

Gustavo Toshi Komura

## **Criando e Adaptando Bibliotecas do Ruby**

Trabalho de conclusão do curso de Ciência da Computação. Departamento de Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Universidade Federal do Paraná

Ciência da Computação

Departamento de Informática

Orientador: Professor Doutor Bruno Müller Junior

Curitiba

Dezembro de 2014

# Lista de ilustrações

Figura 1 – Resultado de Translate na View . . . . .	24
Figura 2 – Diagrama de Classes Google-Maps-For-Rails . . . . .	27
Figura 3 – Diagrama de Atributos Google-Maps-For-Rails . . . . .	28
Figura 4 – Diagrama de Herança Google-Maps-For-Rails . . . . .	29
Figura 5 – Novo Diagrama de Herança Google-Maps-For-Rails . . . . .	32
Figura 6 – Caminho entre São Paulo e Curitiba . . . . .	36

# Lista de Códigos

3.1	Cria Gema Forma Geral	13
3.2	Cria Gema gemtranslatetoenglish	13
3.3	Execução que cria gema gemtranslatetoenglish	13
3.4	gemspec gemtranslatetoenglish	14
3.5	gemtranslatetoenglish.rb	17
3.6	translatetoenglish.rb	18
3.7	Testa translate gemtranslatetoenglish	19
3.8	Rakefile gemtranslatetoenglish	20
3.9	Execução que Cria e Instala gemtranslatetoenglish	21
3.10	Execução rake gema gemtranslatetoenglish	21
3.11	Executa rails new para gemtranslatetoenglish	22
3.12	Executa rails generate para gemtranslatetoenglish	23
3.13	Adiciona gemtranslatetoenglish no Gemfile	23
3.14	Exemplo do translate() na view	23
4.1	Classe Primitives com atributo de Directions	31
4.2	Funções adicionais do Handler	33
4.3	Exemplo CoffeeScript que Cria Mapa com Direção	34
4.4	Exemplo Locations view que Cria Mapa com Direção	35
B.1	Tudo é objeto para o Ruby	44
B.2	Ruby flexível	45
B.3	Convenção Ruby	45
E.1	Instala RVM	50
E.2	Instala Ruby	51
E.3	Configura Variáveis RVM	51
E.4	Instala Gemas Essenciais	52
F.1	Exemplo de uso do IRB	53
F.2	Teste IRB da gema gemtranslatetoenglish	54

# Sumário

<b>1</b>	<b>Introdução</b>	<b>6</b>
<b>2</b>	<b>Bibliotecas do Ruby</b>	<b>8</b>
2.1	Bibliotecas do Ruby	8
2.1.1	O programa gem	9
2.2	Segurança do Ruby	9
2.3	Segurança das Bibliotecas do Ruby	10
<b>3</b>	<b>Criação de bibliotecas do Ruby</b>	<b>11</b>
3.1	Estrutura de uma Biblioteca do Ruby	11
3.2	Modelo de criação	12
3.2.1	Criando a Estrutura	13
3.2.2	Gemspec	13
3.2.3	Desenvolvimento de código de funcionalidade ou teste	15
3.2.4	Código de Funcionalidade no Diretório Lib	15
3.2.5	Código de Teste no Diretório Test ou Spec	19
3.2.6	Execução de testes	21
3.2.7	Exemplo de uso de gemtranslatetoenglish	22
<b>4</b>	<b>Adaptação de bibliotecas do Ruby</b>	<b>25</b>
4.1	Engenharia Reversa	26
4.2	Entendimento da Biblioteca do Ruby	29
4.3	Adaptações	30
4.4	Exemplo de uso de Google-Maps-for-Rails	34
<b>5</b>	<b>Conclusão</b>	<b>37</b>
	<b>Referências</b>	<b>38</b>
	<b>Apêndices</b>	<b>40</b>
	<b>APÊNDICE A História e Classificação de Bibliotecas</b>	<b>41</b>
A.1	História	41
A.2	Classificação	41
A.2.1	Formas de ligamento	42
A.2.2	Momentos de ligação	42
A.2.3	Formas de compartilhamento	42

<b>APÊNDICE B</b>	<b>Conceitos e História do Ruby</b>	<b>44</b>
B.1	Ruby	44
B.2	História	45
B.3	API	46
<b>APÊNDICE C</b>	<b>Uso da Ferramenta gem</b>	<b>47</b>
<b>APÊNDICE D</b>	<b>Ferramentas utilizadas</b>	<b>48</b>
D.1	VMware® Player	48
D.2	RVM	48
D.3	Ruby On Rails	48
D.4	Git	49
<b>APÊNDICE E</b>	<b>Preparação do ambiente</b>	<b>50</b>
<b>APÊNDICE F</b>	<b>IRB</b>	<b>53</b>
<b>APÊNDICE G</b>	<b>API do Google Maps</b>	<b>55</b>

# 1 Introdução

Até o ano de 2005, a rederização de mapas em uma página web custava caro. Isso ocorria, porque para apresentar um mapa, era necessário fazer a transferência de uma grande quantidade de dados do servidor para o cliente. Neste caso, existia uma grande quantidade de dados, porque era feito a transferência de um mapa inteiro. Como consequência, também existia um alto custo no cliente para transformar os dados em um mapa.

Graças ao *Google*, em Fevereiro de 2005, essa dor de cabeça foi aliviada com a criação de uma nova solução de rederização, implementada na ferramenta *Google Maps*. Esta nova solução, consistia na divisão do mapa em pedaços. No caso, estes pedaços poderiam ser requisitados um a um para formar uma parte ou um mapa completo. Com essa nova rederização, foi possível solucionar o problema da grande quantidade de dados transferidos, pois somente se transfere as partes requisitadas do mapa, e também se solucionou problema da transformação, pois a quantidade de dados para se transformar, foi reduzida razoavelmente.

Muitas companhias, percebendo a solução revolucionária adotada pelo *Google* na ferramenta *Google Maps*, não ficaram atrás, adotando soluções sinilares que também reduziam a carga de trabalho nos servidores. Entre estas ferramentas estão, o *Yahoo Maps* do *Yahoo*, o *Bing Maps* da *Microsoft*, e o *Yandex Maps* do *Yandex*.

Para os desenvolvedores, esta nova solução de rederização, tornou-se interessante quando se analisou as operações que poderiam ser realizadas sobre os mapas. Por exemplo, nos mapas do *Google Maps*, podemos criar um mapa em uma página web, utilizar marcadores para marcar locais ou regiões do mapa que consideramos importante, e até determinar caminhos para ir de um local ao outro.

Por causa destas utilidades, vários programadores perceberam que a nova ferramenta do *Google* poderia ser muito útil em seus projeto, e por este motivo decidiram se mobilizar na procura de uma forma de incorporar o *Google Maps* em suas aplicações. O *Google*, percebendo essa mobilização, resolveu facilitar o acesso a ferramenta, divulgando em Junho de 2005 uma *API* (*Application Programming Interface*) para o *Google Maps*, descrevendo nesta *API*, detalhes de como importar e utilizar as funções da ferramenta.

O *Ruby On Rails* que é um *framework* da linguagem *Ruby*, possui bibliotecas que simplificam o acesso da *API* do *Google Maps*. Basicamente estas bibliotecas preparam internamente os objetos do mapa e depois mapeiam as funções da ferramenta do *Google*. Por exemplo, quando se está utilizando uma destas bibliotecas, ao tentar rederizar um mapa em uma página web, não é necessários saber quais os objetos do *Google Maps*

precisam ser criados, pois estes objetos serão gerados automaticamente na chamada da função de renderização de mapas da biblioteca.

O objetivo deste trabalho, é mostrar uma forma de adaptar uma biblioteca do *Ruby* que faz o mapeamento da *API* do *Google Maps*. Para isso será necessário apresentar conceitos sobre as bibliotecas do *Ruby*, bem como a criação do mesmo.

Neste caso, o texto apresenta um passo-a-passo da implementação de uma biblioteca simples e funcional do *Ruby*. Como exemplo, será desenvolvido um biblioteca que faz a tradução de palavras do português para o inglês. E utilizando este conhecimento de criação de bibliotecas, o texto também apresenta os passos básicos que devem ser realizados para adaptar uma biblioteca do *Ruby*. Neste caso, será feito a adição da funcionalidade de desenhar caminhos entre dois locais, utilizando uma biblioteca do *Ruby* que mapeia a *API* do *Google Maps*.

Este trabalho está organizado da seguinte maneira: bibliotecas do *Ruby* no capítulo 2, onde será apresentado conceitos sobre bibliotecas do *Ruby*, criação de bibliotecas do *Ruby* no capítulo 3, onde será apresentado um passo-a-passo de como criar uma bibliotecas do *Ruby*, adaptação de bibliotecas do *Ruby* no capítulo 4, onde será apresentado um tutorial de como adaptar uma biblioteca do *Ruby*, e a conclusão no capítulo 5.

## 2 Bibliotecas do Ruby

Este capítulo tem o objetivo de apresentar brevemente os conceitos de biblioteca, mostrar algumas definições básicas sobre as bibliotecas do *Ruby*, e os riscos e procedimento de segurança que existem para elas.

Biblioteca do inglês *library*, é um conjunto de fontes de informação que possuem recursos semelhantes. Para a computação, uma biblioteca é um conjunto de subprogramas ou rotinas, agrupadas em um mesmo arquivo, que tem por função principal prover funcionalidades usualmente utilizadas por desenvolvedores em um determinado contexto. Neste caso, os desenvolvedores não precisam ter nenhum conhecimento sobre o funcionamento interno das bibliotecas, mas precisam saber para que serve cada uma das funcionalidades e como usá-las.

Geralmente as bibliotecas possuem uma *API* (*Application Programming Interface*), onde é disponibilizado as suas funcionalidades e a sua forma de uso, mostrando quais são os parâmetros de entrada e saída, e os seus respectivos tipos.

A utilização de uma biblioteca torna-se importante, porque além de modularizar um *software*, ela permite que os desenvolvedores não se preocupem em fazer implementações repetitivas, ou seja, fazer cópias de funções de um produto para outro. Isso se deve ao fato que quando a biblioteca for incluída em um projeto, a função já vai estar disponível para uso neste projeto.

Para mais informações sobre bibliotecas, consulte o apêndice [A](#).

Em seguida, neste capítulo, na seção [2.1](#), vamos apresentar alguns conceitos das bibliotecas do *Ruby*, depois na seção [2.2](#), vamos falar um pouco sobre a segurança do *Ruby*, e por fim na seção [2.3](#), vamos falar sobre a segurança das bibliotecas do *Ruby*. Caso ainda não conheça a linguagem *Ruby*, consulte o apêndice [B](#).

### 2.1 Bibliotecas do Ruby

Assim como muitas linguagens, como por exemplo *C*, *C++*, *Java*, *Python* e muitas outras, o *Ruby* também possui um vasto conjunto de bibliotecas, sendo que a maior parte delas é distribuída na forma de *gem*. Seu processo de instalação é feito por meio do programa *gem*, que será explicado na sub-seção [2.1.1](#) [\[1\]](#).

Também existe um menor número de bibliotecas que são lançadas como arquivos compactados em “.zip” ou “.tar.gz”. Seu processo de instalação geralmente é feito por meio de arquivos de “README” ou “INSTALL”, que possuem as instruções de instalação [\[1\]](#).



As bibliotecas do *Ruby* foram apelidadas com o nome de *gem* ou gemas, justamente pelo motivo que a maior parte das bibliotecas é desenvolvida no formato de *gem*.

### 2.1.1 O programa gem

O *gem*<sup>1</sup> é um sistema de pacotes do *Ruby*, desenvolvido para facilitar a criação, o compartilhamento, e a instalação de bibliotecas. Esta ferramenta possui características sinilares ao sistema de distribuição de pacotes *apt-get*, no entanto ao invés de fazer a distribuição de pacotes para *Debian GNU/Linux distribution* e seus variantes, ela faz a distribuição de pacotes *Ruby* [1].

O projeto *RubyGems*, desenvolvedor da ferramenta *gem*, foi criado em Abril de 2009 por *Nick Quaranto*<sup>2</sup>, numa tentativa de proporcionar uma melhor *API* para acessar as gemas [2].

Com o passar do tempo, o projeto cresceu, chegando a possuir mais de 115 *Rubyists* contribuintes e milhões de gemas baixadas [2].

Até a versão “1.3.6”, o nome do projeto era *Gemcutter*, sendo renomeada a partir desta versão para *RubyGems*, objetivando a solidificação do papel central do site na comunidade do *Ruby* [2].

Para mais informações, consulte o apêndice C.

## 2.2 Segurança do Ruby

A proteção de dados sempre foi uma questão muito discutida e na comunidade do *Ruby* isso não é diferente, justamente porque a todo momento estamos sujeitos a sofrer ataques, e com isso podemos ser prejudicados, perdendo dinheiro e informações sigilosas.

A comunidade do *Ruby*, sabendo destes problemas, construiu um esquema para corrigir as vulnerabilidades de segurança. Neste esquema, as vulnerabilidade descobertas são reportadas via e-mail para [security@ruby-lang.org](mailto:security@ruby-lang.org), que é uma lista privada com membros que administram o *Ruby*, como por exemplo *Ruby committers*. Neste esquema, os membros da lista de segurança, por medidas de segurança, somente compartilham as vulnerabilidades quando elas já estão solucionadas. Na publicação da vulnerabilidade é informado o tipo de erro, os problemas que o erro causa, e a solução que deve ser tomada para sua correção [3].

Um exemplo é a vulnerabilidade publicada em 10/04/2014, referenciado com o identificador *CVE* (*CVE-2014-0160*), fala sobre um grave problema na implementação no *OpenSSL's* do *TLS/DTLS* (*transport layer security protocols*). Neste problema, uma

<sup>1</sup> gem: <https://rubygems.org/>

<sup>2</sup> Nick Quaranto: <https://twitter.com/qrush>

pessoa mal intencionada poderia roubar dados da memória, tanto na comunicação entre o servidor com o cliente, como na comunicação do cliente para o servidor [4].

## 2.3 Segurança das Bibliotecas do Ruby

Assim como na linguagem *Ruby*, também é necessário tomar alguns cuidados ao se utilizar as suas bibliotecas, pois podemos estar abrindo brechas, e com isso correndo riscos de perder informações.

Antes de instalar qualquer biblioteca do *Ruby*, devemos verificar se a fonte da gema é confiável. Isso é válido para a instalação de qualquer biblioteca em qualquer linguagem.

O *RubyGems* a partir da versão “0.8.11”, permite a assinatura criptografada de uma *gema*. Essa assinatura funciona com o comando “*gem cert*”, que cria um par de chaves e empacota o dado da assinatura dentro da *gema*. O comando “*gem install*” permite que se defina uma política de segurança, onde se pode verificar a chave da assinatura antes da instalação [5].

Apesar deste método de chaves criptografadas ser benéfico, ele geralmente não é usado, pois é necessário vários passos manuais no desenvolvimento, e também não existe nenhuma medida de confiança bem definida para estas chaves de assinatura [5].

Assim como o *Ruby*, o *RubyGems* também possui um esquema para reportar vulnerabilidades. Este esquema é dividido em duas vertentes. Na primeira se reporta erros na gema de outros usuários e na segunda se reporta erros da própria gema.

No caso para reportar vulnerabilidade de *gemas* de outros usuários, sempre é necessário verificar se a vulnerabilidade ainda não é conhecida. Caso ela ainda não seja conhecida, é recomendado que se reporte o erro por um e-mail privado, diretamente para o dono da gema, informando o problema e indicando uma possível solução.

Por outro lado caso descubra uma vulnerabilidade em uma de suas *gema*, primeiramente é necessário que se requisite um identificador *CVE* via e-mail para [cve-assign@mitre.org](mailto:cve-assign@mitre.org), pois deste modo existirá um identificador único para o problema. Com o identificador em mãos, é necessário trabalhar em uma possível solução. Assim que encontrar uma solução, será necessário criar um *patch* de correção. E finalmente depois de criar o *patch*, deve-se informar a comunidade que existia um problema na *gema* e que essa vulnerabilidade foi corrigida no *patch* “*x*”.

Para este segundo caso, também recomenda-se adicionar o problema em um *database open source* de vulnerabilidade, como por exemplo o [OSVDB](https://osvdb.org). Além disso, também recomenda-se enviar um e-mail para [ruby-talk@ruby-lang.org](mailto:ruby-talk@ruby-lang.org) com o *subject*: “[ANN]/[Security]”, informando detalhes sobre a vulnerabilidade, as versões que possuem esse erro, e as ações que devem ser tomadas para corrigir o problema.

## 3 Criação de bibliotecas do Ruby

Como visto no capítulo anterior ??, a maior parte das bibliotecas do *Ruby* são distribuídas na forma de *gems*, e também vimos na seção 2.1.1 deste mesmo capítulo que a ferramenta *gem* é um sistema de distribuição que facilita o compartilhamento e a instalação de *gemas*.

Deste modo, como já temos alguns conhecimento básicos sobre bibliotecas e sobre o *Ruby*. Este capítulo tem o objetivo de mostrar um passo-a-passo que pode ser seguido para criar uma gema.

Antes de iniciar o processo de criação de uma gema, precisamos entender para que serve cada um de seus componentes, e isso será explicado logo a seguir na seção 3.1. Depois com o conhecimento de cada componente da gema, apresentaremos um modelo de criação de gemas na seção 3.2. Caso tenha dúvidas sobre as ferramentas utilizadas ou de como preparar o ambiente de desenvolvimento, consulte os apêndices D ou E, respectivamente.

### 3.1 Estrutura de uma Biblioteca do Ruby

Uma gema do *Ruby*, obrigatoriamente deve possuir um nome, um número de versão e uma plataforma. Internamente ela deve possuir códigos de funcionalidade, uma documentação e um *gemspec*.

A sua estrutura geralmente é organizada em 3 arquivos bases: o *gemspec*, o *Rakefile* e o *README*, e em 3 diretórios principais: *bin*, *lib*, e *test* ou *spec*. Na listagem abaixo veremos para que serve cada um destes arquivos e diretórios.

- O *gemspec* é um arquivo do tipo “*.gemspec*” que possui as informações básicas de uma gema, como por exemplo o seu nome, sua descrição, seu autor, seu endereço, e suas dependências.
- O *bin* é um diretório que possui os arquivos executáveis da gema que serão carregados quando a gema for instalada.
- O *lib* é um diretório que possui todos os códigos *Ruby* referentes ao funcionamento da gema.
- O *Rakefile* é um arquivo que possui código *Ruby* que faz a otimização de algumas funcionalidades por meio da execução do programa *rake*<sup>1</sup>. Um exemplo, é a execução de todos os arquivos de testes do diretório *test* ou *spec*.

---

<sup>1</sup> rake: <https://github.com/jimweirich/rake>

- O **test** ou **spec** é um diretório que possui todos os códigos *Ruby* de testes. Eles podem ser executados manualmente ou por meio do *Rakefile*.
- O **README** é um arquivo que usualmente possui a documentação da gema. Esta documentação, usualmente é retirada de comentários que estão dentro do código, e gerados automaticamente quando a gema é instalada. A maioria das gemas possuem a documentação *RDoc*<sup>2</sup>, e as outras, em menoria, possuem a documentação *YARD*<sup>3</sup> [6].

## 3.2 Modelo de criação

O primeiro passo para se criar uma gema, é construir uma solução de um certo problema que será utilizado com frequência. Por exemplo, a função que calcula a raiz quadrada de um número, é uma funcionalidade que usualmente utilizamos quando estamos fazendo cálculos.

Após encontrar uma ideia para a criação de uma gema, deve-se elaborar um projeto, fazendo o levantamento de requisitos, o *design*, a implementação, os testes e a entrega.

Por simplificação, nesta seção somente apresentaremos a parte de implementação do modelo de criação de uma gema, mas nunca se deve esquecer de seguir todos os passos de um projeto, desde o momento da formação da ideia até a sua entrega. Caso esses passos não sejam seguidos, existem altos riscos de se perder recursos, como tempo e dinheiro.

Para facilitar a apresentação deste modelo utilizaremos como exemplo a gema *gemtranslatetoenglish*<sup>4</sup>, que tem como objetivo fazer a tradução de um texto em português para um texto em inglês.

Futuramente pretendemos aumentar o vocabulário da gema de exemplo, mas até o momento de término deste trabalho, ela possuía somente a tradução de duas palavras, “OLÁ” para “HELLO” e “MUNDO” para “WORLD”. Apesar de possuir pouco vocabulário, a gema “*gemtranslatetoenglish*” possui características suficientes para a apresentação do tutorial deste trabalho.

Nesta seção, apresentaremos primeiramente como se pode criar uma estrutura básica de uma gema na sub-seção 3.2.1, em seguida mostraremos na sub-seção 3.2.2 como se deve editar o arquivo *.gemspec*, depois iremos apresentar como se pode desenvolver o código de funcionalidades de uma gema na sub-seção 3.2.4, sequencialmente apresentaremos na sub-seção 3.2.5 como implementar o código de testes, em seguida na sub-seção 3.2.6

<sup>2</sup> RDoc: <http://rdoc.sourceforge.net/doc/>

<sup>3</sup> YARD: <http://yardoc.org/>

<sup>4</sup> *gemtranslatetoenglish* : [https://github.com/toshikomura/gemtranslatetoenglish/tree/without\\_path](https://github.com/toshikomura/gemtranslatetoenglish/tree/without_path)

apresentaremos uma forma para executar os testes, e por fim na sub-seção 3.2.7 apresentaremos um exemplo de uso da gema “*gemtranslatetoenglish*” dentro de um projeto.

### 3.2.1 Criando a Estrutura

O primeiro passo é fazer a criação da estrutura da gema e isso pode ser feito de forma manual ou automática. A forma manual implica em criar todos os diretórios e arquivos manualmente. A forma automática implica na execução de um simples comando.

Contudo podemos perceber que a forma manual não é muito aconselhável, pois perderíamos muito tempo criando arquivo por arquivo. Deste modo, utilizaremos a forma automática que além de criar a estrutura, prepara um código base dentro dos arquivos. A execução do comando apresentado no código 3.1 no terminal faz a criação da estrutura básica de forma automática.

```
1 bundle gem 'nome da gema'
```

Código 3.1 – Cria Gema Forma Geral

No caso da nossa gema de exemplo, foi feita a execução do seguinte comando apresentado no código 3.2.

```
1 bundle gem gemtranslatetoenglish
```

Código 3.2 – Cria Gema gemtranslatetoenglish

Ao se fazer a execução do comando “*bundle gem gemtranslatetoenglish*”, obtemos a seguinte estrutura de gema mostrada no código 3.3. Podemos perceber neste código, que foi feita a criação do diretório “*gemtranslatetoenglish*” com os arquivos “*Gemfile*”, “*Rakefile*”, “*LICENSE.txt*”, “*README.md*”, “*.gitignore*”, “*gemtranslatetoenglish.gemspec*”, e o diretório “*lib*”. E dentro do diretório “*lib*” foi criado o arquivo “*gemtranslatetoenglish.rb*”, e o diretório “*gemtranslatetoenglish*” com o arquivo “*version.rb*” dentro dele.

```
1 gtk10@ubuntu:~$ bundle gem gemtranslatetoenglish
2   create  gemtranslatetoenglish/Gemfile
3   create  gemtranslatetoenglish/Rakefile
4   create  gemtranslatetoenglish/LICENSE.txt
5   create  gemtranslatetoenglish/README.md
6   create  gemtranslatetoenglish/.gitignore
7   create  gemtranslatetoenglish/gemtranslatetoenglish.gemspec
8   create  gemtranslatetoenglish/lib/gemtranslatetoenglish.rb
9   create  gemtranslatetoenglish/lib/gemtranslatetoenglish/version.rb
10 Initializing git repo in /home/gtk10/gemtranslatetoenglish
```

Código 3.3 – Execução que cria gema gemtranslatetoenglish

### 3.2.2 Gemspec

Agora que possuímos a estrutura da gema, devemos fazer a edição do arquivo *gemspec* para informar os dados básicos da gema, e isso pode ser feito editando o arquivo

“ ‘nome da gema’,gemspec ”. No nosso exemplo, fizemos a edição do arquivo “gemtranslatetoenglish.gemspec” resultado no arquivo mostrado no código 3.4 , onde cada linha será explicada com mais detalhes logo a seguir.

```
1 # coding: utf-8
2 lib = File.expand_path( '../lib', __FILE__)
3 $LOAD_PATH.unshift(lib) unless $LOAD_PATH.include?(lib)
4 require 'gemtranslatetoenglish/version'
5
6 Gem::Specification.new do |spec|
7   spec.name           = "gemtranslatetoenglish"
8   spec.version        = Gemtranslatetoenglish::VERSION
9   spec.authors        = ["Gustavo Toshi Komura"]
10  spec.email          = ["gtk10@c3sl.ufpr.br"]
11  spec.summary         = %q{Gema para traduzir portugues para ingles.}
12  spec.description     = %q{Gema que recebe um texto em portugues e retorna um texto em
13    ingles.}
14  spec.homepage        = ""
15  spec.license         = "MIT"
16
17  spec.files           = `git ls-files -z`.split("\n")
18  spec.executables     = spec.files.grep(%r{^bin/}) { |f| File.basename(f) }
19  spec.test_files      = spec.files.grep(%r{^(test|spec|features)/})
20  spec.require_paths   = ["lib"]
21
22  spec.add_development_dependency "bundler", "~> 1.5"
23  spec.add_development_dependency "rake"
24  spec.add_development_dependency "action_controller"
25 end
```

Código 3.4 – gemspec gemtranslatetoenglish

- “`# coding: utf-8`” na linha “1” indica que o texto do arquivo está no formato *UTF-8*.
- “`lib = File.expand_path('../lib', __FILE__)`” na linha “2” indica onde se encontra o diretório **lib** da gema.
- “`$LOAD_PATH.unshift(lib) unless $LOAD_PATH.include?(lib)`” na linha “3” faz o carregamento dos arquivos que estão no diretório **lib**, somente se o diretório já esta definido.
- “`require 'gemtranslatetoenglish/version'`” na linha “4” requisita o arquivo de versão da gema.
- “`Gem::Specification.new do |spec| ... end`” da linha “6” a “23” define a especificação da gema como *spec*, ou seja, ao invés de escrever “*Gem::Specification*” a todo momento que for definir uma especificação da gema se escreve somente “*spec*”.
- “*spec.* ” da linha “7” a linha “14” defini-se especificações básicas da gema, como nome, versão, autor, e-mail do autor, breve descrição, descrição completa, página e tipo de licença.

- “`spec.files = ‘git ls-files -z’.split("\x0")`” na linha “16” indica os arquivos que devem ser incluídos na gema. Esses arquivos são incluídos dinamicamente através do comando “`git ls-files -z`”. Este comando do *git*, traz como resultado todos os arquivos que estão naquele repositório, colocando entre as *PATHs*, o caracter “`\0`” (*line termination on output*). Por consequência, com a adição do comando Ruby “`.split("\x0")`”, é feita a divisão por “`\0`”, separando as *PATHs* dos arquivos. Deste modo, os arquivos adicionados na gema, são todos que estão no repositório.

Para se adicionar arquivos no repositório é necessário executar o comando “`git add PATH`”, onde *PATH* é o caminho do arquivo que se deseja adicionar no repositório.

- “`spec.executables = ...`” e “`spec.test_files = ...`” nas linhas “17” e “18” indicam os arquivos executáveis e os arquivos de teste respectivamente. Também indica que os arquivos dentro destes diretórios, devem ter permissão de execução.
- “`spec.require_paths = [“lib”]`” requisita o diretório da **lib** da gema.
- “`spec.add_development_dependency = ...`” nas linhas “21”, “22” e “23” requisitam como dependências as gemas “*bundle*” versão “1.5”, “*rake*” e “*action\_controller*” respectivamente.

### 3.2.3 Desenvolvimento de código de funcionalidade ou teste

Nesse momento podemos tomar 2 caminhos e isso depende da metodologia de projeto que adotamos no início do desenvolvimento, ou seja, é nesse momento que podemos desenvolver o código das funcionalidades ou implementar o código de testes.

Na metodologia tradicional se faz a implementação do código de funcionalidades e depois se desenvolve o código para testar as funcionalidades. Por outro lado, na metodologia voltada para testes, se implementa o código de testes para depois se desenvolver o código de funcionalidades.

Seguindo a metodologia tradicional, primeiramente iremos fazer o código das funcionalidades da gema. Depois ao terminar de criar essas funcionalidades, iremos elaborar os arquivos testes, mas nada o impede de desenvolver os códigos de testes que serão apresentado na sub-seção 3.2.5, antes de desenvolver os códigos de funcionalidade mostrados na sub-seção 3.2.4.

### 3.2.4 Código de Funcionalidade no Diretório Lib

Nesta sub-seção vamos aprender a fazer o código de funcionalidade de uma gema, mas caso deseje fazer primeiro os códigos de casos de testes, pode-se consultar a sub-seção 3.2.5, e depois retornar para esta sub-seção para dar continuidade ao desenvolvimento.

Caso esse código de funcionalidade seja desenvolvido por meio de código *Ruby*, devemos fazer a edição e a criação de arquivos no diretório **lib**. Este diretório obrigatoriamente deve possuir um arquivo “ nome da gema.rb ” e um diretório, com um arquivo de versão, também com o nome da gema. No nosso exemplo, podemos verificar no código 3.3, que após a execução do comando “*bundle gem gemtranslatetoenglish*”, é feita a criação do arquivo “*lib/gemtranslatetoenglish.rb*” na linha “8”, e a criação do diretório “*lib/gemtranslatetoenglish*” com o arquivo “*version*” na linha “9”.

O arquivo “*version*”, somente define a versão que a gema está. No nosso exemplo da gema “*gemtranslatetoenglish*”, a primeira versão é a “*0.0.1*”.

Basicamente a descrição da versão de uma gema é uma string com números e pontos. Também é permitido colocar ao final a palavra chave “*pre*”, caso seja um pré-lançamento de alguma versão. Por exemplo, “*1.0.0.pre*” é um pré-lançamento da versão “*1.0.0*”.

O *Rubygems* recomenda seguir as seguintes políticas mencionadas logo abaixo que foram consultadas em [semantic-versioning](http://guides.rubygems.org/patterns/#semantic-versioning)<sup>5</sup> e em [specification-reference-version](http://guides.rubygems.org/specification-reference/#version)<sup>6</sup>.

- PATH : “0.0.X” para pequenas alterações, como por exemplo correção de pequenos *bugs*.
- MINOR: “0.X.0” para médias alterações, como por exemplo alteração/adição de funcionalidades.
- MAJOR: “X.0.0” para grandes alterações, como por exemplo remoção de alguma funcionalidade.

Antes de iniciarmos a codificação das funcionalidades da gema, precisamos entender algumas diferenças básicas de conceitos do *Ruby*, como por exemplo a diferença entre *module* e *class*,

Os “*modules*” ou módulos se preferir, definem um conjunto de métodos e constantes. Podemos dizer que os métodos dos módulos são estáticos, pois não precisamos instanciar o módulo para usar os seus métodos. Contudo podemos dizer que os módulos são parecidos com o conceito de interface do *Java*.

Por outro lado a “*class*” também é um conjunto de métodos e constantes, no entanto para usar os seus métodos e constantes é necessário instanciá-la, ou seja, é necessário criar um objeto da “*class*” na memória para usar os seus respectivos métodos.

<sup>5</sup> semantic-version: <http://guides.rubygems.org/patterns/#semantic-versioning>

<sup>6</sup> specification-reference-version: <http://guides.rubygems.org/specification-reference/#version>



Contudo por essas características podemos dizer que uma “*class*” é basicamente uma subclasse do “*module*”, pois a “*class*” possui 4 métodos a mais, que no caso são os métodos “*initialize()*”, “*superclass()*”, “*allocate()*” e “*to\_yank()*”.

Agora que temos alguns conceitos do *Ruby* apresentados, podemos continuar com a implementação da nossa gema.

No arquivo “*lib/’nome da gema’.rb*” temos a possibilidade de escrever todas as funcionalidades desejadas. No nosso exemplo o arquivo “*lib/gemtranslatetoenglish.rb*” é mostrado no código 3.5, onde cada linha é explicado logo a seguir.

```
1 require "gemtranslatetoenglish/version"
2 require "gemtranslatetoenglish/translate_to_english.rb"
3
4 module Gemtranslatetoenglish
5   # Your code goes here...
6 end
7
8 ActionController::Base.helper Gemtranslatetoenglish::Helpers::
   TranslateToEnglish
```

Código 3.5 – *gemtranslatetoenglish.rb*

- “*require "gemtranslatetoenglish/version"*” na linha “1” é feita a requisição do arquivo de versão.
- “*require "gemtranslatetoenglish/translate\_to\_english.rb"*” na linha “2” é feita a requisição do arquivo “*translate\_to\_english.rb*” contido no diretório “*gemtranslatetoenglish*”.
- “*module Gemtranslatetoenglish ... end*” na linha “4” a “6” define o módulo da gema.
- “*ActionController::Base.helper Gemtranslatetoenglish::Helpers::TranslateToEnglish*” na linha “8” define uma extensão da classe “*ActionController::Base.helper*”, onde a classe a ser acrescentada é a classe “*Gemtranslatetoenglish::Helpers::TranslateToEnglish*”. Esta extensão foi adicionada para que no momento de uso das funcionalidades da gema na *view* não fosse necessário fazer a chamada de tradução escrevendo toda *PATH*. Por exemplo para chamar a função de tradução, ao invés de chamar “*gemtranslatetoenglish.TranslateToEnglish.translate('Olá')*”, se faz a chamada “*translate('Olá')*” na *view*. Isto é possível, pois nos projetos do *framework Ruby On Rails*, os métodos de auxílio das *view*, por simplificação já possuem a *PATH* “*ActionController::Base.helper*”, deste modo, ao se incorporar um método nesta classe, não é mais necessário digitar a *PATH* por completo.

Podemos perceber que o comando “*require*” é utilizado para fazer a chamada de código de outros arquivos e isso serve para fazer a modularização da gema que no caso

é uma boa prática de programação. Suponha que desenvolvemos uma gema e depois de um certo tempo precisamos fazer a manutenção do seu código. Neste caso se não modularizamos a gema, a correção de *bugs* ou mesmo a adição de novas funcionalidades, torna-se uma tarefa muito complexa, pois não existe nenhuma organização estrutural na gema preparada para facilitar esse tipo de operação.

Observando novamente o código 3.5, podemos perceber que na linha “4” foi feito o *require* do arquivo “*gemtranslatetoenglish/translateoenglish.rb*” que será mostrado no código 3.6, e explicado logo a seguir.

```
1 module Gemtranslatetoenglish
2   module Helpers
3     module Translateoenglish
4
5       def translate( phrase)
6         # Check if phrase has something
7         if phrase == nil
8           return ""
9         end
10        if phrase.empty?
11          return ""
12        end
13        # To each word in the phrase
14        i = 0
15        phrase = phrase.split
16        result = ""
17        for word in phrase
18          word = word.upcase
19          case word
20            when "OLÁ"
21              result = result + "HELLO"
22            when "MUNDO"
23              result = result + "WORLD"
24            else result
25              "- ({word}) NAO PODE SER TRADUZIDO -"
26            end
27          # If is not in the end of phrase
28          if i < phrase.length
29            result = result + " "
30            i = i + 1
31          end
32        end
33        return result
34      end
35    ...
```

Código 3.6 – translateoenglish.rb

- “*module ... end*” nas linhas “1”, “2” e “3” define a árvore de módulos “*Gemtranslatetoenglish*” na raiz, “*Helpers*” no segundo nível e “*Translatetoenglish*” no terceiro.
- “*def translate( phrase) ... end*” da linha “5” até a linha “34”, define a função de tradução da gema, onde foi definido somente duas traduções, “*OLÁ*” para “*HELLO*” e “*MUNDO*” para “*WORLD*”.

A árvore definida no código 3.6 foi necessário por causa do código inserido na linha “8” “*ActionController::Base.helper Gemtranslatetoenglish::Helpers::Translatetoenglish*” no código 3.5 que serve para evitar a necessidade de escrever a *PATH* completa na *view* para chamar uma função da gema na *view*.

### 3.2.5 Código de Teste no Diretório Test ou Spec

Lembrando que podemos fazer o desenvolvimento de código de funcionalidades ou o código de testes, e isso é dependente da metodologia adotada no início do projeto. Esta sub-seção tem o objetivo de mostrar como se pode desenvolver o código de casos de testes, mas caso queira fazer o código de funcionalidade antes, se pode consultar a sub-seção 3.2.4.

Para se fazer os testes, deve-se criar os arquivos de testes dentro do diretório “*test*” ou se preferir “*spec*”. Não existe um padrão especificado, mais recomenda-se criar um arquivo de teste com o nome “*test/test\_‘definição do teste’.rb*”. No nosso exemplo, criamos o arquivo “*test/test\_check\_translate.rb*” no código 3.7, explicado logo a seguir.

```
1 require 'minitest/autorun'
2 require 'action_controller'
3 require 'gemtranslatetoenglish'
4
5 include Gemtranslatetoenglish::Helpers::Translatetoenglish
6
7 class TranslateTest < MiniTest::Unit::TestCase
8   def test_world_translation
9     assert_equal "HELLO ", Gemtranslatetoenglish::Helpers::
10    Translatetoenglish.translate("OLÁ")
11     assert_equal "WORLD ", Gemtranslatetoenglish::Helpers::
12    Translatetoenglish.translate("MUNDO")
13   end
14   def test_text_translation
15     assert_equal "HELL WORLD ", Gemtranslatetoenglish::Helpers::
16    Translatetoenglish.translate("OLÁ MUNDO")
17     assert_equal "WORLD HELLO ", Gemtranslatetoenglish::Helpers::
18    Translatetoenglish.translate("MUNDO OLÁ")
19   end
20 end
```

Código 3.7 – Testa translate gemtranslatetoenglish

- Nas linhas “1”, “2” e “3” nos códigos “`require ‘..’`” requisitamos respectivamente o “`autorun`” da gema “`minitest`” que utilizaremos para realizar os testes, “`action_controller`” que utilizamos para evitar a obrigação de digitar a “`PATH`” completa na *view*, e “`gemtranslatetoenglish`” que é a nossa gema de exemplo.
- Na linha “5” fomos obrigados a fazer o “`include`” do módulo “`Gemtranslatetoenglish::Helpers::Translatetoenglish`” para que todas as funções deste módulo fossem disponibilizadas para uso. No nosso caso, era o método “`translate()`”.
- Da linha “7” a linha “16” é definido a classe de teste “`TranslateTest`” que herda as características de “`MiniTest::Unit::TestCase`”.
- Da linha “8” a linha “11” é definido o teste por palavra, onde é verificado através do “`assert_equal`” se a “`string`” esperada no primeiro parâmetro é retornada pela chamada da função “`Gemtranslatetoenglish::Helpers::Translatetoenglish.translate()`” no segundo parâmetro.
- Da linha “12” a linha “15” é definido o teste por texto, onde é verificado através do “`assert_equal`” se a “`string`” esperada no primeiro parâmetro é retornada pela chamada da função “`Gemtranslatetoenglish::Helpers::Translatetoenglish.translate()`” no segundo parâmetro.

Agora que temos o nosso arquivo de teste, precisamos criar o arquivo “*Rakefile*”. Depois na próxima sub-seção 3.2.6, iremos explicar como realizar os teste com a execução da ferramenta “*rake*”.

No nosso exemplo da gema “*gemtranslatetoenglish*” desenvolvemos o seguinte arquivo “*Rakefile*” que pode ser visualizado no código 3.8, sendo explicado em detalhes logo a seguir.

```
1 require "bundler/gem_tasks"
2 require 'rake/testtask'
3
4 Rake::TestTask.new do |t|
5   t.libs << 'test'
6 end
7
8 desc "Run tests"
9 task :default => :test
```

Código 3.8 – Rakefile gemtranslatetoenglish

- Nas linhas “1” e “2” nos códigos “`require ‘..’`” requisitamos respectivamente o “`gem_tasks`” do “*bundler*” e “`testtask`” do “*rake*”, ambos necessários para a execução dos testes.

- Da linha “4” a “6” é feito a criação de uma nova “*task*” de teste para cada arquivo que esteja no diretório “*test*”.
- A linha “5” com o código “ *t.libs « 'test'* ” inidica que os arquivos de testes estão no diretório “*test*”.
- Por fim na linha “9” com o código “*task :default => :test*”, se requisita a execução dos testes.

### 3.2.6 Execução de testes

Esta sub-seção tem o objetivo de mostrar como se pode realizar os teste, após criar os códigos de funcionalidade, visto na sub-seção 3.2.4, e os códigos de teste, visto na sub-seção 3.2.5.

Antes de realizarmos os testes, precisamos criar a gema com comando “*gem build 'nome da gema'.gemspec*” e fazer a instalação com o comando “*sudo gem install 'nome da gema' -'versão da gema'.gem*”. No nosso caso é fazer a criação e a instalação da gema “*gemtranslatetoenglish*”, que pode ser feito da mesma forma como no código 3.9.

```
1 # build gem
2 gem build gemtranslatetoenglish.gemspec
3 # install gem
4 sudo gem install gemtranslatetoenglish -0.0.1.gem
```

Código 3.9 – Execução que Cria e Instala gemtranslatetoenglish

Agora após ter implementado o código das funcionalidades, o arquivo de teste “*test/test\_check\_translate.rb*”, o arquivo *Rakefile*, e ter feito a criação e a instalação da gema de exemplo, podemos realizar os testes com a execução do comando “*rake*” no terminal.

Com a execução dos testes, obtemos como resultado o código 3.10, explicado logo a seguir.

```
1 gtk10@ubuntu:~/gemtranslatetoenglish$ rake
2 Run options: --seed 65419
3 # Running tests:
4 Finished tests in 0.000414s, 4834.6197 tests/s, 9669.2395 assertions/s.
5 2 tests, 4 assertions, 0 failures, 0 errors, 0 skips
```

Código 3.10 – Execução rake gema gemtranslatetoenglish

- Na linha “1” é feito a execução do comando “*rake*” para se realizar os testes.
- Na linha “5” é mostrado que foram feitos 2 testes, no caso os 2 que definimos no código 3.7 nas linhas “8” a “11” no teste “*test\_world\_translation*” e nas linhas “12” a “15” no teste “*test\_text\_translation*”. Também é apresentado na linha “5” que

foram feitos 4 “*assertions*”, no caso dois para cada caso de teste feitos por meio do “*assert\_equal*”. E além disso é apresentado que não ocorreram *failures*, “*errors*” e “*skips*”.

Deste modo, ao se realizar estes testes garantimos que pelo menos a função “*translate()*” da gema “*gemtranslatetoenglish*”, esta funcionando como o esperado.

Contudo no momento do desenvolvimento do código de testes, o aconselhável é para cada função adicionada na gema, fazer testes com todas as possíveis entradas, verificando se o resultado para cada entrada está correto.

Para informações de outra forma de teste, consulte o apêndice [F](#).

### 3.2.7 Exemplo de uso de *gemtranslatetoenglish*

Esta sub-seção tem o objetivo de mostrar a utilização da gema de exemplo “*gemtranslatetoenglish*”, que criamos no tutorial, em um projeto do *framework Ruby On Rails*.

Até o momento falamos muito da utilização do *action\_controller* para simplificar o uso da função *translate()* na *view*, e agora vamos apresentar essa facilidade através de um exemplo, fazendo o uso da gema “*gemtranslatetoenglish*” em um projeto.

O primeiro passo é criar um projeto no *framework rails* e isso pode ser feito executando o seguinte comando apresentado no código [3.11](#), explicado logo a seguir.

```
1 gtk10@ubuntu:~$ rails new projeto_teste_gemtranslatetoenglish
2   create
3   ...
4   create  Gemfile
5   create  config/routes.rb
6   ...
```

Código 3.11 – Executa rails new para *gemtranslatetoenglish*

- O comando “*rails new*” implica na criação de um projeto básico do *Ruby On Rails*.
- O nome “*projeto\_teste\_gemtranslatetoenglish*” é o nome do projeto a ser criado.
- Os códigos a partir da linha “2” não representam execuções. No caso estes códigos somente mostram os passos realizados por causa da execução do comando na primeira linha.
- A execução deste comando implica na criação de alguns diretórios e arquivos e por simplificação somente explicaremos aqueles que vamos utilizar neste exemplo:
  - “*Gemfile*” arquivo que contém as *gemas* que são utilizadas no projeto.
  - “*config/routes.rb*” arquivos que possui as rotas utilizadas no projeto.

Agora que criamos o projeto, precisamos fazer a criação de pelo menos um *controller* e uma *view*. O *controller* serve para receber uma requisição e determinar a partir dos parâmetros desta requisição, a *view* e os dados que devem ser apresentados. A *view* serve para determinar um formato e mostrar os dados no *browser*.

Para o nosso exemplo, criamos o *controller* “traducao” e a *view* “index”, com a execução do comando que pode ser visto no código 3.12, explicado logo a seguir.

```
1 gtk10@ubuntu:~/projeto_teste_gemtranslatetoenglish$ rails generate controller traducao
  index
2   create app/controllers/traducao_controller.rb
3   route  get "traducao/index"
4   invoke erb
5   create app/views/traducao
6   create app/views/traducao/index.html.erb
7   ...
```

Código 3.12 – Executa rails generate para gemtranslatetoenglish

- Na linha “1” é feito a execução do comando “*rails generate controller traducao index*” no terminal para gerar o *controller* “*traducao*”, e a *view* “*index*” para “*traducao*”.
- Os códigos a partir da linha “2” não representam execuções. No caso estes códigos somente mostram os passos realizados por causa da execução do comando na primeira linha.
- Na linha “2” foi criado o *controller* com o nome “*traducao\_controller.rb*”
- Na linha “3” foi adicionado no arquivo “*config/routes.rb*” o método *get* para a *view* “*traducao/index*”.
- Na linha “6” foi criado a *view* “*traducao/index.html.erb*”.

Agora para fazer o uso da nossa gema de exemplo em um projeto feito no *Ruby On Rails*, basta fazer a inclusão da gema no final do arquivo *Gemfile* da mesma forma como mostrado no código 3.13.

```
1 gem 'gemtranslatetoenglish'
```

Código 3.13 – Adiciona gemtranslatetoenglish no Gemfile

Agora que a gema “*gemtranslatetoenglish*” já está incluída no nosso projeto, podemos fazer o uso dela em uma *view* da mesma maneira como apresentada no código 3.14, explicado logo a seguir.

```
1 <h1>Traducao#index</h1>
2 <p>Find me in app/views/traducao/index.html.erb</p>
3 <p><%= phrase = translate "Olá Mundo" %></p>
```

Código 3.14 – Exemplo do translate() na view

- As linhas “1” e “2” já existiam, pois foram criadas automaticamente após a execução do comando “*rails generate controller traducao index*”, que mostramos no código 3.12.
- Na linha “3” inserimos uma *tag* `<p>` indicando para o *browser* que vamos inserir um texto. Depois inserimos a *tag* `<%= ... %>` que indica que entre essas *tags* será inserido um código *Ruby*. E dentro destas *tags*, chamamos o nosso método *translate()* da gema “*gemtranslatetoenglish*” com o parâmetro “Olá Mundo”.

Agora que temos a nossa função “*translate()*” dentro da *view* “*traducao/index*”, podemos verificar se a tradução funciona corretamente. Para isso devemos dentro do diretório do nosso projeto, iniciar o servidor no terminal através do comando “*rails server -p2342*”, onde o parâmetro “-p2342” indica que o servidor vai usar a porta “2342”.

Depois com o servidor funcionando, podemos verificar na imagem 1 que ao se acessar o endereço “localhost:2342/traducao/index” no *browser*, a nossa função de tradução funciona corretamente, pois na página é apresentado o texto “HELLO WORLD”.

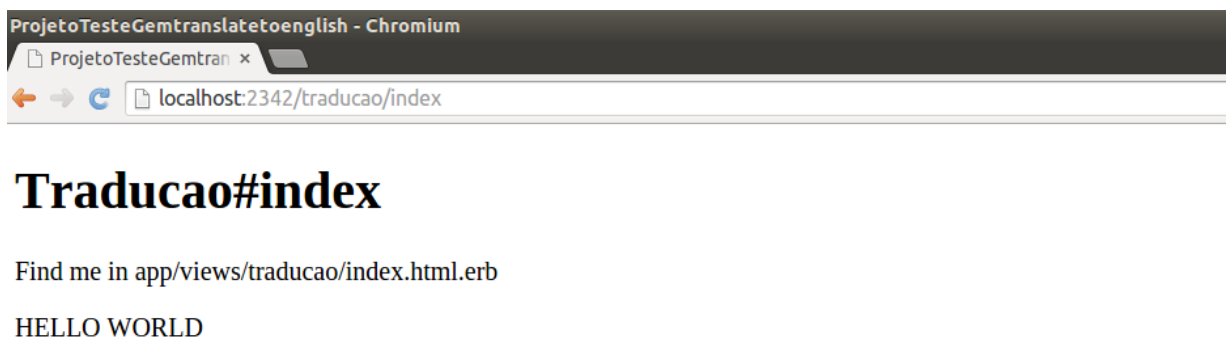


Figura 1 – Resultado de Translate na View



## 4 Adaptação de bibliotecas do Ruby

Agora que sabemos como criar uma gema, visto no capítulo 3, podemos partir para a ideia de fazer modificações em uma gema que já existe, ou seja, fazer a adaptação de uma gema, adicionando novas funcionalidade.

Suponha o cenário aonde utilizamos com frequência uma certa gema, no entanto apesar dela comportar várias funcionalidades, ela não possui tudo que desejamos. Neste caso, basta fazer a adaptação desta gema, não sendo necessário criar uma nova gema do zero.

É válido verificar se a funcionalidade que esteamos adicionando, esta no mesmo contexto da gema. Pois por exemplo, se estamos utilizando uma gema matemática e ela não possuir uma função para calcular a raiz de um número, podemos fazer uma modificação nesta gema para que ela possa calcular a raiz. Por outro lado, não faz nenhum sentido incluir uma funcionalidade de criar mapas do *Google*, pois esta funcionalidade não está no mesmo contexto da gema.

Para facilitar o entendimento utilizaremos para a explicação a gema *Google-Maps-for-Rails*<sup>1</sup> que é um *branch* da gema *Google-Maps-for-Rails*<sup>2</sup> criada por *Benjamin Roth*<sup>3</sup> e *David Ruyer*<sup>4</sup>.

Essa gema criada por tem como objetivo criar mapas de forma simplificada, proporcionando a inclusão de sobreposições oferecidas pelo *Google*, como por exemplo, marcadores e círculos. Ela também possui um código bem flexível que permite a aceitação de outros provedores de mapas, como por exemplo o *Bing Maps* [7].

Este capítulo tem o objetivo de mostrar os passos que devem ser realizados para se adaptar uma biblioteca do *Ruby*. Na seção 4.1, apresentamos o processo de engenharia reversa para gerar diagramas que representam o sistema em um alto nível de abstração, sendo seguida pela seção 4.2, onde apresentamos como fazer a leitura dos diagramas de alto nível. depois na seção 4.3, mostraremos o processo de adaptação e os cuidados que devemos tomar, e fecharemos o capítulo com a seção 4.4, onde apresentamos o uso da gema de exemplo que adaptamos. Para mais informações sobre o *Google Maps*, consulte o apêndice G.

<sup>1</sup> Google-Maps-for-Rails : <https://github.com/toshikomura/Google-Maps-for-Rails>

<sup>2</sup> Google-Maps-for-Rails : <https://github.com/apneadiving/Google-Maps-for-Rails>

<sup>3</sup> Benjamin Roth: <https://github.com/apneadiving>

<sup>4</sup> David Ruyer: <https://github.com/MrRuru>

## 4.1 Engenharia Reversa

Esta seção tem o objetivo de apresentar a importância da *engenharia reversa* no processo de adaptação de uma gema. Inicialmente veremos uma definição do processo de *engenharia reversa* e depois apresentaremos este processo sendo realizado na gema *Google-Maps-For-Rails*.

Para conseguirmos entender o funcionamento de uma gema, precisamos obrigatoriamente fazer uma tradução do código fonte para diagramas, pois não conseguiremos fazer nenhuma modificação consistente se não tivermos uma visão geral de seu funcionamento, e esse procedimento de tradução se chama *engenharia reversa*.

*Engenharia reversa* é um processo de análise para a extração de informações de algo que já existe em um modelo de abstração de alto nível. Essas informações podem estar no formato de código fonte ou mesmo em um executável. O processo de análise para a extração de dados deve ser feita de forma minuciosa, pois pode ocorrer uma grande perda de recursos, caso alguma funcionalidade seja entendida de forma incorreta. E o modelo de abstração de alto nível, pode ser por exemplo um diagrama de caso de uso ou um diagrama de sequência.

Aplicando a *engenharia reversa* na gema *Google-Maps-For-Rails* conseguimos obter como resultado: o diagrama de classe na imagem 2, o diagrama de atributos na imagem 3 e o diagrama de herança na imagem 4, que serão explicados em mais detalhes logo a seguir.

Nenhum dos 3 diagramas segue os seus respectivos padrões, isso se deve ao fato de não existir espaço suficiente na imagem para representar o sistema da gema por completo. Por esse motivo optamos por definir novos diagramas que possuem características sinilares aos seus padrões, mas com algumas características adicionais.

Para as imagens de diagrama de classes na imagem 2 e o diagrama de atributos na imagem 3 as seguintes explicações são válidas:

- Cada retângulo representa uma classe da gema.
- O nome acima dos traços “—” representa o nome da classe.

As seguintes explicações são válidas para o diagrama de classe na imagem 2:

- O digrama não mostra os atributos das classes.
- Todos os nomes seguidos de “(...)” abiaxo do traço “—” representam os métodos da classe.

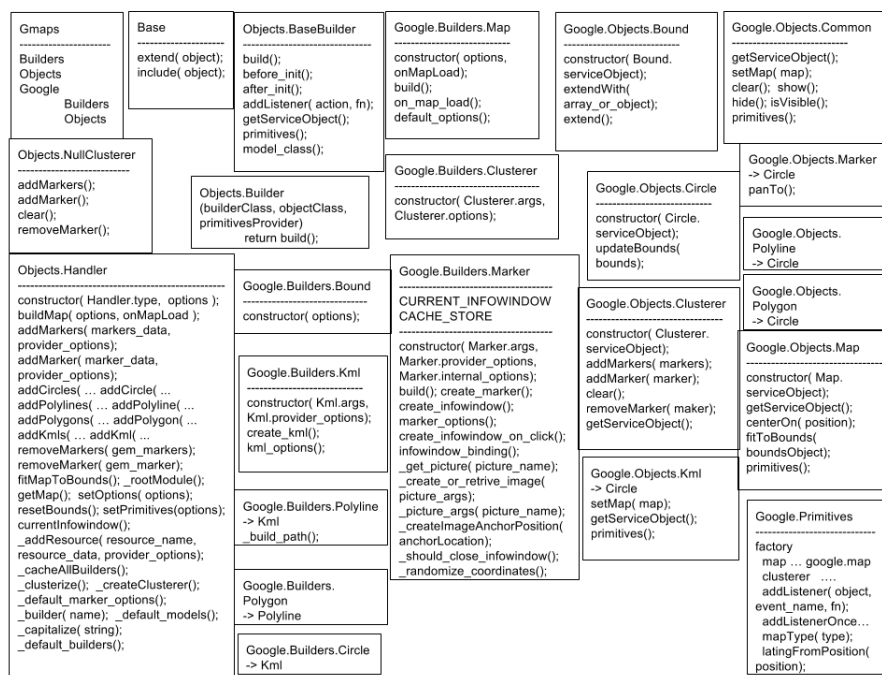


Figura 2 – Diagrama de Classes Google-Maps-For-Rails

- Existem classes que não possuem os traços “—”, neste caso elas possuem o nome delas seguida do símbolo “->” e depois um nome de outra classe. Isso significa que esta classe possui os mesmo métodos da classe que vem depois do símbolo “->”. Por exemplo a classe “*Google.Builder.Circle -> Kml*” representa a classe “*Google.Builder.Circle*” e ela possui os mesmo métodos da classe “*Kml*”, ou seja, ela possui os métodos “*constructor()*”, “*create\_...()*” e “*...\_option()*”. E também existe o caso onde esta classe possui métodos além dos da outra, e nesse caso esses métodos são colocados na linha de baixo. Por exemplo a classe “*Google.Builder.Polyline -> Kml*” que é a classe “*Google.Builder.Polyline*”, além de possuir os métodos da classe *Kml*, ela possui o método “*\_build\_path()*”.

As seguintes explicações são válidas para o diagrama de atributos apresentado na imagem 3:

- Todos os nomes abaixo do traço “—” representam os atributos da classe. Também existe o caso onde esse métodos são seguidos pelos símbolos “{ ... }”, onde o método é um *objeto* e os nomes separados por “,” entre os símbolos “{ ... }” são os atributos do *objeto*.
- Existem classes que não possuem os traços “—”, neste caso elas possuem o nome delas seguida do símbolo “->” e depois um nome de outra classe. Isso significa que esta classe possui os mesmo atributos da classe que vem depois do símbolo “->”. Por

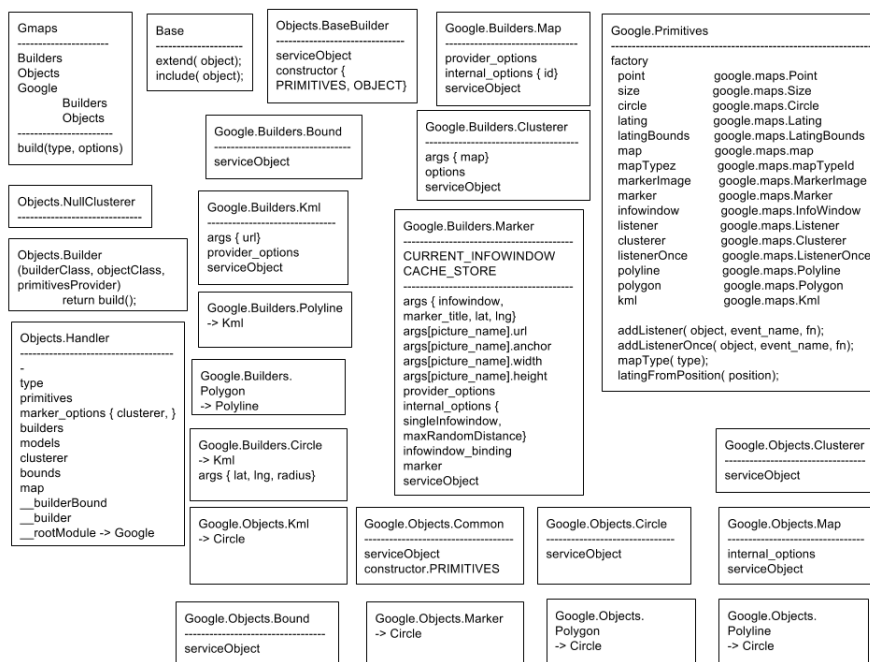


Figura 3 – Diagrama de Atributos Google-Maps-For-Rails

exemplo a classe “*Google.Builder.Polyline -> Kml*” que representa a classe “*Google.Builder.Polyline*”, ela possui os mesmo atributos da classe “*Kml*”, ou seja, ela possui os atributos “*args { url }*”, “*provider\_option*” e “*serviceObject*”. E também existe o caso onde esta classe possui atributos além dos da outra classe e nesse caso esses atributos são colocados na linha de baixo. Por exemplo a classe “*Google.Builder.Circle -> Kml*” que é a classe “*Google.Builder.Circle*”, além de possuir os atributos da classe *Kml*, ela possui o atributo “*args { lat, lng, radius }*”.

As seguintes explicações são válidas para o diagrama de herança na imagem 4:

- As linhas pretas indicam a organização da gema sendo que a classe *Gmaps* é a classe principal.
- Os retângulos normais representam as classes.
- A classe *Gmaps* possui os atributos *Builders*, *Objects* e *Google*, onde o *Google* possui os atributos *Builders* e *Objects*.
- As linhas vermelhas com pontas de seta representam a herança entre duas *classes*, onde a classe que está com a seta, é a classe que herda as características da classe na outra ponta da linha.

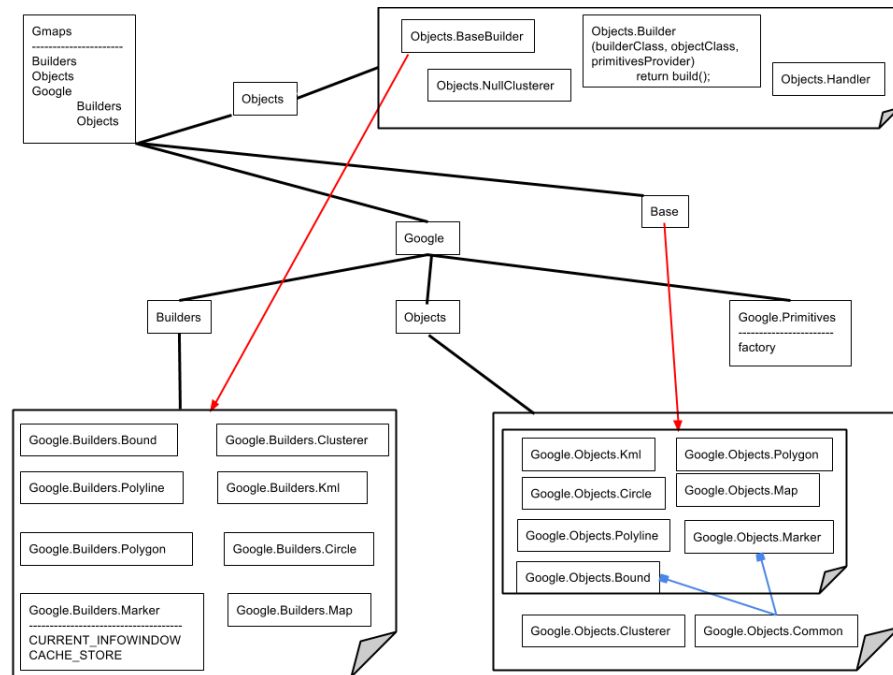


Figura 4 – Diagrama de Herança Google-Maps-For-Rails

- As linhas azuis com pontas de quadrado representam a inclusão de uma classe na outra, onde a classe que está com o quadrado, é a classe que incluí a classe que está na outra ponta da linha.
- Os retângulos que tem uma dobra no canto inferior direito representam um conjunto de classes, onde estas classes possuem uma característica em comum. Por exemplo as classes “Kml”, “Polygon”, “Polyline”, “Circle”, “Map”, “Marker” e “Bound” que são “Objects” do “Google”, estão em um conjunto onde todas elas herdam as características da classe “Base”.

## 4.2 Entendimento da Biblioteca do Ruby

Agora que realizamos a *engenharia reversa* da *gema* para construídos os diagramas de alto nível na seção anterior 4.1. Podemos apresentar nesta seção, a análise de algumas características dos diagramas, respondendo à algumas perguntas, como por exemplo:

- Quais as principais classes da *gema* ?
- Quais são as classes mais usadas na *gema* ?
- Quais são as classes de controle da *gema* ?
- A *gema* possui classes de configuração ? Quais são elas ?

Para a gema de exemplo, encontramos as seguintes características que serão listadas e explicadas logo a seguir.

- Apesar do “*GMaps*” ser a classe principal da gema, ela não é a mais importante, pois todas as funcionalidades da gema são controladas pela classe “*Handler*”. A única funcionalidade da classe “*GMaps*” é fazer a chamada para a criação de “*Handler*”, ou seja quando se requisita o método “*GMaps.build('Google')*” o método verifica se o objeto “*Handler*” já existe, e caso ele não exista, o “*GMaps*” faz a criação chamando o método “*new Gmaps.Objects.Handler(type, options)*”.
- “*Handler*” é a classe que controla todo o funcionamento da gema e basicamente ela possui dois momentos:
  - No primeiro momento ela prepara a estrutura da *gema* para criação e manipulação do mapa, criando e setando os objetos de configuração, como por exemplo criando o objeto “*Primitives*”.
  - No segundo momento ela cria o mapa com as configurações e permite a manipulação do mapa, possibilitando a criação e inserção de sobreposições como *circles* e *polylines*.
- A classe “*Primitives*” possui as definições que são comuns na gema, como por exemplo, é ela possui a definição do tipo “*Marker: google.maps.Marker*” que é a classe *Marker* do *Google Maps*.
- O atributo “*serviceObject*” de todas as classes de “*Builders*” do “*Google*”, representam o atributo que recebe o objeto do *Google Maps*.

### 4.3 Adaptações

Esta seção tem o objetivo de mostrar como podemos fazer adaptações em gemas, após realizar o processo de *engenharia reversa* e o processo de entendimento da gema. Inicialmente veremos quais os objetos que precisam ser incluídos, depois quais funcionalidades adicionais são necessárias, e por fim, analisaremos o quanto essas modificações afetaram o funcionamento original da gema.

Tendo a abstração de alto nível para a gema *Google-Maps-For-Rails*, podemos partir para a adaptação dela, ou seja, agora que temos alguns diagramas que nos auxiliam a visualizar o funcionamento geral da gema, podemos tentar acrescentar novas funcionalidades, analisando os locais das possíveis modificações e os impactos que essas mudanças podem causar.

A gema já possui sobreposições como *markers* e *circles*, mas até o momento não possui a funcionalidade de criar direções entre um ponto de origem e um ponto de destino.

A ideia é criar uma funcionalidade que receba como parâmetro um local de origem e um local de destino, e retorne como resultado uma sequência de ruas e direções a serem seguidas para ir do local de origem ao local de destino.

Para realizarmos essa modificação foi necessário consultar a *API* do *Direction Service*<sup>5</sup> (*Serviço de Direção*) do *Google*. Verificamos que seria necessário o uso de pelo menos 4 *classes* que serão listadas e explicadas logo a seguir:

- “*DirectionService*” (*google.maps.DirectionsService*) é a classe que tem o objetivo de requisitar e receber o caminho entre o local de origem e o local de destino.
- “*DirectionRender*” (*google.maps.DirectionsRenderer*) é a classe que tem o objetivo de rederizar no mapa o caminho, entre o local de origem e o local de destino.
- “*TravelMode*” (*google.maps.TravelMode*) é a classe que tem o objetivo de informar a forma como esse caminho deve ser percorrido, que pode ser caminhando (walking), de carro (driving), bicicleta (bicycling) e/ou por meios de locomoção públicos (transit).
- “*DirectionsStatus*” (*google.maps.DirectionsStatus*) é a classe que tem o objetivo de informar o *status* da requisição feita pela objeto da classe “*DirectionService*”.

Sabendo da necessidade da inclusão de “*DirectionService*”, “*DirectionRender*”, “*TravelMode*” e “*DirectionsStatus*”, decidimos que a primeira modificação na gema seria incluir estas quatro classe nas definições da classe “*Primitives*”. E isso foi feito da seguinte forma, como apresentado no código 4.1 logo abaixo.

```
1 @Gmaps.Google.Primitives = ->
2   factory = {
3     ...
4     directionSer: google.maps.DirectionsService
5     directionRen: google.maps.DirectionsRenderer
6     directionTM:  google.maps.TravelMode
7     directionSta: google.maps.DirectionsStatus
8     ...
```

Código 4.1 – Classe Primitives com atributo de Directions

Em seguida criamos quatro *classes* para o “*Google*”, sendo que duas são “*Builders*” de “*DirectionService*” e “*DirectionRender*”, e as outras duas são “*Objects*” também das classes “*DirectionService*” e “*DirectionRender*”. Para facilitar a compreensão, elaboramos o diagrama representado na imagem 5 para mostrar o local aonde inserimos as classes e

<sup>5</sup> Direction Service: <https://developers.google.com/maps/documentation/javascript/directions>

quais as dependências que elas possuem. No caso este diagrama é o mesmo diagrama de herança que desenvolvemos na *engenharia reversa*, mostrado na imagem 4, com a adição das quatro classes que são representadas por retângulos tracejados.

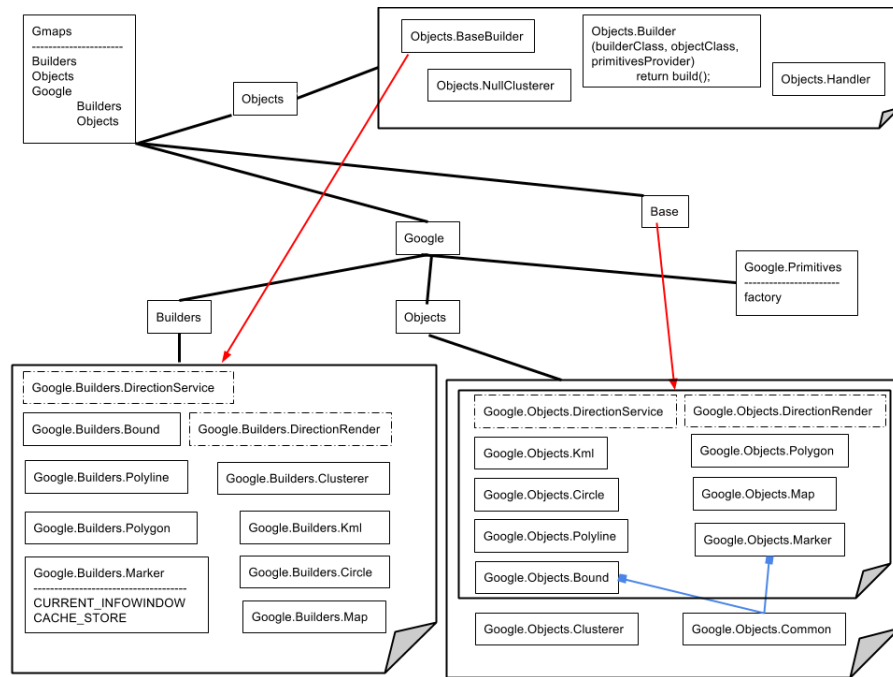


Figura 5 – Novo Diagrama de Herança Google-Maps-For-Rails

Agora que acrescentamos estas quatro classes na gema adaptada, devemos adicionar no “*Handler*”, novas funções para manipular essas classes. E neste caso, inserimos as funções “*addDirection()*” e “*calculate\_route()*” que podem ser vistas no código 4.2 que será explicado logo a seguir.

- Na linha “2” é adicionado a função “*addDirection*” que recebe como parâmetro “*direction\_data*”, que possui o informações do local de origem e local de destino que são obrigatórios para criar a direção, e “*provider\_options*” que pode conter as opções da forma como esse direção deve ser gerada. Essa função tem por objetivo criar as direções e colocá-las no mapas.
- Na linha “13” é adicionado a função “*calculate\_route*” com o parâmetro “*direction\_data*”, que tem por principal objtivo fazer a requisição para o *Google* de uma possível direção entre o local de origem ao local de destino.
- Da linha “3” a linha “10” é o conteúdo da função “*addDirection*”, onde se cria os atributos “*direction\_service*” e “*direction\_render*” para o “*Handler*”. Para cada um deste atributos, é atribuido o seu respectivo objeto do *Google Maps* com a



chamada da função “@\_builder(...)”. Depois é feito a chamada da função “*calculate\_route(...)*” para encontrar uma possível direção. E finalmente com o código “@direction\_render.getServiceObject().setMap(@getMap())”, a direção encontrada é colocado no mapa.

- Da linha “13” a linha “20” é o conteúdo da função “*calculate\_route(...)*”, onde inicialmente é colocado em uma variável local o *status Ok* de requisição que será utilizado para verificar se a requisição de direção foi feita com sucesso, e também é criada uma variável local para o *objeto DirectionRender* do *Google Maps*. Em seguida, através da chamada da função “*route(...)*” do *objeto DirectionService*. Nesta função os parâmetros de entrada são, o local de origem e o local de destino, e como resposta se recebe o *status* da requisição e o *response*. Neste caso, *response* pode ou não conter o caminho solução. Após a execução desta função, caso a resposta venha com o *status Ok*, o *objeto DirectionRender* recebe o caminho solução.

```

1  # return Direction object
2  addDirection: (direction_data, provider_options)->
3      if direction_data.origin? and direction_data.destination?
4          @direction_service = @_builder('DirectionService').build(
5              direction_data)
6          @direction_render = @_builder('DirectionRender').build(
7              provider_options)
8          @calculate_route( direction_data)
9          @direction_render.getServiceObject().setMap(@getMap())
10         @direction_render.getServiceObject()
11     else
12         alert "Need direction origin and destination\n and you inform\n
13         origin: " + direction.origin + "destination: " + direction.
14         destination
15
16 # calculate routes of direction
17 calculate_route: ( direction_data)->
18     statusOk = @direction_service.primitives().directionStas('OK')
19     direction_render_serviceObject = @direction_render.getServiceObject
20     ()
21     @direction_service.getServiceObject().route direction_data, (
22     response, status) ->
23     if status is statusOk
24         direction_render_serviceObject.setDirections response
25     else
26         alert "Could not find direction"

```

Código 4.2 – Funções adicionais do Handler

Analisando as modificações, percebemos que somente inserimos 4 classes e 2 funções. Com isso podemos perceber que as alterações que fizemos, não afetaram o funcionamento da gema, pois não foi necessário mexer no código que já existia.

## 4.4 Exemplo de uso de Google-Maps-for-Rails

Esta seção tem o objetivo de mostrar a utilização da gema de exemplo “*Google-Maps-for-Rails*”, que adaptamos no tutorial, em um projeto do *framework Ruby On Rails*.

Como exemplo de uso da gema “*Google-Maps-for-Rails*” adaptada, criamos o projeto “*DiseasesMap*”<sup>6</sup> que tem como objetivo representar a frequência de doenças no mapa do *Google*, utilizando sobreposições. Até o momento de término deste trabalho, essa função ainda não havia sido implementada, mas mesmo assim, fizemos o uso da funcionalidade de direções, somente para exemplificar o uso da gema modificada.

Inicialmente fizemos a instalação e inclusão da gema adaptada no arquivo *Gemfile* do projeto. Depois criamos uma estrutura básica de *model/view/controller* de “*locations*”. E para fazer o uso da função de direções utilizamos o código 4.3, explicado logo abaixo.

```
1 $(document).ready(->
2   handler = Gmaps.build("Google")
3   handler.buildMap
4     provider: {}
5     internal:
6       id: "map"
7   , ->
8     markers = handler.addMarkers([
9       lat: 0
10      lng: 0
11      picture:
12        url: "https://addons.cdn.mozilla.net/img/uploads/addon_icons
13        /13/13028-64.png"
14        width: 36
15        height: 36
16        infowindow: "hello!"
17    ])
18    handler.bounds.extendWith markers
19    handler.fitMapToBounds()
20    handler.addDirection( { origin: "Sao Paulo", destination: "Curitiba"
21    })
22  )
```

Código 4.3 – Exemplo CoffeeScript que Cria Mapa com Direção

<sup>6</sup> DiseasesMap : <https://github.com/toshikomura/DiseasesMap>

- Na linha “2” é feita a preparação da estrutura de configuração do mapa com a chamada “*GMaps.build('Google')*”, sendo feita a criação do *objeto Handler*, que é atribuída a variável local “*handler*”, juntamente com as outras configurações básicas que o mapa necessita.
- Na linha “3” é feita a chamada de “*handler.buildMap(...)*” que tem como função, fazer a criação do mapa a partir das configurações básicas já definidas. No caso, estas configurações básicas são definidas quando é feita a chamada de “*GMaps.build('Google')*”. São passados como parâmetros as variáveis, “*provider*” que no exemplo está vazio, e *internal* que define o “*id*” do mapa como “*map*”. Neste caso, o “*id*” serve para identificar o mapa a ser modificado.
- Na linha “7” é criada uma *function* determinada pelo símbolo “*->*”. Esta *function* somente será executada depois que a função “*handler.buildMap(...)*” terminar, ou seja, quando a criação do mapa terminar.

No exemplo do código essa função executa as seguintes operações:

Na linha “8” é feita a criação de um *marker* com a chamada da função “*handler.addMarker(...)*”, sendo passado como parâmetro, a sua posição que no caso é (0,0) definido “*lat*” e “*lng*”, a sua imagem definida por “*picture*”, e sua informação definido por “*infowindow*”.

Na linha “17” é feita a extensão de fronteiras incluindo o novo *marker* com a chamada da função “*handler.bounds.extendWith(...)*”, sendo passado como parâmetro o *marker* criado anteriormente.

Na linha “19” é feita a criação de direções com a chamada da função “*handler.addDirection(...)*” que incluímos no “*Handler*”, sendo passado como parâmetro, um local de origem definido por “*origin: ‘São Paulo’*”, e um local de destino definido por “*destination: ‘Curitiba’*”.

E para mostrar o mapa na view de “*locations*” adicionamos o código mostrado em 4.4 que é parte do código da *view*, explicado logo a seguir.

```

1 <h1>Listing locations</h1>
2 ...
3 <div style='width: 800px;'>
4   <div id="map" style='width: 800px; height: 400px;'></div>
5 </div>
6 ...

```

Código 4.4 – Exemplo Locations view que Cria Mapa com Direção

- Na linha “1” com a tag `<h1>...</h1>` é definido como título principal da *view* o texto “*Listing locations*”.

- Os “...” indica que existe código, mas por simplificação na explicação, ele não foi mostrado.
- Da linha “3” a “5” é definido uma *div* com “800px” de largura. Dentro dessa *div*, é definido o local para a criação do mapa com “800px” de largura e “400px” de altura. No caso, o local de criação do mapa, é referenciado pelo atributo *id* que é o mesmo *id* utilizado no código 4.3 na linha “10”.

Como resultado ao se acessar o *index* de locations, obtemos como resultado a imagem 6. Neste caso, a nossa gema adaptada com a nova funcionalidade de direções, funcionou corretamente, pois o caminho mostrado é entre “São Paulo” e “Curitiba”, como requisitamos na linha “24” do código 4.3.

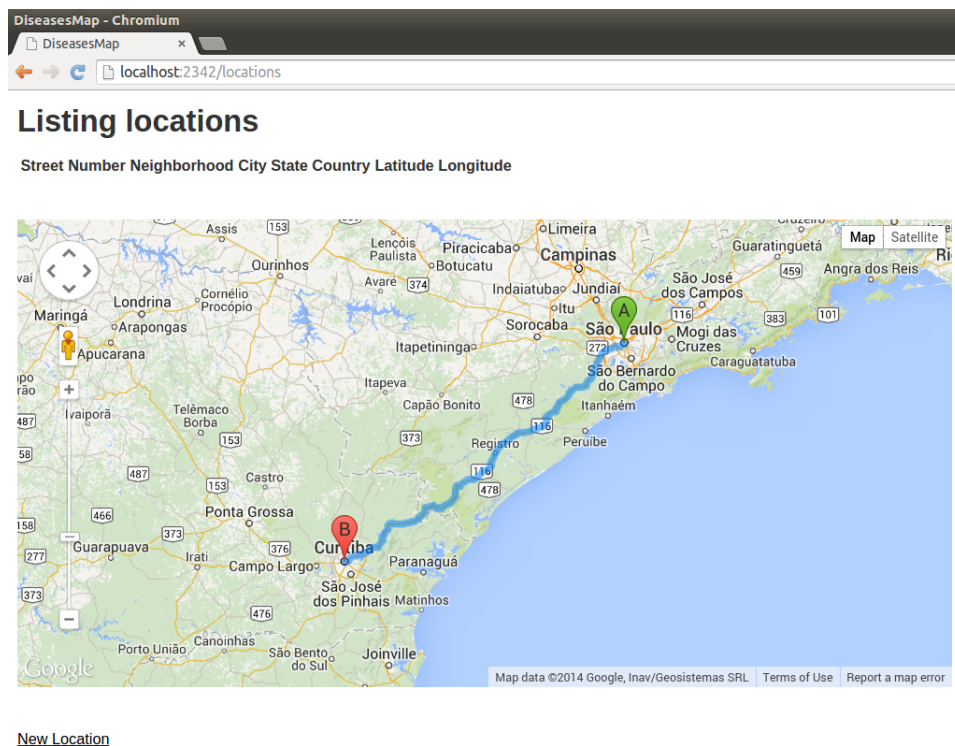


Figura 6 – Caminho entre São Paulo e Curitiba

## 5 Conclusão

Neste trabalho apresentamos alguns conceitos básicos sobre bibliotecas, descrevendo a importância de seu uso no capítulo ???. Depois dedicamos um pouco do tempo no capítulo ??? para falar sobre as bibliotecas do *Ruby*, e a possibilidade de se utilizar chaves para se ter segurança na instalação destas bibliotecas.

Percebendo que somente a utilização de bibliotecas de terceiros, as vezes não é o suficiente para se ter economia de tempo, passamos para um outro nível, aonde possibilitamos uma equipe ou mesmo somente uma pessoa a desenvolver a sua própria biblioteca quando uma certa funcionalidade é muito utilizada em vários tipos de projetos.

Contudo nos capítulos 3 e 4, apresentamos um tutorial básico de como se pode criar e modificar uma biblioteca do *Ruby On Rails*, descrevendo em detalhes os passos de implementação que devem ser seguidos para se ter mais chances de conseguir sucesso no projeto. Nestes capítulos, também detalhamos os comandos utilizados durante o processo de desenvolvimento, apresentando exemplos para facilitar a compreensão do tutorial.

Acreditamos que com a apresentação deste trabalho, uma equipe dependendo das suas necessidades tem a possibilidade de desenvolver do zero ou adaptar bibliotecas do *Ruby On Rails*, podendo assim economizar tempo de desenvolvimento em projetos futuros.

Para este trabalho no exemplo de criação de *gemas*, desenvolvemos uma *gema* simples de tradução, mostrando somente alguns detalhes da linguagem *Ruby*, e por esse motivo para trabalhos futuros, poderia ser criada uma nova biblioteca ou mesmo fazer uma adaptação da *gema* de exemplo para mostrar mais conceitos da linguagem. Também para a biblioteca *Google-Maps-For-Rails* adaptada para aceitar a funcionalidade de gerar direções, poderia ser feito o incremento de mais funcionalidades, como por exemplo, a inclusão da escolha do tipo de locomoção a ser utilizada no percurso e a possibilidade de adicionar locais intermediários no caminho.

# Referências

- 1 COMMUNITY, R. *Ruby Libraries*. 2014. Ruby Site - Libraries. Disponível em: <http://www.ruby-lang.org/en/libraries/>. Acesso em: 17 out. 2014. Citado 2 vezes nas páginas 8 e 9.
- 2 COMMUNITY, R. *RubyGems About*. 2014. RubyGems Site - About. Disponível em: <http://rubygems.org/pages/about>. Acesso em: 21 out. 2014. Citado na página 9.
- 3 COMMUNITY, R. *Ruby Security*. 2014. Ruby Site - Security. Disponível em: <http://www.ruby-lang.org/en/security/>. Acesso em: 22 out. 2014. Citado na página 9.
- 4 COMMUNITY, R. *OpenSSL Severe Vulnerability in TLS Heartbeat Extension (CVE-2014-0160)*. 2014. Ruby Site - News. Disponível em: <http://www.ruby-lang.org/en/news/2014/04/10/severe-openssl-vulnerability/>. Acesso em: 22 out. 2014. Citado na página 10.
- 5 COMMUNITY, R. *RubyGems Guides Security*. 2014. RubyGems Site - Guides - Security. Disponível em: <http://guides.rubygems.org/security/>. Acesso em: 21 out. 2014. Citado na página 10.
- 6 COMMUNITY, R. *RubyGems Guides What is a gem*. 2014. RubyGems Site - Guides - What is a gem. Disponível em: <http://guides.rubygems.org/what-is-a-gem/>. Acesso em: 23 out. 2014. Citado na página 12.
- 7 ROTH, D. R. B. *Google-Maps-For-Rails*. 2014. Google-Maps-For-Rails. Disponível em: <http://github.com/apneadiving/Google-Maps-for-Rails>. Acesso em: 30 out. 2014. Citado na página 25.
- 8 WEXELBLAT, R. *History of Programming Languages*. 1. ed. New York, NY: Academic Press: ACM Monograph Series, 1981. Citado na página 41.
- 9 COMMUNITY, R. *Ruby*. 2014. Ruby Community. Disponível em: <http://www.ruby-lang.org/>. Acesso em: 10 out. 2014. Citado 2 vezes nas páginas 44 e 45.
- 10 MATSUMOTO, Y. M. *Re: More code browsing questions*. 2000. Ruby Talk Main List. Disponível em: <http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/2773>. Acesso em: 17 out. 2014. Citado na página 44.
- 11 MATSUMOTO, Y. M. *About Ruby*. 2014. Ruby Site - About Ruby. Disponível em: <http://www.ruby-lang.org/en/about/>. Acesso em: 17 out. 2014. Citado na página 44.
- 12 VMWARE, I. *VMware® Player*. 2014. VMware® Player. Disponível em: <http://www.vmware.com/products/player>. Acesso em: 23 out. 2014. Citado na página 48.
- 13 SEGUIN, M. P. W. E. *RVM*. 2014. RVM. Disponível em: <http://rvm.io/>. Acesso em: 22 out. 2014. Citado na página 48.

- 
- 14 SEGUIN, M. P. W. E. *RVM*. 2014. RVM. Disponível em: <[http://rvm.io/rvm-  
/about](http://rvm.io/rvm-<br/>/about)>. Acesso em: 22 out. 2014. Citado na página 48.
- 15 HANSSON, D. H. *Ruby On Rails*. 2014. Ruby On Rails. Disponível em: <[http:-  
//rubyonrails.org/](http://<br/>/rubyonrails.org/)>. Acesso em: 22 out. 2014. Citado 2 vezes nas páginas 48 e 49.
- 16 TORVALDS, L. *Git*. 2014. Git. Disponível em: <<http://git-scm.com/>>. Acesso em: 23 out. 2014. Citado na página 49.
- 17 COMMUNITY, R. *IRB*. 2014. Ruby Community. Disponível em: <[http://www-  
.ruby-doc.org/stdlib-2.0/libdoc/irb/rdoc/IRB.html](http://www-<br/>.ruby-doc.org/stdlib-2.0/libdoc/irb/rdoc/IRB.html)>. Acesso em: 28 out. 2014. Citado na página 53.
- 18 SVENNERBERG, G. *Beginning Google Maps API 3*. 2. ed. New York, NY 10013: Apress, 2010. Citado na página 55.

## Apêndices



# APÊNDICE A – História e Classificação de Bibliotecas

Este capítulo tem o objetivo de complementar os conceitos sobre bibliotecas. Na seção A.1, será apresentado um pouco sobre a histórias das bibliotecas, e depois na seção A.2, será apresentado as formas de classificação que existem para as bibliotecas.

## A.1 História

Os primeiros conceitos que tinham proximidades com a definição de bibliotecas apareceram publicamente com o *software* “COMPOOL” (*Communication Pool*) desenvolvido em JOVIAL, que é uma linguagem de alto nível parecido com ALGOL, mas especializada para desenvolvimento de sistemas embarcados. O “COMPOOL” tinha como propósito compartilhar os dados do sistema entre vários progrmas, fornecendo assim informação centralizada. Com essa visão o “COMPOOL” seguiu os princípios da ciência da computação, separando interesses e escondendo informações [8].

As linguagens COBOL e FORTRAN também possuíam uma prévia implementação do sistema de bibliotecas. O COBOL em 1959 tinha a capacidade de comportar um sistema primitivo de bibliotecas, mas segundo Jean Sammet esses sistema era inadequado. Já o FORTRAN possuía um sistema mais moderno, onde ele permitia que os subprogramas poderiam ser compilados de forma independente um dos outros, mas com essa nova funcionalidade o compilador acabou ficando mais fraco com relação a ligação, pois com essa possibilidade adicionada ele não conseguia fazer a verificação de tipos entre os subprogramas [8].

Por fim chegando no ano de 1965 com a linguagem Simula 67, que foi a primeira linguagem de programação orientada a objetos que permitia a inclusão de suas classes em arquivos de bibliotecas. Ela também permitia que os arquivos de bibliotecas fossem utilizadas em tempo de compilação para complementar outros programas [8].

## A.2 Classificação

Esta seção tem o objetivo de apresentar as formas de classificação das bibliotecas. Estas classificações podem ser definidas pela maneira como elas são ligadas, explicado na sub-seção ??, pelo momento que elas são ligadas, explicado na sub-seção ??, e pela forma como elas são compartilhadas, explicado na sub-seção ??.

### A.2.1 Formas de ligamento

O processo de ligamento implica em associar uma biblioteca a um programa ou outra biblioteca, ou seja, é nesse momento que as implementações das funcionalidades são realmente ligadas ao programa ou outra biblioteca.

Esse processo de ligamento pode ser feito de 3 formas diferentes:

- **Tradicional** que significa que os dados da biblioteca são copiados para o executável do programa ou outra biblioteca.
- **Dinâmica** que significa que ao invés de copiar os dados da biblioteca para o executável, é somente feito uma referência do arquivo da biblioteca. Neste caso o ligador não tem tanto trabalho na compilação, pois ele somente grava a biblioteca a ser utilizada e um índice para ela, passando todo o trabalho para o momento onde a aplicação é carregada para a memória ou para o momento que a aplicação requisita a biblioteca.
- **Remoto** que significa que a biblioteca vai ser carregada por chamadas de procedimento remotos, ou seja, a biblioteca pode ser carregada mesmo não estando na mesma máquina do programa ou biblioteca, pois ela é carregada pela rede.

### A.2.2 Momentos de ligação

O momento de ligação diz respeito ao momento em que a biblioteca vai ser carregada na memória e para esse caso existem 2 formas:

- **Carregamento em tempo de carregamento** que significa que a biblioteca vai ser carregada na memória quando a aplicação também estiver sendo carregada.
- **Carregamento dinâmico ou atrasado** que significa que a biblioteca só vai ser carregada na memória quando a aplicação requisitar o seu carregamento e isso é feito em tempo de execução.

### A.2.3 Formas de compartilhamento

As formas de compartilhamento de bibliotecas se divide em 2 conceitos. Sendo que o primeiro se refere ao compartilhamento de código em disco entre os vários programas. E o segundo se refere ao compartilhamento da biblioteca na memória em tempo de execução.

O compartilhamento em memória traz a vantagem de que dois ou mais programas podem compartilhar o acesso ao mesmo código da biblioteca na memória, com isso se evita que a mesma biblioteca seja colocada mais de uma vez na memória.

Por exemplo para o segundo conceito, suponha que os programas  $P1$  e  $P2$  necessitem de uma biblioteca  $B$  que é carregada no *momento de carregamento*. Suponha agora que executamos o programa  $P1$  e depois de um tempo executamos o programa  $P2$ , desta forma primeiramente o programa  $P1$  e a biblioteca  $B$  são carregadas na memória. Como também executamos o programa  $P2$ , ele também é carregado na memória, mas sem a necessidade de carregar a biblioteca  $B$ , pois ela já havia sido carregada anteriormente.

Pensando pelo outro lado, o compartilhamento de memória pode ser um pouco prejudicial ao desempenho, pois essa biblioteca deve ser escrita para executar em um ambiente *multi-tarefa* e isso pode causar alguns atrasos.

## APÊNDICE B – Conceitos e História do Ruby

Este capítulo tem o objetivo de mostrar alguns conceitos básicos sobre o *Ruby*, suas história, suas bibliotecas, riscos e procedimento de segurança que existem. Na seção B.1 iremos apresentar alguns conceitos da linguagem *Ruby*, depois vamos falar um pouco sobre sua história na seção B.2, e no fim vamos ver a importância de uma *API* na seção B.3.

### B.1 Ruby

*Ruby* é uma linguagem de programação dinâmica de código aberto com foco na simplicidade e produtividade, tem uma sintaxe elegante, é natural de ler e escrever, e foi inventada por Yukihiro “Matz” Matsumoto<sup>1</sup> que tentou cuidadosamente criar uma linguagem balaceada, misturando as suas linguagens favoritas ( *Perl*, *Smalltalk*, *Eiffel*, *Ada*, e *Lisp*) [9].

Yukihiro “Matz” Matsumoto sempre enfatiza que ele estava “tentando fazer o *Ruby* natural, não simples” e também com toda a sua experiência acrescenta que “*Ruby* é simples na aparência, mas é muito complexo internamente, assim como o corpo humano.”. “Matz” também lembra que procurava uma linguagem de script que fosse mais poderosa que *Perl* e mais orientada a objeto do que *Python*, tentando encontrar a sintaxe ideal nas outras linguagens [10] [11].

Em muitas linguagens números e outros tipos primitivos não são *objetos*, no entanto no *Ruby* cada pedaço de informação possui propriedades e ações, ou seja, tudo é *objeto*. Neste caso o *Ruby* possui esse tipo de característica, pois ele seguiu a influência do *SmallTalk*, onde todos seus tipos, inclusive os mais primitivos como inteiros e booleanos são objetos.

Logo abaixo no código *Ruby* B.1 segue um exemplo onde construímos um *método* “tipo” para o *objeto* inteiro “5”. Suponho que se requisite o *método* “5.tipo” o retorno seria “*Tipo inteiro*”.

```
1 5.tipo { print "Tipo inteiro" }
```

Código B.1 – Tudo é objeto para o Ruby

<sup>1</sup> Yukihiro “Matz” Matsumoto: <http://www.rubyist.net/~matz/>

O *Ruby* também é flexível, pois possibilita remover ou redefinir seu próprio *core*, por exemplo no código B.2 acrescentamos na classe “*Numeric*” o método “vezes()” que possui a mesma característica do operador “\*”. Ao se fazer a execução deste código a variável “y” recebe o valor “10” como resultado da operação “5.vezes 2”.

```
1 class Numeric
2   def vezes(x)
3     self.*(x)
4   end
5 end
6
7 y = 5.vezes 2
```

Código B.2 – Ruby flexível

Apesar de ser flexível esta linguagem possui algumas convenções com por exemplo no código B.3, a definição de “var” ou qualquer outro nome de variável dependendo do contexto sozinha deve ser uma variável local, “@var” deve ser uma instância de uma variável por causa do caracter “@” como primeiro caracter e “\$var” deve ser uma variável global por causa do caracter “\$” como primeiro caracter.

```
1 var = 2 // variavel local
2 @var = 4 // instancia de uma variavel
3 $var = 6 // variavel global
```

Código B.3 – Convenção Ruby

## B.2 História

O *Ruby* foi criado em 24 de fevereiro de 1993 e o com o passar do tempo foi ganhando espaço na comunidade de desenvolvedores, chegando em 2006 a ter uma grande massa de aceitação, possuindo várias conferências de grupos de usuários ativos pelas principais cidades do mundo [9].

O nome *Ruby* veio a partir de uma conversa de chat entre *Matsumoto* e *Keiju Ishitsuka*, antes mesmo de qualquer linha de código ser escrita. Inicialmente dois nomes foram propostos “*Coral*” e “*Ruby*”. *Matsumoto* escolheu o segundo em um e-mail mais tarde. E depois de um tempo, após já ter escolhido o nome, ele percebeu no fato de que *Ruby* era o nome da *birthstone* (pedra preciosa que simboliza o mês de nascimento) de um de seus colegas.

Sua primeira versão foi anunciada em 21 de dezembro de 1995 na “*Japanese domestic newsgroups*”. Subseqüencialmente em dois dias foram lançadas mais três versões juntamente com a “*Japanese-Language ruby-list main-list*” (*RubyTalk*).

*Ruby-Talk* <sup>2</sup> a primeira *lista de discussão da Ruby* <sup>3</sup> chegou a possuir em média 200 mensagens por dia em 2006. E com o passar dos últimos anos, essa média veio a cair, pois o crescimento da comunidade obrigou a criação de grupos específicos, empurrando os usuários da lista principal para as lista específica.

## B.3 API

Uma *Application Programming Interface (API)* possui como principal função definir de forma simplificada o acesso as funcionalidades de um certo componente, informando ao usuário somente para que serve cada função e como usá-la, indicando os parâmetros de entrada e saída com os seus respectivos tipos.

Basicamente uma *API* diminui a complexidade para o seu usuário. Por exemplo por analogia a *API* de um elevador seria o seu painel, e a entrada seria o andar que se deseja ir. Desta forma supondo que um usuário deseja ir do segundo para o décimo andar, bastaria ele apertar o botão para chamar o elevador. Depois quando o elevador chegasse bastaria ele entrar e apertar o botão (10), e mesmo sem ter nenhum conhecimento mecânico do elevador, ele conseguiria obter como saída o décimo andar.

A *API* do *Ruby* não é diferente, ou seja, não é necessário que se tenha conhecimento do funcionamento do *core* do *Ruby* <sup>4</sup>, basta saber para que serve cada uma de suas funções disponibilizadas, seus parâmetro de entrada e saída com seus respectivos tipos.

---

<sup>2</sup> Ruby-Talk: <https://www.ruby-forum.com/>

<sup>3</sup> Listas de discussões do Ruby: <https://www.ruby-lang.org/en/community/mailling-lists/>

<sup>4</sup> Core do Ruby: <http://www.ruby-doc.org/core-2.1.3/>

## APÊNDICE C – Uso da Ferramenta *gem*

A instalação do *gem* pode ser feita pelo terminal, executando “*sudo apt-get install gem*” com privilégios de administrador ou acessando <https://rubygems.org/pages/download/> e baixando a última versão do *RubyGems*. Para o caso de baixar a instalação, deve-se descompactar o pacote, entrar no diretório e executar “*ruby setup.rb*” com privilégios de administrador.

Caso haja uma versão do *gem* instalada, pode-se fazer a atualização para a última versão executando “*gem update -system*” com privilégios de administrador.

Se ocorrer algum problema no momento da instalação, atualização ou se necessitar de mais alguma informações, se pode executar “*ruby setup.rb -help*” para obter ajuda.

A partir da versão “1.9” do *Ruby* esse processo de instalação não é mais necessário, pois o *gem* vem por *default* instalado junto com *Ruby*, mas para as versões anteriores é necessário fazer a instalação manualmente.

Após a instalação, a ferramenta *gem* nos auxilia a fazer a busca de gemas com o comando “*gem search 'nome da gema'*”. Esta ferramenta, também nos permite fazer a instalação de gemas utilizando o comando “*gem install 'nome da gema'*” ou com o comando “*gem install 'nome da gema'.gem*” quando o código da gema já está na nossa máquina.

## APÊNDICE D – Ferramentas Utilizadas

Este capítulo tem o objetivo de apresentar as ferramentas que foram utilizadas no tutorial. Na seção D.1, apresentaremos a ferramenta *VMware® Player*, em seguida na seção D.2, apresentaremos a ferramenta *RVM*, depois na seção D.3, apresentaremos o *framework Ruby On Rails*, e por fim na seção D.4, apresentaremos a ferramenta *git*.

### D.1 VMware® Player

*VMware® Player* é um aplicativo de virtualização de *desktop* que pode rodar um ou mais sistemas operacionais ao mesmo tempo no mesmo computador sem a necessidade de reiniciar a máquina. Possui uma interface fácil de usar, suporte a vários sistemas operacionais, como por exemplo *Windows*, *MAC* e *Linux* e é portátil [12].

Com o *VMware® Player* pode-se fazer o compartilhamento de vários recursos, como por exemplo pode-se fazer a transferência de arquivos do sistema operacional virtual com o sistema operacional que está rodando na máquina física.

### D.2 RVM

*Ruby Version Manager*<sup>1</sup> é uma ferramenta de linha de comando que permite facilmente instalar, gerenciar e trabalhar com múltiplos ambientes do *Ruby* para interpretar um certo conjunto de gemas [13].

Wayne E. Seguin<sup>2</sup> iniciou o projeto do *RVM* em outubro de 2007 e a partir de então com uma considerável experiência programando em *Bash*, *Ruby* e outras linguagens obteve o conhecimento suficiente para criar o gerenciador [14].

### D.3 Ruby On Rails

*Ruby On Rails*<sup>3</sup> é um *web framework* de código aberto que tem por finalidade facilitar a programação visando a produtividade sustentável. Este *framework* permite escrever códigos bem estruturados favorecendo a manutenção de aplicações [15].

---

<sup>1</sup> RVM: <http://rvm.io/>

<sup>2</sup> Wayne E. Seguin: <https://github.com/wayneeseguin>

<sup>3</sup> Ruby On Rails: <http://rubyonrails.org/>



O *Rails* foi criado em 2003 por *David Heinemeier Hansson*<sup>4</sup> e desde então é estendido pela equipe do *Rails Core Team*<sup>5</sup> e mais de 3.400 usuários [15].

## D.4 Git

*Git*<sup>6</sup> é um sistema distribuído de controle de versão livre e de código aberto, desenhado para controlar projetos pequenos e grandes com rapidez e eficiência. É uma ferramenta fácil de manipular com alto desempenho [16].

Por ironia do destino o *git* foi criado em 2005 por *Linus Torvalds*<sup>7</sup> graças ao fim da relação entre a comunidade que desenvolvia o *Linux* e a companhia que desenvolvia o *BitKeeper*, ferramenta que até aquele ano fazia o gerenciamento de código do *Linux*. Sem uma ferramenta *SCM* (*Source Code Management*), *Torvalds* resolveu desenvolver uma ferramenta parecida com o *BitKeeper* que possuiria mais velocidade, *design* simples, grande suporte para *non-linear development* (centenas de *branches*), completamente distribuído e capacidade de controlar grandes projeto com grandes quantidades de dados.

---

<sup>4</sup> David Heinemeier Hansson: <http://david.heinemeierhansson.com/>

<sup>5</sup> Rails Core Team: <http://rubyonrails.org/core/>

<sup>6</sup> Git: <http://git-scm.com/>

<sup>7</sup> Linus Torvalds: <http://torvalds-family.blogspot.com.br/>

## APÊNDICE E – Preparação do Ambiente

Para seguir com este trabalho, será necessário a instalação de alguns softwares para que se possa realizar o passo-a-passo de como criar ou adaptar uma *gema* do *Ruby*. Inicialmente por comodidade utilizaremos o sistema operacional *Ubuntu 12.04 LTS*, pois a comunidade prometeu manter essa versão por pelo menos 5 anos, com o *Ruby 1.9.3p547 (2014-05-14 revision 45962) [i686-linux]* que era a versão mais recente do *Ruby* no momento de início desse trabalho, e o *Rails 3.2.12* que era uma versão que já tínhamos experiência trabalhando no projeto *Agendador*.

Primeiramente deve-se instalar o *Ubuntu 12.04* no *VMware® Player*, e no nosso caso foi utilizado o *VMware Player 6.0.3 build-1895310*. Após a instalação do *Ubuntu* deve-se logar no *Ubuntu*, acessar um *terminal*, e fazer a instalação do *RVM 1.25.28* com os seguinte comandos no código E.1 que serão explicados logo a seguir:

```
1 # Precisa ser root para instalar os outros softwares
2 sudo -i
3 # digite a senha de root
4
5 # Instalar Curl, pois o RVM depende dele
6 apt-get install curl
7
8 # Baixa o script para instalar o RVM
9 wget https://raw.githubusercontent.com/wayneeseguin/rvm/master/binscripts/rvm-installer
10
11 # Executa o script para instalar o RVM
12 bash rvm-installer
```

Código E.1 – Instala RVM

- O comando “*sudo -i*” na linha “2” é necessário para acessar o usuário root, pois para instalar os outros *softwares* é preciso ser um administrador do sistema.
- O comando “*apt-get install curl*” na linha “6” é necessário pois o script de instalação do *RVM* depende do *curl* para executar corretamente.
- O comando “*wget https://raw.githubusercontent.com/wayneeseguin/rvm/master/binscripts/rvm-installer*” na linha “9” é necessário para baixar o script (“*rvm-installer*”) de instalação do *RVM*.
- O comando “*bash rvm-installer*” na linha “12” é necessário para instalar o *RVM*.

Ao terminar de instalar o *RVM* deve-se fazer a instalação do *Ruby* que pode ser feita com a sequência de comandos do código E.2, onde cada comando será explicado logo em seguida:

```
1 # Carrega o RVM
2 source "/usr/local/rvm/scripts/rvm"
3
4 # Instala as dependencias do RVM
5 rvm requirements
6
7 # Instala o Ruby 1.9.3
8 rvm install 1.9.3
9
10 # Seta o Ruby 1.9.3 como default do RVM
11 rvm --default use 1.9.3
12
13 # Informa o RVM para usar o Ruby 1.9.3
14 rvm use 1.9.3
```

Código E.2 – Instala Ruby

- O comando “`source "/usr/local/rvm/scripts/rvm"`” serve para carregar o código do *RVM*.
- O comando “`rvm requirements`” serve para instalar as dependências do *RVM* caso elas ainda não estejam instaladas.
- O comando “`rvm install 1.9.3`” serve para fazer a instalação do *Ruby* versão 1.9.3.
- O comando “`rvm --default use 1.9.3`” serve para informar o *RVM* para usar o *Ruby* 1.9.3 como default. Essa é uma medida preventiva, pois podem existir outras versões do *Ruby* instaladas.
- O comando “`rvm use 1.9.3`” serve para informar o *RVM* para usar o *Ruby* 1.9.3.

Também pode-se configurar as variáveis de ambiente do *RVM* para não precisar a todo momento executar o comando “`source "/usr/local/rvm/scripts/rvm"`”, e isso pode ser feito executando o seguinte código E.3 que é exibido e explicado logo abaixo:

```
1 echo '
2     # This loads RVM
3     [[ -s "/usr/local/rvm/scripts/rvm" ]] && source "/usr/local/rvm/scripts/rvm"
4     source /etc/profile
5     rvm use 1.9.3
6 ' >> ~/.bashrc
```

Código E.3 – Configura Variáveis RVM

- O comando “`echo '...' » ~/.bashrc`” pega todo o código, aqui representado por “...” e insere no final do arquivo “`~/.bashrc`”. Uma alternativa seria somente copiar o código dentro do “`echo`” e colar no final do arquivo “`~/.bashrc`”.

E depois da execução de todas essas instalações, ainda devemos instalar as gemas essenciais, “*rails*” e “*bundle*”, executando os seguintes comandos no código E.4 explicado logo a seguir:

```
1 gem install rails --version "3.2.12"  
2  
3 gem install bundle
```

Código E.4 – Instala Gemas Essenciais

- O comando “ *gem install rails --version '3.2.12'* ” faz a instalação da gema *rails* versão *3.2.12*.
- O comando “*gem install bundle*” faz a instalação da gema *bundle*.

## APÊNDICE F – IRB

O *IRB*<sup>1</sup> (*Interactive Ruby Shell*) é uma ferramenta do *Ruby* que serve para executar expressões interativamente, fazendo a leitura da entrada padrão [17].

Caso se deseje fazer os testes de uma gema manualmente sem a necessidade de incluí-la em um projeto, podemos fazer o uso do “*IRB*”, chamando o comando “*irb*”.

Um exemplo de uso do *IRB* pode ser visto no código F.1 abaixo, explicado em mais detalhes na listagem abaixo.

```

1 gtk10@ubuntu:~$ irb
2 1.9.3-p547 :001 > 1 + 1
3 => 2
4 1.9.3-p547 :002 > 1 == 1
5 => true
6 1.9.3-p547 :003 > def hello( name)
7 1.9.3-p547 :004?>   print "Hello " + name
8 1.9.3-p547 :005?>   end
9 => nil
10 1.9.3-p547 :006 > hello("Maria")
11 Hello Maria => nil

```

Código F.1 – Exemplo de uso do IRB

- Primeiramente na linha “1” é feita a chamada da ferramenta *IRB* com o comando “*irb*” no terminal.
- Depois na linha “2” é requisitado a soma entre “*1 + 1*” resultando em “2” na linha “3”.
- Em seguida na linha “4” é verificado se “*1 == 1*” resultando em “*true*” na linha “5”.
- E no fim entre as linhas “6” e “8” é criado uma função chamada de “*hello*” com o parâmetro “*name*” e ao se chamar essa função é devolvido na tela “*Hello*” mais o parâmetro passado para a função. O resultado pode ser visto quando se requisita “*hello(“Maria”)*” na linha “10”, obtendo como resultado “*Hello Maria*” na linha “11”.

No nosso exemplo da gema “*gemtranslatetoenglish*” fizemos alguns testes simples mostrados na código F.2 explicado com mais detalhes nos itens abaixo.

<sup>1</sup> IRB: <http://www.ruby-doc.org/stdlib-2.0/libdoc/irb/rdoc/IRB.html>

```
1 gtk10@ubuntu:~$ irb
2 1.9.3-p547 :001 > require 'action_controller'
3 => true
4 1.9.3-p547 :002 > require 'gemtranslatetoenglish'
5 => true
6 1.9.3-p547 :003 > Gemtranslatetoenglish::Helpers::Translatetoenglish.
   instance_method_names
7 => ["translate"]
8 1.9.3-p547 :004 > include Gemtranslatetoenglish::Helpers::Translatetoenglish
9 => Object
10 1.9.3-p547 :005 > Gemtranslatetoenglish::Helpers::Translatetoenglish.translate('Oi')
11 => "HELLO "
```

Código F.2 – Teste IRB da gema gemtranslatetoenglish

- Primeiramente na linha “1” é feita a chamada da ferramenta *IRB* com o comando “*irb*” no terminal.
- Na linha “2” é executado o comando “ *require 'action\_controller'* ” para buscar a gema “*ActionController*” necessária no uso da nossa gema de exemplo quando evitamos digitar a *PATH* completa na “*view*”.
- Na linha “4” é executado o comando “ *require 'gemtranslatetoenglish'* ” para buscar a nossa gema de exemplo.
- Na linha “6” é executado o comando “ *instance\_method\_names*” para verificar se o nosso método *translate()* existe.
- Na linha “8” é executado o comando “*include*” para incluir as funções do módulo “*Translatetoenglish*”.
- Na linha “10” é executado o comando “*translate('Oi')*” para verificar se a função funciona como o esperado.
- E no fim na linha “11” podemos verificar que a função *translate()* funcionou corretamente, pois obtemos como resultado a palavra “*HELLO* ”.

## APÊNDICE G – API do Google Maps

Para fazer a adaptação da gema foi necessário fazer um estudo sobre como utilizar a *API* do *Google* e para isso foi utilizado como base o livro *Beginning Google Maps API 3* [18] e a *Google Maps API V3*<sup>1</sup>, onde ambos se complementam ensinando os passos básicos para criar e manipular mapas do *Google*.

O *Google Maps* e sua respectiva *API* foram criadas por dois irmãos *Lars* e *Jens Rasmussen*, cofundadores da “*Where 2 Technologies*”, companhia dedicada a criação de mapas que foi comprada pelo *Google* em 2004 [18].

Até o início do ano 2005 a rederização de mapas pela rede possuía um alto custo, necessitando de servidores altamente equipados para suportar a carga de trabalho. Mas em Fevereiro de 2005 através de um *post* em seu *blog*, o *Google* anunciou uma nova solução de rederização, possibilitando ao usuário interagir com um mapa em uma página *web* [18].

Depois de fazer o lançamento da nova forma para criar mapas, o *Google* percebeu que muitos desenvolvedores gostariam de incorporar essa nova solução em seus projetos, e por esse movtio em Junho de 2005 anunciou a primeira versão pública da *API* do *Google Maps* [18].

O *Google Maps* funciona de uma forma bem simples fazendo a criação e a manipulação do mapa por meio de *HTML*, *CSS* e *Javascript*. Basicamente o usuário por meio do *browser* requisita algum local do mapa informando a coordenada e o zoon desejado. Desta forma o servidor retorna a imagem do mapa que representa a posição requisitada [18].

---

<sup>1</sup> Google Maps API V3: <https://developers.google.com/maps/>