Universidade Federal Fluminense
Instituto de Computação
Departamento de Ciência da Computação

Carlos Heitor Diniz Moreira
Leonardo Sousa de Freitas

Aprendendo o *framework Ruby on Rails* através do desenvolvimento de um *software* gerenciador das Olimpíadas da UFF

Niterói-RJ
2011

CARLOS HEITOR DINIZ MOREIRA
LEONARDO SOUSA DE FREITAS

APRENDENDO O FRAMEWORK RUBY ON RAILS ATRAVES DO DESENVOLVIMENTO DE UM SOFTWARE GERENCIADOR DAS OLIMPÍADAS DA UFF

Monografia apresentada ao Curso de Graduação em Ciência da Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Orientador: Prof. Marco Antonio Monteiro Silva Ramos, D.Sc.

Niterói-RJ
2011

CARLOS HEITOR DINIZ MOREIRA
LEONARDO SOUSA DE FREITAS

APRENDENDO O FRAMEWORK RUBY ON RAILS ATRAVES DO DESENVOLVIMENTO DE UM SOFTWARE GERENCIADOR DAS OLIMPÍADAS DA UFF

Monografia apresentada ao Curso de Graduação em Ciência da Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Aprovada em dezembro de 2011.

BANCA EXAMINADORA

Prof. Dr. MARCO ANTONIO MONTEIRO SILVA RAMOS - Orientador
UFF

Prof. Dr. MAURÍCIO KISCHNHEVSKY
UFF

Prof. Me. DANTE CORBUCCI FILHO
UFF

Niterói-RJ
2011

Às nossas famílias, pelo amor incondicional.

"Noventa por cento do sucesso se baseia simplesmente em insistir."

— Woody Allen

"A educação é a arma mais poderosa que voce pode usar para mudar o mundo."

— Nelson Mandela

Agradecimentos

A Carlos Augusto e Mara, por serem meus pais e por todo amor com o qual talharam a minha vida.

À Bianca, por ser minha irmã e atenuar as pedras da minha caminhada com seus conselhos.

Ao meu amigo Leonardo, pela ilustre companhia, regada de sorrisos e poesia.

Aos meus amigos, pela amizade e cumplicidade nos melhores anos da minha vida.

Carlos Heitor Diniz Moreira

A Deus, por, após tantas provações, ter me dado forças e me guiado à graça desse momento tão esperado da formatura.

Aos meus pais, Antônio e Isabel, e à minha irmã Fernanda, pelo amor, torcida e confiança.

À minha esposa Fabrícia, pelo amparo incondicional e pelos incansáveis incentivos perante às adversidades.

Ao Professor Maurício Kischnhevsky, pela amizade e por todo o aprendizado proporcionado nestes anos prazerosos de convívio acadêmico.

Ao amigo Carlos Heitor, pela parceria nesta monografia e pelo prazer inenarrável da amizade.

Aos amigos Douglas Mareli, Flávio Rocha, Leandro Oliveira, Magno Mathias e Luiz Guilherme, por terem feito de suas casas a minha segunda.

Aos amigos Bernardo Breder, Giancarlo Taveira, Heraldo Borges, Daniel Martins, Raphael Bottino, Gabriel Ayres, Felipe Ribeiro e Vinícius Figueiredo, por todo o apoio, conselhos e compreensão.

À Universidade Federal Fluminense, minha augusta alma-mater, pelos anos de convívio e aprendizado com discentes e doutores do mais alto nível.

Leonardo Sousa de Freitas

Lista de Figuras

Lista de Tabelas

Lista de abreviaturas e siglas

AJAX : *Asynchronous Javascript and XML* (Javascript e XML Assíncronos);

CRUD : *Create, Read, Update and Delete* (Criar, Ler, Atualizar e Deletar);

DRY : *Don't Repeat Yourself* (Não Se Repita);

UFF : Universidade Federal Fluminense;

URL : *Uniform Resource Locator* (Localizador Padrão de Recursos);

Sumário

Resumo

O conceito de desenvolvimento ágil vem se tornando uma preocupação crescente na área de Ciencia da Computação. Atualmente, o *framework Ruby on Rails* vem se apresentando como uma opção muito produtiva de desenvolvimento *web*, que alia robustez de recursos e agilidade no processo de desenvolvimento de *software*. Um problema prático constatado na UFF é a gestão de seus jogos estudantis, que atualmente é feita com auxílio de formulários manuais e planilhas eletrônicas. Neste contexto, esta monografia propõe o SIGOUFF - Sistema Gerenciador das Olimpíadas da UFF, um aplicativo que controla todas as atividades concernentes a gestão das Olimpíadas estudantis. A intenção deste trabalho é facilitar a aprendizagem de *Ruby on Rails* através do desenvolvimento de um sistema real.

Palavras-chave: *Ruby on Rails*. Gestão. Olimpíadas estudantis.

Abstract

The concept of agile development has becoming a growing concern in the Computer Science area. Nowadays, the Ruby on Rails framework is presenting itself as a very productive option for web development, bringing together resource strenght and agility in the software development process. A pratical problem found at UFF is the management of its student games, that is today made with the aid of manual forms and eletronic spreadsheets. In this context, this work proposes the SIGOUFF - UFF Olympics Management System, an application that controls all activities concerning the management of the student olympics. The intention of this work is to facilitate the learning of Ruby on Rails through the development of a real system.

Keywords: Ruby on Rails. Management. Student Olympics.

Capítulo 1

Introdução

1.1 Visão geral e organização do trabalho

O presente trabalho monográfico tem a finalidade de demonstrar como o processo de compreensão do *framework Ruby on Rails* pode ser facilitado através do aprendizado orientado pelo desenvolvimento de um *software*, no caso, um *software* gerenciador de Olimpíadas estudantis.

Esta monografia está dividida em quatro capítulos.

Este capítulo apresenta uma visão geral do trabalho e sua organização, as motivações existentes para o seu desenvolvimento e seus objetivos.

O capítulo 2 detalha a história da linguagem *Ruby* e sua motivação. Após a descrição da estrutura da linguagem, será apresentado o *framework Rails*, e serão explicitados seus componentes. Por fim será feito um apanhado das principais características de *Ruby on Rails*.

O capítulo 3 aborda o desenvolvimento do Sistema Gerenciador de Olimpíadas Estudantis que foi elaborado para a implementação deste trabalho. Será apresentada a realidade do gerenciamento das Olimpíadas Estudantis na UFF. Baseados nessa realidade, serão apresentados os Diagramas de casos de uso e o Diagrama de classes referentes ao sistema. Alguns casos de uso serão descritos de forma detalhada. Será ilustrada sua interface e será descrita a reação de usuários ao produto gerado nesse processo de desenvolvimento.

Por fim, o capítulo 4 finaliza esta monografia apresentando a relação ocorrida entre os autores deste trabalho e o framework *Ruby on Rails*. Serão explicitadas as facilidades e dificuldades encontradas no processo de desenvolvimento e os trabalhos futuros que

podem enriquecer os objetivos desta monografia.

1.2 Necessidade

1.2.1 Construtivismo

O Construtivismo é uma das correntes teóricas empenhadas em explicar como a inteligência humana se desenvolve, partindo do princípio de que o desenvolvimento da inteligência é determinado pelas ações mútuas entre o indivíduo e o meio. [1] afirma que *"Construtivismo significa isto: a idéia de que nada, a rigor, está pronto, acabado, e de que, especificamente, o conhecimento não é dado, em nenhuma instância, como algo terminado. Ele se constitui pela interação do indivíduo com o meio físico e social, com o simbolismo humano, com o mundo das relações sociais; e se constitui por força de sua ação e não por qualquer dotação prévia, na bagagem hereditária ou no meio, de tal modo que podemos afirmar que antes da ação não há psiquismo nem consciência e, muito menos, pensamento."*

"Entendemos que construtivismo na Educação poderá ser a forma teórica ampla que reúna as várias tendências atuais do pensamento educacional. Tendências que têm em comum a insatisfação com um sistema educacional que teima (ideologia) em continuar essa forma particular de transmissão que é a Escola, que consiste em fazer repetir, recitar, aprender, ensinar o que já está pronto, em vez de fazer agir, operar, criar, construir a partir da realidade vivida por alunos e professores, isto é, pela sociedade – a próxima e, aos poucos, as distantes. A Educação deve ser um processo de construção de conhecimento ao qual ocorrem, em condição de complementaridade, por um lado, os alunos e professores e, por outro, os problemas sociais atuais e o conhecimento já construído ('acervo cultural da Humanidade')."

"Construtivismo, segundo pensamos, é esta forma de conceber o conhecimento: sua gênese e seu desenvolvimento – e, por conseqüência, um novo modo de ver o universo, a vida e o mundo das relações sociais."

Um exemplo de processo construtivista pode ser visto nas sessões de *Coding Dojo* ocorridas semanalmente na UFF, onde programadores praticam técnicas de Programação em pares e de Desenvolvimento Orientado a Testes, se revezando em duplas para solucionar um dado problema de programação [15]. Os partipantes relatam que o aprendizado das linguagens de programação abordadas nas diferentes sessões é facilitado pela abordagem

prática dessas linguagens em problemas reais.

Por tudo isso, se viu relevante e justificada uma abordagem para o aprendizado do *framework Ruby on Rails* que envolvesse a prática do processo de construção de um *software*, de maneira que essa prática facilitasse a absorção das metodologias e da sintaxe utilizadas por essa linguagem.

1.3 Motivacão

1.3.1 Objetivos

Este trabalho tem por objetivo descrever os primeiros estudos sobre o *framework Ruby on Rails*, de tal forma que, no processo de aprendizagem, poderá se fazer um contato geral com a linguagem, detalhar suas vantagens, desvantagens e dificuldades. O estudo é integrado pelo desenvolvimento de um sistema real que resolve um problema na Universidade Federal Fluminense, que é a gestão das Olimpíadas Estudantis.

Capítulo 2

O framework Ruby on Rails

2.1 Histórico

A linguagem *Ruby* [5] foi concebida em 1995 por Yukihiro "matz" Matsumoto. Sua proposta foi de aliar programação funcional e programação imperativa, incorporando à linguagem os princípios de orientação a objeto, tipagem forte e dinâmica.

2.2 Motivacão

Matz [12] desejava criar uma linguagem que soasse o mais natural possível, vislumbrando uma comunicação expressiva, de tal forma que a leitura e o entendimento da linguagem fossem facilitados. Por muitas vezes Matz se referia à máxima: "Tentar tornar a *Ruby* natural, não simples".

O pensamento inicial de Matz nasceu de uma necessidade. O próprio afirmava: "Eu queria uma linguagem de *script* que fosse mais poderosa do que *Perl*, e mais orientada a objetos do que *Python*. É por isso que eu decidi desenvolver minha própria linguagem.".

2.3 Estrutura da linguagem

A linguagem *Ruby* tem sua estrutura muito bem definida e apoiada em algumas características, as quais a diferencia de outras linguagens [12]. A seguir faremos uma compilação das qualidades mais relevantes da linguagem *Ruby*.

- A linguagem *Ruby* é puramente baseada em objetos, pois cada parte do código está apta a receber suas próprias propriedades e ações. Explicando um pouco melhor,

pode-se entender que as propriedades serão as variáveis de instância e as ações serão os métodos. Esta característica pode ser demonstrada pela figura 2.1.

```
1 3.times {``Eu adoro Ruby!!''}
```

Figura 2.1: Exemplo de código *Ruby*

Essa ação é possível em *Ruby*. Contudo, em muitas linguagens, números e outros tipos primitivos não são objetos. Esse comportamento foi herdado da linguagem *SmallTalk* e facilita o aprendizado e uso da linguagem *Ruby*, pois tudo que se aplica a objetos, se aplica em *Ruby*.

- A *Ruby* é tida como uma linguagem flexível, visto que permite alteração em partes da linguagem. Ou seja, seus utilizadores têm toda liberdade para remover ou redefinir códigos essenciais da própria linguagem. Esta medida visa expandir o escopo de ações possiveis dos programadores ao utilizar a linguagem *Ruby*.

  Consoante dito, tal recurso pode ser exemplificado como na figura 2.2.

```
1 class Numeric
2   def plus(x)
3     self.+(x)
4   end
5 end
```

Figura 2.2: Exemplo de código *Ruby*

Verifica-se acima que uma função de soma pode ser chamada de uma forma diferente da habitual utilizada pelos programadores. Desta maneira, a chamada à função seria como exemplificado na figura 2.3.

```
1 Y = 10.plus 2
```

Figura 2.3: Exemplo de código *Ruby*

- Os operadores do *Ruby* (+, - *etc*) são *"syntax sugar"* para os métodos e também podem ser redefinidos.

- Os blocos em *Ruby* são vistos como um bom exemplo de flexibilidade. A linguagem suporta alteração de partes essenciais de sua estrutura. Em *Ruby*, a adição é realizada com o operador mais (+). Como ilustrado na figura 2.2, o desenvolvedor tem a prerrogativa de modificar essa operação para "*plus*", por exemplo, adicionando esse método à classe nativa do *Ruby Numeric* [5].

  Existem outros métodos em *Ruby* os quais deixam em aberto ao programador a possibilidade de escrever e definir seus comportamentos, permitindo a alteração de seus códigos de maneira a modificar seus funcionamentos;

- Um ponto importante a ser notado na linguagem *Ruby* é que não há necessidade de declaração de variáveis. Utiliza-se uma simples convenção, a seguir:

  - var pode ser uma variável local;
  - @var é uma variável de instância;
  - $var é uma variável global.

  Tal convenção facilita a leitura e o entendimento do código, permitindo ao desenvolvedor facilmente identificar o papel da variável em questão;

- Para facilitar o tratamento de erros, *Ruby* tem a capacidade de gerenciar exceções, tal como ocorre com a linguagem Java;

- Caso o sistema operacional permita, é suportado o carregamento de extensões de bibliotecas;

- A linguagem oferece excelente portabilidade, pois pode ser utilizada em diversos ambientes, tais como UNIX, Mac OS X, Windows 95/98/Me/NT/2000/XP *etc*;

- Seu sistema de *threading* é independente do sistema operacional em que está sendo utilizada, o que otimiza o processamento das aplicações.

2.4 O framework Rails

O Rails [9] é um *framework* de desenvolvimento *web* escrito na linguagem *Ruby*. Tem sido muito apreciado por implementar uma nova perspectiva no desenvolvimento de

aplicações, facilitando o processo de programação.

Esse *framework* objetiva o aumento de produtividade no desenvolvimento de aplicações, conciliando grandes volumes de resultados com redução de tempo de programacão. Para que resultados satisfatórios sejam obtidos, alguns princípios são necessários, a seguir:

- DRY – Transmite a ideia de que escrever o mesmo código várias vezes torna o mesmo deselegante;
- REST (ou RESTful) - Técnica que dita um modelo para aplicações *web*. Esse modelo organiza as aplicações em torno do padrão HTTP [16].

2.4.1 MVC

O framework *Rails* respeita um padrão de arquitetura de *software* que dita um modelo de organização de código específico, onde são separadas as lógicas de negócio e de apresentação (que representa a interface para o usuário). Esta medida torna o código de um sistema melhor definido e sua manutenção e leitura facilitadas. Tal arquitetura chama-se Modelo-Visão-Controlador (*Model-View-Controller*), comumente chamada de MVC. A MVC facilita o uso do DRY, pois estimula a modularização de código. A seguir será detalhada a estrutura desta arquitetura:

Modelo (*Model*)

É o modelo da arquitetura MVC que institui as regras para manipular os dados de uma aplicação. Na perspectiva do *Rails*, ele gerencia a relação entre a aplicação e o banco de dados. Uma tabela do banco de dados usualmente representa um *model* na aplicação. A lógica de negócio da aplicação ficará, em grande parte, concentrada nos *models*. O modelo de equipe do SIGOUFF pode ser visto na figura 2.4.

Visão (*View*)

Esta face da arquitetura MVC representa a interface de usuário na aplicação. Na perspectiva do *Rails*, as *views* gerenciam a função de fornecer dados para o navegador *web*. Normalmente ela é implementada através de arquivos HTML, incrementados com código *Ruby*. Resumidamente, é responsável pela apresentação de dados ao usuário.

```
1 class Equipe < ActiveRecord::Base
2   belongs_to :curso
3   belongs_to :categoria
4
5   has_many :atletas
6 end
```

Figura 2.4: Modelo de equipe do SIGOUFF

Controlador (*Controller*)

Por fim, é o controlador que implementa a relação entre cada *model* e *view*. Na perspectiva do *Rails*, é responsável por delegar o que cada *view* deve apresentar, a partir de dados fornecidos pelos *models*. Sendo assim, o *controller* processa as requisições que chegam do navegador *web* para a aplicação.

2.4.2 Componentes do *Rails*

Em [2] estão detalhados os diversos componentes do *framework Rails*. Serão compilados a seguir os componentes mais adequados para apresentação em um primeiro contato com a linguagem:

Action Controller

Trata-se do componente responsável pelo gerenciamento dos *controllers* em uma aplicacação *Rails*. O *Action Controller* processa requisições que chegam para uma aplicação *Rails* e define o caminho para a ação pretendida. Ele oferece diversos serviços relacionados ao *controller* da aplicação, como, por exemplo, renderizar um objeto.

Action View

Componente responsável pelo gerenciamento das *views* de uma aplicação *Rails*. A partir deste componente realiza-se a renderização de objetos aninhados e parciais (denominados *partials*, assunto que será visto mais adiante neste capítulo).

Active Record

Tem por competência gerenciar os *models* de uma aplicação *Rails*. O *Active Record* fornece funcionalidades CRUD básicas, independência de bancos de dados e serviços como relacionamentos entre *models*.

Action Mailer

Responsável pela gestão dos serviços de correio eletrônico. Esse componente é utilizado para projetar as ações referentes ao processamento de *e-mails*.

Active Resource

Fornece um gerenciamento de conexão entre objetos de negócio e serviços *web* RESTful. Este componente mapeia recursos para objetos locais com semântica CRUD.

Railties

Esse componente estrutura o núcleo do código que constrói novas aplicações *Rails*. Ele é responsável por orquestrar os diversos *frameworks* em uma aplicação *Rails*.

Active Support

Trata-se de uma coleção de classes utilitárias e bibliotecas padrão *Ruby* que são usadas no *Rails*.

2.5 *Ruby on Rails*

A seguir serão descritas informações relevantes sobre as características e passos para o desenvolvimento de uma aplicação utilizando *Ruby on Rails*.

2.5.1 Criação do diretório do *Rails*

Para a criação de uma nova aplicação baseada em *Ruby on Rails*, basta que no terminal seja digitado o comando como na figura 2.5.

Em seguida, automaticamente será criada uma estrutura de diretórios composta de arquivos, os quais farão parte da configuração inicial da aplicação. A estrutura de diretórios gerada após esse comando está explicitada abaixo:

```
1 rails 'nome_do_projeto'
```

Figura 2.5: Criação do diretório do *Rails*

- app – Esse diretório abriga a maior parte do código gerado, além de enquadrar a arquitetura MVC;
- config - Configurações da aplicação;
- db – Banco de dados, documentação e esquemas;
- doc – Documentação do sistema;
- lib – Bibliotecas;
- log – Informações de log;
- public – css, imagens e *javascripts* (arquivos públicos introduzidos via *web*);
- *script* – *scripts* pré-definidos do *Rails*;
- *test* – Testes unitários, funcionais e de integração;
- tmp – Pasta temporária, que abriga, por exemplo, dados em *cache* e informações de sessão;
- vendor – *Plugins* e programas externos.

Esta estrutura permite ao programador uma visão bem ampla e organizada da sua aplicação.

2.5.2 Banco de dados

A interface com bancos de dados é um dos grandes atrativos desse *framework*, a medida que o *Rails* apresenta facilidade nesse sentido. Tal facilidade é vista no momento em que é criada uma nova aplicação, pois, nesse instante, é criado também um arquivo de configuração com bancos de dados. O banco de dados SQLite3 é a base padrão usada pelo *Rails*, tendo em vista a simplicidade do seu funcionamento.

Tal arquivo, database.rb, contém a configuração de três ambientes. São eles: desenvolvimento (*development*), teste (*test*) e produção (*production*). Cada ambiente é usado

em uma etapa do sistema. O primeiro é utilizado na fase de desenvolvimento da aplicação, o segundo se refere à base de dados utilizada para executar os testes automatizados do sistema e o último é usado em produção.

2.5.3 Scaffold

O *framework Rails* oferece a opção de uso do serviço scaffold. Ele faz a geração de toda a estrutura do MVC para uma aplicação. Além disso o scaffold também gera operações básicas, tais como listagem, inserção, deleção, exclusão e visualização, para os dados presentes nas tabelas do banco de dados em questão.

Além dessas medidas, que por si já muito reduzem o tempo para desenvolvimento de uma aplicação, o scaffold também constrói os *helpers*, testes e o *layout* do sistema, e os organiza em seus devidos lugares dentro do projeto da aplicacão. Também são gerados os *scripts* de criação das tabelas referentes aos modelos de dados criados.

O scaffold é um alicerce muito poderoso para o desenvolvedor. Contudo, é aconselhado apenas para aqueles que experimentam o *framework* pela primeira vez, na medida em que facilita um primeiro contato com esse ambiente de programação. Nas últimas versões do *Rails* o scaffold foi marcado como *deprecated* - ou seja, teve sua utilização desaconselhada, pois será descontinuado. A alegação é que o serviço gera muito código que muitas vezes sequer é utilizado.

2.5.4 Geração dos modelos

A geração de modelos é uma das funcionalidades mais simples do *Rails*. Basta o uso de um comando no terminal e o *framework* executa um *script* de criação de modelos. Tal comando é ilustrado na figura 2.7.

```
1 rake db:migrate
```

Figura 2.6: Geração de modelos do *Rails*

Para cada modelo, o *Rails* gera um campo de nome “id”, responsável por ser a chave primária de cada modelo, além dos campos “data de criação” e “data da última atualização”, os quais são automaticamente preenchidos.

2.5.5 Relacionamentos entre modelos

Para se gerar um relacionamento entre modelos, é preciso informar ao *Rails* o tipo de relacionamento desejado. Quando isso é realizado, o *Rails* automatiza o processo de manipulação desses elementos. Os tipos de relacionamentos que o *Rails* disponibiliza são:

- tem_muitos *(has_many)* – Associação que gera um mapeamento voltado para uma relação de um-para-muitos (*one-to-many*) entre dois modelos;
- pertence_a (*belongs_to*) – Associação que provê que um modelo terá como um de seus atributos o id de outro modelo. Isso implicitamente significa que o modelo pertencerá a outro modelo. Sendo assim, será gerada uma relação de muitos-para-um ou de um-para-um (*many-to-one* ou *one-to-one*);
- tem_e_pertence_a_muitos (*has_and_belongs_to_many*) – Associação que provê uma relação de muitos-para-muitos (*many-to-many*);
- tem_um (*has_one*) – Associação bidirecional que provê uma relação de um-para-um (*one-to-one*).

2.5.6 Documentação do *Rails*

Todo código-fonte em *Ruby* gera uma documentação, denominada RDoc. A documentação da versão corrente do *Ruby on Rails* está disponível em diversos *sites*, geralmente diferenciados apenas pela abordagem [9] e [12]. Devido à rápida dinâmica da linguagem é fortemente recomendada a consulta a esse tipo de fonte.

O RubyGems é outra fonte que disponibiliza uma documentação para o *Rails*, contudo voltada para as gems. Esse conceito será especificamente abordado mais adiante neste capítulo. A figura 2.7 mostra o comando no terminal para inicialização do servidor de documentação, que instala a documentação das gems em uma aplicação.

Após a inicialização do servidor, será possível, via navegador, obter acesso aos RDocs de todas as gems instaladas.

Como base para utilização do *framework Ruby on Rails*, outra documentação importante é a da linguagem *Ruby*, assim como a da sua biblioteca padrão [7], [10] e [14].

Como já dito, a veloz dinâmica de evolução da linguagem favorece a consulta de fontes virtuais em detrimento da literatura tradicional, que sofre um processo rápido e contínuo de defasagem. Para os programadores iniciantes que desejam um primeiro

```
1 gem server
2
3
4 # -> Starting gem server on http://localhost:8808/
```

Figura 2.7: Inicialização do servidor de documentação do *Rails*

contato com *Ruby on Rails*, existem guias na *internet* disponíveis gratuitamente, como em [2].

Outra opção razoável para o aprendizado de *Ruby on Rails* são os *screencasts*, que são vídeos que contém aulas, palestras ou conferências sobre o *framework* ou sobre alguma vertente relacionada ao assunto. É possível encontrar materiais de excelência disponíveis *online*. Em [11] são disponibilizados gratuitamente pequenos vídeos, com duração de 15 a 45 minutos, em média. Vídeos de maior duração, porém pagos, podem ser encontrados em [8]. Em [3] estão disponíveis diversas conferências sobre *Ruby on Rails* e *screencasts* de diversas fontes podem ser obtidos em [13].

2.5.7 Servidor

Após a criação de uma aplicação, o *framework* disponibiliza ao usuário um servidor, que estabelece uma comunicação capaz de receber requisições *web* e de interpretar código *Ruby*.

O servidor é inicializado pelo terminal, através do comando *rails server*, e o acesso é feito através da URL http://localhost:3000.

2.5.8 Controlador

É o serviço responsável por dar sentido a uma requisição ao sistema, executando e direcionando as ações nele disponíveis. A partir dos métodos presentes no controlador será realizada a execução de uma determinada ação. Dessa forma, o controlador também se mostra capaz de redirecionar o navegador pelas páginas da aplicação. Normalmente as páginas retornadas são em formato HTML, entretanto o controlador pode gerar páginas em XML e em outras extensões.

O método que será executado no controlador é definido via URL. Os métodos

notificados via URL são denominados de ações (*actions*). O controlador de equipe do SIGOUFF pode ser visto nas figuras 2.8, 2.9 e 2.10.

```
1  class EquipesController < ApplicationController
2    before_filter :usuario_logado?
3    # GET /equipes
4    # GET /equipes.json
5    def index
6      @equipes = Equipe.all
7      #     debugger
8      respond_to do |format|
9        format.html # index.html.erb
10       format.json { render :json => @equipes }
11     end
12   end
13
14   # GET /equipes/1
15   # GET /equipes/1.json
16   def show
17     @equipe = Equipe.find(params[:id])
18     respond_to do |format|
19       format.html # show.html.erb
20       format.json { render :json => @equipe }
21     end
22   end
23
24   # GET /equipes/1/edit
25   def edit
26     @equipe = Equipe.find(params[:id])
27   end
```

Figura 2.8: Controlador de equipe do SIGOUFF

2.5.9 Visão

São as interfaces do sistema com o usuário. Internamente são compostas de comandos HTML e códigos *Ruby*. Neste contexto, para se utilizar um código *Ruby* é necessária a inserção de caracteres especiais, a fim de diferenciar em um arquivo de visão as seções em HTML das seções em *Ruby*. Exemplos de caracteres especiais de *Ruby* são os delimitadores ilustrados na figura 2.11.

As visões são compostas por arquivos de extensão ERb (implementação de eRuby, que já acompanha a linguagem *Ruby*). Por conterem comandos HTML, esses arquivos são considerados recipientes (*templates*), onde elementos advindos do controlador são dinamicamente inseridos, em tempo de renderização.

Uma visão deve ter o mesmo nome de sua ação descrita no controlador, para que permita ao *framework*, a partir da sua URL, montar a relação entre uma ação e uma visão. A figura 2.12 ilustra a visão referente à listagem de equipes cadastradas no SIGOUFF.

O detalhamento de uma equipe é outro exemplo de visão do SIGOUFF, que está representado na figura 2.13.

2.5.10 Ajudante (*Helper*)

Este módulo disponibiliza métodos que serão usados nas visões. O intuito principal é promover o encapsulamento de código, estimulando o reuso. Ou seja, o ajudante busca otimizar a escrita do programador.

Dessa forma, o ajudante simplifica a utilização de visões, ao garantir que todo método escrito no mesmo esteja disponível na respectiva visão. Existe um caso especial, denominado application_helper.rb, cujos métodos ficam visíveis para todas as visões de uma aplicação.

O método helper_method permite que um método de um dado controlador passe a ser do tipo ajudante e, com isso, passe a ser visível em uma determinada visão.

2.5.11 Parcial (*Partial*)

Parcial é um grande exemplo do uso do conceito de DRY. Trata-se de fragmentos de código que seriam escritos por diversas vezes, contudo, ao serem encapsulados em um arquivo de extensão “html.erb”, passam a poder ser renderizados em qualquer visão.

Para definir um arquivo como um Parcial, é obrigatório que o seu nome inicie com o caracter “_”.

Os parciais implementam a reutilização de código em *Rails*. Ao torná-lo mais enxuto, sua manutenção e legibilidade são otimizadas.

A figura 2.14 ilustra o uso de um parcial no SIGOUFF, relacionado à visão de equipes.

2.5.12 *Layout*

O serviço de *layout* remete à ideia de parcial, pois um parcial deve ser declarado em todas as páginas em que se deseja utilizá-lo. Nessa linha de raciocínio, pode-se aferir que, caso seja necessário um conteúdo que esteja presente em todas as páginas do sistema, a melhor alternativa para essa abordagem de implementação é o serviço de *layout*.

Ainda que cada controlador possa ter o seu *layout* e que qualquer alteração reflita nas visões desse controlador, existe um *layout* que funciona para todos os controladores, que é representado pelo arquivo application.html.erb.

As figuras 2.15 e 2.16 ilustram o uso deste serviço no SIGOUFF.

2.5.13 Rotas

Rotas são entradas no arquivo config/routes.rb que dizem ao *Rails* como mapear as requisições HTTP que chegam para a ação dos controladores.

No *Rails*, um fator interessante é que as URLs das rotas podem ser usadas para capturar parâmetros.

A funcionalidade de roteamento embutida no *Rails* é bastante poderosa e o uso de rotas permite uma grande flexibilidade para a criação de URLs.

O roteamento traçado para cada uma das URLs disponíveis em uma aplicação *Rails* pode ser visto com a ajuda de um comando digitado no terminal. A figura 2.17 ilustra este comando.

Esta funcionalidade facilita bastante o trabalho do desenvolvedor quando se há dúvidas em relação a navegabilidade do sistema. Um correto roteamento garante ao desenvolvedor que uma determinada ação de um controlador está realmente sendo chamada quando requisitada.

O uso de rotas no SIGOUFF pode ser aferido através da figura 2.18.

2.5.14 AJAX

O *framework Ruby on Rails* é voltado para o desenvolvimento *web*, então é natural que exista a presença da linguagem de *script javascript* em seu conjunto de funcionalidades.

Assim como o uso de *javascript*, o uso de AJAX também foi incorporado ao *Ruby on Rails*, pois uma aplicação *web* necessita de uma interface o mais interativa e intuitiva possível.

Existe um ajudante especifico para o uso de *javascript* com *Ruby on Rails*, o qual possui algumas alternativas para otimizar o uso dessa linguagem.

2.5.15 Gems

Ruby conta com um suporte que possibilita a instalação e o *download* de ferramentas escritas nessa linguagem, tornando simples o uso de bibliotecas e a reutilização de código.

Esse suporte é chamado de RubyGems e essas ferramentas são disponibilizadas em forma de gems. Após a instalação da linguagem *Ruby*, a instalação do RubyGems é automática.

A instalação de uma gem é muito simples, como descreve a figura 2.19.

2.5.16 Testes

Nesta subseção será descrito o funcionamento dos testes utilizados no processo de desenvolvimento do *software* proposto neste trabalho monográfico.

Testes unitários

Testes unitários são criados pelo desenvolvedor para certificar-se de que o funcionamento de uma parte específica da aplicação está se comportando como o esperado. Como ilustração desse conceito, no produto gerado neste trabalho foram testadas as validações dos diversos cadastros presentes no sistema.

Essa parte do processo de desenvolvimento de *software* é uma das mais importantes, pois permite uma depuração completa de todo o sistema.

Para a execução de um teste unitário em *Ruby on Rails*, basta digitar um comando no terminal, conforme ilustra a figura 2.20.

Testes do modelo da aplicação

Para os testes referentes ao modelo implementado na aplicação, será utilizado o comando spec, o qual é gerado automaticamente pelo *Rails* durante a criação de uma aplicação. Esse teste pode ser acionado pelo terminal, através do comando ilustrado pela figura 2.21.

Para o de teste das visões utilizadas na aplicação proposta, será utilizado o cucumber, que é uma gem constantemente utilizada nos projetos que envolvem desenvolvimento em *Ruby on Rails*. Para acionar o seu uso, basta a digitação de um comando no terminal, conforme na figura 2.22.

```
1   # POST /equipes
2   # POST /equipes.json
3   def create
4     @equipe = Equipe.new(params[:equipe])
5     respond_to do |format|
6       if @equipe.save
7         format.html { redirect_to @equipe,
8 :notice => 'Equipe was successfully created.' }
9         format.json { render :json => @equipe,
10 :status => :created, :location => @equipe }
11      else
12        format.html { render :action => "new" }
13        format.json { render :json => @equipe.errors,
14 :status => :unprocessable_entity }
15      end
16    end
17  end
18  # PUT /equipes/1
19  # PUT /equipes/1.json
20  def update
21    @equipe = Equipe.find(params[:id])
22    respond_to do |format|
23      if @equipe.update_attributes(params[:equipe])
24        format.html { redirect_to @equipe,
25 :notice => 'Equipe was successfully updated.' }
26        format.json { head :ok }
27      else
28        format.html { render :action => "edit" }
29        format.json { render :json => @equipe.errors,
30 :status => :unprocessable_entity }
31      end
32    end
33  end
```

Figura 2.9: Controlador de equipe do SIGOUFF

```
1   # DELETE /equipes/1
2   # DELETE /equipes/1.json
3   def destroy
4     @equipe = Equipe.find(params[:id])
5     @equipe.destroy
6
7     respond_to do |format|
8       format.html { redirect_to equipes_url }
9       format.json { head :ok }
10    end
11  end
12 end
```

Figura 2.10: Controlador de equipe do SIGOUFF

```
1 ``<% ``Codigo Ruby'' %>''
```

Figura 2.11: Exemplo de caracteres especiais em *Ruby*

```
1  <h1>Listando as equipes</h1>
2
3  <table>
4    <tr>
5      <th>Nome</th>
6      <th>Quantidade integrantes</th>
7      <th>Categoria</th>
8      <th>Curso</th>
9      <th colspan="3">Atividades</th>
10   </tr>
11
12 <% @equipes.each do |equipe| %>
13   <tr>
14     <td><%= equipe.nome %></td>
15     <td><%= equipe.quantidade_integrantes %></td>
16     <td><%= equipe.categoria.nome %></td>
17     <td><%= equipe.curso.nome %></td>
18     <td><%= link_to 'Show', equipe %></td>
19     <td><%= link_to 'Edit', edit_equipe_path(equipe) %></td>
20     <td><%= link_to 'Destroy', equipe, :confirm => 'Are you sure?',
21 :method => :delete %></td>
22     <td><%= link_to 'Atletas', equipe_atletas_path(equipe) %></td>
23   </tr>
24
25 <% end %>
26 </table>
27 <br />
28 <%= link_to 'New Equipe', new_equipe_path %>
```

Figura 2.12: Visão das listagens de equipes cadastradas no SIGOUFF

```
1  <p id="notice"><%= notice %></p>
2
3  <p>
4    <b>Nome:</b>
5    <%= @equipe.nome %>
6  </p>
7
8  <p>
9    <b>Quantidade integrantes:</b>
10   <%= @equipe.quantidade_integrantes %>
11 </p>
12
13 <p>
14   <b>Categoria:</b>
15   <%= @equipe.categoria_id %>
16 </p>
17
18 <p>
19   <b>Curso:</b>
20   <%= @equipe.curso_id %>
21 </p>
22
23 <%= link_to 'Edit', edit_equipe_path(@equipe) %> |
24 <%= link_to 'Back', equipes_path %> |
25 <%= link_to 'atleta', new_equipe_atleta_path(@equipe) %>
```

Figura 2.13: Visão do detalhamento de uma equipe no SIGOUFF

```
1  <%= form_for(@equipe) do |f| %>
2    <% if @equipe.errors.any? %>
3      <div id="error_explanation">
4        <h2><%= pluralize(@equipe.errors.count, "error") %> prohibited this
5  equipe from being saved:</h2>
6        <ul>
7        <% @equipe.errors.full_messages.each do |msg| %>
8          <li><%= msg %></li>
9        <% end %>
10       </ul>
11     </div>
12   <% end %>
13   <div class="field">
14     <%= f.label :nome %><br />
15     <%= f.text_field :nome %>
16   </div>
17   <div class="field">
18     <%= f.label :quantidade_integrantes %><br />
19     <%= f.text_field :quantidade_integrantes %>
20   </div>
21   <div class="field">
22     <%= f.label :categoria_id %><br />
23     <%= select("equipe", "categoria_id",
24  Categoria.all.collect {|p| [ p.nome, p.id ] }, :include_blank => true) %>
25   </div>
26   <div class="field">
27     <%= f.label :curso_id %><br />
28     <%= select("equipe", "curso_id",
29  Curso.all.collect {|p| [ p.nome, p.id ] }, :include_blank => true) %>
30   </div>
31   <div class="actions">
32     <%= f.submit %>
33   </div>
34 <% end %>
```

Figura 2.14: Exemplo de parcial com uma visão de equipes no SIGOUFF

```
1  <html>
2    <head>
3      <title>SIGOUFF</title>
4      <link href='http://fonts.googleapis.com/css?family=Pacifico'
5  rel='stylesheet' type='text/css'></link>
6      <%= stylesheet_link_tag "barra_de_aplicacoes" %>
7      <%= stylesheet_link_tag "barra_do_governo" %>
8      <%= stylesheet_link_tag "cores_azul" %>
9      <%= stylesheet_link_tag "estiloLayout" %>
10     <%= stylesheet_link_tag "reset" %>
11     <%= stylesheet_link_tag "style" %>
12     <%= stylesheet_link_tag "tablecloth" %>
13     <%= javascript_include_tag "application" %>
14     <%= csrf_meta_tag %>
15   </head>
16
17   <body>
18
19     <div id="user_nav">
20       <% if current_user %>
21         Logged in as <%= current_user.email %>
22         <%= link_to "Log␣out", log_out_path %>
23       <% else %>
24         <%= link_to "Sign␣up", sign_up_path %> or
25         <%= link_to "Log␣in", log_in_path %>
26       <% end %>
27     </div>
```

Figura 2.15: Implementação de um layout no SIGOUFF

```
1       <div id="main">
2         <%= render "layouts/barra_governo" %>
3         <%= render "layouts/barra_de_aplicacoes" %>
4
5         <div class="header">
6           <p style="font-family: 'Pacifico', arial, serif; font-size:50px; font-
7         </div>
8
9         <div id="container">
10          <%= render "layouts/menu_vertical" %>
11
12          <div id="conteudo" >
13            <%= yield %>
14            <%= render "layouts/footer" %>
15          </div>
16        </div>
17      </div>
18    </body>
19  </html>
```

Figura 2.16: Implementação de um layout no SIGOUFF

```
1 rake routes
```

Figura 2.17: Visualização dos roteamentos de uma aplicação *Ruby on Rails*

```
1  Olimpiadas::Application.routes.draw do
2    get "sessions/new"
3    get "log_in" => "sessions#new", :as => "log_in"
4    get "log_out" => "sessions#destroy", :as => "log_out"
5    get "sign_up" => "users#new", :as => "sign_up"
6    resources :sessions
7    resources :users
8    resources :categorias
9    resources :equipes do
10     resources :atletas, :except => [:show]
11   end
12   resources :atletas, :only => [:show]
13   resources :cursos
14   root :to => "application#index"
15 end
```

Figura 2.18: Uso de rotas no SIGOUFF

```
1 gem install will_paginate
```

Figura 2.19: Instalação de uma gem em *Ruby*

```
1 ruby teste_unitario.rb
```

Figura 2.20: Chamada de um teste unitário em *Ruby on Rails*

```
1 rake spec
```

Figura 2.21: Chamada de um teste de modelo em *Ruby on Rails*

```
1 rake cucumber
```

Figura 2.22: Chamada de um teste de visão em *Ruby on Rails*

Capítulo 3

O SIGOUFF

Neste capítulo será apresentada a realidade do cenário atual abordado pelo produto proposto por esta monografia, que trata da gestão dos jogos estudantis da UFF. Será descrita a arquitetura do *software* proposto e serão ilustrados sua interface e os testes utilizados no sistema.

3.1 A Olimpíada Estudantil da UFF

A Olimpíada Estudantil da UFF é um evento em que diversos cursos da Universidade competem entre si em diversas modalidades esportivas, com o intuito de promover agradáveis experiências de interação entre a comunidade acadêmica, incentivando a prática desportiva [4]. O evento é organizado pelo corpo discente.

Dezessete modalidades esportivas compõem atualmente o evento, como mostra a tabela 3.1.

A primeira edição ocorreu em 2009 e teve em torno de 16 cursos participantes. Na segunda edição, ocorrida em 2010, o número de cursos participantes aumentou para cerca de 40 e o número de alunos participantes também aumentou. A partir desse ano foi permitida a participação de Professores e estudantes dos cursos de especialização da UFF. A terceira e recente edição, de 2011, contou com exatos 40 cursos participantes [4].

Sua organização é composta pela divisão das Olimpíadas em:

- Edições, onde cada uma faz referência a um ano;
- Modalidades, onde cada uma faz referência a um esporte;
- Atletas, onde cada um faz referência a um aluno regular da Universidade participante dos jogos;

Tabela 3.1: Tabela das modalidades dos Jogos Olímpicos da UFF

Índice	Nome da modalidade
1	Futsal Masculino
2	Futsal Feminino
3	Vôlei Masculino
4	Vôlei Feminino
5	Basquete Masculino
6	Basquete Feminino
7	Handebol Masculino
8	Handebol Feminino
9	Tênis de mesa individual
10	Tênis de mesa por equipe
11	Futebol de Campo
12	Atletismo Masculino
13	Atletismo Feminino
14	Judô Masculino
15	Judô Feminino
16	Natação Masculina
17	Natação Feminina

- Cursos, onde cada um faz referência a um curso de nível superior proferido pela Universidade;
- Equipes, onde cada uma faz referência a um conjunto de um ou mais alunos de um curso que participam de uma modalidade;
- Partidas, onde cada uma faz referência a um confronto entre duas equipes;
- Categorias, onde cada uma especifica um tipo de modalidade.

Segundo o gestor principal dos Jogos Olímpicos da UFF para o ano de 2011, José Carlos Vieira Júnior, aluno do curso de Educação Física, todo esse controle atualmente é feito da seguinte forma: o responsável por uma equipe envia um formulário via correio eletrônico com os dados da equipe e dos atletas dessa equipe. São geradas listas que descrevem as representações estudantis (vide apêndice A), classificadas por curso e gênero. Todo o restante que tange ao gerenciamento dos jogos é feito via planilhas eletrônicas e os anúncios das partidas e dos resultados é feito via correio eletrônico para os responsáveis pelas equipes, por uma comunidade na rede social *Orkut*, por um grupo na rede social *Facebook* e pelo *blog* do evento [4].

3.2 Arquitetura do SIGOUFF

Nesta seção serão descritos componentes essenciais da arquitetura do SIGOUFF: seu Diagrama de classes e seu Diagrama de casos de uso. Alguns casos de uso serão mostrados em sua forma detalhada.

3.2.1 Classes

Uma classe consiste de variáveis e métodos que representam características de um conjunto de objetos semelhantes [6]. O conceito de classe é dos pilares da programação orientada a objetos, por permitir a reutilização efetiva de código.

3.2.2 Diagrama de classes

Um diagrama de classes representa a estrutura do sistema, recorrendo ao conceito de classe e suas relações. Através da figura 3.1 é possível observar todas as classes que o sistema possui e o o relacionamento entre elas.

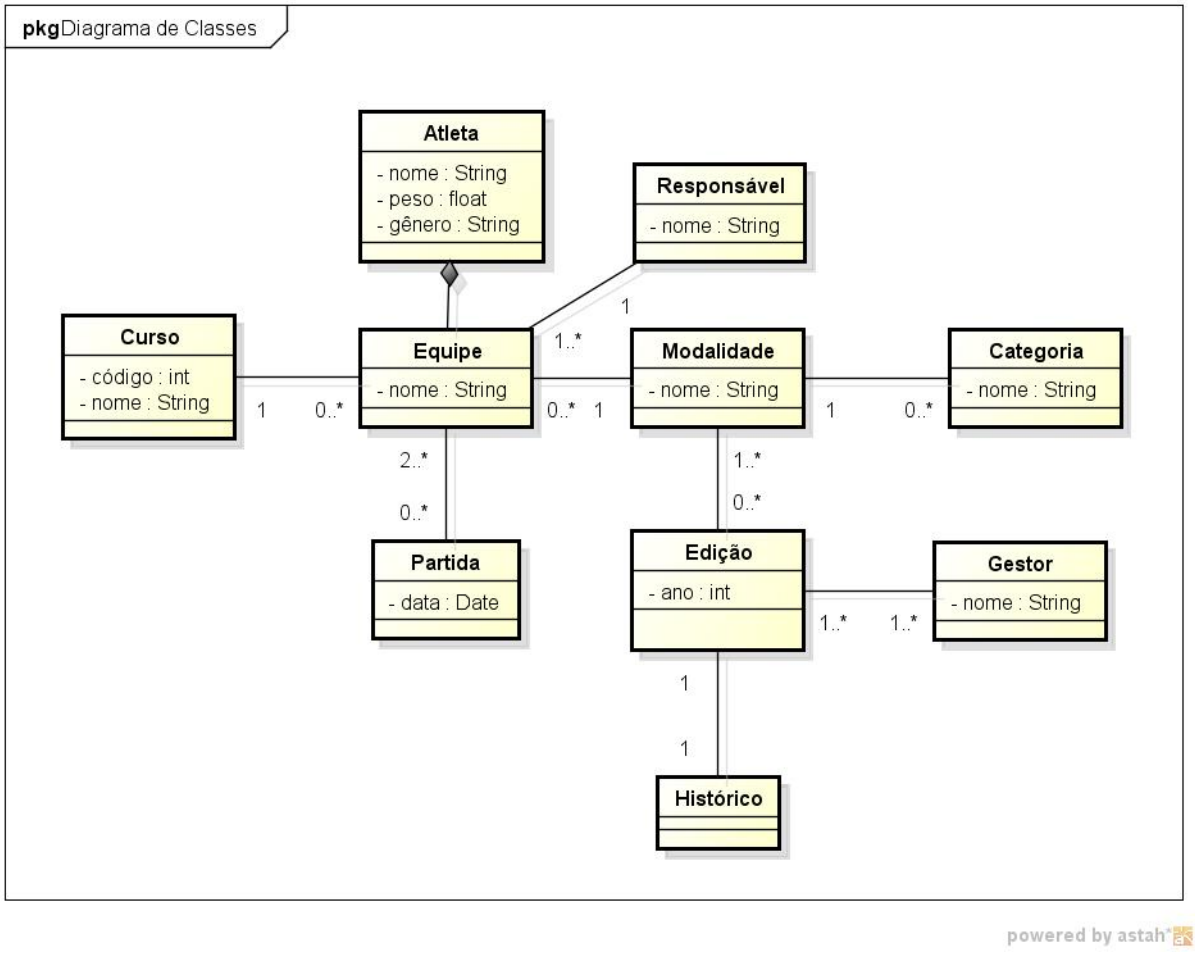

Figura 3.1: Diagrama de classes do SIGOUFF

3.2.3 Casos de uso

Caso de uso pode ser definido como uma sequência de passos a qual descreve uma interação entre um ator e o sistema.

Atores

Atores são usuários e/ou entidades externas (*hardwares* ou *softwares*) que desenvolvem algum papel em relação ao sistema. Os atores geram informações para o sistema ou necessitam de informações geradas a partir do sistema. Para o SIGOUFF foram identificados dois atores, que serão definidos a seguir:

- **Responsável**: pessoa que gerencia a participação de uma equipe na Olimpíada. Normalmente um aluno do curso a que pertencem os atletas daquela equipe.
- **Gestor**: membro do comitê organizador da Olimpíada.

3.2.4 Diagrama de casos de uso

Através da figura 3.2 serão descritos os fluxos executados pelo sistema, iniciados após estímulos dados pelos atores.

3.2.5 Casos de uso detalhados

Esta subseção apresenta alguns casos de uso do SIGOUFF, de forma detalhada.

Caso de uso "Autenticar usuário"

Este caso de uso tem por finalidade identificar o usuário que utiliza o sistema. A tabela 3.2 mostra que, através de verificação de nome de usuário e senha, o sistema verifica a autenticidade do usuário para que ele tenha acesso à aplicação.

Caso de uso "Finalizar sessão"

Este caso de uso tem por finalidade não deixar uma sessão identificada disponível para outros usuários, visando segurança. Seu fluxo de execução está descrito na tabela 3.3

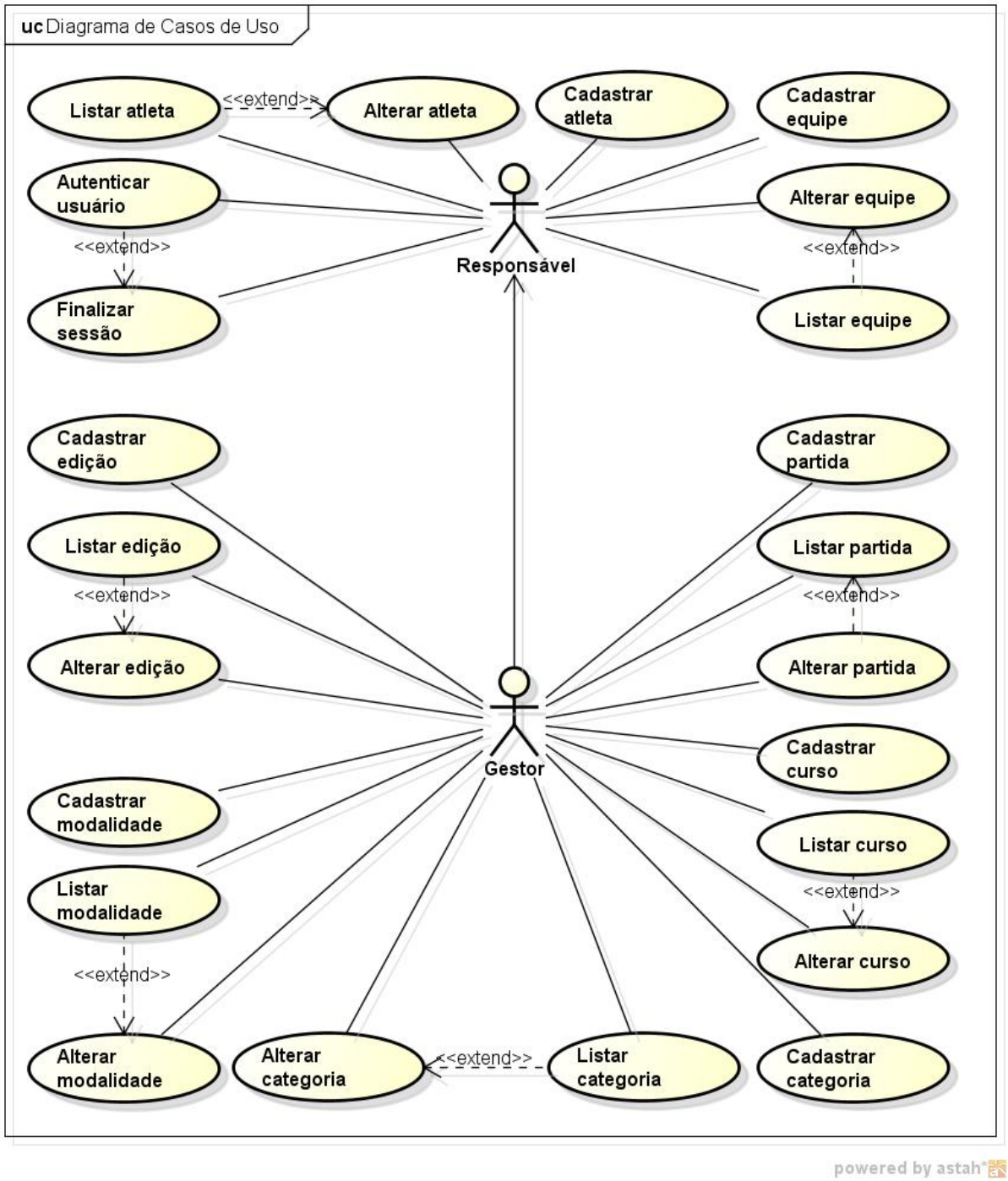

Figura 3.2: Diagrama de casos de uso do SIGOUFF

Tabela 3.2: Caso de uso “Autenticar usuário”

Atores	Responsável, Gestor
Pré-condições	• O usuário deve ter acesso ao site do sistema para realizar a autenticação.
Fluxo básico	
1. Usuário acessa o site através do navegador; 2. Sistema exibe tela inicial com a opção “Identifique-se”; 3. Usuário informa login e senha e seleciona opção “OK”; [**E1**] 4. Sistema autentica usuário e registra evento; Sistema exibe tela inicial trocando o estado do usuário para autenticado; 5. O caso de uso termina.	
Fluxo alternativo	Não se aplica
Fluxo de exceção	
[E1] Sistema não encontra login e/ou senha iguais às informadas pelo usuário.	
1. Sistema exibe a mensagem “Login e/ou senha inválidos!”; 2. Retorna ao passo 2.	
Pós-condições	• Usuário autenticado.

Tabela 3.3: Caso de uso “Finalizar sessão”

Atores	Responsável, Gestor
Pré-condições	• Execução do caso de uso “Autenticar usuário”.
Fluxo básico	
1. Sistema exibe tela principal com opção “Sair”; 2. Usuário seleciona opção “Sair”; 3. Sistema troca estado do usuário para inicial, registra evento e redireciona usuário para página de autenticação; 4. O caso de uso termina.	
Fluxo alternativo	Não se aplica
Fluxo de exceção	Não se aplica
Pós-condições	• Usuário não autenticado – estado inicial.

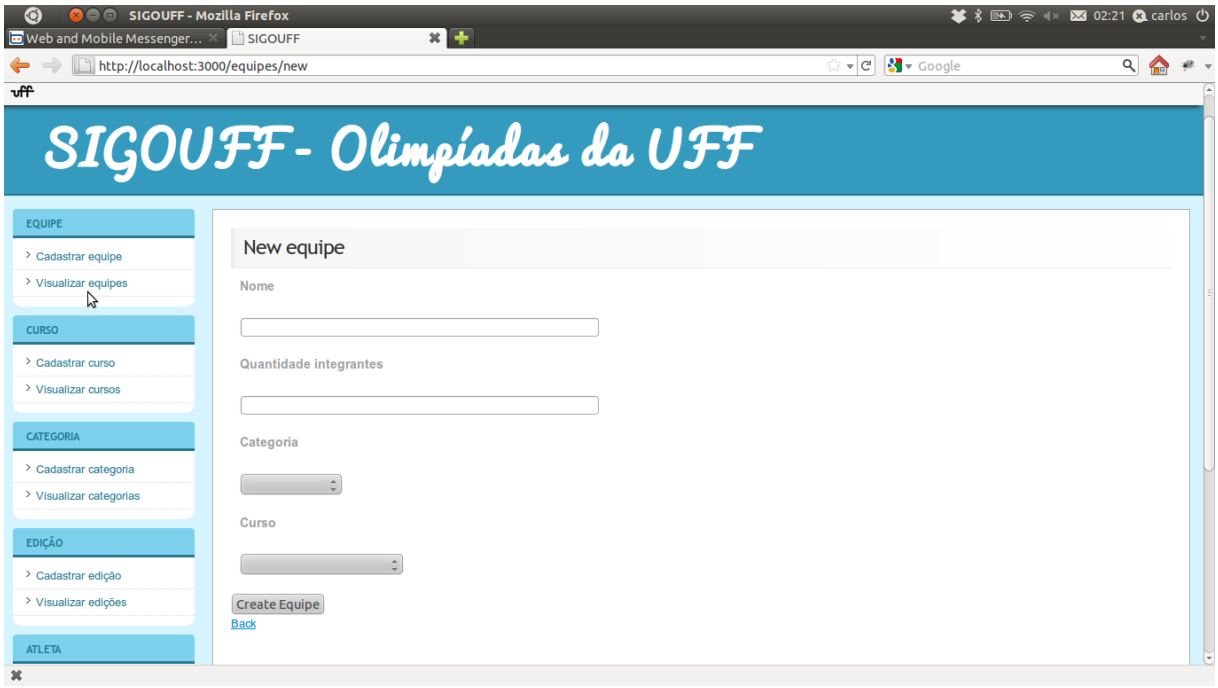

Figura 3.3: Tela de cadastro de equipe do SIGOUFF

Caso de uso "Cadastrar atleta"

Este caso de uso tem por finalidade obter todos os dados cadastrais de um atleta necessários para viabilizar sua participação nas Olimpíadas. Seu fluxo de execução está descrito na tabela 3.4.

Caso de uso "Listar atleta"

Este caso de uso tem por finalidade exibir uma lista de todos os atletas cadastrados no SIGOUFF, através de uma lista que poderá ser inicialmente filtrada por nome, matrícula, ou curso e nome. Seu fluxo de execução está descrito na tabela 3.5.

Caso de uso "Alterar atleta"

Este caso de uso tem por finalidade alterar ou incluir dados relativos a um cadastro de atleta já existente. Seu fluxo de execução está descrito na tabela 3.6.

3.3 Interface do SIGOUFF

Nesta seção serão exibidas algumas interfaces do SIGOUFF, a fim de ilustração do sistema.

A tela de cadastro de equipes do sistema é ilustrada pela figura 3.3.

Tabela 3.4: Caso de uso “Cadastrar atleta”

Atores	Responsável
Pré-condições	• Execução do caso de uso “Autenticar usuário”.
Fluxo básico	
1. Usuário autenticado; 2. Usuário seleciona opção “Cadastrar atleta”; 3. Sistema exibe tela de cadastro de atleta; 4. Usuário preenche no mínimo os dados pessoais obrigatórios e seleciona a opção “salvar alterações”; 5. Sistema valida e armazena as informações do atleta na base de dados, registra o evento e exibe tela principal do sistema; 6. O caso de uso termina.	
Fluxo alternativo	Não se aplica
Fluxo de exceção	Não se aplica
Crítica de entrada de dados	
No passo 5 do Fluxo Básico, caso existam críticas sobre as entradas de dados:	
1. O sistema deverá retornar alerta informando a descrição e localização do erro; 2. O caso de uso retorna ao passo 4 do Fluxo Básico.	
Pós-condições	• Atleta cadastrado.

Tabela 3.5: Caso de uso “Listar atleta”

Atores	Responsável
Pré-condições	• Ter atletas cadastrados na base de dados; • Execução do caso de uso “Autenticar usuário”.
Fluxo básico	
1. Usuário autenticado; 2. Sistema exibe menu inicial; 3. Usuário seleciona a opção “Listar atleta”; 4. Sistema exibe opções de filtro; 5. Usuário seleciona as opções convenientes e clica em “listar”; 6. Sistema busca na base de dados os atletas de acordo com as opções escolhidas; **[E1]** 7. Sistema exibe na tela os atletas de acordo com as opções escolhidas; 8. O caso de uso termina.	
Fluxo alternativo	Não se aplica
Fluxo de exceção	
[E1] Sistema não encontra currículos na base de dados	
1. Sistema exibe a mensagem “Nenhum currículo cadastrado”; 2. O caso de uso retorna ao passo 2 do Fluxo Básico.	
Pós-condições	- Usuário tem acesso a lista de atletas.

Tabela 3.6: Caso de uso “Alterar atleta”

Atores	Responsável
Pré-condições	• Execução do caso de uso “Autenticar usuário”. • Execução do caso de uso “Listar atleta”.
Fluxo básico	
1. Usuário autenticado 2. Usuário seleciona opção “Alterar dados de atleta”; 3. Sistema exibe tela de alteração de dados de atleta; 4. Usuário adiciona e/ou altera dados e seleciona a opção “salvar alterações”; 5. Sistema valida e armazena as informações do atleta na base de dados, registra o evento e exibe tela principal do sistema; 6. O caso de uso termina.	
Fluxo alternativo	Não se aplica
Fluxo de exceção	Não se aplica
Crítica de entrada de dados	
No passo 5 do Fluxo Básico, caso existam críticas sobre as entradas de dados:	
1. O sistema deverá retornar alerta informando a descrição e localização do erro; 2. O caso de uso retorna ao passo 4 do Fluxo Básico.	
Pós-condições	- Dados do atleta alterados.

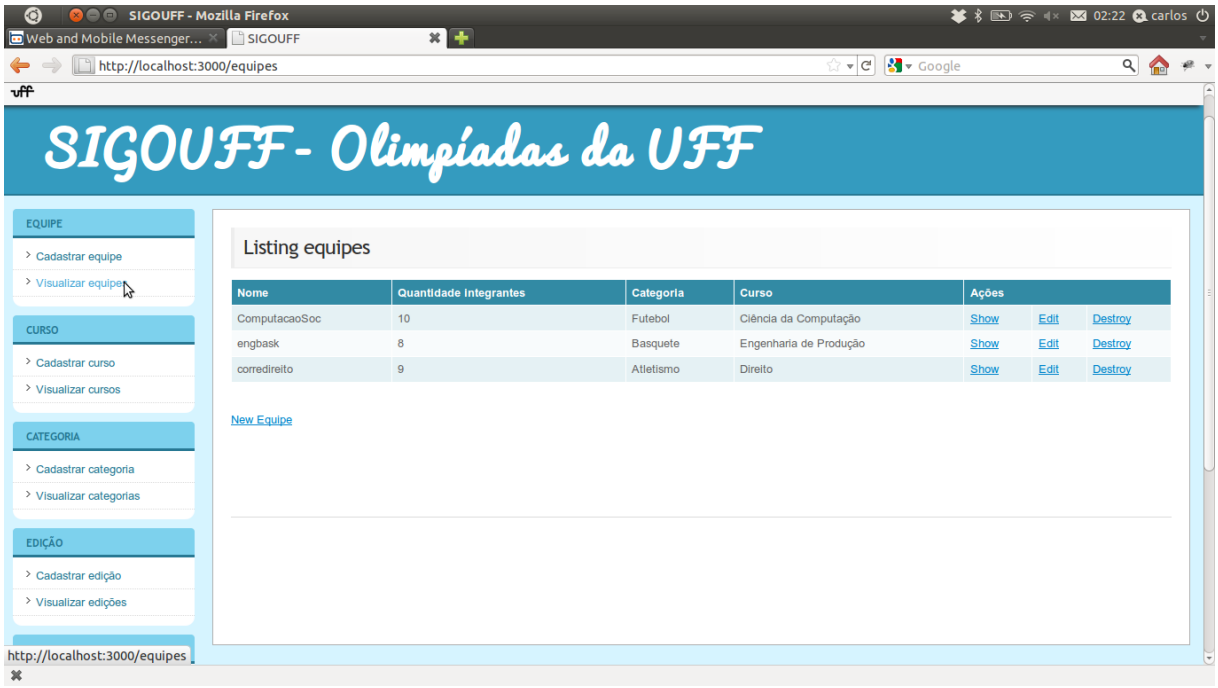

Figura 3.4: Tela de listagem de equipe do SIGOUFF

Outro exemplo de interface do SIGOUFF dá-se pela figura 3.4, que ilustra a tela do sistema associada à listagem de equipes.

Por fim, a figura 3.5 ilustra a tela do *software* relacionada à alteração de equipes.

3.4 Reação ao produto

Nesta seção será descrita a reação de um usuário final ao produto gerado, aferida por meio de duas entrevistas realizadas com estudantes da Universidade envolvidos com a Olimpíada estudantil, cujos resultados serão apresentados a seguir.

Bárbara Borges, graduanda em Ciência da Computação da UFF e uma das responsáveis pela delegação do Curso de Ciência da Computação nos jogos, testou o SIGOUFF. Segundo sua avaliação foi dado um passo importante na informatização do controle dos jogos estudantis. A estudante afirmou que a substituição do controle por planilhas trará importantes avanços para a gestão da Olimpíada, como o fim da preocupação com o controle de versão das planilhas e a disponibilidade *online* dos dados.

João Paulo Coutinho Lethier de Mello, formando em Ciência da Computação da UFF que participou de processos de desenvolvimento utilizando o *framework Ruby on Rails* no STi (Serviço de Tecnologia da Informação da UFF), onde inclusive teve seu primeiro contato com essa tecnologia, afirmou que o contato com um *software* básico

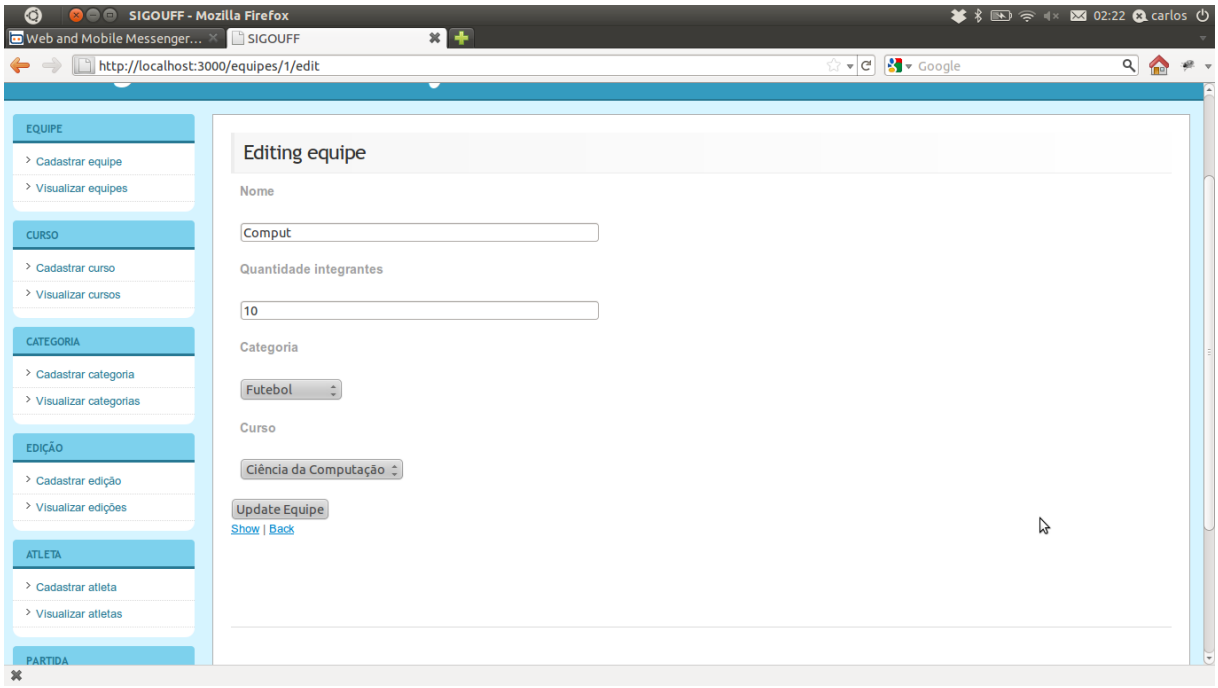

Figura 3.5: Tela de alteração de equipe do SIGOUFF

como o SIGOUFF dá a um estudante que tem o primeiro contato com o *framework Ruby on Rails* uma perspectiva mais concreta dos propósitos e do funcionamento do *framework*.

Capítulo 4

Conclusão e trabalhos futuros

Neste capitulo será descrito o resultado da experiência dos autores com o processo de aprendizagem do *framework Ruby on Rails* e possíveis soluções futuras para o enriquecimento deste trabalho.

4.1 Conclusão

No processo de aprendizagem do *framework Ruby on Rails*, considerando que os autores detinham prévio conhecimento e contato com a linguagem Java e seus *frameworks web*, foram verificadas as seguintes facilidades: o código se mostra enxuto, de maneira que, com um conhecimento mais apurado, é possível construir um elegantes soluções utilizando *Ruby on Rails*.

Foi constatado que a linguagem *Ruby* busca aproximar sua sintaxe à forma natural, fato que facilita a interação com o programador inexperiente. O desenvolvimento se mostrou mais ágil do que o usual em outras linguagens, de maneira que as expectativas de resultado de um processo de desenvolvimento são atendidas rapidamente. Verificou-se que a linguagem *Ruby* possui uma boa legibilidade, característica que facilita a manutenabilidade do processo de desenvolvimento de *software*. O contato com o *framework Ruby on Rails* foi facilitado pela prévia experiência com os *frameworks web* da linguagem *Java*, na medida em que ambas as plataformas suportam a arquitetura MVC (ou seja, o desenvolvimento em três camadas).

Como dificuldade, pode-se citar o acesso restrito a material atualizado sobre a linguagem na academia, o que estimulou os autores a buscarem conteúdo em *blogs*, sites especializados e outros meios informais. Foi verificado que a literatura têm abordado cada vez mais largamente o desenvolvimento do *framework Ruby on Rails*, do que pode-se

concluir que o *framework* encontra-se em uma curva constante de crescimento. Como dificuldade técnica, os autores podem citar a adaptação ao uso do *interactive Ruby*, ou seja, o processo de depuração do código, que utiliza linha de comando. Outra dificuldade relevante é a que é natural em todo processo de mudança, que é a adaptação a uma realidade diferente.

Por fim, com o desenvolvimento do SIGOUFF, pôde-se concluir que o *framework Ruby on Rails* permite um rápido processo de prototipação e de desenvolvimento de um produto final. Acredita-se que o produto gerado possa estimular os leitores a terem seu primeiro contato com essa promissora ferramenta de programação.

4.2 Trabalhos futuros

Este trabalho abordou o desenvolvimento de um *software* gerenciador dos jogos estudantis da UFF como base para ilustrar o aprendizado do *Ruby on Rails*. Sendo assim, existem soluções para a gestão dos jogos que se mostram necessárias, mas que não foram abordadas por não atingirem o foco do trabalho. Essas soluções estão no escopo dos trabalhos futuros.

A gestão financeira dos jogos, que inclui as taxas de inscrição pagas pelas equipes e os gastos com os custos do evento, é feita de forma manual e se faz necessária sua automação. Outras funcionalidades de grande importância já existentes manualmente que podem ser abordadas em outros trabalhos são: o controle dos cruzamentos entre as equipes durante os jogos; a divisão do torneio em etapas (oitavas de finais, quartas de finais, semi-finais e finais, ou primeiras e segundas fases, dependendo do esporte); a divisão de modalidades por grupos, onde cada grupo corresponde a um conjunto de cursos que se enfrentam para eleger o vencedor do grupo, que seguirá na competição; o controle da disposição dos locais em que ocorrerão os jogos e, por fim, relatórios mais elaborados contendo a tabela dos jogos e os resultados da competição.

Como possível aprimoramento, pode-se citar o controle das multas proferidas aos participantes, atletas e equipes, por ausência em alguma partida ou por outras penalidades. O sistema poderia registrar o lançamento dessas multas e gerenciar seu acompanhamento, por exemplo, enviando e-mails para os devedores avisando sobre o prazo e a forma de pagamento da multa. Outra evolução seria se o sistema, três dias antes e na véspera de cada partida, lembrasse automaticamente por correio eletrônico os atletas de seus compromissos.

Referências Bibliográficas

[1] F. BECKER. Série Idéias n. 20. São Paulo: FDE. Páginas: 87 a 93, 1994.

[2] Começando com *Rails*. http://guias.rubyonrails.com.br/getting_started.html, acessado em 13 de dezembro de 2011.

[3] Confreaks. http://www.confreaks.com, acessado em 13 de dezembro de 2011.

[4] Olimpíadas da UFF. http://www.olimpiadas.uff.br, acessado em 13 de dezembro de 2011.

[5] Sobre o *Ruby*. http://www.ruby-lang.org/pt/sobre-o-ruby/, acessado em 13 de dezembro de 2011.

[6] Programção OO. http://www.training.com.br/lpmaia/pub_prog_oo.htm, acessado em 13 de dezembro de 2011.

[7] Ruby-Doc.org: *Documenting the Ruby Language*. http://ruby-doc.org, acessado em 13 de dezembro de 2011.

[8] PeepCode | *Programming and Development Tutorial Screencasts for Web Developers and Alpha Geeks*. http://peepcode.com, acessado em 13 de dezembro de 2011.

[9] *Ruby on Rails Guides*. http://guides.rubyonrails.org, acessado em 13 de dezembro de 2011.

[10] *Ruby on Rails Screencasts* RailsCasts. http://railscasts.com, acessado em 13 de dezembro de 2011.

[11] *Ruby on Rails*: *Ruby on Rails* Brasil. http://rubyonrails.com.br, acessado em 13 de dezembro de 2011.

[12] RubyTu.be *Ruby Screencasts and Videos for Ruby People*. http://rubytu.be/, acessado em 13 de dezembro de 2011.

[13] *The Ruby Toolbox - Know Your Options*! http://ruby-toolbox.com/, acessado em 13 de dezembro de 2011.

[14] uff *Coding Dojo* Rio. http://dojorio.org/tag/uff/, acessado em 12 de dezembro de 2011.

[15] R. URUBATAN. *Ruby on Rails - Desenvolvimento Fácil e Rápido de Aplicações Web*. Novatec Editora, 2009.

Glossário

Framework: Abstração que une códigos comuns entre vários projetos de software provendo uma funcionalidade genérica. Em Português, se diz arcabouço.

Coding Dojo: Um *Coding Dojo* é um encontro onde um grupo de programadores se reúne para trabalhar em conjunto em um desafio de programação.

Pair programming: Programação em Pares é um método de programação em que duas pessoas trabalham juntas em uma estação de trabalho. Uma pessoa, o "piloto", digita no teclado. A outra pessoa, o observador (ou "navegador") revê cada linha de código enquanto ela é digitada, verificando erros e pensando sobre o projeto global.

Syntax sugar: É o princípio de se aproximar uma sintaxe à forma mais natural possível, facilitando seu entendimento.

Test Driven Development: Desenvolvimento Orientado a Testes ou Desenvolvimento Guiado por Testes — é uma técnica de desenvolvimento de *software* onde primeiro são criados os testes e somente depois é escrito o código necessário para passar por eles.

Linguagem de script: Linguagens de programação executadas do interior de programas e/ou de outras linguagens de programação, não se restringindo a esses ambientes.

Threading: É uma forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas concorrentemente.

HTML: *HyperText Markup Language*. Linguagem de marcação utilizada para produzir páginas na *web*.

Template: *Template* (ou "modelo de documento") é um documento sem conteúdo, com apenas a apresentação visual (apenas cabeçalhos, por exemplo) e instruções sobre onde e qual tipo de conteúdo deve entrar em cada parcela da apresentação.

XML: Formato para a criação de documentos com dados organizados de forma hierár-

quica.

HTTP: *Hypertext Transfer Protocol*. Protocolo de Transferência de Hipertexto - Utilizado para sistemas de informação de *hipermedia* distribuídos e colaborativos.

Apêndice A

Lista modelo das delegações estudantis

A figura A.1 ilustra o formulário manual atualmente utilizado como lista modelo das delegações estudantis na Olimpíada da UFF.

CURSO:

NOME DO REPRESENTANTE:

TELEFONE:

DELEGAÇÃO

MASCULINO

NOME DO ATLETA	MATRÍCULA

(NÃO GRADUANDOS)

NOME DO ATLETA	MATRÍCULA	SITUAÇÃO

FEMININO

NOME DA ATLETA	MATRÍCULA

(NÃO GRADUANDOS)

NOME DA ATLETA	MATRÍCULA	SITUAÇÃO

Figura A.1: Lista modelo das delegações estudantis