JULIO CEZAR ZANONI

Um Método Semiautomático de Documentação para Código Fonte Produzido por Pequenas Equipes de Desenvolvimento de Software

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática.

CURITIBA

2012

JULIO CEZAR ZANONI

Um Método Semiautomático de Documentação para Código Fonte Produzido por Pequenas Equipes de Desenvolvimento de Software

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática.

Área de Concentração: *Descoberta de Conhecimento e Aprendizagem de Máquina*

Orientador: Prof. Dr. Emerson Cabrera Paraiso

CURITIBA

2012

Zanoni, Julio C.
Um Método Semiautomático de Documentação para Código Fonte Produzido por Pequenas Equipes de Desenvolvimento de Software. Curitiba, 2012. 68p.

Um Método Semiautomático de Documentação para Código Fonte Produzido por Pequenas Equipes de Desenvolvimento de Software – Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática.

1. documentação de código fonte 2. recuperação de informação 3. pequenas equipes de desenvolvimento de software. I. Pontifícia Universidade Católica do Paraná. Centro de Ciências Exatas e de Tecnologia. Programa de Pós-Graduação em Informática II-t

Aos meus pais, em especial minha mãe (*in memoriam*).

À minha amada companheira.

Aos meus amigos.

Agradecimentos

Agradeço a todos que, de alguma forma, tiveram participação neste trabalho, seja dando opiniões, conselhos ou mesmo fazendo críticas, pois, de alguma forma, acabaram sendo construtivas. Em especial quero agradecer aos meus amigos do Tecpar, Dr. Eng. Milton Pires Ramos, Me. Eng. Geraldo Boz Junior, Me. Eng. Bruno Campagnolo de Paula e Me. Eng. Nilton B. Armstrong Júnior. Ao professor Me. Bacharel Márcio Fuckner pelo apoio fornecendo material para testes do sistema.

Agradecimento especial é dirigido aos meus pais Artivo Zanoni e Maria Barranco Zanoni (*in memorian*), que são meu apoio e minha direção desde a minha concepção. Foram eles que de uma forma amorosa e rígida, quando necessário, fizeram com que eu trilhasse e continue trilhando meu caminho. Eles são e sempre serão meu maior exemplo de vida e aqueles que proporcionaram meu desenvolvimento físico, intelectual e moral.

Especial agradecimento à minha companheira Patrícia R. Silva, pelo amor, paciência, compreensão, apoio e ajuda.

Também agradeço a todos os meus amigos, pela ajuda, apoio e confiança.

Ao meu orientador, Prof. Dr. Eng. Emerson Cabrera Paraiso, pela oportunidade, auxílio e dedicação.

À Cheila, secretária do PPGIa.

Sumário

Agradecimentos ..... **vii**
Sumário ..... **viii**
Lista de Figuras ..... **x**
Lista de Gráficos ..... **xi**
Lista de Tabelas ..... **xii**
Lista de Quadros ..... **xiii**
Lista de Equações ..... **xiv**
Lista de Listagens ..... **xv**
Lista de Abreviaturas ..... **xvi**
Resumo ..... **xvii**
Abstract ..... **xviii**
Capítulo 1 ..... **1**
Introdução **1**
1.1. Hipótese de Trabalho 4
1.2. Objetivos 4
1.3. Contribuições Científicas 5
1.4. Organização do Documento 5
Capítulo 2 ..... **7**
Fundamentação Teórica **7**
2.1. Recuperação de Informação 7
2.2. Casamento de Padrões 10
2.3. Documentação no Desenvolvimento de Software 15
2.4. Pequenas Equipes de Desenvolvimento de Software 17
2.5. Linguagem do Código Fonte: Java 19
2.6. Ferramenta de Documentação: Javadoc 21
2.7. Lucene 25
2.8. Recuperação de Informação em Código Fonte 26
Capítulo 3 ..... **32**
Método Proposto **32**
3.1. Documento de Referência 32
3.2. Método Semiautomático de Documentação de Código Fonte 36
3.2.1. Recuperação de Informação 41
3.2.2. Análise das Informações 50

3.2.3. Atualização das Informações 51
Capítulo 4 53
Experimentos e Resultados 53
4.1. Plano de Testes 53
4.1.1. Hipóteses a Verificar 53
4.1.2. Corpora 54
4.1.3. Métricas de Avaliação 55
4.2. Resultados 58
4.2.1. Capacidade de recuperação das informações dos códigos fonte 58
4.2.2. Relevância das passagens recuperadas com o Lucene 58
4.2.3. Capacidade de geração de comentários 60
4.2.4. Grau de comentários ..... 62~~61~~
4.3. Discussão dos Resultados ..... 62~~61~~
4.3.1. Capacidade de recuperação das informações dos códigos fonte ..... 63~~62~~
4.3.2. Relevância das passagens recuperadas com o Lucene ..... 63~~62~~
4.3.3. Capacidade de geração de comentários ..... 66~~65~~
4.3.4. Grau de comentários ..... 66~~65~~
Capítulo 5 ..... 69~~67~~
Conclusão e Trabalhos Futuros ..... 69~~67~~
Conclusão ..... 69~~67~~
Trabalhos Futuros ..... 70~~68~~
Referências ..... 71~~69~~
Apêndice A ..... 77~~75~~
Lucene ..... 77~~75~~
Apêndice B 95
Resultados da Recuperação de Informação na Documentação de Projeto utilizando o Lucene 95

Lista de Figuras

Figura 1: Trecho de código da biblioteca String, do Java. 20
Figura 2: Documento gerado pelo Javadoc para a classe Stopwords.java, com comentário Javadoc no código fonte. 24
Figura 3: Documento gerado pelo Javadoc para a classe *Stopwords.java*, sem comentário Javadoc no código fonte. 25
Figura 4: Exemplo de pré-processamento do JSearch. 29
Figura 5: Estrutura do documento de referência. 35
Figura 6: Principais elementos do método. 37
Figura 7: Constituição do corpus. 37
Figura 8: Diagrama geral do método. 38
Figura 9: Diagrama do bloco de Recuperação de Informações. 39
Figura 10: Diagrama do bloco de Análise das Informações. 40
Figura 11: Diagrama do bloco de Atualização das Informações. 41
Figura 12: Sistema para recuperação de passagens com o Lucene. 49

Lista de Gráficos

Gráfico 1: Relação entre o número de passagens recuperadas e sua classificação antes e depois do uso do operador AND, para o SE-Telecom. ..... 65~~64~~

Gráfico 2: Comparação entre os números de comentários por projeto. ..... 67~~66~~

Gráfico 3: Comparação do grau de comentários antes e depois do Comente+. ..... 67~~66~~

Lista de Tabelas

Tabela 1: Corpora..... 54
Tabela 2: Total de sentenças recuperadas pelo Lucene na análise do arquivo da documentação dos projetos com o analisador BrazilianAnalyzer.....**Erro! Indicador não definido.**
Tabela 3 - Classificação das passagens do descritivo de projeto do SE-Telecom pelo desenvolvedor 59
Tabela 4: Avaliação das passagens recuperadas pelo Lucene para o projeto SE-Telecom, caso (1). 59
Tabela 5: Avaliação das passagens recuperadas pelo Lucene para o projeto SE-Telecom, usando o operador AND, caso (2). 60
Tabela 6: Valores de precisão, revocação e medida F das passagens recuperadas pelo Lucene para o projeto SE-Telecom 60
Tabela 7: Contagem manual de comentários dos projetos antes e depois de analisados pelo Comente+. 6160
Tabela 8: % de geração de comentários calculado. 6160
Tabela 9: Grau de comentários dos projetos analisados pelo Comente+..... 6261

Lista de Quadros

Quadro 1: Caracteres de escape para expressões regulares. 12
Quadro 2: Classes de caracteres para expressões regulares. 13
Quadro 3: Afirmações de posição para expressões regulares..... 13
Quadro 4: Quantificadores para expressões regulares..... 14
Quadro 5: Agrupadores para expressões regulares..... 15

Lista de Equações

(Equação 1) 56
(Equação 2) 56
(Equação 3) 56
(Equação 4) 57
(Equação 5) 57

Lista de Listagens

Listagem 1: Trecho do código fonte para a classe *Stopwords.java*.....23
Listagem 2: Exemplo de comentários gerados pelo sistema.....50

Lista de Abreviaturas

API	*Application Programming Interface*
CMM	*Capability Maturity Model*
CSCW	*Computer Supported Cooperative Work*
HTML	*Hypertext Markup Language*
IDE	*Integrated Development Environment*
IEEE	*Institute of Electrical and Electronics Engineers*
ISO	*International Organization for Standardization*
ISO/IEC	*International Organization for Standardization /International Electrotechnical Commission*
JVM	*Java Virtual Machine*
RI	Recuperação de Informação
RUP	*Rational Unified Process*
SRI	Sistema de Recuperação de Informação
SQL	*Structured Query Language*
UML	*Unified Modeling Language*
URL	*Uniform Resource Locator*

Resumo

Desenvolvedores de software sempre se deparam com a tarefa de documentação do código fonte. Documentar o desenvolvimento do código, para muitos desenvolvedores, é uma tarefa tida como enfadonha, sendo pouco valorizada. Porém a documentação é importante, principalmente, quando se tratam de grupos de desenvolvedores. Uma documentação atualizada permite que os participantes do grupo tenham maior visibilidade sobre o que foi e está sendo desenvolvido, permitindo também o reuso de código fonte. Esta pesquisa visa conceber, desenvolver e validar um método de documentação semiautomático de código fonte a partir da documentação existente sobre um determinado projeto em desenvolvimento por uma pequena equipe, bem como a atualização desta documentação a partir de informação colhida do código fonte em desenvolvimento. Entende-se, neste trabalho, por documentação de projeto, aqueles documentos ou trechos de documentos que estejam ligados diretamente ao código em construção.

Palavras-Chaves: recuperação de informação, documentação de código fonte, pequenas equipes de desenvolvimento de software.

Abstract

Software developers often face the task of documenting source code. Documenting the development of code for many developers is a task seen as boring and undervalued. However, source code documentation is an important task, especially when dealt by groups of developers. An up-to-date documentation allows the group members to have greater visibility on what has been and is being developed, allowing the reuse of source code. This research aims to design, develop and validate a semi-automatic documentation method of source code from the existing documentation on a particular project being developed by a small team, as well as updating this documentation from information gathered from the source code under development . It is understood as design documentation, those documents or parts of documents that are linked directly to the code under construction.

Keywords: *information retrieval, source code documentation, small teams of software development.*

Capítulo 1

Introdução

Documentar o código fonte de um software que esteja em desenvolvimento ou que tenha sido finalizado é muito importante. Sua importância é devida ao seu valor como ferramenta auxiliar na detecção e correção de problemas, principalmente quando se faz necessário adequar o código fonte de um software com as alterações de projeto ou mesmo para realizar manutenções, corretivas ou preventivas, necessárias ao bom funcionamento do mesmo. Com o apoio de uma documentação de código fonte bem organizada e atualizada, além da facilidade para a localização do ponto no qual se fará a alteração ou manutenção, é de se esperar que ocorra uma redução no tempo gasto para a sua realização (Souza, 2004).

Paduelli e Sanches (2006) referindo-se a sistemas legados, afirmam que tais sistemas "*apresentam dificuldade de manutenção em função de sua complexidade e tamanho, porém essa dificuldade é agravada pela rotatividade de pessoal,* ***documentação insuficiente*** *e extensão das manutenções, muitas vezes realizadas sobre relações desconhecidas ou não triviais sobre os componentes do software*". A partir dessa afirmação de Paduelli e Sanches, podem-se levantar três informações acerca do processo de manutenção de código fonte. A primeira refere-se à questão da rotatividade de pessoal, fato que muitas vezes faz com que a pessoa que irá realizar a manutenção de um determinado código fonte não seja necessariamente aquela que o escreveu e talvez esta pessoa nem tenha participado do projeto que deu origem aos códigos a serem trabalhados. A segunda informação diz respeito à documentação gerada no desenvolvimento e durante as manutenções realizadas, onde os pesquisadores afirmam que uma **documentação insuficiente ou faltante** gera dificuldade na realização dessas manutenções. O terceiro ponto diz respeito à extensão ou tamanho da manutenção a ser realizada e que sem uma documentação atualizada, com teor suficiente para

descrever os códigos, promove aumento no tempo necessário para localizar e corrigir as falhas.

Nesse mesmo trabalho, Paduelli e Sanches relatam a realização de uma pesquisa com diretores, gerentes e supervisores da área de informática de algumas organizações. Nessas organizações existiam sistemas legados instalados e em operação. Através da pesquisa, foram identificados os principais problemas de manutenção enfrentados. Novamente percebe-se a **baixa qualidade da documentação** existente para estes sistemas como um dos problemas de manutenção.

Como exemplo dessa baixa qualidade da documentação, Paduelli e Sanches citam que o código fonte que está sendo escrito ou modificado têm atribuídas pequenas notas de comentário sem muito significado ou somente o número do chamado que resultou na modificação do trecho de código em questão. Quando é utilizado o número de chamado, tem-se como principal problema dele estar se referenciando a outro documento onde foi feito o registro de necessidade de manutenção pelo cliente. Se o detalhamento dessa necessidade de manutenção não foi bem descrito ou esse documento não foi devidamente guardado vindo a se extraviar, não poderá mais ser rastreado e, desta forma, a informação acaba por se perder completamente.

Mesmo diante do fato de existirem vantagens em se manter uma boa documentação de código fonte, percebe-se que para muitos desenvolvedores de software ela é vista como uma tarefa enfadonha e de pouca utilidade prática, exceção feita as grandes organizações nas quais existe a obrigatoriedade de se fazer tal documentação e, por vezes, até mesmo equipes dedicadas na realização dessa tarefa.

Supõe-se que a indisposição por parte dos desenvolvedores em gerar a documentação de código fonte muito provavelmente está relacionada ao fato de que tal tarefa exija a produção de textos em linguagem natural. Produzir estes textos leva o desenvolvedor a deixar de fazer algo que lhe agrada, escrever códigos fonte, para se dedicar a algo que por vezes ele tem dificuldade em fazê-lo. A indisposição pode ser ainda maior quando a documentação a ser trabalhada não é aquela que está relacionada somente aos comentários, escritos diretamente no código fonte, mas aquela usada para a gestão de projeto[1].

[1] Considera-se como documentação de gestão de um projeto de software o conjunto dos seus documentos integrantes, que vão desde os contatos com o cliente, as definições de requisitos, a definição da equipe de trabalho envolvida até aqueles de finalização e entrega do sistema, como por exemplo, o manual do usuário.

Os problemas causados pela documentação incompleta ou ineficiente do desenvolvimento de software podem ter seu impacto amplificado quando levado ao escopo de um projeto colaborativo, ou seja, aquele desenvolvido por várias pessoas simultaneamente.

Nas grandes empresas de desenvolvimento de software, essa realidade pode ser diferente, pois existem pessoas incumbidas unicamente de fazer a documentação de software. Porém, em se tratando de pequenas equipes de desenvolvimento, que normalmente se traduz em pequenas empresas, esta atividade acaba sendo realizada em parte ou em sua totalidade pelos próprios desenvolvedores, sobrecarregando-os (Campagnolo et al., 2009).

Considerou-se como definição para pequena equipe, aquela composta de até 10 participantes, como definido por Pollice et al. (2004). Esse número se adéqua ao cenário de Curitiba, de acordo com informações da Agência Curitiba de Desenvolvimento S/A, através de pesquisa socioeconômica realizada em 2008 e publicada no Guia do Investidor de 2010. Esta pesquisa aponta que o setor de software de Curitiba e região Metropolitana era composto por 847 empresas que geravam 1736 empregos, estes dados levam a uma média de pouco mais de dois empregados por empresa (Curitiba, 2010).

Existe uma área científica interdisciplinar que estuda a forma como o trabalho em grupo pode ser auxiliado por tecnologias de informação e comunicação, de forma a melhorar o desempenho destes grupos na execução das suas tarefas, o trabalho cooperativo auxiliado por computador ou CSCW (*Computer Supported Cooperative Work*) (Moeckel, 2003, Michel, 1999, Jonathan, 1994). O CSCW, aplicado ao desenvolvimento de software, é bastante difundido e vários trabalhos já foram desenvolvidos (Cook e Churcher, 2005, Story et al., 2006, Jiang et. al., 2006 e Sarma et al., 2003). Estes trabalhos, em sua grande maioria, privilegiam dois aspectos: melhorar a infraestrutura de apoio ao desenvolvimento distribuído (através de ambientes integrados ou *groupwares*) e motivar a comunicação dos participantes do projeto colaborativo. Em todos estes trabalhos, o foco é suportar grandes equipes distribuídas de desenvolvimento de software. Os pesquisadores, porém, não têm desenvolvido trabalhos direcionados às pequenas equipes colocadas. Estas possuem requisitos específicos que não têm sido levados em consideração em pesquisas recentes tais como o fato de comporem uma equipe pequena e estarem todos geograficamente próximos.

Logo, de alguma forma é interessante conseguir reduzir o trabalho de documentação atribuído aos desenvolvedores e gestores das pequenas equipes de desenvolvimento, deixando o processo o mais automatizado possível, tendo como foco não somente a documentação no

próprio código fonte (comentários de classes, métodos e atributos), mas também a documentação de gestão do projeto.

Dentro do contexto apresentado, esta pesquisa visa desenvolver um método de documentação semiautomático de código fonte, utilizando informações retiradas a partir da documentação existente sobre um determinado projeto em desenvolvimento. Também está sendo proposta no Capítulo 3, a adoção de um documento, denominado de documento de referência, cuja principal função é agrupar informações voltadas à gestão do projeto de software, sendo uma delas a síntese da documentação de código fonte. Deve-se, desde já, deixar claro que o documento de referência não visa suprimir outros documentos necessários para contratar, descrever e acompanhar o projeto, mas servir como alternativa às metodologias tradicionais de desenvolvimento de software que propõem a criação de uma grande quantidade de documentos para gerir o projeto de software.

1.1. Hipótese de Trabalho

Esta pesquisa busca comprovar que a partir da utilização de técnicas de recuperação de informação, em textos não estruturados (documentação de gestão de projeto) e semiestruturados (código fonte), é viável conceber um método semiautomático que permita auxiliar na criação e atualização da documentação do código fonte. Entendem-se como documentação de código fonte os trechos ou documentos da documentação de gestão de projeto, que estejam ligados diretamente ao código fonte em construção, assim como a documentação escrita no próprio código fonte, na forma de comentários.

1.2. Objetivos

O principal objetivo desta pesquisa é criar um método de documentação semiautomática de código fonte a partir da documentação existente em um projeto de software. O objetivo do método é auxiliar o desenvolvedor indicando elementos de código não documentados ou parcialmente documentados. Também é objetivo do método buscar informações nos próprios elementos de código e na documentação de gestão do projeto para complementar tais comentários.

São objetivos específicos desta pesquisa, visando auxiliar na criação e/ou atualização dessa documentação:

- determinar os elementos relevantes da documentação do código fonte, no paradigma de orientação a objetos, e da documentação de gestão de um projeto de software em andamento. Esses elementos são aqueles que podem sofrer alteração, sem prejuízos ao projeto, e têm influência direta na atualização da documentação;
- especificar e implementar algoritmos capazes de extrair informações relevantes da documentação de gestão e do código fonte.;
- especificar e implementar algoritmos capazes de analisar as informações recuperadas, gerando ou atualizando os comentários dos códigos fonte;
- especificar e implementar algoritmos capazes de inserir, adequadamente, os textos dos comentários gerados ou atualizados nos códigos fonte; e
- conceber a estrutura de um documento de referência para a documentação de gestão no âmbito do desenvolvimento de software em pequenas equipes;

1.3. Contribuições Científicas

As principais contribuições científicas desta pesquisa residem na especificação, desenvolvimento e validação de um método semiautomático de documentação de código fonte. Esta pesquisa visa ser aplicada na documentação de projetos de software, tanto naqueles em desenvolvimento quanto naqueles já finalizados, além disso, trazer a proposta de um documento de referência para ser utilizado por toda a equipe e pelo gestor do projeto durante seu desenvolvimento.

1.4. Organização do Documento

O documento está organizado da seguinte forma:

O Capítulo 2 apresenta alguns tópicos da fundamentação teórica realizada para conceituação e levantamento do estado da arte, visando localizar possíveis trabalhos desenvolvidos com relação aos objetivos desta pesquisa. Ou seja, criação e atualização da documentação de software, além de mostrar outras ferramentas, métodos e conceitos que serão utilizados para desenvolver o método proposto.

O Capítulo 3 descreve como foi desenvolvido o método, mostrando detalhadamente os passos a serem utilizados para alcançar os objetivos descritos e apresenta a proposta do documento de referência.

No Capítulo 4 está descrito como os experimentos foram realizados, mostrando seus resultados e a discussão dos mesmos.

Finalizando, no Capítulo 5, são apresentadas as conclusões e trabalhos futuros.

Capítulo 2

Fundamentação Teórica

O objetivo deste capítulo é conceituar e apresentar algumas técnicas, métodos e padrões que auxiliaram na concepção e no desenvolvimento do método semiautomático de documentação de código fonte. Dentre elas estão a recuperação de informação e o casamento de padrões que foram usados para extrair, trabalhar e filtrar as informações dos códigos fonte e dos arquivos texto da documentação de gestão envolvidos no processo. Também será feito um breve relato sobre a documentação no desenvolvimento de software e serão mostrados alguns projetos que tratam de recuperação de informação aplicada em de códigos fonte.

2.1. Recuperação de Informação

Antes de apresentar a teoria sobre a recuperação de informação (RI), será feita uma breve análise histórica para ver como o homem, há muito tempo, vem armazenando e recuperando informações.

Um grande exemplo de armazenamento de informações é a Biblioteca de Alexandria que, segundo a história, tem sua fundação estimada no século III a.C. e guardava cerca de 400.000, podendo ter chegado a 1.000.000 de papiros. Porém, armazenar toda essa informação não tem nenhuma serventia se não for possível recuperá-la. Para isso, começou-se a desenvolver técnicas para busca e recuperação de informações.

Por vários séculos, essa busca para recuperar informação tem sido feita através da catalogação manual dos documentos que as contém, de onde se extraem termos de indexação que os representem. Esta operação apresenta alto custo, já que para a criação dos termos de indexação existe a necessidade de se conhecer a obra detalhadamente onde a única forma de se conseguir isso é fazendo sua leitura. Um problema que surge neste tipo de indexação é que

cada catalogador, de forma bastante particular, faz a escolha dos termos de indexação que mais lhe convierem para identificar o conteúdo de cada obra.

A definição de RI, com o aparecimento da informática, sofreu algumas modificações e, atualmente, apresenta várias conotações, podendo ser considerada desde uma consulta a um fichário, um simples banco de dados ou um sistema integrado para uma biblioteca ou mesmo a integração de várias bibliotecas, na intenção de localizar os livros armazenados em suas estantes. Ele também pode tratar da análise de um texto completo de um acervo pessoal, hoje disponível em qualquer microcomputador, para retornar quais documentos atendem à necessidade de busca do usuário.

Foi em 1951 que o cientista em computação Calvin Mooers (Sales e Viera, 2007) criou o termo Recuperação de Informação (RI) e dessa forma definiu um novo ramo da ciência da informação, como transcrito a seguir:

"*A RI trata dos aspectos intelectuais da descrição da informação e sua especificação para busca, e também de qualquer sistema, técnicas ou máquinas que são empregadas para realizar esta operação*."

Já Ferneda (2003), compilou outras definições para a RI:

"*Operação pela qual se selecionam documentos, a partir do acervo, em função da demanda do usuário*."

"*Tratamento da informação, catalogação, indexação e classificação*."

Segundo as definições apresentadas por Ferneda (2003), a RI não passa de sistemas de recuperação de textos, documentos ou até mesmo *sites* da Internet que contenham a informação solicitada por um usuário que as está buscando. Diferente da definição de Baeza-Yates e Ribeiro-Neto (1999) a qual diz que a RI "c*onsiste em recuperar informações a respeito de um assunto desejado, e não simplesmente recuperar documentos que satisfaçam sentenças de consulta*". Conforme Mooers, (1951 *apud* Sales e Viera., 2007) e Baeza-Yates e Ribeiro-Neto (1999), sistemas de RI devem realizar a busca da informação desejada, recuperá-la, tratá-la e devolver a informação sem que haja a necessidade do solicitante realizar um segundo tratamento para buscá-la dentro do texto, documento ou página da Internet que lhe foi devolvida como resposta.

Um dos sistemas de RI mais comumente usado em sistemas informatizados é o banco de dados (Lancaster, 1993). Nele tem-se armazenada uma grande quantidade de informações as quais podem ser acessadas através do uso de termos de busca, normalmente chamados de

queries (termo em inglês que significa pesquisa ou consulta), definidos por padrões próprios para cada banco de dados ou através de uma linguagem padronizada, o SQL (*Structured Query Language*).

A RI voltada à busca de informações pode ser feita em textos livres, semiestruturados ou estruturados (Martha, 2005, Silva, 2003). Textos livres são aqueles onde normalmente têm-se as informações na forma discursiva e em linguagem natural, dispostas de maneira desorganizada, ou seja, não seguem um padrão nem ordem específicos definidos por campos predefinidos. Já textos estruturados apresentam um formato padronizado, por exemplo: formulários de cadastro. Os textos semiestruturados são um subgrupo dos estruturados, onde seu formato não apresenta um padrão tão rígido, por exemplo: o código fonte de um software segue um padrão de escrita, porém sua ordem não é tão rígida onde se pode escrever vários métodos dentro de uma única classe.

A chamada RI em documentos completos (*Full-text Information Retrieval*) (Fay, 1971) surgiu da ideia de que a RI pode ser realizada considerando que os termos de indexação são o texto por completo (Dantas, 2002), isso devido ao alto poder de processamento e capacidade de armazenamento dos computadores e servidores atuais, que estão em poder dos usuários e provedores. Como a tecnologia de alguns anos atrás não permitia esse alto desempenho computacional e normalmente os links de comunicação tinham banda restrita, além de serem deficientes, era necessário desenvolver métodos que sobrepujassem essas limitações. Um desses métodos é o uso de alguns termos de indexação para catalogar os documentos que se mostraram ótimos para tal fim. Segundo Dantas (2002), os termos de indexação nada mais é do que palavras que fazem parte do documento e de certa forma o descrevem ou geram uma visão lógica dos mesmos.

Atualmente, o principal objetivo da RI é prover a informação diretamente ao usuário, ou seja, não responder apenas indicando quais documentos contêm em seu interior os termos da expressão de busca fazendo com que o usuário tenha de encontrar, dentro destes documentos a informação pontual que deseja. Conforme explicam Baeza-Yates e Ribeiro-Neto (1999) o processo de recuperação que simplesmente retorna quais são os documentos de um acervo, que contêm os termos de busca, não satisfaz a necessidade de informação do usuário.

Para essa pesquisa, o interesse em recuperar a informação não está relacionado com a busca de documentos, mas em obter a informação para essa busca de um documento para ser usada num método de auxílio à documentação de software. Mais à frente será mostrado um mecanismo para recuperar informações de textos semiestruturados, como é o caso dos códigos fonte.

2.2. Casamento de Padrões

Na seção anterior, foi apresentada a RI como uma forma de realizar a indexação e a busca de arquivos que contenham no seu corpo a informação solicitada. Mudando-se o foco para quando se tem um corpus[2] no qual os arquivos tenham um escopo conhecido e necessita-se buscar as informações internamente a eles.

Para esse tipo de busca, podem-se utilizar mecanismos mais simplificados, principalmente se os arquivos forem estruturados ou semiestruturados. O casamento de padrões (do inglês, *Pattern Matching*) (Dantas, 2002) é um destes mecanismos e nada mais é que um estudo de como formular consultas ou buscas baseadas em padrões definidos, que permitem a recuperação de pedaços de um texto com algumas propriedades. Segundo Dantas (2002) um padrão pode ser definido como um conjunto de características sintáticas que devem ocorrer num segmento do texto. Os segmentos de texto ou de código fonte que satisfazem às especificações do padrão são ditos "casados" com ele. Cada sistema permite a especificação de alguns tipos de padrões, que vão dos mais simples (palavras) até os mais complexos (expressões regulares, por exemplo). Geralmente, quanto mais poderoso é o conjunto de padrões permitidos, mais definidas são as regras de consulta que o usuário pode formular e mais complexa e precisa a realização da busca.

A seguir são mostrados os tipos mais comuns de casamento de padrões:

- **palavras**: é o padrão mais básico, onde é feito o casamento exato do padrão, que é uma palavra, com uma palavra no texto analisado;
- **prefixos**: uma *string* que deve formar o início de uma palavra do texto. Por exemplo, dado o prefixo "comput", todos os documentos contendo palavras tais como "computação", "computador", "computando", etc. serão recuperados;

---

[2] Segundo definição do dicionário Priberam da Língua Portuguesa, corpus é uma coletânea acerca de um mesmo assunto ou um conjunto de documentos que servem de base para a descrição ou o estudo de um assunto.

- **sufixos**: uma *string* que deve formar a terminação de uma palavra do texto. Por exemplo, dado o sufixo "ação", todos os documentos contendo palavras tais como "computação", "motivação", "recuperação", etc. serão recuperados;
- ***substrings***: uma *string* que pode aparecer no corpo de uma palavra do texto. Por exemplo, dada a *substring* "nut", todos os documentos contendo palavras tais como "manutenção", "inutiliza", "desnutrição", etc. serão recuperados. Também se pode recuperar *substrings* em qualquer lugar do texto (e com separadores). Por exemplo, uma busca por "mente poder" vai casar com "infelizmente poderia";
- ***range*** (intervalo): um par de *strings* que casam com qualquer palavra no intervalo entre elas, em uma ordem lexográfica, ou seja, como elas aparecem num dicionário. Por exemplo, palavras entre o intervalo limitado por "pane" e "pano" vão recuperar documentos que possuam as palavras "panela", "pânico", "panificadora", etc.;
- **permitindo erros**: uma palavra com um limite de erro. Este padrão de busca recupera todas as palavras do texto que são similares a uma palavra dada. O conceito de similaridade pode ser definido de várias formas, geralmente quer dizer que o padrão pode conter erros (de datilografia, de ortografia, etc.) e a consulta deve tentar recuperar uma dada palavra e suas variantes errôneas;
- **expressões regulares** (*regular expressions* - ~~regex~~RegEx): alguns SRIs (Sistemas de Recuperação de Informação) permitem busca por expressões regulares. Uma expressão regular é uma boa maneira de construir padrão com *strings* simples associados a alguns operadores, como união, concatenação e repetição; e
- **padrões estendidos**: pode-se usar uma linguagem de consulta mais amigável para representar alguns casos de expressões regulares mais comuns. Padrões estendidos são subconjuntos de expressões regulares (Thompson, 1968) que são expressos numa sintaxe mais simples. Por exemplo, classes de caracteres (uma ou mais posições do padrão podem casar com qualquer caractere pré-definido), expressões condicionais (uma parte do padrão pode ou não aparecer), combinações que permitem casamento exato e partes com erros, dentre outros.

De todos os tipos de casamento de padrões apresentados, o das expressões regulares foi escolhido para ser utilizado na pesquisa. As expressões regulares oferecem um mecanismo flexível e eficiente para processamento de textos. Por meio de uma notação, que oferece uma extensa gama de funções, é possível analisar uma grande massa de dados a procura de

padrões, permitindo a extração, edição e substituição de textos, sendo essencial para aplicações que lidam com processamento de texto (Jones, 2003).

Vários são os caracteres e construtores que podem ser usados na definição de uma expressão regular: caracteres de escape, classes de caracteres, afirmações, quantificadores e agrupadores. que serão mostrados a seguir:

Caracteres de Escape: são caracteres que têm significado especial quando precedidos pelo caractere "\", ou seja, a associação de um caractere com a contra barra "\". Permite que se possa localizar, por exemplo, uma entrada de nova linha dentro de um texto, utilizando-se o caractere de escape \n. No Quadro 1 se pode ver uma lista desses caracteres de escape.

Quadro 1: Caracteres de escape para expressões regulares.

Caractere	Descrição
\a	Mapeia o caractere \u0007 (alarme)
\b	Mapeia um backspace se estiver entre []
\t	Mapeia um tab \u0009
\r	Mapeia um retorno de carro \u000D
\v	Mapeia um tab vertical \u000B
\f	Mapeia uma alimentação de papel (form feed) \u000C
\n	Mapeia uma nova linha \u000A
\e	Mapeia um esc \u001B
\040	Mapeia um caractere ASCII como Ovctal
\x20	Mapeia um caractere ASCII usando representação hexadecimal
\cC	Mapeia um caractere de controle ASCII, por exemplo, \cC é Control-C
\u0020	Mapeia um caractere Unicode usando representação hexadecimal
\	Quando não é seguido por um caractere de escape, mapeia o próprio caracter, por exemplo, *.

Adaptado de Microsoft MSDN (2010)

Classes de Caracteres: no Quadro 2, são mostrados alguns caracteres que, diferente dos caracteres de escape, realizam mapeamentos. Pode-se dizer que realizam algum tipo de função, como por exemplo, se for utilizada a classe de caracteres [A-Z], significa que pode ocorrer qualquer letra no intervalo de ‘a’ a ‘z’, desde que sejam maiúsculas. Pode-se perceber também, pelo Quadro 2, que se existir uma letra minúscula na descrição de uma classe de caracteres, tem-se o inverso usando a mesma classe em letra maiúscula. Além disso, o caractere "^" pode ser usado para negar o conteúdo de um grupo.

Quadro 2: Classes de caracteres para expressões regulares.

Caractere	Descrição
[grupo]	Mapeia qualquer caractere especificado no grupo, por exemplo, [aeiou]
[^grupo]	Mapeia qualquer caractere que não esteja especificado no grupo, ou seja, negação do grupo, por exemplo, [^aeiou]
[primeiro-último]	Mapeia qualquer caractere no intervalo, por exemplo, [A-Z a-z 0-9]
\w	Mapeia qualquer letra ou número, equivalente a [a-z A-z 0-9]
\W	Mapeia qualquer caractere que não seja letra ou número, equivalente a [^a-z A-z 0-9]
\s	Mapeia qualquer caractere que seja espaço em branco, equivalente a [\f \n \r \t \v]
\S	Mapeia qualquer caractere que não seja espaço em branco, equivalente a [^ \f \n \r \t \v]
\d	Mapeia qualquer caractere que seja um dígito, equivalente a [0-9]
\D	Mapeia qualquer caractere que não seja um dígito, equivalente a [^0-9]
.	Mapeia qualquer caractere, com exceção do \n, mas como a opção SingleLine seja usada, mapeia todos os caracteres, sem exceção

Adaptado de Microsoft MSDN (2010)

Afirmações de posição: caracteres que indicam se a validação foi bem sucedida, ou não, dependendo da posição corrente da *string*. Percebe-se no Quadro 3 que o caractere "^", neste caso tem um significado diferente daquele apresentado no Quadro 1, onde ele deve acontecer dentro dos colchetes, indicando negação.

Quadro 3: Afirmações de posição para expressões regulares.

Afirmação	Descrição
^	Indica que a combinação deve ocorrer no início da string ou no início da linha
$	Indica que a combinação deve ocorrer no fim da string, antes de um \n no fim da string, ou no fim da linha
\A	Indica que a combinação deve ocorrer no início da string, ignora a opção Multiline
\Z	Indica que a combinação deve ocorrer no fim da string ou antes de um \n no fim da string, ignora a opção Multiline
\z	Indica que a combinação deve ocorrer no fim da string, ignora a opção Multiline
\G	Indica que a combinação deve ocorrer no ponto onde a combinação anterior terminou. Quando usado com Match.NextMatch(), assegura que todas as combinações são adjacentes
\b	Indica que a combinação deve acontecer numa fronteira entre um \w (alfanumérico) e \W (não alfanumérico). Quando ocorre numa fronteira \w, quer dizer que o primeiro e último caracteres devem ser um \W (não alfanumérico)
\B	Indica que a combinação não ocorre numa fronteira \b

Adaptado de Microsoft MSDN (2010)

Quantificadores: são usados para indicar a quantidade de vezes um padrão deve acontecer. Os quantificadores podem ser aplicados a um caractere, a um grupo ou a uma classe de caracteres. No Quadro 4 , tem-se os quantificadores.

Quadro 4: Quantificadores para expressões regulares.

Quantificador	Descrição
*	Indica zero ou mais combinações. Por exemplo, \w* ou (abc)*, equivalente a {0,}
+	Indica uma ou mais combinações. Pro exemplo, \w+ ou (abc)+, equivalente a {1,}
?	Indica zero ou uma combinação. Por exemplo, \w? ou (abc)?, equivalente a {0,1}
{n}	Indica o número de combinações, por exemplo, [casa]{2}, onde os caracteres "c", "a" e "s" devem aparecer duas vezes
{n,}	Indica que deve acontecer pelo menos n combinações, por exemplo, (abc){2,}
{n,m}	Indica que deve acontecer pelo menos n combinações, não mais do que m, por exemplo, (abc){2,4}
*?	Indica que a primeira combinação deve consumir o menor número de repetições possíveis
+?	Indica menos combinações, tendo pelo menos uma
??	Indica uma ou mais repetições
{n}?	Equivalente a {n}
{n,}?	Indica menos combinações possíveis, pelo menos n
{n,m}?	Indica menos combinações possíveis, entre n e m

Adaptado de Microsoft MSDN (2010)

Agrupadores: são usados para definir subexpressões de uma expressão regular e capturar *substrings* da *string* de entrada. Um exemplo é o mostrado no próprio Quadro 5, onde se define a expressão de um agrupador como *(?<=19)99*, que continua a combinação de um número, no caso o número 99, somente se o número 19 for seu precedente.

Como se pode ver, as expressões regulares apresentam uma grande quantidade de opções, que podem ser combinadas para gerar expressões de busca flexíveis e abrangentes. Outro exemplo simples de expressões regulares pode ser mostrado a seguir.

Dado o padrão de uma expressão regular definida por, *test:OP[123]*.*, e o texto que apresenta uma lista de opções de retorno para o teste de um sistema:

test:OP1 – grau de operação baixo
test:OP2 – grau de operação elevado para o controle 25
test:OP3 – grau de operação irrelevante
test:OP4 – grau de operação baixo para o controle 34

Aplicando a expressão regular sobre a lista de opções, obtém-se como resultado as linhas:

test:OP1 – grau de operação baixo
test:OP2 – grau de operação elevado para o controle 25
test:OP3 – grau de operação irrelevante

Observe que para a última linha de texto, mesmo com o casamento do texto "*test:OP"*, a presença do número 4 invalida o casamento, pois foi especificado que devem existir os números 1, 2 ou 3, desta forma essa linha foi desprezada.

Quadro 5: Agrupadores para expressões regulares.

Agrupador	Descrição
(subexpressão)	Captura a subexpressão e são numeradas automaticamente com base na ordem do caractere "(", iniciando em 1. A primeira captura, o elemento zero, é o texto
(?<nome>subexpressão)	Captura uma subexpressão dentro de um nome ou número de grupo. A string usada para o nome não deve ter pontuação e não pode começar por um número. Pode-se usar apóstrofos no lugar dos caracteres "<" e ">"
(?<nome1 nome2>subexpressão)	(Balanceamento da definição do grupo). Exclui a definição do grupo nome2, definido anteriormente, e armazena no grupo nome1 o intervalo entre o grupo nome2 e o grupo corrente. Como a exclusão da última definição do nome2 revela a definição anterior de nome2, este construtor permite que a pilha de capturas para o grupo nome2 seja usada como contador para manter o rastro dos construtores aninhados, tal como parênteses. Neste construtor, o nome1 é opcional
(?:subexpressão)	Não captura a substring combinada pela subexpressão
(?imnsx imnsx:subexpressão)	Aplica ou desabilita opções específicas na subexpressão. Por exemplo, (?i-s:) liga o *case insensitive* e desabilita o nome de única linha
(?=subexpressão)	Continua a combinação somente se a subexpressão combina na posição da direita. Por exemplo, \w(?=\d) combina uma palavra seguida por um dígito
(?!subexpressão)	Continua a combinação somente se a subexpressão não combina na posição da direita. Por exemplo, \b(?!un)\w+\b combina palavras que não começam com "un"
(?<=subexpressão)	Continua a combinação somente se a subexpressão combina na posição da esquerda. Por exemplo, (?<=19)99 combina instâncias de 99 precedidas por 19
(?<!subexpressão)	Continua a combinação somente se a subexpressão não combina na posição da esquerda
(?>subexpressão)	A subexpressão é combinada por completo somente uma vez, e depois não participa do retorno de trilha gradativo, ou seja, a subexpressão combina somente strings que combinariam com a subexpressão sozinha. Por padrão, se a combinação não for bem sucedida, o retorno da trilha procura por outras combinações possíveis

Adaptado de Microsoft MSDN (2010)

Na subseção seguinte é apresentada a documentação criada num desenvolvimento de software, para se compreender quais artefatos podem ser utilizados para se fazer as buscas de informações.

2.3. Documentação no Desenvolvimento de Software

Pode-se considerar a existência de dois tipos de documentação para projetos de software: a documentação de gestão e a documentação de usuário. A documentação de gestão é aquela na qual se encontram todas as informações referentes ao projeto em

desenvolvimento, desde o contato com os clientes, o contrato de trabalho, a documentação técnica do software, entre outras e que são usadas para acompanhamento e gerenciamento do projeto. Dentro da documentação técnica, existem dicionários e modelos de dados, fluxogramas de processos, regras de negócios, requisitos de software, dicionários de funções e documentação de código fonte. Já a documentação de usuário, como o nome diz, é destinada ao usuário final, normalmente são manuais que mostram e descrevem como utilizar e as funcionalidades disponíveis no sistema que foi produzido.

Algumas ferramentas de desenvolvimento de software como, UML (*Unified Modeling Language*), RUP (*Rational Unified Process*), estruturada, etc. (Shiki et al., 2004; Massoni et al.; 2003, OMG, 2010) estão apoiadas sobre diversos documentos ou artefatos[3], utilizados nas diversas etapas do processo, como forma de registro do desenvolvimento. Estes documentos vão sendo elaborados e aperfeiçoados desde a especificação do problema, passando pela análise, projeto, desenvolvimento, até a entrega do produto ao cliente. Garantir o perfeito preenchimento e completude destes artefatos é de responsabilidade de uma ou várias pessoas envolvidas na gestão do projeto. A complexidade pode aumentar se o projeto precisar estar em conformidade com alguma norma, como, por exemplo, a ISO 9001 (*International Organization for Standardization*) ou algum modelo de maturidade, como o CMM (*Capability Maturity Model*) (Laryd e Orci, 2000). Estar em conformidade com as normas não significa que o grupo deva ser certificado, mas objetiva garantir a qualidade do produto (software) através da definição e normatização de processos de desenvolvimento. Apesar desses modelos aplicados na garantia da qualidade de software atuarem principalmente no processo, o principal objetivo é garantir um produto final que satisfaça às expectativas do cliente, dentro daquilo que foi acordado inicialmente.

Todas as exigências impostas pelas normas ou modelos citados acima, têm impacto significativo em pequenas equipes, pois exigem maior quantidade de documentos além das exigências de qualidade dessa documentação, permitindo a rastreabilidade do processo. Existem alguns pesquisadores preocupados com isto e propondo alternativas: Pollice et al. (2004), Land e Walz (2006) e Campagnolo et al. (2009). Uma das alternativas é a apresentada por Land e Walz (2006) que apresentam um modelo para documentação de projeto de software para pequenas equipes de desenvolvimento, o *Small Software Project Management*

[3] (*Informática*) produto de uma ou mais atividades dentro do contexto do desenvolvimento de um software ou sistema.

Plan (Plano de Gestão de Pequenos Projetos de Software). O livro tem como principal objetivo fazer um comparativo com as várias normas de qualidade, que dizem respeito à engenharia de software. Por exemplo: a ISO 9001, ISO 90003, IEEE 12207 e ISO/IEC 15504 (*International Organization for Standardization /International Electrotechnical Commission*) e, a partir desta comparação, delineia alguns requisitos que, posteriormente foram organizados dando origem ao *Small Software Project Management Plan*, definido como um documento de referência para documentação de projetos. Este documento leva em consideração as necessidades de definição, documentação e melhoria de produtos de software, para utilização pelas empresas com pequenas equipes de desenvolvimento de software. Para Land e Walz (2006), pequenas equipes de desenvolvimento são constituídas de até 20 (vinte) participantes, diferente de Pollice et al. (2004) e Campagnolo et al. (2009) que limitam este número a 10 (dez) participantes.

Na introdução, comentou-se a respeito de pequenas equipes de desenvolvimento, a seguir tem-se uma breve descrição do que é e quais suas dificuldades e necessidades.

2.4. Pequenas Equipes de Desenvolvimento de Software

A atividade de desenvolvimento de software é complexa, composta por várias etapas e naturalmente suportada por ferramentas especialmente projetadas para tal fim. Em geral, um software é produzido de forma colaborativa, com a participação de vários especialistas (gerente, analistas, programadores, entre outros).

Pequenas equipes de desenvolvimento de software possuem características específicas que devem ser levadas em consideração durante o projeto do método de documentação semiautomática que buscamos, sendo as principais descritas sucintamente a seguir (Campagnolo et al., 2009):

- **comunicação**: como, em geral, os participantes trabalham em um mesmo ambiente físico, ferramentas de comunicação são menos importantes dado que a comunicação face a face é mais fácil e rápida. Assim, textos gerados a partir da troca de mensagens eletrônicas entre participantes do projeto, não serão utilizados nesta pesquisa;
- **atividades dos participantes**: os participantes do projeto são alocados em atividades específicas, porém recebem atividades diferentes em momentos distintos do ciclo de vida do desenvolvimento do software, ou seja, na prática um participante desempenha

diferentes papéis no projeto independente da função. Por exemplo, um programador pode desempenhar o papel de analista e testador;

- **sobrecarga**: os participantes tendem a se sobrecarregar de atividades rapidamente, deixando em segundo plano atividades importantes como a documentação do processo decisório;
- **gerenciamento de código fonte**: a gestão do código fonte é importante e deve ser privilegiada, tanto no nível de desenvolvimento como no de gestão;
- **reuso e manutenção**: o reuso e manutenção de software é fundamental, sendo, portanto, de vital importância dar suporte a tarefa de documentação do código e do projeto;
- **integração de ferramentas**: os participantes utilizam diversas ferramentas disponíveis na Web, na maior parte das vezes gratuitas, e gastam tempo em gerenciar este uso, por exemplo, para encontrar documentação ou mesmo para fazer *login* e executar vários cliques no aplicativo até efetuar a operação desejada.

O método, apresentado nesta seção, visa atuar sobre algumas das características mencionadas anteriormente. Uma delas e talvez a mais importante, e que remete a um dos problemas apontados na pesquisa, é a sobrecarga de atividades dos participantes. O método tem, a princípio, como principal consequência, a diminuição desta sobrecarga de trabalho, permitindo que o desenvolvedor possa atuar em outras tarefas do projeto e que os prazos para o desenvolvimento sejam cumpridos.

Em segundo plano, porém não menos importantes são: o gerenciamento, o reuso e a manutenção do código fonte. Espera-se que através do método exista maior interação dos desenvolvedores com a documentação, possibilitando que eles tenham mais informação sobre o que está sendo desenvolvido e, desta forma, possa ser feito o reuso de software.

A linguagem Java foi adotada como padrão para o código fonte a ser documentado nos projetos alvo da pesquisa. A escolha desta linguagem de programação deve-se à sua grande difusão, o que torna maior a abrangência do sistema a ser desenvolvido. A seguir é feita uma breve descrição de como essa linguagem surgiu e quais características sintáticas são importantes, identificando elementos relevantes da documentação do código fonte conforme os objetivos definidos para a pesquisa.

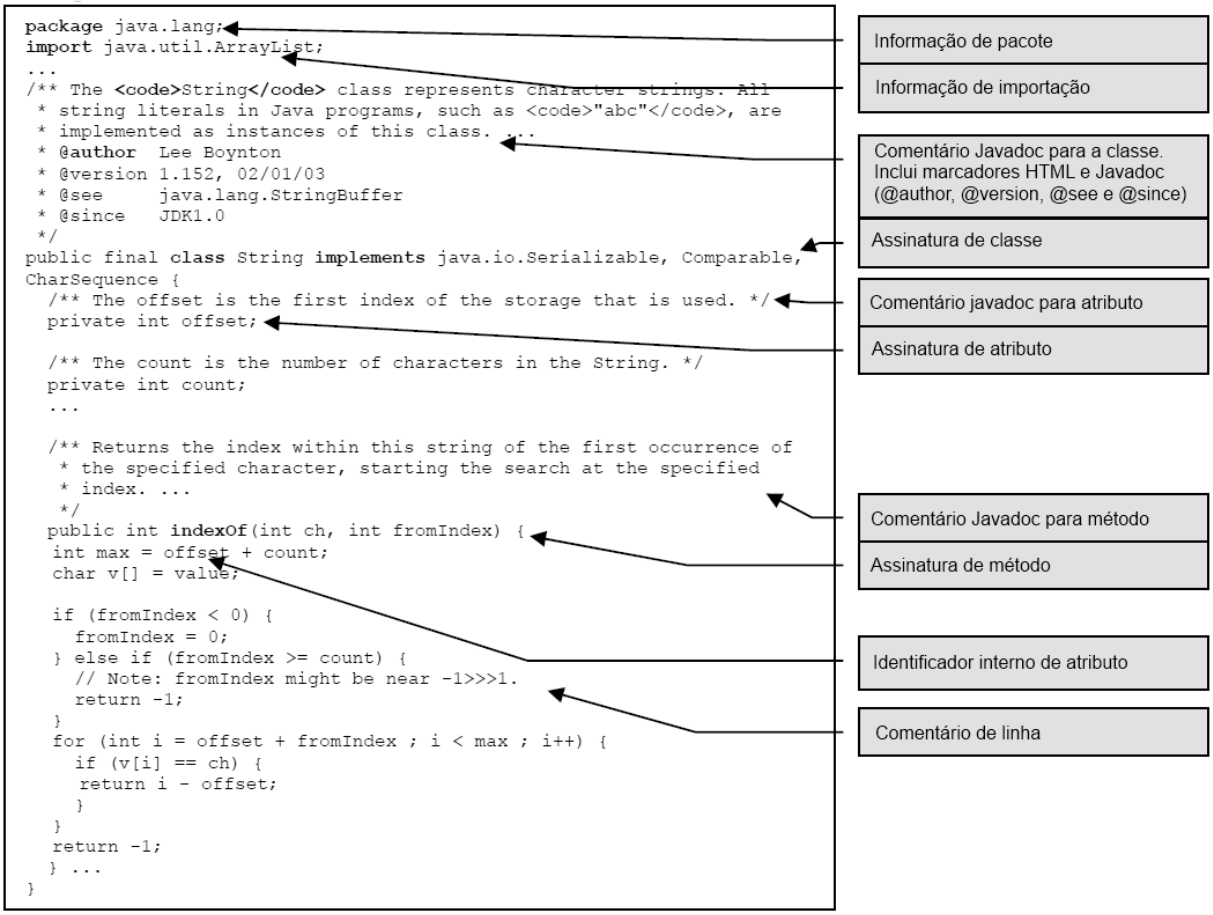
2.5. Linguagem do Código Fonte: Java

O Java surgiu na década de 90 por intermédio da Sun Microsystems que iniciou em 1991 um projeto chamado *Green Project* (Java.net, 2010). É uma linguagem compilada para um código intermediário, chamado *bytecode*. Esse *bytecode* é executado por meio de uma máquina virtual, no caso o JVM (*Java Virtual Machine*), que é um software que executa programas como um computador real. O uso da JVM permite que os programas escritos em Java possam funcionar em qualquer plataforma de hardware e software, que possua uma JVM implementada para ela, tornando essas aplicações independentes da plataforma.

Quanto à linguagem de programação, na Figura 1 é apresentado um exemplo de um bloco de código que descreve uma classe em Java, chamada String.java. Esta classe é um dos componentes da biblioteca padrão da linguagem Java. Serão utilizados códigos fonte que utilizam o paradigma de orientação a objetos, que apresentam o formato do código como o mostrado na Figura 1.

Rech (2005) em seu artigo explica que os nomes e identificadores de classes e métodos utilizam a notação *CamelCase* (Sun Microsystems, 1999). O *CamelCase* é a denominação, em inglês, para uma notação quando se escrevem palavras compostas ou frases e cada palavra é iniciada com maiúsculas de maneira que estejam unidas, ou seja, escritas sem a colocação de espaços entre elas. A primeira palavra de um *CamelCase*, pode ser iniciada em minúscula. Por exemplo: *indexOf* que em texto corrido seria *index of* (*índice de*, em português).

Outra característica dessa linguagem de programação, como se pode ver na Figura 1, é que para algumas das linhas do código, estas foram identificadas como assinaturas. Assinaturas são especificações ou padronizações que definem as algumas das estruturas de um código fonte, como é o caso das classes, métodos ou atributos no Java. Por exemplo, analisando a assinatura da classe ***String*** da Figura 1 observa-se que: "***String***" é o nome ou identificador da classe, "***public***" representa sua visibilidade e *"**implements***" diz que as interfaces *java.io.Serializable*, *Comparable* e *CharSequence* serão utilizadas (Rech, 2005).



Adaptado de (Rech, 2005)

Figura 1: Trecho de código da biblioteca String, do Java.

Percebe-se que no trecho de código mostrado na Figura 1, existem algumas características ou padrões, cuja informação pode ser recuperada com a utilização de técnicas de casamento de padrões. É o caso das assinaturas que fornecem informações que podem indicar quais as funcionalidades desse código fonte.

Um dos objetivos específicos, apresentados na subseção 1.2, é a determinação dos elementos relevantes dos códigos fonte. Neste ponto, já se pode determinar esses elementos, desta forma, definem-se como elementos relevantes as assinaturas de classes e de métodos e as definições de variáveis. Estes são os elementos principais de um código fonte. Uma classe é uma implementação de um tipo de objeto, especifica uma estrutura de dados e os métodos operacionais permissíveis que se aplicam a cada um de seus objetos. Os métodos especificam a maneira pela qual os dados são manipulados e os passos pelos quais uma operação deve ser executada. Eles são a ação que um objeto ou uma classe podem desempenhar, sendo similares às funções e procedimentos (*procedures*) do universo da programação estruturada. A partir

deles é que se determina como o sistema, ou software, irá se comportar, logo, a partir destes elementos é que serão obtidas as informações que permitirão ao método realizar todas as demais operações.

Para documentar as APIs (*Application Programming Interface*) desenvolvidas para o Java, a partir do seu código fonte, a Sun Microsystems criou um gerador de documentação, que é constituído basicamente por algumas etiquetas (*tags*), muito simples, inseridas nos comentários do código fonte, este gerador de documentação é chamado de Javadoc.

2.6. Ferramenta de Documentação: Javadoc

A ferramenta Javadoc analisa as declarações e os comentários em um conjunto de arquivos de código fonte, escritos na linguagem Java, e produz um conjunto de páginas HTML (*HyperText Markup Language*) descrevendo, por padrão, as classes públicas e privadas, classes aninhadas, interfaces, construtores, métodos e campos. Podendo ser utilizado para gerar a documentação de uma API ou a documentação da implementação de um conjunto de arquivos de código fonte.

Para realizar o processamento de arquivos de código fonte a ferramenta Javadoc processa arquivos que terminam com a extensão ".java", a menos que outros nomes de arquivos lhe tenham sido definidos, ou seja, não está preso ao tipo de extensão do arquivo a ser analisado.

Se a ferramenta for executada explicitamente, quer dizer, passando nomes de arquivos, podem-se determinar exatamente quais arquivos serão processados, no entanto, a maioria dos desenvolvedores não trabalha desta maneira, pois é mais simples passar nomes de pacotes, onde estão as classes do software, do que escrever uma lista de arquivos. De fato, existem três maneiras de executar o Javadoc, sem indicar explicitamente os arquivos: (1) passando os nomes dos pacotes; (2) passando os sub pacotes; e (3) usando coringas conjuntamente com partes dos nomes dos arquivos fonte, por exemplo: "*.*java*" indica que devem ser analisados todos os arquivos com a extensão ".java" dentro do diretório escolhido.

Para que o Javadoc possa encontrar as informações no código fonte e gerar a documentação, algumas etiquetas (*tags*) são necessárias. A etiqueta que assinala o início de um bloco de comentários, que será analisado pelo Javadoc, é definida pelo conjunto de caracteres "**/****", que delimita o início de um bloco de comentários do Javadoc e por "*****/**",

marcando o final desse bloco. Para a linguagem Java os blocos de comentário, que não serão analisados pelo Javadoc, são delimitados pelos conjuntos de caracteres "**/*"** , para início e "***/**", para o final de bloco. As demais etiquetas Javadoc sempre começam com o caractere **@**, seguido do descritor da etiqueta. Na Listagem 1 é mostrado o código fonte de uma classe (*Stopwords.java*) onde se pode verificar o uso destas etiquetas (textos realçados).

Por padrão, apenas os membros das classes públicas são mostrados no documento gerado pelo Javadoc. Outro detalhe da ferramenta Javadoc é que ela produz um documento completo a cada vez que é executada, ou seja, não pode fazer compilações incrementais, de maneira a somente modificar ou incorporar novas informações para execuções anteriores.

De acordo com sua implementação, a ferramenta Javadoc exige e depende do compilador Java para fazer seu trabalho, o javac, já que utiliza parte do compilador para fazer a análise das declarações. Por se basear no compilador, fica garantido que o documento HTML corresponde exatamente à aplicação e que pode invocar características implícitas do código fonte, por exemplo, documentando os construtores padrão que estão presentes nos arquivos ".class" e que não aparecem escritos no código fonte que é visto pelo desenvolvedor. Na Figura 2 é apresentado um trecho do documento HTML gerado pelo Javadoc, pode-se observar que o comentário Javadoc, mostrado na Listagem 1, aparece escrito na integra no HTML.

A ferramenta Javadoc cria *links*, nas páginas HTML, com de referência cruzada para o pacote, as classes e os nomes de membros que estão sendo documentados, permitindo que se possa navegar pelo documento gerado. Tais *links* são criados de seguintes maneiras: (1) quando o Javadoc encontra declarações no código fonte consideradas por ele como passíveis de receber um link, (2) pela existência de etiquetas que são processadas diretamente pelo Javadoc. A seguir são mostradas algumas das considerações que o Javadoc faz para criar os *links*:

- declarações de classe, argumentos e métodos (tipos de retorno, tipos de argumentos, tipos de campos, etc.);
- seções "*See also*" do HTML criado, quando encontra a etiqueta *@see*;
- quando encontra a etiqueta {*@link*};
- nos nomes de exceção, a partir da etiqueta *@throws*;
- nas tabelas de resumo, onde são listados pacotes, classes e membros;
- nos nomes dos pacotes e das classes das árvores de herança;

- no índice da página HTML criada.

Listagem 1: Trecho do código fonte para a classe *Stopwords.java*.

```
package Stopwords;
import extractingParagraphsAndSentences.TextManager;
import java.io.BufferedReader;
import java.io.BufferedWriter;

/**
 * Esta classe testa se uma palavra é uma <i>stopword</i>, colocando todas as palavras em
 * minúsculas antes do teste.
 * O formato para leitura e escrita é de uma palavra por linha,
 * as linhas iniciadas por # são interpretadas como comentáros
 * e não serão analisadas.<p/>
 *
 * Aceita os seguinte parâmetros:<br/>
 * <b>-i nomeArquivo</b><br/>
 * carrega as <i>stopwords</i> do arquivo definido por nomeArquivo<br/>
 * <b>-o nomeArquivo</b><br/>
 * salva as <i>stopwords</i> o arquivo definido por nomeArquivo<br/>
 * <b>-p</b><br/>
 * mostra as <i>stopwords</i> correntes na tela<br/>
 *
 * @author Eibe Frank
 * @version $Revision: 1.4 $
 */

public class Stopwords {
    String charSet = "ISO-8859-1";
    public static void main(String[] args) throws Exception {
```

O Javadoc será capaz de criar um documento rico em informações de acordo com a quantidade de comentários e etiquetas que o desenvolvedor escreve no código fonte, Figura 2. Caso o Javadoc gere um documento HTML utilizando apenas com as informações dos elementos de código fonte, percebe-se, Figura 3, que não existem as informações a respeito da classe como mostrado na Figura 2. Para o método da pesquisa será adotado o padrão Javadoc para escrever os comentários do código fonte.

All Classes

Stopwords

Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Stopwords

Class Stopwords

```
java.lang.Object
  └ Stopwords.Stopwords
```

```
public class Stopwords
extends java.lang.Object
```

Esta classe testa se uma palavra é uma stopword, colocando todas as palavras em minúsculas antes do teste. O formato para leitura e escrita é de uma palavra por linha, as linhas iniciadas por # são interpretadas como comentários e não serão analisadas.

Aceita os seguintes parâmetros:
-i nomeArquivo
carrega as *stopwords* do arquivo definido por nomeArquivo
-o nomeArquivo
salva as *stopwords* no arquivo defindo por nomeArquivo
-p
mostra as *stopwords* correntes na tela

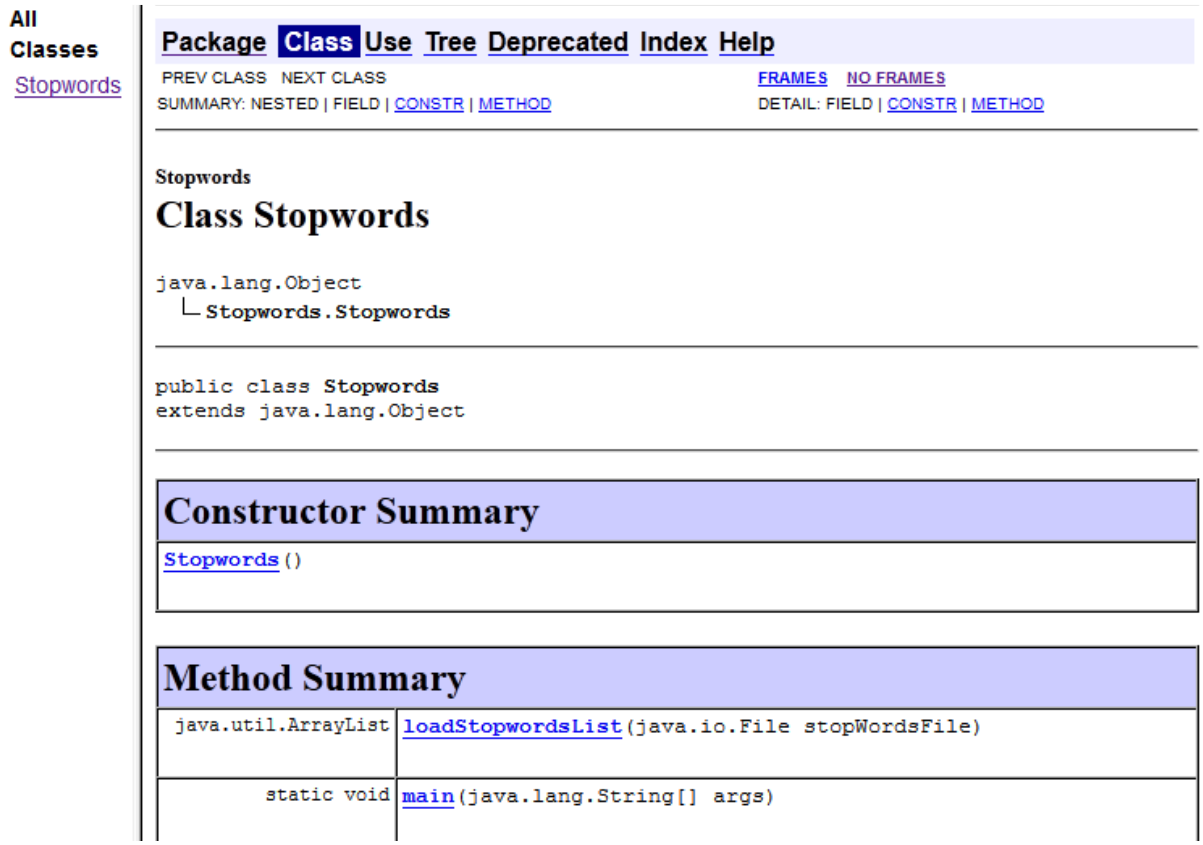
Constructor Summary

Stopwords()

Method Summary

java.util.ArrayList	loadStopwordsList(java.io.File stopWordsFile)

Figura 2: Documento gerado pelo Javadoc para a classe Stopwords.java,
com comentário Javadoc no código fonte.


All Classes
Stopwords

Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Stopwords

Class Stopwords

java.lang.Object
└ Stopwords.Stopwords

public class Stopwords
extends java.lang.Object

Constructor Summary

Stopwords()

Method Summary

java.util.ArrayList	loadStopwordsList(java.io.File stopWordsFile)
static void	main(java.lang.String[] args)

Figura 3: Documento gerado pelo Javadoc para a classe *Stopwords.java*, sem comentário Javadoc no código fonte.

Durante a pesquisa bibliográfica, identificou-se o Lucene como a biblioteca para indexação e consulta de textos a ser usada neste trabalho. Ele apresentava funcionalidades que supriam as necessidades do projeto e, por isso, foi escolhida como a biblioteca de indexação e consulta de textos para ser utilizada na pesquisa, sendo mostrada na seção seguinte.

2.7. Lucene

O Lucene é uma biblioteca, de código aberto, para a recuperação de informação. Escalável e de alto-desempenho, foi escrita inteiramente em Java e disponibiliza recursos que permitem realizar a análise e indexação de textos (McCandless, 2010).

O Lucene[4] processa qualquer informação baseada em texto para depois poder recuperá-la com base em vários critérios de pesquisa, fornecendo a estrutura básica voltada principalmente à indexação e à busca. Possui várias características e recursos, como:

- permite a execução de vários tipos de consultas: *PhraseQuery*, *WildcardQuery*, *RangeQuery*, *FuzzyQuery* e *BooleanQuery*;
- apresenta sintaxe de busca simples;
- realiza a normalização e outros processamentos nos textos através dos seus analisadores;
- possui vários analisadores: *StandardAnalyzer*, *BrazilianAnalyzer*, *SimpleAnalyzer*, *WhitespaceAnalyzer*, *StopAnalyzer*, etc.;
- permite a ordenação por campos (*fields*);
- calcula a pontuação (*score*) para os documentos recuperados e os retorna por ordem de relevância;
- permite a classificação, filtragem e análise da expressão de consulta; e
- permite o bloqueio de índices para impedir modificações simultâneas.

O Anexo 1 apresenta o Lucene de maneira mais detalhada, mostrando algumas das suas funcionalidades. Mais detalhes podem ser encontrados em (McCandless, 2010).

A seguir, será apresentada uma síntese a respeito de alguns trabalhos que utilizam a RI e ao mesmo tempo envolvem código fonte.

2.8. Recuperação de Informação em Código Fonte

Durante o levantamento bibliográfico inicial, buscou-se por trabalhos relacionados à documentação automática ou semiautomática de código fonte. Até a publicação desta dissertação, não foi localizados trabalhos com estas características. Por outro lado, a RI em código fonte está fortemente ligada à busca e recuperação de códigos ou trechos de código que realizem funções similares, de acordo com a necessidade do usuário, através de uma busca ou solicitação (Sindhgatta, 2006).

[4] Embora só trabalhe com textos puros, existem complementos (*add-ons ou plugins*) que permitem fazer um pré-processamento para a extração do texto, a partir de vários tipos de documentos, como por exemplo: arquivos do Microsoft Word™, PDF, XML e HTML, e posteriormente passá-lo ao Lucene.

Artigo 1 - *Using an Information Retrieval System to Retrieve Source Code Samples* (Sindhgatta, 2006)

Este trabalho, apresenta um sistema denominado "JSearch", que faz a busca de códigos fonte e está focado na utilização de um sistema de recuperação de informação para trabalhar com um grande repositório (corpus) de códigos fonte. O autor explica que os sistemas de recuperação de informação, baseados em textos, têm sido bastante utilizados para localizar documentos relevantes dentro do corpus e também é amplamente utilizado nas organizações para a mineração de dados da organização. Estes sistemas são conhecidos por sua escalabilidade e simplicidade. Porém, quando utilizados para busca e recuperação de código fonte não obtém resultados relevantes, uma vez que o código fonte é estruturado e a palavra-chave e a sua localização dentro do código fonte precisam ser considerados. Uma palavra-chave presente num comentário do código fonte pode ter significado diferente em relação à sua ocorrência no bloco de método escrito, assim, algumas extensões devem ser fornecidas para um sistema padrão de recuperação de informações de modo a permitir a busca e recuperação de código fonte.

Sindhgatta (2006) propõe uma ferramenta onde os códigos fonte, publicados pelos desenvolvedores e que formam um corpus, são pré-processados e indexados. Os índices contêm informações relevantes para a linguagem de programação usada e permitem uma pesquisa mais específica no código fonte. Para avaliar a abordagem de Sindhgatta (2006), foi construída uma ferramenta, chamada JSearch, com suporte à linguagem Java, utilizando o mecanismo de busca Lucene[5].

Segundo o autor, a parte cliente da ferramenta está disponível como um *plug-in* do IDE (*Integrated Development Environment*) Eclipse[6] e está publicada no *site* da organização. A parte servidor da ferramenta "JSearch" consiste em um sistema de recuperação de informação, que cria os índices para os arquivos de código fonte do repositório e permite que sejam realizadas consultas sobre estes índices para localizar os arquivos ou os trechos de código que satisfaçam a necessidade do usuário que os está procurando.

Existem algumas etapas necessárias para fazer a indexação destes arquivos. Começa com um pré-processamento que envolve a análise de arquivos de código fonte, escritos em linguagem Java, e sua tradução para um arquivo temporário onde os nomes das variáveis são

[5] (http://lucene.apache.org/)
[6] (http://www.eclipse.org/)

convertidos para os nomes de suas classes. Na Figura 4 pode-se ver um trecho de código, que descreve um método, antes e depois desse pré-processamento. O arquivo resultante é então utilizado para a criação dos índices para este arquivo.

Pode-se notar, na Figura 4, que alguns itens que definem variáveis e objetos, marcados em cinza, foram usados para substituir estas variáveis e objetos nas listagens de código, resultando no código pré-processado que se encontra após a barra de separação entre as duas listagens.

Outra etapa é o processo de indexação propriamente dito. Sindhgatta (2006) definiu a linguagem Java como padrão e também definiu os elementos sintáticos dessa linguagem que podem ser considerados como campos de indexação, tais como: declaração de importação de bibliotecas, classe que está sendo implementada, classe que é herdada, variáveis de método, nomes de método, código fonte do método e comentários.

Os índices podem ser criados para cada um dos elementos sintáticos do código fonte. Definiu-se como campos relevantes para serem avaliados pelo JSearch: **o nome da classe, classe herdada, os nomes de métodos, os tipos de retorno, os comentários do código fonte e as declarações de importação de bibliotecas**. Os índices criados devem ser ótimos o suficiente para garantir que apenas os aspectos pertinentes do código fonte sejam considerados. Um SRI de documentos analisa o texto e o processa, removendo ocorrências comuns de palavras (conhecidas como *stopwords*, por exemplo: um, uma, a, ante, até, ...), analisa as palavras restantes e aplica o método de lematização (*stemming*) para remover terminações morfológicas e de inflexão (por exemplo: computar, computação, computador após a lematização se transformam em *computa*). No contexto de código fonte Java também existem certas palavras da linguagem que precisam ser descartadas durante a indexação, da mesma forma que as *stopwords*, como é o caso da maioria das palavras reservadas para a linguagem Java, por exemplo: o comando *for*, a declaração *int* e outras, que são descartadas para otimizar o tamanho do índice criado.

Neste artigo Sindhgatta (2006) não apresenta resultados para a ferramenta "JSearch", somente cita que ao avaliar os tipos de consultas a partir de um fórum de discussão, cerca de 23% das consultas poderiam ser tratadas pelo "JSearch". Também cita como trabalho futuro a pretensão de implantar a ferramenta para analisar códigos de projetos de uma organização, onde se poderia ter maior utilização da ferramenta. Porém, para isso seria necessário um projeto piloto de sua implantação em uma organização e fazer o acompanhamento do uso da

ferramenta por um período de três a seis meses para que se pudessem quantificar os resultados, verificando sua utilidade e aceitação pelos usuários.

```
public static Document convertProductToXML (Product p){
        DocumentBuilderFactory factory = DocumentBuilderFactory.newIntance();
        Document document = null;
        try {
                DocumentBuilder builder = factory.newDocumentBuilder();
                document = builder.newDocument();
        }
        catch(ParserConfigurationException ex) {
                ex.printStackTrace();
                return null;
        }
        Element root = document.createElement("Product");
        root.setAttribute("name", p.name);
        root.setAttribute("price", p.price);
        document.appendChild(root);
        return document;
}
```

```
public static Document convertProductToXML (Product p){
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        Document document = null;
        try {
                DocumentBulider builder = DocumentBuilderFactory.newDocumentBuilder();
                Document = DocumentBuilder.newDocument();
        }
        catch(ParserConfigurationException ex) {
                ParserConfigurationException.printStackTrace();
                return null;
        }
        Element root = Document.createElement("Product");
        Element.setAttribute("name", Product.name);
        Element.setAttribute("price", Product.price);
        Document.appendChild(Element);
        return Document;
}
```

(adaptado de Sindhgatta, 2006)

Figura 4: Exemplo de pré-processamento do JSearch.

Artigo 2 - *GURU: Information Retrieval for Reuse* (Maarek et al., 1994)

Este trabalho destina-se a construir uma biblioteca de sistemas que fornecem os meios para a representação, armazenamento e recuperação de componentes de software reutilizáveis. Maarek et al. (1994) começam explicando que a primeira etapa na construção dessa biblioteca consiste na indexação dos objetos a serem armazenados. De forma, a produzir um conjunto de atributos, ou perfil, que caracterizam cada um desses objetos, sendo, portanto, a qualidade dessa indexação crucial para a qualidade da biblioteca a ser construída. Os autores relatam que a funcionalidade do código fonte é um aspecto importante dos componentes de software,

assim, é necessário incluir informações conceituais sobre sua funcionalidade nos índices criados. Todavia, essa informação conceitual é difícil de obter automaticamente e poucos programadores fornecem esses índices conceituais para os seus códigos. Além disso, mesmo que fossem fornecidos, dificilmente poderiam ser expressos em um formalismo comum, uma vez que pedaços de código geralmente se originam de várias fontes. A solução então seria indexar manualmente os componentes de software, *a posteriori*, e de acordo com um procedimento de classificação predeterminado, já que essa tarefa é bastante cara computacionalmente. Como alternativa, Maarek et al. (1994) propuseram identificar automaticamente os índices por meio da análise da documentação em linguagem natural, na forma de páginas de manual ou comentários, que geralmente está associada ao código. A documentação em linguagem natural é uma rica fonte de informações conceituais, no entanto, esta informação está implicitamente contida de forma não estruturada. A fim de extrair informações úteis de documentação em estilo livre, os autores propõem o uso de técnicas de recuperação de informação. Uma vez que os índices para determinado código fonte foram produzidos, os componentes podem ser automaticamente classificados, armazenados e recuperados de acordo com seus perfis.

O estágio de classificação na construção de uma biblioteca consiste em recolher objetos em classes de tal forma que os membros dessa classe compartilhem algum conjunto de propriedades. A motivação básica para a classificação é de facilitar a navegação entre os componentes similares, a fim de identificar os melhores candidatos para reutilizar ou, pelo menos, um conjunto de componentes potencialmente adaptáveis. A capacidade de navegação pelos códigos fonte recuperados é uma funcionalidade importante para as bibliotecas de software, até mais importante do que para outros tipos de bibliotecas, uma vez que na recuperação realizada pelo sistema, raramente existe um componente que case perfeitamente com a consulta do usuário. Além disso, a navegação pelas respostas do sistema permite ao usuário descobrir oportunidades inesperadas de reutilização.

Como ferramenta de avaliação para o método descrito por Maarek et al. (1994), foi projetada e implementada a ferramenta chamada "Guru", que incorpora as características descritas anteriormente. O "Guru", de forma automática, monta bibliotecas de software estruturadas conceitualmente a partir de um conjunto de componentes de software não indexados e desorganizados.

Na primeira etapa, o "Guru" extrai os índices a partir da documentação em linguagem natural associada ao componente de software, a ser armazenado, usando um esquema de indexação que é baseado em afinidades léxicas e na sua distribuição estatística, identificando desta forma um conjunto de atributos para cada documento que representa uma descrição funcional da unidade de software associada. Na segunda etapa, o "Guru" reúne os objetos indexados a uma hierarquia de navegação usando uma técnica de agrupamento hierárquico que trata exclusivamente das informações dos índices identificados na etapa anterior. Assim, o "Guru" suporta a recuperação de informação clássica (modelos de busca clássicos), em que os candidatos são classificados de acordo com uma medida numérica (peso) que avalia o quão bem eles responderam à consulta.

Os resultados dos testes realizados com o "Guru" mostraram que ele tem boa precisão na recuperação das solicitações de usuários e que essa precisão pode ser melhorada quando se faz uso da funcionalidade de navegação pelos códigos-fonte.

A principal semelhança destes trabalhos com a pesquisa é a utilização de técnicas de RI para a busca de informações nos códigos fonte. Um fator de contribuição do primeiro trabalho é quanto à definição dos campos relevantes, aqui chamados de elementos relevantes, de código fonte que o sistema avalia que foram os mesmos utilizados pelo método desta pesquisa.

Capítulo 3

Método Proposto

O presente capítulo apresenta detalhes sobre a pesquisa desenvolvida. Serão abordados as técnicas e conceitos utilizados e serão descritos os desenvolvimentos realizados com os quais se atingiu o objetivo da pesquisa. Primeiramente serão apresentados alguns pressupostos que serviram de base para a pesquisa e, por fim, será apresentado o método detalhadamente.

Como mencionado na introdução, esta pesquisa também propõe a estrutura de um documento de referência. O documento de referência visa simplificar o processo de documentação existente em um projeto de software. Permitindo, diferentemente da documentação através de múltiplos artefatos, que os possíveis documentos relacionados, diretamente ao seu desenvolvimento, possam ser agrupados em apenas um documento. Deve-se lembrar de que tal documento não é premissa para o desenvolvimento do método, somente uma maneira de facilitar e organizar as informações para projetos de desenvolvimento de software.

3.1. Documento de Referência

Pretende-se, com a proposta do Documento de Referência, que através de somente um documento seja possível acompanhar todo o desenvolvimento do projeto, como se ele fosse um histórico, sendo construído dinamicamente. O dinamismo desse documento está no fato de ele não ser escrito para ser parte de um dossiê ou uma das etapas do projeto, mas sim ser um documento que tem seu corpo construído conforme o projeto avança. Seu conteúdo no início do projeto é o mínimo necessário para especificar o que é e para quem se destina o projeto a ser desenvolvido e, com o passar do tempo, vai sendo enriquecido ou atualizado com os conteúdos pertinentes de cada uma das fases do projeto.

A essa abordagem deu-se o nome de Documento de Referência, documento esse que tem como principal função reunir todas as informações possíveis que dizem respeito ao projeto e que seriam produzidas em documentos ou artefatos separados, conforme descrito no Capítulo 2 a respeito da documentação no desenvolvimento de software.

Nele figuram as etapas e informações do projeto. Por exemplo: as necessidades para a construção, as normas ou padrões a serem seguidos, os responsáveis por cada uma das etapas e os processos a serem executados. Também fazem parte dele: o histórico do projeto e do desenvolvimento, os requisitos e as restrições do software a ser desenvolvido, cronograma, documentação do código fonte e outros itens.

O documento é dividido em seções, sendo as iniciais relativas à documentação do projeto como um todo (escopo do projeto) e as subsequentes às suas partes ou detalhes, como o código fonte e sua documentação. Trata-se de um documento destinado a fazer o acompanhamento de pequenos projetos de software e pretende estar em concordância com a norma ISO 9001, mesmo que isso não seja uma necessidade para quem o utilizar.

Como apresentado no Capítulo 2, existe um trabalho desenvolvido em torno da definição das especificações exigidas por organismos normalizadores o qual apresenta como ferramenta um documento para ser aplicado em projetos de softwares que envolvem pequenas equipes e que está conforme com as normas ISO 9001. O documento apresentado é chamado de *Small Software Project Management Plan* (Land e Walz, 2006) e foi o ponto de partida para a definição de um documento de referência a ser utilizado nesta pesquisa.

O documento que se propõe está dividido em cinco partes, mostradas na Figura 5 e descritas a seguir:

Introdução, subdividida em apresentação (escopo do projeto), onde é feita uma descrição sucinta do sistema e do software, objetos do plano; histórico, que descreve todas as ações realizadas no decorrer do projeto, desde os primeiros contatos com o cliente, até a fase de finalização do projeto e, por fim, um item destinado aos documentos complementares, que faz referência à documentação produzida para o projeto como códigos fonte, documentação dos códigos fonte, manual do usuário e manual técnico, descrevendo suas funções, algoritmos e estruturas internas.

Visão Geral, descreve o projeto como um produto, mais detalhadamente. Relaciona o material que será entregue ao cliente, datas e locais de entrega e quantidades requeridas para satisfazer os termos do contrato. Provê um delineamento dos requisitos do sistema/software e

vai conter referências à lista detalhada de requisitos do produto, também fazem parte deste item os eventuais subprodutos do projeto, incluindo software reutilizável, aquisição de competências, etc. Traz alguns subitens como: arquitetura, código fonte, documentação a ser produzida, gestão de configuração de software, composição da equipe e restrições do projeto.

Processo de Software, dividido em subitens como processo de desenvolvimento de software, onde são definidas as relações entre as funções principais do projeto e as suas atividades, especificando o tempo de cada uma por meio de datas de referência, revisões, produtos secundários, entregas e respectivos aceites. O modelo do processo pode ser descrito por uma combinação de elementos gráficos e textuais, que abrangem o projeto do início até o fim: atividades de engenharia de software, gestão de requisitos críticos, registro de decisões, recursos utilizados, software reutilizável e teste de software.

Definições e Abreviaturas, lista todas as definições e abreviaturas usadas no documento.

Referências, subdivididas em três tipos, internas que são os processos, políticas e procedimentos de desenvolvimento de software; externas, compostas por leis federais, portarias, manual da empresa do cliente, etc.; e científicas, constituídas por artigos e outras publicações utilizados para respaldar decisões dentro do projeto.

A utilização do documento de referência oferece uma maneira conveniente de reunir a informação crítica necessária do projeto em desenvolvimento, sem incorrer na sobrecarga de documentação necessária. Ele também ajuda a eliminar documentos redundantes, ou seja, documentos que têm conteúdo similar a outros. Isto facilita no que diz respeito a recuperar as informações do projeto sem a necessidade de especificar vários arquivos, além dos arquivos de código fonte, que deverão ser analisados separadamente e, consequentemente, também terá de serem atualizados de maneira diferenciada.

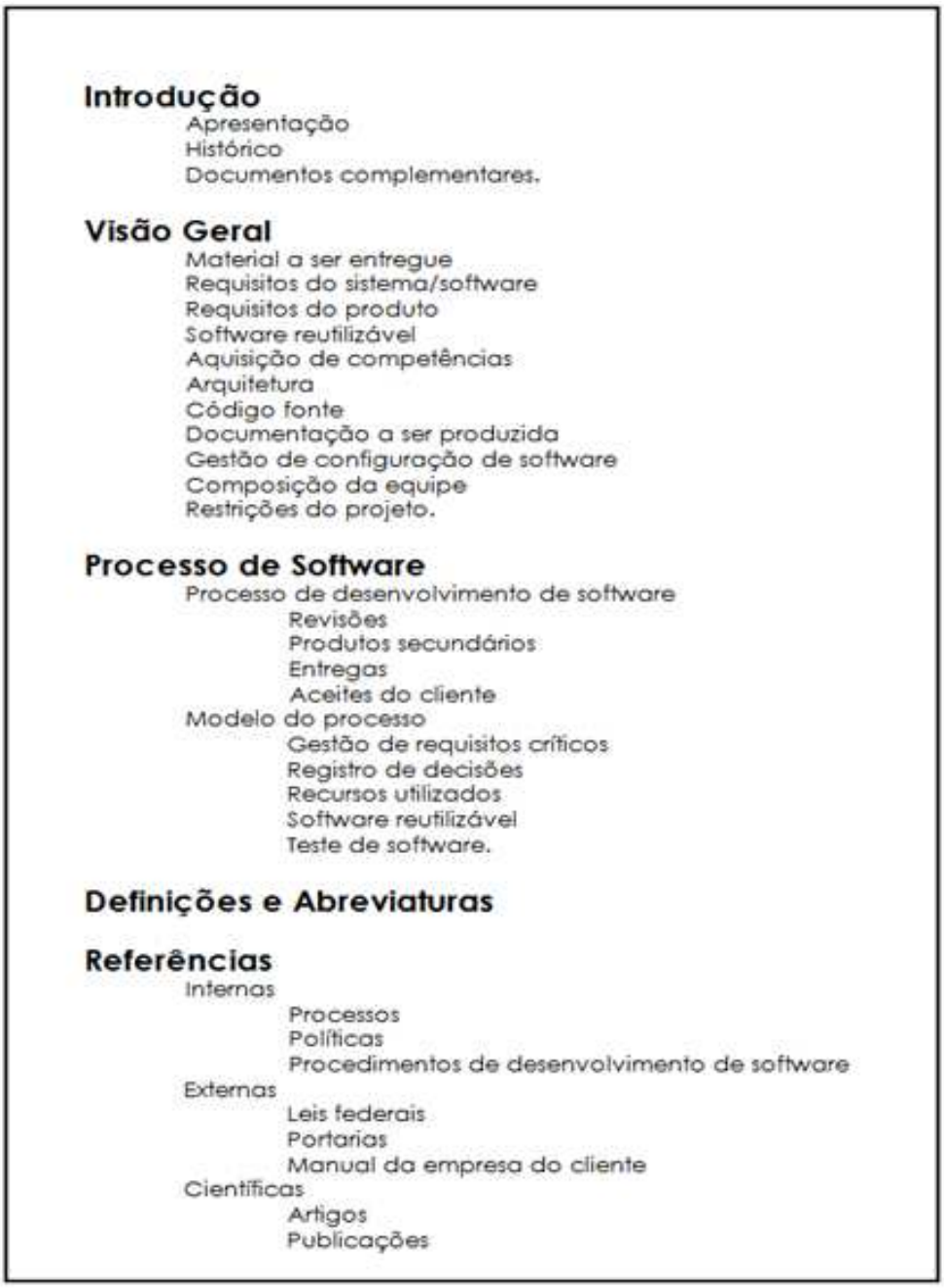


Figura 5: Estrutura do documento de referência.

A partir de agora será apresentado o desdobramento do escopo do projeto. Seguindo os objetivos descritos na Seção 1.2. A pesquisa visa especificar um método para a utilização da RI em documentos de código fonte: os códigos propriamente ditos e no documento de referência, para a retirada de informações, a avaliação e atualização destas informações, na intenção de manter a documentação do projeto.

3.2. Método Semiautomático de Documentação de Código Fonte

O princípio básico do método tem três passos: (1) coletar informações, (2) analisá-las e (3) criar ou atualizar os comentários dos códigos fonte de um projeto de software.

Esses passos, em linhas gerais trabalham da seguinte forma:

(1) Partindo da estrutura do código fonte escrito pelo desenvolvedor, são coletadas determinadas informações ditas relevantes, tais informações, previamente definidas e descritas na Seção 3.2, são as seguintes:

- assinaturas de classes;
- assinaturas de métodos;
- assinaturas de variáveis; e
- todos os tipos de comentários (Java, Javadoc e de linha).

(2) São feitas verificações para encontrar as estruturas de código fonte que não possuem comentários, estas estruturas quando identificadas terão textos de comentários gerados pelo. Já para aquelas que tenham comentários escritos, é feita uma confrontação das informações existentes nos seus textos com as informações da estrutura de código respectiva. Se necessário são adicionadas informações a esses comentários ou, caso tudo esteja correto, nada é feito. Também é feita a verificação das informações recuperadas das classes e métodos com o(s) texto(s) da documentação do projeto, na tentativa de localizar passagens de texto correlatas a essas informações e, caso existam, serão realizadas as devidas operações para complementar os comentários dos códigos fonte. As informações provenientes da documentação do projeto só serão usadas para complementar os comentários de código fonte.

(3) Após a recuperação e análise das informações é feita a atualização do conteúdo do arquivo de código fonte.

Todos os comentários gerados ou atualizados seguem o padrão do Javadoc.

Um diagrama no qual aparecem os principais processos do sistema é apresentado na Figura 6. Observando esta figura, vê-se um bloco representando o corpus e outros três blocos, pertencentes ao método, que estão contidos dentro da área delimitada pela linha.

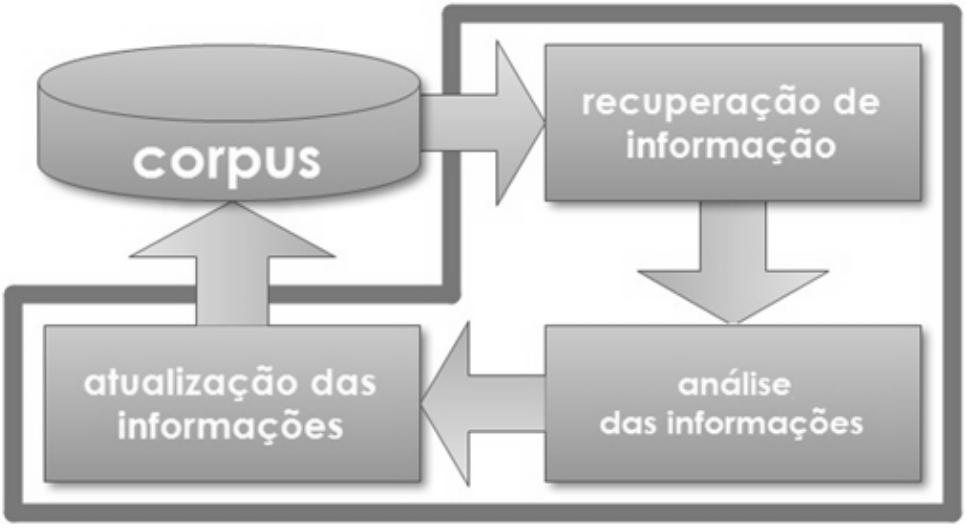


Figura 6: Principais elementos do método.

Para melhor entendimento, será feito um detalhamento de cada um destes blocos pertencentes ao método e do corpus.

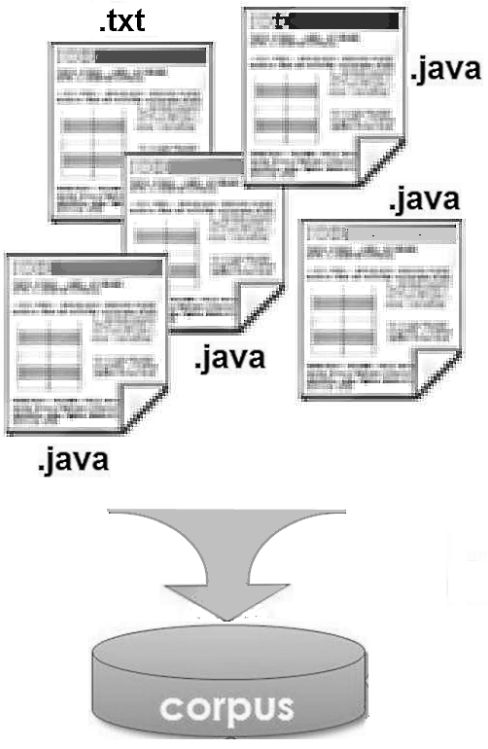


Figura 7: Constituição do corpus.

Corpus: é uma coleção que contém toda a documentação de software existente para o projeto em desenvolvimento. Desta forma, para ser usado pelo método, esse corpus é composto por um documento que contenha a descrição do projeto ou o documento de referência em texto puro, ou seja, um arquivo sem formatação, e diretamente pelos arquivos de código fonte, com extensão .java, conforme a Figura 7.

Na Figura 8 podem ser identificados os três blocos mostrados na Figura 6, porém, expandidos para seus principais processos internos.

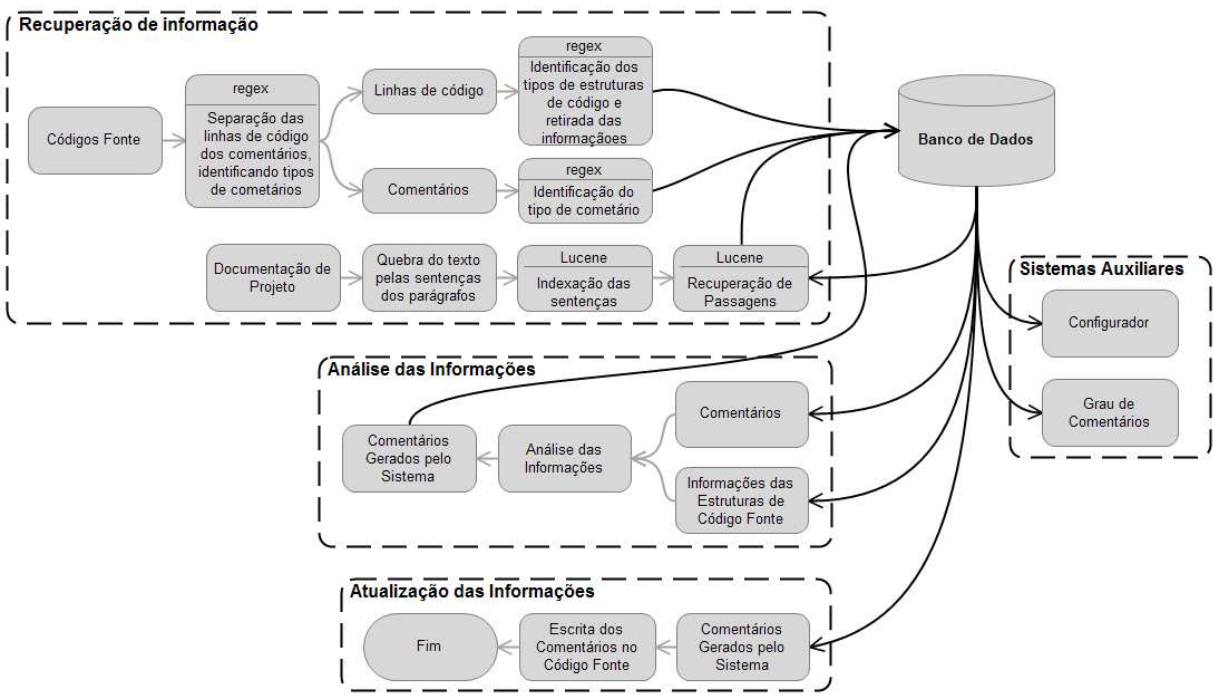


Figura 8: Diagrama geral do método.

Cada um dos blocos será comentado e explicado a seguir:

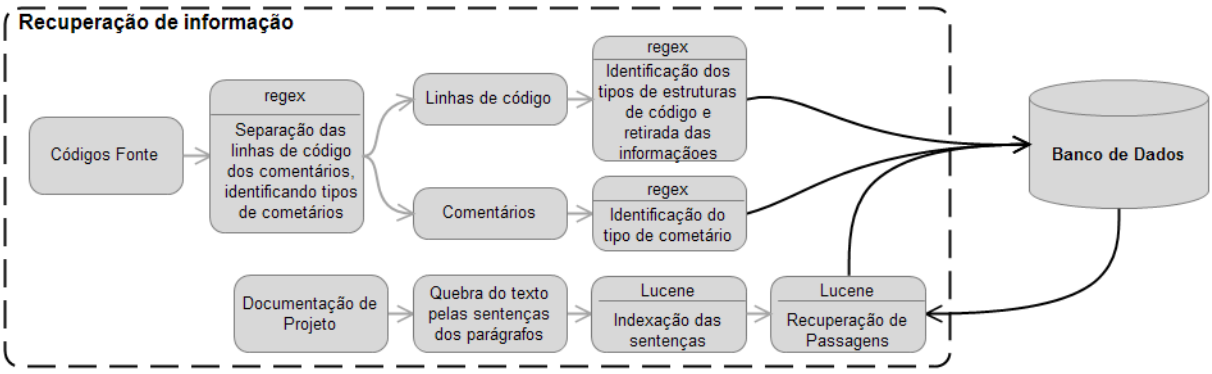


Figura 9: Diagrama do bloco de Recuperação de Informações.

Bloco de Recuperação de Informação: realiza o processamento inicial do Corpus em função do tipo de arquivo em análise, ou seja, o processamento é diferenciado para arquivos de código fonte e de descrição do projeto. Para se recuperar as informações de interesse, depois de definidos os arquivos a serem manipulados, é feito um processamento para a retirada dessas informações e posterior análise.

O início do processo acontece pela análise dos arquivos de código fonte, que são os primeiros arquivos analisados por esse bloco. Isto se deve ao fato de que as linhas de codificação ou o código propriamente dito, escrito pelo desenvolvedor, contém a informação relevante que se deseja ter documentado nos comentários de código e são estas informações que serão utilizadas como termos de busca das informações no documento de referência. Deve-se atentar ao fato que as linhas de código escritas pelo desenvolvedor não podem, nem devem, sofrer alterações pelo método. Portanto, o método atua somente sobre os comentários e sobre as passagens de interesse (Callan, 1996, Cardoso, 2002) do documento de referência, nas seções relacionadas com o código fonte, adicionando ou alterando o seu conteúdo de acordo com as informações recuperadas nas linhas de código. Por exemplo, podem-se identificar as seguintes informações nas linhas de código: nome das classes, dos métodos e dos argumentos utilizados, visibilidade, etc., que servem de termos de busca a serem recuperados dentro dos comentários e do documento de referência.

Para a extração de informações nos arquivos de código fonte, foi usado o casamento de padrões (Crochemore, 1997), utilizando expressões regulares (Thompson, 1968). Os índices, que num método de RI tradicional seriam levantados através da retirada de palavras

não significativas (*stopwords*), lematização (*stemming*) e outras técnicas, são palavras ou estruturas predefinidas, sendo elas as palavras reservadas da linguagem de programação, como por exemplo: *class*, *int*, *float*, etc. Isto se deve principalmente devido à característica semiestruturada dos arquivos de código fonte e do domínio com o qual se está trabalhando serem conhecidos.

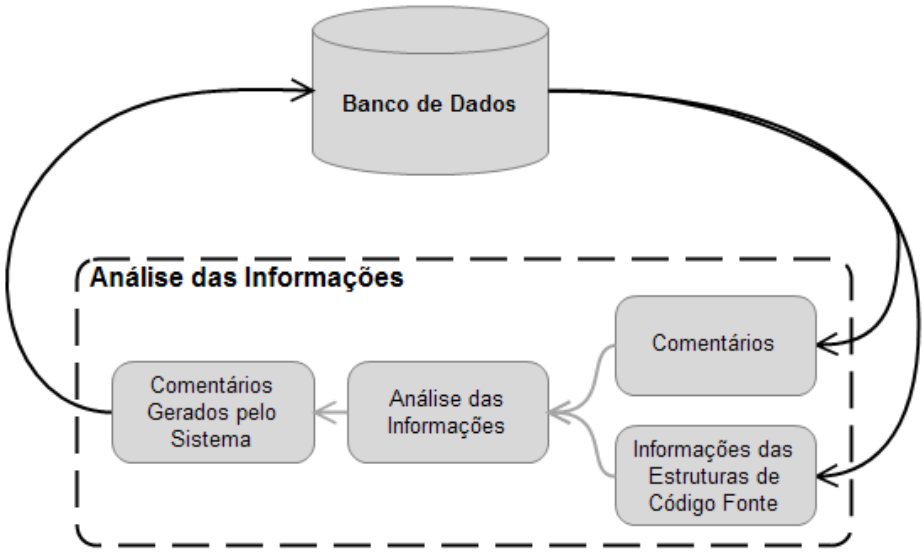


Figura 10: Diagrama do bloco de Análise das Informações.

Análise das Informações: aqui são processadas as informações levantadas pelo processo anterior, para verificar quais informações estão ou não atualizadas, marcando aquelas que têm relação com outras informações, as que não estiverem atualizadas e as que foram modificadas. Neste momento três situações podem ocorrer: (1) a informação não está documentada, se durante a busca das informações do código fonte, nos comentários do tipo Javadoc (Javadoc 2010) ou nas passagens do documento de referência, não se pode verificar a documentação dessa informação. Neste caso, devem ser marcadas onde estas informações devem ser inseridas; (2) a informação está documentada, mas está incompleta, então existe a necessidade de atualizá-la; (3) a informação está documentada e correta, neste caso não há nada a ser feito.

Outro processo realizado é o de análise das informações que foram recuperadas da documentação do projeto. Caso existam passagens para o elemento de código fonte que está sendo analisado, estas passagens serão incorporadas ao comentário. A incorporação destas

passagens acontece na forma de cabeçalho do comentário, ou seja não são associadas às marcações do Javadoc.

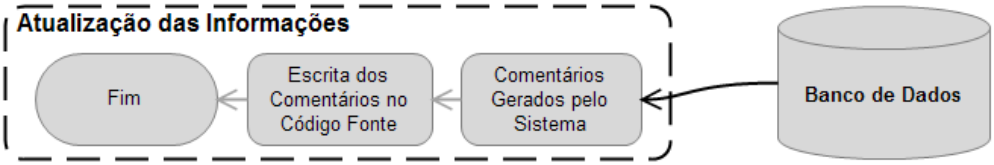


Figura 11: Diagrama do bloco de Atualização das Informações.

Atualização das Informações: uma vez que se sabe o que e onde atualizar é realizada a atualização das informações no descritivo do sistema ou no documento de referência e nos arquivos de código fonte. Para isso, são escritos tópicos nestes arquivos, para que o usuário venha completá-los posteriormente, lembrando que o sistema realiza a atualização de forma semiautomática.

Após a apresentação dos blocos do sistema, segue-se uma descrição detalhada de como acontecem os processos que culminam com o atingimento dos objetivos da pesquisa.

3.2.1. Recuperação de Informação

O processo tem seu início com a leitura do arquivo de código fonte. Este arquivo é lido na íntegra e armazenado como uma sequência de caracteres na memória do computador. Faz-se a análise desta sequência de caracteres com o uso de expressões regulares. Para isso, é feita uma pré-análise buscando identificar se a linha lida contém qualquer um dos grupos de caracteres que identificam o início de comentários, diferenciando-o pelo tipo[7]:

/* para bloco de comentário no padrão do Java;

/** para bloco de comentário no padrão do Javadoc; e

// para comentário de linha.

Encontrando um desses grupos de caracteres, tenta-se identificar possíveis porções de código executável que estejam colocados antes ou depois dos comentários, como se pode ver pelos códigos marcados nos exemplos 1, 2 e 3. Se uma parte da linha for de código

[7] A diferenciação dos comentários pelo seu tipo não apresenta importância para o método, que gerará seus comentários sempre no padrão do Javadoc, porém como é possível e fácil realizar essas identificação, ela será feita e armazenada no banco de dados.

executável, então ela é e armazenada em uma variável específica e eliminada da linha em análise, restando somente o comentário.

Exemplos:

1. `public class EncontrarPassagens { /* //Esta classe é responsável por dividir o texto em passagens e localizar as passagens que tenham relevância de acordo com os termos de busca */`
2. `/** Esta classe é responsável por dividir o texto em passagens e localizar as passagens que tenham relevância de acordo com os termos de busca */ public class EncontrarPassagens {`
3. `public class EncontrarPassagens { // Esta classe é responsável ...`

A ordem com que os padrões são verificados é importante, ou seja, se existir uma sequência que define um comentário de linha e o mesmo estiver dentro de um bloco de comentário do padrão Java ou Javadoc, vide exemplo 1, ele é extraído e identificado como um comentário de linha. Logo, torna-se necessário fazer a busca por uma ordem específica: primeiro tenta-se encontrar um comentário de linha e, caso seja encontrado, faz-se a verificação para os outros tipos. Se após identificar o comentário de linha também puder ser verificado um início de bloco de comentário, no padrão do Javadoc, prevalece como tipo do comentário o do padrão Javadoc. O mesmo acontece com o padrão básico do Java. O primeiro caso ocorre quando é encontrada a sequência de caracteres /** e o segundo quando a sequência de caracteres /* é encontrada. Desta forma, seguindo a ordem determinada, caso tenha sido encontrado o padrão de comentário do Javadoc, o mesmo prevalece sobre os demais, seguido pelo padrão de comentário do Java e por último o de comentário de linha.

Uma peculiaridade da linguagem de programação Java é que blocos de comentário podem ser escritos misturados com a linha de código, sem afetar seu significado, por exemplo, para a linha de código:

```
public SNMPAgent(String logName) throws InterruptedException {
```

É possível escrevê-la da seguinte forma:

```
public SNMPAgent(String logName) /* Agente para leitura das mensagens SNMP
*/ throws InterruptedException {
```

Assim, foi necessário criar um procedimento para identificar e separar a porção de texto de comentário da porção do código executável, de modo a permitir que eles não causem interferência quando um elemento de código é analisado, evitando que informações erradas sejam recuperadas.

Após encontrar um grupo de caracteres de início de comentário o sistema busca identificar se encontra, na própria linha, um grupo de caracteres de final de comentário, se este grupo de caracteres não existir na linha em análise então ele continua lendo as linhas do arquivo de código, armazenando estas informações como um comentário único, até que encontre a sequência de final de comentário, */. Esta sequência de fechamento vale tanto para bloco de comentário do Java quanto do Javadoc. É bom reforçar que isso somente é válido para os comentários de bloco, pois os comentários de linha tem seu fechamento com o caractere de nova linha, \n. Quando um fechamento de comentário é encontrado, o sistema escreve as informações desse comentário no banco de dados. Estas informações armazenadas são: o identificador do arquivo a qual pertence esse comentário (*fatherID*); a data em que esse comentário foi verificado pela primeira vez (*verifiedDate*); a data em que esse comentário foi atualizado (*actualizationDate*); o número da linha de início do comentário (*lineStart*); o número da linha onde ele termina (*lineEnd*); o tipo do comentário (*type*) e o seu conteúdo (*contents*). Estes dados são utilizados pelos outros blocos do sistema.

Depois de identificadas e extraídas as linhas de comentário, o sistema realiza uma nova avaliação, também utilizando expressões regulares, para montar as sequências de linhas de código. Isto acontece através da busca dos padrões de terminação de linha. Para este propósito, fez-se a determinação de quais são as terminações de linha existentes em Java. No caso, as possíveis terminações encontradas e utilizadas no sistema são as seguintes:

- o ponto-e-vírgula (;), usado como terminação para os comandos do Java;
- o caractere de abre chaves ({), que demarca o início de um bloco de comandos; e
- o caractere de fecha chaves (}), que finaliza um bloco de comandos.

Outros tipos de elementos podem aparecer como é o caso das anotações (*@Override*, por exemplo), porém, eles serão desprezados.

Quando uma linha de código contiver um destes caracteres, tem-se o que foi denominado de "linha completa". Essa linha completa é então passada para um analisador, utilizando expressões regulares, que identifica a qual tipo ela pertence. Os tipos de linha completa podem ser: pacote (*package*), importação (*import*), classe, método, definição de

variável ou outros tipos. Os classificados como "outros tipos" são: *if*, *while*, *switch*, *for*, *catch*, *else*, *try* e *finally*.

Quando uma linha de código é classificada como "outro tipo", é descartada e o processo retorna a busca por uma nova linha de código ou de comentário. Se a linha encontrada for de um dos tipos de interesse, ela passa por um processamento para a retirada das informações e é feita a escrita dessas informações no banco de dados, onde a tabela e as informações variam de acordo com o tipo de linha analisada. As informações gerais, pertencentes a todos os tipos de linha são: o identificador do arquivo a que pertencem (*fatherID*), a data em que foi verificada pela primeira vez (*verifiedDate*) e a data em que foi atualizada (*actualizationDate*).

Além das informações anteriores, para cada tipo de linha são guardadas as seguintes informações:

- linha que descreve pacote (package) ou importação (import): nome do pacote/importação (*name*) e seu tipo (*type*), "*package*" ou "*import*".
- assinatura de classe (*class*): o número da linha de início do bloco (*lineStart*); o número da linha de fim do bloco (*lineEnd*); o nome da classe (*name*); sua condição (*status*); o tipo de acesso (*accessType*) e quando herdar uma outra classe, armazena o nome da classe herdada (*extendsTo*).
- assinatura de método: o número da linha de início do bloco (*lineStart*); o número da linha de fim do bloco (*lineEnd*); o nome do método (*name*); sua condição (*status*); o tipo de acesso (*accessType*); o tipo de retorno (*returnType*); se é um método estático ou não (*isStatic*); quando existir(em) argumento(s) guarda seus tipos e nomes (*arguments*) e quando lançar uma exceção, guarda o tipo de exceção (*throwsWhat*).
- assinatura de variável: o número da linha de início (*lineStart*); o número da linha de fim (*lineEnd*); o nome da variável (*name*); o tipo (*type*) e se é uma variável estática ou não (*isStatic*).

Até o momento comentou-se sobre a utilização de expressões regulares (~~regex~~RegEx) para recuperar as informações dos arquivos de código fonte. Para melhor entender esse processo, pode-se fazer uma analogia desta maneira de recuperar informação com um sistema de RI, onde as expressões regulares são comparadas aos termos de busca e o casamento destes

padrões das expressões pode ser comparado ao mecanismo que faz a ligação dos termos de busca com os documentos ou passagens que se quer recuperar.

A seguir são apresentados alguns exemplos das expressões regulares usadas na pesquisa e que fazem o casamento com as linhas de código fonte, com os comentários e com as linhas de código. As expressões regulares apresentam-se como extensas cadeias de símbolos. Por exemplo, para a identificação de um método, o padrão de expressão regular utilizada é mostrado abaixo:

([a-z]{0,}[\s]){0,1}([a-z]{0,}[\s]){0,1}([a-zA-Z[\[\]]]{0,}[\s])([a-zA-Z[_]]{0,}[0-9]{0,}[\s]{0,})(\()([a-zA-Z[\[\]][<[[a-zA-Z[_]]{0,}]>]\s]{0,}[a-zA-Z[_]]{0,})(\)[\s]{0,})([a-zA-Z_]{0,}[\s]{0,})([a-zA-Z_]{0,}[\s]{0,})(\{)

A expressão regular acima está dividida em partes que recebem a designação de grupos, delimitados por parênteses. Logo quebrando a expressão pelos seus grupos e tem-se:

1. ([a-z]{0,}[\s]){0,1}
2. ([a-z]{0,}[\s]){0,1}
3. ([a-zA-Z[\[\]]]{0,}[\s])
4. ([a-zA-Z[_]]{0,}[0-9]{0,}[\s]{0,})
5. (\()
6. ([a-zA-Z[\[\]][<[[a-zA-Z[_]]{0,}]>]\s]{0,}[a-zA-Z[_]]{0,})
7. (\)[\s]{0,})
8. ([a-zA-Z_]{0,}[\s]{0,})
9. ([a-zA-Z_]{0,}[\s]{0,})
10. (\{)

Tomando como exemplo a assinatura de método mostrada abaixo, vamos determinar o que cada grupo identificará:

```
static public String[] extractCommentsFromLines(ArrayList<String>
textToAnalyse) extends analyzeLines {
```

Feito o casamento da expressão regular para o exemplo, tem-se os seguintes resultados retornados para cada grupo:

1. *static;*
2. *public;*
3. *String[]*
4. *extractCommentsFromLines*
5. *(*
6. *ArrayList<String> textToAnalyse*
7. *)*
8. *extends*
9. *analyzeLines*
10. *{*

Assim como o padrão mostrado, outros padrões de expressões regulares foram definidos para os vários elementos de código fonte, sendo eles:

- Final de bloco:
  (}[\s]{0,}[^;])
- Final de bloco:
  (}[\s]{0,})$
- Métodos:
  ([a-z]{0,}[\\s]){0,1}([a-z]{0,}[\\s]){0,1}([a-zA-Z[\\[\\]]]{0,}[\\s])([a-zA-Z[_]]{0,}[0-9]{0,}[\\s]{0,})(\\()([a-zA-Z[\\[\\]][<[[a-zA-Z[_]]{0,}]>]\\s]{0,}[a-zA-Z[_]]{0,})(\\)[\\s]{0,})([a-zA-Z_]{0,}[\\s]{0,})([a-zA-Z_]{0,}[\\s]{0,})(\\{)
- Variáveis:
  ^([a-z]{0,}[\s]){0,1}([a-zA-Z[\[\]][<[[a-zA-Z[_]]{0,}]>]]{0,}[a-zA-Z[_]]{0,}\s)([a-zA-Z0-9_\.]{1,})([\s]{0,}[=;])
- Pacotes:
  (package[\s]{1,})([a-zA-Z_\s]{1,})(;)
- Importações:
  (import[\s]{1,})([a-zA-Z*_\.\s]{1,})(;)

- Classes:
  ([a-z\\s]{0,})(class\\s)(\\s{0,}[a-zA-Z0-9_]{0,}\\s{0,})(\\<[a-zA-Z0-9_\\[\\]\\s,]{0,}\\>){0,1}([a-zA-Z0-9_\\s,]{0,}\\s{0,}){0,1}(\\s{0,}\\{\\s{0,})
- Outros:
  ((if|while|switch|for|catch)[\s]{0,})(\()(.*)(\)[\s]{0,})(\{)
  ((else|try|finally)[\s]{0,})(\{)

Algumas observações para estes padrões podem ser feitas:

- existem dois padrões para o elemento final de bloco, isto é necessário devido a algumas particularidades existentes. Um final de bloco é encontrado quando aparecer o símbolo fecha chaves, seguido por nenhum ou vários símbolos de espaço. No primeiro caso, não podendo ser seguido por um ponto-e-vírgula e, para o segundo caso, a expressão a ser casada deve estar no final da linha, não podendo existir qualquer outro caractere ou símbolo, além de espaços, depois do fecha chaves.

- também existem dois padrões para o elemento outros. Estes dois padrões buscam por elementos que não fazem parte dos elementos desejados, são eles: *if*, *while*, *switch*, *for*, *catch*, *else*, *try* e *finally*; porém, como não contribuem com informações importantes para os objetivos da pesquisa, serão descartados. A necessidade de dois padrões para outros elementos acontece devido à sua sintaxe. No primeiro padrão, tem-se aqueles que se iniciam pelas palavras *if*, *while*, *switch*, *for* e *catch*, seguidos por nenhum ou vários símbolos de espaço e, obrigatoriamente, um símbolo de abre parênteses ou de abre chaves, conforme o caso. No segundo padrão tem-se as palavras *else*, *try* e *finally*, seguidos por nenhum ou vários símbolos de espaço e, obrigatoriamente, por um símbolo de abre chaves.

Com exceção dos elementos classificados como "Outros", quando um padrão é casado com as linhas do texto significa que estas linhas possuem as informações de interesse. Desta forma elas são quebradas para que se possam encontrar suas partes e, então, obter as informações deste elemento do código fonte. Essa quebra nada mais é do que a análise da declaração ou assinatura do Java em busca dos campos que o constituem, separando-os e escrevendo estas informações no banco de dados.

Com base nas informações levantadas das estruturas de código fonte, conforme descrito anteriormente, e através da RI no texto da documentação de gestão. Tenta-se encontrar passagens para serem utilizadas na complementação dos comentários. A intenção é

localizar passagens que possam ser usadas para explicar os elementos de código fonte, desta forma os comentários podem ser melhorados.

Para realizar a RI tomou-se como ferramenta o Lucene (descrito na Seção 3.4 e detalhado no Apêndice A). O Lucene provê vários analisadores integrados, que diferem na maneira pela qual quebram o texto em palavras (*tokenization*) e aplicam ou não filtros para lematização (*stemmer*) e retirada de termos comuns (*stopwords*), por exemplo. Portanto pretende-se verificar qual filtro terá melhor desempenho qualitativo e quantitativo quando aplicado sobre o texto do descritivo do projeto ou documento de referência na busca pelas informações relativas aos elementos de código fonte.

Basicamente são utilizados os nomes das classes e métodos que passam por um processo para desmembramento do seu nome, normalmente escritos no padrão *CamelCase* (Sun Microsystems, 1999) e a busca das passagens dentro do documento de referência ou descritivo do projeto, usando o Lucene.

Um exemplo de um nome em *CamelCase* é mostrado a seguir:

verificarAlteraçõesInformação

Para utilizar esse nome no Lucene, é necessário desmembrá-lo, dividindo o *CamelCase* em palavras. Dessa forma o nome dado como exemplo passa a ser a sequência de palavras mostrada a seguir:

verificar Alterações Informação

O processo para a recuperação de informações na documentação de gestão começa pela divisão do texto em seus parágrafos e, na sequência, cada parágrafo é quebrado em suas sentenças. Então estas sentenças são passadas para indexação pelo Lucene como se fossem textos completos, permitindo recuperar cada uma delas de acordo com sua pontuação (*score*) de relevância e também determinar sua posição dentro do texto. O Lucene permite rotular o documento durante o processo de indexação. Como cada documento é indexado no momento de seu uso, o número do documento de referência é atribuído diretamente a uma variável do sistema. Já para a identificação do parágrafo e da sentença, um código constituído do número do parágrafo seguido de um ponto e o número da sentença dentro desse parágrafo, é gerado

durante o processo de extração das sentenças e é passado como rótulo para o Lucene. Logo, quando se fizer a busca das sentenças, será obtida a posição exata em que ela aparece no texto.

As sentenças indexadas são armazenadas na memória do microcomputador[8].

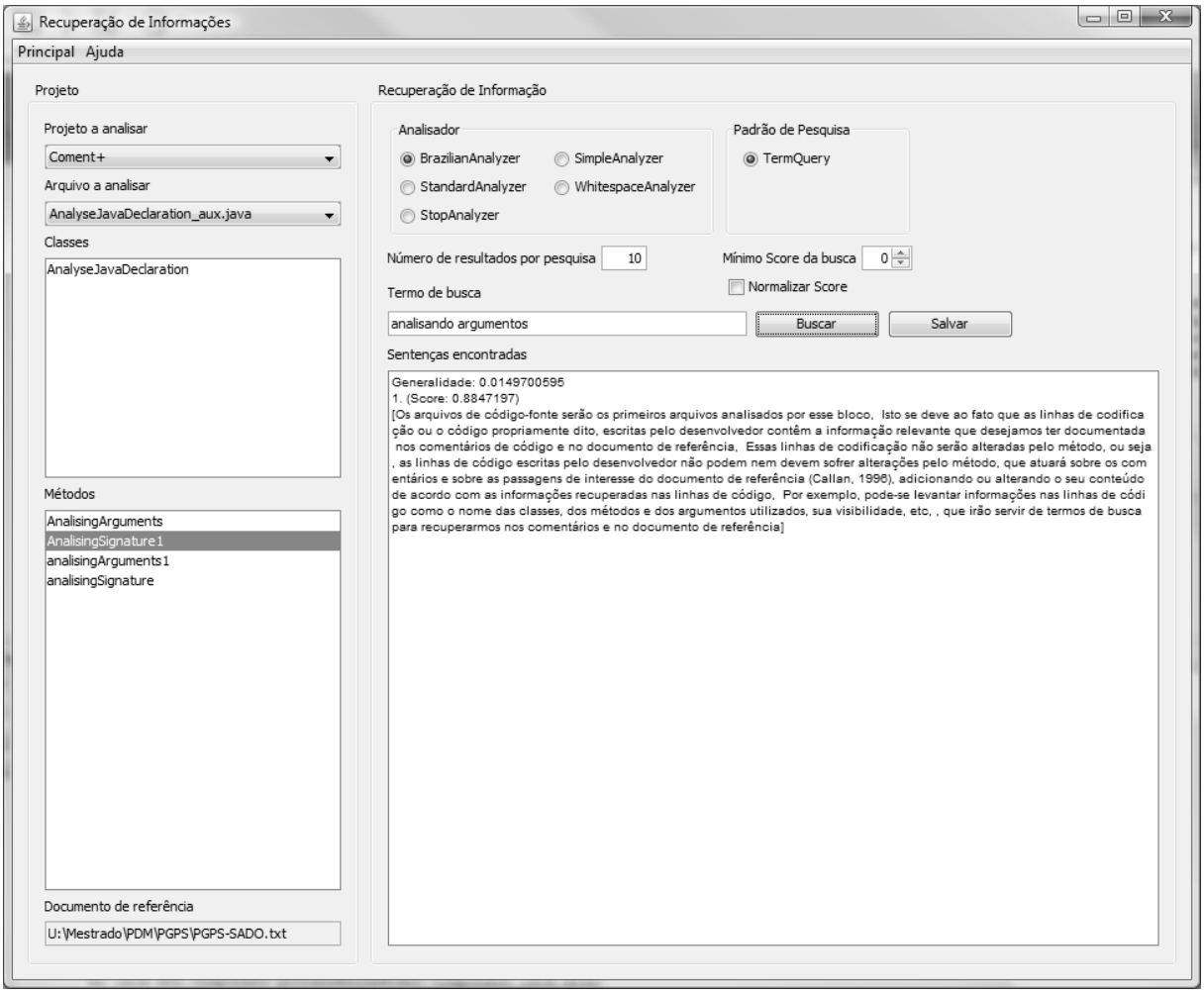


Figura 12: Sistema para recuperação de passagens com o Lucene.

Na Figura 12 é mostrada a tela do sistema auxiliar, desenvolvido para a recuperação de passagens com o uso do Lucene, o qual foi chamado de Recuperação de Informações. Esse sistema foi desenvolvido para contornar a necessidade de traduzir os nomes de classes e de métodos, pois foram identificados códigos onde todos os documentos e comentários estavam escritos em português brasileiro e os nomes dados às classes, métodos e variáveis estão

[8] Por não ser necessário armazenar definitivamente estes índices, os mesmos são destruídos para cada documento de gestão utilizado.

escritos na língua inglesa. Neste sistema são mostradas, à esquerda, as informações recuperadas dos projetos cadastrados e analisados pelo Comente+[9]. Ele também possibilita alterar os parâmetros do Lucene, como o analisador a ser usado na indexação e busca e a quantidade e o valor mínimo de *score* dos resultados que devem ser retornados.

3.2.2. Análise das Informações

Após a etapa de verificação e extração das informações do código fonte, começa o segundo bloco do processo. Nele faz-se a confrontação das informações encontradas como comentários contra as informações levantadas para os elementos de código fonte, as classes, métodos e variáveis. Além dessa confrontação, este bloco também procura criar comentários que são armazenados no banco de dados.

O processo acontece pelo uso das informações anteriormente coletadas, informações essas que são divididas em dois grupos: o primeiro formado pelos comentários existentes no código fonte e o segundo grupo formado pelas informações coletadas das classes, métodos e variáveis. Estas informações são buscadas diretamente nos comentários identificados como próximos a uma estrutura de código fonte, ou seja, quando existe uma assinatura de classe precedida por um comentário, esse comentário é dito próximo dessa classe.

Seguindo essa colocação, são lidas as informações dos elementos de código fonte para cada um dos arquivos de código dos projetos ativos. Para cada elemento de código, procura-se por um comentário próximo. Caso exista, são feitas buscas para as informações relativas àquele elemento de código no texto do comentário, buscando verificar se as informações existem. Se existirem, o sistema não faz nada e passa a procurar por outro elemento de código para a verificação. Caso sejam verificadas somente algumas ou nenhuma das informações, o sistema cria um aviso, especificando quais itens não estão presentes no comentário, e o armazena no banco de dados. Por fim, caso não exista um comentário próximo do elemento de código em análise, este é criado seguindo as recomendações do Javadoc. Sendo escrito um texto explicativo e na sequência algumas informações, na forma de etiquetas do Javadoc, conforme mostra a Listagem 2.

---

[9] Comente+ é o nome dado ao sistema desenvolvido para avaliar o método descrito nesta pesquisa.

Listagem 2: Exemplo de comentários gerados pelo sistema

```
/*
 * A classe Atendente, cuja acessibilidade é pública foi implementada para...
 * *
 * Esta classe tem os seguintes métodos:
 * Atendente:
 * run:
 * Atendente:
 * Login:
 * run:
 *
 */
public class Atendente implements Runnable {
    private Socket s; // Esta variável está definida para ...

/*
 * O método público Atendente tem como principal função ...
 *   @param (Socket) s
 *   @return void
 */
    public Atendente ( Socket s )
    {
        this.s = s;
    }
```

Também é tarefa deste bloco, verificar se existem passagens que possam complementar os comentários. Encontradas as passagens de interesse, após a realização de uma busca, elas podem ser adicionadas aos comentários encontrados/produzidos pelo bloco anterior. Esses comentários serão armazenados no banco de dados até que o terceiro bloco do método ocorra.

3.2.3. Atualização das Informações

Depois de feita toda a pesquisa das passagens e atualização/criação dos comentários para o projeto de software em análise, tem início o terceiro bloco. Ele simplesmente realiza a operação de escrita dos comentários no código fonte e de eventuais textos no descritivo de projeto ou documento de referência.

Uma vez que se tem a posição dos comentários, pode-se escrever neles. O posicionamento dos comentários, definido pelo seu número de linha, se dá no momento em que eles são extraídos dos códigos fonte ou quando são gerados para as classes, métodos e variáveis, que não os possuem ou os possuem de forma incompleta.

Uma vez indexados os documentos, feita a busca dos termos e encontradas as passagens relevantes, elas são adicionadas ao comentário do elemento do qual se utilizaram as

informações para a busca com o Lucene. Por exemplo, se foi utilizado o nome de um método para a busca, ao serem encontradas passagens para essa busca, estas passagens são adicionadas ao comentário desse método e atualizado o banco de dados com esse novo comentário, para posterior atualização do código fonte.

O método descrito pode ter sua operação automatizada, permitindo que a documentação fique sempre atualizada para períodos previamente estipulados através de configuração, sem a necessidade de intervenção humana.

Capítulo 4

Experimentos e Resultados

O objetivo desta seção é apresentar e avaliar os resultados dos experimentos realizados durante a pesquisa para o sistema desenvolvido segundo o método proposto. Este capítulo Está dividido em três partes: Plano de testes (onde foram definidas as hipóteses a verificar, a constituição do corpora utilizado nos experimentos e as métricas de avaliação), Resultados (onde se apresentaram os resultados dos experimentos realizados) e Discussão dos resultados (onde se fez a análise e discussão dos resultados).

4.1. Plano de Testes

Para a realização dos testes, foram definidas algumas hipóteses a serem verificadas, o corpora utilizado nos experimentos e as métricas de avaliação.

4.1.1. Hipóteses a Verificar

Definiram-se quatro hipóteses a serem verificadas:

1. **Capacidade de recuperação das informações dos códigos fonte:**
   O sistema deve ser capaz de recuperar as informações ditas relevantes dos arquivos de código fonte. Para a verificação desta hipótese o sistema será testado com pelo menos três projetos distintos.
2. **Relevância das passagens recuperadas pelo Lucene:**
   O sistema deve ser capaz de recuperar passagens relevantes da documentação de gestão do projeto ou qualquer outro documento do tipo texto que contenha informações que possam explicar os códigos fonte em análise. Para testar esta

hipótese será verificada a precisão e a revocação do Lucene e a qualidade das passagens recuperadas pela avaliação de especialistas em pelo menos um projeto.

3. **Capacidade de geração de comentários.**

   O sistema deve ser capaz de criar ou atualizar comentários nos códigos fonte a partir de informações recuperadas dos próprios códigos fonte e do descritivo de projeto. Para verificar essa hipótese o sistema deve ser testado com pelo menos três projetos.

4. **Grau de comentários.**

   Será verificado o quão comentado estão os códigos fonte escritos pelo desenvolvedor, antes e depois do sistema analisar estes códigos fonte. Esta hipótese deve ser testada com pelo menos três projetos.

4.1.2. Corpora

Para a experimentação, foram utilizados três projetos com seus respectivos descritivos para compor o corpus, conforme a Tabela 1.

Tabela 1: Corpora.

Projeto	**Sistema Inteligente de Gestão Integrada de Alarmes e Diagnóstico de Falhas em Redes de Telecomunicações (Tecpar, 2011)**			
Nome reduzido	Total de arquivos	Total de classes	Total de métodos	Desenvolvedores
SE-Telecom	25	25	162	Nilton Barbosa Armstrong Júnior
Descritivo do Projeto	PGPS - SETELECOM.txt			
Total de páginas	7	Total de palavras		3.323
Projeto	**Um Método de Identificação de Emoções em Textos Curtos Para o Português do Brasil (Martinazzo, 2011)**			
Nome reduzido	Total de arquivos	Total de classes	Total de métodos	Desenvolvedores
Emoções	1	1	5	Barbara Martinazzo
Descritivo do Projeto	BM-0111.txt			
Total de páginas	53	Total de palavras		17.119
Projeto	**Especificação do Modelo do Participante de Um Projeto de Desenvolvimento Colaborativo de Software (Wanderley, 2011)**			
Nome reduzido	Total de arquivos	Total de classes	Total de métodos	Desenvolvedores
MODUS-SD	5	6	23	Gregory Moro Puppi Wanderley
Descritivo do Projeto	Relatorio Final-v1.txt			
Total de páginas	11	Total de palavras		3.795

A seguir apresenta-se um breve descritivo de cada um dos projetos componentes do corpora.

- **SE-Telecom** - é um sistema inteligente de gestão integrada de alarmes e diagnóstico de falhas em redes de telecomunicações. Usando técnicas de Inteligência Artificial e Engenharia do Conhecimento (Sistemas Especialistas), o projeto tem como principal função apoiar os operadores de rede e especialistas na análise das situações de falha, diagnóstico dos problemas ocasionados pelo crescimento acelerado das redes de telecomunicações nos últimos anos e a proposição de ações corretivas cabíveis ou necessárias para garantir a eficiência e qualidade dos serviços prestados aos clientes de empresas de telecomunicações.
- **Emoções** - sistema de identificação de emoções em bases textuais escritas em português do Brasil. O objetivo é identificar uma das seis emoções básicas (alegria, raiva, tristeza, desgosto, medo e surpresa) em notícias curtas. A partir da identificação destas emoções, elas serão utilizadas para a animação facial de um avatar (agente conversacional animado) que lê tais textos. O avatar que será responsável pela leitura do texto modificará seu comportamento, basicamente suas expressões faciais, de acordo com as emoções encontradas pelo sistema de identificação de emoções no decorrer do texto.
- **MODUS-SD** - o objetivo fundamental desta pesquisa é desenvolver um estudo teórico e prático para a construção do modelo (perfil) dinâmico dos participantes de um projeto de desenvolvimento de software em pequenas equipes colocadas. Para tal, técnicas de aprendizagem de máquina, mineração de textos, recuperação da informação e processamento de linguagem natural serão utilizadas.

4.1.3. Métricas de Avaliação

As métricas usadas estão descritas de acordo com cada uma das hipóteses a serem provadas. São elas:

1. **Capacidade de recuperação das informações dos códigos fonte.**

   Arquivos de códigos fonte são definidos por uma gramática específica (Gosling, 2005), no caso o *Java Language Specification*.~~,~~ ~~desta forma~~Asim, é necessário que o sistema consiga recuperar as informações sem apresentar erros, ou seja, todas as estruturas de código fonte devem ser identificas e suas informações recuperadas.

2. Relevância das passagens recuperadas com o Lucene.

Utilizou-se as medidas padrão de avaliação de sistemas de recuperação de informações (Weiss, 2005). Estas medidas ou métricas são a precisão (Equação 1) e a revocação (Equação 2). A precisão determina quantas passagens relevantes foram recuperadas com relação ao total de passagens recuperadas (compreendendo passagens relevantes e não relevantes). A revocação determina a proporção de passagens relevantes que foram retornadas como resultado a uma consulta do usuário com relação ao número total de passagens relevantes existentes.

$$Precisão = \frac{\#\ de\ passagens\ relevantes\ recuperadas}{\#\ total\ de\ passagens\ recuperadas} \qquad \text{(Equação 1)}$$

sendo:

# de passagens relevantes recuperadas	*número de passagens relevantes que foram recuperadas pelo Lucene; e*
# total de passagens recuperadas	*número total de passagens recuperadas, compreendendo as passagens relevantes e não relevantes.*

$$Revocação = \frac{\#\ de\ passagens\ relevantes\ recuperadas}{\#\ total\ de\ passagens\ relevantes} \qquad \text{(Equação 2)}$$

sendo:

# de passagens relevantes recuperadas	*número de passagens relevantes que foram recuperadas pelo Lucene; e*
# total de passagens relevantes	*número total de passagens relevantes existentes no descritivo de projeto.*

A partir dos valores de revocação e precisão, pode-se calcular a medida F que fornece a média ponderada dos valores de precisão e revocação (Equação 3).

$$Medida\ F = \frac{2 * precisão * revogação}{precisão + revogação} \qquad \text{(Equação 3)}$$

3. **Capacidade de geração de comentários.**

Para avaliar a capacidade do sistema de gerar comentários, verificou-se se existiam ou não comentários para cada um dos elementos relevantes dos códigos fonte. Para isso fez-se a contagem manual de quantos elementos relevantes continham comentários antes da aplicação do método, e quantos comentários foram gerados pelo sistema para estes elementos. Calculou-se o percentual de comentários gerados em relação ao total de comentários existentes (Equação 4)

$$\%\ geração\ de\ comentários = \frac{\#\ comentários\ gerados}{\#\ comentários\ existentes} \qquad \text{(Equação 4)}$$

sendo:

# comentários gerados	*número de comentários gerados após a análise do código fonte pelo Comente+; e*
# comentários existentes	*número de comentários existentes nos códigos fonte antes da análise com o Comente+.*

4. **Grau de comentários.**

O grau de comentários do código fonte é um índice que foi criado para esta pesquisa. Esse índice não avalia a qualidade dos comentários, mas sim, o quão comentado está o código fonte, sendo ~~e está~~ definido de acordo com a (Equação 5).

$$grau\ de\ comentários = \frac{total\ de\ comentários}{número\ de\ classes + número\ de\ métodos + número\ de\ variáveis} \qquad \text{(Equação 5)}$$

sendo:

total de comentários	*total de comentários existentes no código fonte ou, após este ser analisado pelo Comente+, o número de comentários existentes no código fonte somado ao número de comentários criados pelo Comente+;*
número de classes	*número total de classes encontrados pelo Comente+;*
número de métodos	*número total de métodos encontrados pelo Comente+; e*
número de variáveis	*número total de variáveis encontradas pelo Comente+.*

É importante observar que o valor do grau de comentários pode ser maior do que um (1), já que os comentários podem ser utilizados indiscriminadamente no código fonte, não estando necessariamente relacionados aos elementos de código fonte.

4.2. Resultados

Foram realizados os testes e os resultados estão detalhados a seguir:

4.2.1. Capacidade de recuperação das informações dos códigos fonte

O teste para verificação da capacidade de recuperação de informações dos códigos fonte utilizando o ~~regex~~ RegEx foi realizado através do processamento de todos os arquivos de código fonte para os três projetos do corpora. Como resultado, verificou-se que o sistema recuperou todas as informações existentes para os elementos considerados relevantes dos códigos fonte, conseguindo identificar as assinaturas de classes, métodos e variáveis e, a partir delas, obter as informações de modificadores, identificador, herança e interfaces das classes, modificadores, tipo de retorno, identificador e parâmetros dos métodos e modificadores, tipo e identificador das variáveis.

4.2.2. Relevância das passagens recuperadas com o Lucene

Para avaliar a relevância das passagens recuperadas pelo Lucene, utilizou-se o projeto SE-Telecom. A escolha desse projeto aconteceu por ele ser o único para o qual se conseguiu contato com o desenvolvedor durante todo o andamento desta pesquisa. Desta forma, ~~para que o mesmo~~o desenvolvedor pôde ~~avaliasse~~ avaliar as passagens do descritivo de projeto e também as passagens recuperadas pelo Lucene permitindo~~, desta forma, calcular a~~o calculo da precisão (Equação 1), revocação (Equação 2) e a Medida F (Equação 3).

Numa primeira etapa foram verificadas todas as passagens que constituíam o descritivo de projeto. Cada uma destas passagens recebeu uma classificação segundo dois critérios:

- **"Possuem relação"** para aquelas passagens que, de alguma forma, descrevem um ou mais elementos dos códigos fonte do projeto; e
- **"Não possuem relação"** para aquelas passagens que não descrevem os elementos dos códigos fonte do projeto.

O descritivo de projeto do SE-Telecom é composto por 155 passagens que foram indexadas pelo Lucene. Estas passagens foram apresentadas ao desenvolvedor do projeto que as classificou pelos dois critérios apresentados anteriormente. Os resultados para essa avaliação estão na Tabela 2.

Tabela 2 - Classificação das passagens do descritivo de projeto do SE-Telecom pelo desenvolvedor

SE-Telecom	
Número de passagens que "Possuem relação"	37
Número de passagens que "Não possuem relação"	118

Na segunda etapa foram realizadas as buscas de passagens com o Lucene, onde se procedeu de duas formas: (1) utilizando como termo de busca o identificador de classe ou método de modo direto, somente com a retirada das palavras comuns (*stopwords*), ou seja, utilizando as palavras que formam o identificador separando-as do padrão *CamelCase* e eliminando as palavras comuns; e (2) utilizando como termo de busca o identificador de classe ou método, desta vez retirando as palavras comuns e adicionando o operador *AND* entre as palavras que formam o identificador, após sua separação do padrão *CamelCase*. Por exemplo: os termos de busca para o identificador "leituraDeDadosDoBanco" seriam: no caso (1) "leitura Dados Banco"; e no caso (2) "leitura AND Dados AND Banco".

Todas as passagens recuperadas, com e sem o uso do operador AND, foram avaliadas pelo desenvolvedor e receberam uma classificação segundo três outros critérios:

- **"Sem relação"** para aquelas passagens que não apresentaram qualquer tipo de relação com o termo de busca analisado;
- **"Pouca relação**" para as passagens que apresentaram alguma relação com o termo de busca analisado; e
- **"Total relação**" para as passagens que, de alguma forma, explicam o termo de busca analisado.

Os resultados da classificação das passagens recuperadas pelo Lucene para o caso (1) estão na Tabela 3.

Tabela 3: Avaliação das passagens recuperadas pelo Lucene para o projeto SE-Telecom, caso (1).

SE-Telecom	Sem relação	Pouca relação	Total relação

Total de passagens recuperadas = 143	72	11	60
% com o total de resultados recuperados	50%	8%	42%

Na Tabela 4 estão as análises das passagens recuperadas pelo Lucene, utilizando o operador lógico AND, caso (2), nos termos de busca.

Tabela 4: Avaliação das passagens recuperadas pelo Lucene para o projeto SE-Telecom, usando o operador AND, caso (2).

SE-Telecom	Sem relação	Pouca relação	Total relação
Total de passagens recuperadas = 48	7	19	22
% com o total de resultados recuperados	15%	39%	46%

A partir destas classificações pode-se então calcular a precisão, revocação e a Medida F para as passagens recuperadas usando o operador AND, conforme a Tabela 5.

Tabela 5: Valores de precisão, revocação e medida F das passagens recuperadas pelo Lucene para o projeto SE-Telecom

Projeto SE-Telecom	
Precisão	0,3958
Revocação	0,5135
Medida F	0,4471

4.2.3. Capacidade de geração de comentários

Para a verificação da capacidade de geração de comentários foi realizada a contagem manual dos comentários para os elementos de código fonte relevantes, antes e depois da sua análise pelo Comente+–, conforme Tabela 6. Esta contagem também serviu para fazer a confrontação com o número de elementos relevantes encontrados pelo Comente+. Na

Tabela 7

~~Tabela 7~~ tem-se os valores do % de geração de comentários pelo Comente+ para os projetos do corpora.

Tabela 6: Contagem manual de comentários dos projetos antes e depois de analisados pelo Comente+.

Corpus	Comentários existentes	Comentários gerados
SE-Telecom	341	501
Emoções	85	25
MODUS-SD	327	350

Tabela 7: % de geração de comentários calculado.

Corpus	% de geração de comentários
SE-Telecom	147%
Emoções	29%
MODUS-SD	107%

4.2.4. Grau de comentários

O valor do grau de comentários pode ser maior do que um (1), já que os comentários podem ser utilizados indiscriminadamente no código, não estando necessariamente relacionados com os elementos de código fonte.

Para a realização dos cálculos do grau de comentários (Equação 5), fizeram-se consultas ao banco de dados para levantar qual foi o número de comentários, classes, métodos e variáveis originalmente encontrados pelo Comente+, ou seja, comentários escritos pelo próprio desenvolvedor. Com estas informações, fez-se um primeiro cálculo determinando o grau de comentários.

Em seguida foi realizada nova busca ao banco de dados para verificar quantos comentários foram criados para serem inseridos nos códigos fonte, pelo Comente+. Esse número do total de comentários gerados foi somado ao número de comentários originais do projeto. Essa soma representa a totalidade de comentários e com ela fez-se então um novo cálculo do grau de comentários.

Os resultados para as buscas dos números de comentários antes e depois do Comente+ e os seus respectivos graus de comentários, para todos os projetos do corpora, estão na Tabela 8.

Tabela 8: Grau de comentários dos projetos analisados pelo Comente+.

Corpus	Comentários existentes	Comentários inseridos	grau de comentários	
			Antes do Comente+	Depois do Comente+
SE-Telecom	341	501	0,681	1,681
Emoções	85	25	3,269	4,231
MODUS-SD	327	350	0,934	1,934

4.3. Discussão dos Resultados

Para discutir os resultados seguiu-se a mesma divisão feita para os testes.

4.3.1. Capacidade de recuperação das informações dos códigos fonte

A busca de informações utilizando expressões regulares se mostrou flexível, já que alterações nas expressões regulares permitem modificar facilmente os padrões de busca. Também pode-se perceber que ela é precisa, ou seja, “todos” os elementos de código fonte, ditos relevantes, foram encontrados.

4.3.2. Relevância das passagens recuperadas com o Lucene

O Lucene possui vários tipos de analisadores (*BrazilianAnalyzer*, *SympleAnalyzer*, *StandardAnalyzer*, *StopAnalyzer* e *WhitespaceAnalyzer)*, que fazem tratamentos diferenciados dos textos, durante a indexação, e também nas expressões de busca, durante a recuperação de informações. Por isso, fez-se uma avaliação prévia, utilizando os projetos do corpus, a fim de verificar qual analisador apresentava melhores e mais resultados para as buscas.

Pode-se perceber que o melhor resultado foi com o uso do analisador *BrazilianAnalyzer*, que retornou mais passagens da documentação de projeto para cada um dos termos de busca, no caso, os nomes de classes e métodos recuperados dos códigos fonte, 17,7% para a documentação do SE-Telecom, conforme a Tabela 9. Desta forma, escolheu-se o *BrazilianAnalyzer* como analisador a ser utilizado pelo Lucene.

Tabela 9 - Total de sentenças recuperadas pelo Lucene na análise do arquivo da documentação dos projetos.

BrazilianAnalyzer			
Projeto	Número de estruturas dos códigos fonte	Número de passagens encontradas pelo Lucene com verificação do score	% de passagens encontradas pelo Lucene
Comente+	129	4	3,1%
SE Telecom	158	19	12,2%
Emoções	6	0	0,0%
MODUS-SD	12	2	1,7%

Após a escolha do analisador, foi feita a avaliação dos resultados retornados pelo Lucene[10]. Observa-se que o SE-Telecom teve melhor resposta, com 12,2% do total de passagens recuperadas. O projeto Emoções, por ter poucos elementos, 6 no total, acabou por não apresentar passagens recuperadas. O projeto MODUS-SD, apesar de também possuir poucos elementos de código fonte, 12 no total, retornou 1,7% do total de passagens indexadas, conforme resultados da Tabela 9. Foram feitas análises para verificar os motivos desta variação, e chegou-se às seguintes conclusões:

- **qualidade dos textos dos descritivos de projeto**: através de observação, percebe-se que existem textos que apresentam maior nível de detalhes na descrição da implementação de software. Este fato tem relação direta com a qualidade e quantidade da recuperação de informações pelo uso do Lucene.
- **nomes dos elementos de código fonte**: também constatado por verificação direta dos textos, percebeu-se que nomes muitos gerais apresentam retorno de informações muito amplas e cuja qualidade de conteúdo não diz respeito ao termo de busca usado pelo Lucene para recuperar as informações. Um exemplo de nome geral seria um método Java, chamado de "leitura", para o acesso a um banco de dados. Se fosse escrito como "leituraDeDadosDoBanco", permitiria ao Comente+ criar o termo de busca "leitura AND dados AND banco", que é menos abrangente do que somente "leitura" e, seguramente, retornaria passagens melhor relacionadas com o método escrito.
- **quantidade de elementos de código fonte**: alguns dos projetos analisados pelo sistema apresentavam poucas linhas de código e, consequentemente, poucas classes e métodos implementados. Um desses casos é o projeto Emoções que utilizava APIs de terceiros e seu código fonte restringia-se a simplesmente acessar estas APIs. Isso resultou na baixa resposta do Lucene quanto à recuperação de informação para o código deste projeto.

Pela análise junto aos desenvolvedores para determinar a qualidade dos retornos das buscas com o Lucene, notou-se que com a utilização direta dos nomes dos elementos de código fonte como termos de busca, o retorno foi de 60 passagens, que apresentavam total relação com o termo de busca. Isso representou 42% do total de 143 passagens recuperadas,

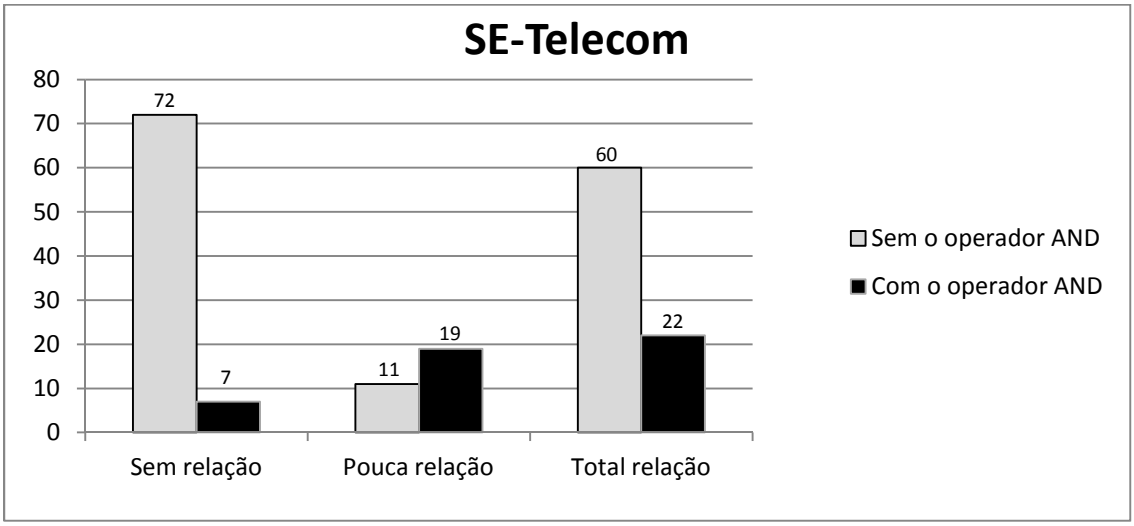
[10] O Apêndice 2 traz os resultados textuais obtidos.

na análise do SE-Telecom, Gráfico 1, 60 destas passagens, 42%, tiveram total relação com os termos de busca.

Quando se utilizou o operador AND nos termos de busca, os percentuais das passagens com total relação com o termo de busca apresentaram um incremento, passando de 42% para 46%, com 22 de um total de 48 passagens recuperadas (Gráfico 1).

É importante observar que, com a utilização do operador AND, o total de passagens recuperadas sofreu redução, de 143 para 48, porém, a quantidade de passagens que apresentavam baixa e média relação com os termos de busca também sofreu redução, passando de 72 para 7, o que melhorou a qualidade da recuperação, diminuindo o risco de que informações erradas sejam escritas na documentação de projeto, como se pode notar no Gráfico 1.

Gráfico 1: Relação entre o número de passagens recuperadas e sua classificação antes e depois do uso do operador AND, para o SE-Telecom.



Percebe-se que a recuperação com o uso do operador AND teve melhores resultados quando comparados àqueles sem esse operador. Isso se deve à necessidade, quando se utiliza o operador, de que as passagens devem possuir todas as palavras do termo de busca, ou seja, ocorre a redução do número de passagens recuperadas, porém, há um incremento da sua relevância em relação ao termo de busca. Diferente de quando não se usa o operador, onde quaisquer passagens que contenham uma das palavras do termo de busca serão recuperadas como sendo de interesse, desta forma, retornando grande quantidade de passagens que não têm correspondência com o significado do nome do elemento utilizado no termo de busca.

Finalizando, outro fator que contribuiu para a qualidade das passagens recuperadas está em como o nome do elemento, do qual se retirou o termo de busca, foi escrito. Nomes mais explicativos e com maior número de palavras retornaram menos passagens, porém, com melhor relação para com os termos de busca.

Quanto à precisão e revocação, Tabela 5, nos permite verificar que aproximadamente 40% das passagens relevantes foram recuperadas (precisão), enquanto que a proporção de passagens relevantes é de 51% (revocação) do total de passagens. Isto significa que das 155 passagens, 79 delas são relevantes e que o Lucene conseguiu recuperar 40% delas, ou seja, 32 passagens relevantes foram recuperadas.

4.3.3. Capacidade de geração de comentários

A análise dos resultados para a capacidade de geração de comentários está na sub seção seguinte, pois para explicar os resultados do grau de comentários é necessário utilizar os dados de geração de comentários.

4.3.4. Grau de comentários

O grau de comentários faz a correlação entre a quantidade de comentários pelo número de elementos de código fonte encontrados. Dessa forma pode-se ter a dimensão de quão comentado está o código de cada um dos projetos. A relação entre o grau de comentários antes do projeto ser analisado pelo Comente+ e após sua análise mostra que houve um aumento no número de comentários, vide Tabela 8 dos resultados. Isso se deve ao fato de que o Comente+, ao encontrar um elemento de código, classe, método ou variável, sem comentário, faz a inserção de um novo comentário para esse elemento.

Quanto maior o número de comentários escritos pelo desenvolvedor, maior será o valor do seu grau de comentários, sendo que o mesmo acontece quando o Comente+ insere comentários nos códigos fonte. Por exemplo, observando a Tabela 8, o grau de comentários do projeto Emoções evoluiu de 3,3 para 4,2. Isto se deve ao fato de a quantidade inicial de comentários, representada por 85 comentários, ter passado a 110 depois da inserção de 25 comentários pelo Comente. Observa-se que a maioria dos projetos analisados teve o número de comentários inseridos com valor superior ao número de comentários escritos pelo desenvolvedor, conforme o Gráfico 2.

Gráfico 2: Comparação entre os números de comentários por projeto.

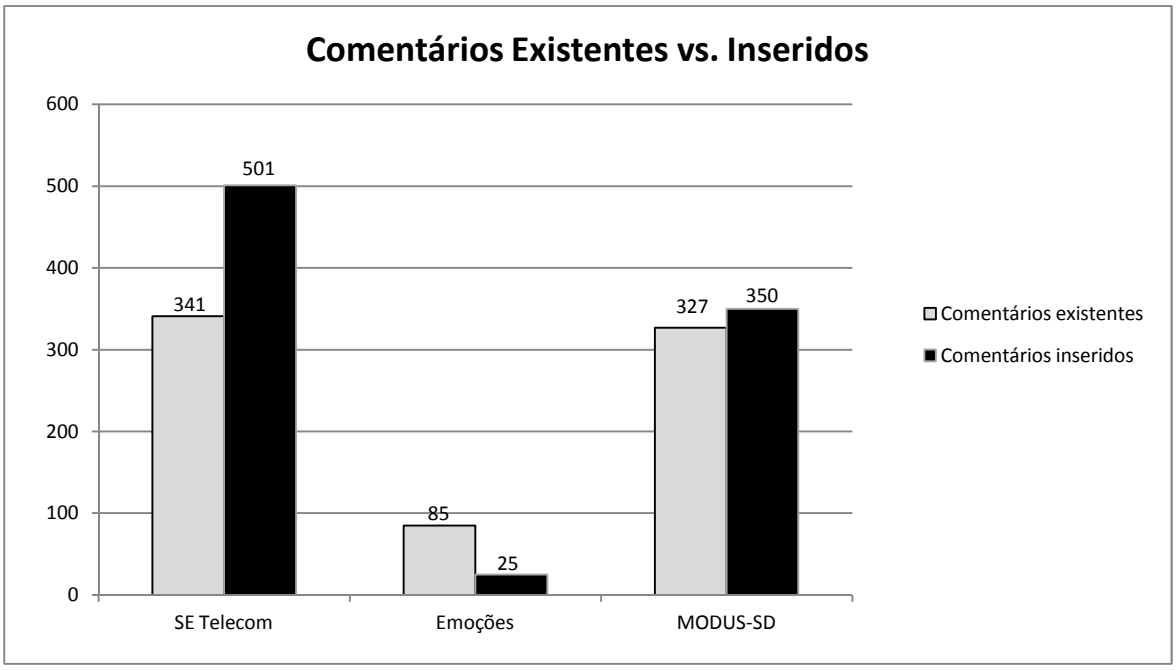
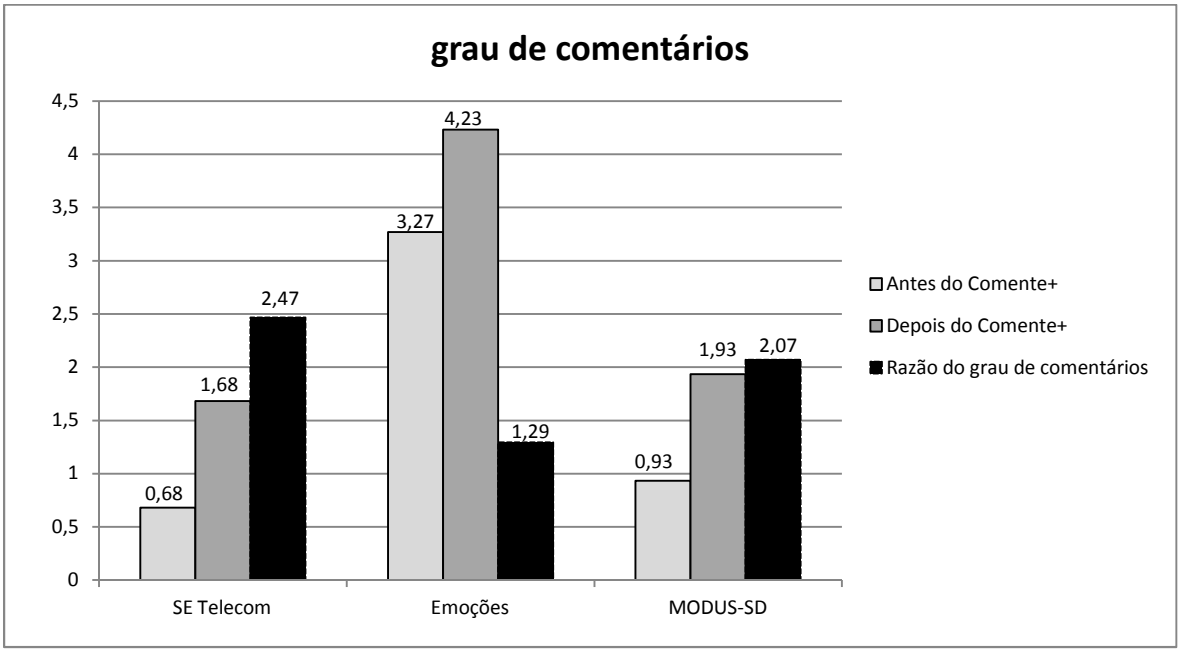


Gráfico 3: Comparação do grau de comentários antes e depois do Comente+.



NoGráfico 3~~Gráfico 3~~, percebe-se pela razão do grau de comentários que os projetos com maior incremento são o Comente+, com razão 2,66, o SE-Telecom, 2,47 e o projeto MODUS-SD, que apesar de ter poucos elementos de código fonte, apresentou razão igual a 2,07. Sem a razão do grau de comentários, tem-se a impressão de que o método proporcionou um incremento constante para todos os projetos.

Capítulo 5

Conclusão e Trabalhos Futuros

Conclusão

Os resultados mostraram que o Comente+ tem potencial como auxiliar na documentação de códigos fonte. Com a recuperação de informações, utilizando expressões regulares sobre os códigos fonte de um projeto de software, o Comente+ consegue fazer a identificação e a extração de informações dos elementos considerados relevantes com 100% de acertos. Eventuais distorções que podem ser facilmente corrigidas com a análise e modificação dos padrões de expressões regulares utilizados, para se adequarem aos padrões da gramática que define a linguagem Java. Tais modificações, caso necessárias, poderão ser feitas pelo próprio usuário do sistema.

A criação e atualização de comentários para os elementos relevantes dos códigos fonte, depois da análise pelo método, representaram um aumento situado entre 29% a 166% com relação aos comentários escritos pelo desenvolvedor.

Também pelo uso da API do Lucene, foi possível identificar passagens na documentação do projeto que contribuíram com a melhoria dos comentários, permitindo explicá-los com informações já existentes. Por exemplo: 17,7% do total de elementos do projeto SE-Telecom tiveram seus comentários melhorados com as passagens recuperadas.

Algumas melhorias podem produzir resultados mais precisos e melhores, principalmente na qualidade e quantidade das passagens recuperadas pelo Lucene (12,2% para o projeto SE-Telecom). Estas melhorias terão influência direta sobre os comentários criados ou atualizados pelo sistema, não se restringindo a gerar ou atualizar os comentários somente com informações do próprio elemento em análise.

O Documento de Referência não pode ser experimentado de maneira a gerar resultados a fim de se concluir sua eficácia, porém, mostrou que sua utilização proporciona maior organização e acesso facilitado às informações de projeto.

Trabalhos Futuros

De imediato, pretende-se melhorar o código do projeto, buscando simplificá-lo, tornando-o mais ágil e preciso. Também se pretende realizar mais testes com o uso do documento de referência.

Durante o desenvolvimento surgiram ideias de funcionalidades para o sistema. Abaixo estão listadas aquelas consideradas mais interessantes:

- aproveitando o trabalho de mestrado do colega Geraldo Boz Junior (Boz, 2011), criar um agente instigador que trabalhará em conjunto com o sistema de documentação, enviando mensagens para os desenvolvedores perguntando se um determinado elemento do código fonte, escrito por ele, poderia ser descrito de acordo com uma sugestão do próprio instigador. Por exemplo, o instigador, de posse das passagens do descritivo do sistema ou Documento de Referência, faria uma escolha, aleatória ou seguindo algum critério predefinido, de uma dessas passagens e perguntaria ao desenvolvedor se ela explica algum ou uma lista de elementos do código fonte.

- o sistema pode também enviar e-mails, solicitando ou informando ao desenvolvedor o estado do projeto, quais classes foram implementadas, se o cronograma está sendo cumprido, dentre outras informações que poderiam ser identificadas, por mineração de texto ou mesmo por outras análises das informações já recuperadas.
- utilizar as informações do banco de dados para verificar as implementações de código recuperadas para utilizá-las para reuso de código.

Referências

Baeza-Yates, R. e Ribeiro-Neto, B., *Modern information retrieval*, Addison-Wesley, primeira edição, ISBN-10: 020139829X, 1999.

Boz Jr., Geraldo, *Noctua: Uma Ferramenta para Construção Colaborativa de Conhecimento com um Catalisador Virtual*, Dissertação de Mestrado, PPGIa/PUCPR, 2011.

Callan, J. P., *Passage-Level Evidence in Document Retrieval*, In: Proceedings of the 19th ACM Conference on Research and Development in Information Retrieval (SIGIR), Zurich, Switzerland, 1996.

Campagnolo, B., Tacla, C. A., Paraiso, E. C., Sato, G. e Ramos, M. P., *An architecture for supporting small collocated teams in cooperative software development*. In: Proceedings of the 13th IEEE Computer Supported Cooperative Work in Design, p. 264-269, Santiago, Chile, 2009.

Cardoso, O. N. P., *RI*. UFLA – Universidade Federal de Lavras DCC – Departamento de Ciência da Computação, Lavras, MG, 2002.

Cook , C. e Churcher, N., *Modelling and Measuring Collaborative Software Engineering*, In: Proceedings of ACSC2005: Twenty-Eighth Australasian Computer Science Conference, volume 38 of Conferences in Research and Practice in Information Technology, Newcastle, Australia2005.

Crochemore, M., Hancart, C., *Automata for matching patterns*, Handbook of formal languages, 2:399–462, 1997.

Curitiba, Prefeitura Municipal de Curitiba, Informações Socioeconômicas, Guia do Investidor, Agência Curitiba de Desenvolvimento S/A, 3ª edição, Curitiba, PR, 2010.

Dantas, S., *Introdução à RI*, Universidade Salgado de Oliveira, Recife, PE, 2002.

Fay, Robert J., *Full-Text Information Retrieval*, 64 Law Libr. J. pg. 167, 1971.

Ferneda, E., *RI: Análise sobre a contribuição da Ciência da Computação para a Ciência da Informação*, Universidade de São Paulo, São Paulo, SP 2003.

Gosling, J., Joy, B., Steele, G., Bracha, G. The Java Language Specification, Third Edition. Addson-Wesley, Santa Clara, USA, 2005.

Java.net. *A brief history of the Green Projet*, disponível online em https://duke.dev.java.net/green/, acessado em 04 de maio de 2010.

Javadoc. *The Java API Documentation Generator*, disponível online em http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/javadoc.html, acessado em 07 de maio de 2010.

Jiang, T., Ying, J., Wu, M. Fang, M., *An Architecture of Process-centered Context-aware Software Development Environment*, In: Proceedings of the 10th Computer Supported Cooperative Work in Design, pp. 1 - 5, Nanjing, China, 2006.

Jonathan, G., *Computer-Supported Cooperative Work: History and Focus*, University of California, Irvine, 1994.

Jones, K., Hobbs, J. *Practical Programming in Tcl and Tk*, 4th Edition, Brent Welch, 2003.

Lancaster, F. W. *Indexação e resumos: teoria e prática*. Editora Briquet de Lemos, pp. 452, Brasília, DF, 1993.

Land, S.K., Walz, J.W., *Practical Support for ISO 9001 Software Project Documentation*, IEEE Computer Society, New Jersey, EUA, 2006.

Laryd, A., Orci, T., *Dynamic CMM for Small Organizations, Measurement*. Umeå University, Umeå, Sweden, 2000.

Maarek, Y.S., Berry, D.M., Kaiser, G.E., *GURU: Information Retrieval for Reuse*, Landmark Contributions in Software Reuse and Reverse Engineering, pp. 1-28,1994.

Martinazzo, B. *Um Método de Identificação de Emoções em Textos Curtos Para o Português do Brasil*. 2011. 61 f. Dissertação (Mestrado em Informática) - Programa de Pós-Graduação em Informática, Pontifícia Universidade Católica do Paraná, Curitiba, 2011.

Massoni, T., Sampaio, A., Borba, P., Freire, A. L., *A RUP-Based Software Process Supporting Progressive Implementation*, *IGI Publishing*, Hershey, EUA, 2003.

Martha, A.S., *RI em campos de texto livre de prontuários eletrônicos do paciente baseada em semelhança semântica e ortográfica*, dissertação de mestrado apresentada à Escola Paulista de Medicina, Unifesp, São Paulo, SP, 2005.

McCandless, M., Hatcher, E., Gospodnetić, O., Lucene in Action, segunda edição, Manning Publications Co., 2010.

Michel, B., *Computer Supported Co-operative Work*. Université de Paris, Paris, 1999.

Moeckel, A. *CSCW: conceitos e aplicações para cooepração*, Centro Federal de Educação Tecnológica do Paraná, CEFET-PR, Curitiba, 2003.

OMG, *Unified Modeling Language (UML), version 2.0*, disponível on-line em http://www.omg.org/technology/documents/formal/uml.htm, acessado em 09/05/2010.

Paduelli, M. M. e Sanches, R., *Problemas em manutenção de software: caracterização e evolução,* In: Anais do III Workshop de Manutenção de Software Moderna - WMSWM, pág. 1, Vila Velha, ES, 2006.

Pollice, G., Augustine, L., Lowe, C. Madhur, J., *Software Development for Small Teams: A RUP-Centric Approach*., Addison-Wesley, ISBN-10: 0321199502, pp. 272, 2004.

Rech, J., *Preprocessing of Object-Oriented Source Code for Code Retrieval*, Citeseer, 2005.

Sales, R.D., Viera, A. F. G., *Grupos e linhas de pesquisa sobre recuperação da informação no Brasil*, Revista Biblios, número 28, junho de 2007.

Sarma, A., Noroozi, Z., Van der Hoek, A., *Palantir: Raising Awareness Among Configuration Management Workspaces,* In: Proceedings 25th International Conference on Software Engineering (ICSE), 2003.

Shiki, N., Ohno, Y., Fujii, A., Murata, T., Matsumura, Y., *Unified Modeling Language (UML) for hospital-based cancer registration processes*. Asian Pacific journal of cancer prevention, APJCP, 2004.

Silva, E. F. A., Barros, F. A., Prudêncio, R. B. C., *Uma Abordagem de Aprendizagem Híbrida para Extração de Informação em Textos Semi-Estruturados*. In: Proceedings of the XXV Congresso da Sociedade Brasileira de Computação - SBC 2003, São Leopoldo, 2003.

Sindhgatta, R., *Using an information retrieval system to retrieve source code samples*. In: Proceedings of the 28th international conference on Software engineering - ICSE '06. New York, ACM Press, 2006.

Souza, S. C. B. de; Avila, J. L. B. ; Avila, J. A. P.; François, N. P.; Oliveira, K. M.; Figueiredo, R. M. C., *Investigação da Documentação de Maior Importância para Manutenção de Software*. In: Proceedings of the 4as Jornadas Iberoamericas en

Ingenieria del Software e Ingenieria del Conocimiento, Madrid, Servicio de Publicaciones de La Facultad de Informática de La U.P.M. v. I. p. 217-230, 2004.

Sun Microsystems. *Code conventions for the Java programming language. Technical report*. Santa Clara, 1999.

Story, M. A., Cheng, L. T., Bull, I. Rigby, P. *Shared Waypoints and Social Tagging to Support Collaboration in Software Development*. In: Proceedings of the CSCW 2006, Alberta, Canada, pp. 195-198, 2006.

Tecpar, *Sistema Especialista de Monitoramento de Alarmes - SE-Telecom*. Projeto em desenvolvimento - Centro de Engenharia de Sistemas Inteligentes. Instituto de Tecnologia do Paraná - Tecpar, Curitiba, 2011.

Thompson, K., *Regular expression search algorithm*, Communications of the ACM, 11(6):419–422, 1968.

Wanderley, G. M. P. *Especificação do Modelo do Participante de Um Projeto de Desenvolvimento Colaborativo de Software*. Relatório de bolsa de iniciação científica PIBIC/CNPq - Engenharia de Computação. Pontifícia Universidade Católica do Paraná, Curitiba, 2011.

Apêndice A

Lucene

Apêndice B

Resultados da Recuperação de Informação na Documentação de Projeto utilizando o Lucene.