projeto iria ser distribuído para a comunidade. Uma ferramenta indicada foi o Github que utiliza GIT para controle de versão. Após um estudo sobre esse mecanismo foi identificado diversas qualidades como velocidade, grande aceitação e facilidade. O GEtna então foi disponibilizado para a comunidade utilizando o GIthub como hospedagem do projeto.

3.5 Testes

Foram realizados testes para verificação do desempenho do gerador. Um fator analisado foi a velocidade em que indica o tempo gasto de acordo com a quantidade de informação utilizada. Verificou-se ainda se os bancos de dados foram totalmente compatíveis com a ferramenta. Desempenho, gasto de memória e processamento para variação de banco, quantidade de informação e sistema operacional foram situações observadas também.

4 GETNA

GEtna é um *plugin* escrito em *Ruby* para o *framework Ruby on Rails*. Tem como principal propósito a otimização do processo de construção de aplicações *Rails* por meio de geração de código. Características gerais sobre o GEtna e seu processo de geração serão descritos com maiores detalhes nesse capítulo.

4.1 Objetivos

O principal objetivo do GEtna é proporcionar um meio de construção de aplicações rápido e com o menor esforço possível. A utilização desse gerador visa a maior produtividade, realizando o trabalho mecânico e deixando para o desenvolvedor apenas a fase criativa de desenvolvimento. O processo de geração foca a melhoria de qualidade do software final, pois inserem no software gerado padrões que aumentam a legibilidade do código.

Outro fator que a ferramenta se propõe é diminuir o tempo de início do projeto, pois procedimentos como a criação de arquivos demoravam um tempo razoável e estavam propensos às falhas humanas. Com a utilização do gerador, estas tarefas podem agora ser realizadas com uma maior rapidez e segurança.

O GEtna se propõe em realizar a criação de um sistema básico de manipulação de dados o CRUD (Create – Criação, Retrieve – Recuperar, Update – Atualizar e Destroy - Deletar). O projeto gerador proporciona um projeto totalmente funcional com especificações básicas de tratamento de dados, como validações, e relacionamentos entre entidades.

O propósito do gerador é proporcionar uma diminuição de até 30% no tempo de desenvolvimento total da aplicação e de até 90% na fase inicial de desenvolvimento. Podemos definir a fase inicial de desenvolvimento como a fase entre a modelagem e a criação das lógicas de negócio, ou seja, na criação estrutural

do projeto. Essa agilidade propicia um ganho significante na produção, diminuindo os custos da empresa.

4.2 Origem do nome

Uma característica importante em qualquer se seja o projeto é a impressão. A primeira característica observada é o nome. Esse deve ser fácil, original, bonito e possuir um significado lógico. A escolha do nome GEtna foi feita, inicialmente, entre vários outros nomes com base em mitologias. O gerador se chamava HefestoDB, nome formado pela junção de Hefesto (Deus grego do fogo e que residia no vulcão Etna) e DB (database). Esse primeiro nome foi atribuido fazendo analogia ao incrível poder dos vulcões de gerar, a partir de rochas magmáticas derretidas, um espetáculo de luz.

Essa nomenclatura se manteve até o início da construção da aplicação, quando se decidiu utilizar um nome mais ligado aos vulcões, aí que entrou o nome da morada do deus Hefesto, Etna. GEtna (lê-se *Guétina*) é a fusão da palavra “Gerador” com a palavra do “Etna”. Acompanhado da definição do nome surgiu também à logomarca do gerador. Pode ser visualizada na Figura 17.

Figura 17: Logo do GEtna

O símbolo do gerador foi criado com base na definição do nome, um vulcão negro com os trilhos vermelhos (símbolo do *Rails*) sendo expelidos, simbolizando a geração de aplicações *Rails*.

4.3 Requisitos e Instalação

Para o perfeito funcionamento do Gerador GEtna necessitamos de três principais requisitos: banco de dados nos padrões *Rails*, definição da origem dos dados e a correta instalação do gerador.

4.3.1 Banco de Dados Padronizado

O *Rails* propõe uma série de normas de nomenclatura em seus projetos. O GEtna se aproveita dessas normas para realizar seus procedimentos com o mínimo esforço do desenvolvedor. Utiliza a padronização dos bancos para identificação de relacionamentos e a padronização dos arquivos para transformar seus *templates* em classes válidas. A Figura 18 possui um exemplo de geração para alguns arquivos.

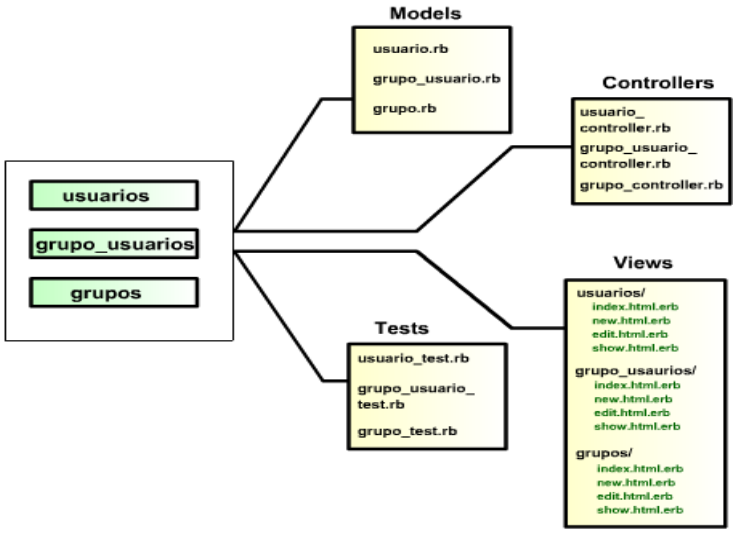


Figura 18: Exemplo de Geração de Arquivos utilizando nomenclatura padronizada.

Neste exemplo, foram criadas três entidades: usuarios, grupo_usuarios e grupos, um conjunto de arquivos com a nomenclatura padrão do *Rails*. Essa padronização garante o funcionamento instantâneo das entidades geradas, que são automaticamente identificadas pelo *framework*. Desse modo, não necessitamos nenhum tipo de configuração após a geração efetuada.

4.3.2 Origem das Informações para Geração

Em qualquer projeto *Rails*, existe um arquivo na pasta *config/* que chamada *database.yml*. Esse arquivo define os parâmetros de conexão que a aplicação *Rails* utilizará para o acesso às informações do banco de dados. Visando a facilidade e o menor esforço de configuração, além do reaproveitamento de informações (DRY), o GEtna utiliza esse mesmo arquivo para realizar suas consultas.

O arquivo *database.yml* é composto, inicialmente, por três ambientes que são utilizados para as três fases de construção de uma aplicação *Rails*. Um exemplo de como esse arquivo é estruturado pode ser visto no Quadro 4.

```
 1
 2 development:
 3   adapter: mysql
 4   database: getna_development
 5   username: root
 6   password:
 7   host: localhost
 8
 9 test:
10   adapter: mysql
11   database: getna_test
12   username: root
13   password:
14   host: localhost
15
16 production:
17   adapter: mysql
28   database: getna_production
29   username: root
30   password:
31   host: localhost
```

Quadro 4: Database.yml - Arquivo de configuração de conexão

Development é o ambiente de desenvolvimento. Utilizamos essas configurações no decorrer da codificação até uma nova fase da implementação. *Test* é o segundo ambiente. Essa fase é normalmente utilizada para a realização de testes na aplicação. *Production* se encaixa na terceira fase da construção de uma aplicação com *Rails*. Nessa última fase, a aplicação é considerada apta para ser repassada aos usuários finais.

O GEtna utiliza esse arquivo, assim como *Rails*, para se conectar com uma determinada base de dados. Um dos recursos oferecidos pelo gerador é a escolha de qual ambiente será retirado às informações para a geração. Essa escolha não está limitada aos três ambientes existentes na aplicação *Rails*, podendo se inserir um novo ambiente especialmente para a geração.

4.3.3 Instalação

Por ser um *plugin* nos padrões *Rails* e não possuindo qualquer dependência com outra ferramenta, a não ser o próprio *Rails*, sua instalação se torna bem trivial. O GEtna pode ser instalado, basicamente, de duas maneiras via GIT e manualmente. A maneira mais recomendada de se obter o gerador é via GIT bastando apenas se localizar dentro da pasta vendor/plugin/ dentro da aplicação *Rails*. Caso esteja no local certo e com GIT instalado na máquina basta utilizar o comando do Quadro 5.

```
git clone git://github.com/LuizCarvalho/getna.git
```

Quadro 5: Instalação via GIT

Esse comando, efetuado via console, pega a última versão do gerador no servidor do Github e o copia para a pasta local. Esse procedimento é o suficiente para que o gerador esteja totalmente funcional e pronto para ser utilizado.

A segunda maneira de instalar o GEtna é via download a partir do website do projeto no Github (http://github.com/LuizCarvalho/getna). A Figura 19 possui um fragmento da página onde se localiza o botão de download do projeto.

Figura 19: Download do GEtna

Após o arquivo baixado ele necessita ser descompactado e, logo após, renomeado para “getna”. Como no procedimento de instalação anterior o gerador necessita ser posicionado em no diretório referente aos *plugins* no projeto (vendor/plugin). Ao final de todo esse procedimento o gerador está pronto para uso.

4.4 Estrutura do gerador

A estruturação do desenvolvimento do gerador foi organizada em quatro partes fundamentais: Classe de Geração, Classe de Mapeamento, Templates e Arquivos de Projeto. Essa estrutura pode ser visualizada na Figura 20.

Primeira parte (área roxa) é o Motor de Geração ou Classe de Geração, encontrado como arquivo *getna_generator.rb,* é onde todo o processo de transformação dos *templates* em arquivos e classes da aplicação ocorre. Nele está definido onde cada arquivo deve se localizar qual nome esse deve ter. Essa classe é acionada quando executamos o comando de geração e a partir desse momento todos os outros objetos são instanciados dentro da classe. Para observar o conteúdo desse arquivo vide anexo B.

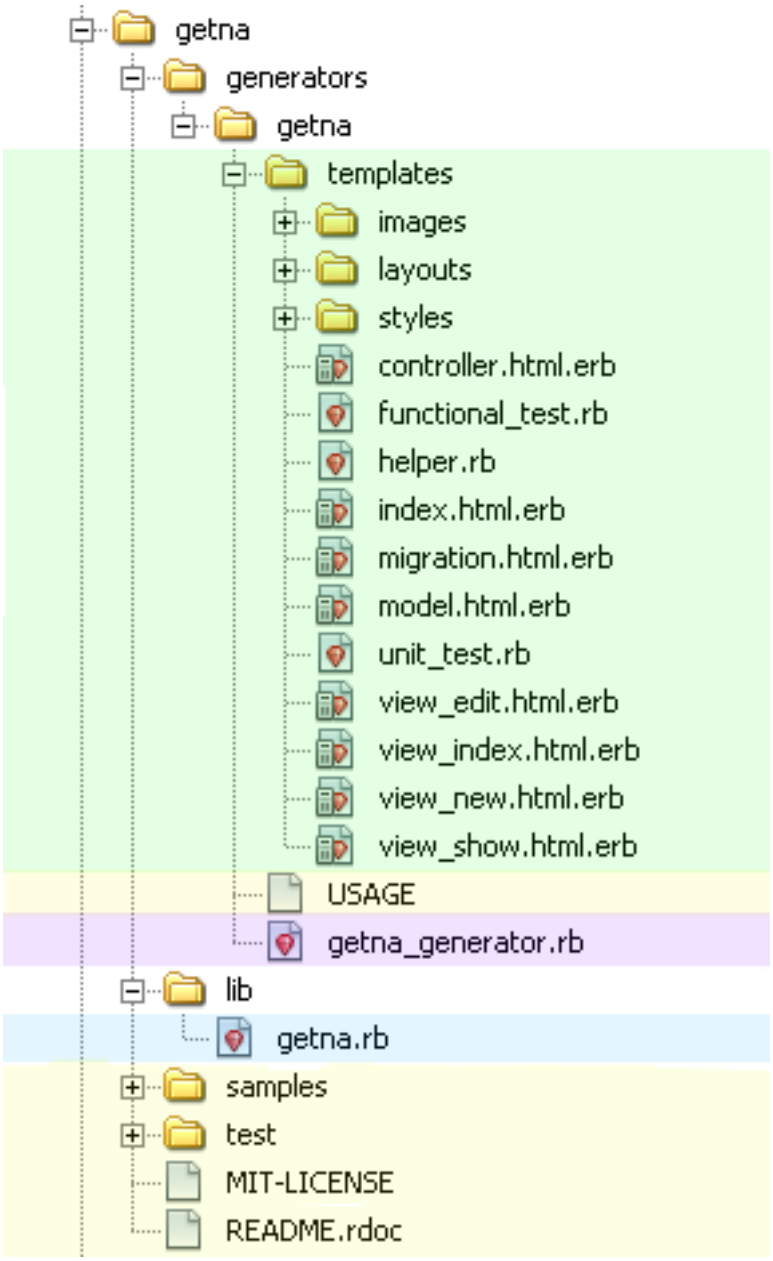


Figura 20: Estrutura do GEtna

A Classe de Geração cria objetos de nomeação de variáveis e métodos e objetos de propriedade (responsáveis por atribuição de relacionamentos, por exemplo), que são responsáveis pela alteração dinâmica de conteúdo dentro dos templates. O Código presente nessa classe se encontra no Anexo A.

A segunda parte, Classe de Mapeamento, realiza a tarefa de obtenção e tratamento de dados, a classe *GEtna* (área azul), localizada inclusive no arquivo

getna.rb, da pasta *lib/.* Contém todos os *módulos Ruby* necessários para a manipulação de dados. O mais importante é a base responsável por:

- efetuar conexão com bancos em ambientes diferentes;
- realizar o mapeamento de tabelas no banco;
- identificar relacionamentos;
- construir as validações;
- tratar e converter dados para interfaces; e
- construir migrações.

Na terceira parte (área verde) encontram-se os *templates*, arquivos modelos de cada arquivo que deverá ser gerado. Nessa parte que focamos a qualidade de código e a estruturação de cada arquivo, devendo ser simples, funcional e facilmente modificável. Contém os seguintes arquivos:

- controller.html.erb – arquivo base para geração de controladores;
- funcional_test.rb – arquivo modelo para criação de testes funcionais;
- helper.rb – modelo para criação de *helpers*;
- index.html.erb – arquivo base utilizado como pagina inicial da aplicação gerada;
- model.html.erb – *template* para criação de modelos;
- unit_test.rb – arquivo base para geração de teste unitário;
- view_edit.html.erb - modelo para geração do formulário de edição da entidade;
- view_index.html.erb – modelo para geração da visão index da entidade;

- view_new.html.erb – base para criação de arquivos da interface de criação de objetos da entidade;
- view_show.html.erb – modelo para a visão show da entidade;
- o diretório images/ contém imagens que são copiadas para a aplicação, por exemplo, a logo GEtna;
- layouts/ e styles/ comportam as definições de layout e css para as interfaces.

Toda a lógica da futura aplicação gerada é descrita utilizando esses arquivos. Os templates definem a estrutura interna dos arquivos, dos métodos e de todas as características necessárias para o bom funcionamento do projeto.

A quarta parte (área amarela) do gerador são arquivos complementares de cunho informativo e o diretório de testes. USAGE, arquivo encontrado no diretório generator/getna/, contém instruções de usabilidade do gerador e opções adicionais. No README encontram-se informações gerais sobre o projeto, modos de instalação e informações sobre o autor.

MIT-LICENSE possui a descrição da licença do projeto, no caso MIT, possui todos os termos de uso do código (em inglês). Por último, o diretório tests/ comporta arquivos de testes utilizados para “cobrir” o aplicativo contra erros.

4.5 Entrada

Como descrito nos capítulos anteriores, o GEtna necessita de um banco totalmente padronizado nas normas do *Rails*. Esta seção trata mais detalhadamente as regras que devem ser utilizadas para o correto funcionamento do gerador.

Uma das primeiras regras que deve ser aplicada é sobre a nomenclatura geral das tabelas. Os nomes das tabelas devem estar no plural (simbolizando um

conjunto de registros). Quando uma tabela representa um relacionamento entre outras duas, essa deve ser composta do nome das duas tabelas, em ordem alfabética, separadas por sublinhado e apenas a última no plural.

Em tabelas que representam relacionamentos, deve se aplicar uma segunda regra sobre chaves estrangeiras. Essa tabela deve possuir duas chaves, uma para cada tabela que ela relaciona, responsável pela relação direta entre essas tabelas. Essas chaves devem possuir o nome da tabela no singular acompanhada do sufixo "_id", tornando assim fácil a identificação. Todo esse esquema de regras aplicado na tabelas pode ser mais bem entendido na Figura 21.

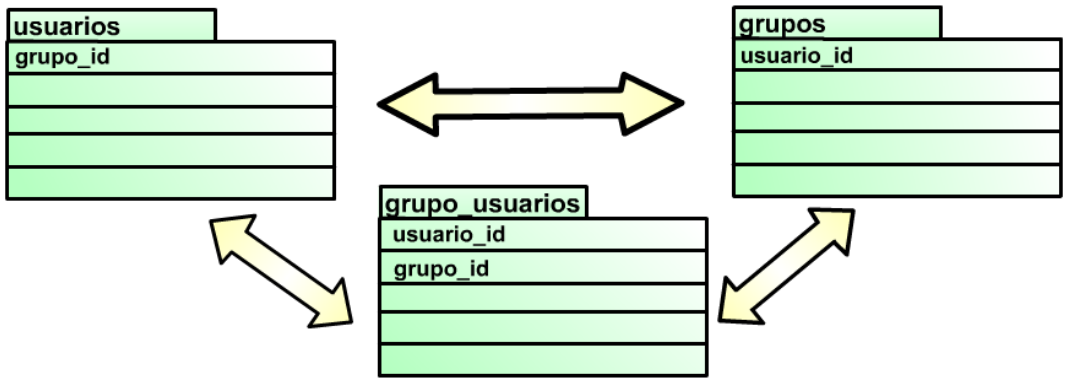


Figura 21: Nomenclatura de tabelas e chaves para relacionamento Muitos para Muitos

Nesse exemplo existe uma relação muitos para muitos entre as tabelas usuários e grupos. Desse relacionamento, surge a tabela grupo_usuarios. Essa terceira tabela simboliza o relacionamento entre as duas anteriores. Os dois atributos usuário_id e grupo_id são as chaves estrangeiras de usuários e grupos, respectivamente. Esse padrão é utilizado pelo ORM Active Record para mapeamento e reutilizado pelo GEtna para geração.

Quando o relacionamento é um para muitos, segue-se a mesma lógica de tabelas com relacionamento muitos para muitos. A diferença entre os dois relacionamentos é que ao invés das chaves estrangeiras estarem em uma tabela intermediária, ela se localiza na tabela que "possui" o relacionamento. Para melhor compreensão observe a Figura 22.

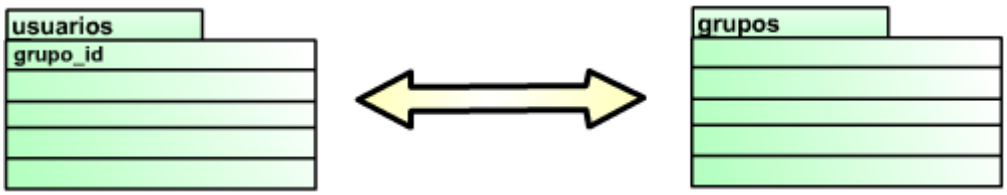


Figura 22: Nomenclatura de tabelas e chaves para relacionamento Um para Muitos

O GEtna partindo desses padrões consegue criar funções, que são inseridas nos modelos, para manipulação dos relacionamentos de forma simples. Da mesma maneira, se inserem nas visões de entidades relacionadas, métodos que permitem persistências de objetos relacionais.

4.6 Processo de Geração de Código

O GEtna é acionado via linha de comando. As instruções enviadas pelo console executam um *script Ruby* que, então, desempenham uma determinada tarefa. No gerador, além do comando padrão para geração (`script/generate`), são informados parâmetros que determinam como o gerador trabalhará. O GEtna possui dois tipos de parâmetros, que são:

- entrada - esse parâmetro é obrigatório e informa a origem das informações. Como descrito nos capítulos anteriores essa informação está localizada no arquivo *database.yml*. Deve-se então informar o ambiente que se deseja retirar os dados para realização da consulta.
- layout – informa-se o “tema” que a aplicação vai possuir. Esse tema é um conjunto de css e configurações de tags htmls que definem a aparência do projeto. Dentre as escolhas disponíveis temos: default (1), depot (2) e rails(3).

O comando para efetuar a geração é construído da seguinte maneira: `script/generate entrada layout:nome_do_layout`. Como visto no Quadro 4, a aplicação possui três ambientes padrões, sendo possível a criação de outro com dados de conexão para o gerador. Utilizando o ambiente development como entrada, um comando válido para a geração está apresentado no Quadro 6.

```
script/generate development layout:default
```

Quadro 6: Comando de geração GEtna

Após esse comando, o GEtna tem toda a informação necessária para iniciar o processo de geração. Realiza a conexão com o banco, busca todas as tabelas existentes e exclui as desnecessárias para a geração (como a schema, que é utilizada pelo *Rails* apenas para guardar o número da versão do banco de dados).

Com todas as tabelas carregadas, inicia-se o processo de geração em que é realizado, para cada tabela do banco, a geração dos seus respectivos arquivos. Em cada arquivo gerado, são passados dados utilizados para estruturação interna como atributos e métodos.

Da mesma forma que realizamos a geração de uma aplicação *Rails* utilizando o GEtna, podemos fazer o processo inverso, excluindo os arquivos criados pelo gerador. O comando necessário para efetuar esse processo de deleção dos arquivos é o seguinte: `script/destroy` entrada.

A diferença entre o comando de criação e o de deleção está na mudança do comando generate por *destroy* e da não necessidade de se passar o layout da geração. Por exemplo, para reverter a geração realizada no Quadro 6 devemos utilizar o comando mostrado no Quadro 7.

```
script/destroy development
```

Quadro 7: Comando para reverter processo de geração do GEtna

Esse comando realiza nova busca no banco, traz todas as tabelas e apaga cada arquivo referente. Por isso, salienta-se a importância de ser informado o mesmo ambiente utilizado na geração.

4.7 Saída

O GEtna produz dezenas de arquivos em cada geração. Podem-se destacar, entre os arquivos gerados, os seguintes: *Models*, *Controllers*, *Views*, *Templates*, *Helpers* e *Migrations*. Gera também routes, validations e relacionamentos.

4.7.1 Models

Os *Models* gerados representam os modelos da arquitetura MVC. São gerados de forma que cada modelo possui sua respectiva tabela no banco de dados. Quatro fatores devem estar presentes para que o arquivo gerado seja considerado um *Model* válido: localização, nomenclatura do arquivo, nomenclatura da classe e herança.

Todos os modelos devem estar localizados no diretório app/model/ para que o *Rails* os reconheça automaticamente. A nomenclatura do arquivo segue os padrões *Rails* assim como as classes. Todos os *Models* necessariamente devem herdar da classe *ActiveRecord::Base*. O Quadro 8 representa um *Model* válido.

```
1 class Usuario < ActiveRecord::Base
2 #Relacionamentos
3
4 has_many :grupos, :through=> :grupo_usuarios
5
6 has_many :grupo_usuarios
7
8 end
```

Quadro 8: Exemplo de Modelo Válido

Este exemplo representa o respectivo *Model* gerado para a entidade *usuarios* mostrada no relacionamento entre as tabelas (Figura 21). Esse relacionamento é descrito pelos métodos *has_many* (possui muitos) e pelo atributo *throught* (através de). Analisando o exemplo desta figura, grupo possui muitos *usuarios* através de *grupo_usuarios* e *usuario* possui muitos grupos através de *grupo_usuarios*. Para que este relacionamento funcione necessita-se ainda informar o *Model GrupoUsuarios* que ele faz parte desta associação. Para uma classe intermediaria a um relacionamento atribui-se dois métodos *belongs_to* (pertence a), cada um referente a uma das classes que esse associa. Um *Model* aplicado a esse exemplo pode ser visto no Quadro 9.

```
1 class GrupoUsuario < ActiveRecord::Base
2 #Relacionamentos
3
4 belongs_to :grupo
5
6 belongs_to :usuario
7
8 end
```

Quadro 9: Exemplo de Modelo Intermediário a um Relacionamento

Com essas definições corretas, é possível realizar persistências de relacionamentos de maneira simples e intuitiva. Algumas das ações possíveis podem ser visualizadas no Quadro 10.

```
 1 # As Variações u1,u2,u3,u4 são Objetos de Usuario
 2 # As Variações g1,g2,g3,g4 são Objetos de Grupo
 3
 4 u1.grupos # Devolve o Conjunto de grupos que esse usuario esta
associado.
 5
 6 g2.usuarios # Devolve o Conjunto de usuarioss que esse usuario esta
associado.
 7
 8 #Podemos Adicionar varios grupos diretamente em um usuario
 9 u1.grupos = [g1,g3]
10
11 #ou
12
13 #Podemos Adicionar varios usuarios a um grupo
14 g1.grupos = [u1,u2,u3]
```

Quadro 10: Métodos Relacionais

O código descrito mostra como se torna trivial o relacionamento entre *Models*. Para adicionar um ou mais usuários a um grupo necessita-se informar um conjunto de usuários a um objeto grupo. Esse método adiciona a relação na tabela *grupo_usuarios*, que gerencia as associações entre as duas entidades. É demonstrado graficamente o resultado obtido no Quadro 8, na visualização da Figura 23.

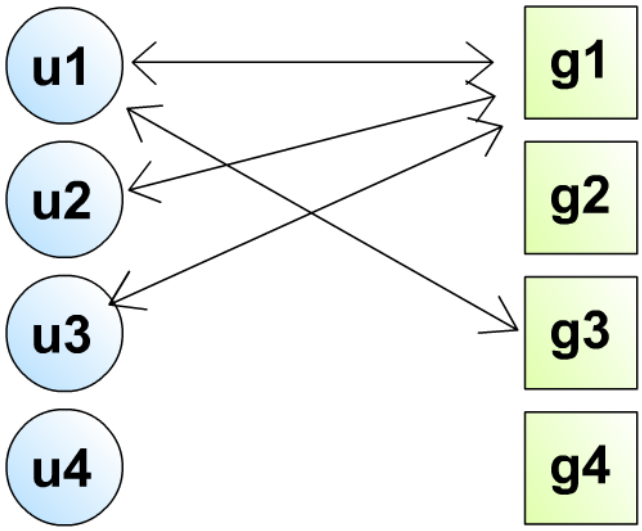


Figura 23: Representação Gráfica de Relacionamentos

De acordo com o exemplo, tem-se como resultados: o usuário `u1` está no grupo `g1` e `g3`, sendo que `g3` possui, unicamente, `u1` como membro. Assim como o grupo `g1` possui `u1`, `u2` e `u3` como membros e, respectivamente, `u2` e `u3` estão no grupo `g1`. O usuário `u4` não se encontra em nenhum grupo e o grupo `g4` está sem membros.

Para completar as gerações de *Models*, são criadas para cada entidade suas respectivas validações. Essas validações são métodos que verificam se determinado dado, antes de ser persistido no banco, possui características válidas para o campo que irá preencher. Existem três validações realizadas pelo gerador:

- presença – verifica se o dado a ser inserido pode estar em branco. Essa validação é criada, caso no banco de dados, o atributo esteja definido como não nulo;
- tamanho – define-se um tamanho máximo para o atributo de acordo com o limite suportado pelo atributo no banco de dados. Caso esse limite seja ultrapassado esse dado não é persistido; e
- numérico – onde dados que não possuam todos seus caracteres formados por números (0 a 9), não poderão ser armazenados no banco.

Essas validações são definidas por métodos no *Model* geradas a partir de dados do banco de dados. O Quadro 11 mostra o trecho de código de um *Model* com suas respectivas validações.

```
(...)
12 validates_presence_of :nome, :message=>"não pode ficar em
branco!"
13
14 validates_length_of  :nome, :maximum=>50, :message=>"não
pode exeder os 50 caracteres!"
15
16 validates_numericality_of  :populacao, :message=>"deve ser
numérico!"
(...)
```

Quadro 11: Validações de Um Model

Os métodos `validates_presence_of`, `validates_length_of` e `validates_numericality_of` são, respectivamente, os responsáveis pelas validações de presença, tamanho e numérico. Para validações de presença e numérico informamos a mensagem que deve ser lançada caso o valor não passe pela validação. No caso da validação de tamanho, informa-se o tamanho máximo que a entrada deve possuir, além da mensagem de falha.

4.7.2 Controllers

Localizados em *app/controllers/,* os *controllers* por ser de grande importância na organização geral da aplicação e por possuir a maior quantidade de código após uma geração, foram tratados com bastante atenção. *Controllers* são responsáveis por receber requisições e redirecioná-las para um próximo processo. Responsabilizam-se ainda por criação de objetos utilizados pelas entidades, sendo esses, da própria entidade ou alguma outra, por exemplo, necessário para um relacionamento.

Controllers Rails devem responder a requisições REST, ou seja, responder aos quatro verbos HTML (GET, POST, SET e DELETE). Cada verbo possui uma função. GET se encarrega da busca de informações, POST envia informações, SET realiza atualizações em determinada informação e DELETE efetua a função de exclusão de informações.

Para responder a esses verbos o *controller* utiliza *actions*, métodos que respondem a uma determinada requisição. O GEtna gera sete actions para cada *controller* de modo a suprir as necessidades REST para a criação do CRUD. Cada action possui uma função distinta mostrada a seguir.

- **Index** – *action* de visualização responsável pela criação de objetos com todos os registros da tabela. Executa requisição da exibição da *view index* com os dados do objeto recém criado. Responde ao verbo HTML, GET. Pode ser visualizado em modo HTML ou XML. A Figura 24 representa a view index repondendo em modo HTML.

Figura 24: Exemplo de Index em modo HTML

- **Show** – esse método mostra apenas um registro. Monta um objeto e solicita a exibição da *view show,* que expõe o conteúdo do objeto Reponde ao verbo HTML, GET acompanhado de um identificador que informa que registro deve ser mostrado. Do mesmo modo que a *index* esse método responde as requisições em HTML ou XML. Exemplo da *view show* em modo XML pode ser visualizada na Figura 25.

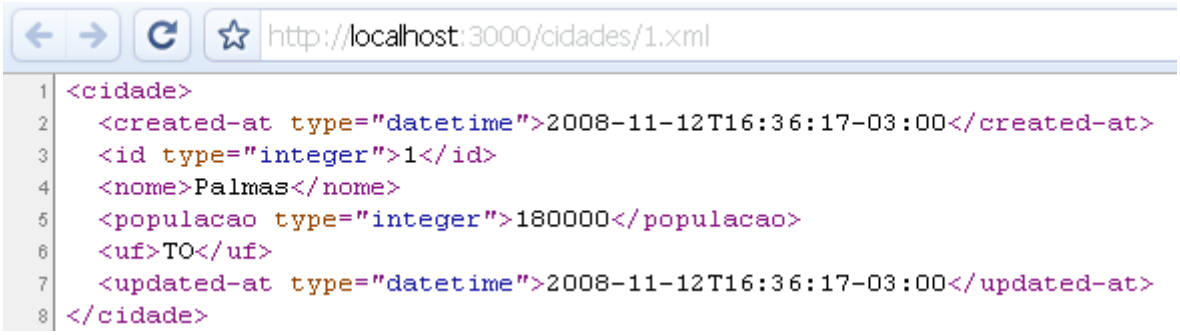


Figura 25: Exemplo de show em modo XML

- **New** – essa *action* instancia um novo objeto para aquela entidade e envia uma resposta informando que um formulário de criação (*view new*) deve ser mostrado. Caso exista relação entre a entidade gerada e alguma outra, nessa *action* carregam as informações pertinentes a

essa associação. Responde ao verbo HTML GET em modo HTML ou XML.

- **Create** – pode ser considerada a segunda fase de criação de um registro após o *new*. Esse método se encarrega de pegar as informações vindas do formulário de criação e então persisti-las no banco. Eventualmente ocorrem erros nessa persistência, esse método então cria um objeto com o erro, que é mostrado juntamente a um novo formulário de criação. Responde a um verbo HTML POST, pode ser apresentado como HTML ou XML;

- **Edit** – tem seu funcionamento similar ao da *action new*, mas ao invés de criar uma nova instancia do objeto ele busca um determinado registro para edição. Na hipótese da existência de uma relação realizam-se ações para carregar os dados dessa associação. Esse método então requisita a criação de um formulário de edição (*view edit*) para o objeto encontrado. Essa *action* responde a um verbo HTML GET acompanhado de um identificador que informa qual objeto deve ser buscado;

- **Update** – assim como o *create* pode ser considerado a segunda fase do *new*, o *update* é a ação subseqüente ao *edit*. Essa *action* recolhe as informações do formulário de edição (*view edit*) e então o armazena no banco caso sejam válidos. De maneira similar ao *create*, se dados não passarem pela validação um novo formulário é mostrado e os possíveis erros são indicados. Esse método responde a um verbo HTML PUT. Como os outros, podem ser representados por uma interface HTML ou por um objeto XML; e

- **Destroy** – função que especifica a exclusão de determinado objeto da base de dados. Reponde ao verbo HTML DELETE juntamente com um identificador que indica qual registro deve ser destruído. Também pode ser representados em modo HTML e XML.

Todo esse processo de requisição e resposta é definido por uma propriedade chamada *routes*. Essa propriedade é definida no arquivo *routes.rb*, localizado no diretório *config/* na raiz do projeto. O GEtna gera *routes resources* para cada entidade, tornando assim possível o fácil direcionamento de requisições REST para *actions* de algum *controller*. Na Tabela 4 podemos observar a relação entre os verbos HTMLs e as *Actions*. Essas *routes* geram também métodos em tempo de execução que são utilizados como *links* dinâmicos.

Verbos HTML	Actions
GET /usuarios	Index
GET /usuario/1	Show
GET /usuarios/new	New
POST /usuarios	Create
GET usuario/1	Edit
PUT usuario/1	Update
DELETE usuario/1	Destroy

Tabela 4: Relação entre Verbos HTML e Actions

O GEtna gera todas essas definições, livrando o desenvolvedor de se preocupar com esse tipo de tratamento inicial. A partir disso *Models, Controllers* e *Views* podem se comunicar de maneira fácil, em que *Controllers* podem facilmente instanciar objetos dos *models* e as *views* possuem todas as informações necessárias para criar suas interfaces.

4.7.3 Views

As *views* são interfaces de criação, edição e apresentação conjunta ou individual de registros. Consecutivamente, para essas quatro funções são criados

quatro arquivos para cada entidade gerada, que são: *new.html.erb*, *edit.html.erb*, *index.html.erb* e *show.html.erb*. Esses arquivos são gerados dentro do diretório *app/views/xxx/*, onde *xxx* é o nome da entidade que está sendo gerada no momento.

As Views são arquivos HTML com tags *Ruby* em seu corpo. Caso haja um situação de relacionamento as *views new* e *edit* alteram seu padrão de geração. Por exemplo, temos o caso que uma cidade possui várias localidades então ao se criar uma localidade, deve-se informar a qual cidade ela pertence. Tomando como exemplo a *view new* ocorre a troca de um campo *text field* (tag HTML que gera campos editáveis) por um *select* (tag que exibe uma lista de opções) com todas as cidades cadastradas. Um exemplo dessa conversão pode ser visualizado na Figura 26.

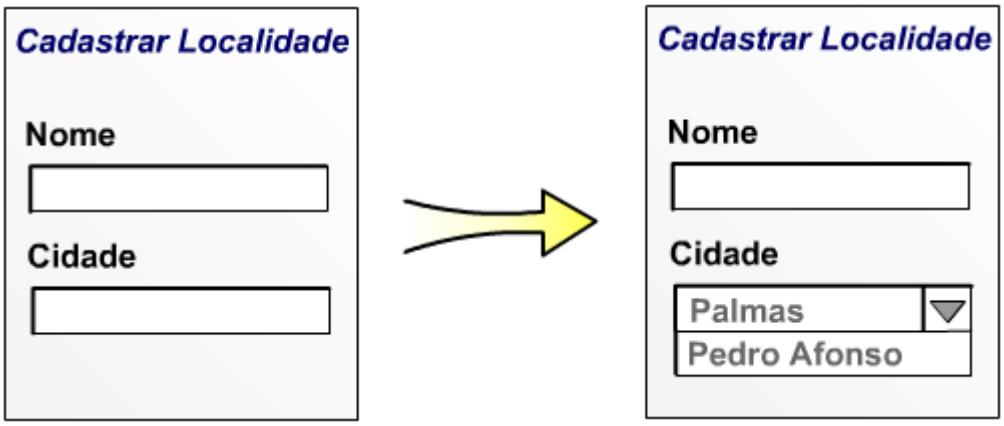


Figura 26: Conversão de interfaces com Relação

Cada uma das quatro *views* possui uma maneira particular de ser gerada, de modo que estas sejam eficazes em seu papel. Cada estrutura é abordada da seguinte maneira:

New – Formulário de criação de um registro. Deve possuir todos os campos editáveis do banco em seu corpo. Utiliza um objeto da entidade gerado na *action new* como base de sua estrutura. Após todos os dados inseridos no formulário e enviados a função da *view new* é englobar toda a informação no objeto da entidade e a enviá para a *action create* utilizando o verbo HTML PUT.

Edit – Após um registro encontrado pelo *controller* essa *view* destrincha o objeto retirando toda a informação necessária para criar um formulário de edição. Após as alterações realizadas e o formulário enviado, a *action update* recolhe as informações e efetua as alterações por meio de métodos do *Model* no banco de dados.

Show – Mostra as informações de um objeto. Esse objeto é requisitado via a requisição GET em conjunto com um identificador para a *action show* que se encarrega de criar um objeto, caso esse registro exista, e apresentá-lo por meio da *view show*.

Index – Essa view mostra todos os registros encontrados para aquela entidade. A *action index* realiza uma busca completa no banco retornando todos os registros de uma tabela em um único objeto, esse por sua vez é desmembrado na *view index*. Esse desmembramento é feito de modo que seja possível realizar todas as ações (criação, edição e exibição e deleção) possíveis para aquele registro. Um exemplo de uma *view index* pode ser vista na Figura 24.

4.7.4 Layouts

Arquivos de layout são utilizados para estilizar determinadas páginas. Trabalham de forma a segmentar a página de modo a facilitar o trabalho de design. O GEtna gera, para cada entidade, um arquivo *xxx_layout.html.erb* localizados em *app/view/layouts*, onde *xxx* é o nome da entidade.

Basicamente o arquivo layout gerado possui cabeçalho e rodapé do arquivo HTML que possuirá o conteúdo gerado pelas quatro *views* de cada entidade. No cabeçalho é definido o título e o arquivo CSS que será carregado para formatação da página.

O GEtna disponibiliza, por padrão, três tipos de layouts, podendo ser adicionado outros layouts sem grandes dificuldades. Cada um desses padrões

possui um *template* de geração e um arquivo CSS distinto que será utilizado de acordo com a entrada informada pelo usuário, essa podendo ser:

default – gerado pela opção `layout:default` ou quando não for informado o layout. É o mais simples dos layouts disponíveis, possuindo apenas definições de cabeçalho da página. O CSS, parte integrante do cabeçalho, acompanha esse layout e possui definições de posicionamento de texto, cor e destaque de elementos;

depot – esse layout possui coloração voltada a tonalidade verde. Exemplifica a possibilidade de posicionamento de elementos não só no topo e no rodapé da página, pois insere uma coluna vertical a esquerda da página. Esse layout pode ser usado informando-se a opção `layout:depot` no momento da geração. O CSS que acompanha esse Layout possui definições de posicionamento de texto de erros, informações, de colunas além de destaques de elementos;

rails – esse layout foi criado para combinar com o estilo da página inicial do projeto *Rails*. Com tonalidades cinza, é o layout mais moderno entre os três, possuindo contornos e a logo do gerador ao lado do título da página que pode ser substituído por qualquer outra imagem. Criado pela utilização da opção `layout:rails` cria todos os arquivos layout e copia o respectivo CSS. Um exemplo dos três Layouts pode ser visto na Figura 27.

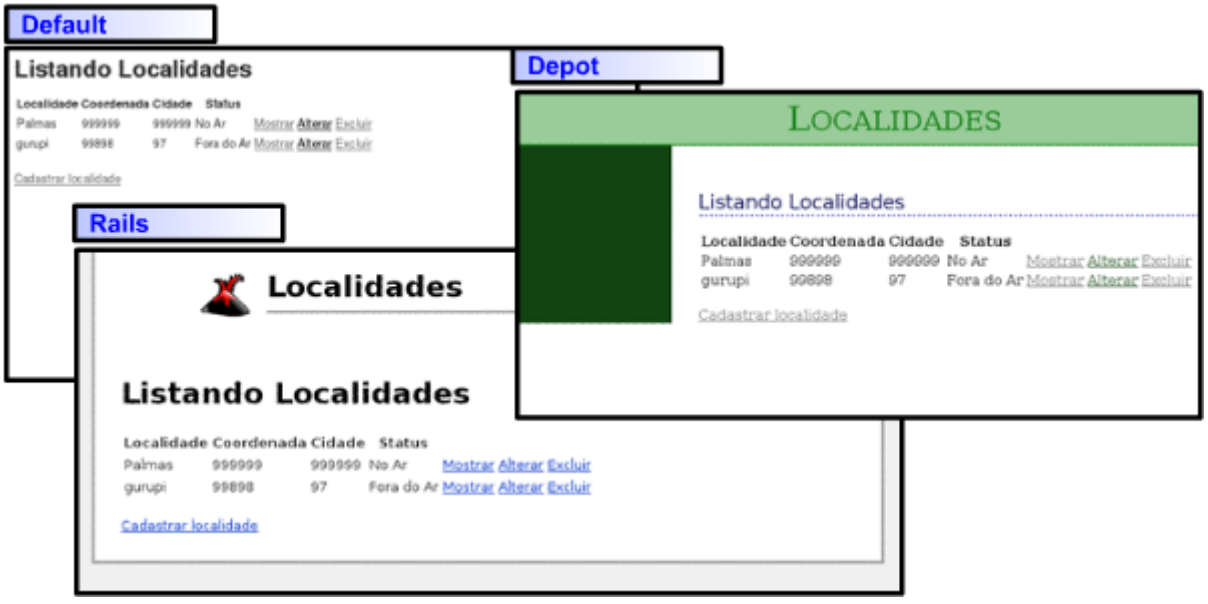


Figura 27: Exemplos de Layouts

Pode-se criar um único arquivo para todas as *views*, isso livraria o desenvovedor de bastante trabalho. O intuito do gerador no caso é possibilitar a fácil *refatoração* de código de modo que se for necessário alterar uma única página (por exemplo, uma página de administração) um único arquivo necessita ser alterado.

4.7.5 Helpers

Helpers servem como biblioteca para métodos adicionais para tratamento de dados nas *views*. O GEtna gera, para cada entidade, um modelo simples de *helpers*, no diretório *app/helpers/*, e caso o desenvolvedor queira utilizá-lo basta definir um método nesse arquivo para que as *views,* automaticamente, tenham acesso sobre ele.

Por exemplo, caso necessitássemos formatar um CPF de um usuário, sendo esse armazenado no banco sem nenhum caractere delimitador (“.” e “-“). Convencionalmente, colocá-se um método na própria interface para que esse dado seja tratado. A melhor solução no caso é a criação de um método que realize essa tarefa. No Quadro 12 podemos observar um exemplo de um *Helper* que desempenha esse papel.

```
1 module UsuariosHelper
2
3   def format_cpf(cpf)
4     cpf =~ /(\d{3})\.?(\d{3})\.?(\d{3})-?(\d{2})/
5     "#{$1}.#{$2}.#{$3}-#{$4}"
6   end
7
8 end
```

Quadro 12: Exemplo de Helper para formatação de CPF

Esse Módulo *Helper* é automaticamente invocado pelas *views* de usuários de modo que caso se deseje realizar a alteração de um CPF, bastaria utilizar o método *format_cpf* informando o CPF. Por exemplo, o número de CPF 11122233344, para

formatá-lo utiliza-se o seguinte método em qualquer das views, format_cpf(‘11122233344’) o resultado seria o cpf formatado (111.222.333-44).

4.7.6 Migrates

As migrações são arquivos que definem o banco de dados utilizado em uma aplicação *Rails*. São utilizados para criação de tabelas, atributos e suas definições. O GEtna gera *Migrates* de modo a espelhar o banco de dados utilizado na geração, permitindo assim que essa mesma aplicação seja utilizada em diversos bancos de dados sem nenhum prejuízo.

As Migrates geradas possuem dois métodos básicos, o de criação e o de destruição. O método de criação cria a tabela com todos os atributos e suas definições. O método de destruição elimina a tabela do banco de dados junto com suas informações.

O GEtna gera esses arquivos de modo que se possa haver um controle de versão do banco de dados. Para cada entidade gerada, se cria um arquivo Migrate com uma determinada versão que pode ser efetuada ou desfeita.

Por exemplo, temos três tabelas, usuarios, grupos e permissões para cada uma dessas tabelas cria-se, respectivamente, as migrates *001_create_usuarios.rb*, *002_create_grupos.rb*, *003_create_permissoes.rb*. cada arquivo migrate é acompanhado de um número (no exemplo, 001, 002 e 003) que representam a versão do banco de dados.

Se por ventura, for necessário que permissões sejam retiradas do banco realizamos um *downgrade* na versão do banco, ou seja, voltar para a versão 002 onde essa tabela ainda não foi criada. No Quadro 13 temos um exemplo de uma migrate para criação de contatos.

```
1  class CreateContatos < ActiveRecord::Migration
2    def self.up
3      create_table :contatos, :force => true do |t|
4
5      t.string :nome , :limit=>40
6      t.string :fone , :limit=>12
7      t.string :endereco , :limit=>200
8      t.string :celular , :limit=>12
9      t.timestamps
10
11     end
12   end
13
14   def self.down
15   drop_table :contatos
16   end
17 end
```

Quadro 13: Exemplo de Migrate

Temos definido nessa *migrate* o método `self.up` que define os dados de criação da tabela contatos. São definidos cinco outros métodos dentro deste, quatro `t.string` que definem o tipo de dado de *nome*, *fone*, *endereco* e celular como *string*. Para cada atributo é definido também o limite máximo de caracteres suportado. O quinto método, `t.timestemps`, se refere a dois atributos `created_at` e `updated_at` que definem, respectivamente, a data de criação e de modificação de um registro.

O segundo método da migrate é o `self.down`, que define as ações para uma regressão de versão. Neste caso acontece a deleção da tabela contatos do banco de dados. Todo esse esquema é gerado a partir de informações retiradas do banco de dados de entrada.

5 RESULTADOS

Esse capítulo apresenta os resultados obtidos durante e após o desenvolvimento do gerador. Muita informação foi gerada em meio dos mais de 11.085.954 de arquivos e 1.268.542 diretórios criados com quantidades de tabelas e versões do gerador diferentes durante testes realizados com o gerador. Será apresentada ainda uma síntese desses resultados para melhor entendimento do projeto, possíveis usos dessa ferramenta além de comparações com geradores existentes.

5.1 Uso

O GEtna pode ser utilizado em diversas tarefas, entre ela podemos destacar a criação de aplicações, a utilização em bancos de sistema legados e a migrações de bancos de dados. O uso convencional feito para essa ferramenta é a criação de aplicações do zero. Projeta-se o software, em seguida, se modela o banco de dados com todas as informações possíveis e a partir desse ponto o gerador se encarrega da geração de um protótipo funcional.

Ao expor o projeto para a comunidade muitos dos membros mostraram interesse em utilizar a ferramenta para sistemas legados de suas empresas. Sistemas implementados em *Rails* que se tornaram inviáveis de serem mantidos e necessitam ser reescritos. O GEtna então realiza de uma maneira rápida o processo de recriação da aplicação.

Outra finalidade acessível ao gerador é a conversão de banco de dados entre aplicações distintas. Por exemplo, no desenvolvimento de uma aplicação que utilizando o banco de dados Mysql e haja uma necessidade da alteração dessa base de dados para Postgres e arquivos de migração do *Rails* não estejam presentes na

aplicação. A utilização de GEtna torna trivial essa conversão e, automaticamente, a criação da aplicação para a nova base de dados.

5.2 Comparações

Para entender melhor a importância da criação de um projeto como GEtna, tendo conhecimento das várias ferramentas de geração já existentes, abordaremos nesta seção comparações entre o gerador desenvolvido e dois dos maiores geradores existentes para o *Rails*, que são: o Scaffold e o Goldberg.

5.2.1 Comparação com Scaffold

Como apresentado anteriormente, a geração por meio do Scaffold possui alguns contras como a geração de entidades individualmente. Ao se criar entidades uma a uma, corre-se um grande risco de falhas, além de ser uma tarefa extremamente trabalhosa. Na Figura 28 temos uma demonstração de como ocorre a geração de uma entidade utilizando-se o Scaffold. Informa-se a entidade, com suas respectivas informações ao Scaffold e ele então cria os arquivos necessários.

Esse processo se torna custoso tanto em relação a tempo quanto a gasto de esforço, ainda por cima sujeito à falhas senão tratado com atenção. Suponhamos um caso de dezenas de entidades em que o desenvolvedor terá que realizar essa tarefa para cada uma das entidades. Depois de gerada, ainda, necessita-se validar todos os campos (como o tamanho da senha do usuário), definir relacionamentos entre entidades e tratar migrações para que espelhem a nova base de dados a ser gerada. Quando se define as validações, como tamanho e presença, e uma nova base de dados for gerada a partir dessa, essa terá que possuir todas essas atribuições da antiga, esse é um processo que demanda muito tempo e cuidado.

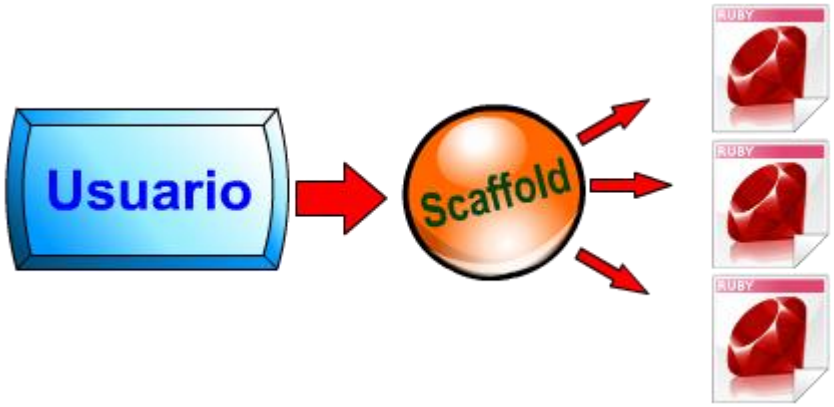


Figura 28: Geração com Scaffold

O GEtna sana a maior parte dessas deficiências, pois ao invés de se passar entidade por entidade, necessita-se informar o banco de dados, que será mapeado e, então, identificadas cada uma das entidades. A partir daí, o gerador realiza todo o processo de criação dos arquivos, como mostra a Figura 29.

Figura 29: Geração com GEtna

A parte de validação de dados também é feita automaticamente, junto com as validações, assim livrando o desenvolvedor de se preocupar com esse detalhe. As migrações e atribuição de validações dentro do banco de dados para replicação ou migração são feitas de acordo os dados do banco de entrada, o local mais correto de se retirar essas informações.

5.2.2 Comparação com Goldberg

O Goldberg permite a criação de uma aplicação completa, com direito a autenticação e área de administração. O custo desse tipo de aplicação é a falta de maleabilidade que isso gera, tornando complicada a modelagem de uma funcionalidade que não esteja presente no escopo do gerador. O GEtna, por sua vez, cria um sistema funcional e totalmente personalizável. Onde o desenvolvedor pode utilizar qualquer sistema de autenticação disponível para a *Rails*.

O gerador Goldberg, similarmente ao Scaffold, necessita que cada entidade seja criada individualmente. Esse gerador apenas garante um sistema de gerenciamento para essas entidades criadas. As classes criadas então se tornam dependentes desse sistema. No GEtna, após realizada a geração, as classes e os arquivos criados totalmente independente de qualquer plugin, inclusive o próprio GEtna.

Um dos fatores que mais pesam negativamente no Goldberg são suas dependências. Ao se instalar o Goldberg, é exigida a presença de outros *plugins* e *gems* para finalidades diversas. Essa dependência acaba amarrando o gerador de modo que se uma dessas dependências deixar de existir ou houver mudanças radicais, o gerador pode parar de funcionar.

O Gerador GEtna, por sua vez, foi desenvolvido de maneira que não necessite de nenhuma outra dependência, a não ser o próprio *Rails*. Essas vantagens aliadas a facilidade de geração e de manipulação do código gerado tornam o GEtna uma ferramenta diferente do Goldberg.

5.2.3 Comparações Gerais

Para melhor compreensão das diferenças entre o GEtna e os geradores Scaffold e Goldberg foram feitas comparações entre as funcionalidades presentes em cada um. Podemos ver essas comparações na Tabela 5.

Funcionalidades	*GEtna*	*Scaffold*	*Goldberg*
Geração múltipla	**Sim**	Não	Não
Geração do MVC	**Sim**	Sim	Sim
Geração de Helpers	**Sim**	Sim	Não
Geração de Migrations detalhadas	**Sim**	Não	Não
Dependência de terceiros	**Não**	Não	Sim
Validações	**Sim**	Não	Não
Rotas	**Sim**	Sim	Sim
Relacionamento	**Sim**	Não	Não
Sistema de Gerenciamento	**Não**	Não	Sim
Fácil Refatoração de Código gerado.	Sim	Sim	Não

Tabela 5: Comparação entre Geradores

5.3 Desempenho

Para medir a performance do gerador em relação ao seu consumo de memória, a CPU e o tempo levado para essa ferramenta efetuar sua função, foi realizado uma bateria de testes. Utilizou-se uma máquina com as seguintes configurações:

- processador: DualCore Intel Pentium D 925, 3000MHz

- placa Mae: GA-VM800PMC, 800MHz
- memória DDR2 1983MB (2GB)
- chipset: VIA P4M800 Pro

Foram realizados testes para os sistemas operacionais Windows XP Home Edition SP3 e Ubuntu 8.04.1 LTS (Hardy Heron), utilizando diferentes quantidades de tabelas no banco de dados. Na Tabela 6 podemos visualizar um resumo dos resultados obtidos nos testes de geração.

Sistema Operacional	Tamanho (Tabelas)	Tempo (segundos)	Consumo Memória	Consumo CPU
Windows	12	8,531000	20,0%	52,0%
Windows	60	15,766000	21,0%	54,0%
Windows	120	24,296000	24,0%	55,0%
Windows	600	80,750000	33,0%	55,0%
Windows	1200	215,360000	52,0%	57,0%
Ubuntu	12	3,257485	16,5%	56,3%
Ubuntu	60	6,963654	17,1%	47,8%
Ubuntu	120	10,606100	17,7%	42,1%
Ubuntu	600	42,034495	18,7%	39,4%
Ubuntu	1200	111,500886	19,9%	37,2%

Tabela 6: Tabela de Informações de Desempenho

Baseado nas informações obtidas, foi possível observar que a criação de gráficos que mostram as mudanças de valores com o aumento do número de tabelas no banco de dados de entrada. Na Figura 30 podemos visualizar a relação entre tempo e tamanho da entrada em relação a tabelas.

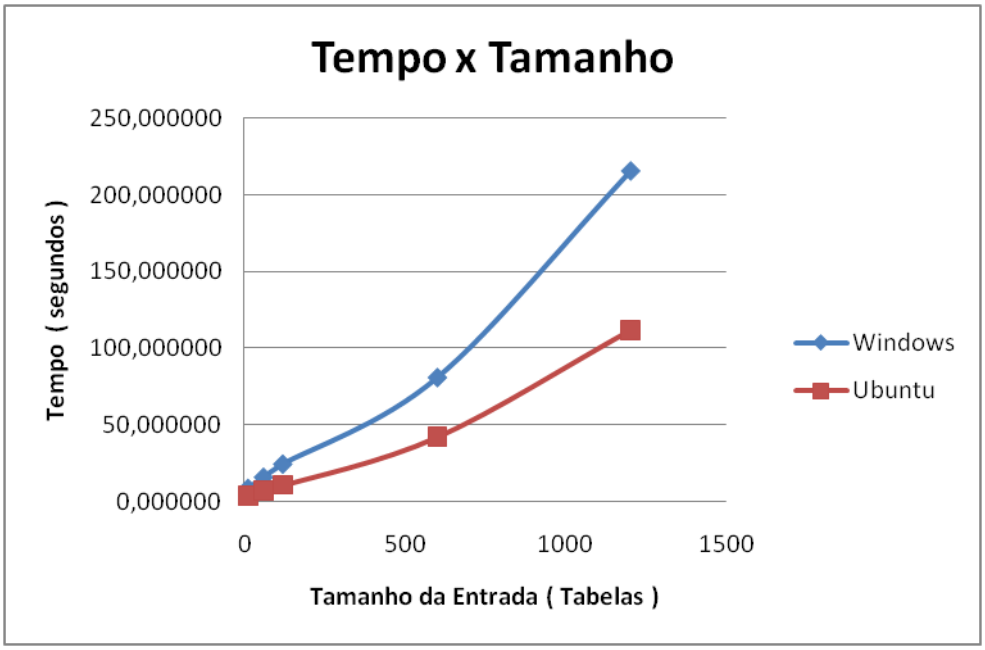


Figura 30: Desempenho - Tempo x Tamanho

O aumento do tempo segue de maneira proporcional ao aumento tamanho da tabela. Há um pequeno acréscimo no tempo de execução no Windows, por esse sistema operacional possuir máquina virtual *Ruby* mais lenta. Na Figura 31, o gráfico demonstra as alterações para cada quantidade de tabela diferente.

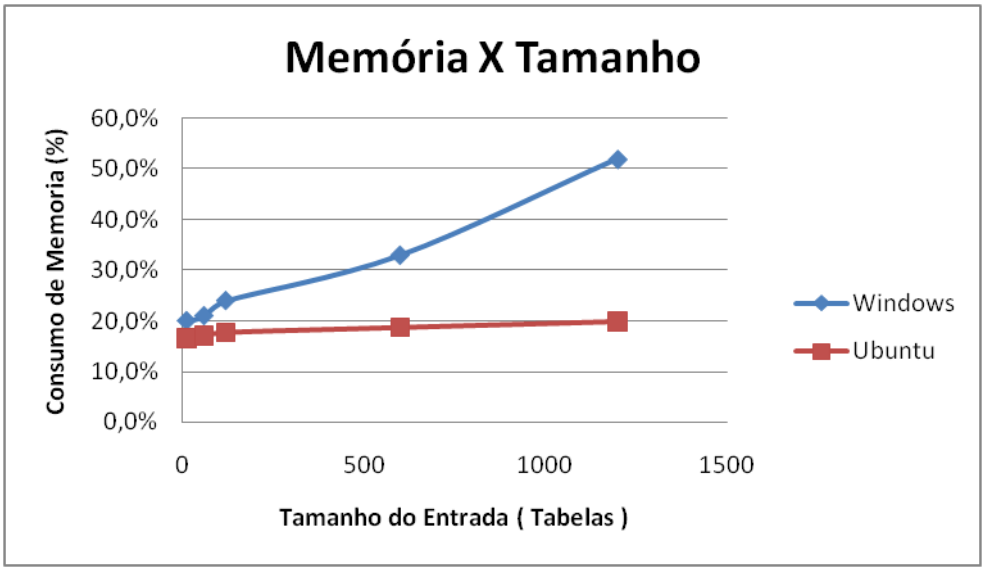


Figura 31: Desempenho - Memória x Tamanho.

O consumo de memória no Ubuntu fica praticamente estável com aumento de pouco mais de 3%. Já no Windows o consumo de memória aumenta de acordo com o tempo de execução do gerador, que é proporcional ao aumento do número de tabelas, chegando a mais de 30% de acréscimo. A Figura 32 mostra os gráficos obtidos no teste de processamento.

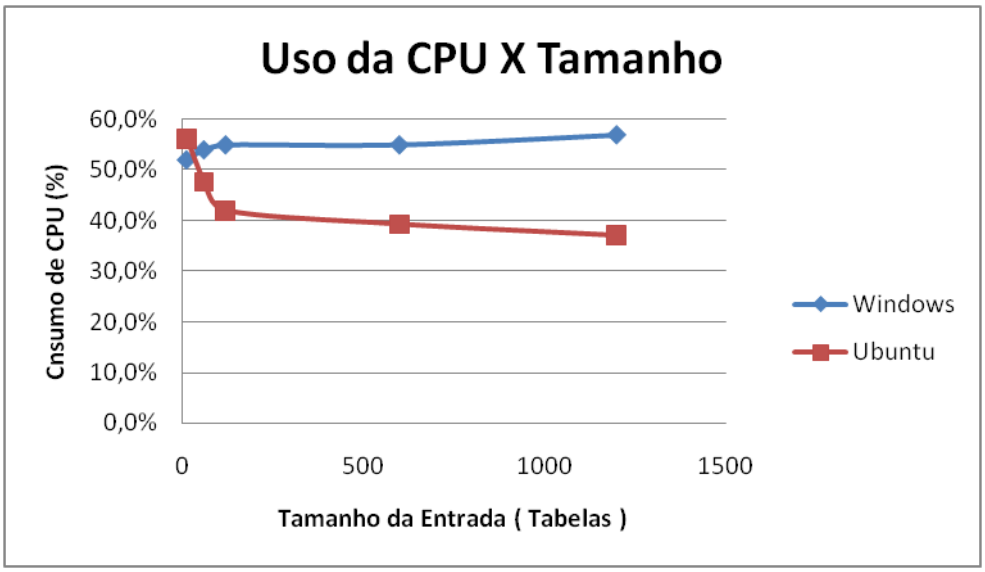


Figura 32: Desempenho - Uso da CPU x Tamanho

Nos resultados obtidos para consumo de CPU no Ubuntu, um comportamento diferente acontece. Com o aumento da quantidade de tabelas, o consumo de memória cai. Esse comportamento está ligado ao fato que o maior custo computacional exigido na fase de geração é feito ao buscar os dados na base de dados. O aumento do tempo gasto para busca de informações é pequeno em relação ao aumento de tempo gasto para criação de arquivos a cada elevação da quantidade de tabelas. Desse modo, a média geral do gasto computacional acaba diminuindo com o aumento do tamanho da entrada.

No Windows, o custo de processamento se mantém estável em todo o processo, de modo que o aumento desse custo é significantemente pequeno em relação ao acréscimo de tabelas.

O GEtna então pode ser considerado uma ferramenta pouco consumista, em relação ao consumo computacional. De modo que bases de dados extremamente grandes, com mais de mil tabelas, poderão ser tranformadas em aplicações sem grandes problemas.

5.4 Compatibilidade

O *Rails* é um *framework* que trabalha sobre várias plataformas e em diversos bancos de dados. O GEtna, de modo a garantir maior portabilidade e compatibilidade, foi desenvolvido possibilitando o trabalho com diversos bancos de dados e sistemas operacionais.

Para garantir que o gerador funcione no máximo de condições possível, foram realizados testes de compatibilidade em diversos ambientes. Todos os testes foram realizados com *Rails* na versão 2.0.2, 2.1.0 e 2.1.2.

Foram criados três ambientes para o Ubuntu onde se utilizou três bancos distintos para cada ambiente, para IDE's e linha de comando. Na Tabela 7, tem-se as características dos três ambientes criados.

Ubuntu 8.04.1 LTS (Hardy Heron)	
Netbeans 6.1	Mysql 5.0
	Postgresql 8.2.6
	Sqlite3-3.6.5
EasyEclipse 1.2.2.2	Mysql 5.0
	Postgresql 8.2.6
	Sqlite3-3.6.5
Linha de Comando	Mysql 5.0
	Postgresql 8.2.6
	Sqlite3-3.6.5

Tabela 7: Ambientes do Ubuntu

Para Windows, outros três ambientes foram criados, utilizando-se as mesmas bases de dados do Ubuntu para cada ambiente. Foi adicionado a *IDE RadRails* para essa plataforma, a Figura 33 demostra esse procedimento. Os ambientes podem ser vistos com mais detalhes na Tabela 8.

Windows Xp Home Edition SP3	
Netbeans 6.1	Mysql 5.0
	Postgresql 8.2.6
	Sqlite3-3.6.5
RadRails.0.7.2	Mysql 5.0
	Postgresql 8.2.6
	Sqlite3-3.6.5
Linha de Comando	Mysql 5.0
	Postgresql 8.2.6
	Sqlite3-3.6.5

Tabela 8: Ambientes no Windows

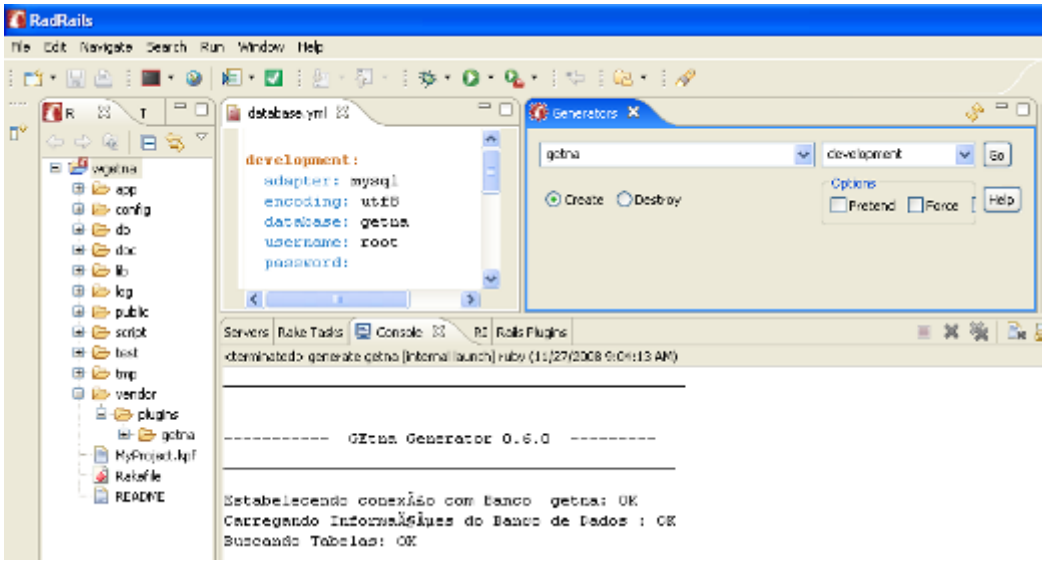


Figura 33: Geração Utilizando IDE RadRails no Windows

O GEtna foi disponibilizado para membros da comunidade que realizaram testes com Mac OS X Leopard, com duas bases de dados por intermédio do

console. Na Tabela 9, são apresentados os ambientes onde foram realizados os testes.

Mac OS X v10.5.4 Leopard	
Linha de Comando	Mysql 5.0
	Sqlite3-3.6.5

Tabela 9: Ambiente no Mac OS Leopard

Testes como esse provaram a grande compatibilidade e portabilidade que o gerador possui, sendo aplicados a várias IDE's e Banco de Dados. A presença de somente alguns ambientes não implica, necessariamente, que o gerador funcione somente nas condições descritas, podendo ser utilizado em outras diversas situações não mencionadas.

6 CONCLUSÃO

O GEtna foi desenvolvido em *Ruby* como um *plugin* para geração de código para o *framework Ruby on Rails,* com o objetivo de automatizar os processos de desenvolvimento de softwares utilizando a técnica de geração de código. O software se torna funcional após o término da geração com propriedades de inserção, leitura, alteração e deleção de dados. Desenvolvido em sua totalidade na linguagem script *Ruby*, o GEtna foi escrito para aplicações *Ruby on Rails*, de forma a minimizar configurações e excesso de códigos.

As maiores dificuldades encontradas ao desenvolver o GEtna foram encontrar uma forma de gerar uma aplicação mais funcional possível sem interferir na boa maleabilizade e fácil entendimento do código final e tratar relacionamentos de forma simples.

O gerador utiliza uma base de dados padronizada como entrada de informações para o procedimento de criação de arquivos. Técnicas e ferramentas ORM existentes no *framework* foram de essencial importância para o bom funcionamento do gerador.

Os arquivos gerados são baseados em *templates* limpos e facilmente legíveis e manipuláveis, fator importantíssimo para um resultado satisfatório. Toda estrutura de geração é baseada no MVC, proposto pelo *Rails*, o que proporciona uma fácil identificação dos arquivos gerados pelo *framework*.

Para que o gerador funcione é necessária pouquíssima configuração extra a configuração já existente em uma aplicação comum. A minimização, em questão de configurações, aumenta a agilidade e flexibilidade proporcionada pelo GEtna.

O gerador foi escrito para ser utilizado em qualquer sistema operacional, banco de dados e ferramenta com suporte ao *framework*. Isso garante a portabilidade e a massiva usabilidade em qualquer plataforma que o desenvolvedor utilize.

6.1 Trabalhos Futuros

O fato do GEtna ser um projeto *Open Source* lhe dá inúmeras possibilidades de acréscimos. Umas das melhorias previstas é a construção de um analisador, que se responsabilizará pela verificação da base de dados, observando falhas nas padronizações.

Um dos desafios a ser implementado no GEtna é a geração em bases não padronizadas, tornando, assim, possível que o sistema em outras linguagens possam ser migradas para o *Rails*.

Como o *framework* e as linguagens possuem constantes atualizações, o GEtna necessita acompanhar essas mudanças de modo a sua funcionalidade estar sempre garantida.

REFERÊNCIAS BIBLIOGRÁFICAS

AKITA, F. **Entendendo Git e Instalando Gitorious (Git via Web)**. 2008, Disponível em: < http://www.akitaonrails.com/2008/10/2/entendendo-git-e-instalando-gitorious-git-via-web>. Acesso em: 30 out. 2008, 21:21:12.

AMBLER, S. **Modelagem Ágil**: Praticas eficazes para a programação eXtrema e o Processo Unificado. 1º Ed, Bookman, Porto Alegre, Brasil, 2004. 352 p

AMBLER, S. W. **Mapping Object to Relational Databases**. 1999. Disponível em: <http://www.ambysoft.com/mappingObjects.pdf>. Acesso em: 16 out. 2008 04:05:12.

BACIC, N. M. **O software livre como alternativa ao aprisionamento tecnológico imposto pelo software proprietário**. 2003. 139f. Trabalho de Conclusão de Curso de Ciência da Computação, Universidade de Campinas. Disponível em: <http://www.rau-tu.unicamp.br/nou-rau/softwarelivre/document/?down=107>. Acesso em: 4 nov. 2008, 09:51:54.

BECK, K. et. al. **Principles behind the Agile Manifesto**. 2001, Disponível em: < http://agilemanifesto.org/principles.html >. Acesso em: 24 out. 2008, 10:09:02.

COAR, K. **The Open Source Definition**. 2006, Disponível em: <http://www.opensource.org/docs/osd>. Acesso em: 1 nov. 2008 15:31:42.

CUNHA NETO, S. M. **Rails Versus Struts**: Um Comparativo de Frameworks. 2007. 57f. Trabalho de Conclusão do Curso de Bacharelado em Sistemas de Informação, Escola de Informática Aplicada da Universidade Federal do Estado do Rio de Janeiro, Rio de Janeiro. Disponível em: <http://www.mergulhao.info/assets/2007/5/2/monografia.pdf>. Acesso em: 22 out. 2008, 21:57:23.

GITHUB,. **Secure Git hosting and collaborative development - GitHub**. 2008. Disponível em: <http://github.com/>. Acesso em: 31 out. 2008, 08:56:59.

GITHUB LANGUAGES. Top Languages. 2008. Disponível em: <http://github.com/languages>. Acesso em: 31 out 17:22:22.

HANSSON, D. **Ruby on Rails**. 2007. Disponível em: <http://www.rubyonrails.org/>. Acesso em: 17 out. 2008, 20:04:32.

HERRINGTON, J. **Code generation in action.** 1ed**.** Manning: Greenwich, CT, EUA, 2003. 450p.

KLUG, M. **Gerador de código JSP Baseado em Projeto de Banco de Dados**. 2007. 221f. Trabalho de Conclusão de Curso de Ciência da Computação, Universidade Regional de Blumenau, Blumenau. Disponível em: <http://www.campeche.inf.furb.br/tccs/2007-I/2007-1maiconklugvf.pdf>. Acesso em: 12 set. 2008, 02:41:33.

KORTH, H.; SILBERSCHATZ, A.; SUDARSHAN, S. **Sistema de Banco de Dado**. 5° ed. Campus: São Paulo, Brasil, 2006, 808p.

MATSUMOTO, Y. **Ruby Programming Language**. 2008. Disponível em: <http://www.ruby-lang.org/en/>. Acesso em: 17 out. 2008, 19:54:57.

RODRIGES, G.; COSTA J.; SILVEIR, P. E. **Projeto Panda Reloaded**. 2001. 15f. Trabalho de Formatura do Curso Ciência da Computação, Instituto de Matemática e Estatística da Universidade de São Paulo, São Paulo. Disponível em: <www.linux.ime.usp.br/~cef/mac499-02/monografias/peas/monografiaA.pdf>. Acesso em: 20 out. 2008, 23:00:02.

SANT'ANA, O et. al. **RailsBox Podcast #4**. RailsBox: 2008, 61 min, Disponível em: <http://railsbox.org/assets/2008/9/29/railsbox_4.mp3>, Acesso em: 1 nov. 2008 15:11:01.

SILVA, J. C. et. al. Estratégias de Persistência em Software Orientado a Objetos: Definição e Implementação de um Framework para Mapeamento Objeto-Relacional, **Revista Eletrônica Científica**, Juiz de Fora, V. 01, p. 1-21, set. 2006.

STRAIOTO, G. **Ruby on Rails e Goldberg seu site em minutos**. , 2007. Disponível em <http://blog.guinux.com.br/category/ruby-on-rails/>, Acesso em: 28 set. 2008 17:15:25.

THOMAS, D.; HANSSON, D. H. **Desenvolvimento Web Ágil Com Rails**. 2º. ed. Bookman: Porto Alegre,Brasil, 2008. 680 p.

VENNERS, B., 2003. **The Philosophy of Ruby**. Disponível em: <http://www.artima.com/intv/ruby4.html>. Acesso em: 17 out. 2008, 20:44:12.

WILLIAMS, J. **Rails Solutions**: Ruby on Rails Made Easy. 1º Ed, Friends of ED, Nova Iorque, EUA, 2007. 268 p.

YODER, J. W., JOHNSON, R. E., WILSON, Q. D. **Connecting Business Objects to Relational Databases.** 1998. Disponível em: <http://www.joeyoder.com/Research/objectmappings/Persista.pdf>. Acesso em: 16 out. 2008, 10:14:52.

APÊNDICE A – MÓDULO GETNA

Localização: */lib/getna.rb*

```
  1 #Copyright (c) 2008  Luiz Arão Araújo Carvalho
  2 #
  3 #Permission is hereby granted, free of charge, to any person obtaining
  4 #a copy of this software and associated documentation files (the
  5 #"Software"), to deal in the Software without restriction, including
  6 #without limitation the rights to use, copy, modify, merge, publish,
  7 #distribute, sublicense, and/or sell copies of the Software, and to
  8 #permit persons to whom the Software is furnished to do so, subject to
  9 #the following conditions:
 10 #The above copyright notice and this permission notice shall be
 11 #included in all copies or substantial portions of the Software.
 12 #
 13 #THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 14 #EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 15 #MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
 16 #NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
 17 #LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
 18 #OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
 19 #WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 20
 21
 22 module Getna
 23   class Base
 24     attr_reader :interrel, :table_names,:relationship, :validations,
:table_id
 25     $VERSION = "0.6.0"
 26     def initialize (env)
 27       start_messenger(env)
 28       #Abre Arquivo de Configuração de banco de dados do Rails
 29       conf = YAML::load(File.open("#{RAILS_ROOT}/config/database.yml"))
 30       if conf[env].nil?
 31         puts "#{red("\n\nErro de atributos: o Environment #{env} não
foi encontrado.")} \n"
 32         puts  "Utilizando development como fonte de dados para
geração!\n"
 33         env = 'development'
 34       end
 35       #Estabelece Conheção de acordo com o tipo de
Enviroment(production,development,teste ou outo)
 36       $stdout.print("\nEstabelecendo conexão com Banco
#{conf[env]['database']}: ")
 37       ActiveRecord::Base.establish_connection(conf[env])
 38       $stdout.print("OK ")
 39       #Realiza uma conexção com as configurações encontradas do
database.yml
 40       #Busca toda a estrutura da base de dados
 41       #Entre elas:
 42       # * Nome do banco, usuario, senha, endereço,
 43       # * adaptador(mysql, postgresql...), codificação, linguagem e
S.O.
 44       $stdout.print("\nCarregando Informações do Banco de Dados : ")
 45       @con = ActiveRecord::Base.connection
```

```
 46        $stdout.print("OK ")
 47        $stdout.print("\nBuscando Tabelas: ")
 48        #Busca todos os nomes de tabelas daquele banco de dados
 49        @table_names = @con.tables
 50        $stdout.print("OK \n")
 51        #Deletamos tabelas que não devem ser geradas(schema_migrations)
 52        @table_names.delete("schema_migrations")
 53        #Sessão Estatística
 54        ents = @table_names.size
 55        $stdout.print("\nExecutando ação para #{ents.to_s} Tabelas.")
 56        $stdout.print("\nAproximadamente  #{(ents*13+3).to_s} Arquivos e
#{(ents*1+2).to_s} Diretórios serão Gerados/Deletados.  \n\n\n")
 57        #Guarda informações sobre Relacionamentos para ser utilizada nas
Views
 58        @interrel = Hash.new
 59        #Guarda Informações sobre relacionamentos para ser utilizados nos
Models
 60        @relationship  = Hash.new
 61        #Guarda informações sobre Atributos de tabelas. Utilizado em
Validações nos Models
 62        @validations = Hash.new
 63        #Guarda um identificador para cada tabela
 64        @table_id = Hash.new
 65        next_id = 0
 66        #inicia Variáveis com a estrutura Necessária
 67        @table_names.each do |table|
 68          @relationship.store(table,[])
 69          @interrel.store(table,[false])
 70          @validations.store(table,[])
 71          @table_id.store(table,(next_id+=1).to_s.rjust(3, '0'))
 72        end
 73        #Iniciando identificação de relacionamentos
 74        has_many_through
 75        has_many
 76        create_validations
 77      end
 78      #=================================================
 79      # == Metodo to_view
 80      # Transforma tipos de dados encontrados em determinada tabela em
tipo de dados
 81      # de view, ou seja, para cada tipo de dados no banco ele transforma
de modo que este
 82      # possa ser convertido em tag html para criação do formulario na
view.
 83      #  == Exemplo
 84      # Tabela Usuario
 85      # nome: varchar
 86      # status: boolean
 87      #  O que nosso método realiza neste ponto é pegar varchar e
converter para text_field
 88      #  e boolean em check_box.
 89      #
 90      # == Parametros
 91      #  É necessário passar apenas o nome da tabela para que seja feito
o processo.
 92      #   to_view('nome_da_tabela')
 93      #   ==== Ex
 94      #   @user.tag_view = to_view('usuarios')
 95      # == Retorno
```

```
 96      # Retorna um Hash com a seguinte estrutura para um exemplo de
Usuario.
 97      #[{:type=>"text_field", :name=>"id"}, {:type=>"text_field",
:name=>"nome"}, {:type=>"text_field", :name=>"endereco"},
{:type=>"text_field", :name=>"grupo_id"}, {:type=>"date_select",
:name=>"created_at"}, {:type=>"date_select", :name=>"updated_at"}]
 98      def to_view(table_name)
 99        attr_view = []
100        #Exceções, são campos da tabela que não necessitam ser gerados
101        exceptions = ["id","created_at","updated_at"]
102        #Método que busca os atributos de cada tabela
103        attrs = columns(table_name)
104        #Inicia o processo de criação da estrutura formando pares de nome
do atributo e tipo do atributo
105        # caso este não esteja na lista de exceções
106        #Adicionado variavel :fixture para ser usada na criacao dos Tests
Fixtures. (by Silvio)
107        attrs.each do |att|
108          attr_view.push({:name
=>att.name,:type=>type_for_tag(att.type.to_s),:fixture=>default_for_value_f
ixture(att.default,att.type.to_s)}) if !exceptions.include?(att.name)
109        end
110        attr_view
111      end
112      #=====================================================
113      #Retorna os atributos de cada tabela dentro de um array, nesse
array contém
114      # todas as informações sobre esse atributo, como: nome, tipo e
tamanho
115      def columns(table_name)
116        @con.columns(table_name)
117      end
118      #=========================================================#
119      #Sessão De Identificação de Relacionamentos NxN
120      #Método que seta todas as variáveis com os Relacionamentos
encontrados
121      def has_many_through
122        @table_names.each do |table|
123          if (decomp_tables = decompounds(table))
124            #test(table)
125            if  (tables_exist?(decomp_tables))
126              if (has_nxn_keys?(decomp_tables, table))
127                create_relation_nxn_for(decomp_tables,table)
128                create_interface_nxn_for(decomp_tables,table)
129              end
130            end
131          end    #END:: if Decomp_tables
132        end #END Table Names Each
133      end #END Has Many Throught
134      #Decompõe o nome de uma tabela composta retornando um array
135      # com os nomes das tabelas que formaram o nome de entrada.
136      def decompounds(word)
137        is_compounds = word.match(/(.*)_+(.*)/)
138        decompoundeds = word.split('_').collect{|table| table.pluralize}
if is_compounds
139        decompoundeds || false
140      end
141
```

```
142     #ENTRADA: Array com nome das tabelas
143     #Retorna true se todas existirem e false se não
144     def tables_exist?(tables)
145       #$stdout.ptinf("TABELAS: #{tables}\n")
146       exist = tables.blank? ? false : true
147       tables.each do |table|
148         exist = (exist and @table_names.include?(table))
149       end
150       exist
151     end
152     #== Descrição
153     # Verifica, uma tabela composta, se ela possui as chaves
estrangeiras
154     # referêntes a elas.
155     #== Entrada
156     #- array com nomes da tabela
157     #- nome da tabela composta
158     #== Retorna
159     #true se sim, e false caso contrário
160     def has_nxn_keys?(rel_tables, thr_table)
161       table_w_keys = []
162       rtables = rel_tables.dclone
163       table_w_keys = has_keys?(thr_table)
164       table_w_keys.each{|table| rtables.delete(table)}
165       rtables.empty?
166     end
167     #Cria Relacionamento NxN
168     #== Entrada
169     #informa-se um array com as tabelas relacionadas e o nome da tabela
interrelacional
170     #EX:
171     # create_relation_nxn_for(['users','groups'],'group_users')
172     #== Saida
173     # Esse metodo seta automaticamente a variável de instância
@relationship e
174     #a retorna.
175     def create_relation_nxn_for(rel_tables,thr_table)
176       my_tables = rel_tables
177       rel_tables.each do |rtable|
178         my_tables.each do |table|
179           @relationship[rtable] << "has_many :#{table}, :through=>
:#{thr_table} "<< "has_many :#{thr_table} " if table !=rtable
180         end
181         # puts "THR: #{thr_table}"
182         @relationship[thr_table].push("belongs_to
:#{rtable.singularize}")
183         # puts "REL_THR: #{rtable}"
184       end
185       @relationship
186     end
187     #Seta informações sobre tabelas para serem usadas na View.
188     # Informa-se quais tabelas possuem relacionamento com a atual
tabela.
189     #possibilitando assim a identificação de chaves estrangeiras nas
Views
190     #
191     def create_interface_nxn_for(rtables, thr_table)
192       @interrel[thr_table]= rtables
193       @interrel
```

```
194       end
195       #==  Fim Da Sessão De Identificação de Relacionamentos NxN
==================#
196       # == Inicio Da Sessão de Identificação de Relacionamentos Nx1
==================#
197       #:: Para cada tabela
198       #- Buscamos campos com sufixo _id
199       #- Verificamos se existe uma tabela com aquele nome
200       #- adicionamos, para aquela tabela, o relacionamento Nx1
201       def has_many
202         @table_names.each do |table|
203           if ((rel_tables = has_keys?(table)) and
(!tables_exist?(decompounds(table) || [])))
204             create_relation_nxone_for(table, rel_tables)
205             create_interface_nxone_for(table, rel_tables)
206           end    #END:: if has_keys?
207         end #END Table Names Each
208       end #END Has Many Throught
209       def create_relation_nxone_for(table, rel_tables)
210         rel_tables
211         rel_tables.each do |rtable|
212           @relationship[table] << "has_many :#{rtable}"
213           @relationship[rtable].push("belongs_to :#{table.singularize}")
214         end
215         @relationship
216       end
217       def create_interface_nxone_for(table, rel_tables)
218         @interrel[table]= rel_tables
219         @interrel
220       end
221       #== Fim Da Sessão de Identificação de Relacionamento Nx1
======================#
222       def create_validations
223         @table_names.each do |name|
224           columns(name).each do |attr|
225             @validations[name].concat(build_validations(attr.name,attr))
226           end
227         end
228       end
229       def build_validations(name,attr)
230         validations = []
231         unnesscesary = (name=='id' or attr.type.to_s == 'boolean')
232         unless unnesscesary
233           if attr.null != true
234             validations.push("validates_presence_of :#{name},
:message=>\"não pode ficar em branco!\"")
235           end #fim null
236           if !is_key?(name)
237             if attr.limit != nil
238               validations.push("validates_length_of  :#{name},
:maximum=>#{attr.limit}, :message=>\"não pode exeder os #{attr.limit}
caracteres!\"")
239             end #fim IF limit
240             if attr.type.to_s == "integer"
241               validations.push("validates_numericality_of  :#{name},
:message=>\"deve ser numérico!\"")
242             end# Fim IF Type
```

```
243           end#fim IF isKEY
244         end #Unnesses
245         validations
246       end #FIM METODO
247       def is_key?(key)
248         res = key.match(/\A(.*_id)\z/)
249         !res.nil?
250       end
251       #========  MIGRAÇÕES ========================================
252       #=============================================================
253       #attr_migrate = [{:name=>"idade",:typo=>"integer",:null=>false,
limit=>2,:default=>false},{:name=>false,
type=>"timestamps",:null=>false,:limit=>false,:default=false}]
254       #
255       #template do objeto
256       #{:name=>false,:type=>false,:limit=>false, :null=>false,
:default=>false}
257       #
258       # Cria-se duas Migrações Defaults
259       #* References (para chaves estrangeiras- sufixo "_id")
260       #{:name=>attr_name,:type=>references,:limit=>false, :null=>false,
:default=>false}
261       #
262       #* Time Stamps (caso ache created_at e updated_at)
263       #{:name=>false,
type=>"timestamps",:null=>false,:limit=>false,:default=false}#
264       def to_migrate(table_name)
265         attr_migrate = []
266         #Exceções, são campos da tabela que não necessitam ser gerados
267         exceptions = ["id","created_at","updated_at"]
268         #Método que busca os atributos de cada tabela
269         attrs = columns(table_name)
270         timestamps = []
271         attrs.each do |att|
272           attr_migrate.push({
273               :name =>att.name,
274               :type=>att.type.to_s,
275               :limit=>(att.limit.nil? ? nil : att.limit.to_s),
276               :null=>(att.null ? nil : "false"),
277               :default=>(att.default.nil? ? nil : ((att.type.to_s ==
"boolean")? att.default : ( "\"#{att.default}\"")))
278             }) if !exceptions.include?(att.name) and !is_key?(att.name)
279           if is_key?(att.name)
280
attr_migrate.push({:name=>att.name.chomp('_id'),:type=>"references",:limit=
>nil, :null=>nil, :default=>nil})
281           end
282           timestamps.push(att.name) if
["created_at","updated_at"].include?(att.name)
283         end
284         if timestamps.size == 2
285           attr_migrate.push({:name=>nil,
:type=>"timestamps",:null=>nil,:limit=>nil,:default=>nil})
286         end
287         attr_migrate
288       end
289       #=== Métodos Comuns aos Relacionamentos ========================#
290       #== Descrição
```

```
291     #Método busca todos os atributos que são chaves estrangeiras
292     #na tabela
293     #== Retorno
294     #Retorna um Array com os nomes das tabelas que possuem chaves
estrangeiras
295     #na tabela.
296     def has_keys?(table)
297       table_w_keys =[]
298       columns(table).each do |attr|
299         if(is_key?(attr.name))
300           table_w_keys.push(attr.name.chomp('_id').pluralize)
301         end
302       end
303       table_w_keys
304     end
305     private
306     #=================================================
307     #Retorna tipos de dados de banco para tipo de dados de views
308     def type_for_tag(type)
309       tag ={
310         "string"=>"text_field",
311         "boolean"=>"check_box",
312         "integer"=>"text_field",
313         "text"=>"text_area",
314         "references"=>"text_area",
315         "datetime"=>"date_select"
316       }
317       tag[type]
318     end
319     #=============================================================
320     #Retorna tipos de dados para Text Fixtures com base nos
321     #valores default e type column. Recebe esses dois parametros
322     #e verifica se default for nil, usa o type column como valor
323     #convertido para valores que seguem o padrao scaffold,
324     #caso contrario o default e usado como valor. (by Silvio)
325     def default_for_value_fixture(default_value,type)
326       tag ={
327         "string"=>"MyString",
328         "integer"=>"MyInteger",
329         "boolean"=>"MyBoolean",
330         "text"=>"MyText",
331         "references"=>"MyText",
332         "datetime"=>DateTime.now.strftime(fmt="%Y-%m-%d %H:%M:%S")
333       }
334       default_value.nil? ? tag[type] : default_value
335     end
336     #===============  MESSAGES ==========================
337     #mensagem de inicialização do gerador
338     def start_messenger(env)
339
$stdout.print("_____________________________________________\n")
340       $stdout.print("\n\n- #{"GEtna Generator #{$VERSION}"}  -\n")
341       $stdout.print("_____________________________________________\n")
342     end
343   end
344   class Utilities
345     attr_accessor :hash_options_for
346     def initialize
347     end
```

```
348     # == Descrição
349     # Método de Tratamento da entrada do usuario
350     # == Exemplo
351     # script/generate getna development style:depot
352     # recebemos então o seguinte array:
353     # ['style:depot'], nosso método então converte em
354     # {'style'=>'depot'}
355     # == Retorno
356     # Retorna um Array com os pares de entrada do console
357     # do usuário.
358     # {"chave"=>"valor","chave"=>"valor"}
359     def hash_options_for(actions)
360       hash = {}
361       actions.each do|i|
362         k,v = i.split(':')
363         hash.store(k,v)
364       end
365       hash
366     end #END:: hash_options_for
367     # Meétodos Responsáveis pela colorização
368     def colorize(text, color_code)
369       "#{color_code}#{text}\e[0m"
370     end
371     def light(text); colorize(text, "\e[1m"); end
372     def red(text); colorize(text, "\e[1;31m"); end
373     def green(text); colorize(text, "\e[1;32m"); end
374     def yellow(text); colorize(text, "\e[1;33m"); end
375   end #Class Utilities
376 end
```

APÊNDICE B – GETNA GENERATOR

Localização: *generator/getna/getna_generator.rb*

```
1  # Vide Licença
2  class GetnaGenerator < Rails::Generator::NamedBase
3    #  Classe GetnaGenerator é responsável pelo processo de coleta de
dados via linha de comando
4    #   Apartir de dados encontrados em uma base de dados como Mysql,
PostgreSQL e SQLite,
5    # gera-se então toda a estrutura de uma aplicação Rails como um
Scaffold, mas em um unico commado.
6    #   Das tabelas são buscados nome das tabelas(para classes), nome dos
campos (para Atributos) e seus
7    #atributos(para validações) e ainda chaves estrangeiras (para
relacionamentos),
8    def initialize(runtime_args, runtime_options = {})
9     super
10    # Instânciamos o Objeto GEtna com as infomações do Banco de dados
11    @geobject = Getna::Base.new(@name)
12    #instâncimamos Objeto Getna::Utilities para tratarmos a entrada
```

```
13     @utility = Getna::Utilities.new
14      #Pega o tipo de Layout, caso não seja informado utiliza-se default
15    options =   @utility.hash_options_for(actions)
16    @style =  options['layout'] || 'default'
17    end
18    def manifest
19      record do |m|
20        # == GENERATE Diretorio de Layouts
21        m.directory("app/views/layouts")
22        # == GENERATE Diretório de Migrações
23        m.directory("db/migrate")
24        # == GENERATE Stilo
25        m.file("/styles/#{@style}.css","public/stylesheets/getna.css")
26       # == COPY getna logo
27        m.file("/images/getna.png","public/images/getna.png")
28        # == GENERATE Pagina inicial do GEtna
29        m.template("index.html.erb","public/index.html",:assigns=>{:entities
=>@geobject} ,:collision => :force)
30        #Para Cada tabela do banco colocamos em nosso Hash
31        # * Name:Singular: Nome do Tabela no Singular
32        # * Name:Plural: Nome do Tabela no Plural(Nome da Tabela)
33        # * Name:Class: Nome da Tabela no Singular e "Camelizado"
34        # * Name:Class Plural: recebe o Nome da Tabela "Camelizado"
35        # e Attrs: Recebe os Atributos daquela tabela transformados em
Atributos
36        # de de View como : String = Text Field, Text = Text Area
37        @geobject.table_names.each do |table_name|
38          name = Hash.new
39          name[:single] = table_name.singularize
40          name[:plural] = table_name
41          name[:class] = table_name.singularize.camelize
42          name[:class_plural] = table_name.camelize
43          attrs = @geobject.to_view(table_name)
44          # ==GENERATE Controllers
45          # Para cada Tabela é então copiado o template controller.rb para
a pasta do projeto
46          #app/controllers/ com o nome no Plural, passamos tambem as
variáveis que devem ser mudadas
47          #dentro dos Templates( Todos os NAMES acima.S ).
48
m.template("controller.html.erb","app/controllers/#{name[:plural]}_controller
.rb",:assigns=>{:object_name=>name,:rel_props=>@geobject.interrel[table_name]})
49          # == GENERATE Models
50           # Para cada Tabela é então copiado o template model.rb para  a
pasta do projeto
51          #app/models/ com o nome no Plural, passamos tambem as variáveis
que devem ser mudadas
52          #dentro dos Templates( Todos os NAMES acima.S ).
53
m.template("model.html.erb","app/models/#{name[:single]}.rb",:assigns=>{:object_na
me=>name,
:relationship=>@geobject.relationship[table_name],:validations=>@geobject.validatio
ns[table_name]})
54          # == GENERATE Views Directory
55          # Criamos o Diretorio que vai conter as Views com o nome da
tabela no plural
56          m.directory("app/views/#{name[:plural]}")
57          # == GENERATE Views Files
```

```
58          #Criamos então para cada tabela os 4 arquivos de CRUD:
edit.html.erb, index.html.erb, show.html.erb, new.html.erb.
59          #na Pasta que acabamos de gerar, utilizando seus respectivos
templates view_edit.html.erb, view_index.html.erb,
60          # view_show.html.erb, view_new.html.erb
61
m.template("view_edit.html.erb","app/views/#{name[:plural]}/edit.html.erb",:as
signs=>{:attributes=>attrs,:object_name=>name,:if_relation_object=>@geobject.interrel[ta
ble_name][0]})
62
m.template("view_index.html.erb","app/views/#{name[:plural]}/index.html.erb",:
assigns=>{:attributes=>attrs,:object_name=>name})
63
m.template("view_show.html.erb","app/views/#{name[:plural]}/show.html.erb",:as
signs=>{:attributes=>attrs,:object_name=>name})
64
m.template("view_new.html.erb","app/views/#{name[:plural]}/new.html.erb",:assig
ns=>{:attributes=>attrs,:object_name=>name,:if_relation_object=>@geobject.interrel[tabl
e_name][0]})
65          # == GENERATE Routes
66          # Geramos a rota para cada objeto gerado.
67          # Adicionado funcao para verificar se a rota ja existe. (by
Silvio)
68          m.route_resources name[:plural] unless
File.read("config/routes.rb").index(name[:plural])
69          # == GENERATE Layout
70          # Geramos a layouts para cada objeto gerado.
71
m.template("layouts/#{@style}_layout.html.erb","app/views/layouts/#{name[:pl
ural]}.html.erb",:assigns=>{:object_name=>name})
72          #CREATE Helpers
73
m.template("helper.rb","app/helpers/#{name[:plural]}_helper.rb",:assigns=>{:object
_name=>name})
74          #CREATE Testes Unitários
75
m.template("unit_test.rb","test/unit/#{name[:single]}_test.rb",:assigns=>{:object_
name=>name})
76          #CREATE Functional Tests
77
m.template("functional_test.rb","test/functional/#{name[:plural]}_controller_
test.rb",:assigns=>{:object_name=>name})
78          #CREATE Test Fixtures
79
m.template("test_fixture.yml","test/fixtures/#{name[:plural]}.yml",:assigns=>{:at
tributes=>attrs})
80          #GENERATE Migrations
81          m.template 'migration.html.erb',
"db/migrate/#{@geobject.table_id[name[:plural]]}_create_#{name[:plural]}.rb",
:assigns=>{:attributes=>@geobject.to_migrate(name[:plural]),:object_name=>name}
82        end #END:: Each Table Name
83    end # END:: do-record(m)
84  end #END:: Manifest
85 end #END:: Class
```