CENTRO UNIVERSITÁRIO FEEVALE

LEANDRO FUSSIGER

NLPSEARCH
FRAMEWORK PARA INTEGRAÇÃO DE SISTEMAS DE PLN E *API'S* DE MECANISMOS DE BUSCA

NOVO HAMBURGO, dezembro 2006.

LEANDRO FUSSIGER

NLPSEARCH
FRAMEWORK PARA INTEGRAÇÃO DE SISTEMAS DE PLN E *API'S* DE MECANISMOS DE BUSCA

Centro Universitário Feevale
Instituto de Ciências Exatas e Tecnológicas
Curso de Ciência da Computação
Trabalho de Conclusão de Curso

Professor Orientador: Rodrigo Rafael Villarreal Goulart

Novo Hamburgo, dezembro de 2006.

RESUMO

O Processamento da Linguagem Natural (PLN) é a área da computação que focada no estudo da língua e de suas aplicações computacionais. Os pesquisadores de PLN desenvolvem estudos sobre fenômenos diferentes da língua, como por exemplo, a análise sintática, resolução de anáforas e no tratamento de diálogos. Os experimentos envolvem geralmente o uso de uma coleção dos textos chamada corpus. Para criar um corpus é necessário coletar os textos, com base em algum tema, e etiquetá-los de acordo com algum aspecto lingüístico, o que consome recursos humanos e computacionais. Para investigar estas deficiências este trabalho sugere o uso de uma larga coleção de textos, a Internet. Para fazê-lo, *API's* de mecanismos de busca serão utilizadas, como por exemplo, as *API's* do *Google*, *Yahoo* e *Technorati*. Usando conceitos de PLN e ferramentas e busca *API's*, este trabalho propõe a construção de um *framework* de PLN que use a Internet como o corpus.

Palavras Chave: PLN. *API's* de Mecanismos de Busca, *Frameworks*.

ABSTRACT

The Natural Language Processing (NLP) is the computation area which focus on the study of the language and its computational applications. NLP researchers develop studies on different language phenomena, as we can see on syntactic analysis, anaphors resolution and the dialogs treatment. The experiments generally involve the use of a collection of texts, called corpus. To create a corpus is necessary to collect texts, based on some subject and label it according to some liguistic aspect, which consumes human and computational resources. To work on this problem, this work suggests the use of a wide collection of texts, the Internet. To make it, Web search APIs (Application Programming Interface) will be used, as the Google, Yahoo and Technorati. Using NLP concepts and tools and search APIs, this work proposes the construction of a NLP framework which uses the web as corpus.

Key words: NLP, Web search APIs, Frameworks

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ADTF	**A**nalysys and **D**esign **T**ask **F**orce
API	**A**pplication **P**rogram **I**nterface
FAQ	**F**requently **A**sked **Q**uestion
HTML	**H**iper **T**ext **M**arkup **L**anguage
IP	**I**nternet **P**rotocol
MIME	**M**ultipurpose **I**nternet **M**ail **E**xtensions
MVC	**M**odel **V**iew **C**ontroller
ODP	**O**pen **D**irectory **P**roject
OMG	**O**bject **M**anagement **G**roup
OMT	**O**bject **M**odeling **T**echnique
OOSE	**O**bject-**O**riented **S**oftware **E**ngineering
ORB	**O**bject **R**equest **B**roker
PLN	**P**rocessamento da **L**inguagem **N**atural
RSS	**R**ich **S**ite **S**ummary/**R**eally **S**imple **S**yndication
RTF	**R**evision **T**ask **F**orce
SGML	**S**tandard **G**eneralized **M**arkup **L**anguage
SMTP	**S**imple **M**ail **T**ransfer **P**rotocol
UML	**U**nified **M**odeling **L**anguage
UTF-8	**8**-bit **U**nicode **T**ransformation **F**ormat
XML	E**X**tensible **M**arkup **L**anguage

SUMÁRIO

INTRODUÇÃO

A investigação cientifica de métodos e técnicas para o processamento da linguagem natural têm se desenvolvido muito nas últimas décadas, e especialmente nos últimos anos, com esforços focados em diferentes fenômenos da linguagem. O processamento de anáforas, sumarização automática, o tratamento de diálogos, e a tradução automática são alguns exemplos do emprego de recursos computacionais e o conhecimento de lingüistas na construção de sistemas inteligentes que consigam tratar de forma individualizada os fenômenos da linguagem (GASPERIN et al., 2003; PARDO et. al, 2003; NUNES et al., 1999; RUSSEL; NORVIG, 2004).

O estudo das primeiras técnicas de PLN são datadas da década de 50, e eram focadas especialmente na tradução automática de textos. Segundo Nunes et al. (1999) o marco inicial das pesquisas de PLN foi a distribuição de 200 cópias de um carta conhecida como *Weaver Memorandum*, escrita por Warren Weaver, exímio conhecedor de criptografia computacional. Nesta carta ele convidava a comunidade científica a desenvolver projetos sobre um novo campo, conhecido como tradução automática.

Nesse contexto de evolução das técnicas de PLN surge a necessidade de desenvolver mecanismos para integração desses esforços, aliados a uma larga base de dados como a internet. Para propor um sistema que integre estes conceitos de forma a ampliar sua utilização em diferentes contextos lingüísticos, este trabalho propõe a construção de um *framework* para o desenvolvimento de aplicações de PLN que utilizem a *Web* como corpus[1].

Existem diversos conceitos para definir um *framework* orientado a objetos, segundo Silva (2000) um *framework* orientado a objetos utiliza o paradigma de orientação a objetos para produzir uma descrição de um domínio para ser reutilizada. Sendo assim um *framework* é uma estrutura de classes inter-relacionadas, que correspondem a uma implementação

[1] Corpus – "Grande coleção de texto, como bilhões de páginas que constituem a web" (RUSSELL; NORVIG, 2004, p. 807).

incompleta para um conjunto de aplicações de um domínio, sendo que esta estrutura de classes deve ser adaptada para a geração de aplicações específicas.

Utilizar um *framework* consiste em criar aplicações a partir deste *framework*. A motivação para se usar um *framework* na construção de aplicações é a perspectiva de um aumento considerável da produtividade e da qualidade do sistema desenvolvido, em função da reutilização promovida. Para produzir uma aplicação utilizando um *framework* como base, deve-se estender sua estrutura de modo a cumprir os requisitos básicos da aplicação a que se quer desenvolver (SILVA, 2000).

Um dos obstáculos encontrados para se desenvolver aplicações através do uso de *frameworks* é justamente compreender seu uso, como funciona e interage internamente. A questão é que se um desenvolvedor de aplicações necessitar empreender um esforço considerável para aprender a utilizar um *framework*, que para desenvolver uma aplicação do inicio, a utilidade deste *framework* acaba se tornando questionável (SILVA, 2000).

Para montar um corpus é necessário à reunião de diversos textos, para isto este trabalho propõe que este corpus seja montado utilizando a base de dados contida na web. Para se ter acesso a essa base de dados alguns mecanismos de busca disponibilizam gratuitamente suas *API's*[2] de pesquisa na web para desenvolvimento de aplicações não comerciais. O presente trabalho irá utilizar as *API's* disponibilizadas pelo *Google*, *Yahoo* e *Technorati* para realizar pesquisas e montar o corpus necessário para o processamento das técnicas de PLN.

Dessa forma a principal proposta deste trabalho é a construção de um *framework* orientado a objetos para o desenvolvimento de ferramentas para PLN que utilizem a web como corpus, através da utilização de *API's* de mecanismos de busca conhecidos.

O presente trabalho esta divido em oito capítulos, organizados seguinte forma:

O primeiro capítulo introduzirá a utilização da orientação a objetos para construção de softwares, este capítulo torna-se muito interessante, pois serve como base para o desenvolvimento do *framework* proposto. O segundo capítulo abordará a utilização da *UML*

[2] API – do inglês *Application Programming Interface* (*Interface* de Programação de Aplicativos). É um conjunto de rotinas e padrões estabelecidos por um software para utilização de suas funcionalidades por programas aplicativos, isto é, programas que não querem envolver-se em detalhes da implementação do software, mas apenas usar seus serviços.

para a modelagem de sistemas de software orientados a objetos. No capítulo seguinte serão apresentados conceitos sobre a utilização de padrões de projeto, muito empregados na construção de *frameworks*. O quarto capítulo apresentará uma abordagem sobre *frameworks* orientados a objetos, onde serão apresentadas vantagens e desvantagens da sua utilização.

O quinto capítulo apresentará algumas noções dos mecanismos de busca que serão utilizados no trabalho.

No sexto capítulo serão apresentados alguns conceitos sobre PLN, assim como algumas definições a respeito das suas principais ferramentas. No sétimo capítulo serão apresentadas algumas ferramentas que se dispões a atender os conceitos abordados no capitulo sexto.

O oitavo capítulo irá explicitar a proposta principal deste trabalho, que será desenvolvida no próximo semestre.

1 ORIENTAÇÃO A OBJETOS

O desenvolvimento de um sistema de *software* não é tarefa simples de ser realizada, pois demanda muita lógica computacional, conhecimento de linguagens de programação, ferramentas de desenvolvimento e por último, mas não com menos importância, do negócio a que se destina o sistema a ser desenvolvido. O programador ou equipe responsável pelo desenvolvimento precisam ter esse conhecimento para produzir *softwares* que primem essencialmente pela qualidade. Hoje em dia para atingir essa qualidade tornou-se essencial a utilização do paradigma[3] de orientação a objetos no desenvolvimento de sistemas de software. As técnicas orientadas a objetos prometem aumentar de forma significativa tanto a qualidade quanto a produtividade do desenvolvimento de software (YOURDON; ARGILA, 1999). Segundo Bezerra (2002) um sistema de *software* orientado a objetos consiste de objetos em colaboração com o objetivo de realizar as funcionalidades desse sistema. Cada objeto é responsável por tarefas especificas. É através da cooperação entre objetos que a computação do sistema se desenvolve. Neste capítulo serão apresentadas algumas noções sobre o paradigma de orientação a objetos.

1.1 Objetos

O paradigma da orientação a objetos visualiza um sistema de software como uma coleção de agentes interconectados chamados objetos. Cada objeto é responsável por realizar tarefas especificas. É através da interação entre objetos que uma tarefa computacional é realizada (BEZERRA, 2002).

---

[3] "Paradigma. [Do grego parádeigma, pelo latim tard. paradgma.] S. m. Termo com o qual Thomas Kuhn designou as realizações cientificas (p. ex., a dinâmica de Newton ou a química de Lavoisier) que geram modelos que, por período mais ou menos longo e de modo mis ou menos explicito, orientam o desenvolvimento posterior das pesquisas exclusivamente na busca da solução para os problemas por elas suscitados" (BEZERRA, 2002, p. 4).
Paradigma é Definido como um "conjunto de teorias, padrões e métodos que juntos representam um modo de organizar conhecimento" (LEE; TEPFENHART, 2001, p. 24).

"Objeto é uma entidade independente, assíncrona e concorrente que "sabe coisas" (armazena dados), 'realiza trabalho' (encapsula serviços) e 'colabora com outros objetos' (por meio de troca de mensagens) para executar as funções finais do sistema" (YOURDON; ARGILA, 1999, p. 7).

Segundo Harmon e Watson (1998) um objeto é um complexo tipo de dado que contém outros tipos de dados, chamados de atributos ou variáveis, e módulos de código, chamados operações ou métodos. Os atributos e valores associados estão escondidos dentro do objeto. Qualquer outro objeto que quiser obter ou alterar um valor associado com um primeiro objeto deve fazer isso enviando uma mensagem para ele.

Pree (1995) define um objeto como sendo uma instância de tipo de dados abstratos. Desta maneira um objeto é uma entidade que contém atributos (dados representando um estado do objeto) e que disponibiliza certas operações que são definidas para um objeto em particular. Os atributos são chamados de variáveis de instância e as operações são chamadas de métodos.

Uma definição mais simples de objeto é apresentada por Rumbaugh et al. (1994) que diz que um objeto é simplesmente alguma coisa que faz sentido no contexto de uma aplicação e ainda define um objeto como sendo um conceito, uma abstração, algo com limites nítidos e significado em relação ao problema em causa. Objetos servem a dois objetivos: eles facilitam a compreensão do mundo real e oferecem uma base real para a implementação em computador.

Para Coleman (1996) um objeto representa simplesmente um elemento que pode ser identificado de maneira única. Em um nível apropriado de abstração praticamente tudo pode ser considerado como sendo um objeto. Dessa forma, elementos específicos como pessoas, empresas, acessórios, automóveis ou eventos podem ser consideras como objetos.

Objetos possuem identidades e são distinguíveis. O termo identidade significa que os objetos se distinguem um dos outros pela sua própria existência, ou seja, mesmo possuindo os mesmos atributos são considerados distintos, pois cada qual representa uma instância diferente de uma mesma classe de objetos (RUMBAUGH et al., 1994).

Quando um objeto quer mudar ou solicitar alguma informação de outro objeto ele envia mensagens, estas mensagens servem como gatilhos para disparar métodos para o outro objeto, são exatamente estes métodos que irão realizar a tarefa solicitada pelo primeiro objeto. Os métodos que podem ser chamados por outros objetos são conhecidos como métodos públicos. Objetos ainda podem conter métodos privados (métodos visíveis somente para o próprio objeto), não sendo possível acessá-los através do envio de mensagens (LEE; TEPFENHART, 2001).

1.2 Mensagens

Objetos não executam suas operações (métodos) aleatoriamente. Para que um método seja executado deve haver um estímulo enviado a este objeto. Um objeto é visto como uma abstração do mundo real, dessa forma faz sentido dizer que um objeto pode responder a estímulos a ele enviados. Independentemente da origem do estimulo, quando ele ocorre, diz-se que o objeto em questão esta recebendo uma mensagem. Esta mensagem é recebida pelo objeto e este objeto deve realizar alguma operação (BEZERRA, 2002). Quando uma mensagem é enviada a um objeto, o método específico que será executado depende de ambos, ou seja, do objeto que enviou a mensagem e do objeto receptor. A associação em tempo de execução de uma solicitação de um objeto e a uma das suas operações é conhecida como ligação dinâmica (GAMMA et al., 2000). Para simplificar a compreensão desse processo de troca de mensagens entre objetos pode-se considerar um esquema cliente-servidor, onde um objeto cliente solicita alguma informação ao objeto servidor, esse por sua vez processa a solicitação e devolve o resultado para o objeto cliente.

A figura 1.1 representa a troca de mensagens entre objetos.

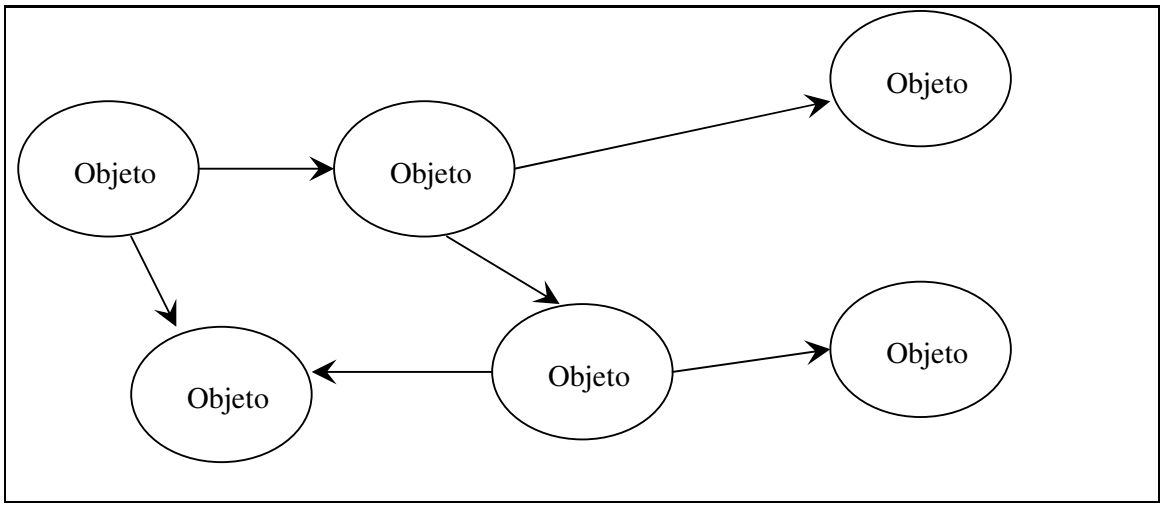

Figura 1.1 - Objetos interagem através do envio de mensagens Classes

Uma classe de objetos descreve um grupo de objetos com propriedades semelhantes (atributos), o mesmo comportamento (métodos), os mesmos relacionamentos com outros objetos e a mesma semântica (RUMBAUGH et al., 1994).

Harmon e Watson (1998) definem uma classe como sendo um modelo que possui métodos e atributos e tipos de informações, mas não carregam valores atuais. Os objetos gerados pelas classes carregam valores atuais. Algumas classes são muito abstratas e são simplesmente utilizadas para prover uma estrutura para outras classes, entretanto grande parte das classes são modelos usados para gerar os objetos que serão responsáveis pela execução do sistema.

As classes são blocos de construção da maior parte das linguagens orientadas a objetos (COLEMAN et al., 1996).

A principal finalidade das classes abstratas é definir uma *interface* para suas subclasses. Um classe abstrata postergará parte de, ou toda, sua implementação para os métodos definidos em suas subclasses, sendo assim uma classe abstrata não pode ser instanciada. Os métodos que uma classe abstrata declara, mas não implementa, são chamados de métodos abstratos. As classes que não são abstratas são conhecidas como classes concretas. Estas classes ao contrário das abstratas podem ser instanciadas (GAMMA et al., 2000).

Novas classes podem ser instanciadas a partir de classes existentes através de um dos princípios mais importantes da orientação a objetos, conhecido como herança de classes. As

subclasses instanciadas a partir de uma classe mãe herdam todos os atributos e métodos que a classe mãe define. Os objetos que são instâncias das subclasses irão conter todos os dados definidos pela subclasse e suas classes mãe, e eles serão capazes de executar todas as operações definidas por esta subclasse e seus "ancestrais" (GAMMA et al., 2000). A notação gráfica de uma classe é representada como sendo uma "caixa" com no máximo três compartimentos exibidos, dependendo do grau de abstração. No primeiro é exibido o nome da classe, no segundo aparecem os atributos e no terceiro as operações ou métodos, conforme mostra a figura 1.2 (BEZERRA, 2002).

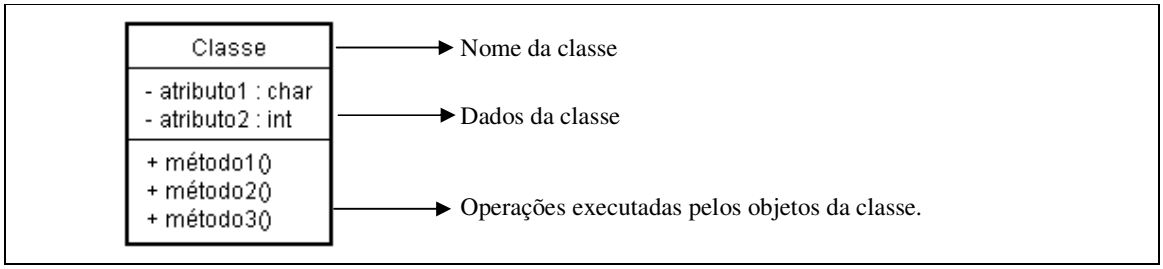

Figura 1.2 - Representação Gráfica de Uma Classe

1.3 Abstração

A abstração consiste em concentra-se nos aspectos essenciais, próprios, de uma entidade e ignorar as propriedades acidentais. Abstrações permitem gerenciar a complexidade e concentrar a atenção nas características essenciais dos objetos. O uso da abstração preserva a liberdade de se tomar decisões evitando, tanto quanto possível, comprometimentos prematuros com detalhes. Uma abstração é dependente da perspectiva: o que pode ser considerado importante em um determinado contexto pode não ter tal importância em outro (RUMBAUGH et al., 1994; BEZERRA, 2002). Levando em consideração essa definição, pode-se afirmar que o Princípio da Abstração é aplicado em todos os demais Princípios da Orientação a Objetos, como mostra a figura 1.3.

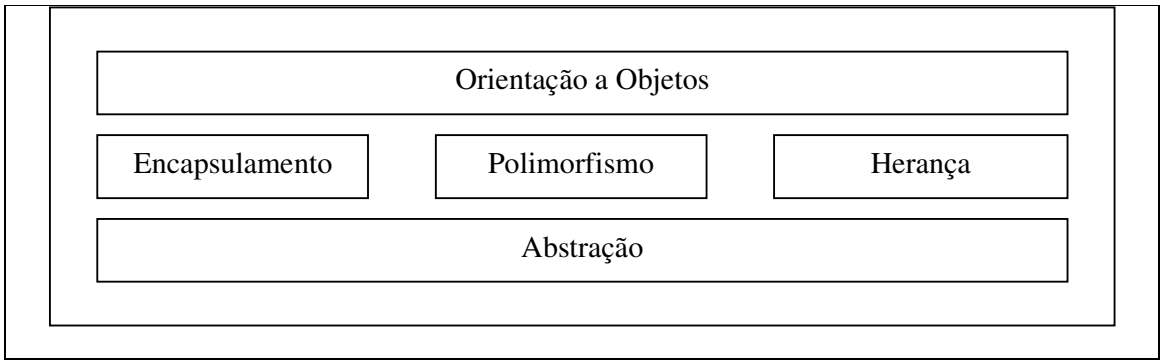

Figura 1.3 - Princípios da orientação a objetos como aplicações do Princípio da Abstração

1.4 Encapsulamento

Também chamado de "ocultar informações", este conceito estabelece que cada objeto contenha todos os detalhes de implementação necessários sobre como ele funciona e oculta todos os detalhes sobre como ele executa suas operações ou gerencia sua estrutura interna (REED, 2000).

Segundo Bezerra (2002) o mecanismo de encapsulamento é uma forma de restringir o acesso ao comportamento interno de um objeto. Um objeto que precise trocar informações com outro objeto para desempenhar alguma tarefa simplesmente envia uma mensagem para este último. O método requisitado não é conhecido dos objetos que requisitam alguma informação.

O encapsulamento impede que um programa se torne tão interdependente que uma simples modificação possa causar grandes efeitos de propagação, dessa forma a implementação de um objeto pode ser modificada sem que essa modificação afete as aplicações que o utilizem (RUMBAUGH et al., 1994).

Para que um objeto requisitante saiba quais tarefas que um outro objeto sabe realizar ou quais informações ele conhece, este precisa ter conhecimento da interface deste outro objeto. A interface de um objeto define os serviços que ele sabe realizar e conseqüentemente as mensagens que ele recebe quando é solicitado (BEZERRA, 2002).

Sendo assim, o encapsulamento também pode ser visto e "comparado" com o conceito de caixa-preta – Um bloco de informações desconhecidas. Simplesmente, utiliza-se este bloco, fornecendo entradas e recebendo saídas, sem saber como essas entradas são

processadas para fornecer as saídas. Não interessa o que tem dentro do bloco. O que interessa é que este bloco fornece as saídas desejadas, de acordo com as entradas fornecidas.

A figura 1.4 ilustra um objeto. Os dados no centro podem ser usados apenas com as operações constantes do anel exterior.

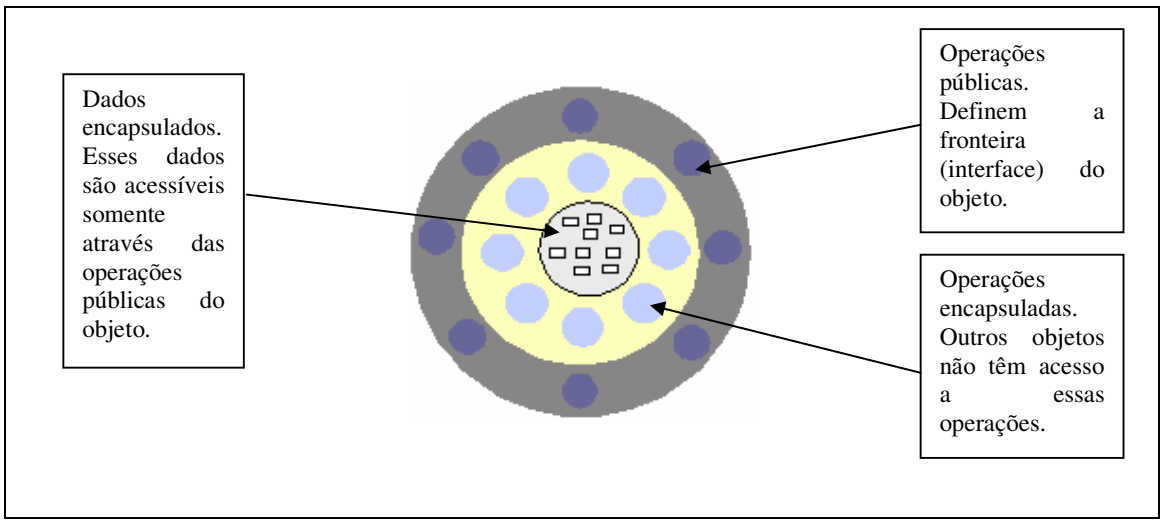

Figura 1.4 - Objeto – Conceito de Encapsulamento

1.5 Herança

A Herança pode ser considerada como uma abstração poderosa para o compartilhamento de similitudes entre classes, ao mesmo tempo em que suas diferenças são preservadas (RUMBAUGH et al., 1994). No conceito de herança as classes semelhantes são agrupadas em hierarquias. Cada nível hierárquico pode ser visto como um nível de abstração, sendo assim, cada classe em um nível de hierarquia herda as características das classes que estão em um nível mais alto (BEZERRA, 2002). A utilização da herança possibilita aos desenvolvedores alavancar o poder da hierarquia de classes criando novos objetos sem ter que desenvolvê-los novamente do zero (REED, 2000).

As classes que recebem a herança são denominadas subclasses, enquanto as classes que geram as subclasses são denominadas superclasses. Os atributos e operações comuns a um agrupamento de subclasses são incluídos na superclasse e são compartilhados por todas as subclasses, por isso diz-se que cada subclasse herda todas as características da sua superclasse (RUMBAUGH et al., 1994).

> [...] Uma instância de uma subclasse é simultaneamente uma instância de todas as suas classes ancestrais. O estado de uma instância inclui um valor para cada atributo de cada classe ancestral. Qualquer operação em qualquer classe ancestral pode ser aplicada a uma instância. Cada subclasse não só herda todas as características de seus ancestrais como também acrescenta seus próprios atributos específicos e suas próprias operações [...] (RUMBAUGH et al., 1994, p. 54).

Existem dois tipos de heranças, a herança de classe (implementação) e a herança de interface (subtipificação). Segundo Gamma et al. (2000) é importante compreender a diferença entre estes dois tipos. A herança de classe define a implementação de um objeto em termos da implementação de outro, ou seja, pode ser considerado como um mecanismo para compartilhar código e representação. A herança de interface descreve quando um objeto pode substituir outro.

A figura 1.5 mostra um exemplo de herança para figuras gráficas, onde as classes em um nível hierárquico inferior herdam da classe que esta em um nível hierárquico superior. Analisando o diagrama abaixo é possível verificar como funciona a herança de operações: Mover, selecionar, girar e exibir são operações herdadas por todas as subclasses. Medir aplica-se a figuras uni e bidimensionais. Preencher aplica-se somente a figuras bidimensionais (RUMBAUGH, et al., 1994).

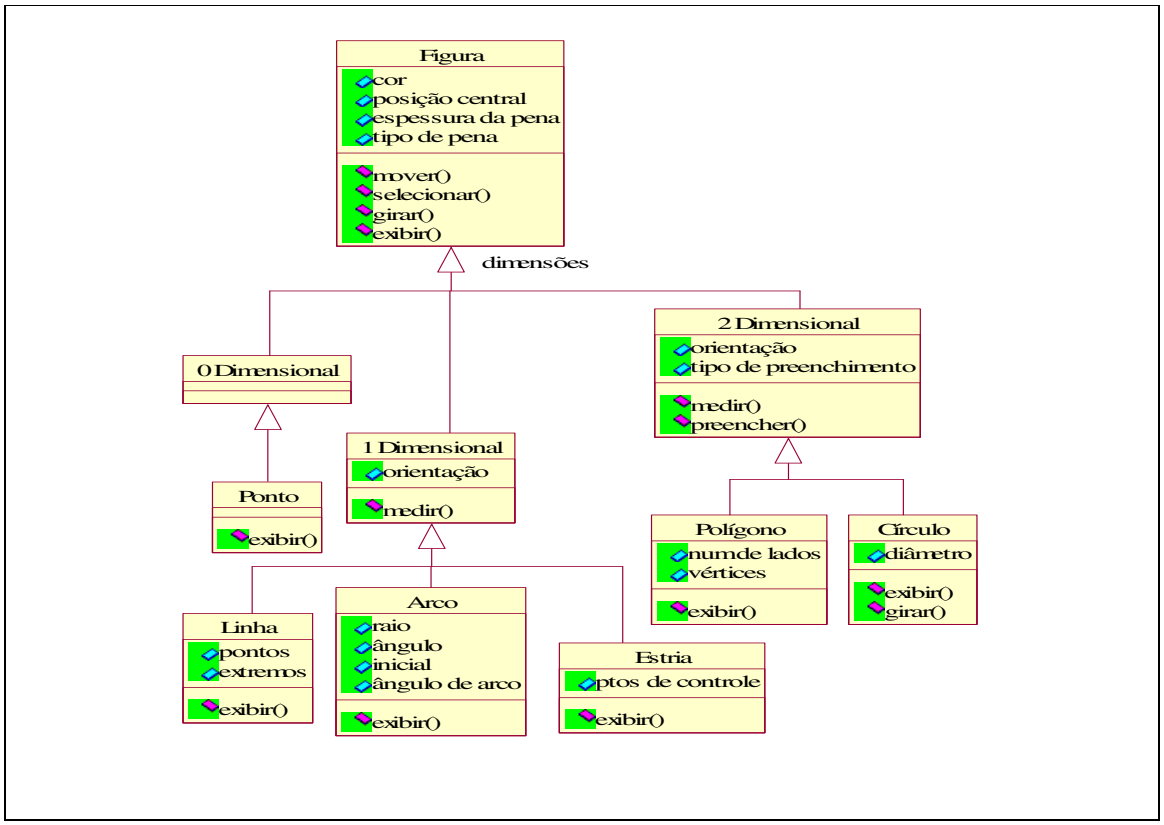

Figura 1.5 - Herança para Figuras Gráficas

1.6 Polimorfismo

A palavra polimorfismo é originada do grego, onde "poli" significa muitas e "morfismo" significa formas, então polimorfismo significa muitas formas. Em se tratando de orientação a objetos polimorfismo é conhecido como um importante conceito, pois através dele uma mesma operação/método pode atuar de modos diversos em classes distintas (RUMBAUGH et al., 1994).

Segundo Bezerra (2002) o polimorfismo indica a capacidade de abstrair várias implementações diferentes em uma única interface.

Para Harmon e Watson (1998) polimorfismo significa que operações iguais poderão se comportar de forma diferente quando aplicadas a objetos de diferentes classes. Isso também quer dizer que diferentes operações associadas com diferentes classes podem interpretar uma mesma mensagem de diversas formas.

Reed (2000) também reforça a idéia de polimorfismo como sendo um importante conceito na orientação a objetos, pois segundo ele é importante poder enviar a mesma mensagem para diferentes objetos e fazê-los responder de forma correta.

A figura 1.6 apresenta um exemplo de polimorfismo aplicado a uma classe chamada de "SetorMatriculas". Neste exemplo um aluno, representado pela classe "Aluno", pode solicitar a realização de sua matrícula em um determinado curso ou em determinadas disciplinas para o Setor de Matriculas. Contudo, a forma como a matricula será realizada vai depender do nível de ensino do curso que o aluno quer se matricular: extensão, especialização, graduação, etc.

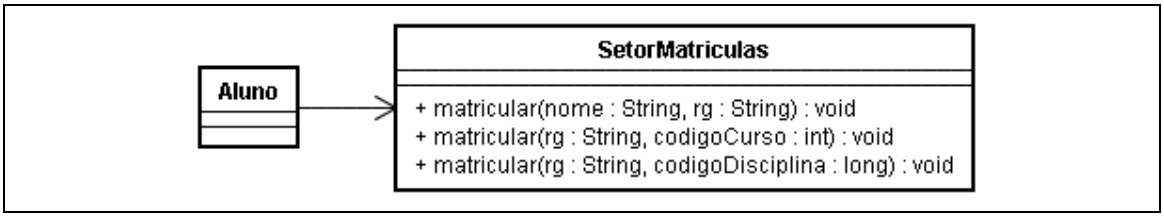

Figura 1.6 - Exemplo de Polimorfismo

Para o aluno realizar a matricula em um curso de extensão será necessário o seu nome e seu RG. Para uma matricula num curso de especialização é necessário o RG e a identificação do curso. E por fim, para realizar a matricula num curso de graduação será necessário o RG e o código da disciplina que o aluno deseja cursar. Somente através do polimorfismo é possível realizar a mesma ação, no caso do exemplo, realizar a ação de *Matricular* de várias formas distintas, dando maior flexibilidade ao sistema.

Saindo um pouco do conceito teórico e entrando em questões de programação, percebe-se que a implementação do polimorfismo é a construção de métodos de classe com o mesmo nome (mesma ação), porém com assinaturas diferentes (mesma ação de forma variada). O conjunto de argumentos e tipos de retorno é chamado de assinatura (REED, 2000). No exemplo citado se vê três métodos com nomes iguais, mas com assinaturas distintas.

O segundo capítulo irá trazer algumas noções sobre a utilização da Linguagem Unificada de Modelagem (*UML*), para modelagem de sistemas orientados a objetos.

2 *UML*

Segundo Booch et al. (2000) a *UML* é uma linguagem padrão para a elaboração da estrutura de projetos de software e pode ser empregada para a visualização, especificação, construção e a documentação de artefatos que façam uso de sistemas complexos de software. A *UML* é considerada uma linguagem de modelagem visual, ou seja, é um conjunto de notações e semântica correspondente com o intuito de representar visualmente uma ou mais perspectivas de um sistema (BEZERRA, 2002).

> A *UML* é adequada para a modelagem de sistemas, cuja abrangência poderá incluir sistemas de informação corporativos a serem distribuídos a aplicações baseadas em web e até sistemas complexos embutidos de tempo real. É uma linguagem muito expressiva, abrangendo todas as visões necessárias ao desenvolvimento e implantação desses sistemas [...] (BOOCH et al., 2000, p. 13).

Este capítulo tem por objetivo trazer algumas noções sobre a *UML*, pois esta linguagem será de grande valia no decorrer deste trabalho.

2.1 Sobre a *UML*

Com o passar dos anos a modelagem de sistemas passou a ser um ponto fundamental na estruturação de projetos bem sucedidos. Por este motivo iniciou-se uma busca por padrões de desenvolvimento de software que agregassem qualidade ao produto desenvolvido. Durante o fim da década de 80 e início da década de 90 surgiram diversas propostas de técnicas para modelagem de sistemas orientados a objetos. No período de 1989 a 1994 a quantidades de métodos orientados a objetos aumentou de um pouco mais de 10 para mais de 50 (BOOCH et al., 2000). No entanto a comunidade de software necessitava de um padrão que fosse aceito tanto pelas indústrias como pelo meio acadêmico.

Sendo assim, em 1996 surge a Linguagem de Modelagem Unificada para justamente unificar e padronizar as técnicas de modelagem existentes. Para a construção da *UML* foram estudadas várias notações, porém a base fundamental foi retirada dos estudos de três pesquisadores, Grady Booch, James Rumbaugh e Ivar Jacobson, freqüentemente chamados de

"os três amigos". No processo de definição da *UML*, procurou-se aproveitar o melhor das características das notações preexistentes, principalmente das técnicas propostas anteriormente pelos três amigos (essas técnicas eram conhecidas pelos nomes *Booch Method*[4], *OMT*[5] *e OOSE*[6]) (BEZERRA, 2002).

Em janeiro de 1997 foi oferecida a *OMG*[7] (*Object Management Group*) a versão 1.0 da *UML* em resposta a solicitação da própria *OMG* para uma linguagem padrão de modelagem. Após a criação de um grupo de tarefas em semântica liderado por Cris Kobryn da *MCI Systemhouse* e administrado por Ed Eykholt da *Rational* com o propósito de formalizar a especificação da *UML* e de integrar a linguagem a *outros esforços de padronizações a versão 1.1 da UML foi oferecida para padronização ao OMG* em julho de 1997. Em setembro do mesmo ano essa versão foi aceita pela *ADTF* (*Analysys and Design Task Force*) e pelo *Archtecture Board* do *OMG*, e foi submetido ao voto de todos os membros do *OMG*. A *UML* versão 1.1 foi adotada pelo *OMG* em 14 de novembro de 1997 como linguagem padrão (BOOCH et al., 2000; FOWLER; SCOTT, 2000; REED, 2000; BEZERRA, 2002). A manutenção da *UML* passou a ser responsabilidade da *RTF* (*Revision Task Force*) que pertence ao *OMG* e dirigida por Cris Kobryn. Dentre os objetivos desta força tarefa estão inseridas as revisões nas especificações da *UML* para propor novas alterações de forma que não provoquem mudanças no escopo original da linguagem. Em julho de 1998 foi apresentada a versão 1.2 da *UML* e no final do ano a versão 1.3. Em maio de 2001 foi lançada a versão 1.4, em agosto do mesmo ano foi submetido um relatório provisório sobre a versão 1.5, publicada em março de 2003. Entre março e junho de 2003 as primeiras especificações da *UML* 2.0 foram adotadas pelo *OMG* como padrão (MELO, 2004). A documentação oficial sobre a *UML* pode ser encontrada no site da OMG, www.omg.org.

---

[4] *Booch Method* – Expressivo especialmente durante as fases de projeto e construção de sistemas (BOOCH, 2000).
[5] *OMT* (*Object Modeling Technique*) – Mais útil para a analise e sistemas de informações com uso intensivo de dados (BOOCH, 2000).
[6] *OOSE* (*Object-Oriented Software Engineering*) – Fornecia excelente suporte para os casos como uma maneira de controlar a captura de requisitos, a análise e o projeto em alto nível (BOOCH, 2000).
[7] "OMG (*Object Management Group*) – Consórcio internacional de empresas que define e ratifica os padrões na área de orientação a objetos. (www.omg.org)" (BEZERRA, 2002, p. 13).

2.2 Diagramas da *UML*

Um processo de desenvolvimento que utilize a *UML* como linguagem de suporte a modelagem envolve a criação de diversos documentos. Estes documentos podem ser textuais ou gráficos. Na terminologia da *UML*, estes documentos são conhecidos como artefatos de software, e são estes artefatos que compõe as diversas visões do sistema (BEZERRA, 2002). Os artefatos gráficos produzidos através da utilização da *UML* são chamados de diagramas.

Segundo Boch et al. (2000) um diagrama é uma representação gráfica de um conjunto de elementos, comumente representada como um gráfico conectado de vértices (itens) e arcos (relacionamentos). São desenhados para permitir a visualização de um sistema sob diferentes perspectivas, sendo assim um diagrama é considerado uma projeção em um sistema. Até a versão 1.4 a *UML* incluía nove diagramas em sua notação. A partir da versão 2.0 este número aumentou consideravelmente, sendo incluídos mais cinco diagramas, e alterado o nome de alguns existentes (MELO, 2004). A tabela 2.1 demonstra quais diagramas da *UML* 2.0 são novos e quais correspondem a diagramas da versão 1.4.

Diagramas da *UML* 1.4	Diagramas da *UML* 2.0
...	Pacotes
...	Estrutura Composta
...	Visão Geral
...	Temporal
Atividades	Atividades
Casos de Uso	Casos de Uso
Classes	Classes
Colaboração	Comunicação
Componentes	Componentes
Gráfico de Estado	Máquina de Estados
Implantação	Implantação
Objetos	Objetos
Seqüência	Seqüência

Tabela 2.1 - Comparativo dos Diagramas da *UML* 1.4 e 2.0

Os diagramas podem ser divididos em duas categorias, diagramas estáticos ou estruturais e diagramas dinâmicos ou comportamentais. A função dos diagramas estruturais é a de mostrar as características de um sistema que não mudam com o tempo, ou seja, servem para visualizar, especificar, construir e documentar os aspectos estáticos de um sistema (BOOCH, 2000; MELO, 2004). A função dos diagramas dinâmicos é mostrar as interações ativas que um sistema suporta, desta forma eles detalham a interação entre os diagramas

estruturais (REED, 2000). A tabela 2.2 mostra os diagramas existentes na *UML* divididos nas duas categorias anteriormente citadas.

Diagramas Estruturais	Diagrama de Classes Diagrama de Objetos Diagrama de Componentes Diagrama de Pacotes Diagrama de Implantação Diagrama de Estrutura Composta
Diagramas Dinâmicos	Diagrama de Casos de Uso Diagrama de Interação: - Diagrama de Visão Geral - Diagrama de Seqüências - Diagrama Temporal - Diagrama de Comunicação Diagrama de Atividades Diagrama de Máquina de Estados

Tabela 2.2 – Diagramas divididos por categorias

A seguir será apresentado um pequeno resumo sobre cada um dos diagramas que compõe a notação atual da *UML.*

2.2.1 Diagrama de Casos de Uso

Quando utilizamos a *UML* para modelar um sistema, um dos primeiros processos a ser realizado diz respeito à modelagem de casos de uso. Um diagrama de caso de uso representa a interação entre usuário e sistema, demonstrando alguma funcionalidade relativa a eles (QUATRINI, 2001). Os diagramas de casos de uso são importantes para visualizar, especificar e documentar o comportamento de um elemento dentro do sistema (BOOCH et al., 2000).

Internamente, um caso de uso é uma seqüência de ações que permeiam a execução completa de um comportamento esperado para o sistema. Este é o ponto de partida para todo o processo, por isso é necessário ter muito cuidado e não incluir detalhes demais. Um Diagrama de Caso de Uso deve ser simples, porém não pode ser incompleto (BOOCH et al., 2000; BEZERRA, 2002; HARMON; WATSON, 1998; MELO, 2004). Desta forma um caso de uso não deve se preocupar com o "como" das funções. Este diagrama não deve prescrever seqüências de execução, mais sim as ações que são executadas pelos Atores.

Um ator está relacionado a um papel. No contexto do sistema, papel diz respeito à visão dada ao sistema (FOWLER, 2000; BEZERRA, 2002). Por exemplo, em um projeto de desenvolvimento de sistemas, um membro da equipe de desenvolvimento pode exercer vários papéis distintos, como projetista de testes, projetistas de banco de dados, revisor de código, etc. Cada papel destaca funções diferentes específicas a ele. Isso deixa claro que a figura do ator não está relacionada de forma direta a uma pessoa. Segundo Fowler (2000) um ator não precisa ser necessariamente humano. Um ator pode ser também um sistema externo que precisa de informações geradas em um sistema corrente.

A notação utilizada para ilustrar os atores de um caso de uso é a figura de um boneco, com o nome do ator definido abaixo dessa figura (a partir da versão 2.0 da *UML* é obrigatório informar o nome do ator). Cada caso de uso é representado por uma elipse, o nome do caso de uso é posicionado abaixo ou dentro da elipse. Um relacionamento de comunicação é representado por um segmento de reta ligando ator e caso de uso (BEZERRA, 2002; MELO, 2004). A figura 2.1 ilustra um Diagrama de Casos de Uso.

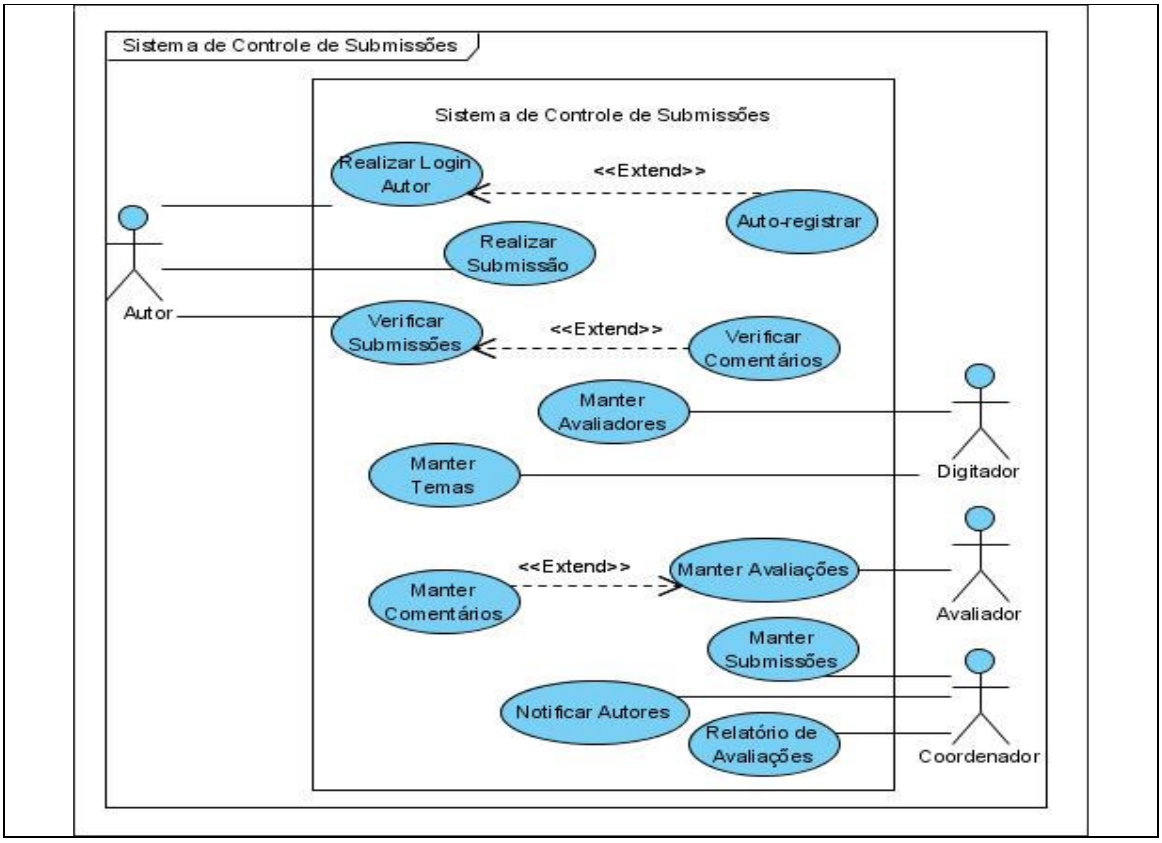

Figura 2.1 – Diagrama de Casos de Uso

2.2.2 Diagrama de Classes

Um diagrama de classes mostra um conjunto de classes, interfaces e colaborações e seus relacionamentos. Os diagramas de classes são os diagramas mais encontrados em sistemas de modelagem orientados a objetos (BOOCH et al., 2000). Segundo Bezerra (2002), o diagrama de classes é utilizado desde o nível de análise até o nível de especificação do projeto. De todos os diagramas da *UML*, esse é o mais rico em termos de notação.

O diagrama de classes demonstra a estrutura estática do software a ser construído, isto é, as classes, os relacionamentos (associações) entre suas instâncias (objetos), restrições e hierarquias. O diagrama é considerado estático, pois a estrutura descrita é sempre válida em qualquer ponto do ciclo da vida do sistema modelado (FALBO, 2001). A figura 2.2 mostra um exemplo de diagrama de classes.

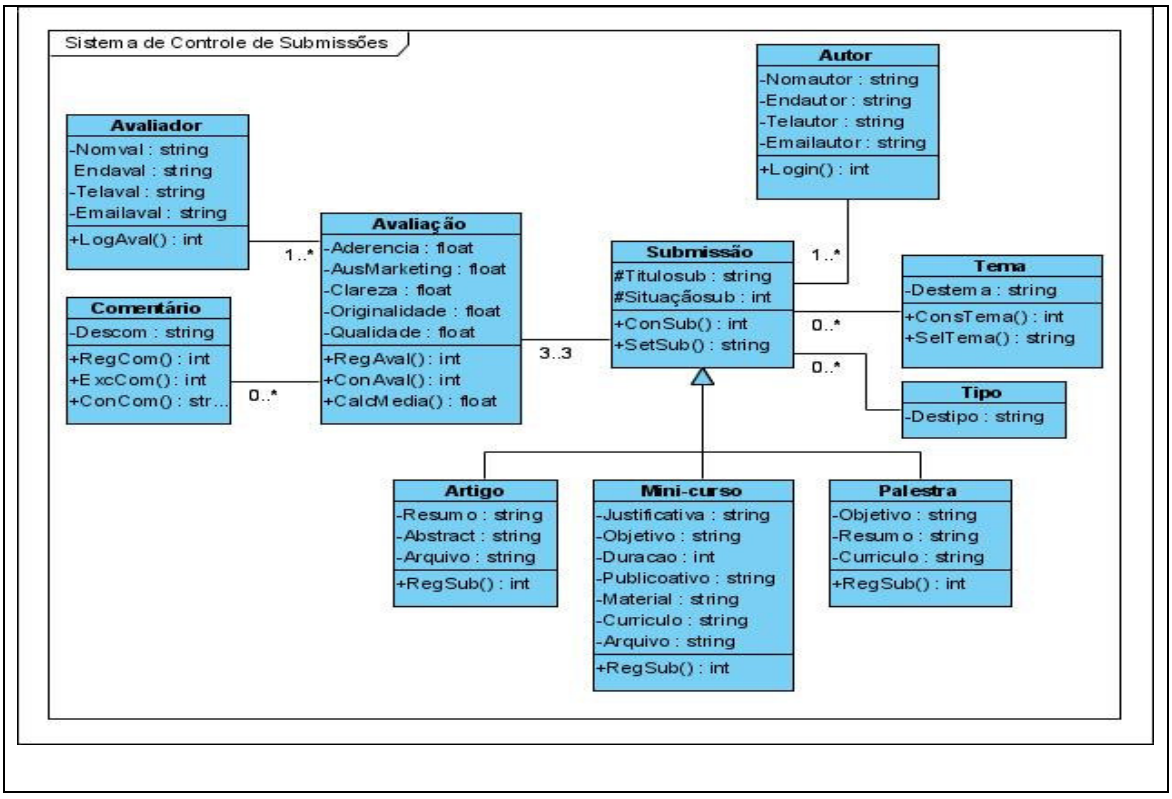

Figura 2.2 – Diagrama de Classes

2.2.3 Diagrama de Seqüência

Segundo Booch et al. (2000) um diagrama de seqüência é um diagrama de interação que dá ênfase à ordenação temporal de mensagens. Um diagrama de seqüência mostra um conjunto de objetos e as mensagens enviadas e recebidas por estes objetos.

Os diagramas de seqüência são compostos por dois eixos: no eixo vertical é exibido o tempo e no eixo horizontal os objetos envolvidos na seqüência de certa atividade, como, por exemplo, na realização de um Cenário de Caso de Uso. Cada objeto é representado por um retângulo e uma linha vertical pontilhada chamada de linha de vida do objeto, indicando a execução do objeto durante a seqüência. A comunicação entre os objetos é representada com uma linha com setas horizontais simbolizando as mensagens (BEZERRA, 2002). A figura 2.3 mostra um exemplo de digrama de seqüência.

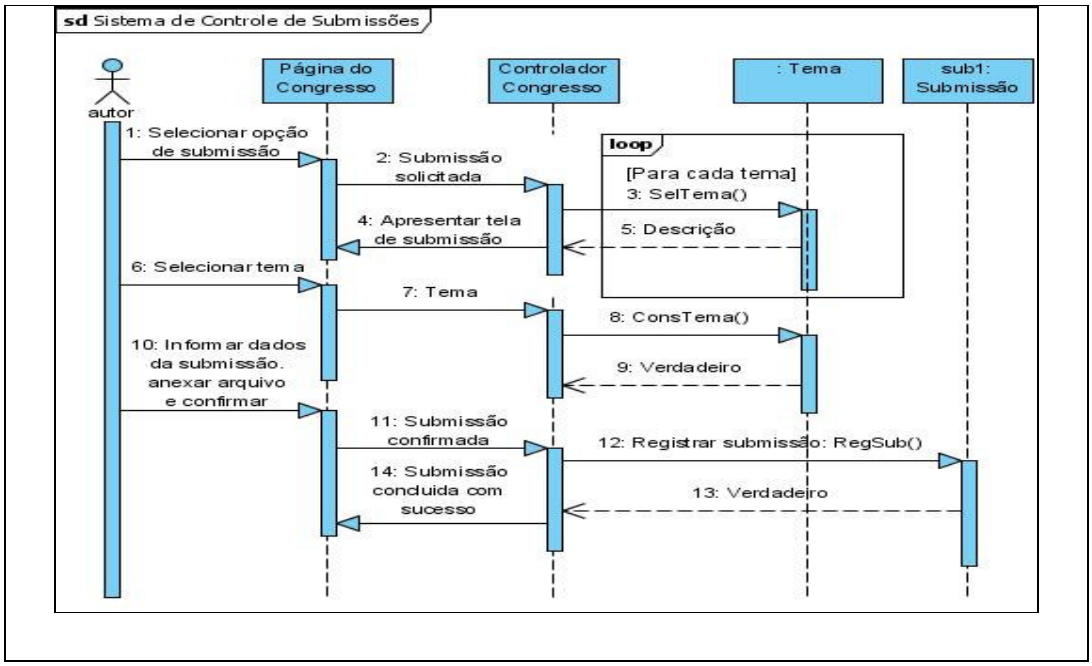

Figura 2.3 – Diagrama de Seqüência

2.2.4 Diagrama de Objetos

O diagrama de objetos mostra um conjunto de objetos e seus relacionamentos. Os diagramas de objetos direcionam a visão estática do projeto de um sistema ou a visão estática do processo de um sistema, tal qual os diagramas de classes, mas considerando casos reais ou protótipos (BOOCH et al., 2000). Sendo assim um diagrama de objetos consiste numa

instância do diagrama de classes, no qual para cada classe existe um objeto (sua instância) em um determinado ponto do tempo (MELO, 2004). A figura 2.4 mostra um exemplo de diagrama de objetos.

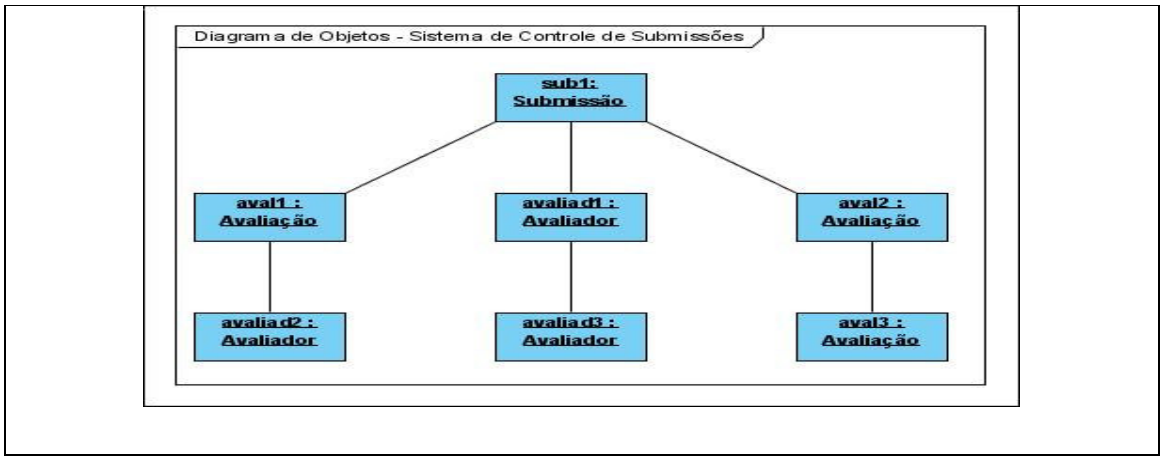

Figura 2.4 – Diagrama de Objetos

2.2.5 Diagrama de Atividades

Um diagrama de atividades mostra o fluxo de uma atividade para outra em um sistema. Eles mostram o controle de fluxo de atividade para atividade no sistema, quais atividades estão sendo feitas em paralelo e quais caminhos alternar através do fluxo. Estes diagramas são importantes principalmente para se fazer a modelagem da função de um sistema. Os diagramas de atividades dão ênfase ao fluxo de controle entre objetos (BOOCH et al., 2000; QUATRANI, 2001). A figura 2.5 mostra um exemplo de diagrama de atividades.

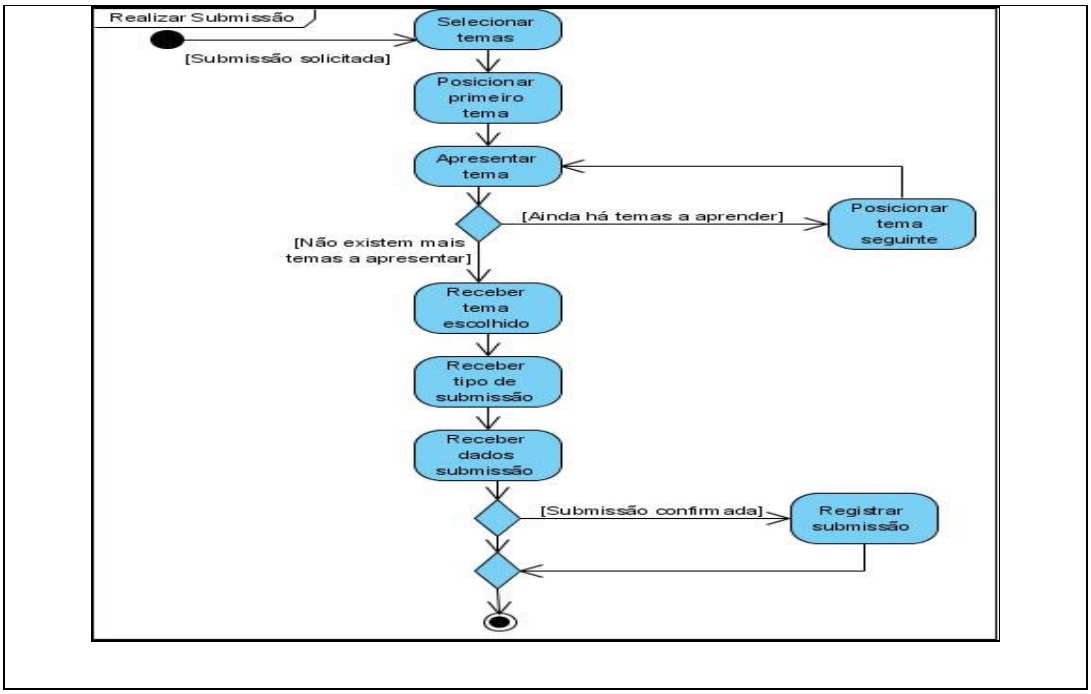

Figura 2.5 – Diagrama de Atividades

2.2.6 Diagrama de Componentes

O diagrama de componentes identifica os componentes que fazem parte de um sistema, um subsistema ou mesmo os componentes ou classes internas de um componente individual. Um componente é identificado como sendo uma unidade modular com interfaces bem definidas e substituíveis dentro de seu ambiente, sendo assim pode representar tanto um componente lógico (um componente de negócio ou de processo) ou um componente físico, como arquivos contendo código fonte, arquivos de ajuda, bibliotecas, arquivos executáveis, etc. Um diagrama de componentes pode ser utilizado como uma forma de documentar como estão sendo estruturados os arquivos físicos de um sistema, permitindo uma melhor compreensão do mesmo, além de facilitar a reutilização de código (GUEDES, 2005; MELO, 2004).

Na versão 2.0 da *UML* a abordagem do diagrama de componentes foi alterada, a versão anterior tratava o diagrama de componentes como sendo um diagrama de classificadores e os artefatos que eles implementavam, este conceito esta relacionado agora a definição de artefatos. A notação gráfica do diagrama também foi alterada, sendo que na versão anterior um componente era representado por um retângulo com dois retângulos menores sobressaindo-se á sua esquerda. Na versão 2.0 este símbolo foi substituído por um

retângulo contendo internamente o antigo símbolo (GUEDES, 2005; MELO, 2004). A figura 2.6 representa um diagrama de componentes.

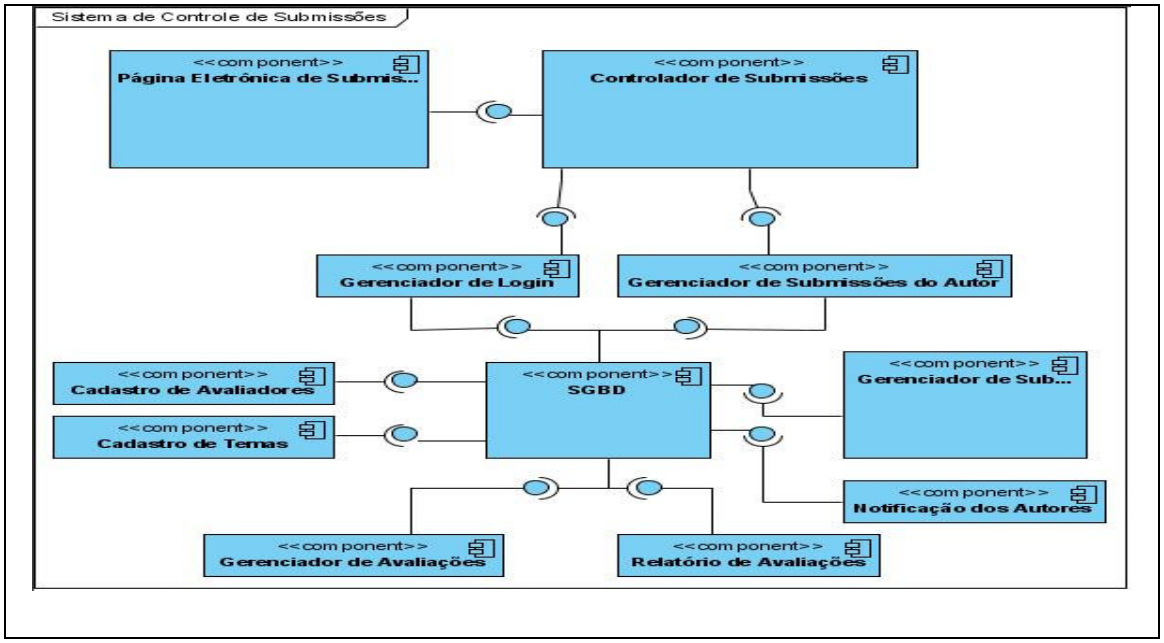

Figura 2.6 – Diagrama de Componentes

2.2.7 Diagrama de Implantação

Um diagrama de implantação mostra a configuração dos nós de processamento em tempo de execução e os componentes neles existentes. Os elementos de um diagrama de implantação são os nós e as conexões. Um nó é uma unidade física que representa um recurso computacional e normalmente possui uma memória e alguma capacidade de armazenamento. Os nós são ligados uns aos outros através de conexões. As conexões mostram mecanismos de comunicação entre os nós: meios físicos ou protocolos de comunicação (BOOCH et al., 2000; BEZERRA, 2002; FOWLER; SCOTT, 2000). A figura 2.7 mostra um exemplo de diagrama de implantação.

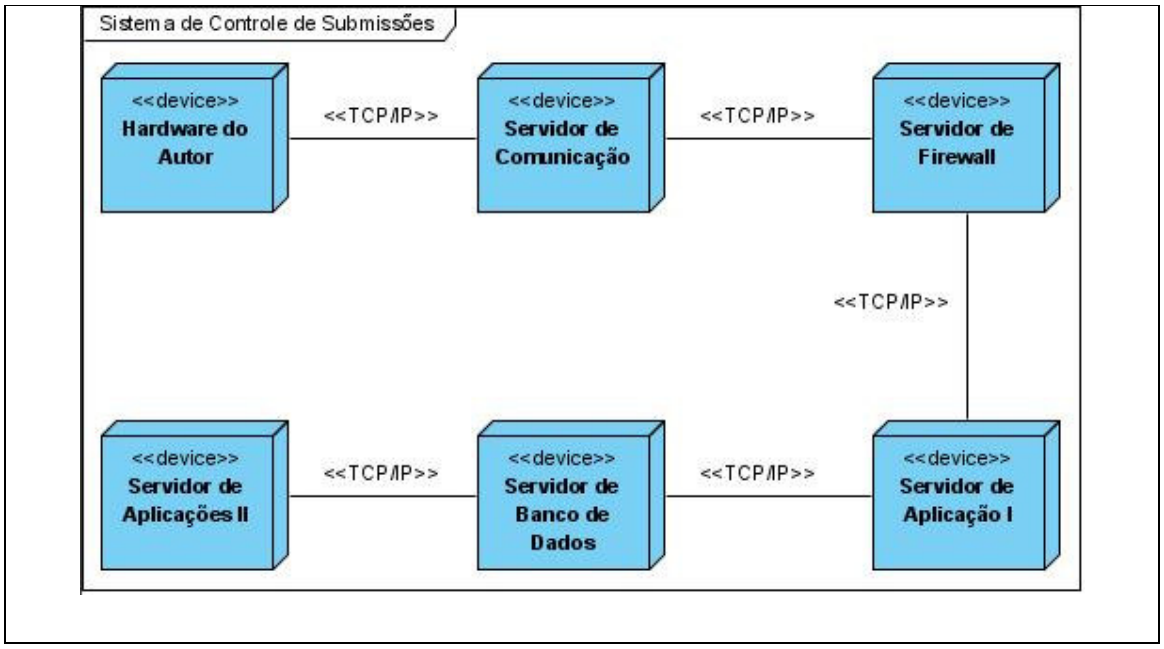

Figura 2.7 – Diagrama de Implantação

2.2.8 Diagrama de Comunicação

O diagrama de comunicação era conhecido como diagrama de colaboração até a versão 1.5 da *UML*, tendo seu nome modificado para diagrama de comunicação a partir da versão 2.0 (GUEDES, 2005).

Um diagrama de comunicação é um diagrama de interação que dá ênfase à organização estrutural dos objetos que enviam e recebem mensagens (BOOCH et al., 2000). A colaboração entre objetos é representada por uma ligação simples acompanhada de uma numeração seqüencial e de outras informações como condições e iterações (MELO, 2004). A figura 2.8 mostra um exemplo de diagrama de comunicação.

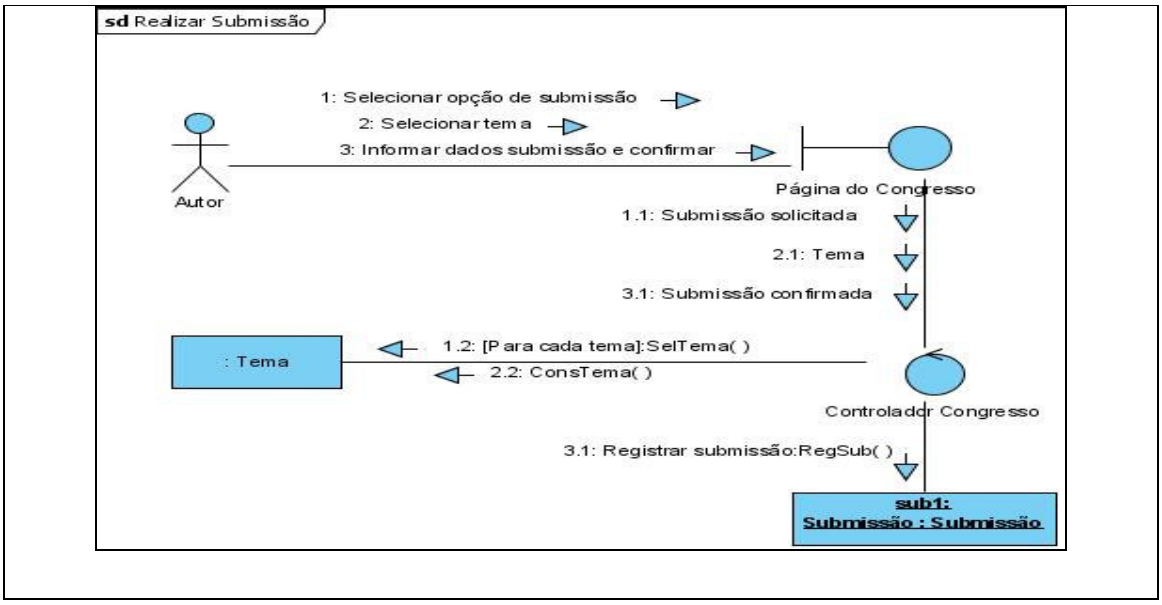

Figura 2.8 – Diagrama de Comunicação

2.2.9 Diagrama de Máquina de Estados

Este diagrama demonstra o comportamento de um elemento através de um conjunto de transições de estado (GUEDES, 2005). Este diagrama é usado para ilustrar a visão dinâmica de um sistema. Os diagramas de máquina de estados são importantes principalmente para se fazer a modelagem do comportamento de uma interface, classe ou colaboração. Eles dão ênfase ao comportamento de um objeto, solicitado por eventos, que é de grande ajuda para a modelagem de sistemas reativos (BOOCH et al., 2000). A figura 2.9 traz um exemplo deste diagrama.

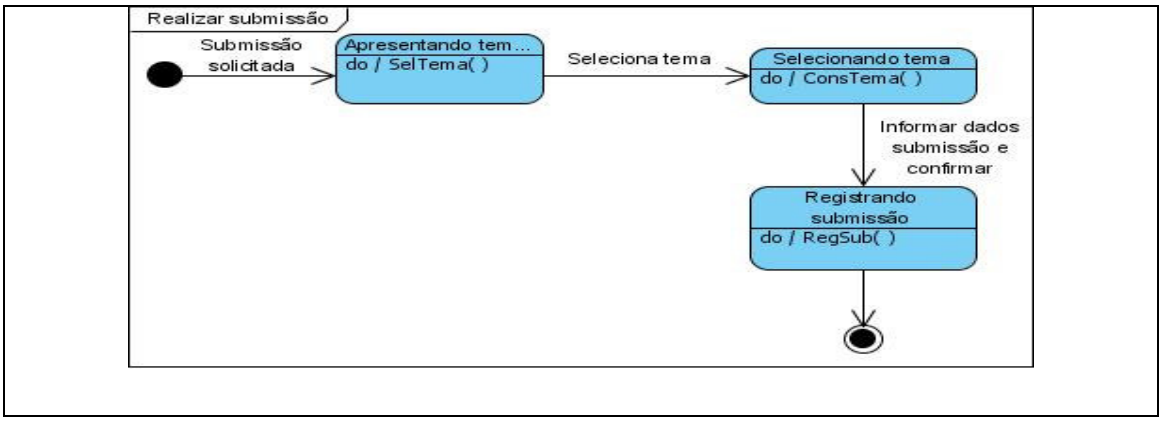

Figura 2.9 – Gráfico de Máquina de Estados

2.2.10 Diagrama de Pacotes

A *UML* 2.0 oficializou este diagrama. Nas versões anteriores, quando era necessário mostrar pacotes e suas dependências era utilizado um diagrama de classes para fazer isto (MELO, 2004).

Um pacote é um mecanismo de propósito geral para a organização de elementos em grupos. Um pacote é representado graficamente como uma pasta com uma guia (BOOCH et al., 2000; BEZERRA, 2002). A figura 2.10 mostra um exemplo de Diagrama de Pacotes.

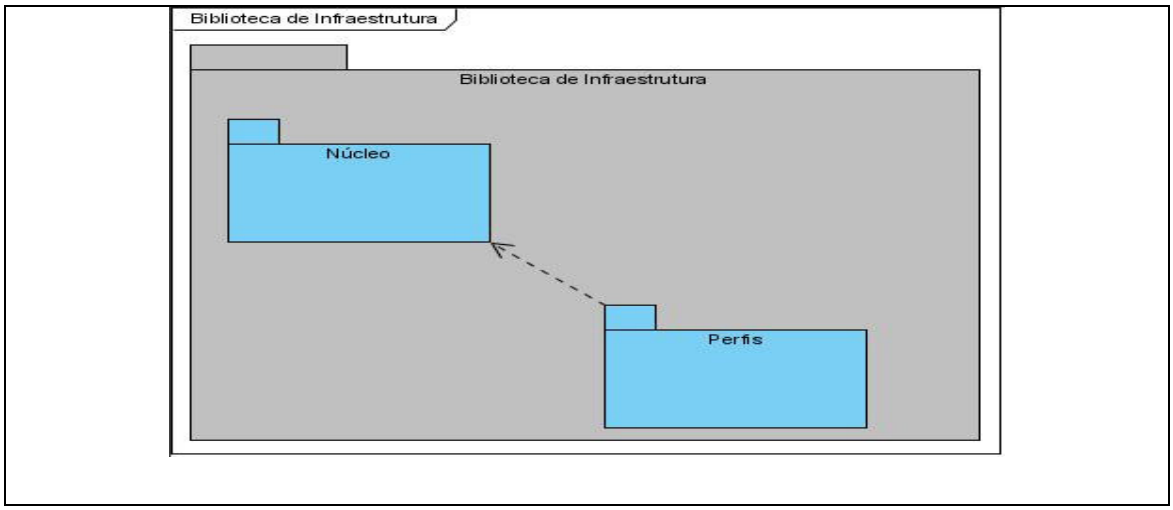

Figura 2.10 – Diagrama de Pacotes

2.2.11 Diagrama de Estrutura Composta

Este é mais um dos diagramas incluídos na versão 2.0 da *UML*. O objetivo deste diagrama, segundo Melo (2004), é permitir que se exiba um pequeno diagrama de classes dentro de uma classe (também pode ser usado com outros classificadores). Este tipo de notação leva a uma apresentação menos confusa de um relacionamento de composição. Guedes (2005) amplia o conceito deste diagrama, segundo ele o diagrama de estrutura composta é utilizado para modelar colaborações. O termo estrutura deste diagrama refere-se a uma composição de elementos interconectados, representando instancias de tempo de execução colaborando através de vínculos de comunicação para atingir algum objetivo comum. A figura 2.11 mostra um exemplo deste diagrama.

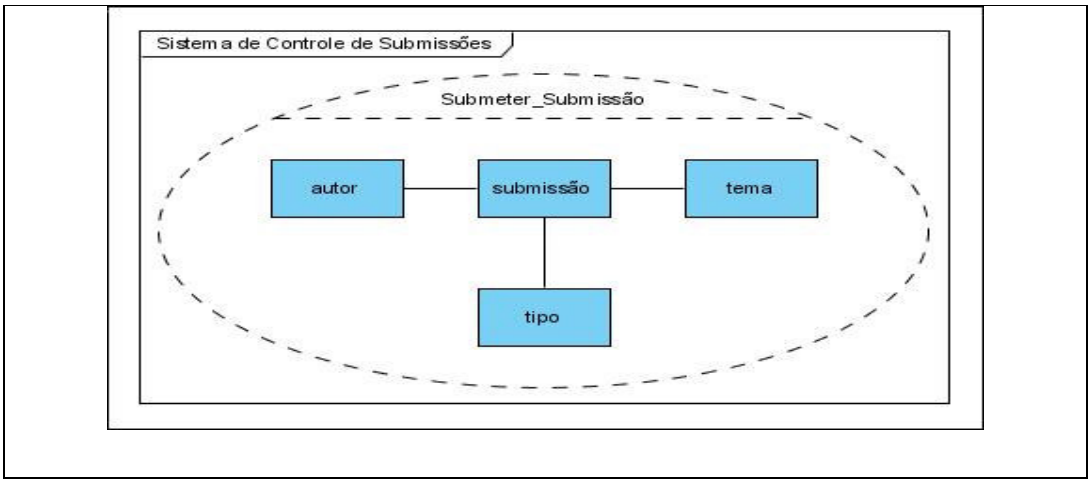

Figura 2.11 – Diagrama de Estrutura Composta

2.2.12 Diagrama de Visão Geral

Este diagrama incluído na versão 2.0 consiste numa especialização do diagrama de atividades representando interações, promovendo uma visão geral do fluxo de controle. Este diagrama utiliza as notações gráficas do diagrama de atividades, onde os nós (atividades) ou são interações ou são ocorrências de interações (MELO, 2004; GUEDES, 2005). O exemplo a seguir, figura 2.12, mostra um diagrama de visão geral.

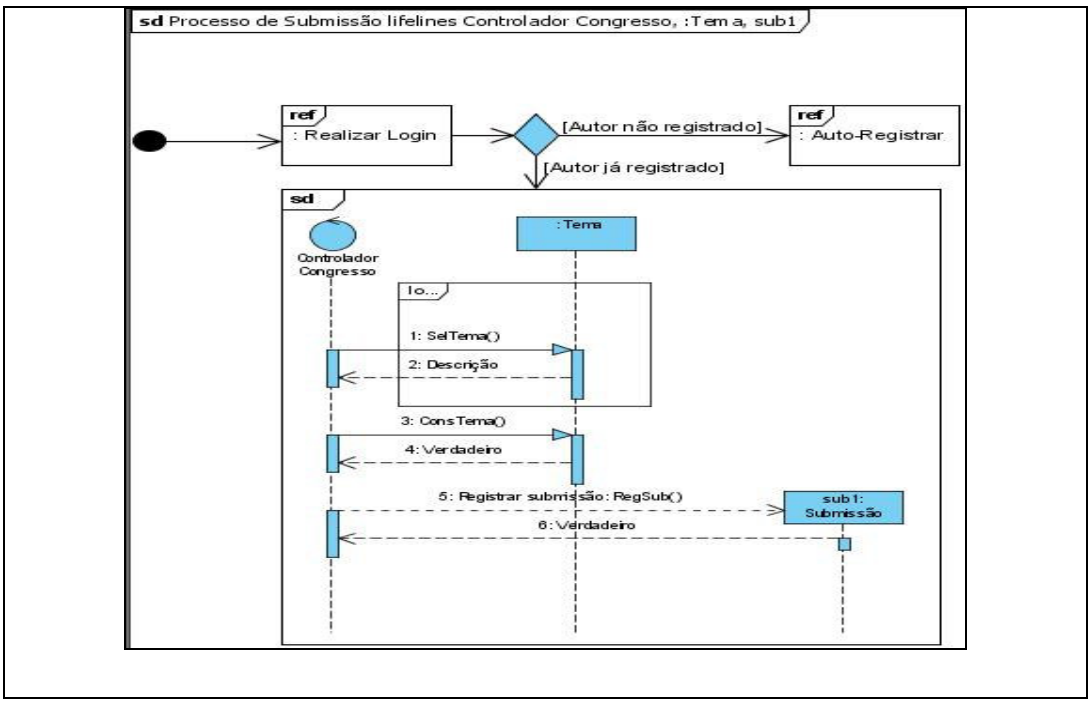

Figura 2.12 – Diagrama de Visão Geral

2.2.13 Diagrama Temporal

O diagrama de tempo ou temporal mostra as mudanças de um objeto ao longo do tempo, em resposta a eventos externos. Este diagrama terá pouca utilidade para modelar aplicações comerciais, porem poderá ser utilizado na modelagem de sistemas de tempo real ou sistemas que utilizem recursos multimídia/hipermídia, onde o tempo em que um objeto executa algo é muitas vezes importante (GUEDES, 2005; MELO, 2004). A figura 2.14 mostra um exemplo deste diagrama.

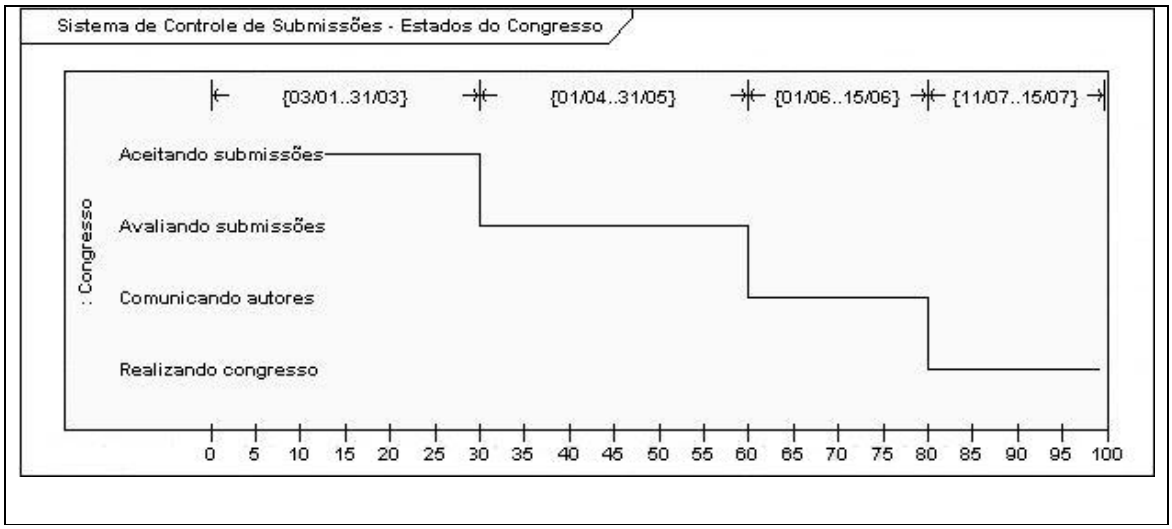

Figura 2.13 – Diagrama Temporal

O próximo capítulo irá introduzir conceitos sobre Padrões de Projeto, tais como seu significado, sua utilização e sua relação com os frameworks orientados a objetos, que são à base do estudo deste trabalho.

3 PADRÕES DE PROJETO

O conceito de padrão – *pattern* – foi extensamente estudado e desenvolvido pelo arquiteto norte-americano Christopher Alexander nas décadas de 60 e 70 do séc. XX em livros como *Notes on Synthesis of Form*, *The Structure of the Environment e A Pattern Language.* Christopher Alexander afirma que cada padrão descreve um problema que ocorre repetidas vezes em nosso ambiente, e então descreve o núcleo da solução para aquele problema, de tal maneira que pode-se usar essa solução mais de um milhão de vezes sem nunca fazê-la da mesma forma. Embora esta definição diga respeito a padrões de arquitetura de cidades, casas e prédios, essa definição pode muito bem ser utilizada para padrões de projeto orientados a objetos, onde pode-se fazer um paralelo entre objetos e interfaces com paredes e portas (GAMMA et al., 2000).

Segundo Booch et al. (2000) um padrão fornece uma solução comum para um problema básico em um determinado contexto.

3.1 Elementos de um Padrão de Projeto

Em geral um padrão tem quatro elementos essenciais (GAMMA et al., 2000):

- **Nome do padrão**: é uma referência usada para descrever um problema de projeto, suas soluções e conseqüências em uma ou duas palavras;
- **Problema**: Quando aplicar o padrão, em que condições;
- **Solução**: Descrição abstrata de um problema e como usar os elementos disponíveis (classes e objetos) para solucioná-lo;
- **Conseqüências**: Custos e benefícios de se aplicar o padrão. Impacto na flexibilidade, extensibilidade, portabilidade e eficiência do sistema.

3.2 Descrevendo os Padrões de Projeto

Segundo Gamma et al. (2000) existem muitas formas de descrever padrões de projeto, tais como notações gráficas. Porem somente este tipo de notação não é suficiente, pois captura somente o produto final de processo do projeto como relacionamentos entre classes e objetos. Para atingir um grau de reutilização de um padrão de projeto é necessário também o registro de decisões tomadas durante o projeto, as alternativas encontradas para dado problema, as análises de custos e benefícios relacionados e ainda alguns exemplos concretos.

Para facilitar a compreensão de padrões de projetos cada padrão foi dividido em seções de acordo com o gabarito a seguir (GAMMA et al., 2000):

- **Nome e Classificação**: O nome que é dado ao padrão expressa sua própria essência de forma sucinta. Este nome é de extrema importância, pois ele se tornará parte do vocabulário do projeto;

- **Intenção e Objetivo**: Corresponde a um texto explicando algumas perguntas básicas sobre o projeto, tais como: o que faz o padrão de projeto? Quais os seus princípios e sua intenção? Que tópico ou problema particular de projeto ele trata;

- **Também conhecido como**: Outros nomes conhecidos que se relacionam com esse padrão, se estes existirem;

- **Motivação**: Um cenário que ilustre um problema de projeto e como o padrão poderá solucionar este problema;

- **Aplicabilidade**: Situações onde o padrão poderá ser aplicado. Exemplos de projetos mal estruturados que o padrão pode tratar. Como reconhecer as situações onde o padrão poderá ser aplicado;

- **Estrutura**: Representação gráfica das classes relacionadas ao padrão;

- **Participantes**: Classes e/ou objetos que participam do padrão de projeto e quais suas responsabilidades;

- **Colaborações**: Colaboração entre os participantes para executar alguma tarefa de suas responsabilidades;

- **Conseqüências**: Como o padrão suporta a realização de seus objetivos? Quais seus custos e benefícios e os resultados? Que aspectos da estrutura do sistema permite varias independentemente;

- **Exemplo de Código**: Trechos de códigos exemplificado como o padrão pode ser implementado;

- **Usos conhecidos**: Exemplos deste padrão que já estão incorporados a alguma aplicação real;

- **Padrões relacionados**: Quais os padrões a que este padrão esta relacionado? Quais as diferenças? Com quais outros padrões este padrão deveria ser utilizado.

3.3 Catálogo de padrões de projeto

O catalogo organizado por Gamma et.al. (2000) divide-se em 23 padrões de projeto, citados e explicados a seguir:

- ***Abstract Factory***: Fornece uma *interface* para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas;

- ***Adapter***: Converte a interface de uma classe em outra interface esperada pelos clientes. O *Adapter* permite que certas classes trabalhem em conjunto, pois de outra forma seria impossível por causa de suas interfaces incompatíveis;

- ***Bridge***: Separa uma abstração da sua implementação, de modo que as duas possam variar independentemente;

- ***Builder***: Separa a construção de um objeto complexo da sua representação, de modo que o mesmo processo de construção possa criar diferentes representações;

- ***Chain of Responsibility***: Evita o acoplamento do remetente de uma solicitação ao seu destinatário, dando a mais de um objeto a chance de tratar a solicitação. Encadeia os objetos receptores e passa a solicitação ao longo da cadeia até que um objeto a trate;

- ***Command***: Encapsula uma solicitação como um objeto, desta forma permitindo que se parametrize clientes com diferentes solicitações, enfileire ou registre (*log*) solicitações e suporte operações que podem ser desfeitas;

- ***Composite***: Compõe objetos em estrutura de árvore para representar hierarquias do tipo parte-todo. O *Composite* permite que os clientes tratem objetos individuais e composições de objetos de maneira uniforme;

- ***Decorator***: Atribui responsabilidades adicionais a um objeto dinamicamente. Os *decorators* fornecem uma alternativa flexível a subclasses para extensão da funcionalidade;

- ***Façade***: Fornece uma interface unificada para um conjunto de interfaces em um subsistema. O *Facade* define uma interface de nível mais alto que torna o subsistema mais fácil de usar;

- ***Factory Method***: Define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe a ser instanciada. *O Factory Method* permite a uma classe postergar (*defer*) a instanciação às subclasses;

- ***Flyweight***: Usa compartilhamento para suportar grandes quantidades de objetos, de granularidade fina, de maneira eficiente;

- ***Interpreter*:** Dada uma linguagem, define uma representação para sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças nessa linguagem;

- ***Iterator***: Fornece uma maneira de acessar seqüencialmente os elementos de um objeto agregado sem expor sua representação subjacente;

- ***Mediator***: Define um objeto que encapsula a forma como um conjunto de objetos interage. O *Mediator* promove o acoplamento fraco ao evitar que os objetos se refiram explicitamente uns aos outros, permitindo que se varie suas interações independentemente;

- ***Memento***: Sem violar o encapsulamento, captura e externaliza um estado interno de um objeto, de modo que o mesmo possa posteriormente ser restaurado para este estado;

- ***Observer***: Define uma dependência um-para-muitos entre objetos, de modo que, quando um objeto muda de estado, todos os seus dependentes são automaticamente notificados e atualizados;

- ***Prototype*:** Especifica os tipos de objetos a serem criados usando uma instância prototípica e cria novos objetos copiando esse protótipo;

- ***Proxy***: Fornece um objeto representante (*surrogate*), ou um marcador de outro objeto, para controlar o acesso ao mesmo;

- ***Singleton***: Garante que uma classe tenha somente uma instância e fornece um ponto global de acesso a ela;

- ***State***: Permite que um objeto altere seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado de classe;

- ***Strategy***: Define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. O *Strategy* permite que o algoritmo varie independentemente dos clientes que o utilizam;

- ***Template Method***: Define o esqueleto de um algoritmo em uma operação, postergando a definição de alguns passos para subclasses. O *Template Method* permite que as subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura;

- ***Visitor*:** Representa uma operação a ser executada sobre os elementos da estrutura de um objeto. O *Visitor* permite que se defina uma nova operação sem mudar as classes dos elementos sobre os quais opera.

3.4 Organizando o catálogo de padrões de Projeto

Segundo Gamma et al. (2000) os padrões de projeto variam na sua granularidade e no seu nível de abstração. Como existem muitos padrões de projeto, é necessário organizá-los.

A classificação dos padrões de projeto se da através de dois critérios:

- **Propósito**: Reflete o que o padrão faz. Os padrões podem ter finalidade de criação, estrutural ou comportamental. Os padrões de criação preocupam-se com o processo de criação de objetos. Os padrões estruturais trabalham com a composição de classes ou objetos. Os padrões comportamentais caracterizam as maneiras pelas quais classes ou objetos interagem e distribuem responsabilidades;

- **Escopo**: Especifica se o padrão se aplica primariamente a classes ou a objetos. Os padrões para classes lidam com os relacionamentos entre classes e suas subclasses. Esses relacionamentos são estabelecidos através do mecanismo de herança, sendo assim, são estáticos. Os padrões para objetos lidam com relacionamentos entre objetos que podem ser mudados em tempo de execução e são considerados mais dinâmicos. Quase todos utilizam a herança em certa medida.

A tabela 3.1 mostra a forma de classificação dos padrões de projetos.

		Propósito		
		De Criação	**Estrutural**	**Comportamental**
Escopo	**Classe**	*Factory Method*	*Adapter*	*Interpreter* *Template Method*
	Objeto	*Abstract Factory* *Builder* *Prototype* *Singleton*	*Adapter* *Bridge* *Composite* *Decorator* *Façade* *Flyweight* *Proxy*	*Chain of Responsibility* *Command* *Iterator* *Mediator* *Memento* *Observer* *State* *Strategy* *Visitor*

Tabela 3.1 – Classificação dos padrões de projeto

3.5 Padrões de Projeto e *Frameworks*

Padrões de projeto e *frameworks* têm algumas similaridades, por este motivo são confundidos muitas vezes. Uma das principais características destas duas técnicas diz respeito à capacidade de reuso que as duas implementam. Para diferenciá-lo pode-se considerar três aspectos principais (GAMMA, et al., 2000):

- **Padrões de projetos são mais abstratos que *frameworks***: *Frameworks* podem ser representados em código, porem somente exemplos de padrões podem ser representados em códigos. Um dos pontos fortes dos frameworks é que eles podem ser escritos utilizando uma linguagem de programação, sendo não apenas estudados, mas executados e reutilizados diretamente. Em contraposição, os padrões tem que ser implementados cada vez que eles são usados. Os padrões também explicam as intenções, custos e benefícios e conseqüências de um projeto;

- **Padrões de projeto são elementos de arquitetura menores que *frameworks***: Um *framework* simples pode conter vários padrões de projeto, enquanto a recíproca não é verdadeira;

- **Padrões de projeto são menos especializados que *frameworks***: Os *frameworks* sempre têm um domínio particular de aplicação.

> Em contraposição a isso, um padrão de projeto pode ser utilizado para qualquer tipo de aplicação.

Sendo assim os padrões de projeto constituem-se numa parte importante na construção de *frameworks*, pois através deles pode-se alcançar um nível mais alto de organização, reusabilidade de projeto e código. Os padrões ajudam a tornar a arquitetura de um *framework* adequada a muitas aplicações diferentes, sem necessidade de reformulação (GAMMA et al.).

A capitulo seguinte versará sobre alguns conceitos a respeito da utilização de *frameworks* orientados a objetos.

4 *Frameworks* Orientados a Objetos

Este capítulo irá introduzir os conceitos a respeito da utilização de *frameworks*. Os *frameworks* estão cada vez mais presentes na área de desenvolvimento de software, pois são a maneira pela qual os sistemas orientados a objetos conseguem a maior reutilização (GAMMA et. al, 2000).

Na literatura podem ser encontradas muitas definições para *frameworks* orientados a objetos, a seguir são apresentadas algumas delas.

Segundo Silva (2000) à abordagem de *frameworks* orientados a objetos utiliza o paradigma de orientação a objetos para produzir uma descrição de um domínio para ser reutilizada. Sendo assim um *framework* é uma estrutura de classes inter-relacionadas, que correspondem a uma implementação incompleta para um conjunto de aplicações de um domínio, sendo que esta estrutura de classes deve ser adaptada para a geração de aplicações específicas.

Para Pree (1995) um *framework* é uma coleção de classes abstratas e concretas que representam um subsistema, estas classes (abstratas e concretas) podem ser estendidas ou adaptadas para construir um novo subsistema.

Para D'Souza e Wills (1998) um *framework* pode ser interpretado como sendo um pacote de *templates*, um pacote desenvolvido para ser importado com substituições. Ele é entendido para prover uma nova versão, baseada nas substituições realizadas.

Segundo Gamma et al. (2000) um *framework* predefine os parâmetros de projeto, de maneira que o projetista/implementador da aplicação possa se concentrar nos aspectos específicos da sua aplicação.

Segundo Mattsson (1996) *framework* é uma arquitetura desenvolvida com o objetivo de se obter a máxima reutilização, representada como um conjunto de classes abstratas e concretas, com grande potencial de especialização.

4.1 *Framework* X Bibliotecas

É importante salientar a diferença entre *frameworks* e bibliotecas de classes. *Frameworks* diferem de bibliotecas porque apresentam uma inversão do fluxo de controle, entre os *frameworks* e os desenvolvedores dos *frameworks*. Projetos baseados em bibliotecas forçam o desenvolvedor a reescrever todo o fluxo de controle da aplicação, enquanto projetos baseados em *frameworks* colocam o *framework* no controle do fluxo, onde o desenvolvedor reescreve apenas métodos especiais que adaptam alguma funcionalidade do *framework*, os quais são chamados pelo próprio *framework* (GAMMA, 2000; JOHNSON; FOOTE, 1988).

4.2 Framework Model View Controler (MVC)

Um dos primeiros *frameworks* que se tem conhecimento é o *MVC* desenvolvido no ambiente do *Smalltalk*-80 (PREE, 1995). Este *framework* é muito utilizado hoje em dia na comunidade de software.

Através do *MVC* é possível dividir uma aplicação em três camadas distintas, desta forma separando a lógica da aplicação chamada de *Model* (modelo), da interface do usuário conhecida como *View* (visão) e do fluxo da aplicação, o *Controller* (controlador).

Cada camada anteriormente citada corresponde a uma classe abstrata do framework MVC, e elas cooperam através de um protocolo de interação bem definido. Essas classes são definidas da seguinte maneira (GERBER, 1999):

- *Model*: um objeto do tipo *Model* pode manter objetos dependentes (*views* e *controllers*), e enviar mensagens para notificar modificações no modelo para os dependentes. A classe *Model* define um protocolo de mensagens especifico para estes propósitos. Dessa forma um modelo de uma aplicação é considerado como sendo uma instância de uma subclasse de *Model*, que deve implementar o comportamento específico da aplicação;

- *View*: um objeto *View* pode ser dividido em outros objetos do tipo *View*, chamados de subvisões. A classe *View* define um protocolo para interagir com objetos do tipo *controller* e *model* e interagir com suas subvisões, desta forma coordenando ações de transformação e apresentação. Um objeto *View* em uma aplicação é uma instância de uma subclasse de *View*. A subclasse tem que fornecer a implementação para o protocolo de apresentação, para definir detalhes específicos do modelo que representa;

- *Controller*: esta classe define os protocolos para manipular os objetos do tipo *Model* e *View* a partir de mensagens de interação enviadas pelo usuário através de determinados dispositivos de entrada de dados.

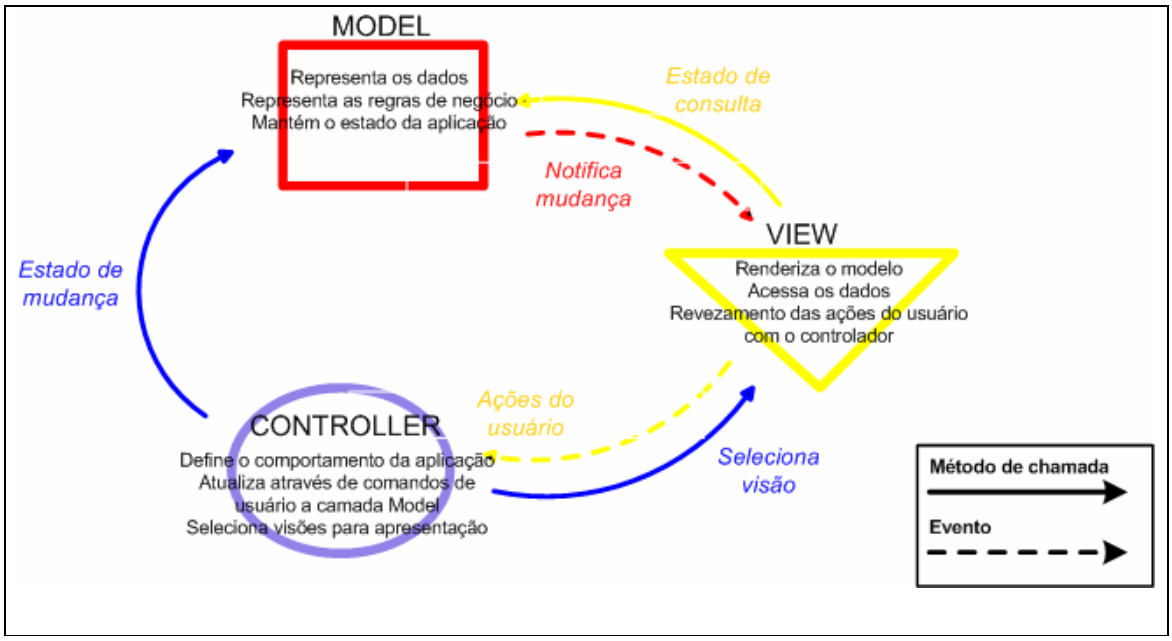

Figura 4.1 - MVC

4.3 Classificação dos *Frameworks*

A classificação de *frameworks* se da de várias formas e de vários pontos de vista, segundo alguns autores, caracterizando tipos distintos de *frameworks* de acordo com seus propósitos e de acordo com as técnicas usadas para entendê-los.

4.3.1 Quanto ao Propósito

Fayad e Schmidt dividem este tipo de *framework* em infra-estrutura, integração e domínio de aplicação (FAYAD; SCHMIDT, 1997).

Frameworks de Infra-estrutura: são usados para dar suporte a diversos tipos de aplicações, independentemente dos domínios endereçados por estas aplicações, desta forma simplificando o desenvolvimento da infra-estrutura. Exemplos conhecidos deste *framework* seriam *frameworks* para sistemas operacionais, sistemas de comunicação, interfaces com o usuário e ferramentas para processamento de linguagens naturais;

Frameworks de Integração: são usados normalmente para integrar aplicações e componentes distribuídos. *Frameworks* de integração são desenvolvidos também para aumentar as habilidades de desenvolvedores de *software* em modularizar, reusar e estender a infra-estrutura do *software* para trabalhar em um ambiente distribuído. Exemplos comuns deste *framework* incluem o ORB (*Object Request Broker*) *Frameworks* e bases de dados transacionais;

Frameworks de Domínio de Aplicação: representam esqueletos de aplicações para domínios específicos, como por exemplo, para telecomunicações, manufatura e engenharia financeira. O desenvolvimento deste tipo de *framework* geralmente é caro, assim como sua compra. Entretanto eles podem prover um retorno substancial do investimento, quando utilizados para o suporte ao desenvolvimento de aplicações para usuário final e para a criação de produtos em domínios específicos. *Frameworks* deste tipo capturam elementos invariantes de domínio, e deixam em aberto os aspectos específicos de cada aplicação.

Algumas diferenças básicas entre *Frameworks* de Domínio de Aplicação e *Frameworks* de Infra-estrutura e Integração esta no fato que estes dois preocupam-se basicamente com problemas internos de desenvolvimento de software e são independentes do domínio de aplicação. Entretanto esses *frameworks* são essenciais para o desenvolvimento de *softwares* de qualidade, porém não geram um retorno significativo do investimento. Para as empresas, este tipo de *framework* é mais vantajoso ser adquirido do que desenvolvido internamente.

4.3.2 Quanto as Técnicas de Reuso

De acordo com esse escopo os *frameworks* são divididos em *Frameworks* de Caixa Branca (*whitebox*), *Frameworks* de Caixa Preta (*blackbox*) e *Frameworks* de Caixa Cinza (*graybox*)(FAYAD; SCHMIDT, 1997).

Em *Frameworks* de Caixa Branca o desenvolvedor deve estender as classes que constituem o *framework* para criar aplicações especificas, isto é, a reutilização é obtida através da herança. Isso dificulta muito a utilização, pois o desenvolvedor precisa entender como funciona cada detalhe do *framework* para poder utilizá-lo. Dessa forma, ensinar como funciona este tipo de *framework* é quase a mesma coisa que ensinar a criá-lo (JOHNSON; FOOTE, 1988; FAYAD; SCHMIDT, 1997).

Em um *Framework* de Caixa Preta o desenvolvedor combina diversas classes concretas existentes do *framework* para obter uma aplicação, desta forma o reuso é obtido através das composições feitas com estas classes. Neste caso o desenvolvedor precisa apenas entender as *interfaces* para poder utilizá-lo (JOHNSON; FOOTE, 1988; FAYAD; SCHMIDT, 1997).

Johnson afirma que o ideal é que cada *framework* de caixa branca evolua para um *framework* de caixa preta com o tempo. Dessa forma ele diz que *frameworks* de caixa branca devem ser vistos como sendo uma fase natural na evolução de um sistema, que será algum dia um passo na evolução de uma coleção de métodos em um conjunto de componentes (JOHNSON; FOOTE, 1988).

Um *Framework* de Caixa Cinza é considerando como sendo uma mistura dos conceitos dos *frameworks* de Caixa Branca e Caixa Preta. Dessa forma o reuso é obtido por meio da herança, por ligação dinâmica e também pelas *interfaces* de definição (BRAGA, 2002).

Um aspecto variável de um domínio de aplicação é chamado de ponto variável (*hotspot*) (BUSCHMAN et al., 1996). Diferentes aplicações dentro de um mesmo domínio são diferenciadas por um ou mais pontos variáveis. Eles representam as partes do *framework* de aplicação que são especificas de cada sistema. Os pontos variáveis são projetados para serem genéricos e podem ser adaptados às necessidades da aplicação (BRAGA, 2002).

Pontos fixos (*frosen-spots*) definem a arquitetura geral de um sistema de *software*, seus componentes básicos e os relacionamentos entre eles. Estes pontos permanecem sem nenhuma modificação em todas as instanciações de um *framework* de aplicação (BRAGA, 2002).

Segundo um estudo intitulado "*Application Frameworks: A Survey*" realizado por Yassin e Fayad (FAYAD; JOHNSON, 1999), a tendência de desenvolvimento é pelos *frameworks* de caixa cinza. Apesar dos *frameworks* de caixa branca serem mais fáceis de desenvolver, a exposição do código fonte aos desenvolvedores pode causar problemas. Já os de caixa preta são mais abstratos e menos flexíveis, dificultando desta forma a manutenção de aplicações derivadas dele. *Frameworks* de caixa cinza quando bem projetados superam as barreiras impostas pelos outros dois tipos, porque possuem alta flexibilidade e uma grande facilidade de extensão, sem expor desnecessariamente informações que não interessam aos desenvolvedores (BRAGA, 2002).

Em resumo, um *framework* caixa branca é muito mais fácil de projetar, pois não existe a necessidade de se prever todas as alternativas de implementação possíveis ao contrário dos de caixa preta, que necessitam dessa previsão. No entanto os *frameworks* de caixa preta são mais fáceis de utilizar, pois basta apenas escolher a implementação desejada, enquanto os de caixa branca é necessário fornecer um implementação completa. *Frameworks* de caixa cinza tentam tirar o máximo proveito das vantagens dos outros dois tipos de *framework*, sendo assim, considerado com um meio termo. *Frameworks* de caixa branca podem evoluir para se tornar cada vez mais caixa preta (JOHNSON, 1997). Isto pode ser alcançado de forma gradativa, implementando-se várias alternativas que posteriormente podem ser utilizadas na instanciação do *framework* (BRAGA, 2002).

4.4 Compreensão e Uso de *Frameworks*

Utilizar um *framework* consiste em criar aplicações a partir deste *framework*. A motivação para se usar um *framework* na construção de aplicações é a perspectiva de um aumento considerável da produtividade e da qualidade do sistema desenvolvido, em função da reutilização promovida. Para produzir uma aplicação utilizando um *framework* como base, deve-se estender sua estrutura de modo a cumprir os requisitos básicos da aplicação a que se quer desenvolver (SILVA, 2000).

Um dos obstáculos encontrados para se desenvolver aplicações através do uso de *frameworks* é justamente compreender seu uso, como funciona e interage internamente. A questão é que se um desenvolvedor de aplicações necessitar empreender um esforço considerável para aprender a utilizar um *framework*, que para desenvolver uma aplicação do inicio, a utilidade deste *framework* acaba se tornando questionável (SILVA, 2000).

Dessa forma torna-se imperativo do processo de desenvolvimento de *frameworks* a produção de subsídios para minimizar o esforço do desenvolvedor em compreender como utilizar os *frameworks* produzidos. Johnson classifica diferentes níveis de compreensão de um *framework*: saber que tipos de *softwares* podem ser desenvolvidas através da utilização do *framework* (a qual domínio de aplicações ele pertence), ser capaz de desenvolver aplicações elementares sob o *framework* e conhecer os detalhes a respeito do projeto do *framework*. Para capacitar ao desenvolvimento de aplicações elementares recomenda o desenvolvimento de uma documentação no estilo *cookbook*[8], que não entre em detalhes de projeto (JOHNSON, 1994). A utilização de *frameworks* a partir de *cookbooks* torna-se limitada quanto a gama de aplicações que podem ser desenvolvidas: se um determinado requisito de um aplicação não é tratado no conjunto de instruções, o mecanismo não consegue auxiliar o desenvolvimento. Para suprir tal deficiência, usuários de *frameworks*, invariavelmente, são forçados a buscar conhecimento sobre o projeto do *framework*. Dessa forma, é extremamente importante que a descrição de um *framework* fornecida a um usuário o habilite a desenvolver aplicações com o menor esforço possível, mas que também forneça os meios de se aprofundar a compreensão do projeto do *framework*, a fim de habilitá-lo a produzir aplicações mais complexas, sem que lhe seja exigido um esforço excessivo. O fator de comparação diz que o esforço compreendido para desenvolver uma aplicação através de um *framework* não deve ser maior que o necessário para produzir a mesma aplicação sem utilizar o *framework* (SILVA, 2000; JOHNSON, 1994).

4.5 Componentes Envolvidos no Desenvolvimento e Uso de *Frameworks*

O desenvolvimento tradicional de aplicações envolve dois tipos de personagens: o desenvolvedor de aplicação e o usuário de aplicação. A tarefa dos desenvolvedores consiste em levantar os requisitos de uma determinada aplicação, desenvolve-la (o que inclui toda a

[8] Livro de Receitas - Documentação que mostra como utilizar um *framework* através de exemplos passo a passo.

documentação que ensina a utilizar a aplicação) e entregá-la aos usuários. Os usuários por sua vez interagem com a aplicação apenas através da *interface* que lhes é oferecida. A figura 4.1 exemplifica essa relação entre desenvolvedores e usuários de uma aplicação (SILVA, 2000).

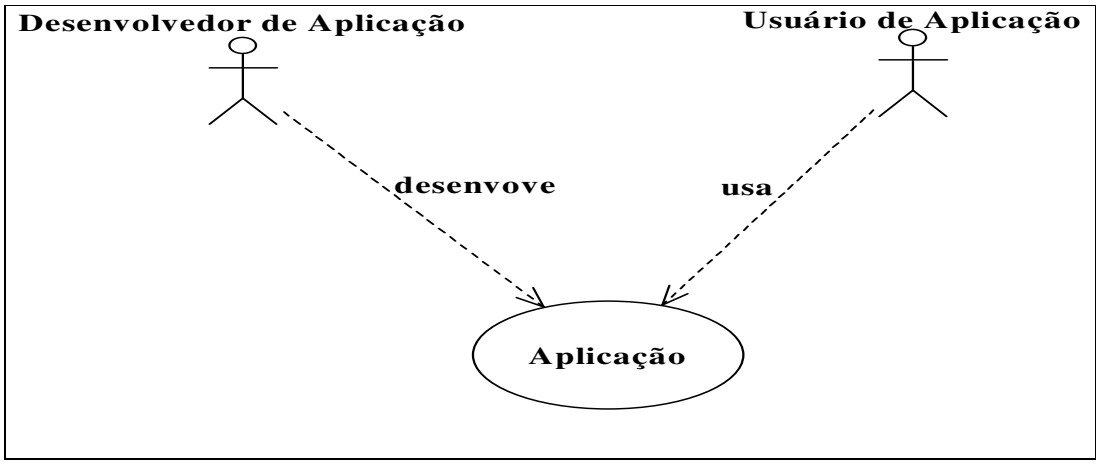

Figura 4.2 – Componentes do Desenvolvimento tradicional de aplicações

O desenvolvimento de *frameworks* introduz outro componente, alem do desenvolvedor e usuário de aplicação: o desenvolvedor de *framework*. No contexto dos frameworks, o papel do usuário de aplicação permanece o mesmo. O papel do desenvolvedor de aplicações difere do caso anterior pela inserção do *framework* no processo de desenvolvimento de aplicações. Dessa forma o desenvolvedor de aplicações se torna um usuário do *framework*, que deve estender e adaptar a estrutura deste *framework* para a produção de aplicações. Ele permanece com as mesmas funções do caso anterior: obter os requisitos da aplicação, desenvolve-la usando o *framework* (o que em geral, não dispensa o desenvolvedor de aplicações da necessidade de produzir código) e entregá-la aos usuários. O novo componente criado no contexto dos *frameworks*, o desenvolvedor de *framework*, tem a responsabilidade de produzir *frameworks* e algum modo de ensinar os usuários do *frameworks* a desenvolver aplicações. A figua 4.2 exemplifica essa nova relação (SILVA, 2000).

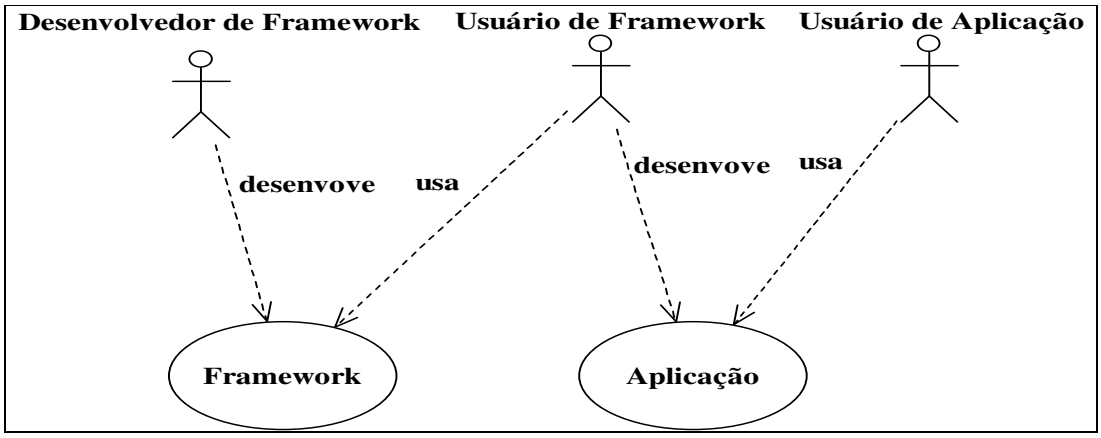

Figura 4.3 – Componentes do desenvolvimento de aplicações baseadas em *frameworks*

4.6 Construção de *Frameworks*

Na construção de um *framework* diversas atividades não seqüenciais se repetem, até se obter uma estrutura de classes que contemple os requisitos de generalidade, flexibilidade e extensibilidade. Para uma melhor compreensão podem-se organizar estas atividades que constituem o processo de desenvolvimento da estrutura de classes de um *framework* nas seguintes etapas: etapa de generalização, etapa de flexibilização, etapa de aplicação de metapadrões, etapa de aplicações de padrões de projeto e etapa de aplicação de princípios práticos de orientação a objetos (SILVA, 2000).

A etapa de generalização consiste no processo de identificar estruturas idênticas nas aplicações analisadas, e na fatoração de estruturas em que identificam semelhanças, de modo a obter uma estrutura de classes que generalize o domínio tratado. Para a busca de generalidade da estrutura do *framework*, um dos primeiros aspectos é a identificação das classes que representam elementos e conceitos do domínio tratado. Isto pode ser obtido através da confrontação de estruturas de classes de diferentes aplicações. Nesta etapa também são pesquisados elementos prontos que podem ser reutilizados, como por exemplo, outros *frameworks* e componentes (SILVA, 2000; BRAGA, 2002).

Na etapa de flexibilização o principal objetivo é identificar o que deve ser mantido flexível na estrutura de classes que generaliza o domínio, de modo que a estrutura possa ser especializada, gerando diferentes aplicações. É um processo de localização de pontos variáveis na estrutura de classes e de escolha de soluções de projeto para modelá-los. A identificação destes pontos variáveis ocorre quando se determinam situações de

processamento comuns às aplicações do domínio (os pontos fixos), porem, com variações de uma aplicação para outra (SILVA, 2000; BRAGA, 2002).

Quando identificado um ponto variável, deve-se comparar o requisito de flexibilização por ele imposto com os casos tratados por padrões existentes. Se um padrão é considerado adequado para a solução do problema, ele pode ser incorporado a estrutura de classes do *framework*. No caso especifico dos meta-padrões, o seu uso consiste em transformar um procedimento geral em um método *template*, cujo comportamento é flexibilizado através da dependência de métodos *hook* (gancho), que podem ter diferentes implementações (SILVA, 2000).

A etapa de aplicação de padrões de projeto consiste em incluir as classes de um padrão selecionado na estrutura de classes do *framework* (ou fazer com que classes preexistentes assumam responsabilidades correspondentes as classes do padrão de projeto)(SILVA, 2000; BRAGA, 2002).

Segundo Johnson e Foote (1988) para se desenvolver um *framework* com uma estrutura de classes flexíveis são necessários seguir alguns princípios básicos da orientação a objetos, como por exemplo, o uso da herança para a reutilização de *interfaces*, ao invés do uso de herança para a reutilização de código. A reutilização de código através de composição de objetos, a preocupação em promover polimorfismo na definição d classes e métodos, de modo a possibilitar acoplamento dinâmico.

4.7 Documentação de *Frameworks*

Frameworks são projetos reusáveis e não apenas linhas de código, são mais complexos de se desenvolver do que simples aplicações, tornando sua documentação mais difícil de ser realizada.

A documentação de *frameworks* deve atender a diversos requisitos. Estes requisitos podem ser alcançados fazendo uma estruturação da documentação através do uso de padrões ou de alguma linguagem de padrões. O principal propósito deste conjunto de padrões é mostrar como utilizar um *framework*, não apenas mostrar como ele funciona. Com a utilização de padrões pode-se descrever o propósito de um *framework*, permitindo aos

usuários do *framework* desenvolverem sem ter a necessidade de conhecer os detalhes internos de funcionamento (RÉ, 2002).

Geralmente as documentações dos *frameworks* descrevem primeiramente como um *framework* trabalha e em seguida como utilizá-lo. No entanto, não é possível entender completamente a estrutura de um *framework* até que ele tenha sido utilizado efetivamente. A utilização de exemplos é uma boa forma de comprovar se a forma de utilização foi ou não bem entendida pelo usuário, alem de tornar os frameworks mais completos e facilitarem o entendimento do fluxo de controle (JOHNSON, 1992).

4.8 Vantagens e Desvantagens do Uso de *Frameworks*

Segundo Fayad e Schmidt (1997) os benefícios primários da utilização de *frameworks* são os seguintes:

Modulariedade: *frameworks* aumentam a modulariedade através de um encapsulamento de detalhes de implementação flexíveis, sob *interfaces* estáveis. A modulariedade do *framework* auxilia no aumento da qualidade do *software* localizando lugares de impacto no projeto e os impactos que mudanças na implementação podem causar. Esta localização reduz o esforço necessário para compreender e manter um *software* existente.

Reusabilidade: as interfaces estáveis providas pelo *framework* aumentam a reusabilidade através da definição de componentes genéricos que podem ser replicados para criar novas aplicações. A reusabilidade fornecida pelo *framework* permite que desenvolvedores utilizem seus conhecimentos no domínio do problema para evitar a recriação e a revalidação de soluções já conhecidas, presentes no *framework*. O reuso de componentes da estrutura do *framework* pode render melhorias substanciais na produtividade dos desenvolvedores, aumentando a qualidade, o desempenho, a confiabilidade e a interoperabilidade do *software*.

Extensibilidade: Provê métodos explícitos que possibilitam às aplicações estenderem suas *interfaces*. A extensibilidade do *framework* é essencial para assegurar uma customização oportuna de serviços e características novas em uma aplicação.

Inversão de Controle: a arquitetura de um *framework* é caracterizada pela inversão de controle. Esta arquitetura permite que a aplicação possa processar eventos que podem ser customizados e chamados pelo próprio *framework*. Quando estes eventos ocorrem o *framework* reage invocando métodos pré-definidos de cada aplicação, que executam processos pré-definidos nos eventos.

Segundo Mattsson (1996) as principais desvantagens no uso e construção de *frameworks* envolvem:

A dificuldade de desenvolver bons *frameworks*, pois é necessário ter um alto conhecimento no domínio do *framework* para poder construí-lo de forma que atenda as necessidades.

A documentação do *framework* é crucial para o desenvolvedor de aplicações. Se esta documentação não estiver bem elaborada irá dificultar o uso do *framework* para desenvolvimento de novas aplicações.

O processo de depuração de aplicações originadas através de um *framework* pode ser muito complicado, pois é muito difícil distinguir se um erro esta no código do *framework* ou no da aplicação. Se os erros estão localizados no *framework* pode ser impossível para o usuário consertá-los.

Em alguns casos a generalidade e a flexibilidade de um *framework* podem trabalhar contra ele, reduzindo a eficiência em uma aplicação em particular.

Quanto mais customizado um componente é, mais ele se adapta a uma situação em particular, mas mais esforço é necessário para aprender a usá-lo e para usá-lo efetivamente.

5 *API's* de Mecanismos de Busca

Este trabalho tem por objetivo primeiramente a construção de um *framework* para PLN utilizando *API's* de mecanismos de busca, e posteriormente o desenvolvimento de uma aplicação simples baseada nesse mesmo *framework*. O processamento da linguagem natural é feito sobre um conjunto de textos chamado corpus. Para montar este conjunto é necessário à reunião de diversos textos, para isto este trabalho propõe que este corpus seja montado utilizando a base de dados contida na *web*. Para se ter acesso a essa base de dados alguns mecanismos de busca disponibilizam gratuitamente suas *API's* de pesquisa na *web* para desenvolvimento de aplicações não comerciais. O presente trabalho irá utilizar as *API's* disponibilizadas pelo *Google*, *Yahoo* e *Technorati* para realizar pesquisas e montar o corpus necessário para o processamento das técnicas de PLN.

5.1 *Google API*

A *Google Web API* nada mais é do que uma licença para se utilizar a base de dados do mecanismo, desde que seja para uso pessoal, onde são disponibilizadas ferramentas e metodologias para troca de dados entre um software e a *API*. Esta *API* foi disponibilizada pelo mecanismo *Google* como forma de manter seu desenvolvimento e comprometimento com seus usuários, dessa forma tornando sua base disponível no futuro a este tipo de utilização, massificar seu uso e se manter como referência em inovação quando se fala em busca de informações na *web* (TAGLIASSUCHI, 2002). Para se obter essa licença ou chave de pesquisa é necessário que o desenvolvedor registre uma conta na página do *Google*.

Através desta *API* o desenvolvedor tem acesso a uma infinidade de informações contidas nas paginas que estão armazenadas na base de dados do *Google*, dessa forma permitindo que aplicações possam buscar e manipular estas informações, de acordo com a vontade do desenvolvedor de *software*.

De acordo com o *Google* (2006) as suas *API's* podem ser usadas para construir diversas aplicações, deixando na mão do desenvolvedor a tarefa de criá-las de acordo com sua genialidade. Alguns exemplos de possibilidades citados pelo mecanismo são:

- Criar mecanismos que ficam monitorando a *web* regularmente em busca de novas informações a respeito de um determinado assunto;
- Criar mecanismos que realizam pesquisas de mercado através da analise de algumas diferenças nas informações disponibilizadas;
- Fazer pesquisas utilizando interfaces não *HTML*[9], como linhas de comando e aplicações;
- Criar jogos inovadores utilizando as informações disponíveis na web.

A tabela 5.1 lista todos os parâmetros válidos que podem ser usados quando for feita uma solicitação de pesquisa através da *API* e descreve como estes parâmetros podem influenciar na pesquisa realizada.

Nome	Descrição
key	Chave fornecida pelo *Google*, esta chave é requerida para ter acesso ao Serviço do *Google*. O *Google* a utiliza para autenticar e logar no mecanismo.
q	Termos de pesquisa que serão utilizados na requisição. Ex.: "+", "-", "OR", etc.
start	Zero baseado no índice do primeiro resultado desejado.
maxResults	Número de resultados desejados por pesquisa realizada. O Valor máximo é de 10 resultados.
filter	Ativa ou desativa o filtro de resultados. Este filtro é responsável por mostrar ou esconder informações similares no resultado.
restricts	Restringe a pesquisa. Ex.: Realizar pesquisas que retornem somente resultados de paginas de um determinado país ou determinado tópico.

[9] *HTML* – do inglês *Hiper Text Markup Language* (Linguagem de Formatação de Hipertexto). Linguagem de marcação utilizada para produzir páginas na Internet. De modo geral são documentos de texto escritos em códigos que podem ser interpretados pelos *browsers* para exibir as páginas da *World Wide Web*. Disponível em http://www.w3.org/MarkUp/.

safeSearch	Um valor boleano que habilita os filtros de conteúdo adulto para a pesquisa.
lr	*Language Restrict* – Restringe a pesquisa a documentos de uma determinada língua.
ie	*Imput Encoding* – Funcionalidade em desuso que será retirada nas próximas versões da *API*. Todas as pesquisas devem ser feitas utilizando *UTF*-8[10].
oe	*Output Encoding* – Funcionalidade em desuso que será retirada nas próximas versões da *API*. Todas as pesquisas devem ser feitas utilizando *UTF*-8.

Tabela 5.1 – Parâmetros de Pesquisa do *Google (GOOGLE, 2006)*

A tabela a seguir, 5.2 mostra algumas limitações impostas pelo mecanismo do *Google* para a realização de pesquisas através do uso da *API*. Segundo o *Google* (2006) algumas destas limitações são decorrentes da infra-estrutura do mecanismo, que esta otimizada somente para uso de usuários finais (usuários que acessam o *Google* para fazer algum tipo de pesquisa). Entretanto futuramente esses limites serão alterados.

Componente	Limite
Tamanho da requisição de pesquisa	2048 *bytes*
Número máximo de palavras na pesquisa	10
Número máximo do site: termos na pesquisa	1 (por cada pesquisa)
Número máximo de resultados por pesquisa	10
Valor máximo de <*start*> + <*maxResults*>	1000

Tabela 5.2 – Limitações da *API* do *Google (GOOGLE, 2006)*

Alem das limitações quanto à realização de pesquisas, a *API* também tem algumas limitações no que diz respeito às linguagens que ela esta preparada para suportar (linguagens que já foram homologadas). As linguagens que esta *API* da suporte são as seguintes: *Java* (*Apache SOAP* e *Apache Axis*), *Perl* (*SOAP*:*Lite version* 0.52), *Ruby* (*SOAP4R*), e *C#* no *MS Visual Studio .NET*. O *Google* adverte que a *API* pode ser usado com qualquer outro tipo de linguagem, deste que essa tenha suporte a *web services* (*GOOGLE*, 2006).

[10] *UTF-8* – Do inglês *8-bit Unicode Transformation Format.* É um tipo de codificação Unicode de comprimento variável criado por Ken Thompson e Rob Pike. Pode representar qualquer caractere universal padrão do *Unicode*, sendo também compatível com o *ASCII*. Por esta razão, está lentamente sendo adotado como tipo de codificação padrão para e-mail, páginas web, e outros locais onde os caracteres são armazenados. Disponível em http://www.unicode.org/.

Para realizar algum tipo de pesquisa através da *API* do *Google* o desenvolvedor deve submeter uma pesquisa ao *Google* com os parâmetros anteriormente citados. O *Google* por sua vez devolve um documento (no formato *XML*[11]) contendo algumas *tags*, conforme tabela 5.3.

Tags	Descrição
documentFiltering	Um valor boleano que indica se um filtro foi utilizado na pesquisa. Será verdadeiro somente quando: (a) Foi requisitado um filtro e (b) o filtro realmente ocorreu.
searchComments	Um texto contendo as palavras mais comuns que foram removidas automaticamente da pesquisa. Estas palavras são conhecidas como "*stop-words*". Ex.: "e", "como", onde, etc.
estimatedTotalResultsCount	Valor estimado do total de resultados para a pesquisa realizada.
estimateIsExact	Um valor boleano indicando se o valor estimado corresponde ao valor real de resultados.
resultElements	Uma lista de itens que corresponde aos resultados da pesquisa.
searchQuery	Onde estarão representados os valores informados em "q", conforme tabela 5.1.
startIndex	Indica o índice do primeiro resultado da pesquisa
endIndex	Indica o índice do ultimo resultado da pesquisa.
searchTips	Um texto contendo algumas informações de como utilizar o *Google*.
directoryCategories	Uma lista de itens contendo as categorias de diretórios. Corresponde ao diretório *ODP*[12] da pesquisa.
searchTime	O tempo em segundos em que a pesquisa foi realizada.

[11] *XML* – Do inglês *Extensible Markup Language* (Linguagem de Marcação Expansível). É um subtipo de *SGML* (acrônimo *de Standard Generalized Markup Language*, ou *Linguagem Padronizada de Marcação Genérica*) capaz de descrever diversos tipos de dados. Seu propósito principal é a facilidade de compartilhamento de informações através da Internet. Disponível em http://www.w3.org/XML/.

[12] *ODP* – Do inglês *Open Directory Project* (Projeto de Diretório Aberto). O ODP é o mais amplo e abrangente diretório da web editado por humanos. Ele é construído e mantido por uma vasta comunidade global de editores voluntários. Disponível em http://dmoz.org/.

summary	Se a pesquisa retornou diretórios *ODP*, o resumo deles é mostrado aqui.
URL	A *URL* do resultado da pesquisa.
snippet	Mostra um resumo do local onde os termos de pesquisa estão presentes, dentro da *URL*.
title	Título do resultado da pesquisa. Geralmente o nome da página.
cachedSize	Texto contendo o tamanho do cachê (inteiro + k). Indica se o cachê da *URL* esta disponível. O tamanho é mostrado em *kilobytes*.
relatedInformationPresent	Booleano indicando se o termo da pesquisa relacionada é suportado pela *URL*
hostName	Nome do *host*.
directoryCategory	Categoria do diretório pesquisado.
directoryTitle	Nome do diretório.
fullViewableName	Texto contendo o nome do diretório ODP, para a corrente categoria.
specialEncoding	Especifica o esquema de codificação da informação do diretório.

Tabela 5.3 – *Tags* retornadas pelo *Google (GOOGLE, 2006)*

Para maiores informações sobre a utilização das *API's Google* disponibiliza para o desenvolvedor diversas formas de suporte, como *FAQ's*, exemplos e alguns grupos de pesquisa sobre o assunto.

5.2 *Yahoo API*

O mecanismo de busca *Yahoo* também disponibiliza sua vasta base de dados através de suas *API's*. Através destes serviços desenvolvedores podem ter acesso a diversas funcionalidades disponibilizadas pelo mecanismo. As aplicações desenvolvidas devem ser reportadas ao *Yahoo*, como uma forma de manter o nível dos serviços prestados e também devem obedecer a política de uso do mecanismo (*YAHOO*, 2006).

A tabela 5.4 mostra a definição dos parâmetros que devem ser enviados para fazer uma requisição de consulta através da *API* do *Yahoo*.

Parâmetro	Valor	Descrição
appid	*string* (obrigatório)	O identificador da aplicação.
query	*string* (obrigatório)	O que o usuário deseja pesquisar.
region	*string*: padrão = *us*	A região em que se deseja pesquisar. Ex.: Brasil = Br, Argentina = ar, etc.
type	*all* (padrão), *any*, or *phrase*	Tipo de consulta: • *All* – retorna os resultados baseados em todos os termos da pesquisa; • *Any* – Qualquer resultado, desde que tenha um ou mais termos da pesquisa; • *Phrase* – Retorna resultados contendo os termos pesquisados em uma única frase.
results	*integer*: padrão *10*, max *100*	O número de resultados que a pesquisa deve retornar.
start	*integer*: padrão *1*	A posição inicial do resultado. A última posição (*start* + *results* – 1) não deve ser superior a 1000.
format	*any* (padrão), *html*, *msword*, *pdf*, *ppt*, *rss*, *txt*, *xls*	Especifica que tipo de arquivo esta sendo procurado.
adult_ok	*no value* (padrão), or *1*	Especifica se a pesquisa deve retornar conteúdo adulto. O número 1 habilita a pesquisa nesse tipo de site.
similar_ok	*no value* (padrão), or *1*	Especifica se deverão ser omitidos os resultados similares.
language	*string*: padrão *no value* (todas)	Linguagem em que será feita a pesquisa.
country	*string*: padrão *no value*	O país para restringir a busca somente em sites alocados em cada país.

site	*string*: padrão *no value*	Um domínio para restringir uma pesquisa.
subscription	*string*: padrão *no value*	Qualquer descrição para um conteúdo Premium que também deve ser consultado. È possível submeter várias descrições.
license	*any* (padrão), *cc_any*, *cc_commercial*, *cc_modifiable*	*Creative Commons license*(site da web: http://creativecommons.org/): licença para utilizar alguns termos.
output	*string*: *xml* (padrão), *json, php*	Formato de retorno da consulta.
callback	*String*	Parâmetro válido somente quando o formato de saída for igual a *json* (*JavaScript Object Notation*).

Tabela 5.4 – Parâmetros de Pesquisa do *Yahoo (YAHOO, 2006)*

Assim como o *Google* o *Yahoo* também limita a utilização da sua *API*. O limite para consultas por IP de usuário é de 5000 requisições por dia, sendo que para cada consulta realizada serão permitidos no mínimo 10 e no máximo 100 resultados. Desta forma se o usuário por algum motivo trocar o IP, terá o seu limite restabelecido. Outra forma de aumentar o limite, segundo o *Yahoo* (2006), é cadastrar mais de um "*appid*" (*Application ID*) no site. Este *appid* é responsável por identificar a aplicação do usuário no *Yahoo*.

A API de busca do *Yahoo* contém bibliotecas que suportam o desenvolvimento nas seguintes linguagens: *Perl, Python, PHP, Java, JavaScript e Flash*.

Através do preenchimento de um formulário fornecido pelo *Yahoo* o desenvolvedor pode solicitar alterações nas *API's* disponibilizadas, essas solicitações serão analisadas pelo *Yahoo* e poderão ser realizadas num momento oportuno (*YAHOO*, 2006).

Para realizar algum tipo de pesquisa através da *API* do *Yahoo* o desenvolvedor deve submeter uma pesquisa ao *Yahoo* com os parâmetros anteriormente citados. O *Yahoo* por sua vez devolve um documento (no formato *XML*) contendo algumas *tags*, conforme tabela 5.5.

Tags	Descrição
ResultSet	Contem todos os resultados da pesquisa. Possui os seguintes atributos: • ***totalResultsAvailable***: O número de resultados encontrados na base de dados do mecanismo, para a consulta realizada; • ***totalResultsReturned***: O número de resultados encontrados e retornados. Este valor pode ser menor que o número de resultados requisitados; • ***firstResultPosition***: A posição do primeiro resultado de toda a pesquisa.
Result	Contem cada resposta individual da consulta realizada.
Title	O título da página encontrada na consulta.
Summary	Resumo associado com a página encontrada na consulta.
URL	Endereço da pagina encontrada na consulta.
ClickURL	O endereço que esta apontando para a página.
MimeType	O tipo *MIME*[13] da página.
ModificationDate	A data da ultima modificação da página.
Cache	O endereço do cachê e o seu tamanho em *bytes*.

Tabela 5.5 – *Tags* retornadas pelo *Yahoo (YAHOO, 2006)*

O *Yahoo* tem diversas formas de suporte para o desenvolvedor que queira utilizar suas *API's* para desenvolvimento de aplicações, entre elas destaca-se o *FAQ* e as listas de e-mail para suporte.

5.3 *Technorati API*

A *Technorati* possui uma dos maiores acervos de *blogs* da *internet*, contendo 59, 8 milhões de *blogs* cadastrados no seu banco de dados. Segundo dados da própria empresa, por dia são cadastrados mais de 175 mil *blogs*, com um total de 1,6 milhões de *posts* diários.

[13] MIME - Do inglês *Multipurpose Internet Mail Extensions* (Extensões Multifunção para Mensagens de Internet). É uma norma da internet para o formato das mensagens de correio eletrônico. A grande maioria das mensagens de correio eletrônico são trocadas usando o protocolo *SMTP* e usam o formato *MIME*. As mensagens na *Internet* têm uma associação tão estreita aos padrões *SMTP* e *MIME* que algumas vezes são chamadas de mensagens *SMTP/MIME*. Disponível em http://www.ietf.org/.

Através da sua *API* de busca o mecanismo disponibiliza toda esta vasta quantidade de *blogs* para desenvolvedores criarem suas próprias aplicações para acessar esses dados (*TECHNORATI*, 2006). A *Technorati* incentiva o desenvolvimento de aplicações que utilizem sua *API's*, para tornar seu mecanismo mais conhecido na comunidade de software.

Assim como as *API's* anteriores, para se ter acesso às funcionalidades o desenvolvedor precisa criar uma conta no site e solicitar uma chave para utilizar as *API's* em suas aplicações.

A tabela 5.6 traz os parâmetros necessários para se realizar uma pesquisa através da *API* de buscas da *Technorati*.

Parâmetros	Descrição
key	A chave para utilizar a *API* (obrigatório).
query	O que o usuário deseja pesquisar (obrigatório).
format	Formato de retorno da consulta. *XML* ou *RSS*[14].
language	A região em que se deseja pesquisar. Ex.: Brasil = Br, Argentina = ar, etc.
authority	Uma forma de fazer um ranking dos *blogs* mais visitados contidos no banco de dados. Possui as seguintes opções: • *n* – Todos os resultados; • *a1 – Blogs* que contém pelo menos um link apontando pra eles; • *a4 – Blogs* que contém muitos links apontado para eles; • *a7 – Blogs* que contém milhares de links apontando para eles.
start	Onde quer se começar a ver os resultados, obrigatoriamente deve ser maior que zero.
limit	Limite de *links* retornados na pesquisa. Mínimo 1 e máximo 100.
Claim	Informações do proprietário do *blog* pesquisado. '1' significa que

[14] *RSS* - Arquivo em *XML* que serve para compartilhar informações e notícias entre sites e entre sites e programas. Disponível em http://www.rssfeeds.com.br.

	deve mostrar e ‘0’ não mostrar.

Tabela 5.6 – Parâmetros de Pesquisa do *Technorati (TECHNORATI, 2006)*

Conforme tabela acima a *API* possui algumas limitações, como por exemplo, o número de links que podem ser retornados, 100 links por pesquisa. Não existe um limite de consultas que podem ser feitas por dia ou por ip como as *API’s* do *Google* e *Yahoo*.

A *API* da *Technorati* tem suporte ao desenvolvimento de aplicações através das seguintes linguagens: C*#, Java, Perl, PHP, Python, REBOL, Ruby, e Visual Basic*.

Para realizar algum tipo de pesquisa através da *API* do *Technorati* o desenvolvedor deve submeter uma pesquisa ao mecanismo com os parâmetros anteriormente citados. O *Technorati* por sua vez devolve um documento (no formato *XML* ou *RSS*) contendo algumas *tags*, conforme tabela 5.7.

Tags	Descrição
`query`	Palavras pesquisadas.
`Querycount`	Número de links encontrados
`Querytime`	Tempo em que a pesquisa retornou os resultados.
`Rankingstart`	Posição em que a pesquisa irá começar.
`name`	Nome do *blog* onde foram encontradas as palavras informadas pelo usuário.
`URL`	Endereço da pagina.
`rssURL`	Endereço do *RSS* da pagina.
`atomURL`	Endereço do *Atom*[15]
`inboundblogs`	*Blogs* contidos no resultado
`inboundlinks`	Links contidos no resultado.
`lastupdate`	Última atualização do *Blog*.

[15] *Atom* - Nome dado a um estilo baseado em conteúdo *XML* e meta data, ou seja, é um protocolo ao nível da aplicação para publicar e editar Fontes web que são periodicamente atualizadas, como por exemplo *Blogs*. Os *feeds* devem ser formados em formato *XML* e são identificados como *application/atom+xml media type*. Disponível em http://www.atomenabled.org/.

`title`	Título da entrada do *Blog*.
`excerpt`	Resumo contendo as palavras pesquisadas destacadas.
`created`	Quando foi criada a entrada.
`permalink`	Endereço para a entrada do *blog*.

Tabela 5.7 – Tags retornadas pelo Technorati (TECHNORATI, 2006)

6 Processamento da Linguagem Natural (PLN)

Este capítulo versará sobre alguns conceitos a respeito de PLN, um pouco do seu histórico e principalmente o estudo de alguns tipos de ferramentas que utilizam seus recursos. Este estudo irá servir para dar uma base para a construção do *framework* proposto. As ferramentas que serão estudadas estão classificadas nas seguintes categorias:

- Tagger, Morphological Analyzer - Analisadores morfológicos e etiquetadores;
- Stemmer – Radicalizadores de palavras;
- Parsers - Analisadores sintáticos;
- Corpus Tools - Ferramentas para Manipulação de Corpus.

6.1 Conceitos de PLN

"A PLN visa tratar a linguagem natural com o objetivo de desenvolver teorias computacionais da linguagem, usando noções de algoritmos e estrutura de dados provenientes da ciência da computação" (ABRAHÃO, 1997, p. 10).

Para Chaves (2003a) o PLN é o conjunto de métodos formais para analisar textos e gerar frases escritas em um idioma humano.

Segundo Vieira e Lima (2001) PLN envolve a construção de programas capazes de interpretar e/ou gerar informação fornecida em linguagem natural. Para o processamento da língua natural, são necessários vários subsistemas para dar conta dos diferentes aspectos da língua: sons, palavras, sentenças e discurso nos níveis estruturais, de significado e de uso.

Para Nunes et al. (1999) PLN compreende a criação de programas computacionais "inteligentes", até mesmo capazes de "compreender" as línguas e, por meio delas, simular uma interação verbal com o usuário.

"O PLN é uma forma artificial desenvolvida com a finalidade de aprender através da linguagem natural (LN), artifícios que possam ser utilizados com recursos computacionais, para desenvolver programas capazes de tratar os problemas relacionados à LN" (VINHAES, 2005, p.20).

Segundo Jackson e Moulinier (2002), o termo Processamento de Linguagem Natural (PLN) é normalmente usado para descrever a função de *softwares* ou componentes de *hardware* em um sistema computacional que analisam ou sintetizam linguagem falada ou escrita. O termo "natural" tem a função de diferenciar a fala e a escrita humana de linguagens mais formais, tais como notações matemáticas ou lógicas, ou ainda linguagens de computador, tais como Java, LISP e C++.

6.2 Histórico do PLN

Na década de 50 a comunidade científica começava com os primeiros estudos sobre PLN. Segundo Nunes et al. (1999) as primeiras investigações institucionalizadas sobre o Processamento da Linguagem Natural começaram a ser desenvolvidas no início da década de 50, depois da distribuição de 200 cópias de uma carta, conhecida como *Weaver Memorandum*, escrita por Warren Weaver, então vice-presidente da Fundação Rockfeller e exímio conhecedor dos trabalhos sobre criptografia computacional. Nessa carta, divulgada em 1949, Weaver convidava universidades e empresas, interessados potenciais, para desenvolver projetos sobre um novo campo de pesquisa que ficou conhecido como ''tradução automática'', ''tradução mecanizada'' ou simplesmente MT (abreviação do inglês "*Machine Translation*").

Em 1960 foram desenvolvidas as primeiras formalizações do significado em termos de redes semânticas, os primeiros tratamentos computacionais das gramáticas livres de contexto e a criação dos primeiros analisadores. Na década de 70 veio à consolidação dos estudos do PLN com a implementação de parcelas das primeiras gramáticas e analisadores

sintáticos e a busca de formalização de fatores pragmáticos e discursivos. Em 1980 começava a sofisticação dos sistemas com o desenvolvimento de novas teorias lingüísticas motivadas pelo estudo do PLN. Nos anos 90 surgiam os sistemas baseados em representação do conhecimento com o desenvolvimento de projetos de sistemas de PLN complexos que buscavam a integração dos vários tipos de conhecimentos lingüísticos e extralingüísticos e das estratégias de inferência envolvidas nos processos de produção, manipulação e interpretação de objetos lingüísticos (NUNES et al., 1999).

6.3 *Tagger, Morphological Analyser* (Etiquetador, Analisador Morfológico)

A morfologia além de estudar a estrutura das palavras também se preocupa com a classificação das palavras em categorias, ou, conforme um termo muito conhecido na área de PLN, as palavras são classificas em partes do discurso (*part-of-speech*, ou *POS*). Essas categorias podem ser divididas em substantivos, verbos, adjetivos, preposições e advérbios. Ainda pode-se dividir as categorias em classes abertas ou fechadas. Abertas são constituídas por categorias que abrangem um grande número de palavras e podem, ainda, abrigar o surgimento de palavras novas. Nessa classe encontra – se os substantivos, verbos e adjetivos. Classes fechadas são aquelas que possuem funções gramaticais bem definidas, como por exemplo, artigos, demonstrativos, quantificadores, conjunções e preposições (VIEIRA; LIMA, 2001; CHAVES, 2003a).

Sendo assim a função de um etiquetador morfológico é associar a cada palavra de uma sentença uma determinada etiqueta, que corresponda a sua categoria morfológica ou gramatical (MENEZES; NETO, 2002).

Para a atribuição das etiquetas morfológicas o etiquetador realiza as seguintes tarefas: associa etiquetas possíveis às palavras da sentença, avalia a probabilidade de cada etiqueta e define a etiqueta mais provável para cada palavra da sentença (examina o contexto) (KINOSHITA et al., 2005).

Uma das grandes dificuldades existentes na tarefa da classificação morfológica encontra-se em sua susceptibilidade à ambigüidade. Um etiquetador morfológico robusto deve levar em conta não apenas as informações lexicais da palavra a ser anotada, mas também

informações a respeito do contexto em que esta inserida (MENEZES; NETO, 2002; VIEIRA; LIMA, 2001; ROCHE; SCHABES, 1995).

6.4 *Stemmer* (Radicalizador de Palavras)

Conflação é o ato de fusão ou combinação para realizar o processo de igualar variantes morfológicas de termos. A conflação pode ser feita de duas formas, uma é a manual, utilizando algum tipo de expressão regular, ou ainda automática, através do uso de programas chamados *stemmers*. Para reduzir as variantes ou variações de uma palavra para uma forma única são utilizados algoritmos de conflação. Existem dois métodos distintos de conflação para se obter tal redução: *Stemming* e redução à forma canônica ou lematização (CHAVES, 2003b; GONZALEZ, 2005)

Stemming consiste no processo de combinar as formas diferentes de uma palavra em uma representação comum, chamada de radical (*stem*). Ele não é necessariamente igual a raiz lingüística, porém permite tratar diferentes variações de uma palavra da mesma forma. Por exemplo, “conector” e “conectores” são essencialmente iguais, mas sem sofrerem a redução por *stemming* serão tratadas como palavras diferentes (DIAS; MALHEIROS, 2006; CHAVES, 2003b). Como pode ser observado a seguir.

No processo de *stemming* podem ser detectados dois tipos de erros, conhecidos como *overstemming* e *understemming*. *Overstemming* ocorre quando a parte removida da palavra não é um sufixo, mas parte do *stem*. Desta forma palavras não relacionadas podem ser combinadas. Por exemplo, a palavra “confortável” após ter sido processada por um radicalizador, é transformada no *stem* “confor”. Como pode ser observado foi removido parte do radical correto, a saber, “confort”. *Understamming* ocorre quando um sufixo não é removido completamente. Por exemplo, quando a palavra “referência” é transformada incorretamente no *stem* “referênc”, quando deveria ter sido transformada no *stem* “refer” (DIAS; MALHEIROS, 2005; CHAVES, 2003b).

Outro problema que se pode observar no processo de *stemming* é que ele não tem sucesso com termos onde a flexão é raramente usada ou inexiste (por exemplo, nomes próprios) (GONZALEZ, 2005).

A forma canônica ou lema é a forma pela qual constam as palavras na nomenclatura de um dicionário usual: masculino singular, para os nomes e adjetivos (pato, gordo), feminino singular, para nomes e adjectivos exclusivamente feminos (mesa, grávida) e infinitivo para os verbos (lavar, comer, fugir). Por exemplo, seguindo as regras acima, a forma canônica de gatos e gatas é gato (BAPTISTA, 1994; NUNES et al., 1996).

Um problema que pode ser detectado no método de lematização é que algumas palavras da mesma família morfológica podem ser classificadas como sendo diferentes. Por exemplo, a palavra "caminhei" ficaria igual a "caminhar", enquanto a palavra da mesma familia "caminhada" ficaria igual a "caminhada" (GONZALEZ, 2005).

6.5 *Parser* (Analisador Sintático)

O processamento automático da sentença com o objetivo do reconhecimento de sua estrutura sintática é chamado de *parsing*. Por extensão, a ferramenta que executa esse conjunto de procedimentos (que permite assinalar funções sintáticas a cada um dos itens lexicais que compõem a sentença) é referenciada como *parser* (NUNES, 1999).

Através da utilização de um *parser* é possível construir uma árvore de derivação, que explicita as relações entre as palavras que formam uma sentença. O analisador sintático faz uso do léxico, que reúne o conjunto de itens lexicais da língua, e de uma gramática, que define as regras de combinação dos itens na formação das frases (sintagmas[16]) (VIEIRA; LIMA, 2001).

Segundo Silva e Chapetta (2003) todo *parser* deve incluir uma gramática que especifique como as sentenças serão interpretadas. Pode-se considerar esta gramática como sendo um espaço de configurações, onde cada configuração representa estágios da combinação dos constituintes da sentença que esta sendo interpretada. Transições entre configurações descrevem como os constituintes são combinados para derivar outros constituintes mais extensos.

[16] Sintagma é a unidade da análise sintática composta de um núcleo e de outros termos que a ele se juntam, formando uma locução que irá compor a oração. O nome do sintagma depende da classe da palavra que forma seu núcleo, havendo assim sintagma nominal (núcleo nome), sintagma adjetivo (núcleo adjetivo), sintagma preposicional (núcleo preposição) etc (COELHO et al., 2005).

6.6 *Corpus Tools*

Corpus Tools são ferramentas destinadas à manipulação e criação de corpus para PLN. A seguir serão apresentadas algumas definições sobre o significado de um corpus:

"Uma coletânea de textos naturais (*naturally occurring*), escolhidos para caracterizar um estado ou variedade de linguagem" (SINCLAIR, 1991, p. 171).

"[Corpus é] um corpo de linguagem natural (autêntica) que pode ser usado como base para pesquisa lingüística" (SINCLAIR, 1991, p. 171).

"Corpus é uma coletânea de porções de linguagem que são selecionadas e organizadas de acordo com critérios lingüísticos explícitos, a fim de serem usadas como uma amostra da linguagem" (PERCY et al., 1996, p. 4).

"Um corpo de material lingüístico que existe em formato eletrônico e que pode ser processado por computador para vários propósitos" (LEECH; MACENERY, 1997, p. 1).

"Corpus de material lingüístico natural (textos inteiros, amostra de textos, ou às vezes somente sentenças desconexas), que são armazenadas em formato legível por máquina" (LEECH, 1991, p. 115-116).

"Uma coletânea grande e criteriosa de textos naturais (BIBER et al., 1998 , p. 4)".

> [...] um conjunto de dados lingüísticos (pertencentes ao uso oral ou escrito da língua, ou a ambos), sistematizados segundo determinados critérios, suficientemente extensos em amplitude e profundidade, de maneira que sejam representativos da totalidade do uso lingüístico ou de algum de seus âmbitos, dispostos de tal modo que possam ser processados por computador, com a finalidade de propiciar resultados vários e úteis para a descrição e análise [...] (SANCHEZ, 1995, p. 8-9).

"Entende-se por corpus um conjunto de textos utilizados, por um sistema de PLN, para treinamento e teste de modelos estatísticos ou para avaliação de componentes de sistemas de PLN" (SANT'ANNA, 2000, p. 36).

6.6.1 Tipos de Corpus

De acordo com o seu tipo, um corpus pode conter textos de gêneros diferentes ou de mesmo gênero. Do mesmo modo, um corpus pode ser classificado como marcado, caso possua informações sintáticas ou semânticas para cada item lexical, ou não-marcado, caso essas informações não estejam disponíveis (SANT'ANNA, 2000).

Um corpus, de acordo com o uso, pode ser classificado como (SANT'ANNA, 2000):

- Corpus de treinamento: usado nos casos em que o programa extrai informações estatísticas de um conjunto de tamanho adequado de textos; serve para a retirada desses padrões estatísticos;

- Corpus de teste: usado para teste do sistema, após a etapa de treinamento;

- Corpus de avaliação: usado quando componentes de um sistema precisam ser avaliados.

6.6.2 Pré-Requisitos para Formação de um Corpus Computadorizado

Segundo Sardinha (2000) os quatro pré-requisitos para formação de um corpus computadorizado são:

- Em primeiro lugar, o corpus deve ser composto de textos autênticos, em linguagem natural. Desta forma, os textos não podem ter sido produzidos com o propósito de serem alvo de algum tipo de pesquisa lingüística. E não podem ter sido criados em linguagem artificial, tais como linguagem de programação de computadores ou notação matemática.

- Em segundo lugar, quando se fala em autenticidade dos textos, subentendem-se textos escritos por falantes nativos. Tanto assim que, quando este não é o caso, deve-se qualificá-lo, falando-se em corpora 'de aprendizes' (*'learner* corpora').

- O terceiro pré-requisito diz que o conteúdo do corpus deve ser escolhido criteriosamente. Os princípios da escolha dos textos devem seguir, acima de tudo, as condições de naturalidade e autenticidade. Mas devem também obedecer a um conjunto de regras estabelecidas pelos seus criadores de modo que o corpus coletado corresponda às características que se deseja dele. Ou seja, o conteúdo do corpus deve ser selecionado a fim de garantir que o corpus tenha uma certa característica.

- O quarto pré-requisito é considerado como sendo o mais problemático: Representatividade. Tradicionalmente, um corpus é visto como um conjunto representativo de uma variedade lingüística ou mesmo de um idioma. É problemático, pois não se consegue medir o quanto um corpus é representativo em relação a outro.

7 Ferramentas de PLN

Este capítulo visa mostrar algumas ferramentas relacionadas com PLN. Serão apresentadas quatro ferramentas distintas, sendo uma para cada categoria apresentada no capitulo anterior: *Tagger*, *Morphological Analyzer*, *Stemmer*, *Parsers* e *Corpus Tools*.

7.1 *WebJspell*

Tradicionalmente, os analisadores morfológicos contêm um dicionário que serve como base (palavras e suas características morfológicas) e um conjunto de regras de formação de novas palavras. Esta idéia é a base do analisador morfológico *Jspell* (SIMÕES; ALMEIDA, 2002).

O *WebJspell* é um serviço desenvolvido pelo Pólo de Braga[17] da Linguateca[18] com objetivo de tornar o analisador morfológico e corretor ortográfico *Jspell* acessível a um maior número de utilizadores. O *Jspell* foi desenvolvido pelo Projeto Natura[19] (LINGUATECA, 2006).

O *WebJspell* e o *Jspell* estão em constante evolução de forma a aumentar a qualidade. É desenvolvido e mantido por Rui Vilela, José João Almeida e Alberto Simões (LINGUATECA, 2006).

O *Jspell* foi desenvolvido com base no corretor ortográfico *open-source Inspell*. Embora o *Inspell* não seja um analisador morfológico ele já incluía a possibilidade de definição e uso de regras morfológicas elementares. Em vez de se utilizar dicionários de

[17] Pólo de Braga - O Pólo de Braga da Linguateca é um dos 5 pólos da Linguateca. O Pólo de Braga é composto pelo investigador Rui Vilela e o aluno de doutoramento Alberto Simões, orientados por José João Dias de Almeida e Diana Santos. O Pólo está sediado no Departamento de Informática da Universidade do Minho. Nasceu da colaboração entre a Linguateca e o Projeto Natura.

[18] Linguateca – É formalmente um Centro de Recursos distribuído para a Língua Portuguesa. Resumidamente, a missão da Linguateca é promover a existência, discussão, avaliação, distribuição e manutenção de recursos relacionados com o processamento lingüístico do Português.

[19] Projeto Natura - O Projecto Natura contempla processamento de Linguagem Natural (PLN) com especial ênfase no português.

extensão, define-se um conjunto de regras morfológicas (tabela de afixos que permite, a partir de uma palavra, obter várias flexões e derivações) e associa-se a cada palavra o conjunto de regras morfológicas que lhe são aplicáveis. Esta associação permite que um dicionário seja substancialmente menor do que a listagem de todas as palavras dele obtidas e também permite uma maior riqueza no tratamento de palavras desconhecidas (SIMÕES; ALMEIDA, 2002).

A seguir é apresentada a tela inicial (figura 7.1) do WebJspell, onde o usuário tem a opção de escolher qual dicionário quer utilizar (Latim, Português ou Inglês) e qual o nível de detalhe da resposta da ferramenta (Normal ou Resumida) ele quer visualizar. Na parte central da ferramenta encontra-se o local onde deverá ser informada a palavra ou frase que se deseja analisar morfologicamente.

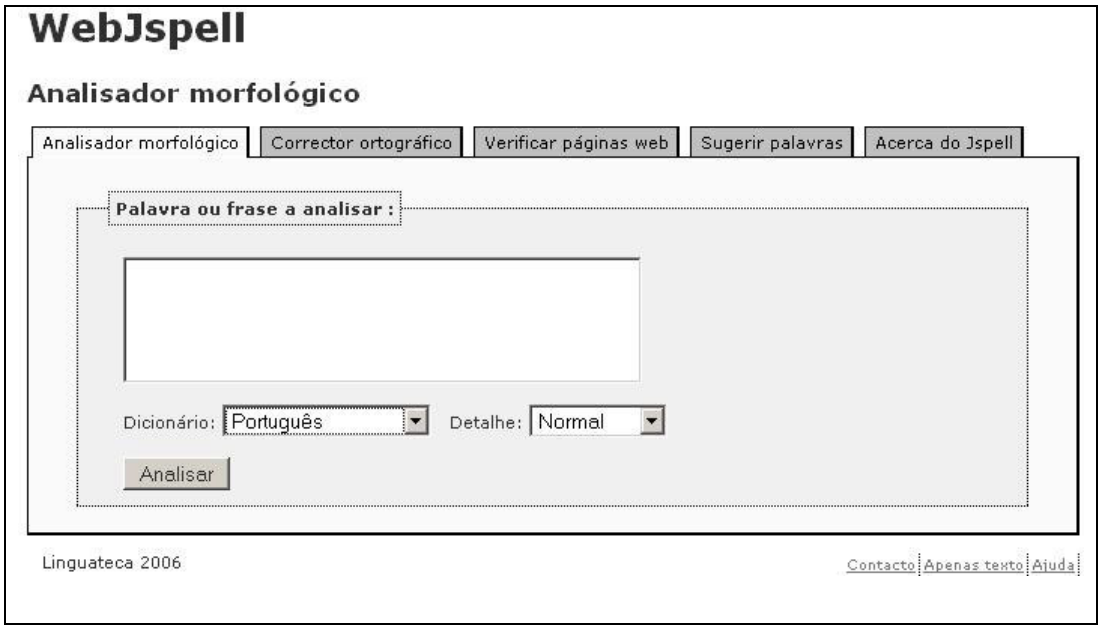

Figura 7.1 – Tela Inicial do WebJspell

A seguir são apresentadas algumas figuras com o resultado da analise morfológica da frase "O Futebol deveria ser a alegria do povo.". Observa-se que o analisador separa cada parte do texto e analisa separadamente.

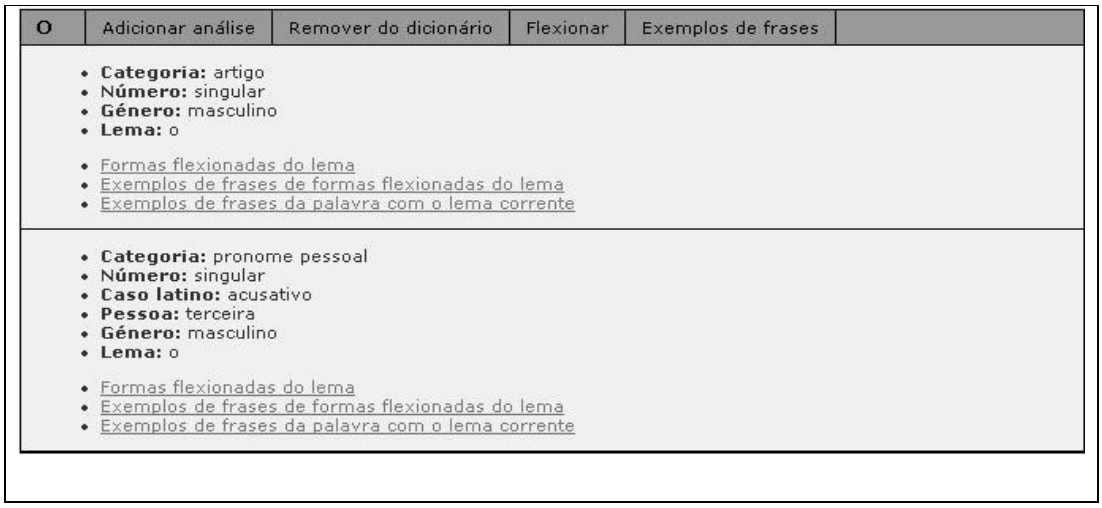

Figura 7.2 – Analise Morfológica de "O"

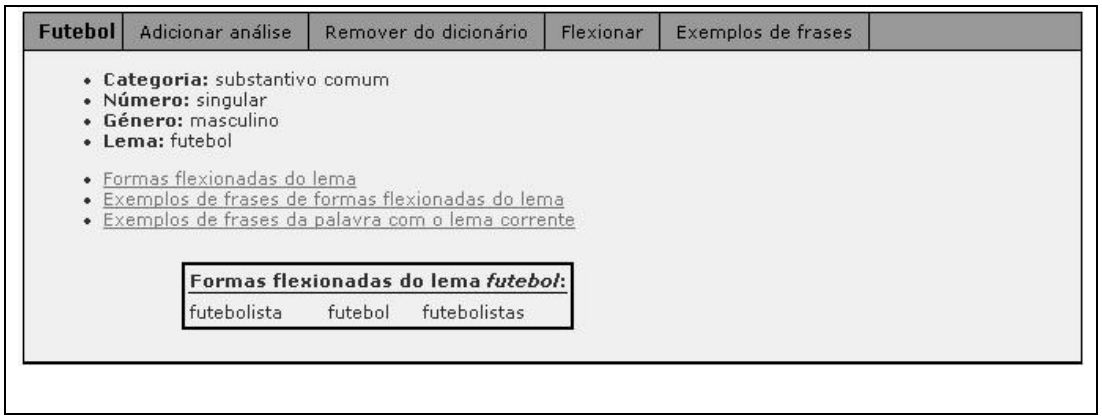

Figura 7.3 – Analise Morfológica de "Futebol"

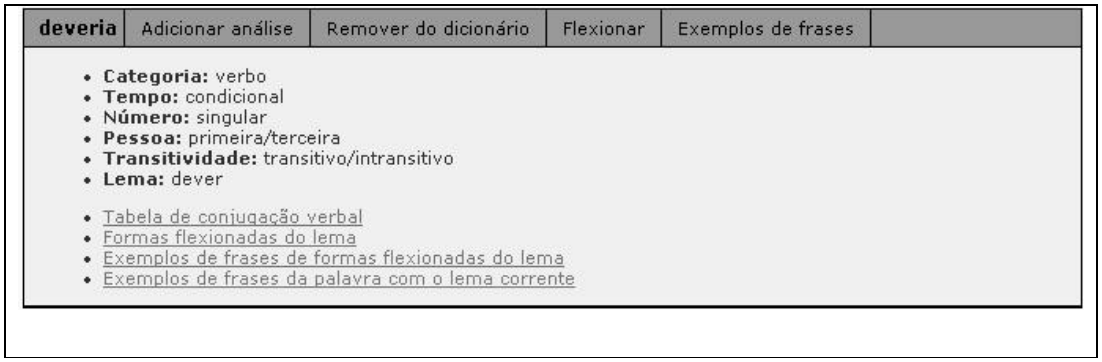

Figura 7.4 – Analise Morfológica de "deveria"

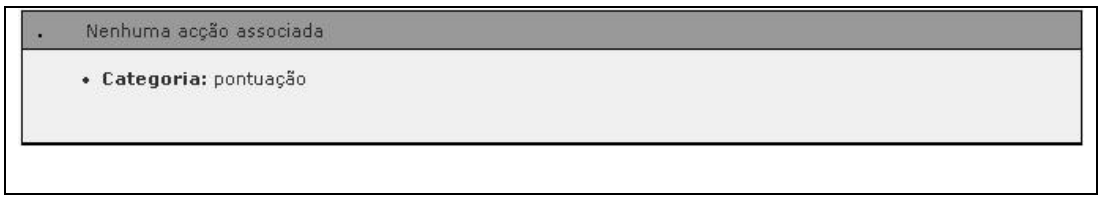

Figura 7.5 – Analise Morfológica de "."

Analisando as figuras apresentadas pode-se verificar que alguns tipos morfológicos tem um tipo de apresentação diferente na ferramenta *WebJspell*. Para a categoria verbo nota-se a presença de um link que aponta para a conjugação verbal da palavra.

Uma das grandes limitações encontradas em qualquer ferramenta de analise morfológica é a limitação do seu dicionário, pois sempre existirão palavras que não constam nos dicionários armazenados nas ferramentas.

7.2 *Snowball*

O *Snowball* (*SNOWBALL*, 2006) é um s*temmer* que possui módulos em várias línguas, sendo uma delas o português. Através de vocabulários armazenados ele processa as palavras informadas e mostra as palavras na sua forma canônica ou radicalizada.

O *Snowball* é baseado em um algoritmo desenvolvido em uma micro-linguagem de mesmo nome. Os algoritmos utilizados para a criação do *Snowball* foram criados por Martin Porter (*SNOWBALL*, 2006).

A figura 7.6 mostra a ferramenta *Snowball*, nesse *demo* somente é possível fazer a radicalização na língua inglesa. A figura 7.7 mostra o resultado obtido através da ferramenta para a palavra "*Several*".

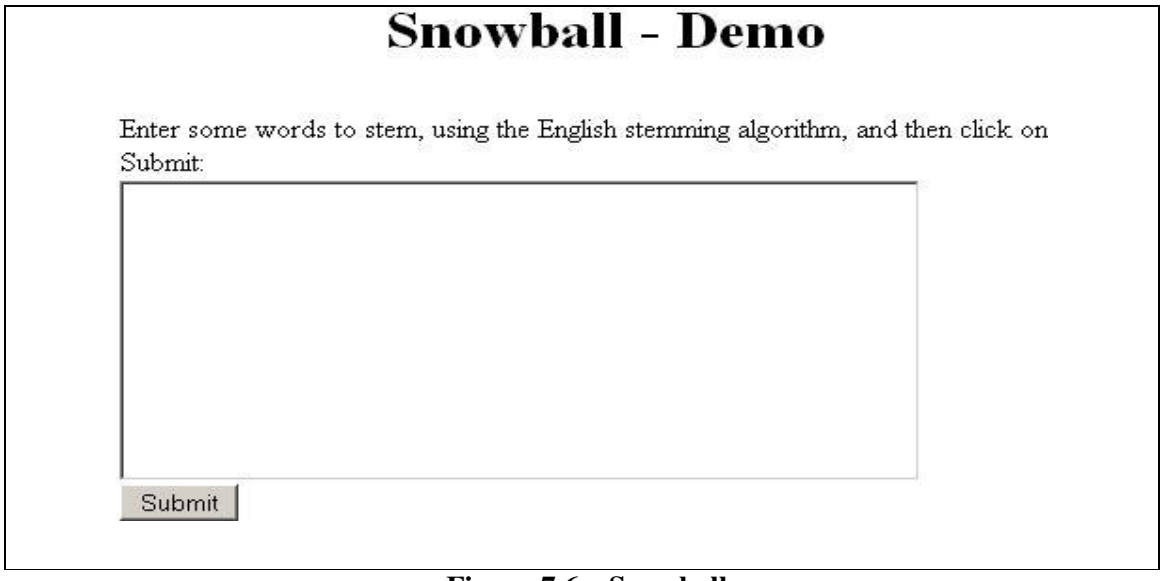

Figura 7.6 – Snowball

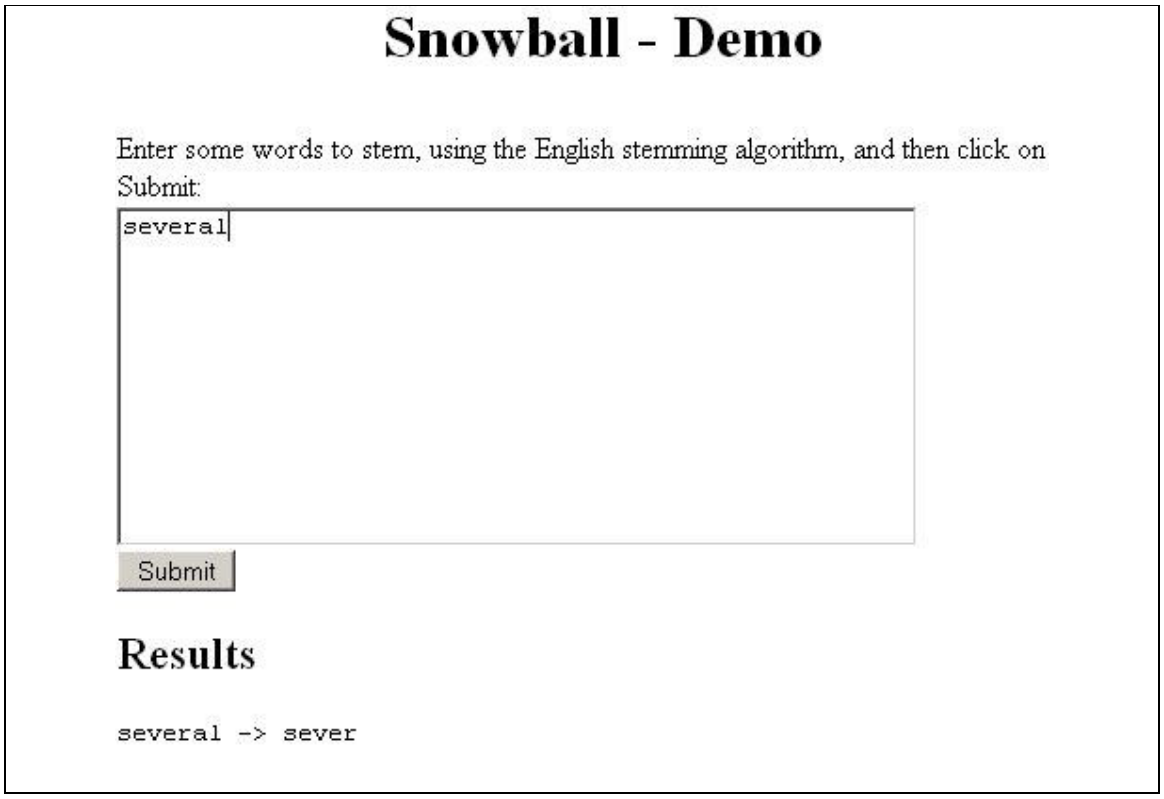

Figura 7.7 – Resultado de Analise do Snowball

Através do resultado pode-se observar que o método utilizado pelo *Snowball* é o *stemming*, onde a palavra é reduzida a seu radical. Observa-se também que a palavra *several* (do inglês vários) não poderia ser radicalizada para *sever* (do inglês cortar algo, romper), este já é um problema encontrado nesta ferramenta,pois a palavra perderia seu significado.

7.3 *FLIP On-Line*

O *Flip on-line* é uma ferramenta na categoria de *parsers* desenvolvida pela empresa Priberam. Segundo *Flip* (2006) a ferramenta foi desenvolvida para facilitar o acesso à tecnologia *Flip*. A Priberam disponibilizou este serviço gratuito *on-line* e também disponibilizou o corretor ortográfico, tanto para português europeu quanto para português do Brasil. A partir da janela disponível no na pagina, o usuário pode digitar as palavras ou as frases (até ao limite de 3000 caracteres) sobre as quais tem dúvidas, selecionar a variedade de português que pretende e visualizar as correções propostas.

A figura a seguir mostra a janela inicial da ferramenta.

Figura 7.8 – *Flip On-Line*

A figura 7.9 mostra o resultado da analise sintática feita sobre uma frase que esta sintaticamente incorreta. Observa-se também que existe a opção de fazer analise sobre a ótica do estilo de escrita, como Formal, Corrente e Informal.

Figura 7.9 – Analise Sintática do *Flip*

O *Flip* apontou um erro sintático nesta frase, o verbo “deveria” deveria estar com a primeira letra maiúscula. Na figura a seguir nota-se que existem dois erros ortográficos que o *parser* identificou.

Figura 7.10 – Analise Sintática do *Flip*

7.4 *JBootCat*

O *JBootCat* (ROBERTS, 2006) é uma ferramenta que se enquadra na categoria de *Corpus Tools*. Esta ferramenta tem uma proposta semelhante com a do corrente trabalho, pois utiliza a web como base de dados para gerar o corpus.

A seguir é apresentada a tela inicial da ferramenta que foi desenvolvida utilizando scripts escritos por Marco Baroni.

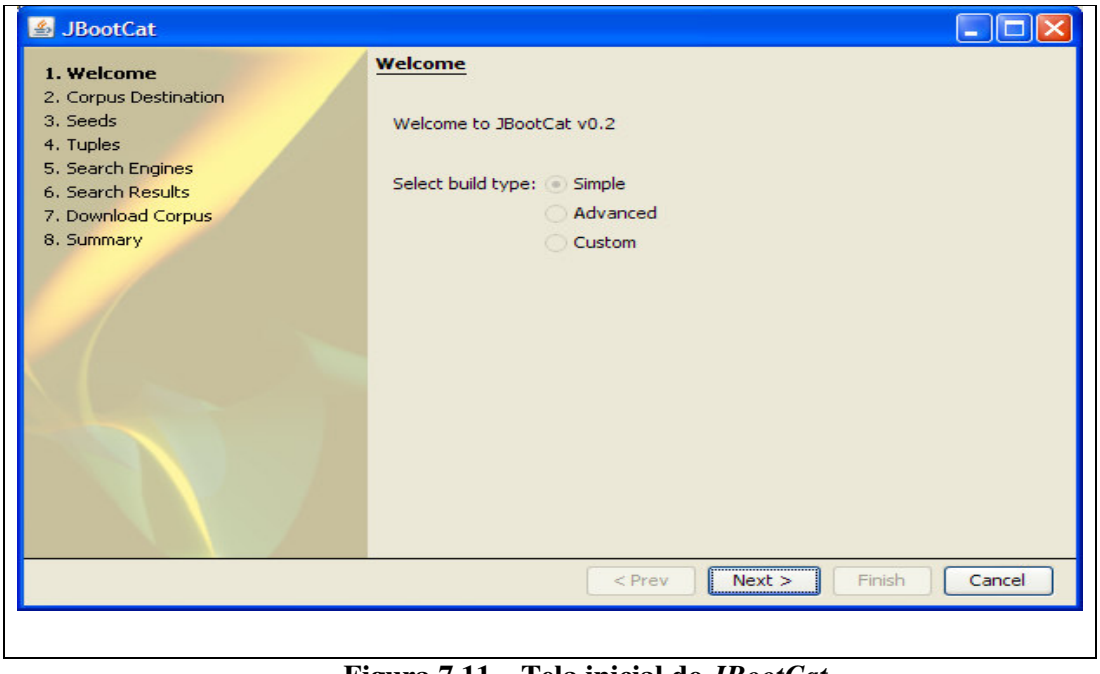

Figura 7.11 – Tela inicial do *JBootCat*

Através desta ferramenta o usuário informa o caminho onde será salvo o novo corpus que será criado pelo usuário (figura 7.12), as palavras que deseja encontrar (figura 7.13), essas palavras ainda poderão ser salvas para uma consulta futura. Após a escolha das palavras o usuário dever gerar as tuplas (poderá escolher qual o número de tuplas e quantas palavras deverão constar em cada uma) para a pesquisa no motor de busca (figura 7.14). O próximo passo é escolher o motor de buscas na web (somente o *Google* por enquanto) e inserir a chave de pesquisa (conforme o capítulo sobre *API'S*) (figura 7.15). No passo seguinte o usuário deverá solicitar que a ferramenta efetue a pesquisa na *web*, e com a pesquisa realizada o mesmo deverá selecionar quais os resultados que deverão constar no corpus (figura 7.16). Na etapa seguinte o usuário deverá solicitar que o sistema faça o *download* do corpus com as informações que ele selecionou (figura 7.17). Na etapa final é apresentado um resumo das ações realizadas pela ferramenta *JBootCat* (figura 7.18).

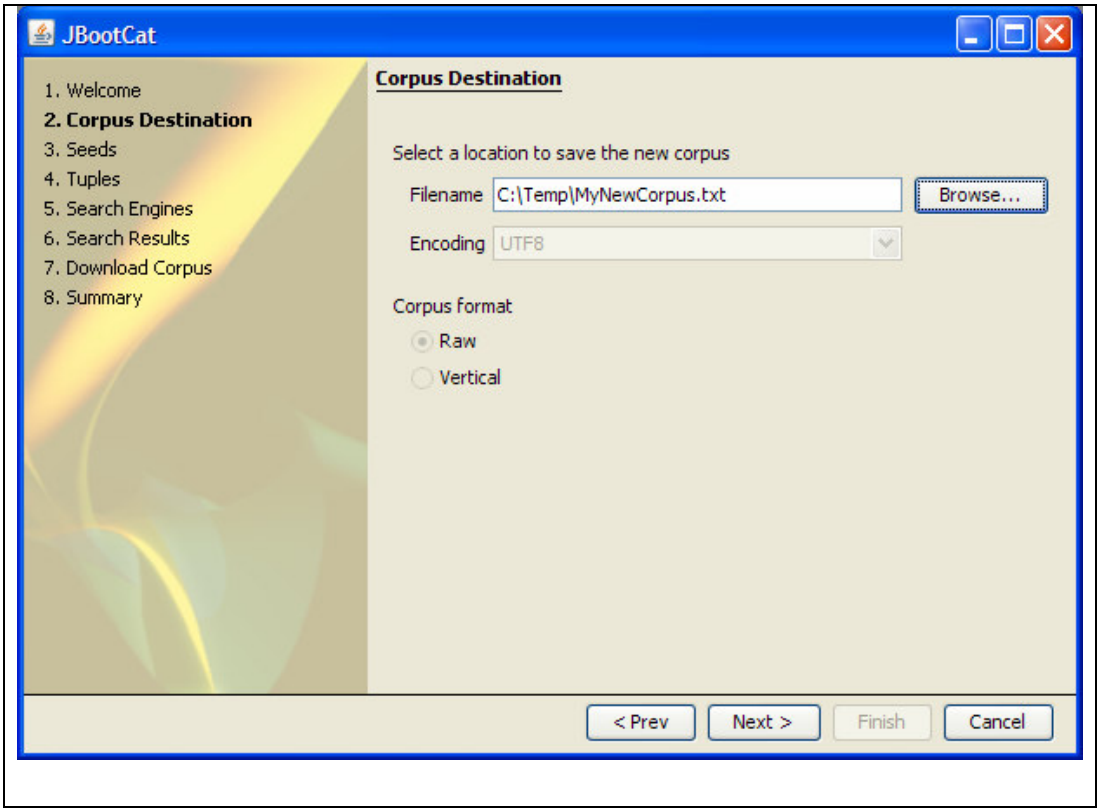

Figura 7.12 – Escolha do local para salvar o novo corpus

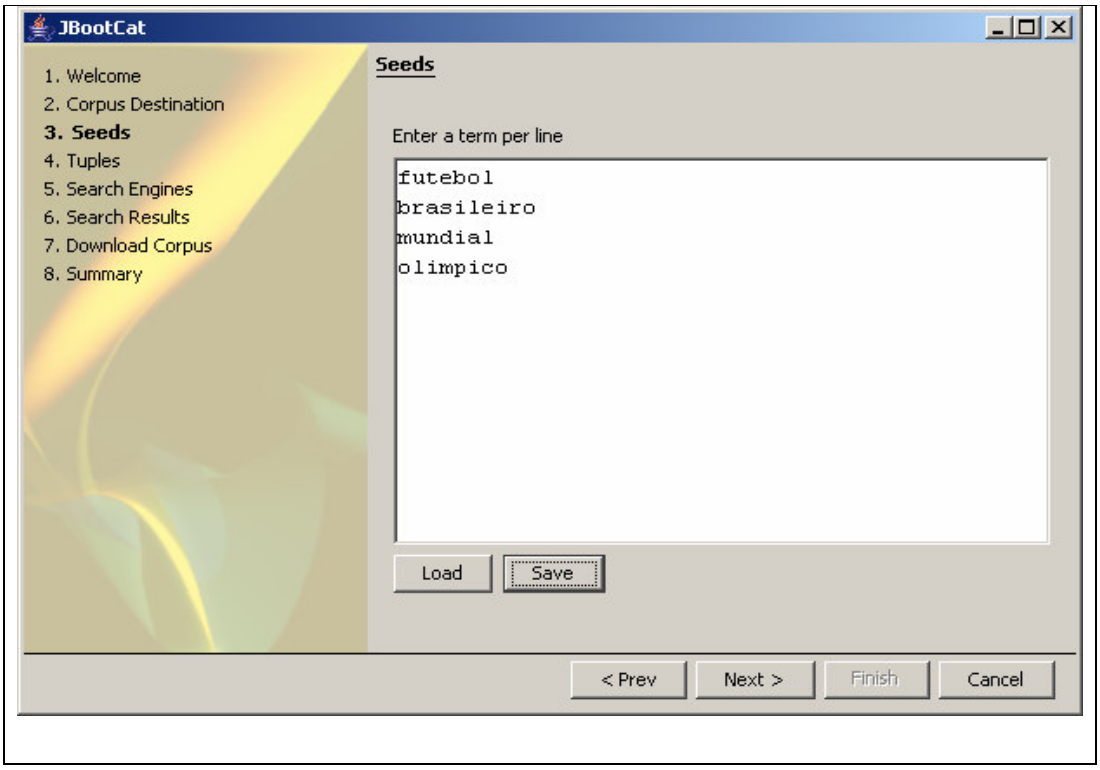

Figura 7.13 – Escolha das palavras

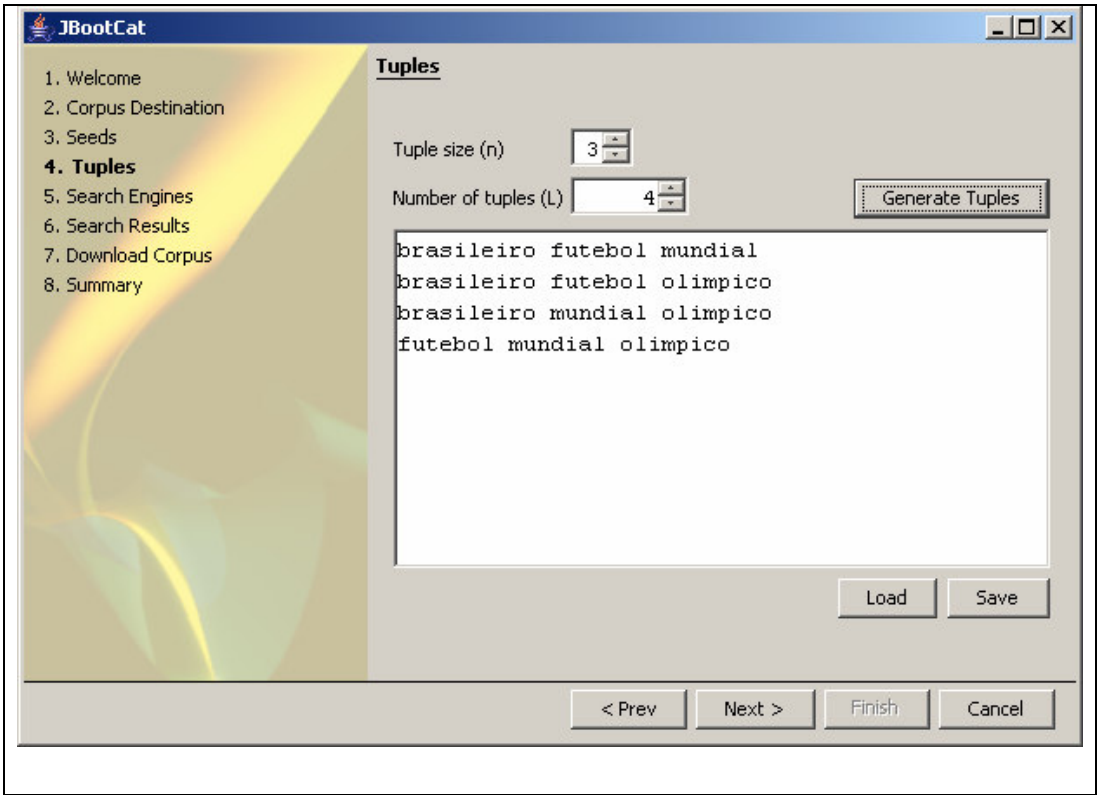

Figura 7.14 – Geração das tuplas para pesquisa

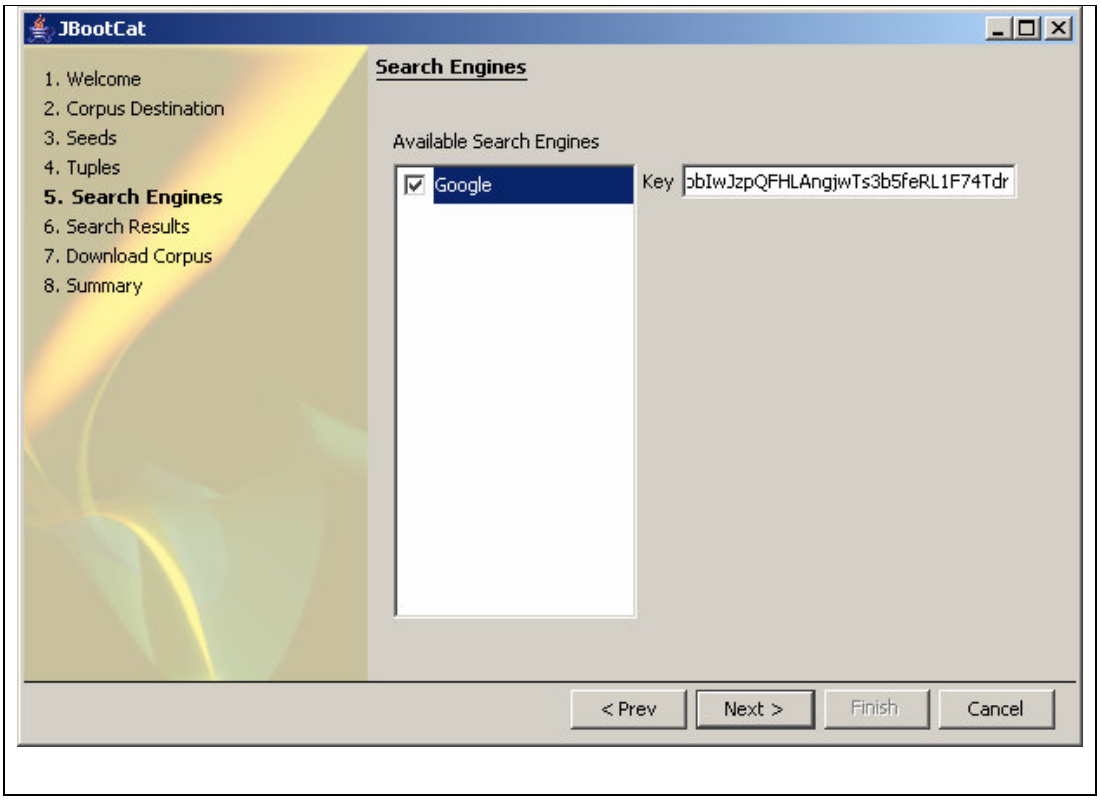

Figura 7.15 – Escolha do motor de busca

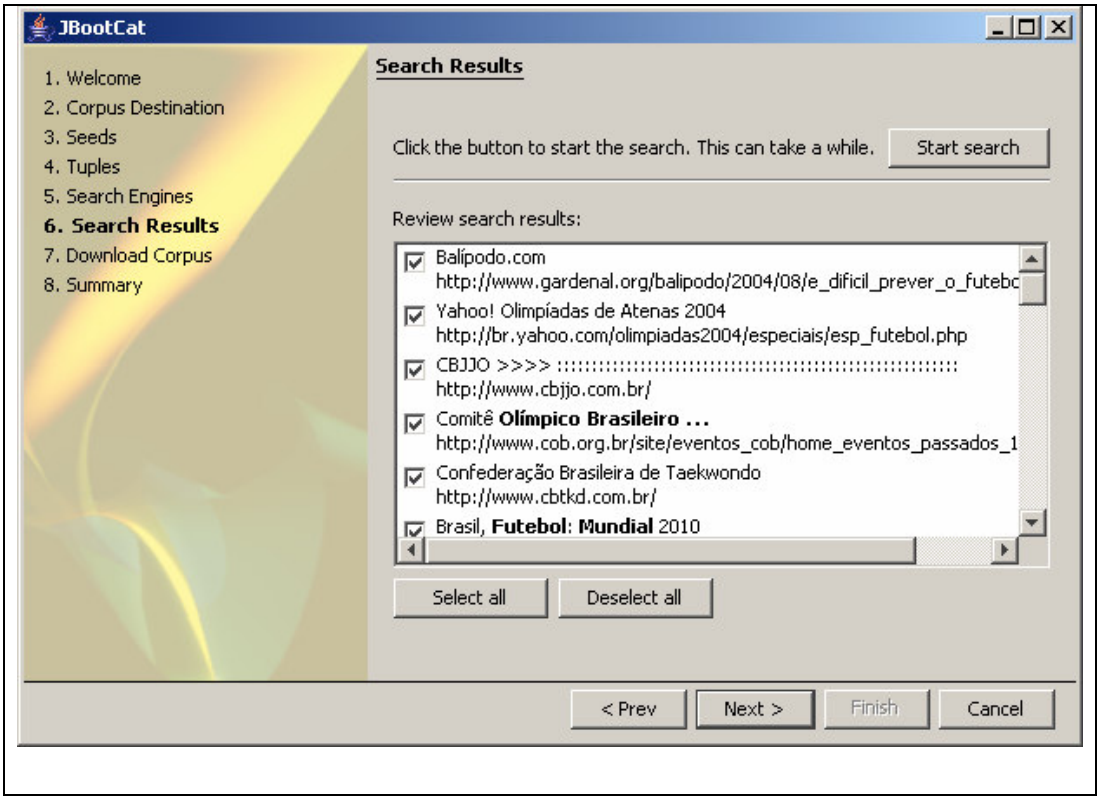

Figura 7.16 – Resultado da pesquisa na web

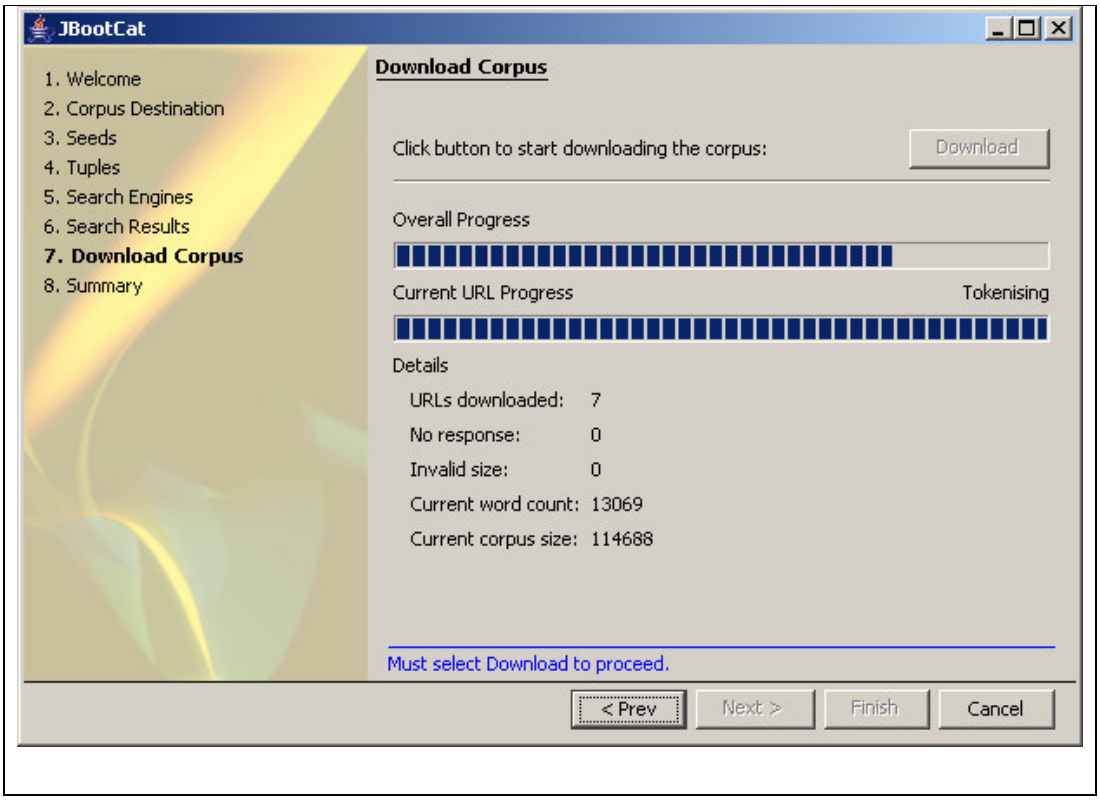

Figura 7.17 – Processamento da pesquisa pela ferramenta

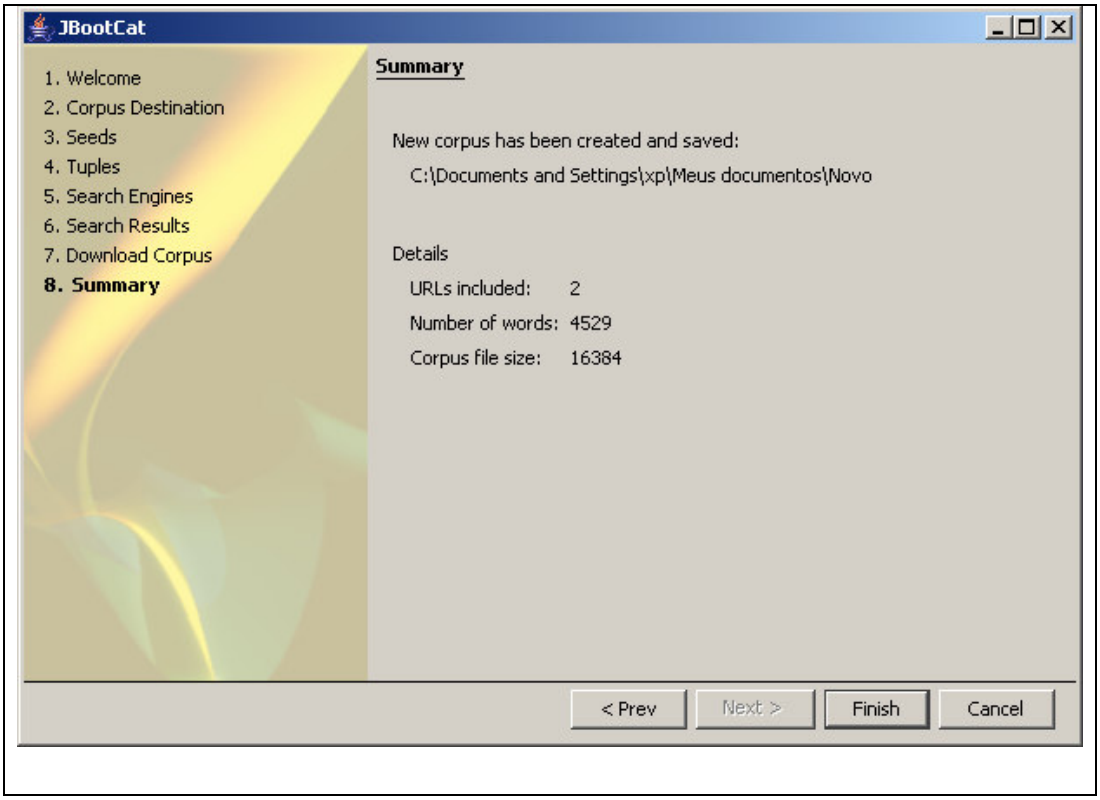

Figura 7.18 – Resumo das ações da ferramenta

8 Proposta para o *Framework*

Nos capítulos anteriores foram estudados conceitos relacionados com orientação a objetos, *UML* e padrões de projetos. Esses padrões são utilizados para dar uma base sólida ao desenvolvimento de *frameworks*. A descrição desses conceitos possibilitou, principalmente, a exposição dos detalhes e das possíveis limitações envolvidas no desenvolvimento de um *framework* orientado a objetos. Também foram estudadas definições a respeito do Processamento da Linguagem Natural (PLN), conceitos relacionados com a sua definição e o seu propósito. Ainda sobre PLN, foram estudadas as suas principais categorias, como Etiquetadores Morfológicos (*Taggers*), Radicalizadores de Palavras (*Stemmers*), Analisadores Sintáticos (*Parsers*) e *Corpus Tools*.

Para unir os conceitos previamente estudados este trabalho propõe a construção de um *Framework* Orientado a Objetos para o desenvolvimento de aplicações para PLN que utilizem a web como corpus. Com esse objetivo, também foram estudadas *API's* de mecanismos de buscas, como *Google*, *Yahoo* e *Technorati*. Estas *API's* foram selecionadas por serem mais comumente citadas na bibliografia e por permitirem que aplicações acessem suas bases de dados e retornem as informações necessárias para a geração do corpus.

O diagrama abaixo (Figura 8.1) demonstra os componentes que estarão envolvidos com o desenvolvimento do projeto:

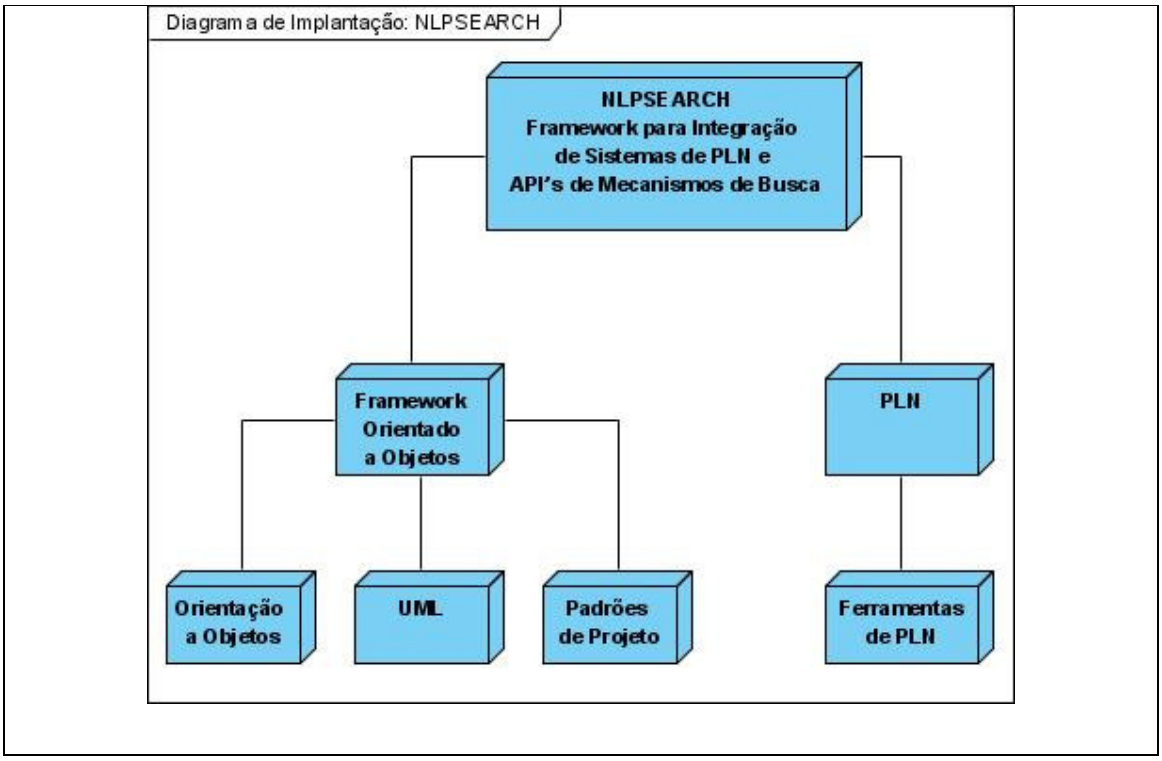

Figura 8.1 – Diagrama de Implantação do *Framework* NLPSearch

CONSIDERAÇÕES FINAIS

O objetivo deste trabalho é a construção de um *framework* para desenvolvimento de aplicações para PLN que utilizem a *web* como corpus. Dessa forma foram apresentados diversos conceitos sobre orientação a objetos, *UML* e padrões de projeto principalmente para dar uma base concreta para o desenvolvimento do *framework* proposto.

O estudo das *API's* dos principais mecanismos de busca na web, como *Google*, *Yahoo* e *Technorati* possibilitou entender como as *API's* disponibilizadas gratuitamente por estes mecanismos podem auxiliar no decorrer do desenvolvimento do trabalho proposto. As três *API's* propostas para estudo geram formulários em *XML* que são disponibilizados como resultado para as pesquisas realizadas, facilitando dessa forma sua utilização, pois a linguagem *XML* esta sendo utilizada largamente hoje em dia pela comunidade de software, tornando a compreensão dos resultados uma tarefa mais simples.

O estudo sobre o Processamento da Linguagem Natural focou-se mais nas categorias das ferramentas que utilizam seus recursos. Procurou-se mostrar quais suas principais funcionalidades, recursos e problemas encontrados. Foi extremamente importante o estudo destas categorias de ferramentas, pois o *framework* proposto terá a função de criar ferramentas que utilizem tais recursos.

Quando se deseja construir um *framework* deve-se ter em mente qual será seu propósito, dessa forma, é importante conhecer exemplos dos softwares que serão desenvolvidos a partir das estruturas desse *framework*. Sendo assim, foram estudadas quatro ferramentas que utilizam recursos de PLN, e estão enquadradas nas categorias apresentadas.

REFERÊNCIAS BIBLIOGRÁFICAS

ABRAHÃO, Paulo Ricardo Carneiro. **Modelagem de Implementação de um Léxico Semântico para o Português.** Porto Alegre: 1997. 118p. Dissertação (Mestrado em Informática) – Instituto de Informática, PUCRS, 1997. Disponível em: <http://www.inf.pucrs.br/~linatural/publicacoes/PRicardoD1.zip>. Acesso em: 20 ago. 2006.

BAPTISTA, Jorge. **Estabelecimento e Formalização de Classes de Nomes Compostos**. Lisboa: 1994. 242p. Dissertação (Mestrado em Lingüística Portuguesa Descritiva) – Faculdade de Letras, Universidade de Lisboa, 1994. Disponível em: <http://w3.ualg.pt/~jbaptis/download/JBaptista1994.htm>. Acesso em: 17 out. 2006.

BEZERRA, Eduardo. **Princípios de Análise e Projeto de Sistemas com UML**. Rio de Janeiro: Elsevier, 2002. 286 p.

BIBER, Douglas; CONRAD, Susan; REPPEN, Randi. **Corpus Linguistics:** Investigating Language Structure and Use (Cambridge Approaches to Linguistics). Cambridge - USA: Cambridge University Press, 1998. 320p.

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML, Guia do Usuário**. Rio de Janeiro: Campus, 2000. 472p.

BRAGA, Rosana Teresinha Vaccare. **Um Processo para Construção e Instanciação de Frameworks Baseados em uma Linguagem de Padrões para um Domínio Específico**. São Paulo: 2003. 44p. Tese (Doutorado na Área de Ciências da Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, USP, 2003. Disponível em: <http://www.teses.usp.br/download.php/teses/disponiveis/55/55134/tde-12052003-102000/publico/TeseFinalCorrigida.pdf>. Acesso em: 17 out. 2006.

BUSCHMAN, Frank et al. **Pattern-Oriented Software Architecture**: A System of Patterns. Chichester – ENG: John Wiley and Sons Ltd, 1996. 466p.

CHAVES, Marcirio Silveira. **Um Estudo e Apreciação sobre Algoritmos de Stemming para a Língua Portuguesa**. In: Jornadas Iberoamericanas de Informática, 4., 2003a, Cartagena de Índias-Colômbia. **Anais...** Cartagena de Índias-Colômbia: 2003, s/p. Disponível em: <http://xldb.di.fc.ul.pt/~mchaves/pg_portugues/public/stemming.pdf>. Acesso em: 17 nov. 2006.

CHAVES, Marcirio Silveira. **Uma Introdução à Lingüística Computacional**. In: Faculdade de Filologia da Universidade da Corunha – Espanha. 2003b, Corunha –ESP. 85p. Disponível em: <http://xldb.di.fc.ul.pt/~mchaves/pg_portugues/master/Apresenta/Intro_LC_4pp.pdf>. Acesso em: 15 nov. 2006.

COELHO, Jorge César B.; COLLOVINI, Sandra; VIEIRA, Renata. **Estudo de Corpus para Classificação de Expressões Anafóricas da Língua Portuguesa**. In: Workshop em Tecnologia da Informação e da Linguagem Humana (TIL 2005), 3., 2005, São Leopoldo. **Anais...** São Leopoldo: SBC, 2005. Disponível em:

<http://www.inf.unisinos.br/~renata/laboratorio/publicacoes/til_scr_05.pdf>. Acesso em: 17 nov. 2006.

COLEMAN, Derek. **Desenvolvimento Orientado a Objetos**: O Método Fusion. Rio de Janeiro: Campus, 1996. 389p.

D'SOUZA, Desmond Francis; WILLS, Alan Cameron. **Objects, Components, and Frameworks with UML**: The Catalysis Approach. . New York - USA: Addison-Wesley Publishing Company, 1998. 785p.

DIAS, Maria Abadia L.; MALHEIROS, Marcelo de Gomensoro. **Automatic Extraction of Keywords for the Portuguese Language**. In: Workshop on Computational Processing of Written and Spoken Language - PROPOR'2006, 7., 2006, Itatiaia. **Anais...** Itatiaia: PROPOR'2006, 2006. p.204-207. Disponível em: <http://ensino.univates.br/~mald/artigo-wsl.pdf>. Acesso em: 17 nov. 2006.

DIAS, Maria Abadia L.; MALHEIROS, Marcelo de Gomensoro. **Estudo de Técnicas de Radicalização para a Língua Portuguesa**. In: Workshop de Engenharia e Tecnologia, 1., 2005, Lageado. **Anais...** Lageado: Univates Editora, 2005. Disponível em: <http://scholar.google.com/url?sa=U&q=http://ensino.univates.br/~mald/artigo-wet.pdf>. Acesso em: 17nov. 2006.

FAYAD, Mohamed E.; JOHNSON, Ralph E. **Domain-Specific Application Frameworks**: Frameworks Experience by Industry. Chichester – ENG: John Wiley and Sons Ltd, 1999. 682p.

FAYAD, Mohamed E.; Schimdt, Douglas C. **Object-Oriented Application Frameworks**: Special Issue on Object-Oriented Application Frameworks. Communications of the ACM, New York - USA, v.40, n.10, p.32-38, Oct. 1997. Disponível em: <http://www.cs.wustl.edu/%7Eschmidt/CACM-frameworks.html>. Acesso em: 6 nov. 2006.

FLIP, **Flip On-line**. Disponível em: <http://www.flip.pt/FLiPOnline/tabid/96/Default.aspx>. Acesso em 15 nov. 2006.

FOWLER, Scott; SCOTT, Kendall. **UML Distilled**: A Brief Guide to the Standard Object Modeling Language. New York, USA: Addison-Wesley Publishing Company, 2000. 185p.

GAMMA, Erich et al. **Padrões de Projeto**: Soluções Reutilizáveis de Software Orientado a Objetos. Porto Alegre: Bookman, 2000. 364p.

GASPERIN, Caroline; GOULART, Rodrigo; VIEIRA, Renata. **Uma Ferramenta para Resolução Automática de Correferência**. In: IV Encontro Nacional de Inteligência Artificial – ENIA. XXII Congresso Nacional da Sociedade Brasileira de Computação. **Anais...** Campinas: SBC, 2003. p.163-172. Disponível em: <http://www.inf.unisinos.br/~renata/laboratorio/publicacoes/art1.pdf>. Acesso em: 14 nov. 2006.

GERBER, Luciano David. **Uma Linguagem de Padrões para Desenvolvimento de Sistemas de Apoio à Decisão Baseado em Frameworks**. Porto Alegre: 1999. 144p. Dissertação (Mestrado em Informática) – Faculdade de Informaica, PUCRS, 1999.

GONZALEZ, Marco Antonio I. **Termos e Relacionamentos em Evidência na Recuperação de Informação**. Porto Alegre: 2005. 182p. Tese (Doutorado em Ciência da Computação) – Programa de Pós-Graduação em Computação, UFRGS, 2005. Disponível em: <http://www.inf.pucrs.br/~gonzalez/tr+/tesemarco.pdf>. Acesso em: 17 nov. 2006.

GOOGLE. **Google Soap Search API (beta)**. Disponível em: <http://www.google.com/apis/reference.html#2_5>. Acesso em: 29 ago. 2006.

GUEDES, Gilleanes T. A. **UML 2**: Guia de Consulta Rápida. São Paulo: Novatec Editora Ltda, 2005. 109p.

HARMON, Paul; WATSON, Mark. **Understanding UML**: the developer's guide: with a Web-based application in Java. San Francisco-USA: Morgan Kaufmann Publishers, Inc, 1998. 367p.

JACKSON, Peter; MOULINIER, Isabelle. **Natural Language Processing for Online Applications**: Text Retrieval, Extraction and Categorization. Amsterdam/Philadelphia: John Benjamins Publishing Company, 2002. 226p.

JOHNSON, Ralph E. **Documenting Frameworks Using Patterns**: Proceedings of the Conference on Object-Oriented Programming Systems Languages and Applications. SIGPLAN Notices, New York-USA, v.27, n.10, p.63-76, oct. 1992. Disponível em: <http://citeseer.ist.psu.edu/johnson92documenting.html>. Acesso em: 7 nov. 2006.

JOHNSON, Ralph E. **Documenting Frameworks**. Framework Digest, New York – USA, v.1, n.13, oct. 1994. Disponível em: <ftp://st.cs.uiuc.edu/pub/FWList/v1n13>. Acesso em: 6 nov. 2006.

JOHNSON, Ralph E. **How frameworks compare to other object-oriented reuse techniques**: Frameworks = (Components + Patterns). Communication of the ACM, New York-USA, v.40, n.10, p.39-42, Oct. 1997. Disponível em: <http://portal.acm.org/citation.cfm?id=141943>. Acesso em: 6 nov. 2006.

JOHNSON, Ralph E.; FOOTE, Brian. **Designing Reusable Classes**. Journal of Object-Oriented Programming, New York-USA, v.1, n.2, p.22-35, Jun./Jul. 1988. Disponível em: <http://www.laputan.org/drc/drc.html>. Acesso em: 6 nov. 2006.

KINOSHITA, Jorge; SALVADOR, Laís do Nascimento; MENEZES, Carlos Eduardo D. **CoGrOO – Um Corretor Gramatical para a língua portuguesa, acoplável ao OpenOffice**. In: Latin American Informatics Conference – CLEI2005, 31., 2005, Cali-Colômbia. **Anais...** Cali-Colômbia: Pontificia Universidad Javeriana-Cali, 2005. Disponível em: <http://cogroo.incubadora.fapesp.br/portal/down/Doc/Artigo_Clei_2005.pdf>. Acesso em: 18 nov. 2006.

LEE, Richard C.; TEPFENHART, William M. **UML e C++**: Guia Prático de Desenvolvimento Orientado a Objetos. São Paulo: Makron Books, 2001. 550p.

LEECH, Geoffrey. **The state of the art in corpus linguistics**: English Corpus Linguistics – Studies in Honour of Jan Svartvik. London-ENG: Longman, 1991.

LEECH, Geoffrey; MACNERY, Anthony. **Corpus Annotation**: Linguistic Information from Computer Text Corpora. London-ENG: Longman, 1997. 281p.

LINGUATECA. **WebJspell**: Questões e respostas relativas ao uso do Jspell. Disponível em: <http://linguateca.di.uminho.pt/jspell/jsolhelp.pl>. Acesso em: 15 nov. 2006.

MATTSSON, Michael. **Object-Oriented Frameworks**: A survey of methodological issues. Ronneby, Suécia: 1996. 122p. Tese (Doutorado) – Department of Computer Science and Business Administration, University College of Karlskrona/Ronneby. Disponível em: <http://citeseer.ist.psu.edu/article/mattsson96objectoriented.html>. Acesso em: 7 nov. 2006.

MELO, Ana Cristina. **Desenvolvendo aplicações com UML 2.0**: Do conceitual à implementação. Rio de Janeiro: Brasport, 2004. 284p.

MENEZES, Carlos Eduardo D. de; NETO, João José. **Um Método Híbrido para a Construção de Etiquetadores Morfológicos, Aplicado à Língua Portuguesa, Baseado em Autômatos Adaptativos**. In: Conferência Ibero-americana em Sistemas, Cibernética e Informática – CISCI, 2002, Orlando-USA. **Anais...** Orlando-USA: CISCI, 2002, p.19-21. Disponível em: <http://www.pcs.usp.br/~lta/artigos/menezes_cisci2002.pdf>. Acesso em: 15 nov. 2006.

NUNES, Maria da Graça V. et al. **A Construção de um Léxico para o Português do Brasil**: Lições Aprendidas e Perspectivas. In: Encontro para o processamento de português escrito e Falado, 2., 1996, Curitiba. **Anais...** Curitiba: CEFET-PR, 1996. p.61-70. Disponível em: <www.icmc.sc.NILC.br/mdgvnune/download/curitilex.ps.gz>. Acesso em: 17 nov. 2006.

NUNES, Maria das Graças V. et al. **Introdução ao Processamento das Línguas Naturais**. Notas Didáticas do ICMC-USP, São Carlos, n.38, jun. 1999.

PARDO, Thiago Alexandre S.; RINO, Lucia Helena M.; NUNES, Maria das Graças V. **NeuralSumm**: Uma abordagem Conexionista para a Sumarização Automática de Textos. In: IV Encontro Nacional de Inteligência Artificial – ENIA. XXII Congresso Nacional da Sociedade Brasileira de Computação. **Anais...** Campinas: SBC, 2003. p.1-10. Disponível em: <http://www.dc.ufscar.br/~lucia/articles/ENIA03-PardoEtAl.pdf>. Acesso em: 14 nov. 2006.

PERCY, Carol E.; MEYER, Charles F.; LANCASHIRE, Ian. **Synchronic Corpus Linguistics**: Papers from the sixteenth International Conference on English Language and Research on Computerized Corpora (ICAME 16). Amsterdam/Atlanta: Rodopi, 1996. 289p.

PRE, Wolfgang. **Design Patterns for Object-Oriented Software Development**. New York-USA: Addison-Wesley Publishing Company, 1995. 268p.

QUATRINI, Terry. **Modelagem Visual com Rational Rose 2000 e UML**. Rio de Janeiro: Editora Ciência Moderna Ltda, 2001. 206p.

RÉ, Reginaldo. **Um Processo para Construção de Frameworks a partir da Engenharia Reversa de Sistemas de Informação Baseados na Web**: Aplicação ao Domínio de Leilões Virtuais. São Paulo: 2002. 132p. Dissertação (Mestrado na Área de Ciências da Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, USP, 2002. Disponível em: <http://www.teses.usp.br/download.php/teses/disponiveis/55/55134/tde-20052003-120738/publico/dissert-f.pdf>. Acesso em: 6 nov. 2006.

REED, Paul R. Jr. **Desenvolvendo Aplicativos com Visual Basic e UML**, São Paulo: Makron Books, 2000. 462p.

ROBERTS, Andrew. **JBootCat**. Disponível em: <http://www.andyroberts.net/software/jbootcat/index.html>. Acesso em: 15 nov. 2006.

ROCHE, Emmanuel; SCHABES, Yves. **Deterministic Part-of-Speech Tagging with Finite State Transducers**. Computational Linguistcs, Cambridge-USA, v.21, n.2, p.227-253, jun. 1995. Disponível em: <citeseer.ist.psu.edu/article/roche95deterministic.html>. Acesso em: 15 nov. 2006.

RUMBAUGH, James et al. **Modelagem e Projetos Baseados em Objetos**. Rio de Janeiro: Elsevier, 1994. 652p.

RUSSEL, Stuart; NORVIG, Peter. **Inteligência Artificial**. Rio de Janeiro: Elsevier, 2004. 1040p.

SANCHEZ, A. **Definicion e historia de los corpus**. In: A. SANCHEZ et al (org.). CUMBRE – Corpus Linguistico de Espanol Contemporaneo. Madrid-ESP: SGEL, 1995. p.7-24.

SANT'ANNA, Victor Martins. **Cálculo de Referências Anafóricas Pronominais Demonstrativas na língua Portuguesa Escrita**. Porto Alegre: 2000. 100p. Dissertação (Mestrado em Ciência da Computação) - Faculdade de Informática, PUCRS, 2000. Disponível em: <www.pucrs.br/uni/poa/info/pos/dissertacoes/arquivos/victor.pdf>. Acesso em: 17 nov. 2006.

SARDINHA, Tony Berber. **Lingüística de Corpus**: Histórico e Problemática. Delta, São Paulo, v.16, n.2, 2000. Disponível em: <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0102-44502000000200005&lng=pt&nrm=iso>. Acesso em: 19 nov. 2006.

SILVA, Luís Felipe S.; CHAPETTA, **Wladmir Araújo. Processamento de Linguagem Natural Aplicada à Inspeção de Documentos de Especificação de Requisitos de Software**. Disponível em: <www.cos.ufrj.br/~ines/courses/cos740/leila/cos740/rel.pdf>. Acesso em: 17 nov. 2006.

SILVA, Ricardo Pereira e. **Suporte ao desenvolvimento e Uso de frameworks e componentes**. Porto Alegre: 2000. 262p. Tese (Doutorado em Ciência da Computação) – Programa de Pós-Graduação em Computação, UFGRS, 2000. Disponível em: <http://www.inf.ufsc.br/~ricardo/download/tese.pdf>. Acesso em: 14 ago. 2006.

SIMÕES, Alberto Manuel; ALMEIDA, José João. **Jspell.pm –Um módulo de Análise Morfológica para Uso em Processamento da Linguagem Natural**. In: Encontro da Associação Portuguesa de Lingüística, 17., 2002, Lisboa-POR. Anais...Lisboa: 2002. p.485-495. Disponível em: <http://alfarrabio.di.uminho.pt/~albie/publications/jspell.pm.pdf>. Acesso em: 15 nov. 2006.

SINCLAIR, John. **Corpus, Concordance, Collocation**. Oxford-USA: Oxford University Press, 1991. 197p.

SNOWBALL. **Snowball**. Disponível em: <http://www.snowball.tartarus.org/>. Acesso em: 15 nov. 2006.

TAGLIASSUCHI, Gustavo. **Mecanismo para Criação de Resumos**: Google Summarizer. Canoas: 2002. 40p. Monografia (Tecnólogo em Processamento de Dados) – Faculdade de Informática, ULBRA, 2002.

TECHNORATI. **Technorati API Libraries**. Disponível em: <http://www.technorati.com/developers/tools/libraries.html>. Acesso em: 29 ago. 2006.

VIEIRA, Renata; LIMA, Vera Lucua S. de. **Lingüística computacional:** Princípios e Aplicações. In: Congresso da Sociedade Brasileira de Computação, 21., 2001, Fortaleza. **Anais...** Fortaleza: SBC, 2001, p.47-88. Disponível em: <www.inf.unisinos.br/~renata/laboratorio/publicacoes/jaia12-vf.pdf>. Acesso em: 15 nov. 2006.

VINHAES, Rêges Faria. **Estudo da Utilização de Técnica de Processamento de Linguagem Natural para Otimização de Tradutores Automáticos**. Rio Verde: 2005. 57p.

Monografia (Bacharelado em Ciência da Computação) – Faculdade de Ciência da Computação, Universidade de Rio Verde, 2005.

YAHOO. **Yahoo! Search Web Services**. Disponível em: <http://developer.yahoo.com/search/>. Acesso em: 29 ago. 2006.

YOURDON, Edward; ARGILA, Carl. **Análise e Projeto Orientados a Objetos:** Estudos de Caso. São Paulo: Makron Books, 1999. 328p.