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

Lista de Códigos

Sumário

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. Isso ocorria, porque era feito a transferência de um mapa inteiro. Também como consquência da grande quantidade de dados transferidos, existia um alto custo no cliente para rederizar a figura.

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. Estes pedaços do mapa, poderiam ser requisitados um a um e colocados lado a lado para formar uma parte ou um mapa completo]1]. 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 o problema da rederização, pois a quantidade de dados para se rederizar, foi reduzida razoavelmente.

Muitas companhias, percebendo o grande potêncial da ferramenta, adotaram soluções similares 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.

Muitas empresas decidiram se mobilizar na procura de uma forma de incorporar o *Google Maps* em suas aplicações, pois perceberam que a ferramenta possuía funcionalidade muito úteis para o dia-a-dia. 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*, apresentando detalhes de como importar e utilizar as funções da ferramenta. Exemplos incluem a criação de mapas e marcadores.

A *API* foi desenvolvida para ser utilizada na linguagem *Javascript*, porém rapidamente o mercado desenvolveu *APIs* de nível mais alto, que usavam a API do google para executar funções. Por exemplo, é possível utilizar uma única função que desenha e cria uma fronteira de visualização para marcadores, fazendo o uso das funções do *Google Maps* de criar marcadores e delimitar fronteiras.

Frameworks de desenvolvimento também criaram *APIs* próprias, cujo o objetivo é mapear e simplificar o acesso a *API* do *Google Maps*. Estão entre as linguagens que possuem estes *frameworks*, o *Java*, o *Python*, o *PHP*, o *Objective-C*, o *.NET*, o *Ruby*, entre muitas outras.

O *Ruby* possui bibliotecas que contém funcionalidade prontas para uso. Algumas destas bibliotecas possuem funcionalidades que simplificam o acesso da *API* do *Google Maps*. Basicamente estas bibliotecas mapeiam as funções da ferramenta do *Google*, preparando internamente os objetos dos mapas. Deste modo, ao utilizar uma funcionalidade da biblioteca, não é necessário saber quais os objetos do *Google Maps* precisam ser criados, pois estes objetos serão gerados automaticamente na chamada da função.

O *Ruby On Rails* é um *framework* da linguagem *Ruby* que facilita o desenvolvimento de projetos *web*, fazendo a criação de estruturas básicas por meio da execução de linhas de comando, e possibilitando a importação de bibliotecas do *Ruby*.

O objetivo deste trabalho, é mostrar uma forma de adaptar uma biblioteca do *Ruby* que faz o mapeamento da *API* do *Google Maps* e apresentar um exemplo de uso desta biblioteca em um projeto do *Ruby On Rails*. Para aprimorar o conhecimento, antes será necessário apresentar conceitos sobre bibliotecas do *Ruby*, bem como o seu modelo de criação.

O texto apresentará um passo-a-passo da implementação de uma biblioteca simples e funcional do *Ruby* através de duas bibliotecas simples. A primeira que faz a tradução de palavras do português para o inglês, será utilizada para mostrar os conceitos da criação de bibliotecas no *Ruby*. A segunda que faz a adição da funcionalidade de desenhar caminhos entre dois locais utilizando uma biblioteca do *Ruby* que mapeia a *API* do *Google Maps*, será utilizada para apresentar conceitos da adaptação de bibliotecas no *Ruby*.

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 biblioteca 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 sub-programas 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 copias 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.

Em seguida, neste capítulo, na seção 2.1, vamos apresentar alguns conceitos das bibliotecas do *Ruby*, uma ferramenta que auxilia o acesso a estas bibliotecas, e um modelo de segurança adotado pela comunidade.

Para mais informações sobre bibliotecas, consulte o apêndice A, e 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, distribuídos em 2 tipos diferentes.

A primeira forma e mais utilizada, é a distrbuição de bibliotecas na forma de *gem*. O seu processo de instalação é feito por meio do programa *gem*. Por outro lado em menor número, existe a distribuição na formata de arquivos compactados em “.zip” ou “.tar.gz”. O seu processo de instalação geralmente é feito por meio de arquivos de “README”

ou "INSTALL", que possuem as instruções de instalação [2]. Em ambos os formatos, as bibliotecas baixadas são códigos fontes, devido ao fato que o *Ruby* é uma linguagem interpretada.

As bibliotecas do *Ruby* foram apelidadas com o nome de *gem* ou gemas, justamente pelo motivo que a maior parte das bibliotecas é desenvolvida na forma de *gem*.

2.1.1 Programa gem

O *gem* [1] é um sistema de pacotes do *Ruby*, desenvolvido para facilitar a criação, o compartilhamento, e a instalação de bibliotecas. Esta ferramenta possui características similares 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* por meio do servidor *RubyGems* [2].

Assim como o programa *apt-get*, o programa *gem* é executado em linhas de comando no formato "*gem <operação> <pacotes>*", onde "*<operação>*" pode ser "*install*" para instalar uma gema, "*search*" para fazer buscas, entre outros. Para mais informações, consulte o apêndice C.

2.1.2 Segurança

Antes de fazer a instalação de qualquer biblioteca em qualquer linguagem é necessário verificar se a fonte é confiável. Isso não é diferente na linguagem *Ruby*, pois a todo momento podemos abrir brechas e com isso correndo riscos de perder informações.

As bibliotecas do *Ruby* possuem como repositório padrão o *RubyGems*. Deste modo, fica claro que existe uma grande quantidade de bibliotecas neste repositório, e também que é quase impossível garantir que 100% destas bibliotecas sejam confiáveis.

A ferramenta *gem* que faz a instalação das gemas no ambiente de desenvolvimento, possuí uma política de segurança que tenta garantir que as bibliotecas instaladas são de uma fonte de confiança. Esta política funciona por meio da verificação de certificados digitais que consiste em um processo de 2 passos.

No primeiro passo, depois da criação da biblioteca, pessoas ou orgãos validam a confiabilidade da gema por meio de verificações e testes de segurança. Após esta verificação, caso a biblioteca seja confiável, ela recebe uma assinatura, feita por meio de uma chave privada de um certificado, para comprovar a sua segurança. Deste modo, todas as bibliotecas que possuem essa assinatura, são confiáveis, pois pela assinatura sabemos que determinada pessoa ou orgão comprovaram que a biblioteca é segura.

[1] gem: `https://rubygems.org/`

No segundo passo, o usuário final no momento da instalação da biblioteca, requisita por meio da política de segurança do *RubyGems*, a validação de confiabilidade da biblioteca a ser instalada. Existem vários níveis de segurança. O mais baixa, é a "*NoSecurity*" que não faz nenhuma verificação no momento da instalação. O mais alta, é a política "*HighSecurity*" que só permite a instalação de gemas que estão assinadas e sem nenhuma modificação. Neste segundo caso, o *RubyGems* ao receber essa requisição, verifica por meio do certificado e da assinatura da biblioteca, se ela recebeu a assinatura de alguma pessoa ou orgão de confiança, e se ela não foi modificada após a sua assinatura, pois modificação podem conter códigos maliciosos.

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 [3].

O *RubyGems* possui um esquema para reportar vulnerabilidades das bibliotecas do *Ruby*. Este esquema é constituído de dois tipos. Em um tipo se reporta erros na gema de outros usuários e no outro se reporta erros da própia gema. O esquema para cada tipo, será explicado a seguir.

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 para uma vulnerabilidade na própia *gema*, é necessário que se requisite um identificador *CVE* [2] via e-mail para `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 [3]. Além disso, também recomenda-se enviar um e-mail para `ruby-talk@ruby-lang.org` com o *subject: "[ANN][Security]"*, informando detalhes sobre a vulnerabilidade, as versões que possuem o erro, e as ações que devem ser tomadas para corrigir o problema.

[2] CVE: `https://cve.mitre.org/`
[3] OSVDB: `http://osvdb.org/`

3 Criação de bibliotecas do Ruby

Como visto no capítulo anterior 2, a maior parte das bibliotecas do *Ruby* são distribuídas na forma de *gems*, e também vimos na seção 2.1.1 que a ferramenta *gem* é um sistema de distribuição que facilita o compartilhamento e a instalação de *gemas*.

Este capítulo tem o objetivo de mostrar um passo-a-passo para criar uma gema. Apresentaremos primeiramente uma modelo de criação de gemas na seção 3.1. Depois na seção 3.2, vamos mostrar na prática como realizar os passos indicados na seção 3.1. E por fim na seção 3.3, apresentaremos um exemplo, criando uma gema que faz a tradução de algumas palavras do português para o inglês. Para obter informações sobre as ferramentas utilizadas e de como preparar o ambiente de desenvolvimento, consulte os apêndices D e E, respectivamente.

3.1 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.

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, somente apresentaremos a parte de implementação do modelo de criação.

Quando o projeto chegar no momento da implementação, devemos criar um diretório com o nome do projeto. Dentro deste diretório, vamos inserir todos os códigos relacionados a biblioteca, como por exemplo, a descrição, a versão, os códigos de funcionalidades, e os códigos de testes.

Após a criação do diretório do projeto, devemos criar um arquivo com a descrição da biblioteca. Esta descrição deve conter informações básicas, como por exemplo, o nome, a descrição, os autores, os contatos, e as dependências da biblioteca.

Com o arquivo da descrição criado, devemos determinar uma versão para a biblioteca, pois desta forma além de construirmos um histórico, podemos marcar e resgatar versões estáveis do projeto.

Com a descrição e a versão da biblioteca determinados, iniciamos o processo de criação de código. Esse processo possui dois tipos de desenvolvimento. O desenvolvimento de códigos de funcionalidades que consiste na implementação das funcionalidades da gema.

E o desenvolvimento de códigos de testes que serve determinar a corretude das funcionalidades.

Por fim, após a implementação dos dois tipos de códigos, devemos realizar a execução dos testes para confirmar que as funcionalidades foram implementadas de forma correta.

Para agilizar o processo de implementação, a comunidade do *Ruby*, desenvolveu ferramentas para facilitar a criação de aplicações e bibliotecas. Estas ferramentas são acionados por meio de linhas de comando para criar automaticamente esqueletos de aplicações ou bibliotecas. Estes esqueletos possuem arquivos com informações básicas que geralmente são utilizadas no desenvolvimento. Veremos mais detalhes sobre esse assunto na próxima seção.

3.2 Processo de Criação

Esta seção, tem o objetivo de mostrar detalhadamente a estrutura básica de uma gema, e também apresentar detalhadamente os passos do modelo de criação, vistos na seção 3.1.

Deste modo, antes de iniciarmos o processo de criação, na seção 3.2.1, vamos analisar detalhadamente todos os elementos da estrutura básica de uma gema, e depois com o este conhecimento adquirido, vamos mostrar na seção 3.2.2, o modelo de criação detalhado.

3.2.1 Estrutura

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

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 com extensão "*.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 arquivos de bibliotecas, usados pelo *Ruby*. Estas bibliotecas são carregadas 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* [1]. Um exemplo, é a execução de todos os arquivos de testes do diretório *test* ou *spec*.

- 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* [2], e as outras, em minoria, possuem a documentação *YARD* [3] [4].

3.2.2 Modelo de Criação Detalhado

Esta seção tem o objetivo de mostrar detalhadamente o processo de criação de uma gema. Apresentaremos na seção 3.2.2.1, uma forma de criação automatica da estrutura básica de uma gema. Depois na seção 3.2.2.2, apresentaremos como inserir os detalhes sobre a gema. Em seguida, na seção 3.2.2.3, apresentaremos que existe a possibilidade de escolher entre, implementar os códigos de funcionalidades e depois os códigos de testes, ou implementar os códigos de testes e depois implementar os códigos de funcionalidades. Depois na seção 3.2.2.4, apresentaremos como inserir os códigos de testes. Em seguida, apresentaremos na seção 3.2.2.5, como inserir os códigos de testes.

3.2.2.1 Criando a Estrutura

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

Podemos perceber que a forma manual não é muito aconselhável, pois perderiamos muito tempo criando os arquivos e também poderiamos cometer erros no momento da criação, como por exemplo, escrever errado o nome de algum arquivo ou diretório. Por estes motivos, utilizaremos a forma automatica que além de criar a estrutura, prepara um código base dentro dos arquivos.

A execução do comando “ ***bundle gem ‘ nome da gema’*** ” no terminal, faz a criação da estrutura básica de forma automatica. Neste caso, “*nome da gema*” pode ser o nome do projeto da biblioteca.

[1] rake: https://github.com/jimweirich/rake
[2] RDoc: http://rdoc.sourceforge.net/doc/
[3] YARD: http://yardoc.org/

Mais detalhes sobre a criação da estrutura serão abordados no exemplo apresentado na seção 3.3.1.

3.2.2.2 Inserindo Descrição

Após criar a estrutura básica 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* ".

Detalhes como, autores e e-mails servem para informar quem criou e como se comunicar com os donos da gema, nome e descrição servem para informar o objetivo da gema, e dependências servem para informar quais as bibliotecas são necessárias para o uso da gema.

Mais detalhes sobre o arquivo " *'nome da gema'.gemspec* ", serão abordados no exemplo na seção 3.3.2.

3.2.2.3 Desenvolver Funcionalidades ou Testes

Nesse momento, depois de detalharmos a gema no gemspec, podemos tomar 2 caminhos. Neste caso, o caminho a ser tomado é depende da metodologia de projeto adotada no inicio do desenvolvimento, ou seja, é nesse momento que podemos desenvolver os códigos das funcionalidades ou implementar os códigos de testes.

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

Seguindo a metodologia tradicional, primeiramente iremos fazer o código das funcionalidades da gema. Depois ao terminar de criar as funcionalidades, iremos elaborar os arquivos de testes. Mas nada o impede de desenvolver os códigos de testes que serão apresentado na seção 3.2.2.5, antes de desenvolver os códigos de funcionalidade, mostrados na seção 3.2.2.4.

3.2.2.4 Desenvolver Funcionalidades

Caso os códigos de funcionalidades sejam desenvolvidos 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, também com o nome da gema, com um arquivo de versão com o nome "*version*".

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* [4] e em *specification-reference-version* [5].

- *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.

No arquivo "*lib/'nome da gema'.rb*" temos a possibilidade de escrever todas as funcionalidades desejadas. No entanto é recomendado fazer a modularização por meio do comando "*require*" que é utilizado para fazer a chamada de código de outros arquivos.

Por exemplo, 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.

Mais detalhes sobre o arquivo "*lib/'nome da gema'.rb* " e os outros códigos de funcionalidades do diretório " *lib/'nome da gema'* ", serão abordados no exemplo na seção 3.3.3.

3.2.2.5 Desenvolver Testes

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*". Neste caso, "*definição do teste*" descreve o nome do teste a ser realizado, como por exemplo, para o arquivo "*test/test_raiz_quadrada.rb*", testa a função que calcula a raiz quadrada.

Após a criação dos arquivos de testes, é recomendado fazer a criação do arquivo "***Rakefile***" para automatizar os testes. Deste modo, quando for necessário testar a gema, não será necessário chamar teste por teste, pois todas as chamadas de teste estarão determinadas no arquivo "***Rakefile***".

[4] semantic-version: http://guides.rubygems.org/patterns/#semantic-versioning

[5] specification-reference-version: http://guides.rubygems.org/specification-reference/#version

Vale ressaltar que no momento do desenvolvimento dos códigos de testes, é aconselhável implementar testes para cada função da gema, verificado se para todas as entradas possíveis, se recebe o resultado esperado.

Mais detalhes sobre os arquivos de testes e o arquivo "***Rakefile***", serão abordados no exemplo na seção 3.3.4.

3.3 Exemplo de Criação

Esta seção tem o objetivo de mostrar um exemplo de criação de uma gema utilizando o modelo de criação. Para o exemplo, vamos fazer a criação da gema *gemtranslatetoenglish* [6], 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, vamos apresentar na seção 3.3.1, a criação da estrutura básica. Depois na seção 3.3.2, vamos mostrar como detalhar a gema no arquivo "*gemspec*". Em seguida, na seção 3.3.3, vamos mostrar a criação dos códigos de funcionalidades. Depois vamos apresentar na seção 3.3.4, a criação dos códigos de testes. E por fim na seção 3.3.5, apresentaremos um exemplo do uso da gema "*gemtranslatetoenglish*" em um projeto *Ruby On Rails*.

3.3.1 Criando a Estrutura Automaticamente

No caso da nossa gema de exemplo, foi feita a execução do comando **"*bundle gem gemtranslatetoenglish*"** para fazer a criação da estrutura básica de forma automatica. Ao se fazer a execução deste comando de criação, obtemos a seguinte estrutura de gema mostrada no código 3.1.

```
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
```

[6] gemtranslatetoenglish : https://github.com/toshikomura/gemtranslatetoenglish/tree/without_path

Código 3.1 – Execução que cria gema gemtranslatetoenglish

Podemos perceber que no código 3.1, foi feita a criação do diretório "*gemtranslatetoenglish*" com os arquivos "*Gemfile*", "*Rakefile*", "*LICENCE.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.

3.3.2 Editando Gemspec

No nosso exemplo, para detalharmos a gema, fizemos a edição do arquivo "gemtranslatetoenglish.gemspec", resultado no arquivo mostrado no código 3.2. Cada linha deste código 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
       ingles.}
13   spec.homepage      = ""
14   spec.license       = "MIT"
15
16   spec.files         = `git ls-files -z`.split("\x0")
17   spec.executables   = spec.files.grep(%r{^bin/}) { |f| File.basename(f) }
18   spec.test_files    = spec.files.grep(%r{^(test|spec|features)/})
19   spec.require_paths = ["lib"]
20
21   spec.add_development_dependency "bundler", "~> 1.5"
22   spec.add_development_dependency "rake"
23   spec.add_development_dependency "action_controller"
24 end
```

Código 3.2 – 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 a 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 dinâmicamente através do comando "*git ls-files -z*". Este comando do *git*, traz como resultado todos os arquivos que estão naquele repositório, colocando entre as *PATH*s, 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 *PATH*s 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.tes_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.

3.3.3 Código de Funcionalidade no Diretório Lib

Nesta seção vamos aprender a fazer os códigos de funcionalidades da gema. No nosso exemplo, podemos verificar no código 3.1, 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*".

No nosso exemplo para criar o módulo principal e modularizar a gema, desenvolvemos o arquivo "*lib/gemtranslatetoenglish.rb*" que é mostrado no código 3.3. Cada linha deste código é explicado logo a seguir.

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

Código 3.3 – gemtranslatetoenglish.rb

- "*require "gemtranslatetoenglish/version" *" na linha "1" é feita a requisição da versão.
- "*require "gemtranslatetoenglish/translatetoenglish.rb" *" na linha "2" é feita a requisição do arquivo "*translatetoenglish.rb*" contido no diretório "gemtranslatetoenglish".
- "*module Gemtranslatetoenglish ... end*" na linha "4" a "6" define o módula 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 auxilio 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.

Observando novamente o código 3.3, podemos perceber que na linha "4" foi feito o *require* do arquivo "*gemtranslatetoenglish/translatetoenglish.rb*'. Este é o arquivo que implementa a funcionalidade de tradução e o seu código pode ser visualizado no apêndice G no código G.1. Em seguida, por simplificação, apresentaremos somente o algoritmo de tradução no código 3.4. Neste algoritmo, a "*Arvore de modules*" na linha "1", define a árvore de módulos com "*Gemtranslatetoenglish*" na raiz, "*Helpers*" no segundo nível e "*Translatetoenglish*" no terceiro. E "*traducao*" na linha "2", define a função de tradução da gema.

```
1 Arvore de modules Gemtranslatetoenglish Helpers Translatetoenglish
2     traducao ( frase) {
3         Verifica se frase nao eh nula ou vazia
```

```
4         Caso seja, devolve uma frase vazia
5         Caso nao seja
6             Inicia uma frase_final vazia
7             Para cada palavra separada por espaco
8                 Procura palavra de traducao e concatena traducao na
    frase_final
9             Devolve frase_final
10     }
```

Código 3.4 – Algoritmo de translatetoenglish.rb

3.3.4 Código de Teste no Diretório Test ou Spec

No nosso exemplo para realizar os testes, criamos o arquivo "*test/test_check_translate.rb*" no código 3.5, 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::
    Translatetoenglish.translate("OLÁ")
10         assert_equal "WORLD ", Gemtranslatetoenglish::Helpers::
    Translatetoenglish.translate("MUNDO")
11     end
12     def test_text_translation
13         assert_equal "HELL WORLD ", Gemtranslatetoenglish::Helpers::
    Translatetoenglish.translate("OLÁ MUNDO")
14         assert_equal "WORLD HELLO ", Gemtranslatetoenglish::Helpers::
    Translatetoenglish.translate("MUNDO OLÁ")
15     end
16 end
```

Código 3.5 – 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.

Depois no nosso exemplo para automatizar os testes, desenvolvemos o arquivo "*Rakefile*" que pode ser visualizado no código 3.6, explicado em mais 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 3.6 – 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'* " indica 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.

Para informações de como realizar a execução dos testes, consulte o apêndice F.

3.3.5 Exemplo de Uso da Biblioteca Criada

Esta seção tem o objetivo de mostrar a utilização da gema de exemplo "*gemtranslatetoenglish*", 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 3.7, 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 3.7 – 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 3.8, 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 3.8 – 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 3.9.

```
1 gem 'gemtranslatetoenglish'
```

Código 3.9 – Adiciona gemtranslatetoenglish no Gemfile

Agora que a gema "*gemtranslatetoenglish*" já esta incluída no nosso projeto, podemos fazer o uso dela em uma *view* da mesma maneira como apresentada no código 3.10, 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 3.10 – 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 3.8.
- 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".

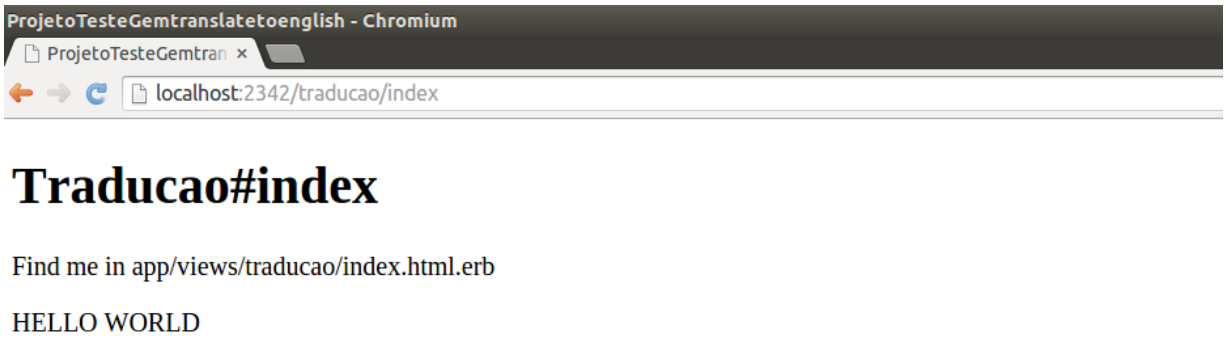

Figura 1 – Resultado de Translate na View

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 adição de novas funcionalidades com base em uma gema já existente.

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.

Este capítulo tem o objetivo de mostrar os passos que devem ser realizados para se adaptar uma biblioteca do *Ruby*. Na seção 4.1, apresentaremos o modelo de adaptação que deve ser seguido para adaptar um gema, e depois apresentaremos na seção 4.2, um exemplo da adaptação de uma gema que mapeia a *API* do *Google Maps*.

4.1 Modelo de Adaptação

Como citado anteriormente, o primeiro passo é encontrar uma gema que atenda boa parte das nossas necessidades e que utilizamo com frequência. Nesse passo, depois de determinar a gema a ser adaptada, devemos tomar cuidado ao adicionar qualquer funcionalidade nela, pois podemos estar comprometendo o contexto da gema, ou seja, comprometendo o objetivo final da biblioteca.

Para não comprometer o objetivo final da gema, é sempre válido verificar se a funcionalidade que estamos adicionando, esta no mesmo contexto da gema. Pois por exemplo em uma biblioteca matemática, podemos adicionar uma função para calcular a raiz quadrada. No entanto por outro lado, não faz nenhum sentido incluir uma funcionalidade de criar mapas do *Google* nesta biblioteca, pois esta funcionalidade de mapas não está no mesmo contexto da biblioteca.

Nesta seção apresentaremos na seção 4.1.1 o processo de engenharia reversa para transformar um sistema em diagramas de alto nível, depois na seção 4.1.2, apresentaremos uma forma para fazer a leitura dos diagramas de alto nível, e por fim apresentaremos na seção 4.1.3, o processo de adaptação.

4.1.1 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.

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.*

A ***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 caso alguma funcionalidade seja entendida de forma incorreta, existe a possibilidade de perda de recursos. E o modelo de abstração de alto nível, pode ser por exemplo um diagrama de classe ou um diagrama de herança.

4.1.2 Entendimento da Biblioteca

Agora que realizamos a *engenharia reversa* da *gema* para construir os diagramas de alto nível na seção 4.1.1. Devemos fazer a anaĺise destes diagramas para identificar os elementos do sistema, como por exemplo, classes e componentes.

Após a identificação dos elementos, devemos verificar para que cada um deste elemento é utilizado, ou seja, verificar o motivo pelo qual, determinado elemento existe no sistema.

Também devemos verificar quais são os principais elementos. Neste caso, os principais elementos são aqueles que são indispensáveis para o funcionamento da gema. Como por exemplo, o objeto mapa é um elemento indispensável para mostrar um mapa na tela.

Depois de fazer a identificação dos elementos, passamos para a parte de análise das funcionalidades. Nesse passo, verificamos quando os elementos são criados e em quais funcionalidades eles são utilizados.

Desta forma com a identificações dos elementos e das funcionalidades, temos um entendimento completo da biblioteca.

4.1.3 Adaptações

Esta seção tem o objetivo de apresentar como podemos fazer adaptações nas gemas, após realizar o processo de *engenharia reversa* e o processo de entendimento da biblioteca.

Inicialmente devemos verificar se é necessário fazer a inclusão de novos elementos, pois com já sabemos quais os elementos existentes no sistema, e o motivo pelo qual eles existem, podemos analisar se é ou não necessário incluir novos elementos.

Depois da análise da inclusão de novos elementos, devemos verificar quais os elementos que vão receber as novas funcionalidades, sempre verificando se não estamos modificando o objetivo final do elemento.

Por fim, após fazer a análise dos elementos e das funcionalidade, devemos considerar os impactos das modificações, verificando se nenhuma funcionalidade, já existente, foi afetada.

4.2 Exemplo de Adaptação

O objetivo desta seção é apresentar um exemplo da adaptação da gema *Google-Maps-for-Rails* [1] criada por *Benjamin Roth* [2] e *David Ruyer* [3]. Essa gema 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 flexível que permite a aceitação de outros provedores de mapas, como por exemplo o *Bing Maps* [5].

Na seção 4.2.1, apresentaremos alguns conceitos da *API* do *Google*. Em seguida na seção 4.2.2, apresentaremos alguns conceitos da linguagem *CoffeeScript*, linguagem utilizada na gema para mapear a *API* do *Google Maps*. Depois na seção 4.2.3, apresentaremos a abordagem utilizada para adaptar a gema. Em seguida apresentaremos na seção 4.2.4, a realização do processo de *engenharia reversa* na gema de exemplo. Depois na seção 4.2.5, apresentaremos um análise dos elementos da gema "*Google-Maps-for-Rails*". Em seguida na seção 4.2.6, apresentaremos as modificações realizadas na gema de exemplo. E por fim apresentaremos na seção 4.2.8, um exemplo do uso de "*Google-Maps-for-Rails*" em um projeto do *Ruby On Rails*.

4.2.1 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* [6] e a *Google Maps API V3* [4], onde ambos se complementam ensinando os passos básicos para criar e manipular mapas do *Google* utilizando *Javascript*.

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 [6].

[1] Google-Maps-for-Rails : https://github.com/apneadiving/Google-Maps-for-Rails
[2] Bejamin Roth: https://github.com/apneadiving
[3] David Ruyer: https://github.com/MrRuru
[4] Google Maps API V3: https://developers.google.com/maps/

Antes de criação do *Google Maps*, existia um grande problema de rederização que ocorria por causa que um mapa era um elemento único. Devido a este fato, podemos perceber que o elemento do mapa possuía uma grande quantidade de informações. Deste modo, fica claro que a transferência de um mapa entre um servidor e um cliente, era uma tarefa cara, pois era necessário fazer a transferência de uma grande quantidade de dados. E além de existir uma alto custo na transferência, também existia um alto custo no cliente no momento da transformação das informações em forma de mapa.

O *Google Maps* fez a divisão dos mapa em vários pedaços, uma vez que o problema da rederização ocorria por causa da representação do mapa em um elemento único [1]. Nesta solução, os pedaços dos mapas poderiam ser requisitados um a um para montar uma parte ou um mapa completo. Desta forma, foi possível solucionar o problema da grande quantidade de dados transferidos, pois somente se transfere as partes requisitadas do mapa, e também se solucionou o problema da transformação, pois a quantidade de dados para se transformar, foi reduzida razoavelmente.

A ferramenta *Google Maps* manipula mapas por meio de *HTML*, *CSS* e *Javascript*. O posicionamento dos mapas é feito por meio de *HTML* e *CSS*. As requisições e o posicionamento dos pedaços dos mapas são feito por meio de *Javascript*.

O processo de manipulação de mapas possui 3 etapas. Na primeira, o usuário por meio do *browser*, requisita algum local do mapa informando a coordenada e o zoon desejado. Na segunda, o servidor recebe a requisição, procura o pedaço, e retorna a imagem do mapa que representa a posição requisitada. Na terceira, o *browser* recebe e rederiza a imagem no mapa [6].

4.2.2 CoffeeScript

Para fazer a adaptação da gema, também foi necessário fazer um estudo sobre a linguagem *CoffeeScript*, pois ela é utilizada na gema para mapear a *API* do *Google Maps*.

O *CoffeeScript* é uma linguagem que simplifica o modo de desenvolver códigos *Javascript*, expondo os melhores pontos de forma simples e clara. A regra de ouro do *CoffeeScript* é que "*CoffeeScript é simplesmente Javascript*", pois o seu código é compilado linha a linha, sendo transformado em *Javascript* [7].

A linguagem permite o uso de qualquer biblioteca escrita em *Javascript*. Seu código compilado, é legivel e identado, podendo executar em um tempo superior ao de um código *Javascript* escrito manualmente [7].

Vimos na seção 4.2.1 que a *API* do *Google Maps* funciona sobre *Javascript*. Como o *CoffeeScript* é *Javascript*, e já que o seu desenvolvimento é mais simples. Ele foi utilizado para mapear a *API* do *Google Maps* na gema original e na adaptada.

4.2.3 Abordagem de Adaptação

Para realizar as adaptações na gema *Google-Maps-For-Rails*, vamos seguir o modelo de adaptação, visto na seção 4.1.

Nas seções seguintes, vamos realizar o processo de engenharia reversa, o entendimento, e as adaptações na gema. Alguns conceitos do processo de engenharia reversa e o entendimento da gema, já foram vistos nas seções 4.1.1 e 4.1.2, respectivamente. Para realizar as adaptações, poderiamos utilizar uma entre duas abordagens.

A primeira consiste em criar uma nova biblioteca que inclui a biblioteca original e acrescenta novas funcionalidades. Neste caso deveriamos criar uma nova biblioteca, importar a biblioteca original, e adicionar novas funcionalidades. A vantagem desta abordagem é que a nova biblioteca sempre receberá as atualizações feitas na biblioteca original.

A segunda consiste em criar uma nova biblioteca, copiar todo código da biblioteca original e acrescentar novas funcionalidades. Neste caso deveriamos criar um *branch* da biblioteca original e adicionar novas funcionalidades. A vantagem desta abordagem é que a nova biblioteca é independente da biblioteca original.

Acabamos optando pela segunda abordagem, pois mudanças na biblioteca original, poderiam causar transtornos nas modificações, prejudicando assim o desenvolvimento deste trabalho.

4.2.4 Engenharia Reversa da Biblioteca de Exemplo

Aplicando a *engenharia reversa* na gema *Google-Maps-For-Rails*, conseguimos obter como resultado O diagrama de atributos na imagem 2 que possui as classes e os seus respectivos atributos. 3 diagramas de métodos na imagem 3 que possui as classes e os seus respectivos métodos. O diagrama de herança na imagem 4 que possui apresenta a heraça entre as classes.

Nenhum dos 3 diagramas existe na definição de diagramas da *UML*, e isso se deve ao fato de não existir espaço suficiente na imagem para representar o sistema da gema por completo em um diagrama de classes. Por esse motivo, optamos por definir novos diagramas que possuem características similares aos padrões da *UML*, mas com algumas representações adicionais, podendo assim representar o diagrama de classe por completo.

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

- 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*.

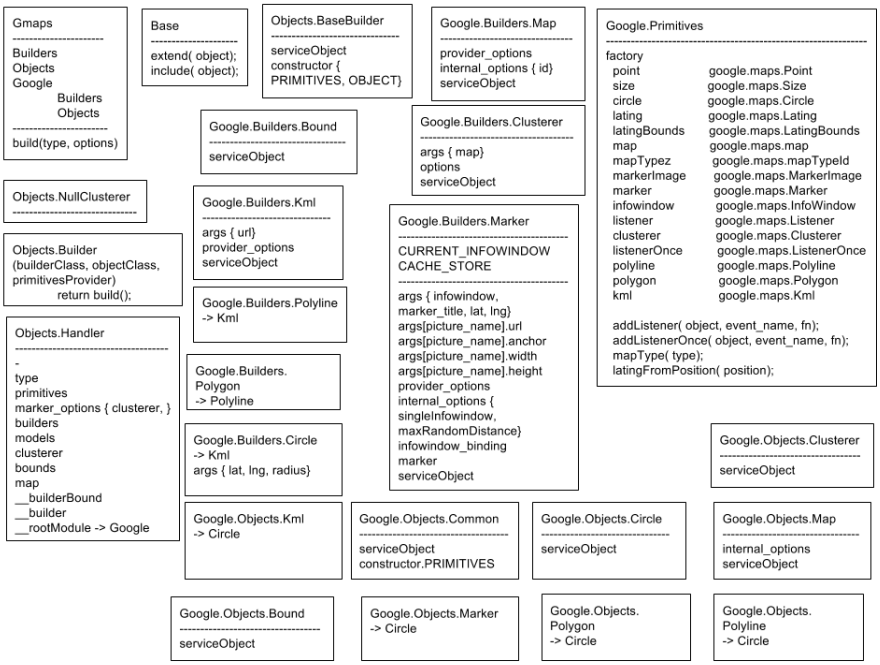

Figura 2 – Diagrama de Atributos Google-Maps-For-Rails

- 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 atributos da classe que vem depois do símbolo "->". Por exemplo a classe "*Google.Builder.Polyline -> Kml*" que representa a classe "*Google.Builder.Polyline*", ela possuí os mesmo atributos da classe "*Kml*", ou seja, ela possuí os atributos "*args { url }*", "*provider_option*" e "*serviceObject*". E também existe o caso onde esta classe possuí 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 possuír os atributos da classe *Kml*, ela possuí o atributo "*args { lat, lng, radius }*".

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

- Todos os nomes seguidos de "(...)" abaixo do traço "——" representam os métodos da classe.

- 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 possuí os mesmo métodos da classe "*Kml*", ou seja, ela possuí os métodos "*constructor()*", "*create_...()*" e "*..._option()*". E também existe

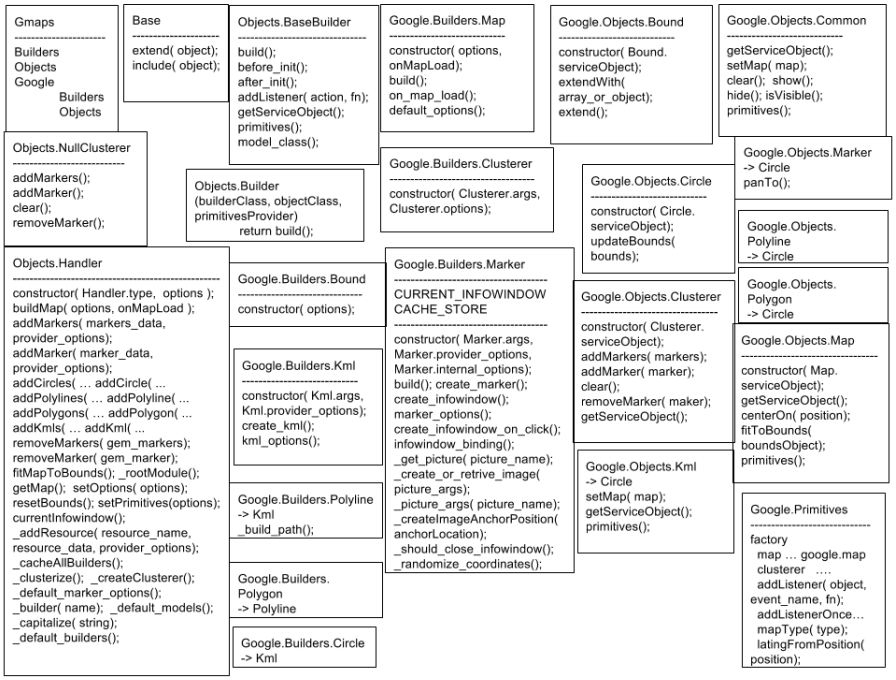

Figura 3 – Diagrama de Métodos Google-Maps-For-Rails

o caso onde esta classe possuí 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 possuír os métodos da classe *Kml*, ela possuí o método "*_build_path()*".

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

- As linhas pretas continuas, 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 pretas em traços com pontas de seta, representam a herança entre duas *classes*. A classe que está com a seta, é a classe que herda as características da classe na outra ponta da linha.
- As linhas pretas em pontos com pontas de quadrado, representam a inclusão de uma classe na outra. 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

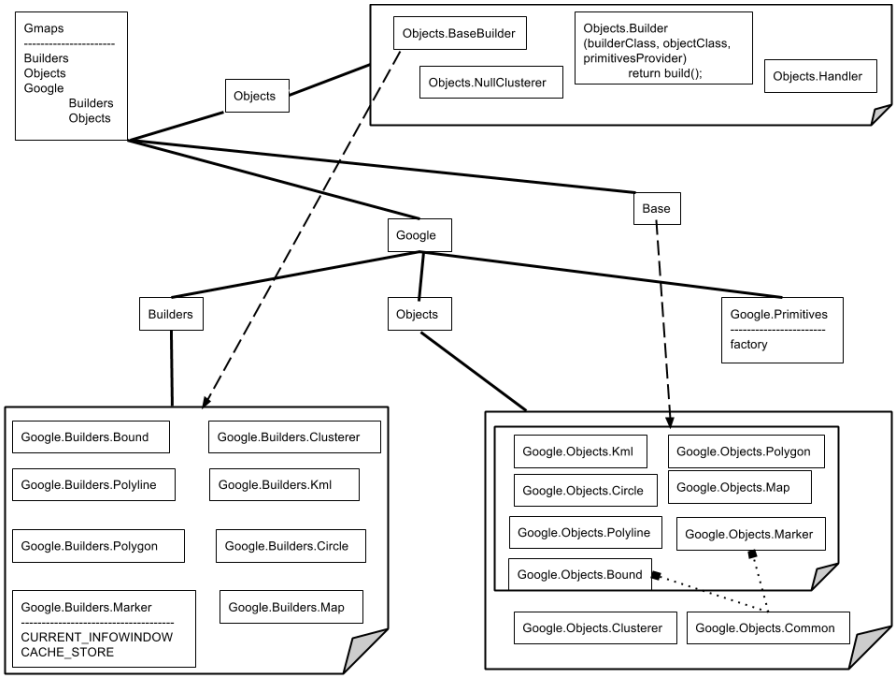

Figura 4 – Diagrama de Herança Google-Maps-For-Rails

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*".

4.2.5 Entendimento da Biblioteca Adaptada

Para a gema de exemplo, seguindo o modelo de entendimento visto no capítulo anterior na seção 4.1.2, encontramos as seguinte características que serão listadas e explicadas 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*.

4.2.6 Adaptações da Biblioteca de Exemplo

Tendo a abstração de alto nível para a gema *Google-Maps-For-Rails* e o entendimento dela, podemos partir para a adaptação, ou seja, agora que jé criamos os diagramas e fizemos uma análise sobre eles, podemos tentar acrescentar novas funcionalidades, analisando os locais das possíveis modificações e os impactos que essas mudanças podem causar.

A gema já possuí sobreposições como *markers* e *circles*, mas até o momento não possuí 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* [5] (*Serviço de Direção*) do *Google*. Deste modo, 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*".

[5] Direction Service: https://developers.google.com/maps/documentation/javascript/directions

Com conhecimento das características da gema de exemplo e dos objetos para criar direções, tivemos a possibilidade de elaborar as modificações necessárias para que a gema comporte a funcionalidade de gerar direções.

Em linhas gerais, as adaptações consistiram em adicionar a definição das classe necessárias na classe "*Primitives*", assim conseguimos mapear os objetos dos mapas do *Google* dentro da gema. Depois incluímos os elementos de "*builders*" e "*objects*" que representem as classes adicionadas, assim conseguimos criar e manipular os novos objetos do *Google Maps*. E por fim, desenvolvemos as funcionalidades para manipular os novos elementos na classe "*Handler*", assim conseguimos por meio da criação e da manipulação dos objetos do *Google Maps*, requisitar e incluir direções nos mapas.

Deste modo, como primeiro passo, 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      ...
5      directionSer: google.maps.DirectionsService
6      directionRen: google.maps.DirectionsRenderer
7      directionTM:  google.maps.TravelMode
8      directionSta: google.maps.DirectionsStatus
9      ...
10
11   }
```

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 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.

Agora que acrescentamos estas quatro classes na gema adaptada, devemos adicionar no "*Handler*", novas funções para manipular essas classes. E neste caso, inserimos as funçãoes "*addDirection()*" e "*calculate_route()*" que podem ser vistas no algoritmo do código 4.2 que será explicado logo a seguir. A implementação deste algoritmo está no apêndice G no código *Ruby* G.2.

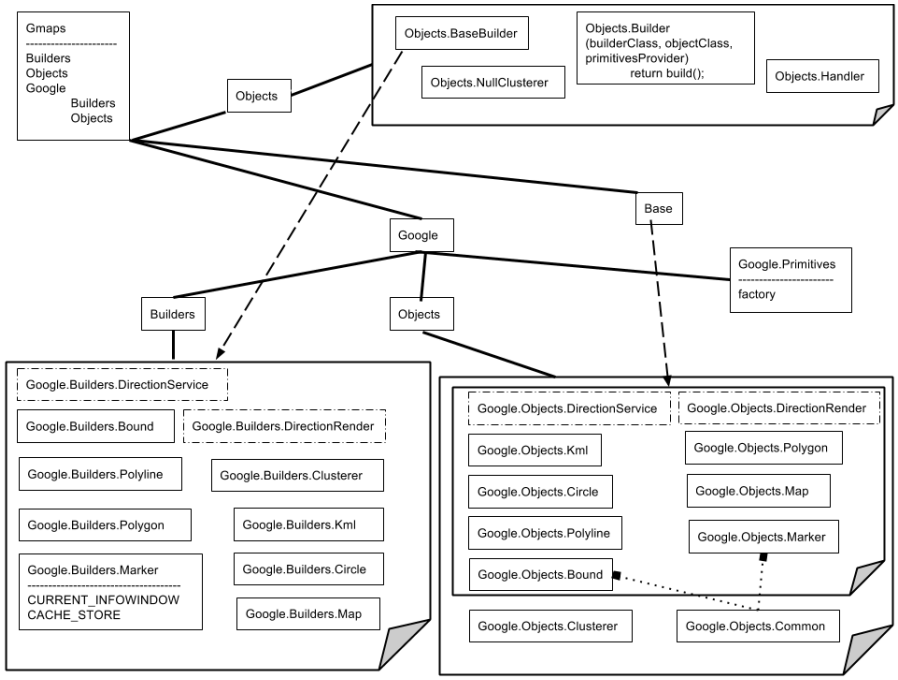

Figura 5 – Novo Diagrama de Herança Google-Maps-For-Rails

```
1  Handler {
2  ...
3      addDirection (dados_da_direcao, opcoes) {
4          se dados_da_direcao possui origem e destino
5              cria DirectionService e DirectionRender
6              chama calcula_direcao para requisitar ao google um caminho
7              Pelo DirectionRender desenhe caminho retornado no mapa
8          senao
9              informa que eh necessario passar um local de origem e destino
10     }
11
12     calculate_route (dados_da_direcao) {
13         Por meio do DirectionService requisita um caminho ao google passando os
       dados_da_direcao
14             caso receba uma resposta positiva, devolve o caminho no DirectionRender
15             caso nao receba uma resposta positivo, devolve uma mensagem de erro
16     }
17 ...
18 }
```

Código 4.2 – Algoritmo de Funções adicionais do Handler

- Na linha "1" temos a definição da classe "*Handler*".
- Os "..." representam os códigos já existentes na classe e que não são importantes para o exemplo.

- Na linha "3" é adicionado a função "*addDirection*" que recebe como parâmetro "*dados_da_direcao*", que possui o informações do local de origem e local de destino que são obrigatórios para criar a direção, e as "*opcoes*" 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 "*dados_da_direcao*", que tem por principal objetivo fazer a requisição para o *Google* de uma possível direção entre o local de origem ao local de destino.

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.

Existem outras possíveis soluções, como por exemplo, a criação de somente uma função para criar e requisitar direções ao *Google*, e também poderiamos pensar em mesclar as direções com os *markers*, devido ao fato que as direçõs utilizam *markers* para marcar o ponto de inicio e o ponto de fim. No entanto, nenhuma das duas modificações é benéfica para a biblioteca, pois a primeira não faz a modularização, e a segunda aumenta a complexidade da biblioteca.

4.2.7 API da Biblioteca Adaptada

Para facilitar o uso da nova funcionalidade de direções, apresentaremos rapidamente nesta seção, a *API* da gema adaptada.

Basicamente a função adicionada é a *addDirection()* que recebe como parâmetro um local de origem e um local de destino. E depois de fazer a requisição ao *Google*, retorna no mapa, o caminho entre a origem e o destino.

```
1  $(document).ready(->
2    handler = Gmaps.build("Google")
3    handler.buildMap
4      provider: {}
5      internal:
6        id: "map"
7    , ->
8      handler.addDirection( { origin: "Insert Origin Location",
       destination: "Insert Destination Location"})
9      return
10   return
11 )
```

Código 4.3 – Exemplo CoffeeScript API Google-Maps-For-Rails Adaptado

Para utilizar a funcionalidade de direção da gema adaptada, basta fazer a instalação da gema. Depois incluir a gema no projeto em que se deseja utilizar a funcionalidade de direções. Em seguida, dentro do projeto, implementar o código de criar mapas que pode ser o mesmo usado na gema original. E no fim para criar a direção, chamar a função *addDirection()* com um local de origem e um local de destino.

O código 4.3, mostra um código base para criar um mapa com uma direção.

4.2.8 Exemplo de Uso da Biblioteca Adaptada

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*" [6] 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.

```
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
     /13/13028-64.png"
13         width: 36
14         height: 36
15       infowindow: "hello!"
16     ])
17     handler.bounds.extendWith markers
18     handler.fitMapToBounds()
19     handler.addDirection( { origin: "Sao Paulo", destination: "Curitiba"
     })
20     return
21   return
22 )
```

Código 4.4 – Exemplo CoffeeScript que Cria Mapa com Direção

[6] DiseasesMap : https://github.com/toshikomura/DiseasesMap

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 4.4, explicado logo abaixo.

- 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 parir 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 criado 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"* ".

```
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 4.5 – Exemplo Locations view que Cria Mapa com Direção

E para mostrar o mapa na view de "*locations*" adicionamos o código mostrado em 4.5 que é parte do código da *view*, explicado logo a seguir.

- Na linha "1" com a *tag* <h1>...</h1> é definido como título principal da *view* o texto "*Listining 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 referênciado pelo atributo *id* que é o mesmo *id* utilizado no código 4.4 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 4.4.

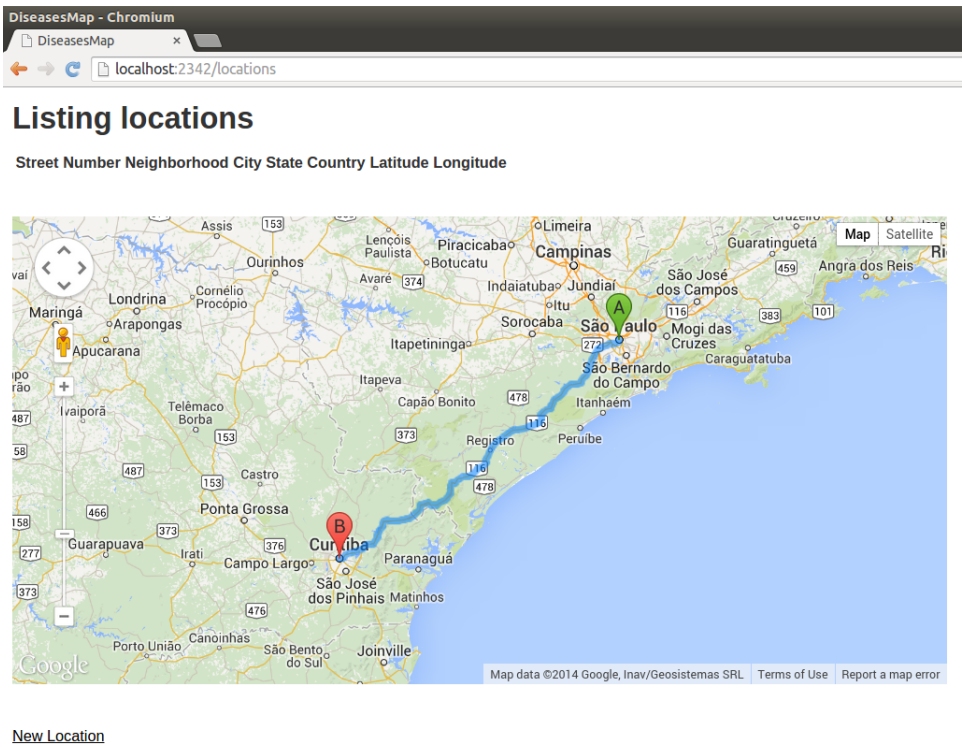

Figura 6 – Caminho entre São Paulo e Curitiba

5 Conclusão

Neste trabalho apresentamos no capítulo 2, alguns conceitos básicos sobre as bibliotecas do *Ruby*, descrevendo a importância do seu uso e a possibilidade do uso de certificados digitais para ter mais segurança.

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 as suas própias bibliotecas.

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 PETROUTSOS, E. *Google Maps: Powwer Tools for Maximizing the API*. 1. ed. New York. NY, 2 Penn Plaza, 10th Floor: McGraw-Hill Osborne Media, 2014. Citado 2 vezes nas páginas 6 e 28.

2 COMMUNITY, R. *Ruby Libraries*. 2014. Ruby Site - Libraries. Disponível em: <http://www.ruby-lang.org/en/libraries/>. Acesso em: 17 out. 2014. Citado na página 9.

3 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.

4 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 13.

5 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 27.

6 SVENNERBERG, G. *Beginning Google Maps API 3*. 2. ed. New York, NY 10013: Apress, 2010. Citado 2 vezes nas páginas 27 e 28.

7 ASHKENAS, J. *CoffeeScript*. 2014. Disponível em: <http://www.coffeescript.org/>. Acesso em: 08 dez. 2014. Citado na página 28.

8 WEXELBLAT, R. *History of Programming Languages*. 1. ed. New York, NY: Academic Press: ACM Monograph Series, 1981. Citado na página 44.

9 COMMUNITY, R. *Ruby*. 2014. Ruby Community. Disponível em: <http://-/www.ruby-lang.org/>. Acesso em: 10 out. 2014. Citado 2 vezes nas páginas 47 e 48.

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 47.

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 47.

12 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 49.

13 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 50.

14 COMMUNITY, R. *RubyGems About*. 2014. RubyGems Site - About. Disponível em: <http://rubygems.org/pages/about>. Acesso em: 21 out. 2014. Citado na página 51.

15 VMWARE, I. *VMware® Player*. 2014. VMware® Player. Disponível em: <http://www.vmware.com/products/player>. Acesso em: 23 out. 2014. Citado na página 52.

16 SEGUIN, M. P. W. E. *RVM*. 2014. RVM. Disponível em: <http://rvm.io/>. Acesso em: 22 out. 2014. Citado na página 52.

17 SEGUIN, M. P. W. E. *RVM*. 2014. RVM. Disponível em: <http://rvm.io/rvm/about>. Acesso em: 22 out. 2014. Citado na página 52.

18 HANSSON, D. H. *Ruby On Rails*. 2014. Ruby On Rails. Disponível em: <http://rubyonrails.org/>. Acesso em: 22 out. 2014. Citado 2 vezes nas páginas 52 e 53.

19 TORVALDS, L. *Git*. 2014. Git. Disponível em: <http://git-scm.com/>. Acesso em: 23 out. 2014. Citado na página 53.

20 COMMUNITY, R. *IRB*. 2014. Ruby Community. Disponível em: <http://www.ruby-doc.org/stdlib-2.0/libdoc/irb/rdoc/IRB.html>. Acesso em: 28 out. 2014. Citado na página 58.

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 programas, 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 A.2.1, pelo momento que elas são ligadas, explicado na sub-seção A.2.2, e pela forma como elas são compartilhadas, explicado na sub-seção A.2.3.

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 indice 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 mémória 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 capitulo 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, depois vamos ver a importância de uma *API* na seção B.3, e por fim na seção B.4, vamos falar um pouco sobre a segurança do *Ruby*.

B.1 Conceitos

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* [1] que tentou cuidadosamente criar uma linguagem balanceada, 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 relembra 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 construimos um *método* "tipo" para o *objeto* inteiro "5". Suponde 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

[1] Yukihiro "Matz" Matsumoto: http://www.rubyist.net/~matz/

O *Ruby* também é flexível, pois possibilita remover ou redefinir seu própio *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*". Subsequêncialmente em dois dias foram lançadas mais três versões juntamente com a "*Japanese-Language ruby-list main-list*" (*RubyTalk*).

Ruby-Talk [2] a primeira *lista de discussão da Ruby* [3] chegou a possuir em média 200 mensagens por dia em 2006. E com o passar dos útlimos 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* [4], basta saber para que serve cada uma de suas funções disponibilizadas, seus parâmetro de entrada e saída com seus respectivos tipos.

B.4 Segurança

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`, que é uma lista privada com membros que administram o *Ruby*, como por exemplo *Ruby commiters*. 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 [12].

[2] Ruby-Talk: `https://www.ruby-forum.com/`
[3] Listas de discussões do Ruby: `https://www.ruby-lang.org/en/community/mailing-lists/`
[4] Core do Ruby: `http://www.ruby-doc.org/core-2.1.3/`

Um exemplo é a vulnerabilidade publicada em 10/04/2014, referênciado 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 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 [13].

APÊNDICE C – Projeto RubyGems

O projeto *RubyGems* em conjunto com a ferramenta *gem*, foi criado em Abril de 2009 por *Nick Quaranto* [1], numa tentativa de proporcionar uma melhor *API* para acessar as gemas [14].

Com o passar do tempo, o projeto cresceu, chegando a possuir mais de 115 *Rubyistis* contribuintes e milhões de gemas baixadas do *RubyGems* através da ferramenta [14].

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* [14].

A instalação do *gem* pode ser feita pelo terminar, executando "*sudo apt-get install gem*" com privilégios de administrador ou acessando o site do projeto *https://rubygems.org/pages/dou* e baixando a última versão do *gem*. 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.

[1] Nick Quaranto: https://twitter.com/qrush

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 portável [15].

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 [1] 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 [16].

Wayne E. Seguin [2] 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 [17].

D.3 Ruby On Rails

Ruby On Rails [3] 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 [18].

[1] RVM: http://rvm.io/
[2] Wayne E. Seguin: https://github.com/wayneeseguin
[3] Ruby On Rails: http://rubyonrails.org/

O *Rails* foi criado em 2003 por *David Heinemeier Hansson* [4] e desde então é extendido pela equipe do *Rails Core Team* [5] e mais de 3.400 usuários [18].

D.4 Git

Git [6] 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 [19].

Por irônia do destino o *git* foi criado em 2005 por *Linus Torvalds* [7] 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 gerênciamento 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 *branchs*), completamente distribuído e capacidade de controlar grandes projeto com grandes quantidades de dados.

[4] David Heinemeier Hansson: `http://david.heinemeierhansson.com/`
[5] Rails Core Team: `http://rubyonrails.org/core/`
[6] Git: `http://git-scm.com/`
[7] 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 inicio 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 isntalação 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.github.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.github.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  ource "/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 "*ource "/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.

Tammbém 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 essa instalações, ainda devemos instalar as gemas essências, "*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 Essênciais

- 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 – Execução de Testes

Este capítulo tem o objetivo de apresentar como se pode realizar a execução de testes da gema de exemplo "*gemtranslatetoenglish*". Na seção F.1, apresentaremos como é feita a execução de testes com o *rake*, e apresentaremos na seção F.2, como realizar testes com a ferramenta *irb*.

F.1 Testes com rake

Esta seção tem o objetivo de mostrar como se pode realizar os teste, após criar os códigos de funcionalidade, visto na sub-seção 3.3.3, e os códigos de teste, visto na sub-seção 3.3.4.

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 exemplo, foi fazer a execução dos comandos mostrados no código F.1 para fazer a criação e a instalação da gema "*gemtranslatetoenglish*".

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

Código F.1 – 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 F.2, 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 F.2 – 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 3.5 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.

F.2 Testes com irb

Esta seção tem o objetivo de mostrar como se realiza teste utilizando a ferramenta *irb* que permite verificar as funcionalidades de uma gema de forma manual, sem a necessidade de inclui-lá em um projeto.

O *IRB* [1] (*Interactive Ruby Shell*) é uma ferramenta do *Ruby* que serve para executar expressões interativamente, fazendo a leitura da entrada padrão [20].

Um exemplo de uso do *IRB* pode ser visto no código F.3 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.3 – 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".

[1] IRB: http://www.ruby-doc.org/stdlib-2.0/libdoc/irb/rdoc/IRB.html

- 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.4 explicado com mais detalhes nos itens abaixo.

```
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.4 – 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 – Códigos Ruby

Este capítulo tem o objetivo de mostrar detalhadamente os códigos *Ruby* que foram mostrados na forma de algoritmo no trabalho.

```ruby
module Gemtranslatetoenglish
    module Helpers
        module Translatetoenglish

            def translate( phrase)
                # Check if phrase has something
                if phrase == nil
                    return ""
                end
                if phrase.empty?
                    return ""
                end
                # To each word in the phrase
                i = 0
                phrase = phrase.split
                result = ""
                for word in phrase
                    word = word.upcase
                    case word
                        when "OLÁ"
                            result = result + "HELLO"
                        when "MUNDO"
                            result = result + "WORLD"
                        else result
                            "- (#{word}) NAO PODE SER TRADUZIDO -"
                    end
                    # If is not in the end of phrase
                    if i < phrase.length
                        result = result + " "
                        i = i + 1
                    end
                end
                return result
            end
...
```

Código G.1 – translatetoenglish.rb

O arquivo mostrado no código G.1, explicado logo a seguir. faz referência ao algoritmo mostrado no código 3.4.

- "*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 G.1 foi necessário por causa do código inserido na linha "8" "*ActionController::Base.helper Gemtranslatetoenglish::Helpers::Translatetoenglish*" no código 3.3 que serve para evitar a necessidade de escrever a *PATH* completa na *view* para chamar uma função da gema na *view*.

O arquivo mostrado no código G.2, explicado logo a seguir, faz referência ao algoritmo mostrado no código 4.2.

- 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 objetivo 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 atribuindo o seu respectivo objeto do *Google Maps* com a chamada da função "*@_builder(...)*". Depois é feito a chamada da função "*calulate_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 so 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(
   direction_data)
5        @direction_render = @_builder('DirectionRender').build(
   provider_options)
6        @calculate_route( direction_data)
7        @direction_render.getServiceObject().setMap(@getMap())
8        @direction_render.getServiceObject()
9     else
10       alert "Need direction origin and destination\n and you inform\n
   origin: " + direction.origin + "destination: " + direction.
   destination
11
12 # calculate routes of direction
13 calculate_route: ( direction_data)->
14    statusOk = @direction_service.primitives().directionStas('OK')
15    direction_render_serviceObject = @direction_render.getServiceObject
   ()
16    @direction_service.getServiceObject().route direction_data, (
   response, status) ->
17      if status is statusOk
18        direction_render_serviceObject.setDirections response
19      else
20        alert "Could not find direction"
```

Código G.2 – Funções adicionais do Handler