UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO



A INFLUÊNCIA DOS PARÂMETROS DE ANÁLISE POR SEMÂNTICA LATENTE APLICADA A LOCALIZAÇÃO DE DEFEITOS DE SOFTWARE

ALLYSSON COSTA E SILVA

Uberlândia - Minas Gerais
2011

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO



ALLYSSON COSTA E SILVA

A INFLUÊNCIA DOS PARÂMETROS DE ANÁLISE POR SEMÂNTICA LATENTE APLICADA A LOCALIZAÇÃO DE DEFEITOS DE SOFTWARE

Dissertação de Mestrado apresentada à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como parte dos requisitos exigidos para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Engenharia de Software.

Orientador:
Professor Doutor Marcelo de Almeida Maia

Uberlândia, Minas Gerais
2011

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da UFU

S586i Silva, Allysson Costa e, 1979-
A influência dos parâmetros de análise por semântica latente aplicada a localização de defeitos de software [manuscrito] / Allyson Costa e Silva. - 2011.
123 f. : il.

Orientador: Marcelo de Almeida Maia.

Dissertação (mestrado) – Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação.
Inclui bibliografia.

1. Software - Teses. 2. Software - Desenvolvimento - Teses. 3. Recuperação da informação - Teses. I. Maia, Marcelo de Almeida. II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU: 681.3.06

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Computação a aceitação da dissertação intitulada "**A influência dos parâmetros de Análise por Semântica Latente aplicada a localização de defeitos de software**" por **Allysson Costa e Silva** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação.**

Uberlândia, 18 de Fevereiro de 2011

Orientador:

_____
Professor Doutor Marcelo de Almeida Maia
Universidade Federal de Uberlândia

Banca Examinadora:

_____
Professor Dr. Eduardo Magno Lages Figueiredo
Universidade Federal de Minas Gerais

_____
Professor Dr. Michel dos Santos Soares
Universidade Federal de Uberlândia

_____
Professor Dr. Ilmério Reis da Silva
Universidade Federal de Uberlândia

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Data: Fevereiro de 2011

Autor:	**Allysson Costa e Silva**
Título:	**A influência dos parâmetros de Análise por Semântica Latente aplicada a localização de defeitos de software**
Faculdade:	**Faculdade de Computação**
Grau:	**Mestrado**

Fica garantido à Universidade Federal de Uberlândia o direito de circulação e impressão de cópias deste documento para propósitos exclusivamente acadêmicos, desde que o autor seja devidamente informado.

_____

Autor

O AUTOR RESERVA PARA SI QUALQUER OUTRO DIREITO DE PUBLICAÇÃO DESTE DOCUMENTO, NÃO PODENDO O MESMO SER IMPRESSO OU REPRODUZIDO, SEJA NA TOTALIDADE OU EM PARTES, SEM A PERMISSÃO ESCRITA DO AUTOR.

©Todos os direitos reservados a Allysson Costa e Silva

Dedicatória

Primeiramente a Deus, pai criador e motivador de toda a ciência.
A minha família, incluindo minha noiva Lucimarly, sem os quais jamais teria alcançado este estágio da vida.
Em especial, à minha querida avó Brígida Maia Costa (in memorian), pelo carinho e amor recebido.
Aos professores da Universidade Federal de Uberlândia, em especial ao professor Doutor Marcelo de Almeida Maia, pela paciência e dedicação em ensinar, aperfeiçoar e colaborar para formação do ser humano.
Aos amigos e colegas de Universidade pelo apoio nas horas difíceis.
À Secretaria Estadual de Educação de Minas Gerais, em especial a Superintendência Regional de Ensino de Montes Claros, pelo apoio incondicional à realização deste trabalho.

Resumo

Durante seu ciclo de vida, um sistema de software sofre modificações contínuas para que defeitos sejam corrigidos e haja um bom alinhamento entre os requisitos dos usuários e as funções implementadas no sistema. Tais modificações, implementadas durante a fase de manutenção de software, consomem grande quantidade de recursos e são influenciadas pelo nível de compreensão do programa apresentado pelo desenvolvedor. Desta forma, a produção e o aperfeiçoamento de ferramentas ligadas à melhoria da compreensão do sistema agiliza e minimiza o trabalho realizado durante a manutenção de software. Neste trabalho foi desenvolvida uma ferramenta para recuperar a rastreabilidade entre a documentação do sistema (descrições de defeitos) e o código fonte baseada em uma técnica de Recuperação de Informação. O objetivo principal do trabalho foi analisar a influência da variação dos valores de alguns parâmetros da técnica Indexação por Semântica Latente (*Latent Semantic Indexing*) - LSI - sob o ponto de vista de acurácia e desempenho. A acurácia foi medida com base no número de métodos que o desenvolvedor deveria verificar para localizar um defeito. O estudo permitiu entender como diferentes valores de parâmetros podem influenciar nos resultados das consultas, os quais tem impacto direto na redução dos esforços de manutenção. Os valores de parâmetros predominantes foram: redução de dimensionalidade k=300; o uso de funções de peso para nomes de métodos e classes, bem como para fragmentos de nomes de métodos e classes; o uso dos filtros de código fonte baseados em rastros de execução (ET) ou baseados em termos de consultas associadas a rastros de execução (QTL).

Como resultado adicional, foi possível a identificação de alinhamentos semânticos entre o vocabulário do usuário e nomes de classes e métodos utilizados.

Palavras chave: defeitos de software, recuperação de informação, parâmetros de busca, código fonte, indexação por semântica latente.

Abstract

During its life cycle, software systems must pass through continuous change to have bugs fixed and get a reasonable conformance between user requirements and implemented software functions. The necessary effort to execute such changes that occur in the software maintenance phase is considerable and is influenced by the developer software comprehension level. In this way, the production and improvement of tools related to software comprehension can speed up and minimize software maintenance efforts. In this work, a tool to retrieve the traceability links between documentation (bug descriptions) and source code was implemented using an Information Retrieval technique. The main purpose of this work is the analysis of the influence of Latent Semantic Indexing (LSI) parameters values on accuracy and performance of the tool. The accuracy was measured with the number of methods that should be checked by the developer to find a bug. This study has shown that the parameter's values have direct impact on software maintenance accuracy and performance. The predominant parameter's values were: dimensionality reduction k=300; positive use of weighting functions for method and class names and for method and class name fragments; use of ET or QTL source code filters. Additionally, it was possible to identify semantic alignments between the user vocabulary and method and class names.

Keywords: bugs, information retrieval, search parameters, source code, latent semantic indexing.

Sumário

Lista de Figuras **x**

Lista de Tabelas **xii**

Lista de Abreviaturas e Siglas **xiii**

1 Introdução **14**

1.1 Exposição do Problema 15
1.2 Hipóteses 16
1.3 Objetivos 17
1.4 Justificativa 18
1.5 Visão Geral da Dissertação 19

2 Referencial Teórico **21**

2.1 Manutenção de Software 21
2.1.1 Tipos de Manutenção de Software 22
2.1.2 Manutenção Corretiva de Software 22
2.1.3 Recuperação de Informação no Contexto da Manutenção de Software 23
2.2 Rastreabilidade entre Artefatos de Software 25
2.3 O uso de Análises Estática e Dinâmica de Código Fonte para Recuperação de Rastreabilidade 27
2.4 LSI - Indexação por Semântica Latente 28
2.4.1 Utilização de Função de Peso Específica na Matriz de Termos Documentos 30
2.4.2 Medidas de Avaliação usadas em Técnicas de RI 31
2.4.3 Outras Técnicas de RI propostas para Localização de Defeitos, Conceitos e Características 33
2.5 Resumo do Referencial Teórico 33

3 Metodologia de Trabalho **35**

3.1 O Processo de Localização de Defeitos 35
3.2 O Processo de Formulação da Consulta 37

3.3 Parâmetros da Técnica LSI 39
3.4 Os Critérios de Escolha dos Defeitos e Versões de Software Analisados . . . 46
3.5 Ferramentas Utilizadas para o Trabalho 47
3.6 Resumo da Metodologia 49

4 Resultados **50**
4.1 Classificação de Combinações Baseada na Quantidade de Métodos Analisados 51
4.2 A Combinação Predominante 56
4.2.1 Efeitos da Aplicação de Diferentes Conjuntos de *stop-words* sobre a Manutenção de Software 62
4.2.2 Comparando a Combinação Predominante com as Melhores Combinações 62
4.3 Análise das Abordagens para Filtragem de Código Fonte 63
4.4 Análise Estatística Complementar para os Filtros de Código Fonte e Função de Peso 66
4.4.1 Diferenciação entre os filtros de código fonte ET e QTL 67
4.4.2 Proposição de Alinhamentos Semânticos com base na Aplicação da Função de Peso 68
4.4.3 Análise do Nome dos Métodos e seus Fragmentos 69
4.4.4 Análise do Nome das Classes e seus Fragmentos 71
4.5 Problemas e Limitações do Trabalho Proposto 73
4.6 Considerações Finais sobre os Resultados 73

5 Trabalhos Relacionados **75**
5.1 Abordagens de Análise Estática e Dinâmica 76
5.2 Transformações da Matriz de Termos-Documentos 79
5.3 Localização de Defeitos a partir de Técnicas de Recuperação de Informação 80
5.4 Outros parâmetros estudados - *Stop-words* e redução de dimensionalidade k 80

6 Conclusões **82**

Referências Bibliográficas **85**

A Aplicação da metodologia para formulação da consulta **90**

B Telas do sistema administrativo **114**

C Armazenamento dos dados referentes às amostras de pesquisa e aos resultados encontrados **116**

D Comparativo entre os software JEdit e ArmedBear-J **119**

Lista de Figuras

2.1 Decomposição da matriz de termos-documentos 29
2.2 Aplicação de LSI para redução de dimensionalidade 29
2.3 Definição das medidas de precisão e revocação 32

3.1 O processo de localização de defeitos 36
3.2 O processo de formulação da consulta 38
3.3 Abordagens para filtragem do código fonte 41
3.4 Teste utilizando todo o código fonte e um defeito do ArgoUML 45
3.5 Log do teste com todo o código fonte de um defeito do ArgoUML 45
3.6 Teste utilizando o filtro QTL e um defeito do ArgoUML 46

4.1 Combinações possíveis para os parâmetros estudados 51
4.2 Diagrama de atividades para busca de valores predominantes de parâmetros 57
4.3 Combinações possíveis fixando-se o filtro de código fonte e o valor de k . . 57
4.4 Gráfico Precisão versus k para 10 bugs do ArgoUML 58
4.5 Trabalho realizado por valor de k 59
4.6 Predominância dos valores de k - abordagens ET e QTL respectivamente . 59
4.7 Predominância na aplicação da função de peso sobre o nomes dos métodos - abordagens ET e QTL respectivamente 60
4.8 Predominância na aplicação da função de peso sobre os fragmentos do nomes dos métodos - abordagens ET e QTL respectivamente 60
4.9 Predominância na aplicação da função de peso sobre o nomes das classes - abordagens ET e QTL respectivamente 60
4.10 Predominância na aplicação da função de peso sobre os fragmentos do nomes das classes - abordagens ET e QTL respectivamente 60
4.11 Predominância de *stop-words* - abordagens ET e QTL respectivamente . . 61
4.12 Número de documentos no corpus 64
4.13 Tempo necessário para indexar e processar o teste 64
4.14 Precisões obtidas nos testes com o ArmedBear-J e JEdit 65
4.15 *Boxplot dos resultados dos testes do ArgoUML para cada filtro* 68
4.16 *Boxplot dos resultados dos testes do JEdit para cada filtro* 68

5.1 Temáticas utilizadas no presente trabalho 75

B.1 Cadastro de código fonte 114
B.2 Cadastro de defeitos 114
B.3 Realização de testes individuais, por defeito ou por lote de defeitos 115
B.4 Emissão de gráficos e relatórios 115

C.1 Diagrama Entidade-Relacionamento do banco de dados utilizado 118

Lista de Tabelas

3.1 Exemplo da aplicação da metodologia para formulação da Consulta 39
3.2 Descrições de defeitos e revisões de software associadas 47

4.1 *Classificação das Combinações para o ArgoUML - Primeiro Método - QTL* 53
4.2 *Classificação das Combinações para o JEdit - Primeiro Método - QTL* . . . 53
4.3 *Classificação das Combinações para o ArgoUML - Primeiro Método - ET* . 54
4.4 *Classificação das Combinações para o JEdit - Primeiro Método - ET* 54
4.5 *Classificação das Combinações para o ArgoUML - Todos os Métodos - QTL* 54
4.6 *Classificação das Combinações para o JEdit - Todos os Métodos - QTL* . . 54
4.7 *Classificação das Combinações para o ArgoUML - Todos os Métodos - ET* . 55
4.8 *Classificação das Combinações para o JEdit - Todos os Métodos - ET* . . . 55
4.9 Teste de Wilcoxon com o JEdit - função de peso aplicada sobre nomes de métodos 70
4.10 Teste de Wilcoxon com o JEdit - função de peso aplicada sobre fragmentos dos nomes de métodos 70
4.11 Teste de Wilcoxon com o ArgoUML - função de peso aplicada sobre os nomes de métodos 70
4.12 Teste de Wilcoxon com o ArgoUML - função de peso aplicada sobre os fragmentos dos nomes de métodos 71
4.13 Teste de Wilcoxon com o JEdit - função de peso aplicada sobre nomes de classes 71
4.14 Teste de Wilcoxon com o JEdit - função de peso aplicada sobre fragmentos dos nomes de classes 72
4.15 Teste de Wilcoxon com o ArgoUML - função de peso aplicada sobre nomes de classes 72
4.16 Teste de Wilcoxon com o ArgoUML - função de peso aplicada sobre fragmentos dos nomes de classes 72

5.1 Comparação de trabalho relacionado - [Marcus et al. 2004] 77
5.2 Comparação de trabalho relacionado - [Poshyvanyk et al. 2007] 78

Lista de Abreviaturas e Siglas

ET	Execution Trace - Filtro de código fonte
QTL	Query Term Like - Filtro de código fonte
RI	Recuperação de Informação
LSI	Latent Semantic Indexing
LSA	Latent Semantic Analysis
NPE	Null Point Exception
MN	Method Name
FMN	Fragment of Method Name
CN	Classe Name
FCN	Fragment of Classe Name
STW	Stop-word
ESTW	English Stop-word
JSTW	Java Stop-word
ALLSTW	English e Java Stop-word
LDA	Latent Dirichlet Allocation
MWW	Mann-Whitney-Wilcoxon

Capítulo 1

Introdução

Durante seu ciclo de vida, um sistema de software deve estar em constante modificação para que defeitos sejam corrigidos e haja um bom alinhamento entre as necessidades dos usuários finais e as funções implementadas no código fonte. Na prática tais modificações podem ser motivadas por diversos fatores, como por exemplo: mudanças nos requisitos, correção de erros, otimização de determinados módulos do código fonte, portabilidade para novas arquiteturas de hardware e adequação às tecnologias surgidas no ambiente operacional onde o software é utilizado [Grubb e Takang 2003]. Tais atividades estão englobadas dentro das atividades de manutenção de software.

Os desenvolvedores podem incorrer em erros quando realizam a introdução de mudanças no sistema, principalmente se a estrutura e o comportamento do sistema não são bem compreendidos ou quando não é possível prever completamente o impacto trazido por tais mudanças [Riebisch 2004]. A necessidade de se trabalhar em software legado, com milhares de linhas de código, documentação insuficiente e desatualizada, é uma situação comum aos desenvolvedores envolvidos na tarefa de compreensão de sistema.

Ainda que se tenha documentação disponível, a falta de rastreabilidade para outras partes do sistema, como o código fonte, dificulta o trabalho do desenvolvedor. A inexistência ou desatualização da documentação torna a implementação do código fonte do software um recurso capital para algumas empresas, já que este representa a fonte mais fidedigna de como são conduzidos os processos dentro da empresa [Nelson 1996]. Neste estágio, o código fonte e sua instrumentação passam a ser fundamentais para entender a dispersão e relacionamento dos conceitos do domínio do problema utilizados na implementação do sistema. Desta forma, alguns trabalhos basearam-se na execução de características com o objetivo de localizar interesses transversais no código fonte e apoiar tarefas de manutenção de software como a localização de defeitos [Sobreira e Maia 2008], [Poshyvanyk et al. 2007].

Conforme definido em [Ramamoorthy e Tsai 1996], falhas e erros de sistema são resultado de erros humanos durante o processo de desenvolvimento. No entanto, produzir um sistema imune a defeitos em suas funções é inviável. Existe portanto a necessidade de se criar mecanismos que documentem os defeitos e permita a interação entre quem descobre e

quem corrige tais defeitos. O custo de manutenção de sistemas legados torna-se acentuado em função da complexidade e tamanho dos sistemas [Ramamoorthy e Tsai 1996]. Esta manutenção pode ser minimizada a partir do constante aperfeiçoamento das ferramentas e técnicas utilizadas com a finalidade de melhorar a compreensão de programas. Neste sentido, algumas iniciativas realizadas em projetos de software livre podem ser citadas:

- código fonte e suas respectivas mudanças são gravados em sistemas de versionamento de código disponíveis na Internet;
- defeitos são reportados em sistemas *on-line* responsáveis por sua triagem;
- discussões sobre o software são arquivadas em listas de e-mail, blogs e outros meios digitais.

O trabalho corrente utiliza repositórios de defeitos de três sistemas de código aberto, como substrato para obtenção de amostras para a pesquisa desenvolvida. A intenção é prover uma solução automatizada para auxílio à localização de defeitos no código fonte e estudar os parâmetros de entrada da técnica de busca utilizada na solução sob os aspectos de acurácia e performance.

1.1 Exposição do Problema

Como exposto em [Pashov et al. 2004], existe uma lacuna entre os diferentes níveis de abstração presentes no código fonte, documentação e arquitetura de software. O conjunto de vínculos entre os artefatos citados são importantes para a compreensão do sistema, mas nem sempre estão documentados, o que requer sua recuperação.

A dificuldade em se estabelecer a rastreabilidade entre os diversos artefatos produzidos no ciclo de vida do sistema é um dos fatores que encarece a manutenção de software. Durante a manutenção o desenvolvedor necessita compreender o sistema para introduzir mudanças relativas às alterações de requisitos ou correção de defeitos. Segundo [Ramesh 1998], a habilidade de se realizar a análise de impacto de mudanças está diretamente relacionada às informações relativas à rastreabilidade. Neste ponto, a documentação sobre os artefatos de software e seus inter-relacionamentos permite ao desenvolvedor realizar uma análise de impacto de mudanças, entender a origem e comportamento dos componentes que compõem o sistema atual além de sustentar processos de manutenção corretiva de software fornecendo informações necessárias à reprodução de falhas. Portanto, a inexistência de rastreabilidade, entre documentação sobre os erros do sistema e o código fonte alterado a cada nova versão de software, constitui um desafio a ser superado por novos desenvolvedores a cada revisão de software.

Processos de recuperação de informação, aplicados à manutenção de software, se beneficiam do conteúdo semântico embutido nas descrições de defeitos do sistema para

formulação de consultas. Utilizar a rastreabilidade entre documentação e código fonte, para a correção de um defeito apresentado pelo software, é um desafio que tem levado a comunidade científica a desenvolver ferramentas automatizadas com o intuito de reduzir os custos da manutenção de software [Andrian Marcus 2003]. Resumindo, o problema geral estudado neste trabalho é a falta de mapeamento de conceitos embutidos nas descrições de defeitos de software, expressos quase sempre em linguagem natural, para as partes relevantes do código fonte que necessitam ser corrigidas.

O uso de técnicas de Recuperação de Informação - RI - para recuperar a rastreabilidade entre documentação e código fonte tem sido aplicada com relativo sucesso [Lukins et al. 2008], [Antoniol et al. 2000]. Neste sentido, um problema que surge é a dificuldade de se ajustar a configuração da técnica de RI escolhida. Neste trabalho são estudados os seguintes parâmetros de busca: o valor do parâmetro de redução de dimensionalidade do espaço algébrico matemático, a função de peso para transformação das matrizes de frequência de termos-documentos, o conjunto de *stop-words* (palavras com pouco valor semântico) e filtros para o código fonte. Em resumo, a prescrição de valores satisfatórios para este conjunto de parâmetros é um problema inerente à recuperação da rastreabilidade realizada através da aplicação de uma técnica de RI [Nakov et al. 2001], [Nakov et al. 2001]. Tal problema é o alvo de estudo deste trabalho.

1.2 Hipóteses

A carência de ferramentas automatizadas, o suporte precário ao processo de manutenção corretiva de software, o nível de experiência dos desenvolvedores, a complexidade e tamanho dos sistemas atuais são fatores que determinam o esforço despendido em tarefas de manutenção de software [Li et al. 2010]. Adicionalmente, o volume de defeitos apresentados em grandes projetos de software livre aliado a necessidade de disponibilização rápida e eficaz de correções para tais defeitos impulsionam o constante aperfeiçoamento das técnicas empregadas na recuperação da rastreabilidade.

A customização de uma técnica de recuperação de informação, através de ajustes em seus parâmetros de entrada, é avaliada neste trabalho dentro do contexto de manutenção de software. Cada parâmetro estudado será avaliado sob sua influência no número de métodos analisados por um desenvolvedor, em uma tarefa de localização de defeitos do código fonte. Os quatro parâmetros estudados neste trabalho foram citados na seção anterior quando relatava-se o problema de se aplicar a técnica de RI. Assim, o valor do parâmetro de redução de dimensionalidade, sobre o espaço algébrico matématico sobre qual atua a técnica de RI, deve ser determinado para cada coleção de documentos provenientes do código fonte. Esta escolha tem implicações significativas sobre o tempo de indexação, busca e quantidade de memória requerida no processo de recuperação da rastreabilidade empreendido [Kontostathis 2007]. Outra questão retrata a transformação das

matrizes de frequência de termos-documentos através de funções de peso, o que exerce importante influência sobre a acurácia dos resultados das buscas [Nakov et al. 2001]. O conjunto de *stop-words* aplicado a coleção de documentos e a redução do número de documentos analisados proporcionada pela utilização de filtros para o código fonte são postos também a prova na tentativa de se encontrar as melhores combinações dos valores destes parâmetros. O estudo experimental das combinações destes parâmetros pode levar a uma diminuição dos custos envolvidos na fase de manutenção do sistema.

Muitas combinações de parâmetros podem ser geradas e testadas na tentativa de se melhorar a performance e acurácia na localização de defeitos que demandem correção. Uma questão que se levanta é a possibilidade de existir uma combinação ideal de valores de parâmetros a ser explicitada de maneira que o desenvolvedor possa se beneficiar em tarefas cotidianas de manutenção de software.

O trabalho corrente identifica nos conceitos do domínio do problema, presentes nas descrições dos defeitos de software, as consultas utilizadas como entrada à técnica de recuperação de informação. A aplicação desta técnica modela a representação semântica de documentos do código fonte e consultas em um espaço algébrico matemático. Até este ponto do trabalho de localização de partes do código fonte com defeito, muitos dos parâmetros citados já devem estar definidos. A escolha dos valores para tais parâmetros define combinações únicas que analisadas em conjunto podem apontar uma tupla de valores para a combinação ideal almejada neste trabalho. Por último, a interpretação das similaridades entre documentos e consulta orientará o desenvolvedor interessado em encontrar métodos relevantes à correção de um defeito. De maneira geral, as hipóteses levantadas para este trabalho estão relacionadas aos parâmetros citados e podem ser assim enunciadas:

- A manipulação de filtros de código fonte, de funções de peso específicas, de conjuntos de *stop-words* diferenciados e do fator de redução do espaço vetorial semântico é benéfica para redução do número de métodos analisados em uma tarefa de localização de defeitos no código fonte quando comparada a uma abordagem sem que os mesmos fatores sejam considerados.

- É possível prover ao desenvolvedor uma calibragem satisfatória de parâmetros de busca no início do processo de recuperação de informação aplicada à localização de defeitos no código fonte de sistemas orientados a objeto.

1.3 Objetivos

Este trabalho mostra uma abordagem para localização de defeitos no código fonte, mensurando-se o trabalho realizado pelo desenvolvedor para tal atividade. Objetiva-se com isso orientar o desenvolvedor na correção de defeitos do sistema diminuindo o

tempo gasto na identificação de quais componentes do sistema necessitam reparação para correção da falha apresentada pelo software.

O trabalho explora a combinação de um conjunto de parâmetros de entrada de uma técnica de recuperação de informação denominada Indexação por Semântica Latente. Tal técnica está baseada na exploração semântica de fragmentos do código fonte que possuem vínculos latentes entre si e entre consultas submetidas pelo usuário. Com este arcabouço constrói-se uma ferramenta que busca automatizar a localização de defeitos em sistemas de software a partir da análise do código fonte.

O objetivo deste trabalho é verificar experimentalmente como a escolha dos valores de tais parâmetros influenciam a performance e acurácia dos resultados obtidos explicitando aqueles valores mais salientes à redução do trabalho de manutenção. Ao iniciar um processo de manutenção um desenvolvedor não possui a calibragem exata para os parâmetros a serem utilizados na técnica de busca escolhida. O trabalho corrente se propõe a evidenciar, através da análise das combinações de parâmetros, a combinação que melhor se adapte em coleções específicas de texto baseadas em código fonte.

1.4 Justificativa

A manutenção é a fase mais cara e trabalhosa do ciclo de vida de um projeto de software, especialmente em sistemas legados de grande complexidade e mantidos por longos períodos de tempo [Shukla e Misra 2008]. Como exposto em [Boehm e Basili 2001], do montante de recursos utilizados durante o ciclo de vida de um projeto de software, aproximadamente 30% é gasto com desenvolvimento e 70% com manutenção. Uma situação ideal é aquela em que os defeitos do software são eliminados durante as fases preliminares à entrada do sistema em produção. Quando um erro não é detectado no início do ciclo de vida do software, problemas e custos podem crescer acentuadamente. Como exemplo, um requisito mal declarado poderá implicar em incorretas decisões de projeto e subsequentes erros no código fonte [Peng e Wallace 1995]. Neste contexto, outras dificuldades podem também ser citadas como aquelas decorrentes da grande rotatividade e dificuldade de recrutamento de desenvolvedores experientes, justificando os altos custos envolvidos na tarefa de manutenção de software [Shukla e Misra 2008]. A utilização de técnicas automatizadas, que auxiliem o trabalho de manutenção, pode ajudar na resolução destes problemas.

Desenvolvedores gastam uma significativa parcela de tempo tentando corrigir defeitos. Alguns fatores são relevantes nesta tarefa, dentre os quais se citam:

- o tempo demandado para detectar quais componentes de software são responsáveis pela ocorrência de uma falha;

- o volume de questões apresentadas nas listas de discussão para correção ou requisição

de melhorias para o software;

- o nível de familiaridade e compreensão do programa que possui o desenvolvedor que será alocado a tarefa de corrigir defeitos.

Em uma tarefa de localização de defeitos de um software, o desenvolvedor usa informações sobre a descrição do defeito e seu contexto de execução para localizar o código fonte relevante. Assim é comum verificar nas descrições do defeito, os passos necessários para reproduzi-lo, mas encontrar o erro no código fonte nem sempre se apresenta como uma tarefa tão trivial. Desenvolvedores empenham considerável esforço realizando esta tarefa. Um desenvolvedor pode lidar com sistemas grandes e complexos onde precisa trabalhar com informações expressas em diferentes níveis de abstração que vão da linguagem natural à complexidade das estruturas de programação expressas no código [Marcus et al. 2004]. Neste caso, a aproximação dos níveis de abstração só se torna possível a partir da junção de conhecimentos de áreas variadas como o domínio do problema, técnicas de programação, idiomas, algoritmos, estruturas de dados, arquitetura de software dentre outros.

Software legado, com documentação deficiente e rotatividade de desenvolvedores em seu ciclo de vida de um software são fatores que tornam dispendiosa a tarefa de compreender o sistema com intuito de repará-lo em seus defeitos. Um desenvolvedor que compreenda bem as funções do sistema pode não mais estar disponível no momento de reparação do software sendo necessário utilizar-se de novos desenvolvedores. Comumente, estes últimos deparam-se com a situação em que a fonte mais confiável e atualizada de informações sobre o sistema é a própria implementação do mesmo. Assim partem para uma análise abrangente do mesmo levando considerável tempo para se familiarizar com as funções do sistema e suas relações com as partes do código fonte.

É reconhecido que os custos de manutenção de sistemas em muitas organizações consome grande parte dos recursos alocados durante todo o ciclo de vida do software [Grubb e Takang 2003]. Isto tem motivado estudos científicos com vistas a automatizar e minimizar as tarefas de manutenção e consequentemente os esforços dos desenvolvedores.

1.5 Visão Geral da Dissertação

Este trabalho apresenta resultados experimentais do uso da técnica de recuperação de informação LSI para análise de código fonte na tentativa de localizar defeitos. O Capítulo 2 apresenta os conceitos necessários que norteiam a aplicação LSI em atividades de manutenção de software, bem como especificidades de parâmetros de entrada utilizados em recuperação de informação. No Capítulo 3, as metodologias empregadas são detalhadas, com enfoque para a geração da consulta, filtragem do código fonte e avaliação dos resultados.

A apresentação e avaliação dos resultados é realizada no Capítulo 4 que apresenta um esboço das tentativas e medições realizadas para se analisar a influência dos parâmetros de busca sobre os resultados obtidos. Adicionalmente, neste capítulo são expostas as deficiências da abordagem proposta bem como a sugestão de trabalhos futuros.

O trabalho corrente explora temáticas relacionadas às combinações de parâmetros de busca utilizados em técnicas de recuperação de informação além das análises estática e dinâmica de código fonte. Tais temáticas estão presentes em diversos trabalhos científicos que exploram o aprimoramento de técnicas se recuperação ou a aplicação destas últimas em tarefas de manutenção de software. Assim, o Capítulo 5 apresenta os trabalhos relacionados em uma abordagem comparativa com a proposta realizada neste trabalho.

Por fim, o Capítulo 6 explora a visão do autor deste trabalho comentando os resultados adicionais providos por este trabalho no que tange ao alinhamento dos vocabulários do código fonte e do usuários.

Capítulo 2

Referencial Teórico

Neste capítulo são destacados alguns fundamentos teóricos relacionados à recuperação de informação aplicada a manutenção de software. No capítulo 5, aparecem os trabalhos relacionados que exploram temáticas derivadas dos conceitos aqui expostos.

2.1 Manutenção de Software

O desenvolvimento de software possui muitas fases. Estas fases incluem o Processo de Negócios, a Engenharia de Requisitos, o Projeto da Arquitetura, o Projeto de Software, a Implementação, Testes, Implantação, Homologação e por fim a Manutenção. Segundo a IEEE [Edelstein e Mamone 1993], a fase de Manutenção pode ser definida como a modificação de um produto de software após a sua implantação para corrigir defeitos, melhoria de performance e outros atributos, além da adaptação do produto para um novo ambiente.

Manutenção é a última fase do ciclo de vida do software. Após a entrada do produto final de software em produção, a fase de manutenção estará iniciada para manter o software atualizado com o ambiente em que é executado e com as alterações das necessidades dos usuários expressas nas mudanças de requisitos, além de sofrer constantes modificações para correção de defeitos. À medida que um produto de software evolui, torna-se gradativamente mais difícil mantê-lo atualizado e funcional caso a manutenção não seja realizada.

Algumas fases do ciclo de vida de um software como a fase de Projeto devem planejar a estrutura do sistema de forma que esta seja facilmente alterável. De forma análoga, a fase de Implementação deve resultar em código fonte com boa legibilidade, facilidade de compreensão e alteração. Destas fases dependem a complexidade e custo do trabalho observado na fase de manutenção.

Em geral, alguns problemas interferem diretamente na facilidade com que é realizada a manutenção de um sistema, por exemplo [Grubb e Takang 2003]:

- a carência de uma arquitetura de software bem definida;
- a falta de padronização e desestruturação do código fonte;
- a inexperiência dos desenvolvedores quanto à compreensão do sistema e domínio do problema;
- a inexistência ou obsolescência da documentação do sistema.

Desta forma, muitos problemas concorrem para que a manutenção de software se torne custosa e complexa. Uma vez iniciada a fase de manutenção esta perdurará até o fim do ciclo de vida do sistema.

2.1.1 Tipos de Manutenção de Software

A flexibilidade nas atividades de manutenção dependerá da minimização de problemas como os citados na seção anterior. A manutenção de software pode ser classificada de acordo com a tarefa executada, conforme definido em [Edelstein e Mamone 1993]:

- manutenção adaptativa - busca adaptar o software para novos ambientes quando o software já foi implantado;
- manutenção perfectiva - provê as atualizações necessárias para que o software atenda a performance e adaptabilidade necessária requisitadas pelas mudanças dos requisitos dos usuários;
- manutenção preventiva - cuida da atualização da documentação e de permitir que o sistema se torne mais flexível à mudanças;
- manutenção corretiva - trata da correção dos defeitos apresentados quando o software já foi implantado.

2.1.2 Manutenção Corretiva de Software

A modalidade corretiva é também conhecida como manutenção tradicional ao passo que as demais podem ser englobadas como evolução de software. No presente trabalho, o enfoque recai sobre a manutenção corretiva de software, especificamente a correção de defeitos. A manutenção corretiva de software lida com a correção de defeitos e falhas encontradas no software. Um defeito pode resultar de erros de projeto, erros lógicos e de código [Erwig e Ren 2007]. A ocorrência de erros de projeto é observada quando mudanças são introduzidas de maneira incorreta e incompleta e são decorrentes de falhas de comunicação entre os projetistas ou da não compreensão da requisição que originou a mudança. Um dos maiores desafios da manutenção de software é o detalhamento dos

efeitos das modificações produzidas em um programa [van der Spek et al. 2008]. O custo total de uma pequena mudança - afetando poucas linhas de código - pode ser extremamente alto. A natureza de alguns tipos de erros mais comuns e suas principais causas são expostos em [Erdil et al. 2003]:

- erros lógicos - resultam de testes inválidos, implementações incorretas de decisões de projeto, fluxos lógicos defeituosos ou testes incompletos com os dados;
- erros de codificação - são causados pela incorreta implementação da lógica do projeto e do código fonte;
- outros erros - erros que podem ser causados no processamento dos dados e decorrentes performances insatisfatórias do sistema.

Os termos "defeitos", "erros" e "falhas" podem trazer confusão, caso não se defina precisamente o que cada termo significa. Em [Zeller 2009], defeito é definido como um código de programa que possua incorreções ("*a bug in the code*"). Erro é definido como a entrada do programa em um estado inconsistente ou incorreto ("*a bug in the state*"). Ainda segundo o autor, falha é um comportamento incorreto do programa que seja visível ("*a bug in the behavior*"). Os conceitos assim enunciados se adéquam a proposta deste trabalho, pois este se dedica a localizar segmentos de código com incorreções (defeitos), utilizando e aprimorando uma técnica de recuperação de informação. A coleta de dados proporcionada pela instrumentação de características em cenários predefinidos levam o sistema a um estado inconsistente (erros) e por consequência à manifestação de comportamentos indesejáveis do sistema (falhas).

2.1.3 Recuperação de Informação no Contexto da Manutenção de Software

Recuperação de Informação (RI) é o campo do estudo responsável por encontrar material (usualmente documentos) de natureza desestruturada (usualmente texto) que satisfaça uma necessidade de informação em grandes coleções de dados armazenadas em computadores [Manning et al. 2008]. Em [Santhiappan et al. 2006], Recuperação de Informação é definida como a ciência que explora, avalia e utiliza avançados métodos matemáticos aplicando-os a uma abordagem qualitativa para a avaliação de relevância de palavras em determinada coleção de dados. No contexto do presente trabalho, sistemas de RI se propõem a descobrir documentos, em um banco de dados textual, que sejam relevantes a uma consulta submetida pelo usuário. Diversas técnicas estatísticas ligadas à área de RI foram desenvolvidas no intuito de se prover um arcabouço ferramental que suportasse as mais variadas atividades ligadas à análise semântica de textos: realização de buscas em coleções de textos [Kontostathis 2007], clusterização ou categorização de

textos [Kuhn et al. 2007], identificação de padrões entre conjuntos de dados [Marcus e Maletic 2001] e outros. A análise por semântica latente (LSA) é uma importante técnica estatística para indexar, recuperar e analisar informação textual usada em diferentes campos da cognição humana [Deerwester et al. 1988], [Nakov et al. 2001] e pertence ao conjunto de técnicas estatísticas referenciado acima. Uma de suas funções é representar o significado de um texto na forma de um vetor em um espaço algébrico semântico de alta dimensionalidade. Esta representação matemática da informação permite identificar os relacionamentos semânticos que não estejam explícitos entre termos, documentos e consultas geradas por um usuário de um sistema de RI. Em Engenharia de Software, LSA tem sido utilizada em muitos contextos para diferentes tarefas, dentre as quais: reuso de software, identificação de tipos abstratos de dados, detecção de clones, recuperação da rastreabilidade entre artefatos de software, clusterização de dados, dentre outras [Poshyvanyk et al. 2007]. Entretanto, em [Nakov et al. 2001], defende-se que a efetiva utilização desta técnica de Recuperação de Informação requer um processo avançado de ajustamento de parâmetros da técnica de busca e pode ser considerado até mesmo um tipo de arte. Segundo o artigo os principais fatores que influenciam os resultados obtidos na aplicação da técnica são:

- o processamento prévio da coleção de dados a ser analisada. Transformações passíveis de serem realizadas neste fase são a remoção de *stop-words* e *stemming* sobre os elementos do código fonte;
- transformações realizadas na matriz de termos-documentos;
- escolha do fator de redução da dimensionalidade do espaço vetorial semântico;
- escolha da medida de similaridade a ser aplicada entre os elementos sob análise (termos e documentos).

O presente trabalho retrata a aplicação da técnica LSA, mais precisamente da indexação LSI (*latent Semantic Indexing*), e a combinação de seus parâmetros de entrada em um contexto de manutenção de software. O objetivo é reduzir o esforço dispendido pelo desenvolvedor em uma tarefa de localização de um defeito no código fonte. Em [Lukins et al. 2008], é definido o processo de localização de defeitos como a tarefa de manutenção de software na qual o desenvolvedor usa informações sobre o defeito presente no sistema de software para localizar a parte do código fonte que deverá ser alterada no intuito de corrigir o mesmo. Neste ponto, deve-se ressaltar a importância dos repositórios de software e de defeitos como fontes de informação essenciais à recuperação da rastreabilidade entre artefatos de software [Hassan 2008] [Williams e Hollingsworth 2005]]. Considerando que a técnica de RI escolhida é o LSI, buscar-se-á verificar experimentalmente, a partir do teste das combinações de parâmetros de entrada da técnica citada, qual configuração

provê melhor acurácia e performance para um conjunto de documentos. Tais documentos são formados por fragmentos do código fonte. As medidas de precisão e revocação são ferramentas para avaliar o desempenho das respectivas combinações. As seções seguintes contextualizam a tarefa descrita anteriormente a partir da definição teórica do conjunto de elementos necessários à realização da análise aqui proposta.

2.2 Rastreabilidade entre Artefatos de Software

Dada a importância da análise de impacto de mudanças em sistemas de software, a Engenharia de Software tem se empenhado em construir ferramentas para recuperar a rastreabilidade entre os artefatos de software em tarefas de compreensão e manutenção do sistema. A tarefa de localização de defeitos é um tipo de recuperação da rastreabilidade entre artefatos externos ao sistema (documentação em forma de descrição de defeitos) e artefatos internos (código fonte do sistema) [Antoniol et al. 2000], [Antoniol e Gueheneuc 2006], [Lucia et al. 2007], [Poshyvanyk et al. 2007].

Rastreabilidade de software é a habilidade de relacionar artefatos de software criados durante o ciclo de vida de desenvolvimento do software tais como documentação, especificação de requisitos, modelos de análise e projeto, código fonte e casos de teste [Antoniol et al. 2002]. A importância de se manter a rastreabilidade entre os artefatos de software criados durante o ciclo de vida do projeto tem sido evidenciada por vários estudos [Egyed e Grunbacher 2002] [Ramesh 1998] nos quais a rastreabilidade entre requisitos e os demais artefatos de software é relacionada como uma medida da qualidade do sistema.

Um dos grandes desafios para se manter um sistema de rastreabilidade efetivo é o fato de que os artefatos rastreados continuam a mudar e evoluir à medida que o ciclo de vida do software se processa [Huang et al. 2003]. Assim, se a rastreabilidade não é atualizada, ela corre o risco de se tornar uma documentação obsoleta que não reflete os relacionamentos entre os artefatos de software em seu estado atual de implementação. Os elos de rastreabilidade precisam ser mantidos continuamente e se degradam com o tempo permanecendo ainda como uma atividade manual para grande parte das empresas [Egyed et al. 2009].

Um esforço considerável tem sido feito pela comunidade científica ligada à Engenharia de Software para o desenvolvimento de ferramentas que explicitem a conexão entre a documentação do sistema e o código fonte [Antoniol et al. 2002]. A documentação do sistema é complexa e possui muitas variantes que podem auxiliar os processos de manutenção de software. Uma destas variantes é a documentação produzida pelos usuários nos relatórios de erro do sistema. Sua rastreabilidade para o código fonte tem sido utilizada em vários trabalhos que buscam automatizar tarefas de manutenção de software, precisamente na localização de defeitos no código fonte [Lukins et al. 2008] [Hassan 2008]. Técnicas para a localização automática de defeitos tomam como entrada informações sobre assuntos rela-

tivos ao software e retornam como resultado uma lista de entidades como classes, métodos ou fragmentos de código fonte.

Os Repositórios de Software e Defeitos funcionam como verdadeiros depósitos de artefatos de software como documentação e código fonte. Sistemas de versionamento de software, como CVS[1], oferecem um enorme potencial como ferramenta no auxílio à manutenção de software, além de permitir investigar e compreender a evolução de grandes sistemas. Outros mecanismos como as ferramentas *on-line* para triagem de defeitos ajudam a compreender que partes do sistema são afetadas pelas falhas ocasionadas pelos defeitos. Este conjunto de artefatos forma um arcabouço no qual o desenvolvedor pode apoiar-se para descobrir relacionamentos que não estejam explícitos. Catalogadores de defeitos, sistemas de controle de versão, wikis e listas de email são ferramentas comuns para ajudar na administração do desenvolvimento e evolução dos projetos de software. Os repositórios são classificados de acordo com a informação que armazenam [Hassan 2008]:

- repositórios de caráter histórico - repositórios de controle de versão, repositórios de defeitos e de comunicações realizadas, guardam diversas informações sobre a evolução e progresso de um projeto;

- repositórios ligados a execução do programa - registros de implantação, explicitam informações sobre a execução e uso do software por um único ou múltiplos sites de implantação;

- repositórios de código - repositórios como *Sourceforge*[2] e *Google Code*[3], caracterizam-se pela presença de código fonte de várias aplicações desenvolvidas.

Cada tipo de repositório abriga informações que podem expressar aspectos evolutivos do projeto de software. Sua exploração tem levado a descoberta de padrões e ligações entre artefatos ali hospedados. Desta forma, se parte do código fonte é refatorado, evidências destas alterações estarão expressas no repositório bem como as discussões que ocasionaram algumas decisões de projeto e arquitetura. Um processo de manutenção corretiva de software inicia-se quase sempre a partir de relatórios de defeitos reportados pelos usuários finais do software através de mecanismos de catalogação *on-line* ou repositórios. Um exemplo tradicional de ferramenta para triagem de defeitos, utilizados por projetos de código aberto, é a ferramenta *Bugzilla*[4].

No desenvolvimento de software, relatórios sobre defeitos fornecem informações cruciais para desenvolvedores. A localização de defeitos é uma tarefa de manutenção de software na qual o desenvolvedor usa informações sobre o defeito apresentado pelo software para localizar a parte do código fonte que deverá ser modificada para corrigir o

[1] http://www.nongnu.org/cvs
[2] http://sourceforge.net
[3] http://code.google.com
[4] http://www.bugzilla.org

defeito [Lukins et al. 2008]. Desenvolvedores geralmente usam sua experiência quando realizam inclusão de novas características ou correção de erros no software. Testadores de software priorizam o teste de funcionalidades susceptíveis a erro baseados em arquivos e relatórios de erros armazenados em repositórios de software e sistemas de triagem de defeitos. Assim, a utilização destes repositórios proporciona aos desenvolvedores menor dependência em relação à intuição e experiência pessoal e um maior embasamento em dados históricos arquivados durante o ciclo de vida do software [Hassan 2008].

2.3 O uso de Análises Estática e Dinâmica de Código Fonte para Recuperação de Rastreabilidade

A localização dos métodos que concorrem para o aparecimento de um defeito é semelhante à localização de quais partes do código concorrem para a execução de uma funcionalidade do software, isto é, localização de características [Poshyvanyk et al. 2007]. O conceito de característica utilizado no presente trabalho pode ser encontrado em [Poshyvanyk et al. 2007]: "Uma característica de um programa representa alguma funcionalidade que está acessível e visível ao desenvolvedor e está usualmente expressa em um requisito."

Geralmente, usuários e engenheiros de software se comunicam em função das funcionalidades que um sistema apresenta. Características são as propriedades de um software que afetam diretamente usuários finais, ou seja, as funcionalidades presentes no software que podem ser observadas e controladas em tempo de execução pelos usuários finais, conforme definido em [Czarnecki e Eisenecker 2000]. Segundo eles, características encapsulam o conhecimento do domínio de um sistema de software representando uma excelente fonte de informação para a engenharia reversa. Diferentemente do requisito de software, a característica, como estudada neste trabalho, é a funcionalidade acionada pelo usuário em tempo de execução. O presente trabalho instrumentaliza o código fonte para obtenção de rastros de execução associados às características. A informação obtida desta maneira subsidia uma análise dinâmica do código fonte, reduzindo o número de métodos analisados na busca por um defeito. Adicionalmente, este trabalho também utiliza um filtro para o código fonte baseado em análise estática para reduzir o volume de código analisado e consequentemente o espaço de busca da técnica LSI. Em [Lukins et al. 2008], afirma-se que técnicas estáticas para localização de defeitos recolhem informações do código fonte e podem ser aplicadas em qualquer estágio do desenvolvimento não necessitando de um programa plenamente funcional. Por outro lado, técnicas dinâmicas usam informações obtidas dos rastros de execução do sistema. A mesma diferenciação é encontrada em [Poshyvanyk et al. 2007], mas aplicada a um contexto de localização de características. Para eles, a localização efetiva de características baseiam-se tanto em análise estática como dinâmica do código fonte. Os dois filtros de código fonte utilizados

neste trabalho, denominados ET (*Execution Traces*) e QTL (*Query Term Like*), utilizam abordagens dinâmica e híbrida, respectivamente, e objetivam diminuir o espaço de busca analisado pela técnica LSI. Uma vez definida a técnica, passa-se à análise dos parâmetros que influenciam a performance e acurácia decorrentes da aplicação da mesma.

2.4 LSI - Indexação por Semântica Latente

Este trabalho explora a técnica de Recuperação de Informação LSI. A origem desta técnica está baseada em um modelo matemático denominado espaço vetorial que é um modelo algébrico capaz de representar documentos de texto como vetores de palavras [Salton et al. 1975]. Dentro de um contexto de busca de documentos, a técnica LSI pemite medir a similaridade entre documentos e consultas com base nos vetores matemáticos que os representam. Se a consulta é considerada semelhante semanticamente a um documento, então, um coeficiente de similaridade pode expressar esta aproximação entre cada documento e consulta.

A técnica LSI explicita as correlações entre termos pertencentes a uma coleção de documentos usando uma representação reduzida da dimensionalidade do espaço vetorial sem grandes perdas de informação. A aplicação desta técnica tem se mostrado mais eficiente quando comparada aos modelos de espaço vetorial e a classificação Bayesiana, além de atacar os problemas de polissemia e sinonímia [Poshyvanyk et al. 2007], isto é, uma palavra poderá ter diversos significados e diferentes palavras poderão referenciar o mesmo significado respectivamente. A idéia principal em torno do LSI é explorar um conjunto implícito de dependências entre as palavras a partir do contexto em que aparecem (frases, conjunto contíguos de dados, documentos).

Algumas técnicas de RI são aplicadas para auxiliar tarefas de manutenção de software e operam sobre informações semânticas oriundas do código fonte incluindo seus identificadores e comentários [Lukins et al. 2008]. Documentos gerados a partir destas informações compõem o que foi definido em [Manning et al. 2008] como corpus, isto é, uma coleção de documentos sobre os quais se aplica a técnica de RI. Um documento individual do corpus é construído a partir de um elemento do código fonte tal como um pacote, uma classe, um método, um conjunto contíguo de dados. Comumente, os resultados providos pela técnica de RI são retornados em uma destas granularidades.

LSA é um processo compreendido por dois estágios: indexar e analisar os dados [Nakov et al. 2001]. A fase de indexação se inicia com a construção de uma matriz A cujas colunas são associadas com os documentos e as linhas com os termos presentes nestes documentos. A célula *(i,j)* da matriz A contém a frequência (possivelmente transformada usando um esquema de pesos) do termo i no documento j. A matriz A é então submetida a um processo matemático denominado decomposição por valores singulares (SVD), resultando

na produção de três outras matrizes T, D (ortonormal) e S (diagonal) tal que

$$A = TSD^t$$

.

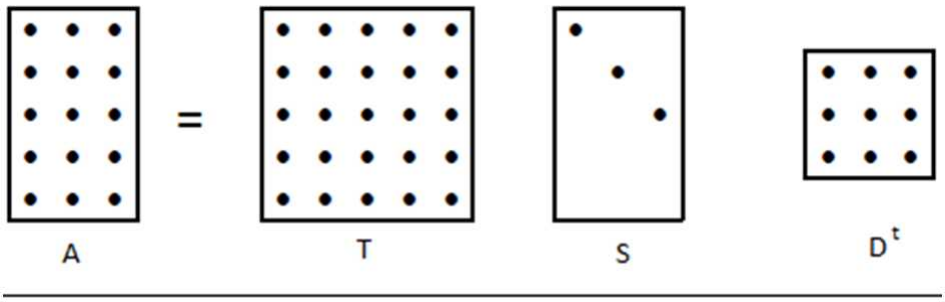


Figura 2.1: Decomposição da matriz de termos-documentos

Algumas das linhas e colunas de T, S e D são removidas obedecendo um fator de redução de dimensionalidade k . Em outras palavras, as matrizes são truncadas para k dimensões. Na prática, isto promove a redução do espaço semântico de busca agora representado pelo produto de três novas matrizes conforme ilustrado na Figura 2.2. Esta redução elimina os ruídos presentes nos vetores representativos de documentos e termos do corpus e mantém uma boa representação semântica dos mesmos [Lukins et al. 2008]. A redução do número de dimensões para um fator entre 50 e 400, conforme sugerido pela literatura correlata, resulta na compressão do espaço semântico original [Nakov et al. 2001], [Kontostathis 2007].

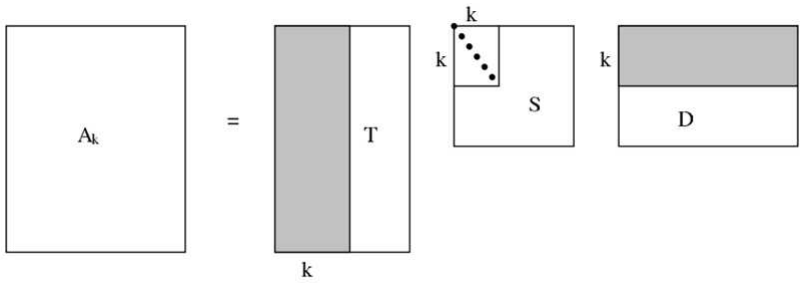


Figura 2.2: Aplicação de LSI para redução de dimensionalidade

A decomposição por valores singulares realizada pelo algoritmo SVD é usada para permitir a acomodação do espaço vetorial de termos-documentos, refletindo os principais padrões associativos entre os dados em detrimento de associações menos significativas [Deerwester et al. 1990]. A nova matriz obtida, $\mathrm{A}_k = \mathrm{T}_k\mathrm{S}_k\mathrm{D}_k^T$, é a representação condensada do espaço vetorial original e pode ser utilizada para mensuração de similaridade entre documentos e consultas originadas pelo usuário do sistema de RI. No novo espaço vetorial reduzido, representado pela matriz A_k, termos e documentos semanticamente semelhantes presumivelmente se colocam próximos uns dos outros. Após a indexação, inicia-se a fase de análise na qual mensura-se a similaridade entre elementos

envolvidos no processo de busca. Para comparar uma consulta de usuário com os documentos do corpus, esta deverá antes ser transformada em um pseudo-documento que especifica a localização da mesma no espaço vetorial semântico. Suponha um vetor cujos valores diferentes de zero contém a frequência dos termos que aparecem na consulta, o pseudo-documento pode ser representado como definido em [Letsche e Berry 1997] como:

$$\hat{q} = q^T T_k \; S_k^{-1}$$

Uma vez projetada a consulta no espaço vetorial de termos e documentos, diversas medidas de similaridade podem ser aplicadas para comparar a posição do pseudo-documento em relação às posições dos documentos do corpus. Em RI, a similaridade entre dois documentos é usualmente computada considerando cada documento como um vetor de frequência de palavras e calculando o cosseno do ângulo formado por dois vetores de frequência [Breese et al. 1998]. Outras medidas também podem ser utilizadas, dentre as quais: distâncias Euclidiana ou de Manhattan, Minkowski e Correlação de Pearson [Nakov et al. 2001]. O presente trabalho utiliza a correlação de Pearson como medida de similaridade entre dois documentos pela sua utilização em trabalhos prévios associados a LSI e por ser uma medida similar ao cosseno [Wild e Stahl 2007].

2.4.1 Utilização de Função de Peso Específica na Matriz de Termos Documentos

Tarefas de RI partem geralmente de matrizes de termos-documentos. Tais matrizes expressam a frequência com que os termos aparecem nos documentos do corpus. Cada célula individual da matriz representa a frequência com que um termo ocorre dentro de um documento específico. Em muitas aplicações de recuperação de informação as frequências são transformadas para refletir a habilidade das palavras em discriminar documentos uns dos outros [Dumais 1996] . Desta forma, aos termos que possuem um caráter discriminatório maior devem ser atribuídos pesos de alto valor enquanto termos pouco discriminatórios recebem baixos pesos. Os termos que permanecem na matriz são resultantes da aplicação de um conjunto de técnicas de *parsing* no código fonte tais como a remoção de *stop-words* e *stemming*. Termos extremamente comuns não são muito úteis em buscas. Exemplos de tais termos comuns incluem "a", "um", "o", dentre outros. Os termos que ocorrem de forma extremamente frequente são chamados de palavras de parada (*stop-words*) [Arun et al. 2009] e os documentos devem ser previamente processados para eliminar tais palavras antes do processo de indexação. Stemming é o processo heurístico de eliminação de afixos decorrente do processo de derivação das palavras [Manning et al. 2008]. Após o processo de *parsing*, deve-se escolher o conjunto de funções de peso a ser aplicado sobre a matriz de termos-documentos. Funções de peso podem alocar peso para

cada termo com base na sua frequência no corpus [Santhiappan et al. 2006]. Cada célula da matriz é substituída pelo produto de uma função de peso local (baseada na frequência do termo no documento em que o mesmo aparece) em conjunto com outra de peso global (baseada na ocorrência do termo em todos os documentos do corpus). Em [Yihong e Wei 2007] são listadas algumas funções de peso locais e globais mais tradicionais e de interesse para o presente trabalho. As funções relevantes citadas são:

- Nenhum peso: *L(i)=tf(i)* onde *tf(i)* é igual ao número de vezes onde o termo *i* ocorre no documento.
- Peso logarítmico: *L(i) = log(1+tf(i))*.

As funções globais citadas são:

- Nenhum peso: *G(i) = 1* para qualquer termo *i*.
- Frequência inversa do documento (idf): *G(i) = log(N/n(i))* onde *N* é o número total de documentos no corpus e *n(i)* é número de documentos que contém o termo *i*.

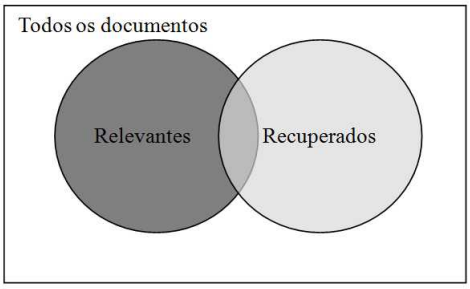
A evidência experimental acumulada durante vários anos indica que os sistemas de indexação de texto, baseados na associação apropriada de esquemas de peso aos termos, produzem resultados superiores àqueles conduzidos de forma tradicional em tarefas relacionadas a recuperação de informação [Salton e Buckley 1987]. Estes resultados dependem fortemente da escolha correta do sistema de pesos a ser aplicado uma vez que tal escolha deve espelhar a melhor distinção entre os termos da coleção. Para este trabalho foi adotado um esquema de pesos em que são realçados nomes de métodos e de classes juntamente com a separação de tais nomes em seus radicais primitivos ou fragmentos. Assim, o trabalho corrente propõe uma metodologia inovadora que permite posteriormente avaliar além da simples frequência do termo, a sua função hierárquica/estrutural dentro da metodologia orientada a objetos. A definição de tal função de peso é feita na seção sobre metodologia e seus efeitos sobre a acurácia são discutidos nos resultados.

2.4.2 Medidas de Avaliação usadas em Técnicas de RI

Uma vez que as similaridades entre a consulta e todos os outros documentos do espaço tenham sido computadas, os documentos podem ser ranqueados em uma lista ordenada para posterior avaliação. Nesta lista de documentos recuperados existem aqueles que são relevantes à consulta e outros que não são relevantes. Em um sistema ideal, estes dois conjuntos de documentos devem ser equivalentes, isto é, devem ser recuperados somente documentos relevantes [Grossman e Frieder 2004]. Na realidade, sistemas de RI recuperam muitos documentos não relevantes e as medidas mais comuns utilizadas para se mensurar

esta performance são precisão e revocação. Precisão é a medida do número de documentos relevantes recuperados sobre o total de documentos recuperados [Grossman e Frieder 2004], isto é, a porcentagem de documentos recuperados que são relevantes para a consulta [Poshyvanyk et al. 2007]. A precisão provê uma maneira de se avaliar a qualidade do conjunto de respostas retornado. Contudo, esta medida não considera o número total de documentos relevantes. Por exemplo, um sistema poderá ter boa precisão se recuperar 10 documentos e encontrar que 9 destes são relevantes (precisão = 0,9). Mas o número total de documentos relevantes também importa. Se somente 9 documentos forem relevantes o sistema logrará êxito em sua tarefa, mas se milhões de documentos forem relevantes não se pode dizer a mesma coisa. Por outro lado, a revocação considera o número total de documentos relevantes. Ela é a medida do número de documentos relevantes recuperados sobre o total de documentos da coleção que se acredita ser relevantes [Grossman e Frieder 2004]. Trata-se em linhas gerais da porcentagem de documentos relevantes no banco de dados em análise que são recuperados em resposta a uma consulta [Poshyvanyk et al. 2007]. Alguns defeitos analisados neste trabalho demandavam a correção de vários métodos para que fossem corrigidos. A precisão de cada teste de combinação de parâmetros realizada para um defeito foi expressa como a média das precisões encontradas em cada ponto de revocação como definido em [Grossman e Frieder 2004].

A Figura 2.3 ilustra a relação entre precisão e revocação. Em um modelo ideal a interseção entre relevantes e recuperados seria total, isto é, todos os documentos relevantes seriam recuperados e todos os documentos recuperados seriam relevantes.



$$\textbf{Precisão} = \frac{\text{Relevantes Recuperados}}{\text{Recuperados}}$$

$$\textbf{Revocação} = \frac{\text{Relevantes Recuperados}}{\text{Relevantes}}$$

Figura 2.3: Definição das medidas de precisão e revocação

No trabalho corrente, a precisão e revocação são ferramentas auxiliares ao processo de avaliação de parâmetros da técnica de busca LSI. Cada combinação de parâmetros gerada é submetida a um teste utilizando LSI e a precisão e revocação são registradas no banco de dados. A granularidade dos resultados é expressa em métodos do código fonte que são ranqueados em uma lista descendente de similaridade com a consulta do usuário. Ao varrer a lista de métodos ranqueados e encontrar um método relevante à correção do

defeito são computadas as precisões e revocações. A precisão também é utilizada para gerar comparativos entre filtros de código fonte conforme proposto por este trabalho.

2.4.3 Outras Técnicas de RI propostas para Localização de Defeitos, Conceitos e Características

Diversas técnicas de RI têm sido aplicadas com sucesso em tarefas de recuperação de informação em auxílio às tarefas de manutenção de software: LSI [Andrian Marcus 2003], pLSI [Poshyvanyk et al. 2007] e LDA [Lukins et al. 2008]. pLSI e LDA são técnicas baseadas em Modelos de Tópicos. Os Modelos de Tópicos são baseados na idéia de que documentos são uma mistura de tópicos. Tópicos podem ser interpretados como assuntos ou idéias que permeiam toda uma coleção de textos. Cada tópico é aprendido da coleção de texto analisada e é representado por uma coleção de palavras com probabilidades associadas. Um modelo de tópico é um modelo generativo para documentos, isto é, ele especifica um simples procedimento probabilístico através do qual documentos podem ser construídos a partir dos tópicos existentes [Landauer 2007].

Uma derivação da técnica LSI é *pLSI (Probabilistic Latent Semantic Indexing)*. Conforme exposto em [Lukins et al. 2008], pLSI é um modelo generativo de tópicos com forte base estatística. Em pLSI cada termo em um documento é modelado como uma mistura realizada sobre um conjunto de variáveis aleatórias polinomiais que representam os tópicos. Cada documento é modelado como uma distribuição de probabilidade em um conjunto fixo de tópicos. Em [Poshyvanyk et al. 2007], a técnica pLSI foi utilizada para localização de características e defeitos.

Outra técnica utilizada é *LDA (Latent Dirichlet Allocation)*. LDA é um modelo generativo de tópicos baseado em probabilidade utilizado para extrair tópicos que estejam latentes ou ocultos em uma coleção de documentos. Este modelo permite modelar cada documento como uma mistura finita sobre o conjunto de tópicos. Cada tópico neste conjunto é a uma distribuição de probabilidade sobre o conjunto de termos que constituem o vocabulário da coleção de documentos. Especificamente, assume-se que K tópicos estão associados com uma coleção e que cada documento expressa estes tópicos em diferentes proporções, isto é, cada documento é uma variada combinação de tópicos. Em [Lukins et al. 2008], a técnica LDA foi utilizada para localização de defeitos.

2.5 Resumo do Referencial Teórico

A recuperação da rastreabilidade de software é um importante instrumento para a realização da manutenção de sistemas. Através dela é possível relacionar artefatos de software provendo ao desenvolvedor maior compreensão do software. Uma maneira de se recuperar a rastreabilidade é utilizar-se de técnicas de recuperação de informação sobre o

código fonte.

LSI é uma técnica de RI baseada na manipulação algébrica de vetores formados pela frequência das palavras nos documentos do corpus. Sistemas de busca convencionais baseados no casamento de padrões de *strings* são inferiores a LSI pois não levam em consideração alguns aspectos como o contexto em que as palavras ocorrem, a sinonimia e as relações latentes não explícitas entre os documentos e termos analisados [Marcus et al. 2004]. No contexto da manutenção de software, LSI pode ser empregada para localização de defeitos, características e demais atividades de natureza semântica.

A utilização da técnica LSI pode ser customizada, através da manipulação dos seus parâmetros de entrada, para que melhores resultados quanto à performance e acurácia dos testes possam ser alcançados. Quando aplicado em um contexto de manutenção de software, esta customização pode diminuir os esforços despendidos pelo desenvolvedor, aumentando a compreensão do sistema e reduzindo os custos relativos à manutenção. Os parâmetros, definidos como alvo de estudo neste trabalho, compreendem os filtros de código fonte obtidos a partir de análise estática e dinâmica do mesmo, o fator de redução da dimensionalidade k e a função de peso aplicados à matriz de termos-documentos utilizada pela técnica LSI, além do conjunto de *stop-words* utilizadas sobre os termos do corpus.

Os conceitos explorados nesta seção alicerçam os elementos utilizados na metodologia exposta no próximo capítulo. Além disso, a utilização da técnica LSI e a combinação de seus parâmetros podem ser melhor compreendidos, durante a exploração dos resultados, a partir dos conceitos aqui definidos. Resumidamente, ao utilizar a técnica LSI aqui definida, objetiva-se recuperar a rastreabilidade entre código fonte e documentação do sistema, analisando-se empiricamente os valores dos parâmetros da técnica que impliquem um menor trabalho por parte do desenvolvedor.

Capítulo 3

Metodologia de Trabalho

Neste trabalho é proposta uma abordagem voltada à localização de defeitos com base na execução de características do software. Isto requer a disponibilidade de informações relativas à descrição dos defeitos para extração dos cenários de execução. Neste ponto, são fundamentais os sistemas de triagem de defeitos disponíveis na Internet e as listas de discussões de desenvolvedores ou históricos de correções do software. Dados extraídos destas fontes subsidiam os testes de combinações de parâmetros da técnica LSI. A intenção é delinear uma metodologia capaz de prover a calibragem satisfatória de tais parâmetros em um contexto de manutenção de software e mais especificamente, localização de defeitos.

3.1 O Processo de Localização de Defeitos

O processo de localização de defeitos, baseado em RI, assemelha-se à tarefa de encontrar informações na Internet usando um mecanismo de busca. A técnica de busca utilizada no presente trabalho é LSI [Deerwester 1988]. Embora outras técnicas possam também ser consideradas para a tarefa de localizar defeitos, LSI é razoavelmente bem estudada no que se refere à recuperação da rastreabilidade [Antoniol et al. 2000], [Lucia et al. 2009], [Lucia et al. 2007]. A Figura 3.1 mostra as principais fases e produtos do processo de localização de defeitos utilizados neste trabalho e está numerada indicando cada parte do processo que é explicada a seguir. Existem duas principais tarefas dentro deste processo: a criação do corpus (passos 1 a 13) e a formulação da consulta (passos 1, 18,19,20 e 21).

Antes que o usuário possa entrar com consultas no mecanismo de buscas criado, o código fonte proveniente do repositório de software precisa ser indexado. Um importante problema que antecede o processo de indexação é a criação do corpus. Na Figura 3.1, a criação do corpus é definida pela extração (passo 1) e filtragem do código fonte (passos 10, 11 e 12). Para extração, o código fonte é submetido a um processo de *parsing* (passo 2) e os documentos são extraídos. Neste processo, cada arquivo fonte Java é fragmentado em documentos menores e cada documento resultante contém a implementação de um

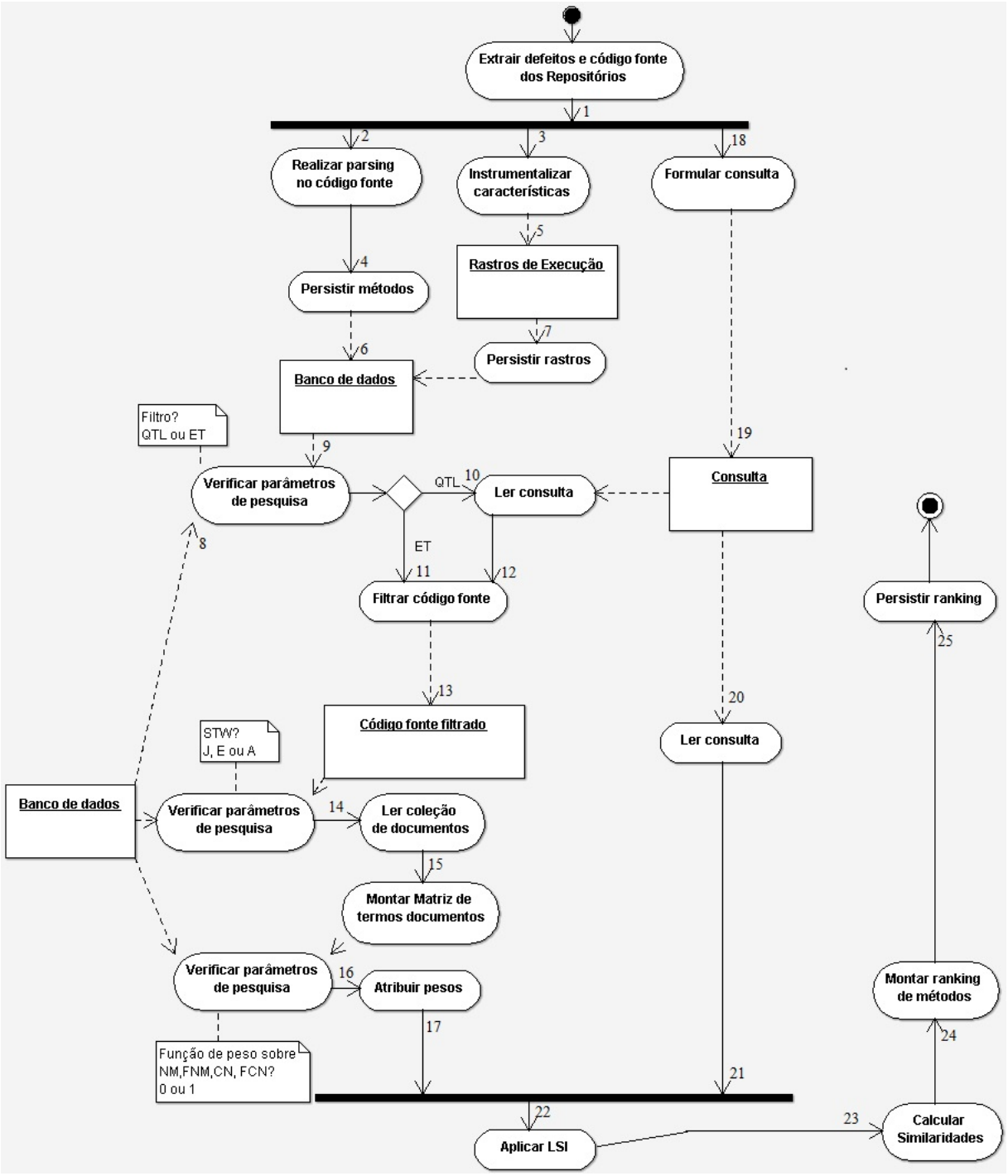


Figura 3.1: O processo de localização de defeitos

método e seus comentários associados, excluindo-se caracteres especiais e números. Um diferencial do trabalho corrente é analisar formas alternativas para filtragem do corpus, medindo-se a influência destas novas estratégias para a localização de defeitos. Com base nos dois filtros definidos neste trabalho (ET e QTL) forma-se uma coleção menor de documentos em relação a todo o código fonte. Posteriormente estes documentos são persistidos em disco sob a forma de arquivos de texto formando o corpus a ser indexado pela técnica LSI (passo 22). Com a indexação realizada é possível projetar consultas do

desenvolvedor interessado em localizar um defeito. O processo de formação da consulta, ilustrado nos passos 10, 11 e 12, com base nas descrições do defeito é detalhado na seção seguinte. Por fim, são calculadas as similaridades entre a consulta e todos os documentos do corpus (passo 23) e gera-se uma lista ordenada de documentos (passo 24) na qual os mais próximos do topo são semanticamente mais relacionados à consulta submetida.

3.2 O Processo de Formulação da Consulta

A qualidade da consulta é essencial para que o processo de localização de defeitos seja eficiente [Lukins et al. 2008]. Por prática, usuários de mecanismos de busca na Internet sabem que consultas ruins resultam em uma listagem inapropriada de resultados. Confirmando tal hipótese, usuários experientes geralmente obtêm melhores resultados em suas buscas porque possuem a habilidade de escolher boas palavras chave. Fato semelhante também se verifica no processo de localização de defeitos.

Este trabalho propõe uma metodologia para a obtenção da consulta sem contudo aferir a eficiência deste mecanismo proposto. No entanto, com base nos resultados de busca encontrados, pode-se afirmar que a metodologia é eficaz ao especificar uma maneira transparente e uniforme para coleta de termos da consulta utilizado no tratamento de descrições de vários defeitos. A Figura 3.2 ilustra detalhadamente o processo de formulação da consulta.

A entrada para a formulação da consulta é a descrição de um defeito disponível em repositórios *on-line* onde os usuários reportam a ocorrência de falhas no software em utilização. As descrições de defeito utilizadas neste trabalho incluem sempre uma descrição curta e uma descrição detalhada trazendo uniformidade ao processo de seleção de termos. A Tabela 3.2 ilustra uma descrição de defeito para o ArgoUML[1]. As linhas 2 e 3 mostram uma descrição curta do defeito e trata-se fundamentalmente de um resumo geral dado pelo usuário ao informar o defeito. A descrição detalhada (linhas 4 à 12) informa dentre outras coisas os passos que compõem a execução das características que culminam com a manifestação do defeito.

Os filtros são aplicados nas descrições curta e detalhada do defeito com o objetivo de extrair os termos da consulta. Os filtros aplicados são os seguintes:

1. Filtragem de Sentenças de Interesse (Filtro 1 da Figura 3.2) - Manter sentenças relacionadas com a reprodução da falha. Algumas sentenças devem ser removidas da descrição do defeito, entre elas as que incluem: o nome do software, a versão do sistema operacional, números identificadores de defeito, dados de entrada providos para exemplificar o problema, confirmações do erro, cenários redundantes que confirmem a presença do erro e preferências relativas ao comportamento do software.

[1] http://argouml.tigris.org

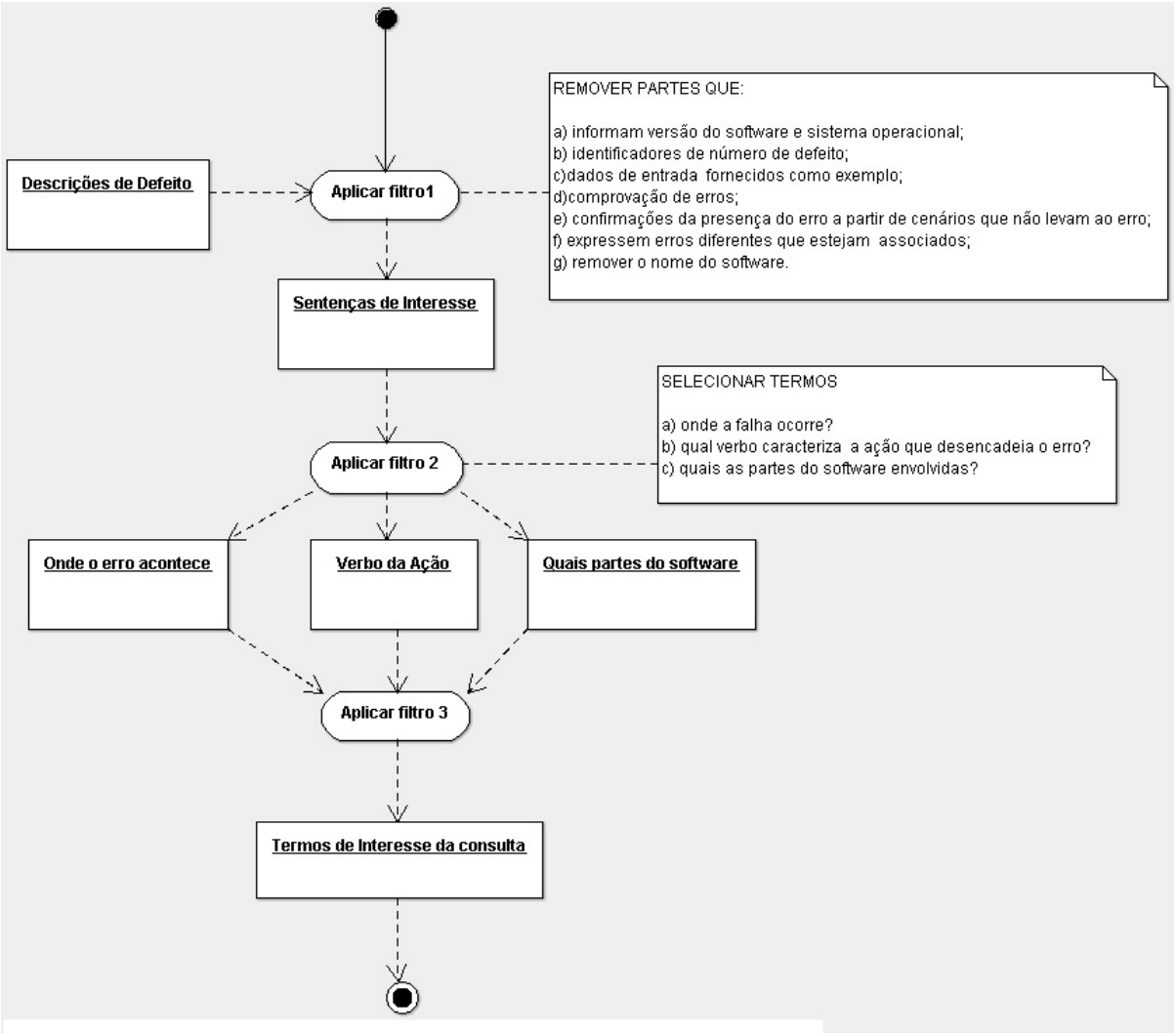


Figura 3.2: O processo de formulação da consulta

Levando-se em consideração a Tabela 3.2, duas sentenças foram filtradas (linhas tachadas 5,6 e 12). A primeira sentença eliminada refere-se a descrição de um erro relacionado e contém uma versão de software. A segunda expressa a preferência do usuário que informa o erro e deve também ser eliminada conforme a metodologia proposta.

2. Filtragem e classificação de termos (Filtro 2 da Figura 3.2)- Selecionar termos a partir das sentenças restantes classificando-os nas seguintes categorias: a) Onde a falha ocorre? b) Qual verbo caracteriza a ação que desencadeia o erro? c) Quais as partes do software estão envolvidas? Na Tabela 3.2, um exemplo deste filtro é mostrado nas linhas 4 a 6.

3. Normalização dos termos selecionados (Filtro 3 da Figura 3.2)- Expressar os termos na forma singular. Separar termos que estejam no padrão *CamelCase* e palavras compostas, preservando o termo original que foi dividido. Termos no padrão *CamelCase* apresentam letras capitalizadas e foram facilmente separadas a partir de recur-

1. Exemplo de Defeito: argo_uml_(rev_15168)
2. Descrição curta 3. Property panel should not be available on multi-target selection
4. Descrição Detalhada ~~5. This first occurs in 0.25.4 and seems to be a side effect of keeping the model~~ ~~6. element properties tab enabled.~~ 7. To reproduce place two classes on a diagram 8. Click on one and then ctrl-click the other. The properties tab is enabled 9. and visible but blank, this is good. 10. Click once on the diagram background and repeat the step above. This 11. time the property panel is visible. ~~12. I prefer that the tab stay enabled with no visiblen controls.~~
13. Onde? property panel
14. Ação? Click
15. Quais partes do software estão envolvidas no cenário? 16. property panel multi target selection classes diagram properties tab 17. diagram background property panel
18. Consulta Final 19. property panel click multi target selection class diagram property tab 20. background
21. Cenário de Execução 22. Expando a árvore do explorer e clico sobre "Diagrama de Classes". Clico 23. sobre o ícone para criação de uma nova classe por duas vezes para criar 24. duas classes. Clico sobre uma e seleciono ctrl no teclado e clico sobre a 25. outra. Clico no fundo da tela. Clico novamente sobre uma classe e seleciono 26. ctrl no teclado e clico sobre a outra.

Tabela 3.1: Exemplo da aplicação da metodologia para formulação da Consulta

sos da linguagem Java. Separar termos hifenizados. Manter abreviações, remover termos repetidos e por fim colocar todos os termos resultantes em minúsculo. A consulta final resultante desta normalização é mostrada na linha 7 da Tabela 3.2.

Um total de 23 defeitos foram analisados de maneira semelhante a realizada na Tabela 3.1. Todos foram submetidos à metodologia proposta para construção da consulta. O leitor interessado em verificar todos os outros defeitos estudados, com seus respectivos processos de formulação de consulta, pode consultar o Apêndice A.

3.3 Parâmetros da Técnica LSI

Alguns parâmetros são conhecidos por sua influência na performance da técnica LSI [Nakov et al. 2001]. Alguns fatores específicos que impactam as tarefas de manutenção de software foram alvo do estudo aqui realizado. Estes fatores são listados a seguir:

- Filtro para código fonte - a utilização de diferentes critérios para a criação do corpus resultando em parcelas menores de código fonte a serem analisadas com aplicação de LSI.

- Fator k referente a redução de dimensionalidade do espaço vetorial semântico, representado por um valor escalar correspondendo ao número de dimensões usadas para representar os documentos do corpus. Neste trabalho foram testados valores discretos entre 60 e 300 dimensões, conforme sugerido pela literatura correlata [Nakov et al. 2001], [Kontostathis 2007].

- Conjuntos de *stop-words* - termos a serem retirados das coleções de documentos por não representarem elementos que fornecem distinção entre os documentos do corpus. O trabalho corrente leva em consideração *stop-words* pertencentes à consulta a ser submetida no processo de localização de defeitos e as mantém na matriz de termos-documentos.

- Aplicação de esquemas de pesos específicos na matriz de termos-documentos que visam aumentar a importância relativa dos nomes de métodos, classes e seus fragmentos a partir da alteração da frequência destes elementos citados.

Os diferentes critérios utilizados para a criação do corpus são definidos como filtros a serem aplicados no conjunto de documentos a serem extraídos do código fonte. A granularidade dos resultados retornados para este trabalho elege o método como documento a ser recuperado e classificado em uma lista ordenada de relevância semântica. Em outras palavras, cada método do programa alvo é transformado em um documento. Logo, como o resultado de uma consulta é uma lista de documentos, tem-se que este resultado na verdade é uma lista de métodos. Optou-se por esta granularidade uma vez que é razoável para o desenvolvedor obter a indicação de qual(is) método(s) estão diretamente relacionado(s) ao defeito em estudo ou quais métodos possuem maior similaridade semântica com a consulta fornecida na busca.

Na seleção de quais documentos (métodos) seriam utilizados para compor o corpus, dois filtros foram estudados:

1. Filtro por rastros de execução (ET). É um filtro baseado na remoção de métodos (documentos) que não ocorrem no rastro de execução que reproduz o defeito.
2. Filtro por rastros de execução e termos da consulta (QTL). É um filtro baseado no rastro de execução e nos termos presentes na consulta gerada através da descrição do defeito. Este filtro incrementa o anterior removendo todos os métodos que não tem relação com qualquer dos termos da consulta e pode ser definido como: seja t um termo da consulta em sua forma stemizada, t' seja sua forma modificada *(%t%)* onde o caractere % funciona como prefixo/sufixo e t" um termo da implementação do

método correspondente a um documento no corpus. O critério para que um método possua uma relação com uma consulta é que a expressão *t' ilike t"* retorne verdadeiro para qualquer par *(t',t")*, considerando *ilike*[2] uma função SQL convencional.

A Figura 3.3 ilustra o tratamento do código fonte para as abordagens estudadas neste trabalho quanto à filtragem do código fonte para composição do corpus. Verifica-se que em duas abordagens (ET e QTL) foram aplicados filtros no intuito de se obter um menor número de documentos (métodos) a serem analisados. Estas abordagens podem ser evidenciadas na figura observando-se as setas de pontas simples. A abordagem que considera todo o código fonte aparece com setas duplas e foi realizada somente com os softwares JEdit[3] e ArmedBear-J [4] em função das limitações computacionais encontradas com o ArgoUML. Tais limitações são discutidas a seguir, nesta mesma seção, a partir da análise das Figuras 3.4, 3.5 e 3.6.

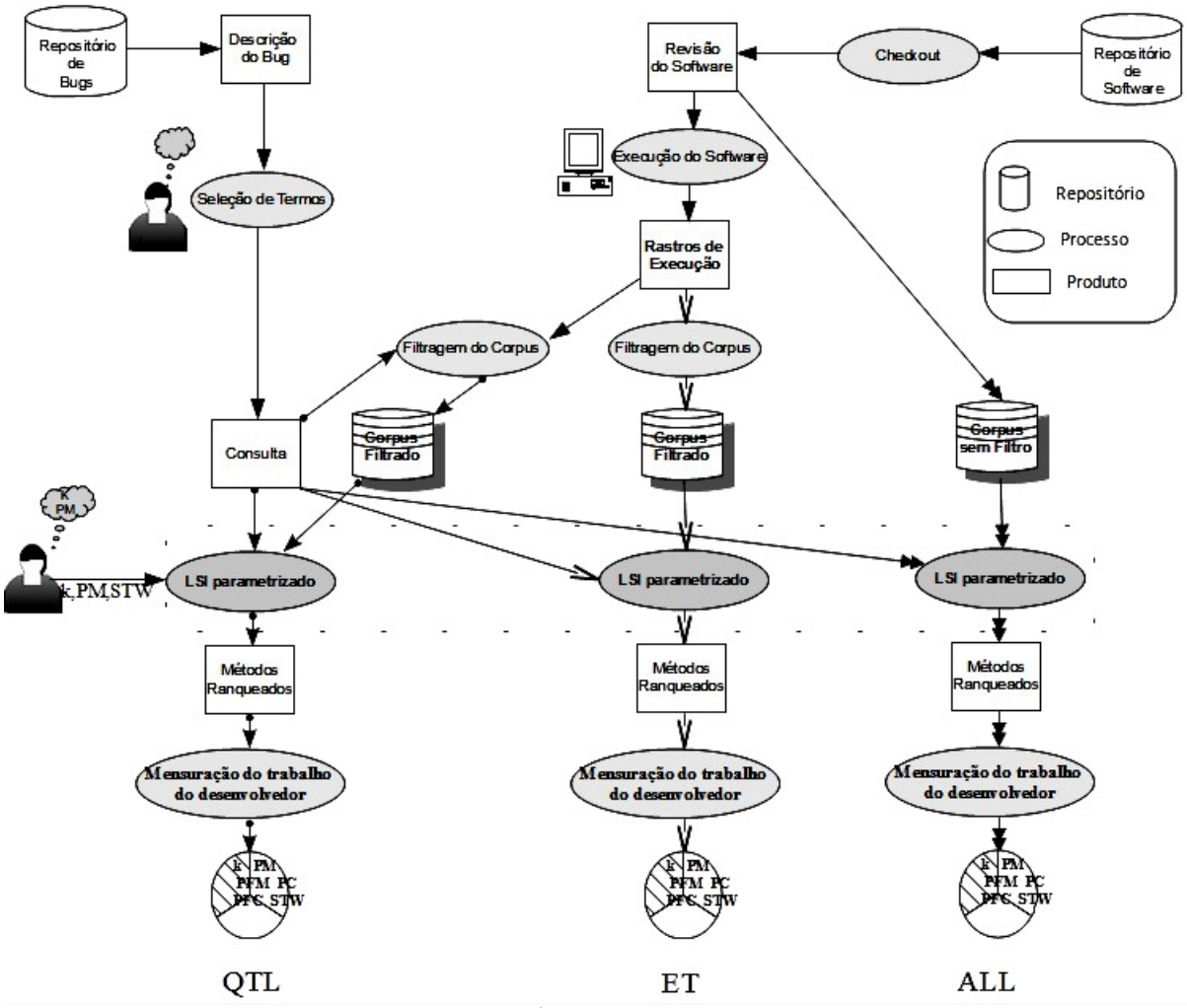


Figura 3.3: Abordagens para filtragem do código fonte

[2]http://www.postgresql.org/docs/7.3/static/functions-matching.html
[3]http://www.jedit.org
[4]http://armedbear-j.sourceforge.net

As palavras de parada (*stop-words*) utilizadas são geradas de maneira dinâmica e respeitam os termos da consulta. Isto significa que *stop-words* que fazem parte da consulta não são eliminadas no momento da montagem da matriz de termos-documentos. Resumem-se em três conjuntos as *stop-words* utilizadas:

- *Stop-words* da língua Inglesa (ESTW) - as palavras utilizadas neste trabalho foram obtidas do grupo de RI da Universidade de Glasgow/Escócia[5].

- *Stop-words* da linguagem Java (JSTW)- as palavras utilizadas neste estudo são as palavras reservadas da linguagem de programação Java.

- Todos os conjuntos de *Stop-words* (ALLSTW) - a união do conjunto de palavras de parada da língua Inglesa e da linguagem de programação Java.

O presente trabalho utiliza esquemas específicos de peso para a matriz de termos-documentos. Alguns experimentos científicos utilizam esquemas de pesos tradicionais como *tf* (frequência do termo) e *tf-idf* (resultado da combinação da frequência de um termo com a frequência inversa do documento, fornecendo um índice mais representativo). O mecanismo de peso proposto neste trabalho está baseado na percepção de que o desenvolvedor está interessado, como resultado de suas buscas sobre o código fonte, em métodos que concorrem para a *manifestação* do defeito. Desta forma, se o desenvolvedor fornecer ao mecanismo de buscas uma consulta que contenha o nome de um método relevante para a correção do defeito em estudo, espera-se que tal método figure entre os primeiros métodos ordenados da lista de métodos retornados. Neste sentido, termos e fragmentos de termos que ocorrem em nomes de métodos e classes devem ter um peso maior de acordo com o mecanismo de peso aqui utilizado. Isto é feito com base na alteração da frequência dos termos *tf*. A aplicação da função de peso sobre a matriz de termos-documentos tem o objetivo de dar maior importância aos termos do código fonte que se enquadrem dentro de classes de palavras do paradigma orientado a objetos. Lembrando que cada método de classe corresponde a um documento no corpus, define-se portanto, os termos que recebem maior importância, a partir da aplicação da função de peso:

- termos obtidos a partir do nome de classe (CN) para o documento analisado na matriz de termos-documentos;

- termos obtidos a partir de fragmentos de nomes de classes que estejam no padrão *CamelCase* (FCN) para o documento analisado na matriz de termos-documentos;

- termos obtidos a partir do nome do método (MN) para o documento analisado na matriz de termos-documentos;

[5]IR Group, "Stop word list", University of Glasgow. http://ir.dcs.gla.ac.uk/resources/linguistic_utils/

- termos obtidos a partir de fragmentos de nomes de métodos (FMN) que estejam no padrão *CamelCase* para o documento analisado na matriz de termos-documentos.

O procedimento para aplicação da função de peso sobre as linhas e colunas da matriz de termos-documentos é composto por diversas fases, assim enumeradas:

1. Montar a matriz de frequência de termos-documentos e persisti-la em disco utilizando o software estatístico R;
2. Utilizando um software desenvolvido em Java carregar a matriz para a memória e atribuir os pesos conforme parâmetros escolhidos para a busca;
3. Persistir a matriz em disco novamente e carregá-la novamente em memória utilizando o R para que se possa criar o espaço vetorial semântico.

Na utilização do software Java, para atribuição de pesos, cada linha da matriz termos-documentos é percorrida. Nas linhas da matriz estão os termos e nas colunas os documentos. Para cada linha analisada da matriz:

1. Verifica-se quais classes de palavras (nome de métodos, classes e seus fragmentos) deverão ter seus pesos alterados de acordo com os parâmetros da busca;
2. Se o peso de alguma das classes de palavras anteriores deve ser alterado, é verificado se o termo analisado na linha atual se enquadra em alguma das classes definidas;
3. Aplica-se a alteração do peso conforme o que foi definido em (1).

A metodologia definida neste trabalho para orientar a aplicação de peso às classes de palavras referenciadas acima obedece a seguinte função:

$$tf_{final} = 2 * tf_{maxima} + tf_{original}$$

onde:

- tf_{final} - novo peso atribuído ao termo na matriz de termos-documentos;
- tf_{maxima} - frequência máxima observada na matriz de termos-documentos;
- $\text{tf}_{original}$ - frequência original do termo na matriz de termos-documentos.

A intenção ao utilizar a frequência máxima está em garantir que nomes de métodos, classes e seus fragmentos possuam maior relevância dentre os termos que aparecem nos documentos em que os métodos ou classes estão implementados. O parâmetro de redução de dimensionalidade k será estudado usando-se os valores discretos 60, 100, 200 e 300 que estão dentro da variação sugerida pela literatura [Wild et al. 2005].

O objetivo deste trabalho é analisar a influência dos fatores mostrados nas seções anteriores sobre o resultado da localização de defeitos no código fonte a partir de técnicas de recuperação de informação. Para avaliar a existência de uma combinação predominante que gere os melhores resultados, foram selecionados 10 defeitos de cada um dos softwares ArgoUML e JEdit. Os defeitos foram selecionados aleatoriamente, mas satisfazendo algumas restrições: possibilidade de reprodução e disponibilidade de informações que atestavam quais métodos foram alterados para a correção do defeito. O corpus foi indexado utilizando-se diversas combinações de parâmetros. Os possíveis valores para os parâmetros são:

- Parâmetro de redução de dimensionalidade do espaço vetorial semântico k: 60,100, 200, 300.
- Parâmetro com peso atribuído ao nome da classe (CN): não, sim.
- Parâmetro com peso atribuído aos fragmentos do nome da classe (FCN): não, sim.
- Parâmetro com peso atribuído ao nome do método (MN): não, sim.
- Parâmetro com peso atribuído aos fragmentos do nome do método (FMN): não, sim.
- Conjunto de *stop-words* utilizado: ESTW, JSTW, ALLSTW.
- Abordagem para redução e filtragem do corpus: ET, QTL.

Os valores de parâmetros mostrados anteriormente definem um total de 288 combinações para serem testadas. Para cada defeito, o processo de formulação da consulta foi aplicado e os termos da consulta foram procurados utilizando-se LSI com todas as 288 combinações de parâmetros citadas. Outra possibilidade levantada é a análise da influência que os filtros propostos para o corpus possuem sobre:

- o tempo de execução para indexar e procurar;
- a precisão e revocação dos métodos ordenados na lista.

Para avaliar a influência dos filtros sobre os itens citados, 3 defeitos do JEdit e 3 defeitos do ArmedBear-J foram testados e comparados conforme exposto na Seção 4.3. O software ArgoUML não foi utilizado para este estudo porque o tempo e consumo de memória necessários para indexar todo o código fonte usando o pacote LSA do software estatístico R mostrou-se não satisfatório nos quesitos tempo e consumo de memória, conforme ilustrado pela Figura 3.4. A figura ilustra o consumo de processador e memória no momento da execução de um teste com o primeiro defeito do ArgoUML (ver Apêndice A).

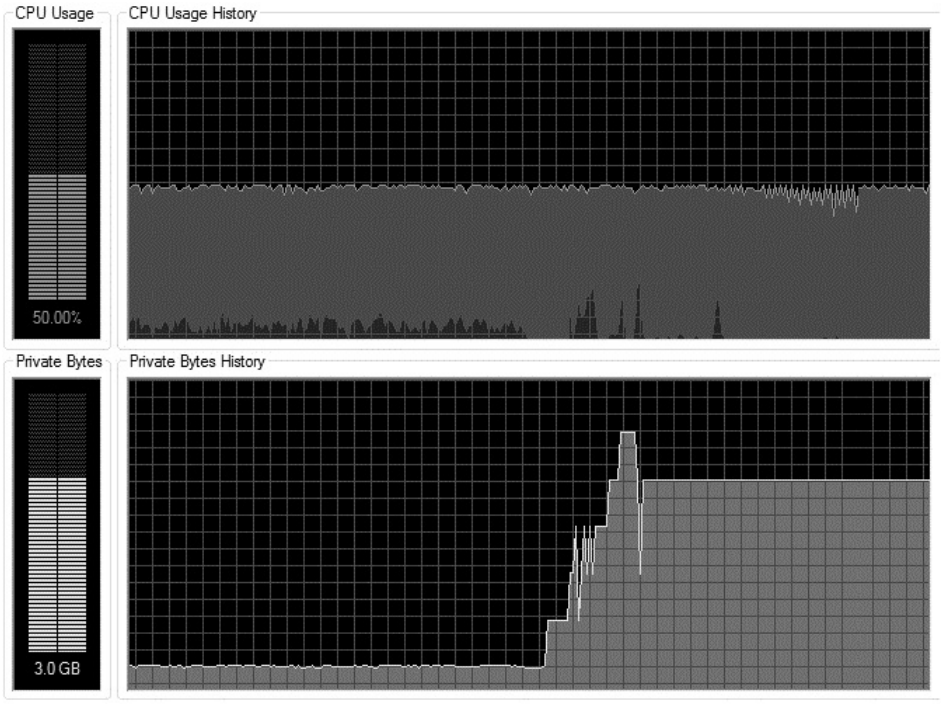


Figura 3.4: Teste utilizando todo o código fonte e um defeito do ArgoUML

```
[Área de trabalho anterior carregada]

> library("lsa")
Carregando pacotes exigidos: Snowball
Carregando pacotes exigidos: RWeka
> trm1 = read.csv("c:/analise/matriz/matriz2.csv",sep=",",header=TRUE,row.names=1)
> trm1 = as.matrix(trm1)
> trm1 = as.textmatrix(trm1)
> trm1 = lw_logtf(trm1)
Erro: cannot allocate vector of size 1.3 Gb
Além disso: Warning messages:
1: In lw_logtf(trm1) :
  Reached total allocation of 7932Mb: see help(memory.size)
2: In lw_logtf(trm1) :
  Reached total allocation of 7932Mb: see help(memory.size)
3: In lw_logtf(trm1) :
  Reached total allocation of 7932Mb: see help(memory.size)
4: In lw_logtf(trm1) :
  Reached total allocation of 7932Mb: see help(memory.size)
5: In lw_logtf(trm1) :
  Reached total allocation of 7932Mb: see help(memory.size)
6: In lw_logtf(trm1) :
  Reached total allocation of 7932Mb: see help(memory.size)
Execução interrompida
```

Figura 3.5: Log do teste com todo o código fonte de um defeito do ArgoUML

O consumo de memória no caso citado requereu mais memória que a disponível no sistema que era de 8 GigaBytes e a combinação utilizada no teste foi a combinação (k=300, NM=SIM, FNM=SIM, CN=SIM, FCN=SIM, STW=ESTW)) derivada da combinação predominante, que é explicada no próximo capítulo. O teste finalizou com erro por insuficiência de memória como pode ser observado pelo log da Figura 3.5. O mesmo teste realizado utilizando o filtro de código fonte QTL foi rápido e bem aceitável quanto ao con-

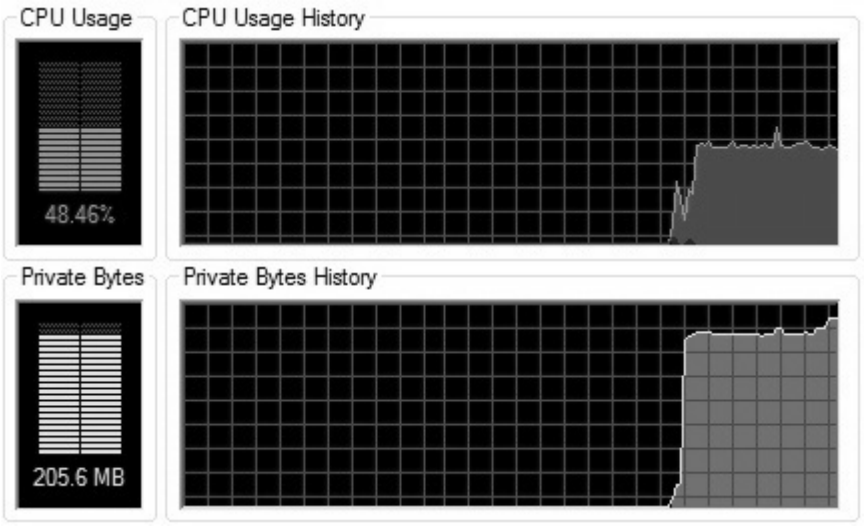


Figura 3.6: Teste utilizando o filtro QTL e um defeito do ArgoUML

sumo de mémoria e tempo de processador conforme pode ser visto na Figura 3.6. O tempo total do teste foi de 78 segundos com pico máximo de memória de aproximadamente 206 MegaBytes.

3.4 Os Critérios de Escolha dos Defeitos e Versões de Software Analisados

Os defeitos foram escolhidos aleatoriamente. Entretanto, defeitos que não possuíam as informações necessárias para sua correção ou reprodução foram descartados. Na Tabela 3.2 são listadas todas as versões de software com suas respectivas revisões e defeitos associados.

Técnicas de RI utilizam a posição dos documentos relevantes recuperados para cálculo de precisão e revocação. Desta forma, foram escolhidos defeitos em que o cenário de execução estivesse bem definido permitindo posterior reprodução do experimento por outras pessoas e utilização de outras técnicas de RI para estudos comparativos.

Outro ponto importante nesta escolha refere-se ao fato de que defeitos que acarretavam erros do tipo *NPE (Null Point Exception)* não se mostraram interessantes como alvos do estudo. Neste tipo de erro a exceção lançada já orienta o desenvolvedor sobre quais métodos alterar na correção do defeito apresentado. Desta forma somente foram utilizados defeitos que implicavam um mau comportamento do sistema do ponto de vista do usuário e que não apresentavam de forma clara a causa do problema.

Código	Descrição do Defeito	Versão/Revisão
1	Ao deletar a última estrutura "figpartition" em um diagrama de atividades fica oculta uma estrutura "figpool".	argo_uml_(rev_15155)
2	Permite selecionar e deletar dois elementos sendo um editável e outro somente leitura quando selecionados simultâneamente.	argo_uml_0_26_alpha_3
3	Permite mover um item da parte profile para a parte de user na árvore do explorer	argo_uml_0_26_alpha_3
4	Ao clicar com o botão direito sobre uma classe pela primeira vez mostra (show full path) ao invés de (hide full path).	argo_uml_rev_15144
5	A elipse cresce desproporcionalmente quando digitando um nome em um diagrama de caso de uso.	argo_uml_(rev_15089)
6	A visibilidade do painel de propriedades tem comportamentos distintos para multi-seleção de elementos.	argo_uml_(rev_15168)
7	Novos atributos criados no padrão a:Foo não podem ser deletados pois não estão no namespace.	argo_uml_(rev_15233)
8	Função Zoom não funciona.	argo_uml_(rev15688)
9	Ao criar um atributo de classe ocorre um erro com as barras de rolagem no campo valor inicial (ocultam o texto digitado).	argo_uml_(rev_15749)
10	Ao criar uma classe antes de soltar o mouse a figura da classe aparece deformada sem os compartimentos.	argo_uml_(rev_16517)
11	Ao localizar o caractere $ e mandar substituir por uma letra fica faltando substituir uma ocorrência.	jedit_(rev_13899)
12	Erro com upper case na procura de arquivos .cpp .h .H em todos os buffers abertos.	jedit_(rev_15833)
13	A barra de ação (action bar) não está acessível no modo tela cheia (full screen).	jedit_(rev_15716)
14	Ações to-lowercase e to-uppercase não funciona onde o cursor está com foco.	jedit_(rev_6684)
15	A barra horizontal não acompanha o movimento das teclas home e end permanecendo estática em linhas longas.	jedit_(rev_8281)
16	Comportamento incorreto do indicador de buffer alterado que não funciona em todas vezes que um buffer é alterado.	jedit_(rev_13506)
17	Erro na cor e distorção ao selecionar um texto com soft wrap e foreground selection ativados.	jedit_(rev_17226)
18	Ícone é incorretamente associado a item da barra de ferramentas ao ser alterado em global_options->toolbar	jedit_(rev_14664)
19	Procura por texto, sublinhando ao final da consulta a linha e o resultado, enquanto somente este último deveria ser sublinhado.	jedit_(rev_15504)
20	Ao criar um novo buffer e salvá-lo não é feita a ordenação alfabética no combobox de buffers já abertos.	jedit_(rev_16406)
21	Erro salvando sessão uma vez que já exista alguma aberta.	j_(rev_8874)
22	Identação java não funciona corretamente dando um shift muito acentuado à direita.	j_(rev_392)
23	Procura e substituição de string na caixa de dialogo não funciona.	j_(rev_2630)

Tabela 3.2: Descrições de defeitos e revisões de software associadas

3.5 Ferramentas Utilizadas para o Trabalho

Uma ferramenta administrativa foi desenvolvida com o objetivo de gerenciar todo o processo de armazenamento do código fonte, rastros de execução, descrição de defeitos e resultados de testes. No Apêndice B, são mostradas figuras que ilustram algumas telas administrativas disponibilizadas no sistema. É possível cadastrar cada código fonte e defeito de software, uma vez que cada defeito estudado corresponde a um código fonte

diferente a ser analisado em sua versão correspondente. A cada defeito cadastrado são associados os métodos que necessitaram ser corrigidos para correção da funcionalidade em estudo e os métodos acionados durante a execução da característica do sistema que apresentam um mal comportamento. Por conseguinte, o aplicativo Java dispara um processo do software estatístico R em separado para que a consulta seja processada e os resultados sejam registrados em um banco de dados. O detalhamento das tabelas do banco de dados está apresentado no Apêndice C. Por fim, é possível a geração dos relatórios e gráficos que caracterizam cada parâmetro de busca estudado.

Outras atividades disponibilizadas pelo software são a execução de testes e geração de relatórios e gráficos a partir da chamada externa a um processo do software de estatística R. Várias modalidades de testes podem ser realizadas: testes individuais de combinação, testes de conjuntos de combinações e testes em lote para um conjunto de defeitos.

Diversas ferramentas foram usadas em conjunto para que todos os conceitos e técnicas definidas nas seções anteriores fossem explorados. Uma síntese das ferramentas utilizadas e suas respectivas finalidades podem ser assim enumeradas:

- Eclipse (Ganymede) com suporte a AspectJ: Execução de cenários predefinidos para as características que culminavam em defeitos de software. Captura de rastros de execução de software através de um componente desenvolvido com AspectJ [Sobreira e Maia 2008].

- Netbeans IDE (6.8) - Desenvolvimento de aplicativo para gerenciamento de código fonte, cadastro de defeitos, administração de testes de parâmetros de busca e emissão de relatórios gerenciais.

- Postgresql (8.4) - Banco de dados relacional utilizado para armazenar o código fonte de maneira estruturada, os rastros de execução, as descrições de defeitos e os resultados dos testes.

- R Statistical Computing (2.11.1) - Ferramenta estatística com adição de bibliotecas que davam suporte a LSI (pacote *lsa*) e geração de gráficos.

- RapidSVN - Aplicativo utilizado para realizar o *checkout* de versões diversas de código fonte a partir de repositórios *on-line*.

- iReport - Ferramenta utilizada para diagramação dos relatórios.

Todas estes softwares foram utilizados para compor a ferramenta administrativa de localização de defeitos definida anteriormente.

3.6 Resumo da Metodologia

Técnicas baseadas em recuperação de informação tentam identificar quais documentos possuem maior similaridade com uma consulta gerada pelo usuário. Ao se utilizar de tais técnicas para localizar defeitos, o resultado será uma lista de documentos que tenham maior similaridade com a consulta. Esta lista será utilizada pelo desenvolvedor para encontrar onde o defeito está localizado. Na realidade, o defeito pode estar em diversas partes do programa, e, portanto, a lista ordenada de métodos deveria idealmente ter todos os métodos relevantes para a correção do defeito ranqueados nas primeiras posições.

Em suma, a abordagem proposta granulariza métodos e seus respectivos comentários formando documentos que serão indexados pela técnica LSI. Em resposta a uma consulta formulada com base em descrições de defeitos de software, um conjunto destes documentos é retornado para análise do desenvolvedor. A posição do documento em relação ao topo da lista indica maior similaridade entre a implementação do método defeituoso e a consulta submetida.

Capítulo 4

Resultados

Nesta seção são analisados os resultados referentes a obtenção de uma combinação de parâmetros de entrada para a técnica LSI que satisfizesse o desenvolvedor durante a localização de defeitos no código fonte. O conjunto de combinações geradas por estes parâmetros foi testado com 10 defeitos do ArgoUML e 10 defeitos do JEdit. Uma análise quantitativa na tentativa de se encontrar uma melhor combinação foi realizada, baseada no número de métodos que demandaram análise em cada combinação. Durante a discussão dos resultados analisou-se a influência da utilização dos diferentes parâmetros de busca discutidos sobre a redução do número de métodos a serem analisados. Os parâmetros discutidos nesta seção de resultados são: filtros do código fonte, fator de redução da dimensionalidade do espaço vetorial, função de peso aplicada a matriz de termos-documentos e conjunto de *stop-words* utilizadas.

A primeira tentativa para se descobrir os valores de parâmetros que mais reduziam o número de métodos analisados foi a ordenação de todas as combinações possíveis de parâmetros obedecendo-se o menor número de métodos analisados. Esta tentativa objetivou verificar a existência de uma combinação que permitisse realizar sempre o menor trabalho de análise de métodos para os filtros - QTL e ET - e softwares - ArgoUML e JEdit.

A não obtenção da combinação satisfatória a partir da aplicação da metodologia proposta anteriormente motivou uma nova tentativa que levou em consideração o maior número de vezes que determinado valor de parâmetro produzia o menor trabalho para o desenvolvedor na tarefa de localização de defeitos. Esta nova caracterização do número de métodos analisados tomou por base os valores de parâmetros utilizados e foi expressa em termos percentuais em gráficos que abrangiam o universo de combinações possíveis para os outros parâmetros analisados. Deste estudo surgiu o que foi denominado combinação predominante, isto é, uma aproximação das melhores combinações que produziram um menor número de métodos analisados em todos os filtros de código fonte e softwares estudados.

Finalizada a busca pela combinação satisfatória (predominante), seus valores foram usados para avaliar a eficácia da aplicação dos filtros de código fonte propostos para

este trabalho. Ao se analisar a importância dos filtros de código fonte, para redução do esforço do desenvolvedor em encontrar os métodos relevantes a correção do defeito, foram utilizados os softwares JEdit e ArmedBear-J que pertencem a um mesmo domínio de aplicação e puderam ser indexados utilizando todo o código fonte conforme discutido na Seção 3.3. Neste estudo analisou-se o número de documentos do corpus obtido após a aplicação de cada filtro, o tempo necessário para indexação e teste, além da precisão encontrada em cada teste. A idéia principal foi contrastar os resultados obtidos nos testes que utilizaram os filtros propostos (ET e QTL) com uma abordagem que utilizou todo o código fonte.

Por fim, a análise da influência do parâmetro função de peso foi realizada tentando-se delinear a importância deste parâmetro em tarefas de localização de defeitos. O corrente trabalho procurou priorizar o estudo de algumas classes de palavras presentes no código fonte (nomes de classes, nomes de métodos e seus fragmentos) a partir da aplicação da função de peso. Tais palavras, ao encontrar correspondência na linguagem natural do usuário expressa nas consultas formuladas, mostraram-se relevantes, cada qual com determinado software, para diminuir o esforço de manutenção. Para explicitar claramente o alinhamento das classes de palavras citadas com os softwares estudados foram realizadas análises estatísticas adicionais. Tal estudo é de grande relevância pois um maior alinhamento das palavras do código fonte com a linguagem natural do usuário pode melhorar a compreensão do sistema afetando positivamente a manutenção de software.

4.1 Classificação de Combinações Baseada na Quantidade de Métodos Analisados

O objetivo principal deste trabalho foi verificar qual combinação de valores de parâmetros de entrada da técnica LSI atenuava mais acentuadamente o trabalho de manutenção de software. Para isto, um total de 288 combinações foram testadas para cada um dos 10 defeitos do ArgoUML, dentro de um contexto de localização de defeitos de software. Procedimento semelhante foi adotado para o JEdit. A Figura 4.1 ilustra como cada uma destas combinações foi gerada com base na combinação dos parâmetros estudados.

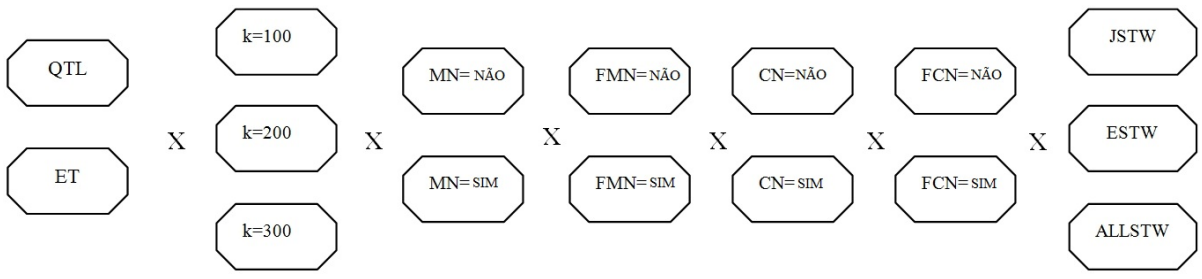


Figura 4.1: Combinações possíveis para os parâmetros estudados

Os testes realizados com o ArgoUML e JEdit permitiram a montagem das Tabelas 4.1 a 4.8. Tais tabelas condensam a classificação de todas as combinações possíveis de parâmetros e estão ordenadas pelo menor número de métodos analisados. Ao se analisar uma tabela deve-se observar os seguintes pontos:

- O software em estudo (ArgoUML ou JEdit).
- A metodologia para filtragem do corpus (ET ou QTL).
- O escopo de métodos vistoriados até se encontrar os métodos relevantes à correção dos defeitos (primeiro método encontrado ou todos os métodos que devem ser corrigidos em um conjunto de 10 defeitos por software).

Cada combinação testada foi classificada nas tabelas utilizando-se o número de métodos que precisaram ser vistoriados em cada conjunto de 10 defeitos analisados por software. Ao se testar uma combinação com um defeito varreu-se a lista de todos os métodos presentes no corpus guardando-se a posição do primeiro e do último método relevante à correção do defeito. A lista de métodos para cada par defeito/consulta foi montada em ordem decrescente de similaridade semântica da consulta com os documentos e foi apurada através da correlação de Pearson [Nakov et al. 2001]. Em cada conjunto de 10 defeitos analisados por software, totalizou-se o número de métodos vistoriados levando-se em consideração o primeiro e o último método. Os métodos relevantes à correção do defeito foram selecionados com base na leitura das mensagens relativas à correção do mesmo disponibilizadas nos *commits* gerados a cada alteração do software. Levando-se em consideração o defeito analisado na Seção 3.2, seu correspondente *commit* que promove a correção do defeito, está disponível na revisão de número 15169[1]. A proposta de se analisar o primeiro método a ser corrigido foi explorada por [Lukins et al. 2008] em um trabalho de localização de defeitos. Através das dependências do primeiro método encontrado pode-se tentar encontrar os outros métodos importantes para a correção do defeito. O trabalho corrente provê visões baseadas nos dois tipos de escopo de localização, sendo que as Tabelas 4.1 à 4.4, enfocam o primeiro método encontrado. Por outro lado, as Tabelas 4.5 à 4.8 estão relacionadas ao último método encontrado durante a análise de 10 defeitos de cada software.

Em cada tabela foi destacada em cinza a combinação predominante. Tal combinação foi resultante da análise individual dos valores de cada parâmetro da técnica LSI detalhada na Seção 4.2. Algumas combinações foram omitidas por questões de espaço sendo substituídas por reticências nas tabelas. No total, 144 combinações foram estudadas por tabela para cada filtro de código fonte aplicado (ET ou QTL) e escopo de métodos analisado (primeiro ou todos). O número 144 resulta das possíveis combinações para os parâmetros k (100;200;300), NM(NÃO;SIM), FNM(NÃO;SIM), CN(NÃO;SIM), FCN(NÃO;SIM), e

[1]http://argouml.tigris.org/ds/viewMessage.do?dsForumId=5905&dsMessageId=48732

RANK	K	MN	FMN	STW	CN	FCN	MÉTODOS
1	200	NÃO	NÃO	ALL	SIM	NÃO	91
2	300	SIM	NÃO	ALL	SIM	NÃO	98
3	300	NÃO	SIM	ALL	SIM	SIM	98
4	300	NÃO	SIM	ESTW	SIM	SIM	99
...							
13	200	NÃO	SIM	JSTW	SIM	SIM	103
14	300	SIM	SIM	JSTW	SIM	SIM	103
15	300	NÃO	SIM	ESTW	NÃO	SIM	104
16	300	SIM	SIM	ESTW	SIM	SIM	104
17	300	SIM	SIM	ALL	SIM	SIM	105
...							
135	100	SIM	SIM	JSTW	NÃO	NÃO	251
136	200	NÃO	SIM	ESTW	NÃO	NÃO	252
137	100	NÃO	SIM	ESTW	SIM	NÃO	254
138	200	NÃO	SIM	ALL	NÃO	NÃO	256
139	100	NÃO	SIM	JSTW	SIM	NÃO	260
140	200	NÃO	SIM	JSTW	NÃO	NÃO	261
141	100	NÃO	SIM	ALL	SIM	NÃO	262
142	100	NÃO	SIM	JSTW	NÃO	NÃO	286
143	100	NÃO	SIM	ESTW	NÃO	NÃO	289
144	100	NÃO	SIM	ALL	NÃO	NÃO	291

Tabela 4.1: *Classificação das Combinações para o ArgoUML - Primeiro Método - QTL*

RANK	K	MN	FMN	STW	CN	FCN	MÉTODOS
1	300	SIM	SIM	JSTW	SIM	SIM	75
2	300	SIM	SIM	ALL	SIM	SIM	78
3	300	NÃO	SIM	JSTW	SIM	SIM	80
4	300	SIM	SIM	ESTW	SIM	SIM	81
5	200	NÃO	SIM	ALL	SIM	SIM	81
6	300	NÃO	SIM	ALL	SIM	SIM	82
7	100	SIM	SIM	JSTW	SIM	SIM	82
8	200	SIM	SIM	JSTW	SIM	SIM	82
9	200	SIM	SIM	ESTW	SIM	SIM	84
10	300	NÃO	SIM	ESTW	SIM	SIM	85
...							
135	200	SIM	NÃO	ESTW	NÃO	SIM	322
136	100	NÃO	NÃO	ALL	NÃO	SIM	323
137	100	NÃO	NÃO	ESTW	NÃO	SIM	323
138	200	NÃO	NÃO	ESTW	NÃO	SIM	323
139	100	SIM	NÃO	ALL	NÃO	SIM	326
140	100	SIM	NÃO	ESTW	NÃO	SIM	330
141	100	SIM	NÃO	ALL	NÃO	NÃO	337
142	100	SIM	NÃO	ESTW	NÃO	NÃO	346
143	100	NÃO	NÃO	ALL	NÃO	NÃO	370
144	100	NÃO	NÃO	ESTW	NÃO	NÃO	385

Tabela 4.2: *Classificação das Combinações para o JEdit - Primeiro Método - QTL*

STW (ESTW; JSTW; ALLSTW), uma vez que se tenha fixado o filtro de código fonte aplicado.

A Tabela 4.1 refere-se ao software ArgoUML, utilizando o filtro QTL e focalizando o primeiro método a ser encontrado. A primeira linha da tabela permite constatar que utilizar a combinação de melhor ranking requereu a análise de 91 métodos para se encontrar o primeiro método que deveria ser corrigido em cada um dos 10 defeitos estudados para o ArgoUML. Isto retrata uma média de 9,1 métodos analisados por defeito e este é o trabalho médio despendido pelo desenvolvedor para a combinação de parâmetros mostrada. Além disso, esta combinação obteve um melhor desempenho utilizando um valor de k=200, todas as *stop-words* (ALLSTW) e aplicação da função de peso somente sobre o nome das classes. Por outro lado, a Tabela 4.2 referente ao JEdit, baseada no mesmo filtro de código fonte e escopo de análise de métodos, apresenta valores de parâmetros distintos, em relação à melhor combinação, quando comparada com a Tabela 4.1. Os valores de parâmetros que se sobressaem na primeira colocação da Tabela 4.2 são k=300, *stop-words* JSTW e peso aplicado sobre nomes de métodos, classes e seus fragmentos.

A observação das combinações que ocupam as primeiras posições em todas as Tabelas (4.1 à 4.8) não permite delinear uma melhor combinação para todos os defeitos analisados em ambos os sistemas (ArgoUML e JEdit). Pode-se observar, ao analisar a primeira linha das Tabelas 4.1 à 4.4, que os valores de parâmetros, com exceção do peso sobre o nome da classe, não se repetem. Fato semelhante acontece nas Tabelas 4.5 à 4.8. Não está clara uma maneira de se delinear valores satisfatórios de parâmetros, que atenda ambos

RANK	K	MN	FMN	STW	CN	FCN	MÉTODOS
1	300	NÃO	NÃO	ALL	SIM	NÃO	87
2	300	NÃO	SIM	ALL	SIM	SIM	97
3	300	NÃO	NÃO	ESTW	SIM	NÃO	99
4	300	NÃO	SIM	ESTW	SIM	SIM	99
5	300	SIM	SIM	ALL	SIM	SIM	100
6	300	SIM	SIM	ESTW	SIM	SIM	100
7	300	SIM	NÃO	ALL	SIM	NÃO	101
8	300	NÃO	SIM	JSTW	SIM	SIM	101
9	300	SIM	SIM	JSTW	SIM	SIM	101
10	300	NÃO	SIM	ESTW	NÃO	SIM	102
...							
135	300	NÃO	SIM	JSTW	NÃO	NÃO	250
136	100	NÃO	NÃO	JSTW	NÃO	NÃO	256
137	100	SIM	SIM	JSTW	NÃO	NÃO	257
138	100	NÃO	SIM	ESTW	SIM	NÃO	262
139	100	NÃO	SIM	JSTW	SIM	NÃO	267
140	100	NÃO	SIM	ALL	SIM	NÃO	271
141	100	NÃO	SIM	ESTW	NÃO	NÃO	277
142	100	SIM	SIM	ESTW	NÃO	NÃO	277
143	100	NÃO	SIM	ALL	NÃO	NÃO	280
144	100	NÃO	SIM	JSTW	NÃO	NÃO	290

Tabela 4.3: *Classificação das Combinações para o ArgoUML - Primeiro Método - ET*

RANK	K	MN	FMN	STW	CN	FCN	MÉTODOS
1	300	SIM	SIM	ALL	SIM	SIM	78
2	300	NÃO	SIM	ALL	SIM	SIM	79
3	300	SIM	SIM	ESTW	SIM	SIM	79
4	300	NÃO	SIM	ESTW	SIM	SIM	79
5	200	SIM	SIM	ALL	SIM	SIM	81
6	200	NÃO	SIM	ALL	SIM	SIM	81
7	300	NÃO	SIM	JSTW	SIM	SIM	82
8	300	SIM	SIM	JSTW	NÃO	SIM	82
9	200	NÃO	SIM	ALL	NÃO	SIM	82
10	200	NÃO	SIM	JSTW	SIM	SIM	83
...							
135	100	NÃO	NÃO	ALL	NÃO	SIM	353
136	100	SIM	NÃO	ALL	NÃO	SIM	363
137	100	NÃO	NÃO	ESTW	NÃO	NÃO	376
138	300	SIM	SIM	JSTW	SIM	SIM	434
139	200	SIM	SIM	JSTW	SIM	SIM	456
140	300	SIM	SIM	ALL	SIM	NÃO	509
141	100	SIM	SIM	JSTW	SIM	SIM	601
142	100	SIM	SIM	ALL	SIM	NÃO	868
143	200	SIM	SIM	ALL	SIM	NÃO	879
144	200	SIM	SIM	ESTW	SIM	NÃO	947

Tabela 4.4: *Classificação das Combinações para o JEdit - Primeiro Método - ET*

RANK	K	MN	FMN	STW	CN	FCN	MÉTODOS
1	300	SIM	NÃO	ALL	SIM	NÃO	104
2	300	SIM	NÃO	ESTW	SIM	NÃO	107
3	200	NÃO	NÃO	ESTW	SIM	NÃO	110
4	300	NÃO	NÃO	ALL	SIM	NÃO	114
5	300	SIM	SIM	ALL	SIM	SIM	114
6	200	NÃO	NÃO	ALL	SIM	NÃO	115
7	300	SIM	NÃO	JSTW	SIM	NÃO	116
8	300	SIM	SIM	JSTW	SIM	SIM	117
9	300	SIM	SIM	ESTW	SIM	SIM	118
10	300	NÃO	SIM	ALL	SIM	SIM	118
...							
135	100	NÃO	SIM	ALL	SIM	NÃO	284
136	100	SIM	SIM	JSTW	NÃO	NÃO	284
137	200	NÃO	NÃO	JSTW	NÃO	NÃO	288
138	100	NÃO	SIM	ESTW	SIM	NÃO	290
139	100	NÃO	SIM	JSTW	SIM	NÃO	291
140	100	NÃO	SIM	ESTW	NÃO	NÃO	310
141	100	NÃO	SIM	ALL	NÃO	NÃO	311
142	100	NÃO	SIM	JSTW	NÃO	NÃO	320
143	100	NÃO	NÃO	ALL	NÃO	NÃO	341
144	100	NÃO	NÃO	JSTW	NÃO	NÃO	351

Tabela 4.5: *Classificação das Combinações para o ArgoUML - Todos os Métodos - QTL*

RANK	K	MN	FMN	STW	CN	FCN	MÉTODOS
1	200	SIM	SIM	JSTW	NÃO	SIM	180
2	300	SIM	SIM	JSTW	NÃO	SIM	188
3	300	SIM	SIM	JSTW	NÃO	NÃO	188
4	300	SIM	SIM	JSTW	SIM	SIM	188
5	200	SIM	SIM	JSTW	NÃO	NÃO	191
6	200	NÃO	SIM	JSTW	NÃO	SIM	196
7	300	NÃO	SIM	JSTW	SIM	SIM	197
8	300	NÃO	SIM	JSTW	NÃO	SIM	202
9	300	NÃO	SIM	JSTW	NÃO	NÃO	209
10	300	NÃO	SIM	JSTW	SIM	NÃO	216
...							
135	200	NÃO	NÃO	ALL	NÃO	NÃO	424
136	100	SIM	NÃO	JSTW	NÃO	SIM	433
137	200	NÃO	NÃO	ESTW	NÃO	SIM	433
138	100	NÃO	NÃO	JSTW	NÃO	NÃO	446
139	100	SIM	NÃO	JSTW	NÃO	NÃO	458
140	100	SIM	NÃO	ESTW	NÃO	NÃO	463
141	100	SIM	NÃO	ESTW	NÃO	SIM	470
142	100	NÃO	NÃO	ESTW	NÃO	SIM	472
143	100	NÃO	NÃO	ALL	NÃO	NÃO	478
144	100	NÃO	NÃO	ESTW	NÃO	NÃO	508

Tabela 4.6: *Classificação das Combinações para o JEdit - Todos os Métodos - QTL*

RANK	K	MN	FMN	STW	CN	FCN	MÉTODOS
1	300	NÃO	NÃO	ALL	SIM	NÃO	107
2	300	SIM	NÃO	ALL	SIM	NÃO	113
3	300	SIM	NÃO	ESTW	SIM	NÃO	114
4	300	NÃO	NÃO	ESTW	SIM	NÃO	116
5	300	SIM	NÃO	JSTW	SIM	NÃO	117
6	300	NÃO	NÃO	JSTW	SIM	NÃO	122
7	300	NÃO	SIM	ALL	SIM	SIM	122
8	300	NÃO	SIM	ESTW	SIM	SIM	124
9	300	SIM	SIM	ALL	SIM	SIM	125
10	300	SIM	SIM	ESTW	SIM	SIM	127
...							
135	100	NÃO	SIM	ALL	SIM	NÃO	289
136	200	NÃO	NÃO	JSTW	NÃO	NÃO	292
137	100	NÃO	SIM	ESTW	NÃO	NÃO	299
138	100	NÃO	SIM	JSTW	SIM	NÃO	299
139	100	NÃO	SIM	ALL	NÃO	NÃO	299
140	100	SIM	SIM	ESTW	NÃO	NÃO	305
141	100	NÃO	NÃO	ESTW	NÃO	NÃO	313
142	100	NÃO	SIM	JSTW	NÃO	NÃO	314
143	100	NÃO	NÃO	JSTW	NÃO	NÃO	344
144	100	NÃO	NÃO	ALL	NÃO	NÃO	346

Tabela 4.7: *Classificação das Combinações para o ArgoUML - Todos os Métodos - ET*

RANK	K	MN	FMN	STW	CN	FCN	MÉTODOS
1	300	SIM	SIM	JSTW	NÃO	SIM	169
2	300	SIM	SIM	JSTW	NÃO	NÃO	188
3	300	NÃO	SIM	JSTW	NÃO	SIM	191
4	300	SIM	SIM	ALL	NÃO	SIM	196
...							
14	200	NÃO	SIM	JSTW	NÃO	NÃO	218
15	300	SIM	SIM	ALL	SIM	SIM	219
16	200	NÃO	SIM	JSTW	NÃO	SIM	220
17	300	NÃO	SIM	ALL	NÃO	SIM	223
18	300	NÃO	SIM	JSTW	SIM	SIM	223
...							
135	100	NÃO	NÃO	ALL	NÃO	NÃO	497
136	100	NÃO	NÃO	ESTW	NÃO	SIM	502
137	100	NÃO	NÃO	ESTW	NÃO	NÃO	525
138	300	SIM	SIM	JSTW	SIM	SIM	564
139	200	SIM	SIM	JSTW	SIM	SIM	621
140	300	SIM	SIM	ALL	SIM	NÃO	641
141	100	SIM	SIM	JSTW	SIM	SIM	759
142	100	SIM	SIM	ALL	SIM	NÃO	1040
143	200	SIM	SIM	ALL	SIM	NÃO	1045
144	200	SIM	SIM	ESTW	SIM	NÃO	1119

Tabela 4.8: *Classificação das Combinações para o JEdit - Todos os Métodos - ET*

os softwares, com base na simples observação das melhores combinações que aparecem na primeira posição.

Em linhas gerais, não foi possível identificar uma única combinação que figurasse na primeira posição em todas as classificações das variadas combinações de parâmetros, expressas nas tabelas. Diante deste fato, buscou-se uma maneira alternativa de elicitar quais valores de parâmetros poderiam servir de guia para futuras consultas submetidas pelo desenvolvedor na localização de defeitos nos softwares citados. Tal combinação, mostrada pela nova metodologia aqui empregada, deveria proporcionar um menor trabalho ao desenvolvedor ou pelo menos um trabalho aceitável quando comparado às combinações que figuram nas primeiras posições das tabelas. Esta combinação denominada predominante é explorada na próxima seção.

As Tabelas 4.3 (100 métodos analisados na combinação predominante) e 4.4 (78 métodos analisados na combinação predominante) também enfatizam a busca pelo primeiro método a ser corrigido. Como elas se diferenciam respectivamente das Tabelas 4.1 (103 métodos analisados na combinação predominante) e 4.2 (75 métodos analisados na combinação predominante) somente pelo filtro de código fonte aplicado, pode-se observar que a mudança de filtro não implica uma diminuição significativa do número de métodos analisados quando se verifica a combinação predominante. Este fato dá indícios de que não existe diferença significativa entre os filtros ET e QTL. Para comprovar tal assertiva, testes estatísticos são realizados na Seção 4.4.

Outra observação que se pode fazer com base em uma análise visual das tabelas é

que combinações que não envolviam qualquer aplicação da função de peso implicaram a análise de um volume maior de métodos e figuraram em uma classificação ruim no ranking de combinações:

- Tabela 4.1: posições 143 e 144.
- Tabela 4.2: posições 143 e 144.
- Tabela 4.3: posição 136.
- Tabela 4.4: posição 137.
- Tabela 4.5: posições 137, 143 e 144.
- Tabela 4.6: posições 135, 138, 143 e 144.
- Tabela 4.7: posições 136, 143 e 144.
- Tabela 4.8: posições 135 e 137.

Esta hipótese é sugerida como um trabalho futuro, portanto não tratada neste trabalho, que vise analisar a eficiência da função de peso na melhoria do ranking de métodos que sejam mais relevantes à correção dos defeitos. A função de peso utilizada neste trabalho, como definida na Seção 3.3, serve para aumentar a importância relativa de algumas classes de palavras dentro do código fonte verificando seu alinhamento com o vocabulário do usuário.

4.2 A Combinação Predominante

Uma nova metodologia foi proposta para a escolha dos valores de parâmetros de busca a serem empregados como um padrão de uso pelos desenvolvedores na tarefa contínua de localizar defeitos no código fonte dos softwares estudados.

No contexto deste trabalho, a predominância de determinado valor de parâmetro de busca é estabelecida pela contagem do número de vezes que este valor proporciona o menor número de métodos analisados. Esta contagem é realizada no universo das possíveis combinações geradas a partir dos outros parâmetros de busca. O diagrama de atividades da Figura 4.2 explicita a metodologia proposta.

Levando-se em consideração a busca do valor predominante do parâmetro de redução de dimensionalidade k, foram observadas quantas vezes um dos valores de k (100, 200 ou 300) implicou a análise do menor número de métodos em mais de 50% das combinações geradas pelos outros parâmetros. Esta observação baseou-se em 10 defeitos de cada software estudado.

Para exemplificar a metodologia proposta, foram escolhidos o software ArgoUML e a abordagem QTL. Com base na Figura 4.3, um filtro de código fonte pode ser combinado

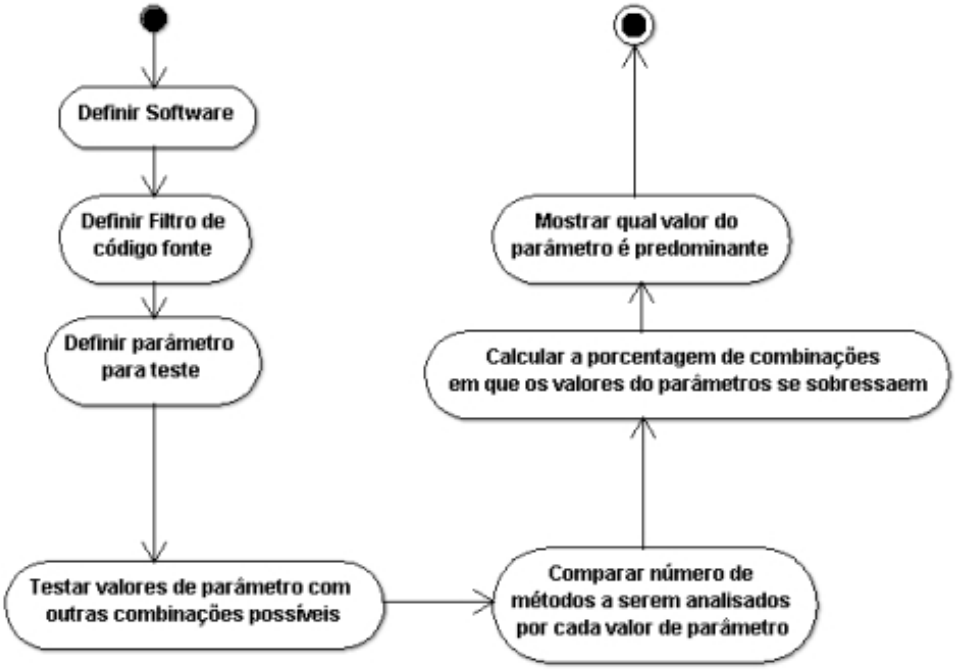


Figura 4.2: Diagrama de atividades para busca de valores predominantes de parâmetros

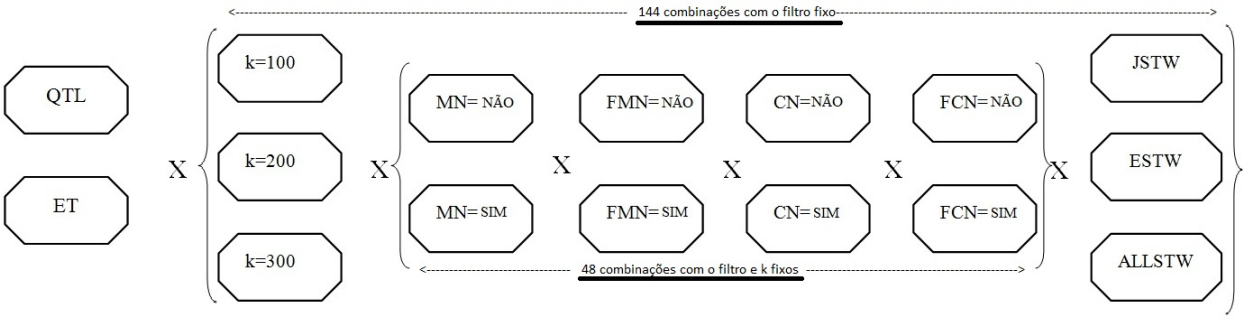


Figura 4.3: Combinações possíveis fixando-se o filtro de código fonte e o valor de k

com 144 possíveis variações distintas de outros parâmetros. Com o filtro fixo verifica-se que um valor discreto de k (100, 200 ou 300) pode ser combinado com 48 possíveis variações distintas dos parâmetros restantes. Em cada uma das 48 combinações possíveis, observa-se qual valor de k corresponde a um menor trabalho realizado para se corrigir os 10 defeitos do ArgoUML. Para cada combinação testada foram gerados dois gráficos: um para plotagem das precisões encontradas nos testes da combinação para cada um dos defeitos e outro para avaliação do número de métodos vistoriados por valor de k, para se encontrar todos os métodos a serem corrigidos por defeito. Assim, o gráfico da Figura 4.4 ilustra as precisões encontradas nos testes de cada um dos 10 defeitos do ArgoUML para cada um dos 3 valores distintos de k. O gráfico da Figura 4.5 ilustra o trabalho realizado para uma das 48 combinações possíveis [NM=SIM, FNM=NÃO, CN=SIM, FCN=NÃO e STW=JSTW] e a variação de valores do parâmetro k. Os valores

das precisões encontradas nos testes foram muito importantes para comparação dos filtros ET e QTL realizada na Seção 4.4. O trabalho realizado pelo desenvolvedor, expresso em número de métodos analisados, subsidiou a obtenção da combinação predominante.

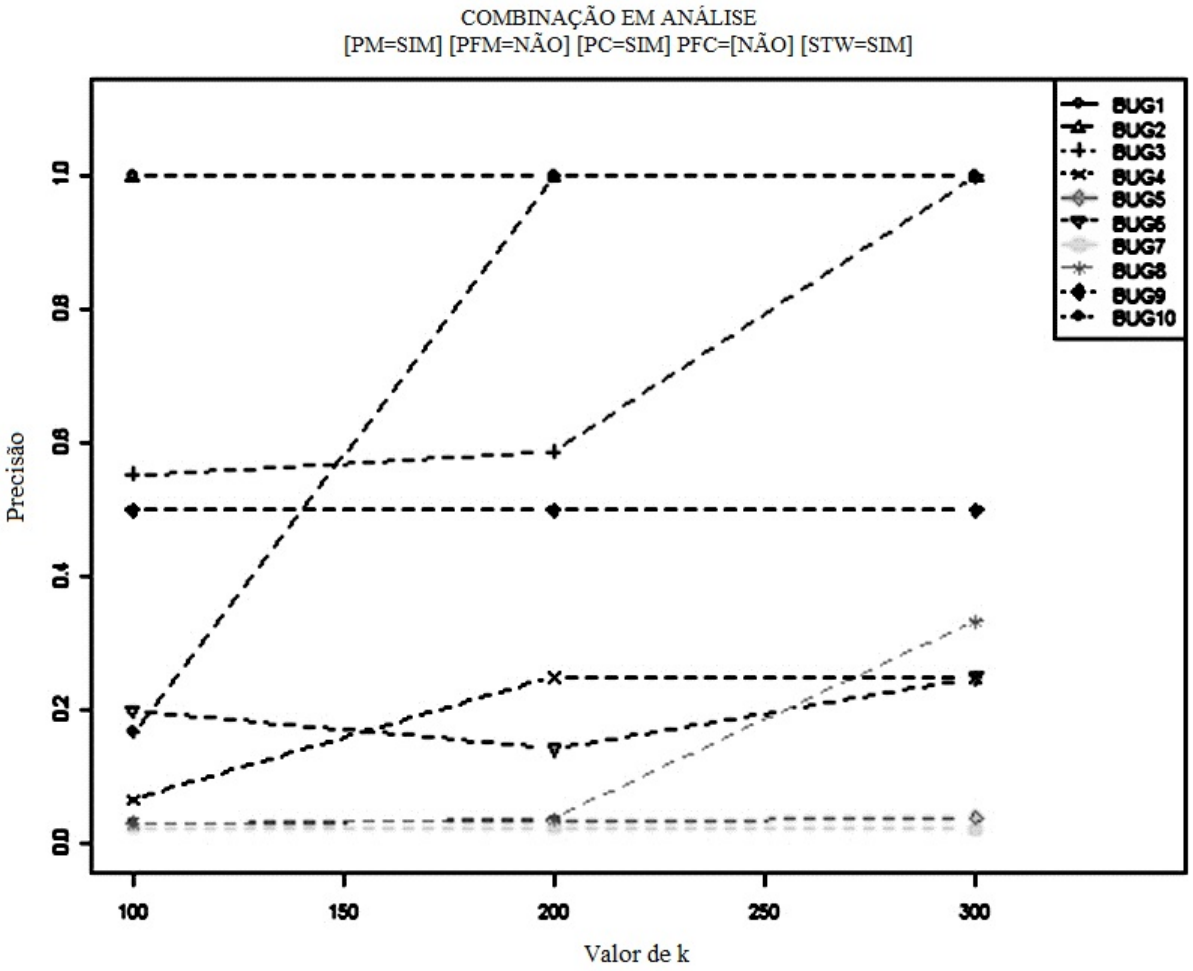


Figura 4.4: Gráfico Precisão versus k para 10 bugs do ArgoUML

No gráfico da Figura 4.5, o valor de k=300 foi o que resultou em um menor número de métodos analisados para se encontrar todos os métodos a serem corrigidos em uma amostra de 10 defeitos do ArgoUML. No total de 48 gráficos traçados de maneira semelhante, levantou-se as porcentagens relativas a um menor trabalho realizado para cada valor de k, chegando-se aos gráficos das Figuras 4.6 (a) e 4.6 (b) para os filtros ET e QTL respectivamente.

O valor de k=300 predominou sobre os outros valores em 81,6% de todas as 48 combinações analisadas do ArgoUML, para filtro QTL. Analogamente, para cada parâmetro estudado identificou-se a porcentagem de combinações em que os valores propostos para os parâmetros de busca implicavam um menor trabalho. A partir desta observação de predominância de valores, para os softwares ArgoUML e JEdit, algumas diretrizes foram traçadas no sentido de orientar o desenvolvedor na escolha da calibragem satisfatória de parâmetros em coleções de texto formadas por fragmentos do código fonte destes 2

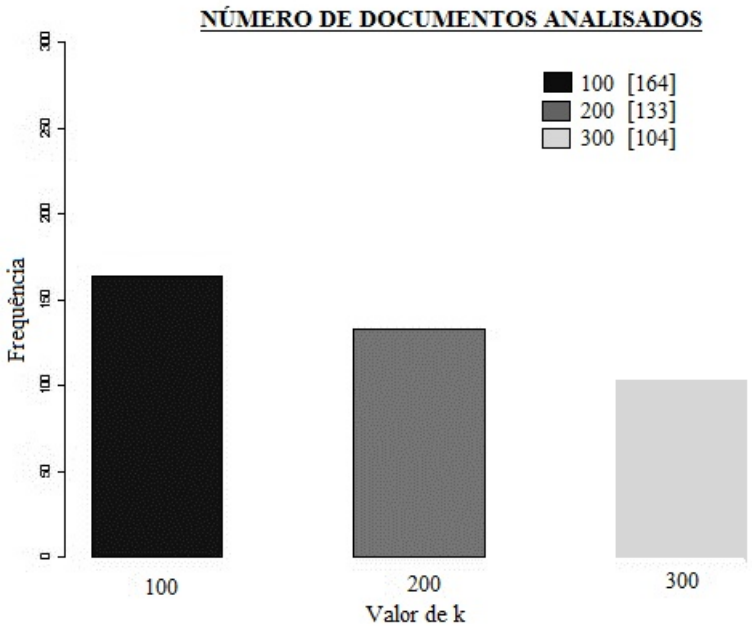


Figura 4.5: Trabalho realizado por valor de k

sistemas escritos na linguagem Java. A predominância para todos os demais parâmetros estudados são mostradas nos gráficos presentes nas Figuras 4.7 (peso sobre nome dos métodos), 4.8 (peso sobre fragmentos do nome do método), 4.9 (peso sobre nome das classes), 4.10 (peso sobre fragmentos dos nomes das classes), 4.11 (*stop-words*). Os gráficos abordam os dois filtros de código fonte ET e QTL. A partir da observação dos gráficos foram escolhidos cada um dos valores de parâmetros para compor a combinação predominante.

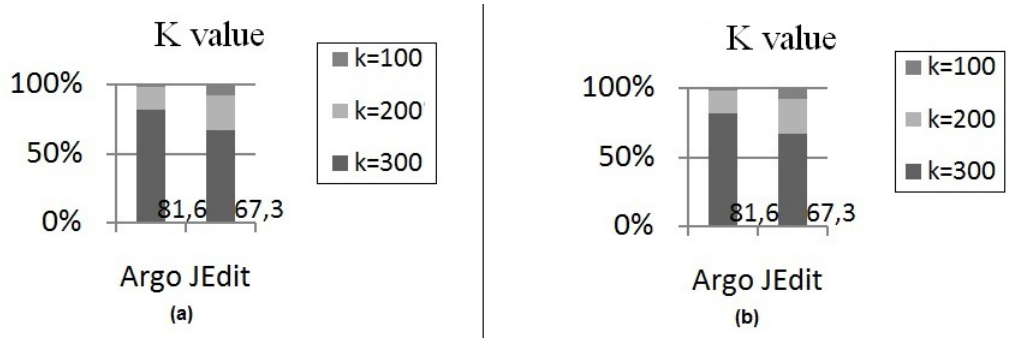


Figura 4.6: Predominância dos valores de k - abordagens ET e QTL respectivamente

Levando-se em consideração o parâmetro de redução de dimensionalidade k, pode-se ver pelos gráficos da Figura 4.6 que tanto no ArgoUML quanto no JEdit é grande a incidência do valor 300 como aquele que resultou no menor número de métodos analisados frente aos outros valores 100 e 200. O valor de k foi assim escolhido para compor a combinação predominante. A menor porcentagem de predominância para o valor de k=300 foi de 67,3% (Figura 4.6 (b)) em todos os conjuntos de 48 combinações analisados. Os outros parâmetros podem ser assim resumidos:

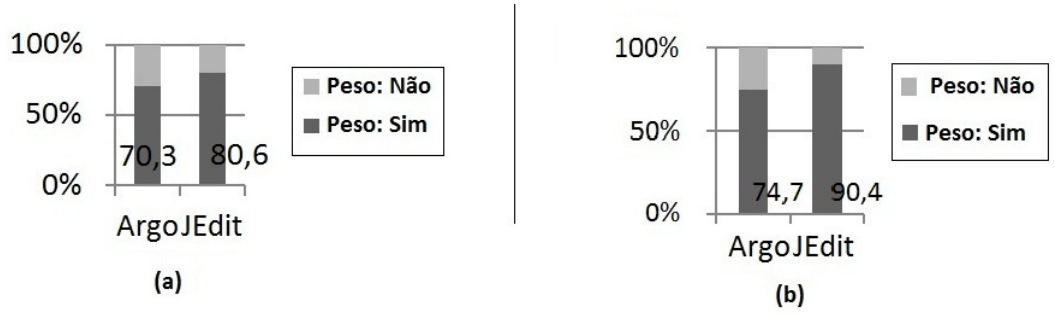


Figura 4.7: Predominância na aplicação da função de peso sobre o nomes dos métodos - abordagens ET e QTL respectivamente

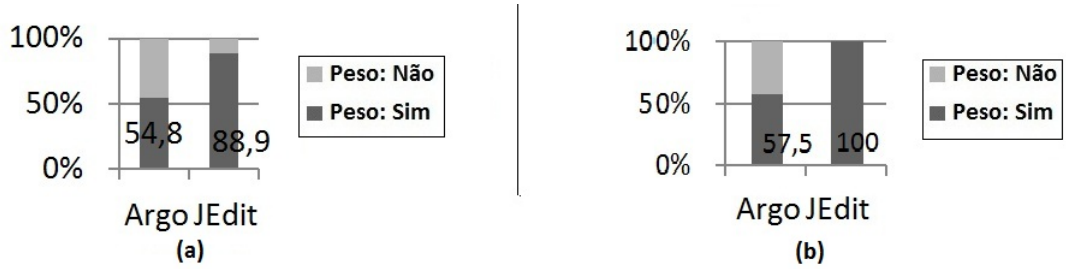


Figura 4.8: Predominância na aplicação da função de peso sobre os fragmentos do nomes dos métodos - abordagens ET e QTL respectivamente

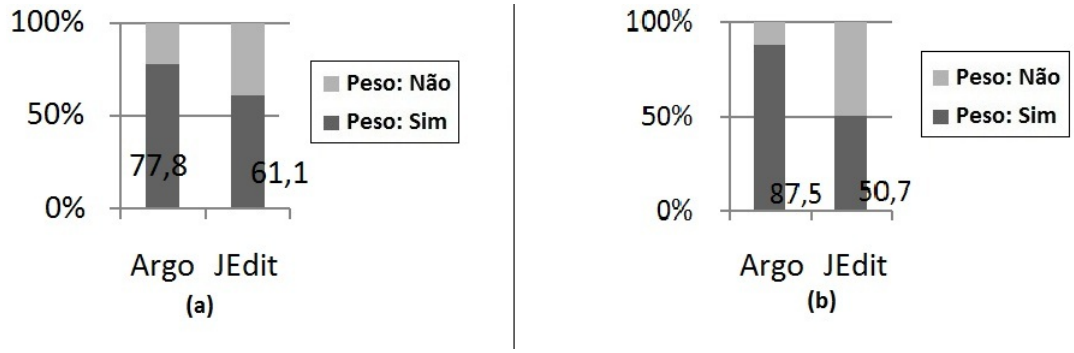


Figura 4.9: Predominância na aplicação da função de peso sobre o nomes das classes - abordagens ET e QTL respectivamente

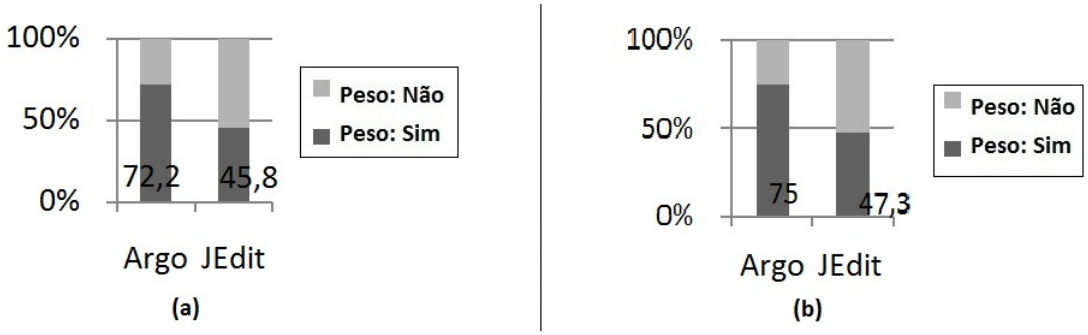


Figura 4.10: Predominância na aplicação da função de peso sobre os fragmentos do nomes das classes - abordagens ET e QTL respectivamente

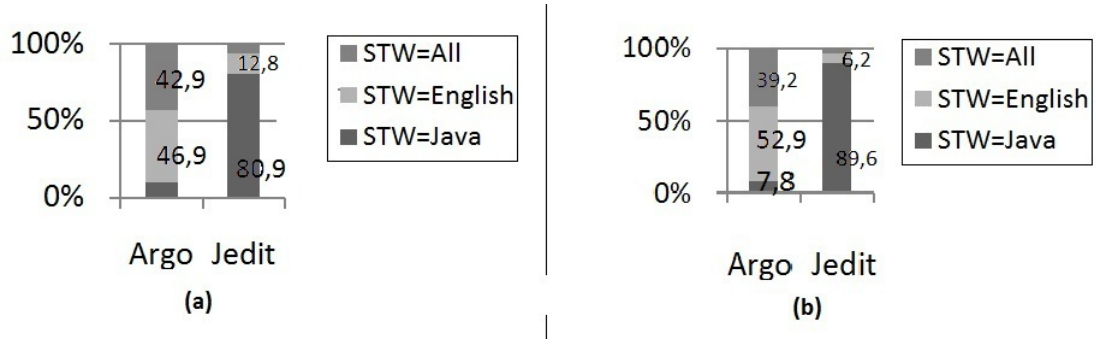


Figura 4.11: Predominância de *stop-words* - abordagens ET e QTL respectivamente

- Peso sobre o nome do método: Porcentagem acima de 70,3% em todos os conjuntos de 72 combinações analisadas (Figura 4.7). Optou-se por utilizar peso neste parâmetro.
- Peso sobre os fragmentos do nome do método: Porcentagem acima de 54,8% em todos os conjuntos de 72 combinações analisadas (Figura 4.8). Optou-se por utilizar peso neste parâmetro.
- Peso sobre o nome da classe: Porcentagem acima de 50,7% em todos os conjuntos de 72 combinações analisadas (Figura 4.9). Optou-se por utilizar peso neste parâmetro.
- Peso sobre fragmentos do nome da classe: Porcentagem acima de 45,8% em todos os conjuntos de 72 combinações analisadas (Figura 4.10). Como trata-se de uma análise conjunta entre ArgoUML e JEdit optou-se por utilizar peso neste parâmetro. Mesmo que se considere que a porcentagem para o JEdit está abaixo de 50%, a diferença é de apenas 4,2 pontos percentuais e levaria a um empate na escolha entre se utilizar ou não peso. Desta forma, optou-se pela utilização de peso em função da porcentagem apresentada pelo ArgoUML que está acima de 72,2%.
- Conjunto de *stop-words*: Analisando a Figura 4.11, ficou caracterizado que utilizar *stop-words* da língua Inglesa (ESTW) para o ArgoUML e *stop-words* da linguagem Java (JSTW) para o JEdit é mais vantajoso quando se deseja diminuir o número de métodos analisados. Desta forma, não se optou por sugerir um valor predominante para este parâmetro.

Com base nos valores considerados predominantes, pode-se expressar a combinação predominante: (k=300, MN=SIM, FMN=SIM, CN=SIM, FCN=SIM). De fato, como pode ser visto pela Tabela 4.2 esta é a combinação que fornece o melhor resultado para o JEdit. Contudo, não podemos verificar o mesmo para o ArgoUML com base na Tabela 4.1. Como já foi comentado, a combinação predominante não é a melhor combinação para todos os casos mas representa um avanço na tentativa de caracterização do conjunto de parâmetros a serem utilizados em processos de busca envolvendo LSI.

4.2.1 Efeitos da Aplicação de Diferentes Conjuntos de *stop-words* sobre a Manutenção de Software

Este trabalho estudou a aplicação de diferentes conjuntos de *stop-words*, sobre o código fonte, na tentativa de reduzir os esforços para a localização de defeitos. Os conjuntos de *stop-words* foram definidos na Seção 3.3 e compreendem: JSTW, ESTW e ALLSTW. Em relação a este parâmetro observou-se que os sistemas estudados mostraram-se sensíveis a sua aplicação. Conforme o software e *stop-words* aplicadas, houve importante variação no número de métodos a serem analisados.

Em síntese, observando-se a variação do parâmetro *stop-word* no universo de combinações de outros parâmetros, notou-se que:

- O software JEdit mostrou melhores resultados quanto à redução do trabalho realizado pelo desenvolvedor quando foi utilizado o conjunto de palavras da linguagem Java (JSTW).

- Tratando-se do ArgoUML, a predominância do conjunto de palavras da língua Inglesa (ESTW) mostrou notória vantagem em reduzir o trabalho em relação aos demais conjuntos de palavras.

Estas observações estão baseadas na atividade realizada para se encontrar a combinação predominante. Com base no gráfico da Figura 4.11, vemos que para ambos os filtros ET e QTL, confirmam-se as tendências de se utilizar ESTW para o ArgoUML e JSTW para o JEdit, no intuito de diminuir a quantidade de métodos analisados. O gráfico relata a predominância percentual de combinações onde cada conjunto de *stop-words* realizou o menor trabalho.

4.2.2 Comparando a Combinação Predominante com as Melhores Combinações

Retomando a análise do ArgoUML e observando o ranking das combinações para todos os métodos a serem corrigidos, podemos notar que a combinação predominante figura na 5ª posição (Tabela 4.5) e 9ª posição (Tabela 4.7), para as abordagens QTL e ET, respectivamente. Para o JEdit, as posições no ranking são 4ª (Tabela 4.6) e 15ª (Tabela 4.8) para as abordagens QTL e ET, respectivamente. Para se encontrar o primeiro método a ser alterado em 10 defeitos analisados do ArgoUML, a combinação em questão foi ranqueada na 14ª posição (Tabela 4.1) usando a abordagem QTL e na 5ª posição (Tabela 4.3) usando a abordagem ET. No caso dos 10 defeitos do JEdit, tal combinação apareceu na 1ª posição (Tabelas 4.2 e 4.4) para ambas as abordagens QTL e ET.

A metodologia proposta permite identificar a combinação predominante entre as 15 melhores combinações em todas as combinações de parâmetros possíveis para todos os

softwares, visões de escopo de métodos analisados e filtros de código fonte aplicados. Para se ter uma idéia do que isto significa em função do número de métodos analisados pelo desenvolvedor, algumas considerações importantes sobre a combinação predominante podem ser feitas:

- A diferença máxima é de 50 unidades, em número de métodos analisados, entre a melhor combinação e a predominante na pior classificação encontrada para esta última (Tabela 4.8 - JEdit - ET - todos os métodos). Nesta situação foram analisados 169 métodos pela melhor combinação, 219 pela predominante e 1119 métodos pela pior combinação. A combinação predominante ficou na 15ª posição do ranking para um total de 144 combinações analisadas.

- Em todas as classificações propostas utilizando o filtro de código fonte QTL (Tabelas 4.1,4.2,4.5 e 4.6), a combinação predominante não dista mais que 12 métodos em relação à melhor combinação (Tabela 4.1). Nesta situação foram analisados 91 métodos pela melhor combinação, 103 pela predominante e 291 métodos pela pior combinação. A combinação predominante ficou na 14ª posição do ranking para um total de 144 combinações analisadas.

- Em todas as classificações propostas para o filtro de código fonte ET (Tabelas 4.3,4.4,4.7 e 4.8), a combinação predominante não dista mais que 50 métodos em relação à melhor combinação. Em 75% destas classificações (Tabelas 4.3,4.4 e 4.7), a combinação predominante não dista mais que 18 métodos em relação à melhor combinação.

A combinação predominante foi o resultado do teste de um conjunto de parâmetros selecionados com base em outros trabalhos científicos [Nakov et al. 2001], [Kontostathis 2007], [Santhiappan e Gopalan 2010], [Santhiappan et al. 2006]. Sua obtenção delimita uma nova metodologia para que outros softwares possam ser testados e caracterizados quanto a valores satisfatórios que proporcionem um menor trabalho em tarefas de manutenção de software.

4.3 Análise das Abordagens para Filtragem de Código Fonte

Neste trabalho foram estudados três filtros de código fonte: ALL, ET e QTL. Eles foram definidos na Seção 3.3 e objetivavam a redução do número de documentos do corpus analisado pela técnica LSI.

O gráfico da Figura 4.12 mostra um comparativo entre o número de documentos presentes no corpus após a aplicação de cada filtro de código fonte. Este estudo se baseou

em defeitos do JEdit e ArmedBear-J. É importante notar que o gráfico está em escala logarítmica e mostra uma drástica redução no número de documentos quando altera-se o filtro de ALL para ET. A passagem do filtro ET para QTL não é tão significativa mas consegue eliminar ainda mais o número de métodos irrelevantes a consulta submetida pelo desenvolvedor, preservando a revocação para os testes.

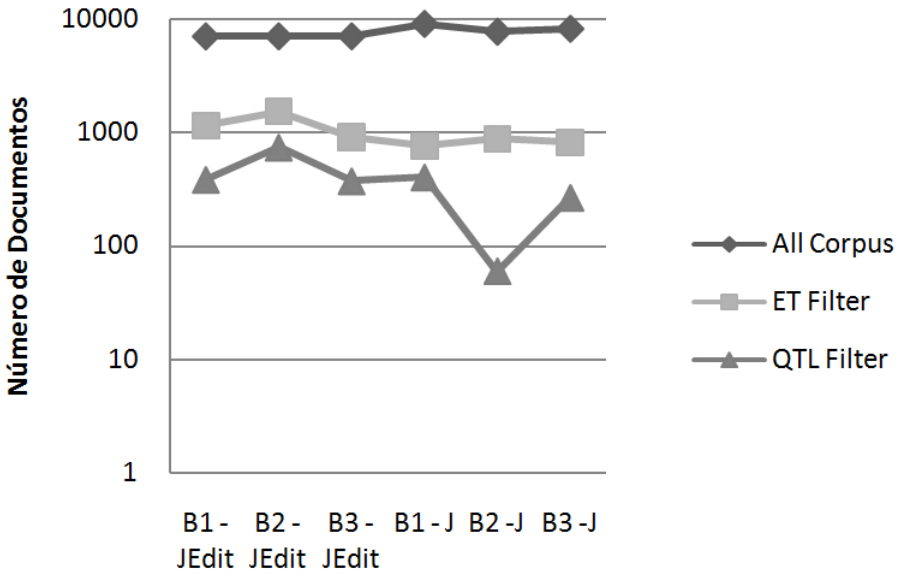


Figura 4.12: Número de documentos no corpus

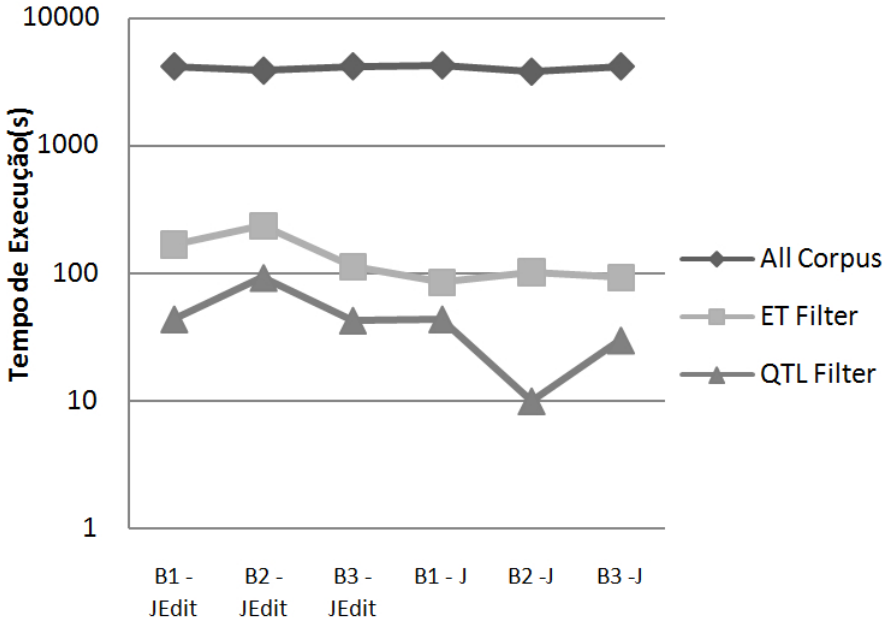


Figura 4.13: Tempo necessário para indexar e processar o teste

Com a visão geral do número de documentos analisados em cada filtro aplicado, é importante verificar agora quanto tempo é necessário para indexar os documentos e realizar os testes para os defeitos citados previamente. O gráfico presente na Figura 4.13,

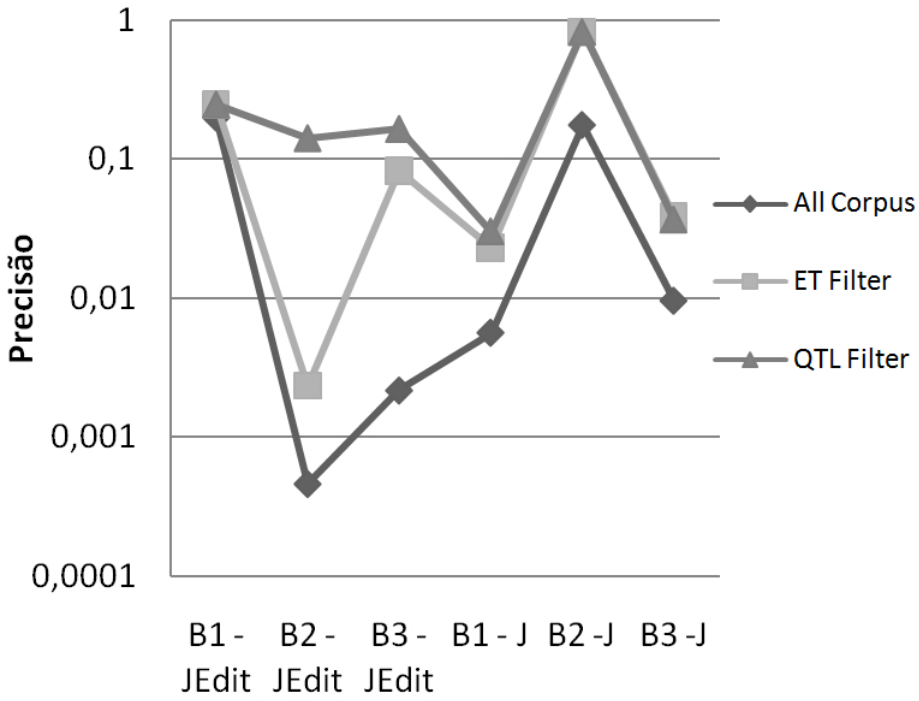


Figura 4.14: Precisões obtidas nos testes com o ArmedBear-J e JEdit

montado utilizando novamente uma escala logarítmica, expressa o tempo relativo a cada indexação e teste para os defeitos estudados. Nota-se, através da comparação com o gráfico da Figura 4.12, a existência de uma clara correlação entre o tamanho do corpus e o tempo de indexação e realização de testes.

Por último, os resultados concernentes às precisões encontradas nos testes estão expressos no gráfico 4.14. Nota-se que a redução do número de documentos no corpus melhorou significativamente a precisão quando comparadas as precisões encontradas nos filtros propostos (ET e QTL) com as precisões para todo o código fonte (filtro ALL). A redução no número de documentos do corpus proporcionada pela aplicação dos filtros, o tempo de indexação e teste além das precisões encontradas, sugerem ao desenvolvedor que opte pelo filtro QTL. O filtro QTL é realmente importante para se obter um ponto de corte na análise de métodos. O ponto de corte é definido neste trabalho como a quantidade de métodos analisados até que o desenvolvedor encerre a atividade de analisar e corrigir determinado defeito, isto é, a posição em que o desenvolvedor encerra sua análise na lista de métodos ranqueados pela técnica LSI. Embora os resultados tenham mostrado que a abordagem QTL produziu melhor resultado na maioria dos casos no que se refere à precisão, isto não foi verdadeiro em todos os casos como pode ser verificado no gráfico da Figura 4.14. Algumas precisões encontradas nos filtros ET e QTL para alguns defeitos são iguais: B1-JEdit, B1-J, B2-J e B3-J. Isto motiva novas análises comparativas entre os dois filtros no tocante à precisão, sendo alvo de estudo na Seção 4.4.

Outro importante resultado está relacionado às revocações obtidas durante os testes. Sem a aplicação de qualquer filtro no código fonte, todos os métodos do código fonte seriam

projetados no espaço vetorial semântico. Nesta situação têm-se sempre uma revocação igual a 100% pois é possível calcular a similaridade da consulta com todos os métodos do código fonte. Contudo, ao aplicar os filtros propostos, incorre-se na possibilidade de que métodos responsáveis pela manifestação do defeito fiquem de fora do corpus resultante. Caso isto fosse verdadeiro, alguns métodos não apareceriam na lista final ordenada e a medida revocação não seria 100%. Neste sentido, foi observado que o terceiro defeito do ArmedBear-J que tinha 6 métodos a serem alterados para correção do defeito teve sua revocação inferior a 100%. Após a aplicação do filtro QTL, um destes métodos foi descartado, não fazendo parte do corpus analisado, e, portanto, não sendo listado e encontrado. Para todos os outros 22 defeitos analisados, as abordagens ET e QTL não removeram métodos relevantes à correção dos respectivos defeitos.

Todos os dados que permitiram montar os gráficos comparativos entre os software ArmedBear-J e JEdit podem ser encontrados no Apêndice D.

O resultado deste trabalho sugere fortemente que as abordagens para filtragem de código fonte influenciam o tempo para indexar o corpus e a precisão dos testes conforme estudado nesta seção.

4.4 Análise Estatística Complementar para os Filtros de Código Fonte e Função de Peso

Esta seção aborda duas questões que ficaram em aberto após a conclusão dos testes realizados nas seções anteriores:

- A questão relativa à existência de diferença entre as precisões encontradas nos testes abordando os filtros QTL e ET.
- O alinhamento entre as classes de palavras estudadas e implementadas no código fonte (nomes de métodos, classes e seus fragmentos) com as descrições de defeitos.

As amostras populacionais analisadas na tentativa de se responder as duas questões levantadas foram compostas respectivamente pelas precisões e número de métodos analisados pelas combinações dos testes envolvendo os softwares ArgoUML e JEdit durante a busca pela combinação predominante. Foram usados os testes estatísticos de Mann-Whitney-Wilcoxon que não pressupõem a utilização de amostras populacionais normais. O teste de Wilcoxon, também denominado Mann-Whitney ou MWW, é um teste não paramétrico usado para determinar se existe diferença entre duas populações [Anderson et al. 2009]. Ele foi proposto inicialmente por Frank Wilcoxon em 1945 para análise de populações de mesmo tamanho. Posteriormente, foi estendido para outras populações de tamanho arbitrário em 1947 por Henry B. Mann e Donald R. Whitney [Salkind 2010]. Testes não paramétricos referem-se a métodos de mensuração estatística que não se apóiam

na suposição de que os dados sejam oriundos de determinadas distribuições. Eles são amplamente usados no meio científico para realização de inferências estatísticas. Na prática, quando a suposição de normalidade dos dados não é satisfeita para uma população em estudo, testes paramétricos podem fornecer resultados incorretos.

4.4.1 Diferenciação entre os filtros de código fonte ET e QTL

Em um primeiro momento, fez-se uma análise das precisões encontradas em combinações que utilizavam os filtros QTL e ET na tentativa de se elicitar um filtro que fosse superior ao outro quanto à obtenção de maiores precisões nos testes. No teste de Wilcoxon realizado na tentativa de se diferenciar as abordagens QTL e ET as amostras foram compostas pelas precisões encontradas nos testes com os 10 defeitos do ArgoUML e JEdit. As hipóteses formuladas para o teste de Wilcoxon foram:

- Hipótese nula (H_0) - Não existe diferença significativa entre a média das precisões das abordagens ET e QTL.
- Hipótese alternativa (H_a)- Existe diferença significativa entre a média das precisões das abordagens ET e QTL.

Os resultados são os seguintes:

1. | Software: ArgoUML | Número de defeitos: 10 | Total de testes analisados: 2880|
   | Total de precisões abordagem ET: 1440| Total de precisões abordagem QTL: 1440|
   wilcox.test(pqtl, pet)
   dados: pqtl and pet
   W = 1060720; p-value = 0,2819

2. | Software: JEdit | Número de defeitos: 10 | Total de testes analisados: 2880|
   | Total de precisões abordagem ET: 1440| Total de precisões abordagem QTL: 1440|
   wilcox.test(pqtl, pet)
   dados: pqtl and pet
   W = 1043287; p-value = 0,7712

O valor de p-value maior que 0,05 encontrado nos dois testes indica que devemos aceitar a hipótese nula, em que não existe diferença significativa entre a média dos dois filtros, em detrimento da hipótese alternativa. Este resultado pode ser confirmado visualmente por meio do *boxplot* das duas amostras citadas para cada um dos softwares estudados: Figuras 4.15 e 4.16. Por esta razão os testes de Wilcoxon subsequentes foram realizados levando-se em consideração as duas abordagens para filtro de código fonte.

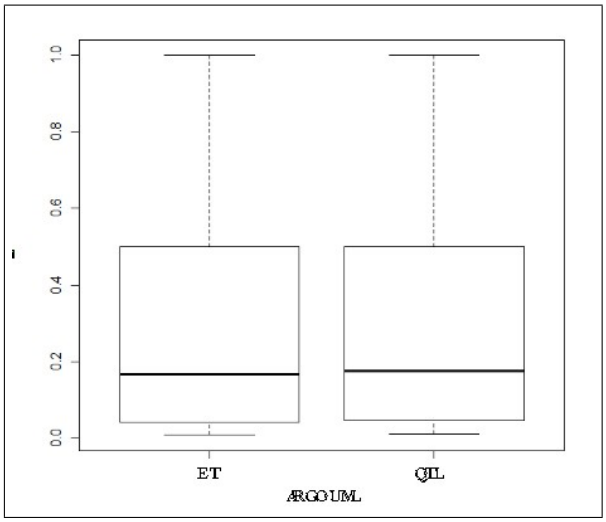


Figura 4.15: *Boxplot dos resultados dos testes do ArgoUML para cada filtro*

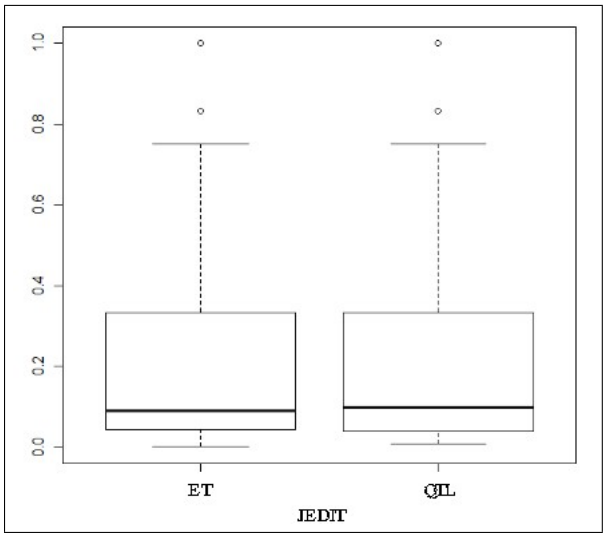


Figura 4.16: *Boxplot dos resultados dos testes do JEdit para cada filtro*

Resumindo, os resultados do teste estatístico aplicado não permitem afirmar que existe diferença notória entre as precisões encontradas nos dois filtros estudados, apesar do filtro QTL estabelecer um ponto de parada mais satisfatório ao desenvolvedor, isto é, na lista ranqueada de métodos o desenvolvedor interrompe seu trabalho de análise primeiramente utilizando o filtro QTL. Isto se deve ao fato do número de documentos no corpus originado pelo filtro QTL ser menor do que aquele originado pelo filtro ET, conforme visto na Seção 4.3.

4.4.2 Proposição de Alinhamentos Semânticos com base na Aplicação da Função de Peso

A evidência de que a função de peso exerce um papel essencial na diminuição do trabalho realizado pelo desenvolvedor para a localização de defeitos permitiu verificar quais classes de palavras (nomes de classes e métodos) estão mais alinhadas semanticamente com o vocabulário do usuário. Esta verificação ocorreu a partir do contraste entre o número de métodos analisados nos testes realizados com os softwares ArgoUML e JEdit. Testes estatísticos aplicados sobre a amostra de populações de métodos analisada mostraram que as descrições de defeito de cada software estudado estão alinhadas de maneira diferente com as diferentes classes de palavras citadas anteriormente.

Na busca pela combinação predominante, cada software apresentou maior afinidade com um dos conjuntos de *stop-words* estudados. Assim, a partir da análise percentual de predominância de cada conjunto de *stop-words*, verificou-se que o ArgoUML teve maior redução no número de métodos analisados através das *stop-words* formadas pelas palavras da língua Inglesa (ESTW) e o JEdit se adaptou melhor ao conjunto de palavras reservadas da linguagem Java (JSTW). Para cada parâmetro susceptível à aplicação da função de

peso (NM,FNM,CN,FCN) foram geradas duas amostras compostas pelo quantitativo de métodos a serem analisados para se corrigir 10 defeitos em cada software estudado. Este estudo é mais elaborado do que aquele realizado para obtenção da combinação predominante pois baseia-se em testes estatísticos utilizados amplamente no campo científico. Assim, em cada filtro de código fonte divide-se os resultados dos testes em dois conjuntos de amostras:

1. testes de combinações de parâmetros em que se aplica a função de peso sobre um parâmetro em estudo;
2. testes de combinações de parâmetros onde não se aplica a função de peso sobre tal parâmetro.

4.4.3 Análise do Nome dos Métodos e seus Fragmentos

A seguir são listados os testes de Wilcoxon realizados com o ArgoUML e JEdit para contrastar amostras em que se atribui peso ao nome dos métodos com aquelas em que tal atribuição não é feita. Em cada teste busca-se verificar se o número médio de métodos analisados sofre grande influência decorrente da aplicação da função de peso proposta neste trabalho sobre o nome dos métodos. Neste ponto utiliza-se o número de métodos analisados pelo desenvolvedor e não a precisão como anteriormente foi feito com filtros de código fonte. O número de métodos analisados em cada teste é uma medida mais interessante já que reflete diretamente o trabalho despendido pelo desenvolvedor.

O teste estatístico deverá comparar o número médio de métodos analisados em 72 combinações com a função de peso aplicada sobre uma determinada classe de palavra (métodos, classes e seus fragmentos) com as 72 combinações em que não se aplica a função de peso. As hipóteses formuladas para o teste de Wilcoxon foram:

- Hipótese Nula (H_0) - Não existe diferença significante entre o número médio de métodos analisados em combinações sem peso e com peso.
- Hipótese Alternativa (H_a) - O número médio de métodos analisados em combinações com peso **é menor** do que em combinações sem peso.

O JEdit se mostrou bastante sensível à aplicação da função de peso sobre os fragmentos do nome do método e parcialmente sobre o nome do método conforme os testes de Wilcoxon realizados e expressos nas Tabelas 4.9 e 4.10. Com exceção do teste de Wilcoxon listado na Tabela 4.9 com o filtro ET, todos os outros testes são unânimes em relacionar a linguagem natural utilizada pelo usuário nas descrições de defeitos com os nomes utilizados para nomear os métodos do software. Os valores de p-value abaixo de 0,05, obtidos nos testes, permitem rejeitar a hipótese nula e aceitar a hipótese alternativa. Conforme foi visto a hipótese alternativa diz que combinações que possuem a função de

Software	**Filtro de código fonte**	**Combinações com peso**	**Combinações sem peso**	**Resultado do Teste de Wilcoxon**
Jedit	ET	72	72	wilcox.test(compeso, sempeso, alternative = "less") dados: compeso e sempeso **W = 2677; p-value = 0,6337**
Jedit	QTL	72	72	wilcox.test(compeso, sempeso, alternative = "less") dados: compeso e sempeso **W = 2137; p-value = 0,03468**

Tabela 4.9: Teste de Wilcoxon com o JEdit - função de peso aplicada sobre nomes de métodos

Software	**Filtro de código fonte**	**Combinações com peso**	**Combinações sem peso**	**Resultado do Teste de Wilcoxon**
Jedit	ET	72	72	wilcox.test(compeso, sempeso, alternative = "less") dados: compeso e sempeso **W = 581; p-value = 4,747e-16**
Jedit	QTL	72	72	wilcox.test(compeso, sempeso, alternative = "less") dados: compeso e sempeso **W = 122; p-value < 2,2e-16**

Tabela 4.10: Teste de Wilcoxon com o JEdit - função de peso aplicada sobre fragmentos dos nomes de métodos

Software	**Filtro de código fonte**	**Combinações com peso**	**Combinações sem peso**	**Resultado do Teste de Wilcoxon**
ArgoUML	ET	72	72	wilcox.test(compeso, sempeso, alternative = "less") dados: compeso e sempeso **W = 2215; p-value = 0,06624**
ArgoUML	QTL	72	72	wilcox.test(compeso, sempeso, alternative = "less") dados: compeso e sempeso **W = 2232,5; p-value = 0,07572**

Tabela 4.11: Teste de Wilcoxon com o ArgoUML - função de peso aplicada sobre os nomes de métodos

peso aplicada realizam em média um trabalho menor na análise de métodos do software do que as não sofreram aplicação da função.

Os resultados apontam que a aplicação da função de peso sobre os nomes de métodos e seus fragmentos não é tão relevante para se diminuir o número de métodos analisados para o software ArgoUML conforme os resultados dos testes de Wilcoxon expressos nas Tabelas 4.11 e 4.12. Assim, é indiferente a aplicação da função de peso sobre o nome dos métodos e seus fragmentos para o ArgoUML.

Software	Filtro de código fonte	Combinações com peso	Combinações sem peso	Resultado do Teste de Wilcoxon
ArgoUML	ET	72	72	wilcox.test(compeso, sempeso, alternative = "less") dados: compeso e sempeso **W = 2898; p-value = 0,8897**
ArgoUML	QTL	72	72	wilcox.test(compeso, sempeso, alternative = "less") dados: compeso e sempeso **W = 2999,5; p-value = 0,9485**

Tabela 4.12: Teste de Wilcoxon com o ArgoUML - função de peso aplicada sobre os fragmentos dos nomes de métodos

4.4.4 Análise do Nome das Classes e seus Fragmentos

Todos os testes realizados encontraram p-value abaixo de 0,05 para o ArgoUML quando analisando a função de peso aplicada sobre nome de classes e seus fragmentos. Isto implica aceitar a hipótese alternativa em que se gostaria de verificar se o número médio de métodos analisados em combinações com peso era menor do que em combinações sem peso.

Software	Filtro de código fonte	Combinações com peso	Combinações sem peso	Resultado do Teste de Wilcoxon
JEdit	ET	72	72	wilcox.test(compeso, sempeso, alternative = "less") Wilcoxon rank sum test with continuity correction dados: compeso e sempeso **W = 2680; p-value = 0,6382**
JEdit	QTL	72	72	wilcox.test(compeso, sempeso, alternative = "less") dados: compeso e sempeso **W = 2397; p-value = 0,2185**

Tabela 4.13: Teste de Wilcoxon com o JEdit - função de peso aplicada sobre nomes de classes

Os testes de Wilcoxon também revelaram que aumentar a importância dos nomes de classes e seus fragmentos para o JEdit, através da aplicação da função de peso, não implicou a diminuição do número de métodos analisados na correção de defeitos, conforme pode ser visto nas Tabelas 4.13 e 4.14. Entretanto, tal fato não ocorreu com o ArgoUML que se mostrou extremamente sensível a aplicação da função de peso sobre os nomes de classes e seus fragmentos conforme os testes de Wilcoxon expressos nas Tabelas 4.15 e 4.16.

Software	**Filtro de código fonte**	**Combinações com peso**	**Combinações sem peso**	**Resultado do Teste de Wilcoxon**
JEdit	ET	72	72	wilcox.test(compeso, sempeso, alternative = "less") Wilcoxon rank sum test with continuity correction dados: compeso e sempeso **W = 2572; p-value = 0,4689**
JEdit	QTL	72	72	wilcox.test(compeso, sempeso, alternative = "less") dados: compeso e sempeso **W = 2617; p-value = 0,5406**

Tabela 4.14: Teste de Wilcoxon com o JEdit - função de peso aplicada sobre fragmentos dos nomes de classes

Software	**Filtro de código fonte**	**Combinações com peso**	**Combinações sem peso**	**Resultado do Teste de Wilcoxon**
ArgoUML	ET	72	72	wilcox.test(compeso, sempeso, alternative = "less") Wilcoxon rank sum test with continuity correction dados: compeso e sempeso **W = 1717,5; p-value = 0,0002394**
ArgoUML	QTL	72	72	wilcox.test(compeso, sempeso, alternative = "less") dados: compeso e sempeso **W = 1696,5; p-value = 0,0001743**

Tabela 4.15: Teste de Wilcoxon com o ArgoUML - função de peso aplicada sobre nomes de classes

Software	**Filtro de código fonte**	**Combinações com peso**	**Combinações sem peso**	**Resultado do Teste de Wilcoxon**
ArgoUML	ET	72	72	wilcox.test(compeso, sempeso, alternative = "less") Wilcoxon rank sum test with continuity correction dados: compeso e sempeso **W = 1196; p-value = 1,229e-08**
ArgoUML	QTL	72	72	wilcox.test(compeso, sempeso, alternative = "less") dados: compeso e sempeso **W = 1130; p-value = 2,614e-09**

Tabela 4.16: Teste de Wilcoxon com o ArgoUML - função de peso aplicada sobre fragmentos dos nomes de classes

4.5 Problemas e Limitações do Trabalho Proposto

As limitações do presente trabalho estão relacionadas a diversidade de parâmetros de entrada e limitações computacionais apresentadas durante a realização dos experimentos. Outras limitações referem-se a pontos não contemplados pelo *parsing* do código fonte podendo ser eliminadas em aprimoramentos do software administrativo desenvolvido.

Uma única consulta foi formulada para cada defeito. Conforme exposto em [Lukins et al. 2008], a qualidade da consulta afeta substancialmente os resultados e novos mecanismos devem ser propostos para auxiliar o usuário na formulação de consultas. Devido a grande diversidade de parâmetros estudados, este trabalho não optou por aprofundar o estudo de formulação de consultas, limitando-se a propor uma metodologia que uniformizasse tal processo. Tal metodologia permitiu delinear uma maneira sistemática de obtenção da consulta sendo passível de ser aplicada em novos experimentos científicos relacionados.

O código fonte completo do software ArgoUML não pôde ser indexado pelo conjunto de ferramentas utilizado devido à demanda de memória principal requisitada. Esta situação foi verificada somente quando todo o código fonte foi indexado. Uma vez aplicados os filtros ET e QTL que diminuem o número de documentos no corpus final, os experimentos transcorreram normalmente.

Este trabalho não contempla classes internas devido a limitações encontradas no *parsing* do código fonte. É importante salientar que de todos os defeitos analisados nenhum continha algum método a ser corrigido que pertencesse a uma classe interna. Em futuras implementações tal problema será resolvido.

Muitas das limitações aqui apresentadas são de caráter provisório sendo passíveis de correção em futuras versões do software desenvolvido. A demanda por recursos computacionais apresentada nos testes que envolviam todo o código fonte requer maior tempo e atenção. Estes testes utilizaram bibliotecas escritas em linguagem Fortran disponibilizadas dentro do pacote de software estatístico R.

4.6 Considerações Finais sobre os Resultados

Os resultados do trabalho foram dispostos conforme a evolução e alternativas para se tentar estabelecer uma combinação que fosse satisfatória à redução do trabalho do desenvolvedor. No primeiro momento foram estudados os rankings das combinações possíveis com base nos filtros de código fonte e software estudados, provendo-se visões auxiliares baseadas no escopo de métodos analisados. Sem que fosse possível visualizar um padrão de parâmetros ou uma combinação que sempre figurasse nas primeiras posições das tabelas de combinações, foi realizada nova análise buscando-se os valores de parâmetros que mais contribuíam para diminuir o número de métodos analisados no universo de combinações

possíveis para cada parâmetro. Deste estudo resultou a combinação predominante que estava bem colocada no ranking de combinações conforme demonstrado pela comparação com as melhores combinações encontradas.

Os resultados baseados em testes comparativos entre o JEdit e ArmedBear-J evidenciaram que a aplicação de filtros no código fonte, com o objetivo de diminuir o número de documentos do corpus, é benéfico para se melhorar o tempo de processamento dos testes e a precisão, com pequena perda na revocação para o terceiro defeito do ArmedBear-J. A diferenciação entre os dois filtros propostos (ET e QTL) não ficou clara quanto a ganhos em termos da precisão obtida no conjunto de testes procedidos com o ArmedBear-J e JEdit. Ficou claro pelos testes que o número de documentos analisados e o tempo de execução dos testes no filtro QTL é inferior a estes mesmos fatores para o filtro ET. Quanto à precisão não pôde ser verificada diferença pelos testes sendo realizada nova análise com base em testes estatísticos, os quais apontaram não existir diferença significativa entre as médias das precisões para os dois filtros.

Em relação à aplicação da função de peso sobre a matriz de termos-documentos, o objetivo de diminuição do trabalho na localização de defeitos foi alcançado com sua utilização como definida neste trabalho. Como consequência deste fato, foi verificado que algumas classes de palavras presentes no código fonte apresentaram um alinhamento semântico com a linguagem natural expressa nas descrições de defeitos do software. Neste ponto ressaltamos que foi feita a recuperação da rastreabilidade entre documentação do software e código fonte conforme proposto.

Todos os parâmetros foram caracterizados para compor a calibragem satisfatória a ser utilizada pelo desenvolvedor em futuras buscas. Sua comparação com as melhores combinações, presentes nos primeiros lugares do ranking de combinações, permitiu ter a dimensão de sua eficiência frente as outras combinações possíveis.

Capítulo 5

Trabalhos Relacionados

O trabalho aqui apresentado utiliza uma combinação de temáticas que estão fragmentadas em diversos trabalhos científicos. As temáticas aqui abordadas são:

- As transformações da matriz de termos-documentos e seu impacto sobre a performance e acurárica dos testes em contextos de RI.
- Abordagens estática e dinâmica de código fonte.
- A localização de defeitos utilizando-se de técnicas de RI.

Assim, propõem-se a discussão dos trabalhos relacionados em função de tais temáticas sempre comparando o que foi feito em cada trabalho com o que está implementado na proposta corrente deste trabalho.

A Figura 5.1 retrata de forma resumida cada temática, seus artefatos de entrada e resultantes, além da visão geral da tarefa de recuperação de informação realizada.

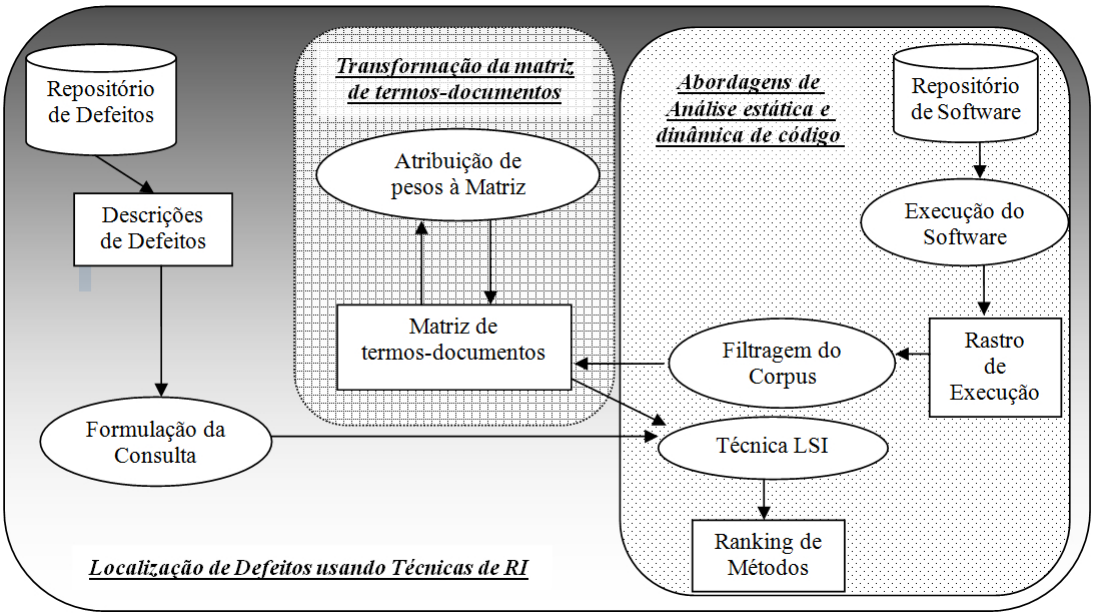


Figura 5.1: Temáticas utilizadas no presente trabalho

As próximas subseções detalham cada temática com base na análise dos trabalhos correlatos. Elas também relacionam cada temática com o que foi implementado neste trabalho.

5.1 Abordagens de Análise Estática e Dinâmica

Este trabalho propôs a filtragem do código fonte a partir da execução de versões do software extraídas do repositório. Com base nos rastros de execução filtrou-se aqueles métodos que posteriormente seriam mapeados para conceitos embutidos em consultas do usuário, isto é, consultas formuladas a partir de descrições de defeitos fornecidas por usuários do software. A filtragem e o mapeamento basearam-se em técnicas de análise dinâmica e estático, respectivamente, como ilustrado na Figura 5.1.

A tarefa de localização de conceitos ou características é definida em [Poshyvanyk et al. 2007] como a identificação de partes do código fonte que correspondem a uma funcionalidade específica do software. Em [Marcus et al. 2004] estuda-se o problema de localização de conceitos utilizando LSI e ressalta-se a importância dos usuários do software como precursores às mudanças a serem introduzidas. Enquanto os usuários finais do software entendem facilmente conceitos do domínio do problema como o pagamento de uma conta, um inventário a ser feito ou mesmo uma venda a ser realizada, eles não tem noção de onde tais conceitos estão implementados no código fonte do sistema. Os autores descartaram a possibilidade de se utilizar qualquer documentação formal do sistema e enfatizaram a participação dos usuários do sistema como agentes responsáveis pela formulação das consultas. Em seguida, eles executaram o processo de *parsing* do código fonte e a definição da granularidade dos resultados a serem obtidos. Até este momento, o trabalho presente aqui proposto assemelha-se muito ao trabalho citado sendo algumas diferenças esquematizadas na Tabela 5.1.

Os objetivos do autor, ao utilizar análise estática de código fonte, são:

- Ajudar o usuário a identificar funções e declarações que implementem conceitos.
- Analisar metodologias de formulação de consulta e seu impacto sobre os resultados.
- Comparar a técnica LSI com outras técnicas baseadas em *grep*[1] na localização de conceitos.

O autor finaliza seu artigo enfatizando que a utilização de LSI em seus experimentos foi mais eficaz que o uso de técnicas tradicionais de análise estática de código fonte como aquelas baseadas em grep ou dependências de grafos.

Em [Sobreira e Maia 2008], os autores estabelecem um método para coleta e análise de rastros de execução, provendo uma análise gráfica da rastreabilidade entre código fonte

[1]http://www.gnu.org/software/grepj

Item analisado	**Presente trabalho**	**Trabalho Comparado**
Origem dos Termos que geram as consultas	- Descrições de defeitos fornecidas por usuários.	- Solicitações de mudanças no software e conhecimento prévio do usuário sobre o código fonte ou domínio do problema. - Um único termo ou frase fornecidos pelo usuário;
Granularidade dos documentos	- A implementação de um método e seus comentários.	- Qualquer módulo contíguo de código fonte dividindo-se módulos que sejam muito extensos em unidades menores.
Define ponto de corte ou critério de parada para análise dos métodos?	Sim. Através do filtro do corpus utilizado.	Sim. Através da definição empírica de graus de similaridade entre os documentos e a consulta.
Realiza fragmentação do corpus como estratégia de análise parcial do código fonte?	Sim. Através dos filtros definidos nas abordagens ET e QTL.	Sim. Através do que o autor chama de partições, isto é, conjunto de documentos baseados na proximidade com a medida de similaridade adotada.
Abordagens utilizadas na análise do código fonte	Estática e Dinâmica	Estática
Softwares analisados	Software escrito em Java - ArgoUML, JEdit e ArmedBear-J	Software escrito em C - NSCA Mosaic (navegador web)

Tabela 5.1: Comparação de trabalho relacionado - [Marcus et al. 2004]

e características selecionadas do sistema. A ferramenta desenvolvida baseia-se na montagem de matrizes que relacionam componentes de software e características permitindo visualizar onde estas últimas estão implementadas no código fonte. A rastreabilidade entre código fonte e características representa a grande contribuição do trabalho. O desenvolvedor tem ao seu alcance a possibilidade de analisar visualmente quais partes do código fonte são afetadas pela alteração de uma característica. Isto se fez possível através da instrumentação e de uma análise dinâmica do código. Entretanto, a quantidade de informação a ser visualizada pode ser muito grande, dificultando a tarefa de localização.

Na tarefa de localização de defeitos realizada no trabalho corrente realiza-se uma aproximação semântica entre documentos. Estes documentos são os métodos do código fonte e a consulta submetida pelo desenvolvedor. Com base na tabela anterior, verifica-se que foram utilizados dois tipos de análise do código fonte: estática e dinâmica. As

limitações de cada uma das abordagens para a tarefa de localização de características são apontadas em [Poshyvanyk et al. 2007]. Conforme exposto no artigo, a abordagem dinâmica não realiza uma correta distinção entre características que utilizam métodos em comum para sua execução. Por outro lado, a análise estática raramente consegue identificar todos os métodos que concorrem para a execução de um cenário específico de execução. Por fim, conclui ser necessária a combinação das abordagens para uma efetiva localização de características dada a contribuição de dados fornecida de forma exclusiva por cada abordagem. De forma esquemática, a Tabela 5.2 resume a metodologia e características do trabalho realizado em [Poshyvanyk et al. 2007], comparando-a com o presente trabalho:

Item analisado	**Presente trabalho**	**Trabalho Comparado**
Abordagens utilizadas na análise do código fonte	Estática e Dinâmica	Estática e Dinâmica
Técnicas de Recuperação utilizadas	LSI com alteração de parâmetros de entrada	LSI clássico e SPR - scenario-based probabilistic ranking
Nível de granularidade dos resultados	Métodos	Métodos e classes
Alvo do estudo	Localização de defeitos	Localização de características e defeitos
Softwares analisados	ArgoUML, JEdit e ArmedBear-J	Mozilla e Eclipse

Tabela 5.2: Comparação de trabalho relacionado - [Poshyvanyk et al. 2007]

Ao final do trabalho tem-se definida uma nova técnica para localização de características intitulada pelos autores como PROMESIR. Trata-se da avaliação dos rankings providos pela técnica baseada em execução de cenários e eventos probabilísticos (*SPR - scenario-based probabilistic ranking*) e do LSI tradicional classificadas como técnicas dinâmica e estática, respectivamente. As duas técnicas são aplicadas em separado para responder a mesma pergunta: "Quais partes implementam as características em estudo?". Pode-se definir como contribuição do trabalho a formulação de uma metodologia para combinar os resultados obtidos nas duas técnicas, conseguindo-se diminuir o esforço do desenvolvedor que deseja encontrar uma característica implementada no código.

O presente trabalho não define uma nova técnica como realizado em [Poshyvanyk et al. 2007], mas procura entender melhor as possibilidades de parametrização da técnica LSI. É utilizada uma combinação de análises estática e dinâmica de código fonte que se mostraram importantes no processo de eliminação de métodos irrelevantes à busca e apresentam bons resultados quando conjugadas com técnicas de recuperação de Informação. A utilização da abordagem dinâmica para filtragem do código demonstrou preservar a

revocação, localizando todos os métodos referentes a correção do defeito. Somente um defeito relativo ao software ArmedBear-J apresentou revocação diferente de 100%, na amostra de defeitos analisados.

5.2 Transformações da Matriz de Termos-Documentos

Diversos trabalhos enfatizam a transformação da matriz de termos-documentos em tarefas de recuperação de informação a partir da aplicação de funções de peso [Nakov et al. 2001], [Santhiappan et al. 2006], [Joshi et al. 2007], [Dumais 1992]. Em [Santhiappan et al. 2006], defende-se que as funções de peso aumentam a eficiência do sistema de recuperação de informação à medida que variam a importância dos termos conforme a frequência da ocorrência dos mesmos no corpus. No experimento são utilizadas duas combinações de funções de peso:

1. IWF + NDV;
2. IDF + NDV;

- IWF - calcula o peso do termo baseado na frequência de ocorrência de um termo através dos documentos.
- NDV - esta função de peso normaliza o vetor de documentos baseado em sua magnitude.
- IDF - um tipo de função global que é baseada no número de documentos em que um termo específico aparece.

A conclusão dos experimentos revelaram que a combinação de funções de peso IDF+NDV se sobressai em performance quando se estuda o tamanho do corpus analisado e também produziu a mais alta precisão para um corpus de tamanho 100000. Em outra análise, coletou-se documentos de diferentes categorias e analisou-se como as funções de peso variam com respeito a mudança no número de domínios para o mesmo tamanho de corpus. A performance geral de IDF+NDV mostrou-se superior quando comparada com outras funções. Em relação a precisão, IDF+NDV mostrou uma curva em uniforme crescimento. Baseado em todos os experimentos realizados os autores concluem que a combinação de IDF e NDV apresenta performance superior em todos os testes. Quanto a apuração da acurácia, as funções de peso estudadas não mostraram expressiva variação nos resultados.

Em Jones(1972, apud [Nakov et al. 2001]) e Dumais(1991, apud [Nakov et al. 2001]) estuda-se um subconjunto clássico de funções de peso que podem ser aplicadas a matriz de termos-documentos. Em [Nakov et al. 2001], o autor busca analisar 12 funções de peso diferentes sobre um corpus formado por fragmentos de textos literários da língua inglesa. Através da exploração visual de matrizes de similaridade, onde a densidade da cor expressa

a proximidade semântica entre os fragmentos, revela-se a estrutura latente do conjunto de documentos analisados objetivando-se em última análise a clusterização de documentos. Enquanto projeta visualmente os fragmentos de texto o autor testa várias combinações de funções de peso. Concluiu-se que a aplicação de um peso local logarítmico se mostrou interessante e o uso de entropia global também é ressaltada como melhor função de peso.

Em um abordagem comparativa ao que foi proposto por [Nakov et al. 2001], o presente trabalho utiliza uma função de peso específica definida para aumentar a importância dos nomes dos métodos, classes e seus fragmentos. Além disso, a expressão da frequência dos termos em escala logarítmica também é realizada para normalizar valores muito altos na matriz de termos-documentos. A atenuação dos valores assim definida melhorou a precisão dos testes.

5.3 Localização de Defeitos a partir de Técnicas de Recuperação de Informação

Em [Lukins et al. 2008], estuda-se a técnica LDA *(Latent Dirichlet Allocation)* como uma nova alternativa à tarefa de localização de defeitos. Ele argumenta que LDA prove melhores resultados em termos de performance que LSI porque o esforço para se encontrar o primeiro método a ser corrigido é sistematicamente menor usando LDA do que LSI. De fato, os resultados fornecidos neste trabalho poderiam também ser usados como uma boa alternativa à tarefa de localização de defeitos e ser testado usando LDA. O trabalho aqui apresentado forneceu uma maneira sistemática de coletar os termos da consulta a partir de uma descrição de defeito. A qualidade da consulta tem uma forte influência sobre a qualidade dos resultados, conforme atestado em [Lukins et al. 2008], da mesma forma que consultas ruins têm similar influência em um contexto de busca na web.

Outros estudos também sugerem o uso de análise dinâmica para aumentar a produtividade na recuperação de rastreabilidade [Antoniol et al. 2000], [Lucia et al. 2009]. Em contraste, o trabalho aqui proposto utiliza-se de rastros de execução de programas para traçar estratégias que visem diminuir o tamanho do corpus e aumentar a precisão sem perda na revocação.

5.4 Outros parâmetros estudados - *Stop-words* e redução de dimensionalidade k

O presente trabalho também estudou a variação no fator k que representa a redução da dimensionalidade do espaço vetorial semântico constituído a partir de documentos do código fonte. Alguns trabalhos estudam a melhor caracterização do valor de k para um conjunto de documentos específicos [Kontostathis 2007], [Nakov et al. 2001]. Na Figura

5.1, o processo de redução da dimensionalidade do espaço semântico está embutido na *Técnica LSI* empregada, onde o algoritmo SVD utilizado pela técnica necessita do valor de k como parâmetro como entrada.

A escolha do valor de k é um fator decisivo para a performance do processo de busca a ser realizado. A escolha de um valor muito pequeno para k acarreta a perda de importantes distinções entre os termos da coleção [Dumais 1996]. Em contrapartida, pondera-se que ao manter um valor de k muito alto, o nível de ruído introduzido é aumentado pela variabilidade na escolha dos termos. A técnica LSI requer que este parâmetro deva ser ajustado para cada coleção de textos a ser analisada. Com base no escopo de valores de k sugerido pela literatura [Santhiappan e Gopalan 2010], o presente trabalho utilizou-se de valores discretos de k (100,200 e 300) na tentativa de elicitar qual valor deve ser utilizado como padrão pelo desenvolvedor em seu trabalho diário de correção de defeitos. Percebeu-se, através da experimentação exaustiva de combinações de parâmetros, que o valor predominante de k para os software ArgoUML e JEdit é 300.

Em coleções de textos maiores, a escolha de um valor elevado para k torna restritiva a abordagem em relação ao tempo demandado para indexar os documentos e processar a consulta além de requerer muito consumo de memória [Kontostathis 2007]. Isto se confirma com base em testes realizados com o ArgoUML no presente trabalho. Ao se considerar todo o código fonte, os referidos testes tornaram-se impeditivos quanto a quantidade de recursos computacionais demandados. Em [Kontostathis 2007] é proposta uma versão estendida da técnica LSI que pode capturar o relacionamento entre os termos utilizando-se de poucas dimensões do espaço vetorial semântico. Sua abordagem contribui para a redução do uso de memória e processador quando comparado a modalidade tradicional da técnica LSI.

Por fim, ao se comparar a utilização de diferentes conjuntos de *stop-words*, delineou-se para cada software, deve ser escolhido um conjunto que melhor se enquadra para redução do esforço do desenvolvedor. A escolha do melhor conjunto de *stop-words* a ser aplicado a cada software foi determinada de maneira experimental, com base na análise percentual de combinações possíveis para o parâmetro *stop-word*. Este procedimento foi realizado na obtenção da combinação predominante.

Capítulo 6

Conclusões

Neste trabalho foi apresentado um estudo da influência dos parâmetros de Análise por Semântica Latente sobre a precisão e performance em tarefas de localização de defeitos. Os parâmetros alvos deste estudo foram filtros de código fonte, valor do parâmetro de redução de dimensionalidade k, função de peso aplicada sobre classes de palavras do código fonte e variabilidade do conjunto de *stop-words*. Os resultados dos parâmetros estudados permitem afirmar que houve uma sensível variação na quantidade de métodos analisados pelo desenvolvedor durante uma tarefa de localização de defeitos no código fonte em função da combinação variada dos parâmetros de entrada da Técnica LSI. Não foi possível encontrar uma melhor combinação de parâmetros para todos os filtros e softwares estudados. Uma calibragem satisfatória para os parâmetros de entrada da técnica LSI foi identificada através da combinação predominante e seus valores são: (k=300, MN=SIM, FMN=SIM, CN=SIM, FCN=SIM).

Foi mostrado, a partir do uso do conjunto de combinações dos parâmetros da técnica LSI, escolhidos como alvo deste estudo, que é possível elicitar uma combinação satisfatória para coleções específicas de texto formadas por fragmentos do código fonte de um software. Esta combinação foi denominada neste trabalho como predominante pois foi obtida a partir da análise de predominância dos valores de parâmetros que implicaram um menor trabalho na busca por métodos que precisavam ser corrigidos. Esta combinação é sugerida para ser utilizada na localização de defeitos especificamente para os softwares estudados neste trabalho. Para novos softwares baseados na linguagem de programação Java, novos testes poderiam ser realizados, utilizando a metodologia proposta e novos valores para a combinação predominante poderiam surgir como resultado de tais testes. Após a execução dos testes e criação do ranking de combinações possíveis, verificou-se também que tal combinação predominante não foi sempre a melhor para os softwares em todos os filtros e escopo de métodos analisados, pois como já foi apontado não houve uma combinação melhor em todos os casos. Entretanto, com base na comparação das melhores combinações obtidas pode-se verificar que a combinação predominante é eficiente em relação ao esforço de manutenção realizado pelo desenvolvedor. Ela representa um valor *default* na busca

pela calibragem ideal que diminua os custos envolvidos na manutenção de software.

As abordagens que norteiam a filtragem do corpus também revelam pontos conclusivos a partir dos resultados dos experimentos. Um importante resultado deste trabalho é que os filtros utilizados afetaram minimamente a revocação dos testes da amostra de defeitos analisada. Somente um dos defeitos do software ArmedBear-J teve revocação diferente de 100%. Testes comparativos realizados com o JEdit e ArmedBear-J demonstraram que os filtros exercem grande influência sobre a performance e precisão obtidas com a aplicação da técnica LSI. Desta forma, os dois filtros propostos, ET e QTL, demonstraram impacto positivo sobre o processo de recuperação de informação usando LSI. Elas são independentes da técnica de indexação utilizada, LSI neste estudo, podendo ser usadas em conjunto com outras técnicas para posteriores estudos comparativos.

Com base em testes estatísticos não foi comprovada diferença entre os dois filtros, ET e QTL, em relação a precisão média dos resultados. Contudo, uma análise descritiva demonstrou uma ligeira vantagem do filtro QTL em relação ao filtro ET. Este fato se deve a maior eliminação de métodos, irrelevantes a correção do defeito, realizada pelo filtro QTL.

A utilização de funções de pesos específicas, conforme propostas neste trabalho, teve o objetivo de aumentar a importância de determinadas classes de palavras: nomes de métodos, classes e seus fragmentos. Foi mostrado que para o JEdit a função de peso aplicada ao nome dos métodos implicou em uma opção decisiva no processo de localização de defeitos. Por outro lado, a aplicação de tal função mostrou-se indiferente quando se considera o nome das classes para o mesmo software. Em relação ao software ArgoUML, o nome das classes exerceram um importante papel no tocante à melhoria do ranking de métodos relevantes à correção do defeito.

Como consequência do resultado anterior foi realizada uma análise estatística complementar para se comprovar a aproximação entre os vocabulários utilizados em cada software com o vocabulário utilizado pelo usuário (ver Seção 4.4). Uma vez que as descrições de defeitos expressam a linguagem natural utilizada pelos usuários, pode-se concluir que esta linguagem tinha maior correspondência com as palavras utilizadas para nomear os métodos do JEdit e as classes do ArgoUML. Este fato é de grande relevância pois um maior alinhamento entre todas as classes de palavras presentes no código fonte com o vocabulário do usuário pode aumentar a compreensão do sistema, diminuindo os custos relativos a manutenção do mesmo. Decisões de projeto adotadas ao nomear elementos de projeto e as especificidades do domínio do problema em que os softwares foram concebidos podem subsidiar um processo de reengenharia de software para melhoria do vocabulário utilizado. No entanto, tais assertivas dependem de um estudo aprofundado e podem ser alvo de futura exploração científica.

Um desenvolvedor interessado em utilizar Recuperação de Informação para localização de defeitos precisará formular consultas a serem submetidas ao mecanismo de busca

desenvolvido. Neste ponto, o estudo realizado foi de grande relevância pois fornece ao desenvolvedor uma metodologia para geração de consultas com base em palavras chaves encontradas nas descrições de defeitos de software. A metodologia para geração de consultas foi definida na Seção 3.2.

Além da consulta, um processo de localização de defeitos baseado em RI utiliza-se de parâmetros de entrada que devem ser fornecidos pelo desenvolvedor no momento de realização da busca. Assim, a metodologia proposta neste trabalho para geração da combinação de parâmetros de entrada da técnica LSI pode ser empregada com eficiência para se gerar aproximações da combinação ideal em outros softwares escritos na linguagem de programação Java. A combinação predominante pode então atenuar o esforço do desenvolvedor em um trabalho contínuo de localização de defeitos.

Como resultado adicional, a análise estatística proposta na Seção 4.4, permite ao desenvolvedor verificar se o vocabulário utilizado no código fonte do software está condizente com o vocabulário do usuário que é expresso em linguagem natural através das descrições dos defeitos. Esta nova possibilidade subsidia a tomada de decisão quanto a refatoração do software de forma que o mesmo se torne mais compreensível, diminuindo consequentemente os custos de manutenção.

Novos trabalhos podem ser derivados com base na análise individual dos parâmetros estudados bem como utilização de novas técnicas de Recuperação de Informação como LDA. Outra vertente de trabalhos derivados, refere-se a avaliação da eficiência da função de peso proposta neste trabalho na diminuição do número de métodos analisados por combinação.

Referências Bibliográficas

[Anderson et al. 2009] Anderson, D. R., Sweeney, D. J., Willians, T. A., e Willians, T. A. (2009). *Statistics for business and economics*. Cengage Learning.

[Andrian Marcus 2003] Andrian Marcus, J. I. M. (2003). Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing. In *25th International Conference on Software Engineering (ICSE)*, número 3, pp. 125–137, Portland, Oregon, USA. IEEE Computer Society.

[Antoniol et al. 2000] Antoniol, G., Canfora, G., Casazza, G., e Lucia, A. D. (2000). Information Retrieval Models for Recovering Traceability Links between Code and Documentation. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, p. 40, Washington, DC, USA. IEEE Computer Society.

[Antoniol et al. 2002] Antoniol, G., Canfora, G., Casazza, G., Lucia, A. D., e Merlo, E. (2002). Recovering Traceability Links between Code and Documentation. *IEEE Transactions on Software Engeering*, 28:970–983.

[Antoniol e Gueheneuc 2006] Antoniol, G. e Gueheneuc, Y.-G. (2006). Feature Identification: An Epidemiological Metaphor. *IEEE Trans. Softw. Eng.*, 32:627–641.

[Arun et al. 2009] Arun, R., Suresh, V., e Madhavan, C. E. V. (2009). Stopword Graphs and Authorship Attribution in Text Corpora. In *Proceedings of the International Conference on Semantic Computing*, pp. 192–196, Washington, DC, USA. IEEE Computer Society.

[Boehm e Basili 2001] Boehm, B. e Basili, V. R. (2001). Software Defect Reduction Top 10 List. *Computer*, 34:135–137.

[Breese et al. 1998] Breese, J. S., Heckerman, D., e Kadie, C. (1998). Empirical Analysis of Predictive Algorithms for Collaborative Filtering. Technical report, Microsoft Corporation, Redmond, WA 98052.

[Czarnecki e Eisenecker 2000] Czarnecki, K. e Eisenecker, U. W. (2000). *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

[Deerwester 1988] Deerwester, S. (1988). Improving Information Retrieval with Latent Semantic Indexing. In *Proc. of the 51st Annual Meeting of the American Society for Information Science*, pp. 36–40.

[Deerwester et al. 1988] Deerwester, S., Dumais, S., Landauer, T., Furnas, G., e Harshman, R. (1988). Using latent semantic analysis to improve access to textual information.

In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '88, pp. 281–285, Washington, D.C., United States, New York, NY, USA. ACM.

[Deerwester et al. 1990] Deerwester, S., Dumais, S., Landauer, T., Furnas, G., e Harshman, R. (1990). Indexing by latent semantic analysis. *Journal of the American Society for Information Science (JASIS)*, 41:391–407.

[Dumais 1992] Dumais, S. T. (1992). Enhancing Performance in Latent Semantic Indexing (LSI) Retrieval. Technical report, Bellcore Laboratories.

[Dumais 1996] Dumais, S. T. (1996). Information Retrieval: Finding Needles in Massive Haystacks. In *Massive Data Sets*, pp. 23–32, Washington, DC, USA. National Academy Press.

[Edelstein e Mamone 1993] Edelstein, D. V. e Mamone, C. (1993). The IEEE standard for software maintenance. Technical report, IEEE Computer Society, New York, NY, USA.

[Egyed e Grunbacher 2002] Egyed, A. e Grunbacher, P. (2002). Automating Requirements Traceability: Beyond the Record & Replay Paradigm. In *Proceedings of the 17th IEEE international conference on Automated software engineering*, ASE '02, pp. 163–, Washington, DC, USA. IEEE Computer Society.

[Egyed et al. 2009] Egyed, A., Grunbacher, P., Heindl, M., e Biffl, S. (2009). Value-Based Requirements Traceability: Lessons Learned. In *Design Requirements Workshop, LNBIP 14*, pp. 240–257. Springer Verlag. Berlin Heidelberg.

[Erdil et al. 2003] Erdil, K., Finn, E., Keating, K., Meattle, J., Park, S., e Yoon, D. (2003). Software Maintenance As Part of the Software Life Cycle. Technical report, Department of Computer Science Tufts University, Medford, MA, USA.

[Erwig e Ren 2007] Erwig, M. e Ren, D. (2007). An Update Calculus for Expressing Type-safe Program Updates. *Science of Computer Programming*, 67:199–222.

[Grossman e Frieder 2004] Grossman, D. A. e Frieder, O. (2004). *Information retrieval: algorithms and heuristics*. Springer.

[Grubb e Takang 2003] Grubb, P. e Takang, A. A. (2003). *Software Maintenance - Concepts and Practice*. World Scientific Publishing Co. Pte. Ltd., Singapore.

[Hassan 2008] Hassan, A. E. (2008). The road ahead for Mining Software Repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pp. 48–57. IEEE.

[Huang et al. 2003] Huang, J. C., Chang, C. K., e Christensen, M. (2003). Event-based traceability for managing evolutionary change. *IEEE Transactions on Software Engineering*, 29(9):796–810.

[Joshi et al. 2007] Joshi, H., Zhang, C., Ramaswamy, S., e Bayrak, C. (2007). Local and Global Recency Weighting Approach to Bug Prediction. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, p. 33, Washington, DC, USA. IEEE Computer Society.

[Kontostathis 2007] Kontostathis, A. (2007). Essential Dimensions of Latent Semantic Indexing (LSI). In *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, HICSS 07, p. 73, Washington, DC, USA. IEEE Computer Society.

[Kuhn et al. 2007] Kuhn, A., Ducasse, S., e Girba, T. (2007). Semantic clustering: Identifying topics in source code. *Journal on Information Systems and Technologies*, 49:230–243.

[Landauer 2007] Landauer, T. K. (2007). *Handbook of latent semantic analysis*. Lawrence Erlbaum Associates, Mahwah, New Jersey.

[Letsche e Berry 1997] Letsche, T. A. e Berry, M. W. (1997). Large-Scale Information Retrieval with Latent Semantic Indexing. *Informatics and Computer Science - Information Sciences*, 100:105–137.

[Li et al. 2010] Li, J., Stalhane, T., Kristiansen, J. M. W., e Conradi, R. (2010). Empirical Studies on Software Corrective Maintenance Activities and Productivities. In *Proceedings of the International Conference on Software Maintenance*, ICSM 2010, Timisoara, Romania.

[Lucia et al. 2007] Lucia, A. D., Fasano, F., Oliveto, R., e Tortora, G. (2007). Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology*, 16(13).

[Lucia et al. 2009] Lucia, A. D., Oliveto, R., e Tortora, G. (2009). Assessing IR-based traceability recovery tools through controlled experiments. *Empirical Software Engg.*, 14:57–92.

[Lukins et al. 2008] Lukins, S. K., Kraft, N. A., e Etzkorn, L. H. (2008). Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pp. 155–164, Washington, DC, USA. IEEE Computer Society.

[Manning et al. 2008] Manning, C. D., Raghavan, P., e Schutze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.

[Marcus e Maletic 2001] Marcus, A. e Maletic, J. (2001). Identification of High-Level Concept Clones in Source Code. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pp. 107–114, Washington, DC, USA. IEEE Computer Society.

[Marcus et al. 2004] Marcus, A., Sergeyev, A., Rajlich, V., e Maletic, J. I. (2004). An Information Retrieval Approach to Concept Location in Source Code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pp. 214–223, Washington, DC, USA. IEEE Computer Society.

[Nakov et al. 2001] Nakov, P., Popova, A., e Mateev, P. (2001). Weight functions impact on LSA performance. In *In Proceedings of the Recent Advances in Natural language processing*, pp. 187–193, Bulgaria. Proceedings of the Recent Advances in Natural language processing.

[Nelson 1996] Nelson, M. L. (1996). A Survey of Reverse Engineering and Program Comprehension. In *ODU CS 551 Software Engineering Survey*, p. 2.

[Pashov et al. 2004] Pashov, I., Riebisch, M., e Philippow, I. (2004). Supporting Architectural Restructuring by Analyzing Feature Models. In *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, pp. 25–36, Washington, DC, USA. IEEE Computer Society.

[Peng e Wallace 1995] Peng, W. W. e Wallace, D. R. (1995). *Software error analysis*. Silicon Press, Summit, NJ USA.

[Poshyvanyk et al. 2007] Poshyvanyk, D., Gueheneuc, Y.-G., Marcus, A., Antoniol, G., e Rajlich, V. (2007). Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *IEEE Transactions on Software Engineering*, 33:420–432.

[Ramamoorthy e Tsai 1996] Ramamoorthy, C. e Tsai, W.-T. (1996). Advances in software engineering. *Computer*, 29(10):47–58.

[Ramesh 1998] Ramesh, B. (1998). Factors influencing requirements traceability practice. *Commun. ACM*, 41:37–44.

[Riebisch 2004] Riebisch, M. (2004). Supporting Evolutionary Development by Feature Models and Traceability Links. In *Proceedings 11th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pp. 370–377, Los Alamitos, CA, USA. IEEE Computer Society.

[Salkind 2010] Salkind, N. J. (2010). *Encyclopedia of Research Design*. SAGE Publications, California, USA.

[Salton e Buckley 1987] Salton, G. e Buckley, C. (1987). Term Weighting Approaches in Automatic Text Retrieval. Technical report, Ithaca, NY, USA.

[Salton et al. 1975] Salton, G., Wong, A., e Yang, C. S. (1975). A vector space model for automatic indexing. *Commun, ACM*, 18:613–620.

[Santhiappan e Gopalan 2010] Santhiappan e Gopalan, V. P. (2010). Finding the Optimal Rank for LSI Models. In *International Conference on Advances and Emerging Trends in Computing Technologies*. SRM-School of Computing.

[Santhiappan et al. 2006] Santhiappan, S., Gopalan, V. P., e Kumar, S. (2006). Role of Weighting on TDM in Improvising Performance of LSA on Text Data. In *IEEE INDICON 2006*, pp. 1–6.

[Shukla e Misra 2008] Shukla, R. e Misra, A. K. (2008). Estimating software maintenance effort: a neural network approach. In *Proceedings of the 1st India software engineering conference*, ISEC '08, pp. 107–112, Hyderabad, India, New York, NY, USA. ACM.

[Sobreira e Maia 2008] Sobreira, V. e Maia, M. (2008). A Visual Trace Analysis Tool for Understanding Feature Scattering. pp. 337–338. 15th Working Conference on Reverse Engineering - WCRE 08.

[van der Spek et al. 2008] van der Spek, P., Klusener, S., e van de Laar, P. (2008). Towards Recovering Architectural Concepts Using Latent Semantic Indexing. In *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, pp. 253–257, Washington, DC, USA. IEEE Computer Society.

[Wild e Stahl 2007] Wild, F. e Stahl, C. (2007). Investigating unstructured text using LSA. In *Advances in Data Analysis*, pp. 383–390. Springer Berlin Heidelberg.

[Wild et al. 2005] Wild, F., Stahl, C., Stermsek, G., Penya, Y., e Neumann, G. (2005). Factors Influencing Effectiveness in Automated Essay Scoring with LSA. In *Proceeding of the 2005 conference on Artificial Intelligence in Education: Supporting Learning through Intelligent and Socially Informed Technology*, pp. 947–949, Amsterdam, The Netherlands, The Netherlands. IOS Press.

[Williams e Hollingsworth 2005] Williams, C. C. e Hollingsworth, J. K. (2005). Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480.

[Yihong e Wei 2007] Yihong, G. e Wei, X. (2007). *Machine Learning for Multimedia content Analysis*. Springer Science+Business Media, LLC.

[Zeller 2009] Zeller, A. (2009). *Why programs fail: a guide to systematic debugging*. Elsevier.

Apêndice A

Aplicação da metodologia para formulação da consulta

A metodologia desenvolvida neste trabalho para formulação da consulta foi aplicada a 23 descrições de defeitos dos softwares ArgoUML, JEdit e ArmedBear-J. Cada defeito é detalhado abaixo e a consulta final é mostrada como resultado da aplicação da metodologia.

---- ---- --- --- ---- --- --- ---- -- ---- ---- --- --- ---- ------ ---- -- ---- ---- --

Bugs do ArgoUML

BUG 1	Ao deletar a última estrutura “figpartition” em um diagrama de atividades fica oculta uma estrutura “figpool”		
Código fonte:	argo_uml_(rev_15155)_(0_25_5)		
Descrição curta	Deleting last Swimlane leaves hidden FigPool		
Descrição Detalhada	When you delete the last FigPartition (a swimlane) from an Activity diagram, then a (hidden) FigPool remains present in the project. You can proof its presence by lookinhg at the PGML file in the zargo, or (much easier) save & reload, after which the FigPool becomes visible.		
Filtro 1	Deleting last Swimlane leaves hidden FigPool When you delete the last FigPartition (a swimlane) from an Activity diagram, then a (hidden) FigPool remains present in the project. ~~You can proof its presence by lookinhg at the PGML file in the zargo, or (much easier) save & reload, after which the FigPool becomes visible.~~		
Filtro 2			
Onde acontece o erro?	activity diagram		
Qual a ação que desencadeia o erro?	delete		
Quais as partes do software envolvidas?	Swimlane FigPool FigPartition swimlane activity diagram FigPool project		
Roteiro de Execução	- Inicialização - Preparar cenário, cria duas estruturas figpartition e apaga todas		
Momento da captura dos rastros relevantes	Inicializei o ArgoUml. **Inicio a captura da Característica.** Crio um diagrama de Atividades. Insiro duas estruturas figpartition no mesmo. Apago a primeira e por último apago a segunda(a última)		
FILTRO3	activity diagram delete swimlane figPool fig pool figPartition fig partition project		
ITEM OBSERVADO	**TODO O CÓDIGO FONTE**	**ET**	**QTL**
NÚMERO DE DOCUMENTOS NO CORPUS	16309	1216	453
TEMPO MÉDIO POR TESTE (SEGUNDOS)	--	59,19	12,4
CONSUMO DE MEMÓRIA OBSERVADO PARA O R COM K=300	--	=~479,3MBytes	=~257MBytes

BUG 2	Permite selecionar e deletar dois elementos sendo um editável e outro somente leitura quando selecionados simultaneamente.		
Código fonte:	argo_uml_0_26_alpha_3		
Descrição curta	Delete should not be in context menu if any **item readonly**		
Descrição Detalhada	Select a profile element and a user element in the explorer. Right click on the user element shows Delete as enabled in the context menu. ~~Pressing delete results in the illegal deletion of the model element from the profile.~~		
Filtro 2			
Onde acontece o erro?	explorer		
Qual a ação que desencadeia o erro?	click		
Quais as partes do software envolvidas?	delete context menu item readonly profile element user element explorer user element delete context menu		
Roteiro de Execução	- Inicialização. - Preparar cenário, Seleciono um item somente leitura e outro não e deleto.		
Momento da captura dos rastros relevantes	Inicializo o ArgoUml. **Inicio a captura da Característica.** Expando a árvore de itens somente leitura e a árvore de itens editáveis pelo usuário. Seleciono um item em cada árvore e clico com o botão direito. Seleciono deletar.		
Filtro 3 (Consulta)	explorer click delete context menu item read only profile element user		
ITEM OBSERVADO	**TODO O CÓDIGO FONTE**	**ET**	**QTL**
NÚMERO DE DOCUMENTOS NO CORPUS	14983	868	393
TEMPO MÉDIO POR TESTE (SEGUNDOS)	--	30,01	12,52
CONSUMO DE MEMÓRIA OBSERVADO PARA O R COM K=300	--	=~364,3MBytes	=~242MBytes

BUG 3	Permite mover um item da parte profile para a parte de user na árvore do explorer		
Código fonte:	argo_uml_0_26_alpha_3		
Descrição curta	DnD to users model part of tree should not work from profile part		
Descrição Detalhada	It should not be possible to drop an element dragged from the profile part of the tree to the user model part of the tree. The only allowable drag-drop from the profile part should be to the diagram.		
Filtro 1	It should not be possible to drop an element dragged from the profile part of the tree to the user model part of the tree. The only allowable drag-drop from the profile part should be to the diagram.		
Filtro 2			
Onde acontece o erro?	tree		
Qual a ação que desencadeia o erro?	drag drop dnd		
Quais as partes do software envolvidas?	users model part tree profile part element profile part tree user model part tree profile part diagram		
Roteiro de Execução	- Inicialização - Preparar cenário, seleciono um item somente leitura e movo para a parte de usuário.		
Momento da captura dos rastros relevantes	Inicializei o ArgoUml. **Inicio a captura da Característica.** Expando a árvore de itens somente leitura e a árvore de itens editáveis pelo usuário. Clico em um item somente leitura e traslado para a árvore de itens editáveis pelo usuário.		
Filtro 3 (Consulta)	tree drag drop dnd user model part profile element diagram		
ITEM OBSERVADO	**TODO O CÓDIGO FONTE**	**ET**	**QTL**
NÚMERO DE DOCUMENTOS NO CORPUS	14983	722	483
TEMPO MÉDIO POR TESTE (SEGUNDOS)	--	22,26	11,94
CONSUMO DE MEMÓRIA OBSERVADO PARA O R COM K=300	--	=~315MBytes	=~256MBytes

BUG 4	Ao clicar com o botão direito sobre uma classe pela primeira vez mostra (show full path) ao invés de (hide full path).
Código fonte:	argo_uml_rev_15144_0_25_5
Descrição curta	Hide/Show path in name setting problems
Descrição Detalhada	I found two problems with this setting. a) When you add a class to a class diagram outside the class package, it shows the full path in the class name. That's ok. But, when you right-click on it, the "show" submenu shows "Show Path in Name" instead of the "hide" option since it's already shown. So, you need to press the "show" and then right-click again and select the "hide" option. b) When you save the project, close it and then open it, this setting gets lost. I think ArgoUML should keep that setting saved for each diagram in a separate manner, because you could want to show the path in one diagram but not in another.
Filtro 1	Hide/Show path in name setting problems I found two problems with this setting. a) When you add a class to a class diagram outside the class package, it shows the full path in the class name. That's ok. But, when you right-click on it, the "show" submenu shows "Show Path in Name" instead of the "hide" option since it's already shown. So, you need to press the "show" and then right-click again and select the "hide" option. ~~b) When you save the project, close it and then open it, this setting gets lost. I think ArgoUML should keep that setting saved for each diagram in a separate manner, because you could want to show the path in one diagram but not in another.~~
Filtro 2	
Onde acontece o erro?	class diagram
Qual a ação que desencadeia o erro?	hide show
Quais as partes do software envolvidas?	class name class package class full path submenu Show Path Name option hide option
Roteiro de Execução	- Inicialização - - Preparar cenário, criar classe com nome e acionar o item do botão direito hide show
Momento da captura dos rastros relevantes	Inicializo o ArgoUml. **Inicio a captura da Característica.** Crio um classe e dou nome a ela. Depois clico com o botão direito sobre a mesma e aciono a opção do menu “Show name in path” e depois “Hide name in Path”.
Filtro 3 (Consulta)	class diagram hide show name package full path submenu sub menu option

ITEM OBSERVADO	**TODO O CÓDIGO FONTE**	**ET**	**QTL**
NÚMERO DE DOCUMENTOS NO CORPUS	16162	1032	532
TEMPO MÉDIO POR TESTE (SEGUNDOS)	--	43,15	14,71
CONSUMO DE MEMÓRIA OBSERVADO PARA O R COM K=300	--	=~434,2MBytes	=~283MBytes

BUG 5	A elipse cresce desproporcionalmente quando digitando um nome em um diagrama de caso de uso.		
Código fonte:	argo_uml_(rev_15089)_(0_26_alpha_1)		
Descrição curta	Use case fig height grows when typing name		
Descrição Detalhada	When drawing a use-case, and typing the name in the properties panel, every character typed causes the height of the oval to increase substantially.		
Filtro 1	Use case fig height grows when typing name When drawing a use-case, and typing the name in the properties panel, every character typed causes the height of the oval to increase substantially.		
Filtro 2			
Onde acontece o erro?	use case		
Qual a ação que desencadeia o erro?	type		
Quais as partes do software envolvidas?	use case fig name use case name properties panel character oval		
Roteiro de Execução	- Inicialização - Preparar cenário, criar caso de uso e digitar nome		
Momento da captura dos rastros relevantes	Inicializei o ArgoUml. **Inicio a captura da Característica.** Expando os diagramas existentes e clico no diagrama de casos de uso. Adiciono um caso de uso em forma de elipse e clico no campo nome. Começo a digitar o nome.		
Filtro 3 (Consulta)	use case type fig name property panel character oval		
ITEM OBSERVADO	**TODO O CÓDIGO FONTE**	**ET**	**QTL**
NÚMERO DE DOCUMENTOS NO CORPUS	16139	996	486
TEMPO MÉDIO POR TESTE (SEGUNDOS)	--	39,1	12,79
CONSUMO DE MEMÓRIA OBSERVADO PARA O R COM K=300	--	=~408MBytes	=~273,2MBytes

BUG 6	A visibilidade do painel de propriedades tem comportamentos distintos para multi-seleção de elementos
Código fonte:	argo_uml_(rev_15168)_0_26_alpha_2
Descrição curta	Property panel should not be available on multi-target selection
Descrição Detalhada	This first occurs in 0.25.4 and seems to be a side effect of keeping the model element properties tab enabled. To reproduce place two classes on a diagram Click on one and then ctrl-click the other. The properties tab is enabled and visible but blank, this is good. Click once on the diagram background and repeat the step above. This time the property panel is visible. I prefer that the tab stay enabled with no visible controls.
Filtro 1	Property panel should not be available on multi-target selection ~~This first occurs in 0.25.4 and seems to be a side effect of keeping the model element properties tab enabled.~~ To reproduce place two classes on a diagram Click on one and then ctrl-click the other. The properties tab is enabled and visible but blank, this is good. Click once on the diagram background and repeat the step above. This time the property panel is visible. ~~I prefer that the tab stay enabled with no visiblen controls.~~
Filtro 2	
Onde acontece o erro?	property panel
Qual a ação que desencadeia o erro?	click
Quais as partes do software envolvidas?	property panel multi target selection classes diagram properties tab diagram background property panel
Roteiro de Execução	- Inicialização - Preparar cenário, criar duas classes e seleciona-las duas vezes.
Momento da captura dos rastros relevantes	Inicializo o ArgoUml. **Inicio a captura da Característica.** Expando a árvore do explorer e clico sobre "Diagrama de Classes". Clico sobre o ícone para criação de uma nova classe por duas vezes para criar duas classes. Clico sobre uma e seleciono ctrl no teclado e clico sobre a outra. Clico no fundo da tela. Clico novamente sobre uma classe e seleciono ctrl no teclado e clico sobre a outra.
Filtro 3 (Consulta)	property panel click multi target selection class diagram property tab background back ground

ITEM OBSERVADO	TODO O CÓDIGO FONTE	ET	QTL
NÚMERO DE DOCUMENTOS NO CORPUS	16178	948	553
TEMPO MÉDIO POR TESTE (SEGUNDOS)	--	36	14,22
CONSUMO DE MEMÓRIA OBSERVADO PARA O R COM K=300	--	=~387MBytes	=~274MBytes

BUG 7	Novos atributos criados no padrão a:Foo não podem ser deletados pois não estão no namespace
Código fonte:	argo_uml_(rev_15233)_(0_26_alpha_2)
Descrição curta	Attribute types newly created by Notation subsystem not place in any namespace
Descrição Detalhada	Because of the Fix for issue 5211, a class without namespace can be created: 1. in a new model, add a new class to the class diagram and name it 'A' 2. add a new attribute by double clicking on the attribute compartement ==> newAttr : Integer is created 3. double click on the attribute so you can edit it in the diagram 4. append ';a:Foo' to it and hit ENTER ==> A now has 2 attributes ==> attribute a has type Foo (class), which cannot be found in the model Save and load works, but there is no way to select or delete Foo.
Filtro 1	Attribute types newly created by Notation subsystem not place in any namespace ~~Because of the Fix for issue 5211, a class without namespace can be created:~~ 1. in a new model, add a new class to the class diagram and name it ~~'A'~~ 2. add a new attribute by double clicking on the attribute compartement ~~==> newAttr : Integer is created~~ 3. double click on the attribute so you can edit it in the diagram 4. append ~~';a:Foo'~~ to it and hit ENTER ==> A now has 2 attributes ==> attribute a has type ~~Foo~~ (class), which cannot be found in the model ~~Save and load works, but there is no way to select or delete Foo.~~
Filtro 2	
Onde acontece o erro?	class attribute compartment
Qual a ação que desencadeia o erro?	hit
Quais as partes do software envolvidas?	attribute types notation subsystem namespace model class diagram attribute attribute compartement attribute diagram attributes attribute type class model
Roteiro de Execução	- Inicialização - Preparar cenário, adicionar uma classe e dois atributos.
Momento da captura dos rastros relevantes	Inicializo o ArgoUml. **Inicio a captura da Característica.** Crio uma classe e um atributo do tipo inteiro clicando duas vezes sobre o compartimento. Edito o atributo e adiciono ;a:Foo a descrição já existente e teclo ENTER.
Filtro 3 (Consulta)	class attribute compartment hit type notation subsystem sub system namespace name space model diagram

ITEM OBSERVADO	**TODO O CÓDIGO FONTE**	**ET**	**QTL**
NÚMERO DE DOCUMENTOS NO CORPUS	16185	1201	870
TEMPO MÉDIO POR TESTE (SEGUNDOS)	--	42	30,42
CONSUMO DE MEMÓRIA OBSERVADO PARA O R COM K=300	--	=~476MBytes	=~361MBytes

BUG 8	Função Zomm não funciona		
Código fonte:	argo_uml_(rev15688)_(0_26_1_beta_1)		
Descrição curta	Zoom In/Out broken		
Descrição Detalhada	Neither the menu selections nor the keyboard shortcuts for Zoom In and Zoom Out work. If you change the zoom using the slider, the Zoom Reset menu option does work correctly.		
Filtro 1	Zoom In/Out broken Neither the menu selections nor the keyboard shortcuts for Zoom In and Zoom Out work. If you change the zoom using the slider, the Zoom Reset menu option does work correctly.		
Filtro 2			
Onde acontece o erro?	---		
Qual a ação que desencadeia o erro?	zoom		
Quais as partes do software envolvidas?	menu selections keyboard shortcuts zoom in zoom out zoom slider zoom reset menu option		
Roteiro de Execução	- Inicialização - Preparar cenário, utilizar zoom in ou out keyboard e menu.		
Momento da captura dos rastros relevantes	Inicio o ArgoUml. **Inicio a captura da Característica.** Insiro uma classe e clico sobre a mesma. Uso o ctrl+- para tentar dar zoom. Uso também o zoom do menu para tentar aumentar ou diminuir o tamanha da classe.		
Filtro 3 (Consulta)	zoom menu selection keyboard key board shortcut short cut in out slider reset option		
ITEM OBSERVADO	**TODO O CÓDIGO FONTE**	**ET**	**QTL**
NÚMERO DE DOCUMENTOS NO CORPUS	16423	868	601
TEMPO MÉDIO POR TESTE (SEGUNDOS)	--	32,69	16 segundos
CONSUMO DE MEMÓRIA OBSERVADO PARA O R COM K=300	--	=~370MBytes	=~295,5MBytes

BUG 9	Ao criar um atributo de classe ocorre um erro com as barras de rolagem no campo valor inicial (ocultam o texto digitado).
Código fonte:	argo_uml_(rev_15749)_(0_22)
Descrição curta	attribute's "initial value" is scrolling wrong
Descrição Detalhada	Steps to reproduce: - click on a class attribute (or create a new project, create a class and add an int attribute) - in the property panel, click on the "initial value" body field (the one on the left) - add a couple of tabs and maybe some text The field scrolls, but the scrolling bar hides the text (at least on my machine)! See also the attachment. Note that the language field on the right scrolls smoothly. Moreover: the text vertical alignment in the language field is not pursuant to the the body field.
Filtro 1	attribute's "initial value" is scrolling wrong Steps to reproduce: - click on a class attribute (or create a new project, create a class and add an int attribute) - in the property panel, click on the "initial value" body field (the one on the left) - add a couple of tabs and maybe some text The field scrolls, but the scrolling bar hides the text (~~at least on my machine~~)! ~~See also the attachment. Note that the language field on the right scrolls smoothly.~~ ~~Moreover: the text vertical alignment in the language field is not pursuant to the the body field~~.

Termos recuperados das descrições	
Onde acontece o erro?	class attribute initial value body field
Qual a ação que desencadeia o erro?	add
Quais as partes do software envolvidas?	attribute's initial value class attribute project class attribute property panel initial value body field text field scrolling bar text
Roteiro de Execução	- Inicialização - Preparar cenário,criar classe e atributo de classe digitando um valor inicial
Momento da captura dos rastros relevantes	Inicio o ArgoUml. **Inicio a captura da Característica.** Adiciono uma classe ao diagrama. Clico duas vezes sobre o compartimento de atributos adicionando um atributo. Com o atributo selecionado clico no campo valor inicial e adiciono um texto e algumas tabulações de modo a acionar a barra de texto.
Filtro 3 (Consulta)	class attribute initial value body field add project property panel scrolling bar text

ITEM OBSERVADO	**TODO O CÓDIGO FONTE**	**ET**	**QTL**
NÚMERO DE DOCUMENTOS NO CORPUS	16414	908	566
TEMPO MÉDIO POR TESTE (SEGUNDOS)	--	34,22	15,7
CONSUMO DE MEMÓRIA OBSERVADO PARA O R COM K=300	--	=~376MBytes	=~290MBytes

BUG 10	Ao criar uma classe antes de soltar o mouse a figura da classe aparece deformada sem os compartimentos.
Código fonte:	argo_uml_(rev_16517)_(0_28_alpha_2)
Descrição curta	Figs shown incorrectly initially
Descrição Detalhada	When drawing a class on the diagram with the diagram tool, the figure initially shown (when the mouse button is down) is an incomplete figure, e.g. it does not show compartments. Releasing the mouse button corrects the effect.
Filtro 1	Figs shown incorrectly initially When drawing a class on the diagram with the diagram tool, the figure initially shown (when the mouse button is down) is an incomplete figure, e.g. it does not show compartments. ~~Releasing the mouse button corrects the effect.~~
Termos recuperados das descrições	
Onde acontece o erro?	class diagram
Qual a ação que desencadeia o erro?	draw
Quais as partes do software envolvidas?	Figs class diagram diagram tool figure figure compartments
Roteiro de Execução	- Inicialização - Preparar cenário, criar classe.
Momento da captura dos rastros relevantes	Inicio o ArgoUml. **Inicio a captura das características.** Clico no ícone na barra de ferramentas para criar uma nova classe e clico sobre o diagrama finalizando a criação da classe.
Filtro 3 (Consulta)	class diagram draw fig class tool figure compartment

ITEM OBSERVADO	TODO O CÓDIGO FONTE	ET	QTL
NÚMERO DE DOCUMENTOS NO CORPUS	16498	773	331
TEMPO MÉDIO POR TESTE (SEGUNDOS)	--	26,51	10,28
CONSUMO DE MEMÓRIA OBSERVADO PARA O R COM K=300	--	=~334,9	=~236MBytes

---

Bugs do JEdit

BUG 11	Ao localizar o caractere $ e mandar substituir por uma letra fica faltando substituir uma ocorrência.		
Código fonte:	jedit_(rev_13899)_(4_3_pre_16)		
Descrição curta	regex '$' does not replace an empty last line		
Descrição Detalhada	Make a new buffer with 15 empty lines. With regex switched on and hypersearch switched on, search for $ This will match all 15 lines. Search for $ again and replace with z. This will put z on all but the last line. So line 15 is still empty, even though it matched before.		
Filtro 1	regex ~~'$'~~ does not replace an empty last line Make a new buffer with 15 empty lines. With regex switched on and hypersearch switched on, search for $ This will match all 15 lines. Search for $ again and replace with z. This will put z on all but the last line. So line 15 is still empty, even though it matched before.		
Filtro 2			
Onde acontece o erro?	buffer		
Qual a ação que desencadeia o erro?	replace		
Quais as partes do software envolvidas?	regex line buffer lines regex hypersearch lines line line		
Roteiro de Execução	- Inicialização - Preparar cenário, procurar por $ e substituir por z.		
Momento da falha fornecido na descrição do bug e de captura dos rastros relevantes	Inicializei o Jedit. **Inicio a captura da Característica** Gero 15 linhas vazias. Apertei control+f. Digitei $ e z nos respectivos campos. Clico em Substituir todos e vejo que a última linha fica sem ser substituída.		
Filtro 3 (Consulta)	buffer replace regex line hypersearch hyper search		
ITEM OBSERVADO	**TODO O CÓDIGO FONTE**	**ET**	**QTL**
NÚMERO DE DOCUMENTOS NO CORPUS	7071	1164	387
TEMPO MÉDIO POR TESTE (SEGUNDOS)	--	52,76	9,32 segundos
CONSUMO DE MEMÓRIA OBSERVADO PARA O R COM K=300	--	=~457MBytes	=~247,8MBytes

BUG 12	Erro com upper case na procura de arquivos .cpp .h .H em todos os buffers abertos.		
Código fonte:	jedit_(rev_15833)_(4_3_pre_17)		
Descrição curta	Filename filter is case sensitive on Windows		
Descrição Detalhada	Using 'search in: all buffers' with the filter {*.cpp,*.h,*.hpp} does not match *.H if the extension is in upper case. I didn't check 'search in: directory' but assume this could have the same issue. Probably the filter match should be case insensitive on OSs like Windows which have case insensitive filenames, i.e. *.h or *.H (in filter) should match *.h or *.H (either in open buffers or on disk).		
Filtro 1	Filename filter is case sensitive ~~on Windows~~ Using 'search in: all buffers' with the filter ~~{*.cpp,*.h,*.hpp}~~ does not match ~~*.H~~ if the extension is in upper case. ~~I didn't check 'search in: directory' but assume this could have the same issue.~~ ~~Probably the filter match should be case insensitive on OSs like Windows which have case insensitive filenames, i.e. *.h or *.H (in filter) should match *.h or *.H (either in open buffers or on disk).~~		
Termos recuperados das descrições			
Onde acontece o erro?	buffers		
Qual a ação que desencadeia o erro?	search		
Quais as partes do software envolvidas?	filename filter search in all buffers filter filter filter		
Roteiro de Execução	- Inicialização - Preparar cenário, procurar por arquivos h cpp hpp e H nos buffers abertos		
Momento da falha fornecido na descrição do bug e de captura dos rastros relevantes	Inicializo o Jedit. **Inicio a captura da Feature.** Abro alguns arquivos com as extensões listadas acima. Clico no menu e no item Search e abro a caixa de diálogo. Seto as extensões no campo filtro e marco buscar em todos os buffers. Clico em “Find”.		
Filtro 3 (Consulta)	buffer search filename file name filter		
ITEM OBSERVADO	**TODO O CÓDIGO FONTE**	**ET**	**QTL**
NÚMERO DE DOCUMENTOS NO CORPUS	7070	1544	748
TEMPO MÉDIO POR TESTE (SEGUNDOS)	--	85,52	24,6 segundos
CONSUMO DE MEMÓRIA OBSERVADO PARA O R COM K=300	--	=~580MBytes	=~331MBytes

BUG 13	A barra de ação(action bar) não está acessível no modo tela cheia(full screen).
Código fonte:	jedit_(rev_15716)_(4_3_pre_17)
Descrição curta	action bar not visible in full screen mode
Descrição Detalhada	version : jEdit 4.3pre17 OS/JDK : java 5 & java 6, multiple OS steps to reproduce : - Utilities > Action Bar the action bar is shown at the bottom of the text area - close the action bar - View > Toggle full screen mode - Utilities > Action Bar the action bar is not shown
Filtro 1	action bar not visible in full screen mode ~~version : jEdit 4.3pre17~~ ~~OS/JDK : java 5 & java 6, multiple OS~~ steps to reproduce : - Utilities > Action Bar the action bar is shown at the bottom of the text area - close the action bar - View > Toggle full screen mode - Utilities > Action Bar the action bar is not shown
Filtro 2	
Onde acontece o erro?	screen
Qual a ação que desencadeia o erro?	toggle
Quais as partes do software envolvidas?	action bar utilities Action Bar action bar bottom of text area action bar view toggle full screen mode utilities action bar action bar
Roteiro de Execução	- Inicialização - Preparar cenário, acionar Action Bar no modo fullscreen.
Momento da falha fornecido na descrição do bug e de captura dos rastros relevantes	Inicializo o Jedit. **Inicializo a captura da Característica** Mudo para o modo fullscreen. Aciono o menu Utilities – Action Bar.
Filtro 3 (Consulta)	screen toggle action bar utility bottom text area view full mode

ITEM OBSERVADO	**TODO O CÓDIGO FONTE**	**ET**	**QTL**
NÚMERO DE DOCUMENTOS NO CORPUS	7073	923	376
TEMPO MÉDIO POR TESTE (SEGUNDOS)	--	33,93	10,47
CONSUMO DE MEMÓRIA OBSERVADO PARA O R COM K=300	--	=~380MBytes	=~250MBytes

BUG 14	Ações to-lowercase e to-uppercase não funciona onde o cursor está com foco.
Código fonte:	jedit_(rev_6684)_(4_3_pre_7)
Descrição curta	**Actions "to-lower" and "to-upper" don't work without selecti - ID: 1538720**
Descrição Detalhada	Actions "to-lower" and "to-upper" just ping if nothing is selected, instead it should operate on the word the caret is in currently, while taking the edit- mode into account for determining what the word is. Sorry if this is a dupe, but I have many bugs to post and am too lazy to check them all for dupes currently. :-) OS: Windows XP Java Version: Sun Java 1.5.0_06-b05 jEdit Version: SVN Revision 6684
Filtro 1	**Actions "to-lower" and "to-upper" don't work without selecti~~— ID: 1538720~~** Actions "to-lower" and "to-upper" just ping if nothing is selected, instead it should operate on the word the caret is in currently, while taking the edit- mode into account for determining what the word is. ~~Sorry if this is a dupe, but I have many bugs to post~~ ~~and am too lazy to check them all for dupes~~ ~~currently. :)~~ ~~OS: Windows XP~~ ~~Java Version: Sun Java 1.5.0_06-b05~~ ~~jEdit Version: SVN Revision 6684~~

Filtro 2			
Onde acontece o erro?	----		
Qual a ação que desencadeia o erro?	----		
Quais as partes do software envolvidas?	to-lower action to-upper action selection to-lower action to-upper action word caret edit mode word		
Roteiro de Execução	- Inicialização - Preparar cenário, converter para lower e upper case.		
Momento da falha fornecido na descrição do bug e de captura dos rastros relevantes	Inicio o Jedit. **Inicio a captura da Característica.** Digito algum texto em maiusculo e aciono os itens de menu toLowerCase e toUpperCase com o cursor na posição sobre o texto digitado.		
Filtro 3 (Consulta)	lower action upper selection word caret edit mode		
ITEM OBSERVADO	**TODO O CÓDIGO FONTE**	**ET**	**QTL**
NÚMERO DE DOCUMENTOS NO CORPUS	6396	797	378
TEMPO MÉDIO POR TESTE (SEGUNDOS)	--	27,16	9,25
CONSUMO DE MEMÓRIA OBSERVADO PARA O R COM K=300	--	=~339MBytes	=~245MBytes

BUG 15	A barra horizontal não acompanha o movimento das teclas home e end permanecendo estática em linhas longas.
Código fonte:	jedit_(rev_8281)_(4_3_pre_9)
Descrição curta	**Horizontal scroll bar on text area does not follow caret - ID: 1534016**
Descrição Detalhada	The horizontal scroll bar at the bottom of the edit pane does not always follow the caret. This is easily reproducible: 1. create a long line, longer than the current visible size of the edit pane so that the horizontal scroll bar appears. 2. create a few more short lines that are completely visible in the edit pane. 3. use the arrow keys, move to the start of the long line. See that the horizontal scroll bar is positioned to the left (as it should be). 4. press the "End" key to move to the end of the long line. At this point, the caret is correctly positioned at the end of the line and the end of the line is visible. However, the horizontal scroll bar is still positioned to the extreme left as it was in #3. It should be moved to the right.
Filtro 1	Horizontal scroll bar on text area does not follow caret ~~- ID: 1534016~~ The horizontal scroll bar at the bottom of the edit pane does not always follow the caret. This is easily reproducible: 1. create a long line, longer than the current visible size of the edit pane so that the horizontal scroll bar appears. 2. create a few more short lines that are completely visible in the edit pane. 3. use the arrow keys, move to the start of the long line. See that the horizontal scroll bar is positioned to the left (as it should be). 4. press the "End" key to move to the end of the long line. At this point, the caret is correctly positioned at the end of the line and the end of the line is visible. However, the horizontal scroll bar is still positioned to the extreme left as it was in #3. It should be moved to the right.

Termos recuperados das descrições	
Onde acontece o erro?	text area
Qual a ação que desencadeia o erro?	press
Quais as partes do software envolvidas?	horizontal scroll bar text area caret horizontal scroll bar bottom of edit pane caret line edit pane horizontal scroll bar lines line horizontal scroll bar line caret line line horizontal scroll bar
Roteiro de Execução	- Inicialização - Preparar cenário, Gerar texto e posicionar o cursor no fim da linha longa
Momento da falha fornecido na descrição do bug e de captura dos rastros relevantes	Inicializo o Jedit. **Inicio a captura da Característica.** Crio uma linha longa que faz a barra horizontal aparecer. Retorno a barra horizontal para a esquerda e uso a tecla END para ir ao fim da linha longa. A barra horizontal não acompanha o último movimento e permanece na esquerda.
FILTRO 3 (Consulta)	text area press horizontal scroll bar caret bottom edit pane line

ITEM OBSERVADO	**TODO O CÓDIGO FONTE**	**ET**	**QTL**
NÚMERO DE DOCUMENTOS NO CORPUS	6494	898	491
TEMPO MÉDIO POR TESTE (SEGUNDOS)	--	33,26	13,68 segundos
CONSUMO DE MEMÓRIA OBSERVADO PARA O R COM K=300	--	371,1	=~274,8MBytes

BUG 16	Comportamento incorreto do indicador de buffer alterado que não funciona em todas vezes que um buffer é alterado.
Código fonte:	jedit_(rev_13506)_(4_3_pre_16)
Descrição curta	DIRTY_CHANGED is not sent after emptying Untitled buffer - ID: 2089713
Descrição Detalhada	With r13466, sometimes BufferTabs shows wrong icon for Untitled buffer which is dirty. This seems to happen since r13446, or related changes. Reproduction recipe: 1.Open a new buffer. 2.Type 'a'. Now the buffer becomes dirty and DIRTY_CHANGED is sent over EditBus. 3.Type BS. Now the buffer becomes not dirty and DIRTY_CHANGED is sent. 4.Type 'a'. Now the buffer becomes dirty but DIRTY_CHANGED is not sent. Because of this, the icon on BufferTabs remain showing "new", but when I close this buffer, the prompt comes up. Then buffer.isDirty() gives true.
Filtro 1	DIRTY_CHANGED is not sent after emptying Untitled buffer ~~- ID: 2089713 With r13466, sometimes BufferTabs shows wrong icon for Untitled buffer which is dirty. This seems to happen since r13446, or related changes.~~ Reproduction recipe: 1.Open a new buffer. 2.Type 'a'. Now the buffer becomes dirty and DIRTY_CHANGED is sent over EditBus. 3.Type BS. Now the buffer becomes not dirty and DIRTY_CHANGED is sent. 4.Type 'a'. Now the buffer becomes dirty but DIRTY_CHANGED is not sent. Because of this, the icon on BufferTabs remain showing "new", but when I close this buffer, the prompt comes up. Then buffer.isDirty() gives true.
Filtro 2	
Onde acontece o erro?	Untitled buffer
Qual a ação que desencadeia o erro?	type
Quais as partes do software envolvidas?	DIRTY_CHANGED untitled buffer buffer buffer DIRTY_CHANGED EditBus buffer DIRTY_CHANGED buffer DIRTY_CHANGED BufferTabs buffer prompt buffer.isDirty
Roteiro de Execução	- Inicialização - Preparar cenário, Digitar texto apagar e digitar novamente.
Momento da falha fornecido na descrição do bug e de captura dos rastros relevantes	Inicializo o Jedit. **Inicio a captura da Característica.** Digito um caractere e em seguida apago usando o backspace. Digito novamente e observo que o ícone que informa buffer alterado está falho.
Filtro 3 (Consulta)	DIRTY_CHANGED dirty changed untitled buffer editBus edit bus bufferTabs tab prompt isDirty

ITEM OBSERVADO	TODO O CÓDIGO FONTE	ET	QTL
NÚMERO DE DOCUMENTOS NO CORPUS	7012	817	812
TEMPO MÉDIO POR TESTE (SEGUNDOS)	--	28,19	25,3 segundos
CONSUMO DE MEMÓRIA OBSERVADO PARA O R COM K=300	--	340MBytes	=~335MBytes

BUG 17	Erro na cor e distorção ao selecionar um texto com soft wrap e foreground selection ativados
Código fonte:	jedit_(rev_17226)_(4_4_pre_1)
Descrição curta	Selection foreground color broken when soft wrap - ID: 2946203
Descrição Detalhada	When the soft wrap is used and the selection foreground color in options is set, the selected lines which are wrapped appear duplicated in foreground color, but still remain underneath in the original color, and the text in both colors is mixed (see attached screenshot). The selection foreground works fine when the soft wrap is turned off. Windows XP jedit 4.4pre1 latest daily build 2010-02-04_12-01-04 Activity log entries that might be useful in a bug report: Log file created on Thu Feb 04 22:42:38 CET 2010 IMPORTANT: Because updating this file after every log message would kill performance, it will be *incomplete* unless you invoke the Utilities->Troubleshooting->Update Activity Log on Disk command! 22:42:38 [main] [message] Log: When reporting bugs, please include the following information: 22:42:38 [main] [message] Log: java.version=1.6.0_18 22:42:38 [main] [message] Log: java.vm.version=16.0-b13 22:42:38 [main] [message] Log: java.runtime.version=1.6.0_18-b07
Filtro 1	Selection foreground color broken when soft wrap ~~- ID: 2946203~~ When the soft wrap is used and the selection foreground color in options is set, the selected lines which are wrapped appear duplicated in foreground color, but still remain underneath in the original color, and the text in both colors is mixed (see attached screenshot). The selection foreground works fine when the soft wrap is turned off. ~~Windows XP~~ ~~jedit 4.4pre1 latest daily build 2010-02-04_12-01-04~~ ~~Activity log entries that might be useful in a bug report:~~ ~~Log file created on Thu Feb 04 22:42:38 CET 2010~~ ~~IMPORTANT:~~ ~~Because updating this file after every log message would kill~~ ~~performance, it will be *incomplete* unless you invoke the~~ ~~Utilities->Troubleshooting->Update Activity Log on Disk command!~~ ~~22:42:38 [main] [message] Log: When reporting bugs, please include the following information:~~ ~~22:42:38 [main] [message] Log: java.version=1.6.0_18~~ ~~22:42:38 [main] [message] Log: java.vm.version=16.0-b13~~ ~~22:42:38 [main] [message] Log: java.runtime.version=1.6.0_18-b07~~

Filtro 2	
Onde acontece o erro?	text foreground
Qual a ação que desencadeia o erro?	set
Quais as partes do software envolvidas?	selection foreground color soft wrap soft wrap selection foreground color options lines foreground color original color text selection foreground soft wrap
Roteiro de Execução	- Inicialização - Preparar cenário, Colar texto selecionando-o e selecionar wrap soft e selection foreground.
Momento da falha fornecido na descrição do bug e de captura dos rastros relevantes	Inicio o Jedit. **Inicio a captura da Característica.** Colo um texto de várias linhas no buffer e seleciono o mesmo. Acesso Utilites->Buffer Options e marco a opção “Soft Wrap”. Acesso novamente Utilites->Global Options e marco a opção “Selection Foreground” e observo o defeito relatado acima.
Filtro 3 (Consulta)	selection foreground fore ground color soft wrap option line original text

ITEM OBSERVADO	**TODO O CÓDIGO FONTE**	**ET**	**QTL**
NÚMERO DE DOCUMENTOS NO CORPUS	6979	1303	535
TEMPO MÉDIO POR TESTE (SEGUNDOS)	--	66,54	16,5 segundos
CONSUMO DE MEMÓRIA OBSERVADO PARA O R COM K=300	--	520,3	=~286MBytes

BUG 18	Ícone é incorretamente associado a item da barra de ferramentas ao ser alterado em global_options->toolbar
Código fonte:	jedit_(rev_14664)_(4_3_pre_17)
Descrição curta	Edit Tool Bar Entry doesn't remember selected built-in icon - ID: 2045733
Descrição Detalhada	Steps to reproduce: 1. Open Global Options / jEdit / Tool Bar. 2. Edit any button that has a built-in icon (like the default buttons). 3. Optionally, click OK. Observed behavior: The dialog box displays the first built-in icon in the list. It is also associated with the button if you click OK. Excpected behavior: The previously selected icon is displayed in the drop-down box, and if nothing is changed before pressing OK, the button stays the same. jEdit version: 4.3pre15 (jedit.org Slackware package) Java version: 1.6.0_06
Filtro 1	Edit Tool Bar Entry doesn't remember selected built-in icon ~~- ID: 2045733~~ Steps to reproduce: 1. Open Global Options / jEdit / Tool Bar. 2. Edit any button that has a built-in icon (like the default buttons). 3. Optionally, click OK. Observed behavior: The dialog box displays the first built-in icon in the list. It is also associated with the button if you click OK. Excpected behavior: The previously selected icon is displayed in the drop-down box, and if nothing is changed before pressing OK, the button stays the same. ~~jEdit version: 4.3pre15 (jedit.org Slackware package)~~ ~~Java version: 1.6.0_06~~
Filtro 2	
Onde acontece o erro?	global options jedit tool bar
Qual a ação que desencadeia o erro?	click
Quais as partes do software envolvidas?	Tool Bar icon global options Tool Bar button icon buttons dialog box icon list button icon drop-down box button
Roteiro de Execução	- Inicialização - Preparar cenário, Editar ícone a partir de global options
Momento da falha fornecido na descrição do bug e de captura dos rastros relevantes	Inicio o Jedit. **Inicio a captura da Característica.** Clico em Utilities->Global Options->Toolbar. Seleciono um ícone padrão e peço para editar. Ao abrir a caixa de diálogo clico em ok sem alterar nada. O programa troca o ícone devido a associação incorreta inicial.
Filtro 3 (Consulta)	global option jedit tool bar click icon button dialog box list drop down

ITEM OBSERVADO	**TODO O CÓDIGO FONTE**	**ET**	**QTL**
NÚMERO DE DOCUMENTOS NO CORPUS	7123	1020	1014
TEMPO MÉDIO POR TESTE (SEGUNDOS)	--	40,77	38,9 segundos
CONSUMO DE MEMÓRIA OBSERVADO PARA O R COM K=300	--	=~406MBytes	393,6MBytes

BUG 19	Procura por texto, sublinhando ao final da consulta a linha e o resultado, enquanto somente este último deveria ser sublinhado.		
Código fonte:	jedit_(rev_15504)_(4_3_pre_17)		
Descrição curta	**Line numbers in Hypersearch Results are wrongly highlighted**		
Descrição Detalhada	Reproduction recipe: Put "1" for an empty buffer, and do hypersearch for "1". The result becomes "1: 1" and both "1" for line number and for the text is highlighted. The line number shouldn't be highlighted. I can reproduce this with jEdit trunk r14991, JRE 6u13, Windows XP.		
Filtro 1	Line numbers in Hypersearch Results are wrongly highlighted Reproduction recipe: Put ~~"1"~~ for an empty buffer, and do hypersearch for ~~"1".~~ The result becomes ~~"1: 1"~~ and both ~~"1"~~ for line number and for the text is highlighted. The line number shouldn't be highlighted. ~~I can reproduce this with jEdit trunk r14991, JRE 6u13, Windows XP.~~		
Filtro 2			
Onde acontece o erro?	buffer		
Qual a ação que desencadeia o erro?	search		
Quais as partes do software envolvidas?	line numbers hypersearch results buffer line number text line number		
Roteiro de Execução	- Inicialização - Preparar cenário, inserir texto e realizar busca pelo mesmo.		
Momento da falha fornecido na descrição do bug e de captura dos rastros relevantes	Inicio o Jedit. **Inicio a captura da Característica** Digito algum texto e aciono ctrl+f para procurar. Com hypersearch ativado digito o texto a ser procurado. Peço para localizar e o resultado é exibido com o defeito mencionado.		
Filtro 3 (Consulta)	line number hypersearch hyper search result buffer text		
ITEM OBSERVADO	**TODO O CÓDIGO FONTE**	**ET**	**QTL**
NÚMERO DE DOCUMENTOS NO CORPUS	7150	1068	468
TEMPO MÉDIO POR TESTE (SEGUNDOS)	--		13,17
CONSUMO DE MEMÓRIA OBSERVADO PARA O R COM K=300	--	423MBytes	=~264MBytes

BUG 20	Ao criar um novo buffer e salvá-lo não é feita a ordenação alfabética no combobox de buffers já abertos.
Código fonte:	jedit_(rev_16406)_(4_3_pre_18)
Descrição curta	**Saving buffer with different name breaks sorting - ID: 2881152**
Descrição Detalhada	The summary doesn't let me explain more, so let's go. If you save a buffer with a different name (easy case: create a new buffer with CTRL-N then save it), the sorting in the buffer switcher (and also in BufferTabs) is broken. Example: I have these buffers opened: A B D T I create a new buffer, now I have: A B D T Untitled-1 I save the new buffer as "C", the list now shows: A B D T C I'd expect it to show: A B C D T Calling "propertiesChanged()" in jEdit.updatePosition() fixes the problem at least in the buffer switcher (not in BufferTabs, IIRC). But it doesn't sound right to do that, since that's not the purpose of that method. I tried to play with creating a new EditBus message sent when the buffer list is changed, but I got lost in all the buffer set code and haven't had time to grok it yet.
Filtro 1	The summary doesn't let me explain more, so let's go. If you save a buffer with a different name (~~easy case: create a new buffer with CTRL-N then save it~~), the sorting in the buffer switcher (and also in BufferTabs) is broken. ~~Example:~~ ~~I have these buffers opened:~~ ~~A B D T~~ ~~I create a new buffer, now I have:~~ ~~A B D T Untitled-1~~ ~~I save the new buffer as "C", the list now shows:~~ ~~A B D T C~~ ~~I'd expect it to show:~~ ~~A B C D T~~ ~~Calling "propertiesChanged()" in jEdit.updatePosition() fixes the problem at least in the buffer switcher (not in BufferTabs, IIRC). But it doesn't sound right to do that, since that's not the purpose of that method.~~ ~~I tried to play with creating a new EditBus message sent when the buffer list is changed, but I got lost in all the buffer set code and haven't had time to grok it yet.~~

Filtro 2			
Onde acontece o erro?	buffer		
Qual a ação que desencadeia o erro?	save		
Quais as partes do software envolvidas?	buffer name the sorting buffer switcher BufferTabs		
Roteiro de Execução	- Inicialização - Preparar cenário, abrir arquivos e gerar novo e salvar novo.		
Momento da falha fornecido na descrição do bug e de captura dos rastros relevantes	Inicio o Jedit. **Inicio a captura da Característica.** Abro os arquivos A.txt,B.txt,D.txt e T.txt. Crio um novo arquivo C.txt e salvo. A ordem dos arquivos é perdida.		
Filtro 3 (Consulta)	buffer save name sorting switcher bufferTabs tab		
ITEM OBSERVADO	**TODO O CÓDIGO FONTE**	**ET**	**QTL**
NÚMERO DE DOCUMENTOS NO CORPUS	7099	1420	637
TEMPO MÉDIO POR TESTE (SEGUNDOS)	--	81,90	19,7
CONSUMO DE MEMÓRIA OBSERVADO PARA O R COM K=300	--	=~569,5MBytes	=~308MBytes

---

Bugs do ArmedBear/J

BUG 21	Erro salvando sessão uma vez que já exista alguma aberta.
Código fonte:	j_rev_8874
Descrição curta	can't save a new session with a new name ~~- ID: 1175869~~
Descrição Detalhada	Often I have several session's I'm using throughout the editor uptime. At the moment, once you have saved or opened a session, you aren't prompted to enter the session name when you save again; so if I'm starting a new session, I have to close all, exit J then come in again. Fix: file>Close All clears the current session name as well as closing all files add setSessionName(null); to Editor#closeAll() line 2271
Filtro 1	can't save a new session with a new name ~~- ID: 1175869~~ Often I have several session's I'm using throughout the editor uptime. At the moment, once you have saved or opened a session, you aren't prompted to enter the session name when you save again; ~~so if I'm starting a new session, I have to close all, exit J then come in again.~~ ~~Fix: file>Close All clears the current session name as well as closing all files~~ ~~add~~ setSessionName(null); to Editor#closeAll() line 2271
Filtro 2	
Onde acontece o erro?	editor
Qual a ação que desencadeia o erro?	save
Quais as partes do software envolvidas?	session sessions editor session session name setSessionName Editor closeAll
Roteiro de Execução	- Inicialização - Preparar cenário e salvar sessão
Momento da falha fornecido na descrição do bug e de captura dos rastros relevantes	Inicio o J. **Inicio a captura da Característica.** Abro uma sessão existente. Realizo alguma operação como criar um novo arquivo. Tendo salvar uma nova sessão.
Filtro 3 (Consulta)	session editor name setSessionName set closeAll close

BUG 22	Identação java não funciona corretamenta dando um shift muito acentuado à direita
Código fonte:	j_rev_392
Descrição curta	Java Mode indentation bug ~~– ID: 648046~~
Descrição Detalhada	J is not indenting properly in cases such as this: public class Blarg { protected void method(Object source, Object[] path, int[] childIndices) { // Guaranteed to return a non-null array int i = 0; } } It instead indents as such: public class Blarg { protected void method(Object source, Object[] path, int[] childIndices) { // Guaranteed to return a non-null array int i = 0; } } Anything after the opening brace of the method gets shifted heavily to the right.
Filtro 1	Java Mode indentation bug ~~– ID: 648046~~ J is not indenting properly in cases such as this: ~~public class Blarg {~~ ~~protected void method(Object source, Object[] path,~~ ~~int[] childIndices) {~~ ~~// Guaranteed to return a non-null array~~ ~~int i = 0;~~ ~~}~~ ~~}~~ It instead indents as such: ~~public class Blarg {~~ ~~protected void method(Object source, Object[] path,~~ ~~int[] childIndices) {~~ ~~// Guaranteed to return a~~ ~~non-null array~~ ~~int i = 0;~~ ~~}~~ ~~}~~ Anything after the opening brace of the method gets shifted heavily to the right.
Filtro 2	
Onde acontece o erro?	---
Qual a ação que desencadeia o erro?	indent
Quais as partes do software envolvidas?	java mode indentation opening brace method
Roteiro de Execução	- Inicialização - Preparar cenário e salvar sessão
Momento da falha fornecido na descrição do bug e de captura dos rastros relevantes	Inicio o J. **Inicio a captura da Característica.** Crio um buffer novo e salvo com a extensão .java. Colo o trecho sugerido para formar uma nova classe. Seleciono o trecho colado e mando indentar.
Filtro 3 (Consulta)	indent opening brace method

BUG 23	Procura e substituição de string na caixa de dialogo não funciona.
Código fonte:	j_rev_2630
Descrição curta	Replace in Files : bad wildcard - ID: 761918
Descrição Detalhada	In the "Replace in Files" dialog box, I can't set the 'Files' textinput to a simple string like "Root". When I click to the OK button, I've got the following error : Unsupported wild card pattern If I set "Root*", it's ok. But my correct expression is realy "Root", because I want to change expression in the files named Root. version : 0.20.0
Filtro 1	Replace in Files : bad wildcard ~~- ID: 761918~~ In the "Replace in Files" dialog box, ~~I can't set the 'Files' textinput to a simple string like "Root".~~ When I click to the OK button, I've got the following error : Unsupported wild card pattern ~~If I set "Root*", it's ok.~~ ~~But my correct expression is realy "Root", because I want to change expression in the files named Root.~~ ~~version : 0.20.0~~
Filtro 2	
Onde acontece o erro?	Replace in Files dialog box
Qual a ação que desencadeia o erro?	click
Quais as partes do software envolvidas?	Bad wildcard Replace in Files dialog Box ok button unsupported wild card pattern
Roteiro de Execução	- Inicialização - Preparar cenário e tentar localizar e substituir string.
Momento da falha fornecido na descrição do bug e de captura dos rastros relevantes	Inicio o J. **Inicio a captura da Caracteristica.** Acesso o menu e a opção “Find e Replace”. Preencho o campo “Files” como sugerido. Realizo as tentativas de substituição mencionadas acima.
Filtro 3 (Consulta)	bad wildcard replace file dialog box ok button unsupported wild card pattern

Apêndice B

Telas do sistema administrativo

Abaixo são listadas algumas telas do sistema administrativo desenvolvido para suportar as atividades de análise de código fonte para localização de defeitos no código fonte conforme discutido no Capítulo 3.

Figura B.1: Cadastro de código fonte

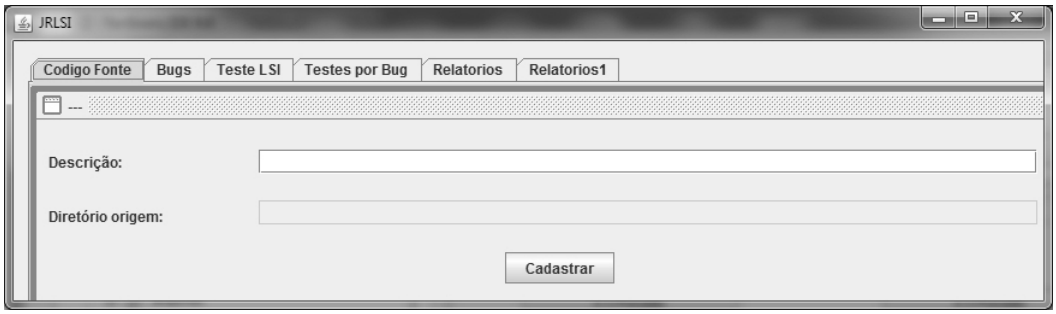


Figura B.2: Cadastro de defeitos

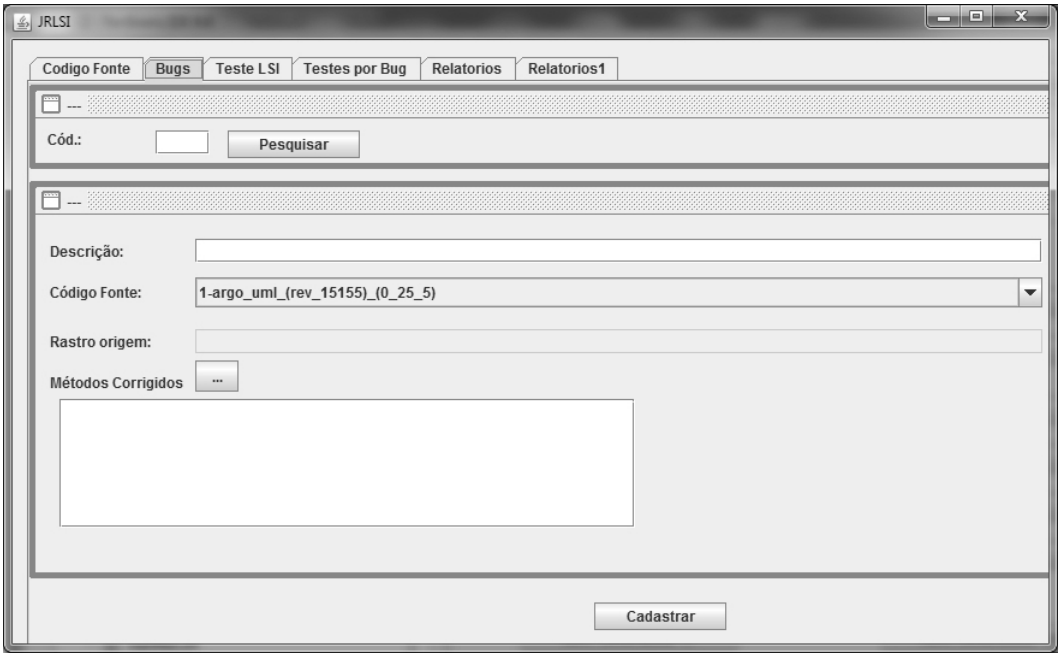


Figura B.3: Realização de testes individuais, por defeito ou por lote de defeitos

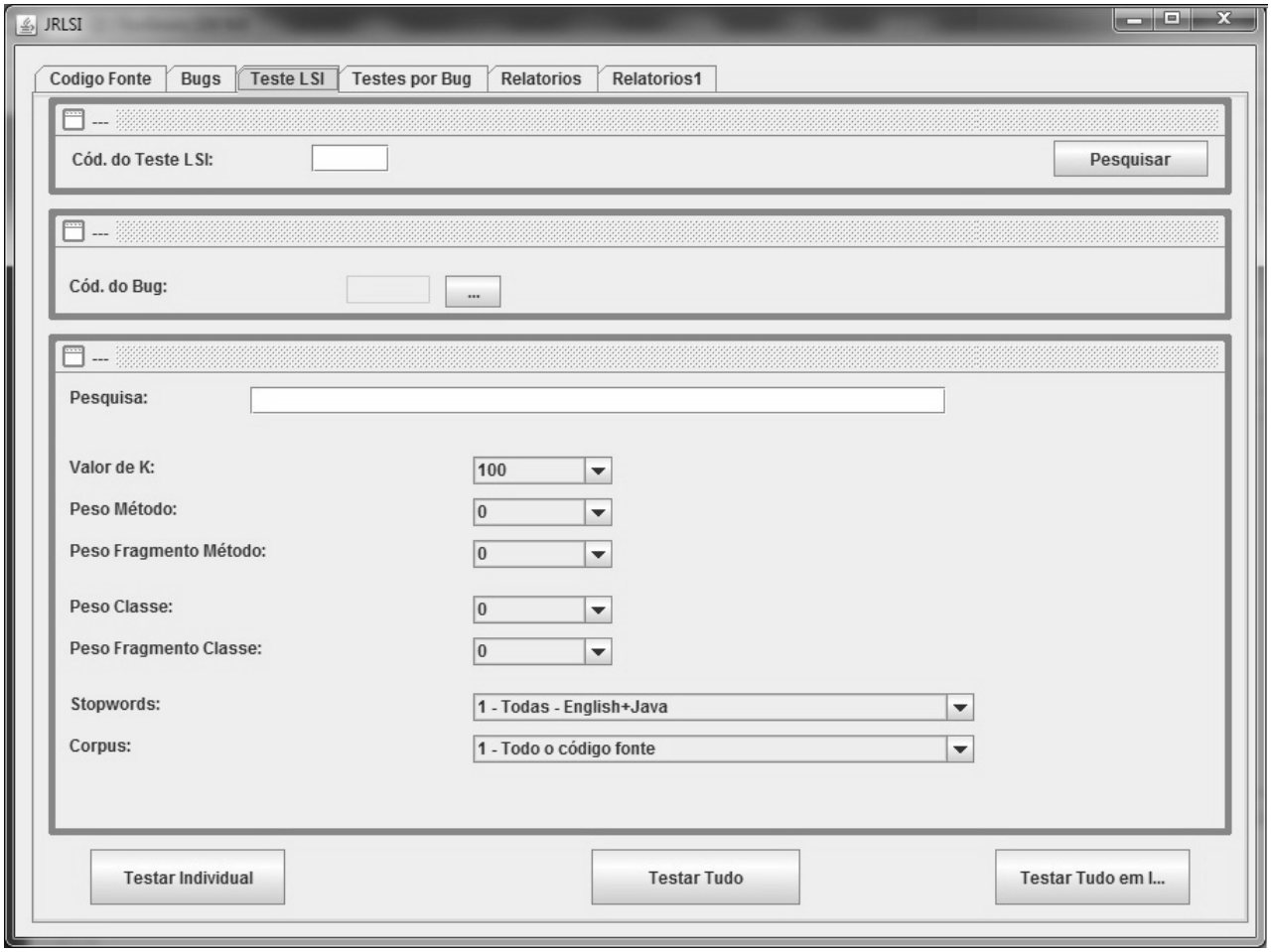


Figura B.4: Emissão de gráficos e relatórios



Apêndice C

Armazenamento dos dados referentes às amostras de pesquisa e aos resultados encontrados

Os dados referentes a cada defeito e seu respectivo código fonte são guardados em um banco de dados relacional. A Figura C.1 mostra o diagrama de Entidade-Relacionamento extraído do banco de dados projetado para dar suporte à atividade de pesquisa.

Cada abordagem para filtro de código fonte foi armazenada em bancos de dados distintos, compondo um total de 3 bancos de dados:

- Banco de dados 1 - armazena dados referentes à abordagem que filtra métodos para o corpus com base nos rastros de execução do defeito (ET).

- Banco de dados 2 - armazena dados referentes à abordagem que filtra métodos para o corpus com base nos rastros de execução do defeito e nos termos da consulta (QTL).

- Banco de dados 3 - armazena dados referentes a testes específicos realizados com todo o código fonte do JEdit e ArmedBear-J, buscando prover comparações entre os dois softwares nos quesitos acurácia e performance.

As tabelas do banco de dados armazenam o código fonte, as informações relativas aos defeitos e resultados dos testes. Uma descrição detalhada de cada tabela pode ser assim enunciada:

- CODIGO_FONTE - armazena uma descrição sucinta da versão e revisão do código fonte.

- METODO_BRUTO - armazena o conjunto de todos os métodos do código fonte para cada versão de software analisada.

- METODO_CARACTERISTICA - armazena os métodos acionados durante a reprodução do defeito.

- BUGS - armazena uma descrição sucinta de cada defeito.

- BUGS_CORRIGIDOS - armazena uma referência aos métodos que precisaram ser corrigidos, com base nas mensagens de *commit* para correção de defeitos.

- TESTELSI - armazena os parâmetros de busca utilizados em um teste, bem como a consulta submetida e a precisão média encontrada para o mesmo.

- RESULTADO_TESTELSI - armazena a lista de métodos ordenados com suas respectivas correlações para cada teste de combinação.

- PREC_REVOCACAO - armazena as precisões encontradas para cada método relevante à correção do defeito e suas respectivas revocações.

Os dados armazenados nestas tabelas são usados para geração de relatórios e gráficos. Estes expressam a influência de cada parâmetro no conjunto de testes e a classificação das combinações de parâmetros quanto ao trabalho de análise demandado para localização de métodos no código fonte. Uma ferramenta desenvolvida na linguagem Java permite gerenciar todo o processo de armazenamento de dados, realização dos testes e emissão de relatórios.

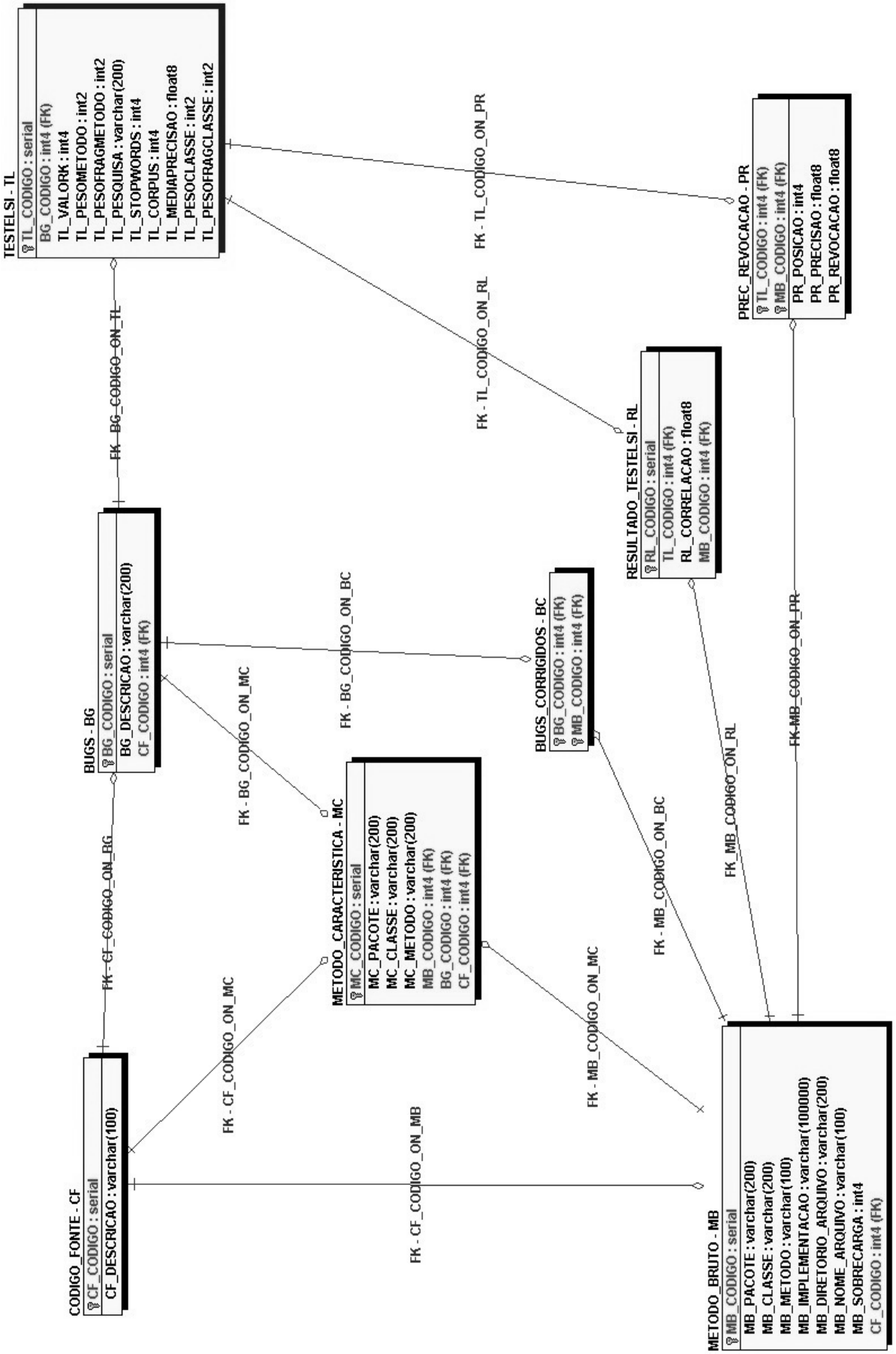


Figura C.1: Diagrama Entidade-Relacionamento do banco de dados utilizado

Apêndice D

Comparativo entre os software JEdit e ArmedBear-J

Testes comparativos entre o Jedit e ArmedBear / J utilizando as três abordagens para a combinação: [k=60] - [PM=SIM] - [PFM=SIM] - [PC=SIM] - [PFC=SIM] - [STW=JSTW] Todos os parâmetros desta combinação, com exceção do valor de k que era 300 e foi alterado para 60 em virtude da redução do corpus do J, se mostraram predominantes para a abordagem QTL utilizando o Jedit.

Bugs do Jedit

BUG	11			
Código Fonte	jedit_(rev_13899)_(4_3_pre_16)			
Consulta	buffer replace regex line hypersearch hyper search			
ABORDAGEM	**TEMPO EM SEGUNDOS**	**[PRECISÃO] [REVOCAÇÃO] [RANKING ULTIMO METODO]**	**NÚMERO DE DOCUMENTOS NO CORPUS**	**QUANTIDADE DE INFORMAÇÃO**
Todo o código fonte	4189	[0,2] [1] [5]	7071	3,60 MB (3.783.881 bytes)
ET	169	[0,25] [1] [4]	1164	668 KB (684.356 bytes)
QTL	44	[0,25] [1] [4]	387	263 KB (270.185 bytes)

BUG	12			
Código Fonte	jedit_(rev_15833)_(4_3_pre_17)			
Consulta	buffer search filename file name filter			
ABORDAGEM	**TEMPO EM SEGUNDOS**	**[PRECISÃO] [REVOCAÇÃO] [RANKING ULTIMO METODO]**	**NÚMERO DE DOCUMENTOS NO CORPUS**	**QUANTIDADE DE INFORMAÇÃO**
Todo o código fonte	3882	[4.6468401486988845E-4] [1] [2152]	7070	3,55 MB (3.732.719 bytes)
ET	236	[0.002364066193853428] [1] [423]	1544	833 KB (853.750 bytes)
QTL	93	[0.14285714285714285] [1] [7]	748	500 KB (512.402 bytes)

BUG	13			
Código Fonte	jedit_(rev_15716)_(4_3_pre_17)			
Consulta	screen toggle action bar utility bottom text area view full mode			
ABORDAGEM	**TEMPO EM SEGUNDOS**	**[PRECISÃO] [REVOCAÇÃO] [RANKING ULTIMO METODO]**	**NÚMERO DE DOCUMENTOS NO CORPUS**	**QUANTIDADE DE INFORMAÇÃO**
Todo o código fonte	4195	[0.002183406113537118] [1] [458]	7073	3,55 MB (3.733.078 bytes)
ET	113	[0,0833333333333333] [1] [12]	923	506 KB (519.130 bytes)
QTL	43	[0.16666666666666666] [1] [6]	376	268 KB (275.041 bytes)

Bugs do ArmedBear/J

BUG	21			
Código Fonte	j_rev_8874			
Consulta	session editor name setSessionName set closeAll close			
ABORDAGEM	**TEMPO EM SEGUNDOS**	**[PRECISÃO] [REVOCAÇÃO] [RANKING ULTIMO METODO]**	**NÚMERO DE DOCUMENTOS NO CORPUS**	**QUANTIDADE DE INFORMAÇÃO**
Todo o código fonte	4262	[0.005681818181818182] [1] [176]	9104	4,44 MB (4.663.048 bytes)
ET	86	[0.023255813953488372] [1] [43]	769	380 KB (389.588 bytes)
QTL	44	[0.030303030303030304] [1] [33]	407	298 KB (305.717 bytes)

BUG	22			
Código Fonte	j_rev_392			
Consulta	indent opening brace method			
ABORDAGEM	**TEMPO EM SEGUNDOS**	**[PRECISÃO] [REVOCAÇÃO] [RANKING ULTIMO METODO]**	**NÚMERO DE DOCUMENTOS NO CORPUS**	**QUANTIDADE DE INFORMAÇÃO**
Todo o código fonte	3812	[0,176923076923077] [1] [13]	7761	4,08 MB (4.278.650 bytes)
ET	103	[0,833333333333333] [1] [3]	887	450 KB (461.731 bytes)
QTL	10	[0,833333333333333] [1] [3]	60	123 KB (126.941 bytes)

BUG	23			
Código Fonte	j_rev_2630			
Consulta	bad wildcard replace file dialog box ok button unsupported wild card pattern			
ABORDAGEM	**TEMPO EM SEGUNDOS**	**[PRECISÃO] [REVOCAÇÃO] [RANKING ULTIMO METODO]**	**NÚMERO DE DOCUMENTOS NO CORPUS**	**QUANTIDADE DE INFORMAÇÃO**
Todo o código fonte	4188	[0,00965288489239472] [1] [430]	8237	4,24 MB (4.447.702 bytes)
ET	93	[0,0389784209411996] [0,833333333333333] [107]	822	380 KB (389.706 bytes)
QTL	30	[0,0369356139356139] [0,833333333333333] [104]	268	219 KB (225.187 bytes)