Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Dissertação de Mestrado

Estudo do Uso de Vocabulários para Analisar o Impacto de Relatórios de Defeitos a Código-Fonte

Diego Tavares Cavalcanti

Campina Grande – Paraíba – Brasil
Novembro de 2012

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Estudo do Uso de Vocabulários para Analisar o Impacto de Relatórios de Defeitos a Código-Fonte

Diego Tavares Cavalcanti

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande – Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Dalton Dario Serey Guerrero
Jorge César Abrantes de Figueiredo
(Orientadores)

Campina Grande – Paraíba – Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

C376e Cavalcanti, Diego Tavares.

Estudo do uso de vocabulários para analisar o impacto de relatórios de defeitos a código-fonte / Diego Tavares Cavalcanti. – Campina Grande, 2012.

76 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.

Orientadores: Prof. Dr. Dalton Serey Guerrero, Prof. Dr. Jorge César Abrantes de Figueiredo.

Referências.

1. Software – Localização de Defeitos. 2. Vocabulário de Software. 3. Análise de Impacto. I. Título.

CDU 004.4(043)

"ESTUDO DO USO DE VOCABULÁRIOS PARA ANALISAR O IMPACTO DE RELATÓRIOS DE DEFEITOS A CÓDIGO-FONTE"

DIEGO TAVARES CAVALCANTI

DISSERTAÇÃO APROVADA EM 26/11/2012

JORGE CESAR ABRANTES DE FIGUEIREDO, D.Sc, UFCG
Orientador(a)

DALTON DARIO SEREY GUERRERO, D.Sc, UFCG
Orientador(a)

TIAGO LIMA MASSONI, Dr., UFCG
Examinador(a)

CHRISTINA VON FLACH GARCIA CHAVEZ, Drª, UFBA
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Localizar e corrigir defeitos são tarefas comuns no processo de manutenção de software. Entretanto, a atividade de localizar entidades de código que são possivelmente defeituosas e que necessitam ser modificadas para a correção de um defeito, não é trivial. Geralmente, desenvolvedores realizam esta tarefa por meio de um processo manual de leitura e inspeção do código, bem como de informações cadastradas em relatórios de defeitos. De fato, é necessário que os desenvolvedores tenham um bom conhecimento da arquitetura e do design do software a fim de realizarem tal tarefa. Entretanto, este conhecimento fica espalhado por entre a equipe e requer tempo para ser adquirido por novatos. Assim, é necessário o desenvolvimento de técnicas que auxiliem na tarefa de análise de impacto de relatórios de defeitos no código, independente da experiência do desenvolvedor que irá executá-la. Neste trabalho, apresentamos resultados de um estudo empírico no qual avaliamos se a análise automática de vocabulários de relatórios de defeitos e de software pode ser útil na tarefa de localizar defeitos no código. Nele, analisamos similaridade de vocabulários como fator para sugerir classes que são prováveis de serem impactadas por um dado relatório de defeito. Realizamos uma avaliação com oito projetos maduros de código aberto, desenvolvidos em Java, que utilizam Bugzilla e JIRA como seus repositórios de defeitos. Nossos resultados indicam que a análise de ambos os vocabulários é, de fato, uma fonte valiosa de informação, que pode ser utilizada para agilizar a tarefa de localização de defeitos. Para todos os sistemas estudados, ao considerarmos apenas análise de vocabulário, vimos que, mesmo com um ranking contendo apenas 8% das classes de um projeto, foi possível encontrar classes relacionadas ao defeito buscado em até 75% dos casos. Portanto, podemos concluir que, mesmo que não possamos utilizar vocabulários de software e de relatórios de defeitos como únicas fontes de informação, eles certamente podem melhorar os resultados obtidos, ao serem combinados com técnicas complementares.

Abstract

Locating and fixing bugs described in bug reports are routine tasks in software development processes. A major effort must be undertaken to successfully locate the (possibly faulty) entities in the code that must be worked on. Generally, developers map bug reports to code through manual reading and inspection of both bug reports and the code itself. In practice, they must rely on their knowledge about the software architecture and design to perform the mapping in an efficient and effective way. However, it is well known that architectural and design knowledge is spread out among developers. Hence, the success of such a task is directly depending on choosing the right developer. In this paper, we present results of an empirical study we performed to evaluate whether the automated analysis of bug reports and software vocabularies can be helpful in the task of locating bugs. We conducted our study on eight versions of six mature Java open-source projects that use Bugzilla and JIRA as bug tracking systems. In our study, we have used Information Retrieval techniques to assess the similarity of bug reports and code entities vocabularies. For each bug report, we ranked all code entities according to the measured similarity. Our results indicate that vocabularies are indeed a valuable source of information that can be used to narrow down the bug-locating task. For all the studied systems, considering vocabulary similarity only, a Top 8% list of entities has about 75% of the target entities. We conclude that while vocabularies cannot be the sole source of information, they can certainly improve results if combined with other techniques.

Agradecimentos

Sou imensamente agradecido a Deus – inteligência suprema e causa primária de todas as coisas – por me permitir ter força de vontade para que eu siga os meus ideais e alcance os meus objetivos, buscando sempre evoluir moral e intelectualmente. Sou grato a Ele e à Espiritualidade superior que me acompanhou durante esta empreitada, por terem mantido acesas em mim a chama da curiosidade e a disposição para o trabalho.

Todo o meu agradecimento também cabe à mainha, por sempre ter me apoiado e trabalhado duro para que minha educação e condições de vida fossem as melhores possíveis. Além disso, agradeço ao meu pai por ter se preocupado várias vezes em perguntar como estavam as coisas e por vibrar junto comigo em cada conquista; e à minha irmã pelo enorme apoio e por estar sempre ao meu lado.

Agradeço também a Elisa, por ser a companheira certa em todos os momentos e por tornar os meus dias mais amenos. Por não me deixar desistir de nada e por, cada vez mais, ser minha fonte de motivação para crescer, tanto como pessoa, quanto como profissional.

A Dalton e a Jorge, dois professores extraordinários os quais eu tive a sorte de ter como orientadores durante o mestrado. Sou extremamente grato por terem me recebido como orientando, por acreditarem no meu trabalho e por todo suporte e ensinamentos que me deram durante essa temporada na qual trabalhamos juntos.

Aos companheiros do GMF/SPLab, em especial Andreza, Dudu, Everton, Franklin e Raquel; e àqueles do grupo de Evolução de Software – Katyusco, João, Eliane, Matheus e Roberto. Muito obrigado por serem, além de companheiros de trabalho, amigos com os quais sempre pude contar, fosse para discutir sobre a pesquisa, auxiliar com ideias ou apenas para compartilhar os fatos do dia-a-dia. Não esquecendo também de Jemão e Bel que já saíram, mas que também deixaram grandes contribuições. Além disso, agradeço também a Catharine, Delano, Samuel e Tercio pela ajuda com o suporte ferramental do trabalho.

Também sou muito grato à professora Joseana, que foi minha tutora por muitos anos, com quem eu tive a honra de aprender imensamente; e aos professores Rohit e Tiago Massoni, aos quais eu devo muito do que sei hoje em dia sobre pesquisa. Sem os ensinamentos dos três durante a minha graduação, eu não seria metade do profissional que eu sou hoje.

Tenho também muito a agradecer aos amigos que conquistei na universidade, em especial Arthur Marques, Camilla, Danilo, Elloá, Lala, Leandro, Lorena, Mari, Mikaela, Solon e Tx; aos amigos da SEJA, que, mesmo não podendo enumerar todos, listo alguns especiais: Amanda, Arthur Chaves, Ceiça, Daniel, Denise, Emily, Felipe, Ibysson, Larissa, Tássio, Thâmisa, Renner e Rodrigo; aqueles do CDT: Alexandre, Anunciada, Betânia, Inaldete, Lila, Marcelo, Paulinho e Socorro Xuxa; e, por fim, aos amigos familiares, em especial: Aluska, Nana, Roseli e Vanderli. Serei eternamente grato por toda a torcida, motivação e apoio que recebi de todos nos momentos que precisei.

Por fim, agradeço ao governo brasileiro, por ter apoiado financeiramente toda a minha pesquisa; e ao pessoal da COPIN, em especial Aninha, Rebeka e Vera, que sempre me atenderam prontamente com as burocracias da pós-graduação.

Epígrafe

"We are dwarfs astride the shoulders of giants. We master their wisdom and move beyond it. Due to their wisdom we grow wise and are able to say all that we say, but not because we are greater than they."

Isaiah di Trani, c. 1180 – c. 1250

Dedicatória

A todos que, desde muito cedo, me incentivaram a seguir pelo caminho da computação.

Conteúdo

Lista de Acrônimos

CVS	*Concurrent Versions System*
IDE	*Integrated Development Environment*
IDF	*Inverse Document Frequency*
RI	Recuperação da Informação
SVN	*Apache Subversion*
TF	*Term Frequency*
VXL	*Vocabulary XML*
XML	*Extensible Markup Language*

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Capítulo 1

Introdução

1.1 Contextualização

Localizar e corrigir defeitos são tarefas rotineiras no dia-a-dia de desenvolvedores de software. De fato, estima-se que entre 60% e 80% dos recursos de um projeto sejam gastos em atividades de evolução de software [1]. Dentre elas, podemos citar, em especial, a atividade de correção de defeitos, que tem duas questões importantes, a partir da perspectiva do desenvolvedor: 1) *qual é o defeito?* e 2) *onde ele está localizado?*

Em relação à primeira pergunta, em grandes projetos, informações sobre quais defeitos foram reportados ficam catalogadas em repositórios de defeitos. Tais repositórios contêm **relatórios de defeitos** (em inglês, *bug reports*), que são relatórios nos quais usuários e desenvolvedores registram informações sobre problemas encontrados no software. Muitas vezes, tais relatórios são a única fonte de informação que a equipe de desenvolvimento tem sobre os defeitos do software.

Por outro lado, para encontrar onde um defeito está localizado (segunda pergunta), os desenvolvedores geralmente têm que se basear apenas em informações contidas nos relatórios. Essa tarefa não é trivial, dado que, nas maioria das vezes, não há mapeamento direto entre o relatório de defeito e as entidades de código defeituosas.

Portanto, a demora em localizar defeitos descritos em tais relatórios é uma realidade de grandes projetos no contexto atual da engenharia de software. O processo que vai desde a submissão de um relatório de defeito até a etapa de correção do mesmo é custoso, pois é preciso, sobretudo, que o desenvolvedor alocado para a tarefa seja capaz de entender o

problema descrito no relatório e mapeá-lo corretamente para a porção do código fonte que precisará ser corrigida.

Essa tarefa é relativamente simples para pequenos projetos, que têm poucas linhas de código e uma equipe menor, que entende de todo o código. Entretanto, à medida que o projeto cresce ou que a equipe muda, nem todos os desenvolvedores estarão aptos a entender completamente o código, de modo que, a partir da leitura de um relatório, muitas vezes escrito com poucos detalhes, todos sejam capazes de identificar exatamente as classes que deverão ser modificadas para a correção do problema. De fato, já foi visto [2] que a tarefa de localizar entidades de código defeituosas em projetos grandes tem um custo significativo, especialmente porque a tarefa de determinar em quais entidades de código o defeito está localizado não escala à medida que o sistema cresce.

Nos últimos anos, vários trabalhos tentaram resolver esse problema utilizando dados históricos extraídos de repositórios de relatórios de defeitos [3–8]. Entretanto, tais abordagens não podem ser utilizadas com novos projetos, nem com novos artefatos de código, já que entidades recém-criadas ainda não possuem um histórico relevante para tais técnicas. Por exemplo, relatórios de defeitos relacionados a novas funcionalidades podem não ser comparáveis aos relatórios anteriores do repositório, dado que eles, possivelmente, conterão um vocabulário inédito, proveniente das novas entidades do projeto. Portanto, abordagens que utilizem histórico seriam comprometidas em casos como esse, uma vez que elas funcionam apenas a partir da análise de similaridade com dados antigos. É o que, em inglês, chama-se de *cold start problem* [9].

Diante do exposto, percebe-se que falta um estudo específico que explore o fator **Vocabulário de Software** como fonte de informação para a realização da tarefa de análise de impacto de relatórios de defeitos no código, em especial para casos nos quais não existem dados históricos de código fonte e de repositório de relatórios de defeitos.

1.2 Estudo Realizado

Neste trabalho, conduzimos um estudo empírico para avaliar se a análise automática de vocabulários de software e de relatórios de defeitos pode ser útil para diminuir o custo da tarefa de localização de defeitos no código. Para tanto, extraímos ambos os vocabulários e utilizamos

técnicas de Recuperação da Informação (RI) para ranquear classes, a partir da similaridade de seus vocabulários com os dos relatórios de defeitos analisados. Assim, foi possível analisar se os rankings criados são úteis para retornar classes que são mais prováveis de serem impactadas para a correção de um defeito descrito em um dado relatório.

Nosso foco foi estudar se similaridade de vocabulários pode ser utilizada para auxiliar a tarefa supracitada. Desse modo, a abordagem proposta neste trabalho utiliza-se apenas de simples algoritmos de RI para analisar os vocabulários e as similaridades que eles têm entre si. Ao considerarmos apenas os vocabulários e suas características, acreditamos que podemos desenvolver um bom estudo sobre qual a possibilidade de utilização de vocabulário de software para a análise de impacto.

Como forma de avaliar a abordagem proposta, analisamos oito versões de projetos maduros de código aberto: três versões do Eclipse e cinco projetos da fundação Apache. No total, analisamos mais de 9.500 relatórios de defeitos e processamos o vocabulário de mais de 27 mil classes implementadas na linguagem Java.

Com a avaliação, procuramos entender o impacto causado na correção dos relatórios analisados e quanto o fator vocabulário pode auxiliar na sugestão de classes prováveis de serem modificadas.

Os resultados confirmaram que a tarefa de encontrar defeitos no código é árdua, dado que desenvolvedores geralmente têm que encontrar de uma a três classes que serão modificadas para correção do defeito, dentre centenas de classes presentes em cada projeto. Além disso, vimos que, apesar de vocabulário não poder ser utilizado como fonte única de sugestão de classes possivelmente impactadas, ele é, de fato, útil para diminuir o espaço de busca dos desenvolvedores, ao analisarem o código. Por exemplo, com nossa abordagem, consegue-se desconsiderar em torno de 92% do projeto e, ainda assim, encontrar uma classe defeituosa em 75% das vezes.

1.3 Estrutura do Documento

Este documento está estruturado da seguinte forma:

Capítulo 2 - Fundamentação Teórica: apresenta conceitos necessários para o entendimento do trabalho apresentado. Tais conceitos abrangem relatórios de defeitos, recuperação

da informação e vocabulário de software;

Capítulo 3 - Uso de Vocabulários para Localização de Defeitos: apresenta dados que nos motivaram a utilizar vocabulário para a localização de defeitos no código; e descreve a abordagem proposta para este trabalho;

Capítulo 4 - Avaliação: descreve o planejamento do estudo desenvolvido, tais como sua metodologia, projetos analisados, coleta e instrumentação das amostras, questões de pesquisa investigadas e a métrica analisada. Além disso, apresenta resultados e sua análise, bem como as limitações do estudo;

Capítulo 5 - Discussão Geral: apresenta uma discussão geral sobre os resultados encontrados e delineia ideias para trabalhos futuros, baseados em nossa experiência;

Capítulo 6 - Trabalhos Relacionados: apresenta trabalhos da literatura que são relacionados ao nosso. Para cada trabalho, apresentamos brevemente sua metodologia, seus resultados e como eles se relacionam com o presente estudo;

Capítulo 7 - Conclusão: apresenta as contribuições da pesquisa desenvolvida e considerações finais sobre o trabalho.

Capítulo 2

Fundamentação Teórica

Para o bom entendimento do estudo apresentado neste documento, faz-se necessário o conhecimento de alguns tópicos, que são relacionados ao trabalho desenvolvido.

Neste capítulo, são apresentados brevemente conceitos importantes referentes a este trabalho. Na Seção 2.1 apresentamos a definição e estrutura de relatórios de defeitos; enquanto que na Seção 2.2, descrevemos, de forma sucinta, conceitos e algoritmos da área de recuperação da informação. Por fim, a Seção 2.3 contém conceitos básicos sobre vocabulário de software.

2.1 Relatórios de Defeitos

Um grande número de projetos de software utiliza sistemas para gerenciar as mudanças que precisam ser feitas em cada versão lançada do software. Tais sistemas são comumente chamados Repositórios de Defeitos ou Sistemas de Rastreamento de Defeitos e, geralmente, suportam o registro de requisições de mudanças e de relatórios de defeitos.

No presente estudo, extraímos informações de **relatórios de defeitos**, mais conhecidos por seu termo em inglês, *bug reports*. Na próxima seção, apresentaremos como se estruturam os relatórios de defeitos dos dois sistemas mais comuns utilizados em projetos de código aberto: Bugzilla[1] e JIRA[2]. Além disso, na Seção 2.1.2, apresentamos o ciclo de vida comum desses relatórios e em qual etapa do processo o nosso trabalho se insere.

[1] `www.bugzilla.org`, verificado em outubro de 2012

[2] `www.atlassian.com/software/jira`, verificado em outubro de 2012

2.1.1 Estrutura de Relatórios de Defeitos

Os relatórios de defeitos utilizados atualmente por projetos de código aberto geralmente contêm a mesma estrutura. Cada relatório é representado por um único ID e contém quatro tipos diferentes de informações [10]: informações pré-definidas, anexos, dependências e texto livre.

A seção de **informações pré-definidas** armazena informações referentes ao sistema, tais como: produto, componente, versão e plataforma utilizada. Além disso, também contém informações para gerenciamento do relatório de defeito, tais como: status, prioridade, *milestone* no qual ele será processado, desenvolvedor designado para resolvê-lo, palavras-chave e datas de registro e de modificação. Muitos desses dados são fornecidos no momento de registro do relatório e o restante é automaticamente gerado ou fornecido pelo gerente do projeto.

Os **anexos** são permitidos de modo que o usuário que reportou o problema possa adicionar informações não textuais, tais como uma imagem da tela com o erro. É também por meio dos anexos que os desenvolvedores que colaboram com o projeto geralmente enviam *patches* de correção para aquele defeito reportado.

As **dependências** são registros de outros relatórios que são pré- ou pós-requisitos daquele.

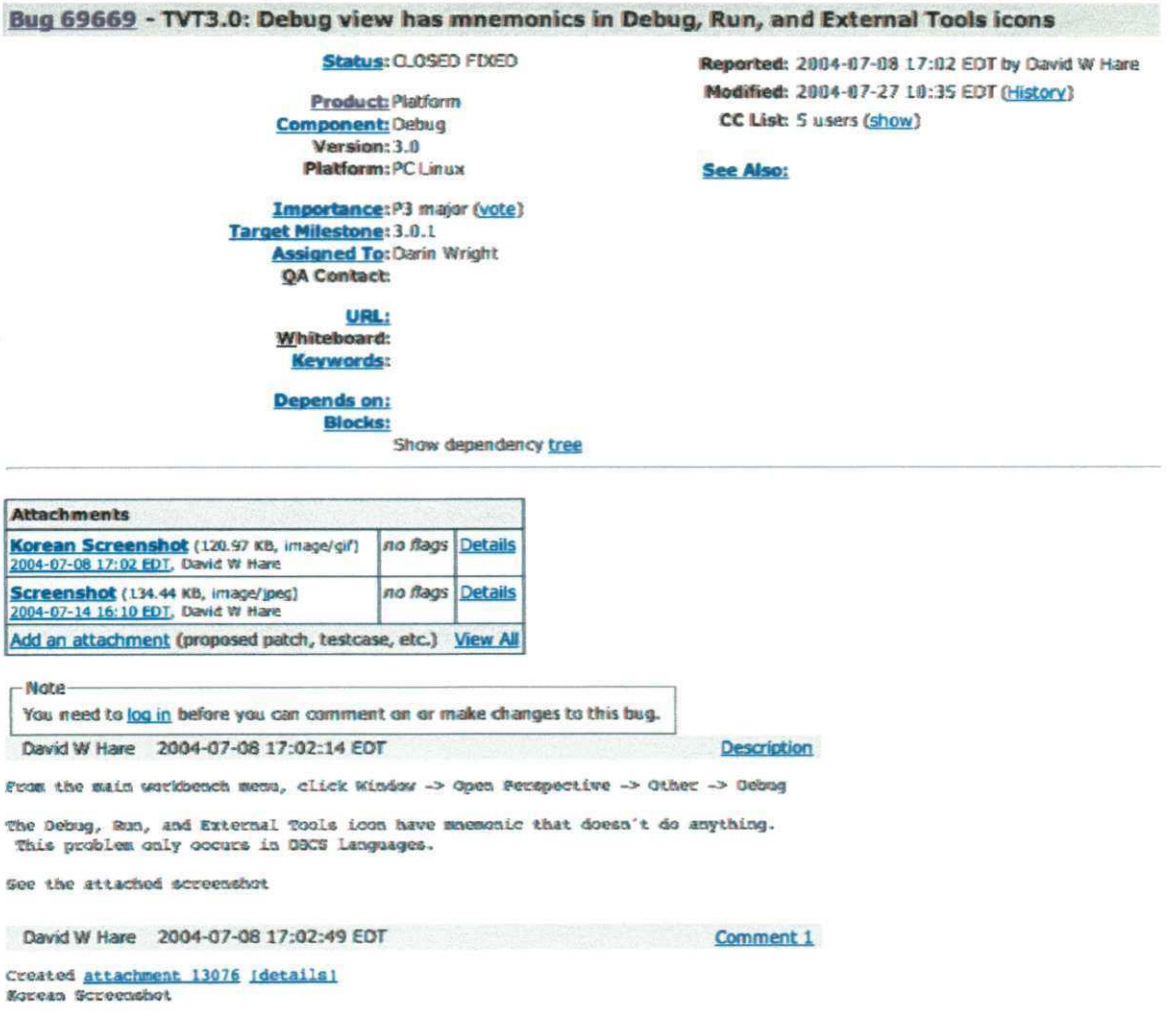
Por fim, o **texto livre** inclui o título do relatório, uma descrição do defeito e comentários. O título deve ser um resumo de uma linha sobre o defeito. Nele, geralmente, estão as palavras-chave que o descrevem. Já a descrição, deve conter informações detalhadas sobre o defeito encontrado, passos para reproduzi-lo e qualquer outro tipo de informação que possa ser útil para os desenvolvedores identificarem e resolverem o problema. Por fim, também há comentários adicionais que representam discussões sobre possíveis abordagens para resolver o defeito ou referências para outros relatórios que sejam relevantes para o problema descrito.

Tomemos como exemplo o relatório de defeito apresentado na Figura 2.1[3]. Ele faz parte do repositório de defeitos do projeto Eclipse, que utiliza o sistema Bugzilla para gerenciar seus defeitos.

Pelas informações pré-definidas que foram registradas para o relatório da figura, é possí-

[3]Relatório de defeito acessível via `https://bugs.eclipse.org/bugs/show_bug.cgi?id=69669`, verificado em outubro de 2012.

Figura 2.1: Exemplo de Relatório de Defeito do Bugzilla



Bug 69669 - TVT3.0: Debug view has mnemonics in Debug, Run, and External Tools icons

Status: CLOSED FIXED
Reported: 2004-07-08 17:02 EDT by David W Hare
Modified: 2004-07-27 10:35 EDT (History)
Product: Platform
Component: Debug
CC List: 5 users (show)
Version: 3.0
Platform: PC Linux
See Also:
Importance: P3 major (vote)
Target Milestone: 3.0.1
Assigned To: Darin Wright
QA Contact:
URL:
Whiteboard:
Keywords:
Depends on:
Blocks:
Show dependency tree

Attachments		
Korean Screenshot (120.97 KB, image/gif) 2004-07-08 17:02 EDT, David W Hare	no flags	Details
Screenshot (134.44 KB, image/jpeg) 2004-07-14 16:10 EDT, David W Hare	no flags	Details
Add an attachment (proposed patch, testcase, etc.)		View All

Note
You need to log in before you can comment on or make changes to this bug.

David W Hare 2004-07-08 17:02:14 EDT — Description

From the main workbench menu, click Window -> Open Perspective -> Other -> Debug

The Debug, Run, and External Tools icon have mnemonic that doesn't do anything.
This problem only occurs in DBCS Languages.

See the attached screenshot

David W Hare 2004-07-08 17:02:49 EDT — Comment 1

Created attachment 13076 [details]
Korean Screenshot

Fonte: Repositório de defeitos do projeto Eclipse

vel notar que ele já está com status fechado e corrigido, o tipo de produto ao qual o defeito se refere foi "*Platform*", o componente é "*Debug*" e a plataforma utilizada foi o "*PC Linux*". A prioridade deste relatório foi considerada como "*P3 major*" e o desenvolvedor designado para corrigi-lo foi Darin Wright. Também é possível perceber que este relatório foi reportado no dia 08 de julho de 2004, por David W Hare e sua última modificação foi em 27 de julho de 2004. Por fim, o relatório possui dois anexos, mas não possui dependências.

Quanto ao texto livre presente no relatório, vemos que seu título foi registrado como: "*TVT3.0: Debug view has mnemonics in Debug, Run, and External Tools icons*" e a descrição que David W Hare escreveu ao registrar o relatório também se faz presente. Por fim, também há um comentário adicional, escrito por ele mais tarde, indicando um anexo que foi submetido.

Analisemos também o relatório de defeito apresentado na Figura 2.2[4]. Ele faz parte do projeto Apache Shiro, que utiliza o sistema JIRA para gerenciar seus defeitos.

De forma similar, é possível perceber que, assim como o Bugzilla, ele também contém informações pré-definidas (prioridade, componentes, status, datas, desenvolvedores designados, etc.), título (no exemplo, "*ServletContainerSessionManager does not honor web.xml's session timeout value*"), descrição e comentários. Anexos e dependências também podem estar disponíveis no JIRA, mas foram desabilitados para o repositório do projeto Apache Shiro.

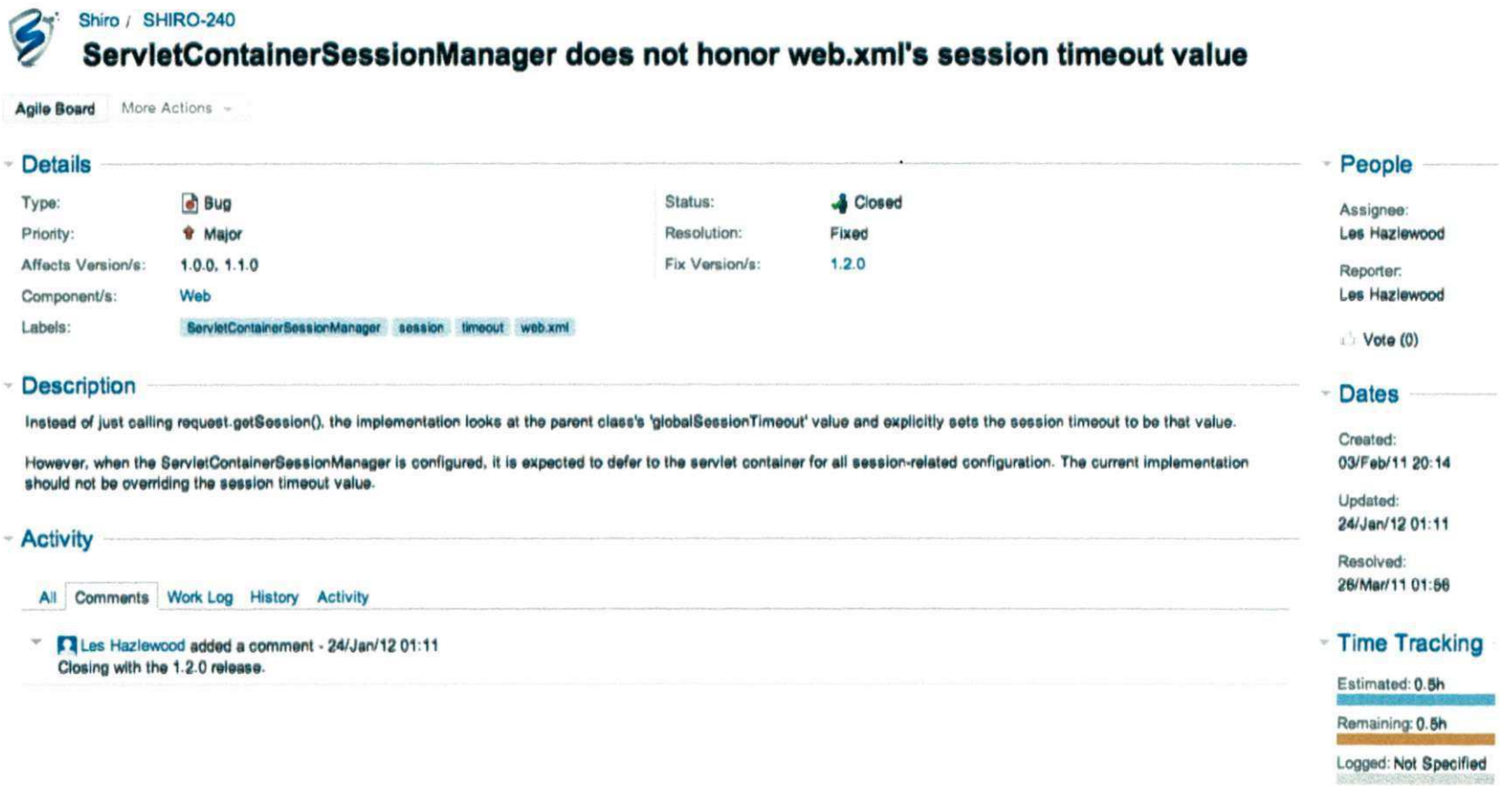
Desse modo, podemos notar que ambos os tipos de relatórios de defeitos contêm os mesmos tipos de informação, de forma estruturada. No presente trabalho, focamos na extração e análise do texto livre, presente tanto no título quanto na descrição dos defeitos.

2.1.2 Ciclo de Vida de Relatórios de Defeitos

Cada projeto de software tem suas particularidades em relação ao gerenciamento de seu repositório de defeitos. Cada gerente de projeto define o melhor fluxo de trabalho para a equipe e dita diretrizes que devem ser seguidas para os seus projetos. Entretanto, os relatórios de defeitos têm, em sua maioria, um ciclo de vida em comum, que está apresentado na Figura 2.3.

[4]Relatório de defeitos acessível via `https://issues.apache.org/jira/browse/SHIRO-240`, verificado em outubro de 2012.

Figura 2.2: Exemplo de Relatório de Defeito do JIRA



Fonte: Repositório de defeitos do projeto Apache Shiro

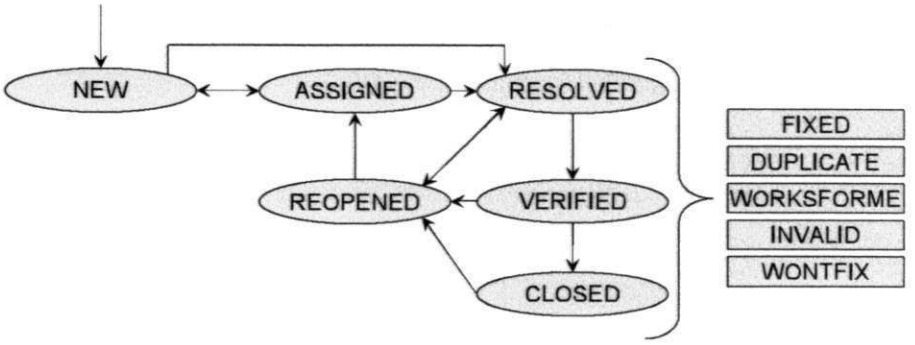
Figura 2.3: Ciclo de Vida de um Relatório de Defeito

Fonte: Documentação do projeto Yocto, acessível via `http://wiki.yoctoproject.org/`, verificado em outubro de 2012.

Quando um defeito é reportado, cria-se um novo relatório de defeito no repositório, contendo um ID único. O relatório recebe, então, o status de novo (caixa 1). Logo após, algum desenvolvedor é designado para resolver o defeito e, ao aceitar a tarefa, ele busca ler e entender o problema descrito (caixa 2). Dependendo da necessidade, o desenvolvedor ou muda o status do relatório para indicar que é preciso acrescentar mais informações sobre o problema encontrado (caixa 3); ou busca as entidades do código que devem ser modificadas e, então, resolve aquele defeito reportado (caixa 4). A partir da resolução do defeito, a pessoa que o reportou (ou o gerente de projetos) verifica se este foi, de fato, corrigido (caixa 5) e, caso tenha sido, encerra aquele relatório (caixa 6) ou, caso contrário, o reabre para ser investigado novamente (caixa 7).

Diante do exposto, é possível perceber que os status dos relatórios de defeitos são variados, durante o seu ciclo de vida. A Figura 2.4 apresenta os possíveis status dos relatórios presentes no repositório de defeitos do projeto Eclipse. Outros projetos apresentam pequenas variações deste modelo, mas nenhuma diferença significativa para o âmbito da nossa pesquisa.

Figura 2.4: Possíveis Status de um Relatório de Defeito



Fonte: Anvik et al. [10]

Quando o relatório é submetido, ele automaticamente recebe o status de novo (*NEW*). Quando um desenvolvedor é designado para resolvê-lo, ele passa a ter o status *ASSIGNED*. Quando o defeito é resolvido, seu status muda para *RESOLVED*. Depois, ele pode ser verificado (*VERIFIED*) e fechado (*CLOSED*) pelo gerente do projeto. Em qualquer momento, entretanto, ele pode ser reaberto (*REOPENED*) para novas investigações. Caso o relatório seja identificado como sendo uma duplicata de outro, reportado anteriormente, ele passa a ter o status *DUPLICATE*. Por fim, se o desenvolvedor não pôde reproduzir o defeito, ele pode marcá-lo como *WORKSFORME*; ou se foi verificado que o relatório refere-se a algo que não é um defeito ou não tem importância para o projeto, seu status pode mudar para *INVALID* ou *WONTFIX*, respectivamente.

Nosso trabalho refere-se à etapa entre os status *ASSIGNED* e *RESOLVED*. Ou seja, na etapa na qual o desenvolvedor aceita resolver o defeito e o investiga para corrigi-lo. Essa é uma das tarefas mais trabalhosas de todo o processo, já que compreende ler o relatório e entender o defeito, além de encontrar quais entidades do código fonte referem-se a ele.

2.2 Recuperação da Informação

Manning et al. [11] definem recuperação da informação (RI) como sendo a atividade que objetiva encontrar material (geralmente, documentos) de natureza não-estruturada (geralmente, texto) em grandes coleções (geralmente, armazenadas em computadores).

Em termos do nosso estudo, entidades de código (i.e. classes Java) são representadas como documentos. A partir disso, utilizamos algoritmos de RI para extrair, tratar e indexar o vocabulário dessas entidades.

Seguem algumas definições básicas de RI [11], que são necessárias para o bom entendimento desse trabalho:

- ***Tokens*** são as unidades mais básicas de um documento. Geralmente, *tokens* são palavras de um texto separadas por espaço, excluindo-se certos caracteres, tais como pontuação.

- **Remoção de *stop words*** é o processo de excluir do vocabulário *tokens* que representam palavras extremamente comuns e que aparecem em quase todo texto. A sua remoção é importante, dado que elas acrescentam pouco ou nenhum valor ao documento. Exemplos de *stop words* do nosso trabalho são: *the*, *is*, *a*, *but*, *bug*, *eclipse*, *apache*. A lista completa de *stop words* que foram utilizadas neste trabalho pode ser encontrada no Apêndice B.

- **Normalização de *token*** é o processo de normalizar um *token* de forma que ele possa ser encontrado, independente de diferenças superficiais em sua sequência de caracteres. Por exemplo, *plug-in* e *plugin*; ou *Class* e *class*.

- ***Stemming*** é o processo de reduzir o *token* ao seu radical utilizando alguma heurística (em nosso caso, utilizamos o algoritmo de Porter [12], por ser o mais utilizado na área de RI). O processo de *stemming* é necessário pois, ao processar um texto, geralmente deseja-se considerar diferentes formas da palavra. Por exemplo, deseja-se considerar *program*, *programs* e *programming* como sendo a mesma palavra, já que elas carregam semântica similar. Além disso, há algumas palavras que tem derivações relacionadas e podem conter significados similares, tais como *am*, *are*, *is* e *be*. Assim, elas precisam ser tratadas como sendo um único termo.

- **Termo** é um *token* que foi normalizado e processado com *stemming*.

- **Documento** é uma sequência de termos.

Nas Seções 2.2.1 e 2.2.2, apresentamos técnicas e algoritmos de RI os quais utilizamos em nosso estudo.

2.2.1 Indexação e Busca de Vocabulário

Conforme já foi visto, a área de recuperação da informação baseia-se na busca de informações relevantes em documentos. Ou seja, dada uma *query* de busca, procura-se nos documentos a informação desejada e retornam-se aqueles que contêm esta informação, ordenados pela probabilidade que cada um tem de ser o mais útil para o usuário.

Uma forma óbvia de fazer tal busca é escanear sequencialmente os arquivos, em busca dos termos desejados. Isso indica realizar buscas em arquivos que não foram pré-processados. Portanto, essa é uma abordagem ingênua para tal tarefa. Uma outra opção, mais utilizada na prática, é construir estruturas de dados (chamadas de índices), que contêm os elementos do texto e, assim, aceleram o processo de busca.

Segundo Baeza-Yates e Ribeiro-Neto [13], existem três técnicas principais de indexação: índice invertido, vetores de sufixos e arquivos de assinatura. Como a técnica de índice invertido é a mais utilizada e é a que utilizamos em nosso trabalho, vamos apresentá-la de forma sucinta a seguir.

Um índice invertido é um mecanismo de indexação composto por dois elementos: vocabulário e ocorrências. Para cada palavra do vocabulário, é armazenada uma lista de posições do texto na qual a palavra aparece. O conjunto de todas essas listas é chamado de ocorrências [13]. No contexto do nosso trabalho, os termos distintos do vocabulário do software são indexados e mapeados com as suas ocorrências, nas diferentes classes.

A partir da indexação, a busca é feita de forma simples: constrói-se uma *query* de busca contendo os termos desejados e recupera-se todos os documentos que contêm ocorrências daqueles termos. Depois, cada documento é ordenado de acordo com algum algoritmo previamente escolhido. No nosso estudo, as *queries* são compostas pelos termos dos relatórios de defeitos. Além disso, utilizamos o Modelo de Espaço Vetorial (detalhado na Seção 2.2.2) para auxiliar na busca por similaridade de vocabulário.

2.2.2 Modelo de Espaço Vetorial

No Modelo de Espaço Vetorial (em inglês, *Vector Space Model*), documentos e *queries* são representados em forma de vetores, nos quais, a cada termo indexado, é atribuído um peso. Ou seja, no contexto da nossa pesquisa, ambos o vocabulário do software e o vocabulário de

cada relatório de defeito analisado são representados em forma de vetores, que contêm cada termo extraído mapeado a um peso. O TF-IDF é a métrica utilizada para representar o peso de cada termo dos vetores.

O TF é calculado a partir da Frequência dos Termos em cada documento (ou *query*) analisado. Ou seja, para cada documento d, é calculado o número de vezes que um termo t aparece nele. Documentos que contêm mais ocorrências de um dado termo, recebem um peso maior.

Já o IDF representa a Frequência Inversa de Documentos. Ele é calculado a partir do inverso do número de documentos no qual o termo t aparece. Isso significa que termos mais raros dão uma contribuição maior ao peso total do documento.

Sendo assim, o TF-IDF de um documento cresce proporcionalmente ao número de vezes que um termo aparece num documento, mas decresce se esse mesmo termo aparece em vários documentos. A ideia por trás disso é que termos que aparecem em vários documentos carregam pouco valor semântico. É o caso, por exemplo, de um projeto ter várias classes contendo uma variável chamada $temp$. Por mais que ela faça parte do vocabulário do software, ao estar presente em várias entidades, ela acrescenta pouco valor semântico ao vocabulário de uma classe.

Dados ambos os vetores, um contendo o vocabulário do software e outro o vocabulário do relatório de defeito, o Modelo de Espaço Vetorial calcula quão similar eles são. Essa similaridade é definida por meio do cálculo do cosseno do ângulo entre os vetores (conhecido como *similaridade de cossenos*). Ou seja, para um documento d_j e uma *query* q, a correlação entre os vetores $\overrightarrow{d_j}$ e $\overrightarrow{q}$ é calculada pelo produto escalar de ambos os vetores dividido pelo seu produto vetorial, tal como definido na seguinte equação [13]:

$$simCos(d_j, q) = \frac{\overrightarrow{d_j} \bullet \overrightarrow{q}}{|\overrightarrow{d_j}| \times |\overrightarrow{q}|}$$

$$= \frac{\sum_{i=1}^{t} w_{i,j} \times w_{i,q}}{\sqrt{\sum_{i=1}^{t} w_{i,j}^2} \times \sum_{j=1}^{t} w_{i,q}^2}$$

onde, t representa a quantidade de termos no vetor, $w_{i,j}$ representa o peso do termo i no documento j e $w_{i,q}$, o peso do termo i na query.

2.3 Vocabulário de Software

Vocabulário de software pode ser definido como sendo um conjunto de termos distintos do software, que são chamados de identificadores.

No paradigma de programação orientada a objetos, identificadores são geralmente utilizados para nomear estruturas estáticas do código fonte de um software, tais como: classes, interfaces, atributos, parâmetros, constantes e variáveis locais. Tais identificadores exercem um papel bastante importante para a compreensão do software [14].

Desenvolvedores precisam seguir regras sintáticas impostas pelas linguagens de programação para a escolha de identificadores (por exemplo, um identificador não pode começar com um dígito). Entretanto, não há restrições para que sejam definidos identificadores referentes a mais de uma palavra. Usualmente, no entanto, programadores que codificam em linguagens específicas tendem a seguir convenções de estilo ao definirem tais identificadores compostos. Por exemplo, de acordo com Enslen et al. [15], para sistemas implementados em Java, em torno de 88% dos identificadores compostos por mais de uma palavra, são definidos no estilo CamelCase. Ou seja, as primeiras letras de cada palavra que forma o identificador são representadas em maiúsculo. Outro estilo bastante utilizado por outras linguagens é a separação de cada palavra por meio de um *underline* (caractere de código 95 da tabela ASCII).

No contexto da nossa pesquisa, um identificador composto é caracterizado da seguinte forma:

1. por conter mais de uma palavra, por exemplo: $ProcessedByManager$;
2. por ser definido seguindo o estilo CamelCase ou ter as palavras separadas por *underline*.

Para o estudo desenvolvido e apresentado neste documento, cada identificador no código é extraído e processado da seguinte forma: primeiro, *tokens* são extraídos dos seus identificadores compostos; depois, em cada um é aplicada a remoção de *stop words* e *stemming*.

Para ilustrar o método de processamento, considere o exemplo hipotético de uma classe Java chamada $Utils$, apresentada no Código Fonte 2.1. Ela contém um atributo identificado como $dataBaseConnection$ e três métodos identificados por: $setDataBaseConnection$, $getDbConnection$ e $isDataBaseConnected$.

Por exemplo, ao processarmos o identificador do método $isDataBaseConnected$, primeiros dividimos os seus quatro tokens e os normalizamos para ficarem apenas com letras minúsculas: is, $data$, $base$ e $connected$. Depois, removemos a *stop word* is e aplicamos *stemming* para a remoção do sufixo *ed* do token $connected$. Por fim, o conjunto resultante do processamento é: $\{data, base, connect\}$.

Código Fonte 2.1: Exemplo de Classe Java

```
public class Utils {
  private int dataBaseConnection;
  public setDataBaseConnection(int db){ }
  public int getDbConnection(){ }
  public boolean isDataBaseConnected(){ }
}
```

É importante notar que cada termo extraído pode ser tanto distinto de outros termos extraídos previamente ou idêntico a qualquer um deles. Termos idênticos contêm exatamente a mesma sequência de caracteres, depois de processados, independentes dos seus identificadores originais.

No caso da classe $Utils$ (Código Fonte 2.1), os termos $utils$, set e get aparecem apenas uma vez; db, duas vezes; $data$ e $base$, três vezes cada; e $connect$, quatro vezes. Desse modo, apesar do vocabulário da classe $Utils$ ser composto por 7 termos distintos, o número total de termos é 15. Para os algoritmos de recuperação da informação que aplicamos, são considerados apenas os termos distintos.

Capítulo 3

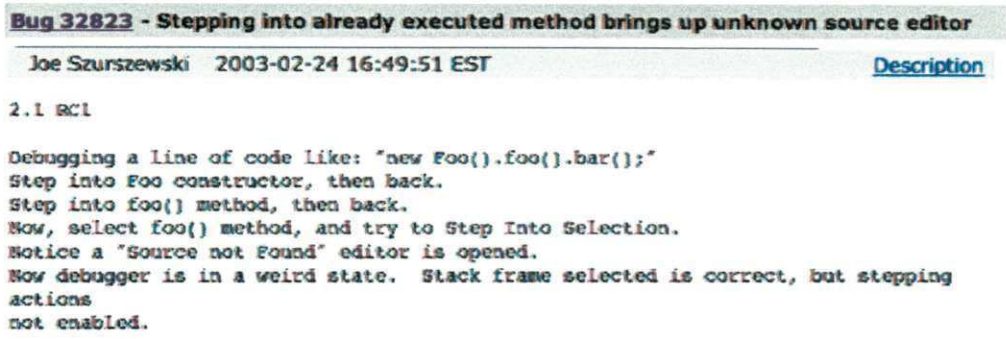
Uso de Vocabulários para Localização de Defeitos

Um dos fatores de qualidade de um software é a sua legibilidade, que pode ser medida, dentre outros aspectos, pela qualidade dos nomes escolhidos pelos desenvolvedores para os seus identificadores. Sabe-se que se a escolha de tais nomes for ruim, isso torna-se uma barreira à compreensão do software [16]. Por isso, as linguagens de programação modernas permitem aos desenvolvedores escolherem identificadores com nomes representativos. Sendo assim, não há justificativa para que os desenvolvedores se esquivem dessa prática de criar bons nomes de identificadores em seu código.

De fato, nos últimos anos, observou-se uma melhoria na qualidade de identificadores em código fonte. Enquanto que, em sistemas legados, era possível encontrar procedimentos e dados nomeados arbitrariamente, muitas vezes com nomes de namoradas ou jogadores favoritos dos programadores [17]; hoje em dia, os nomes de identificadores são escolhidos, em sua maioria, referentes ao domínio do problema do qual o software faz parte [18]. Assim, vocabulário de software passou a ser tema de interesse de pesquisadores da engenharia de software, já que, a partir dele, é possível a extração de informação relevante sobre os sistemas.

Outra boa fonte de informação sobre os domínios do problema das aplicações é o repositório de relatórios de defeitos, que muitos projetos utilizam. Isso ocorre porque os relatórios são escritos para descrever defeitos referentes às funcionalidades do software. Assim, é esperado que a maioria dos relatórios no repositório contenham termos relevantes ao sistema.

Figura 3.1: Exemplo de Relatório de Defeito do Projeto Eclipse 2.1



Fonte: Repositório aberto de relatórios de defeitos do projeto Eclipse.

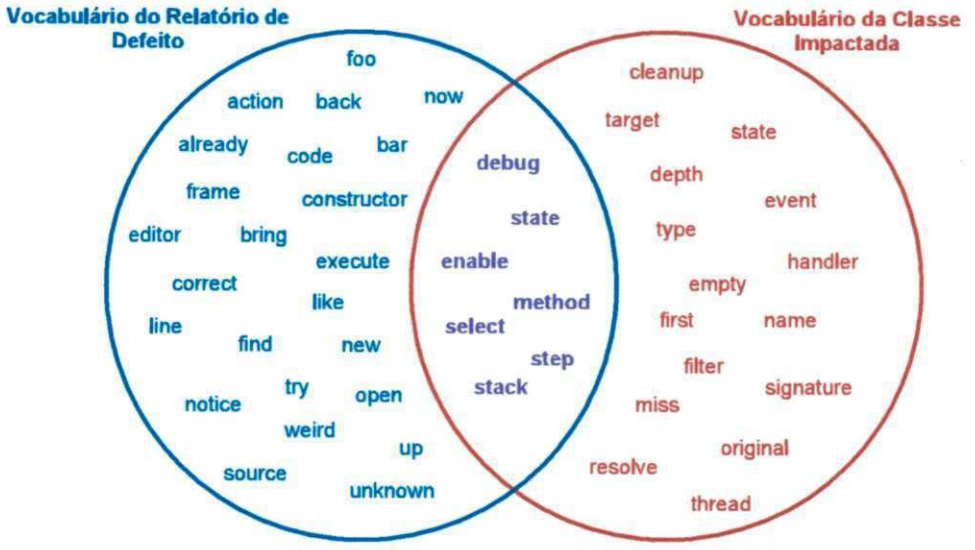
O fato de que desenvolvedores, ao lerem relatórios de defeitos, investigam e encontram no código os defeitos relatados, nos reforça a ideia de que essa similaridade entre vocabulários de relatórios de defeitos e código fonte esteja realmente presente na prática, na maioria dos relatórios existentes nos repositórios. De fato, tais relatórios são utilizados largamente para gerenciar o processo de correção de defeitos em diversos projetos de software.

Diante do exposto, dado que os nomes dos identificadores, que formam o vocabulário do software, são escolhidos geralmente de acordo com o domínio do problema, e que os textos presentes nos relatórios de defeitos também geralmente estão de acordo com o mesmo domínio do problema, então consideramos que é possível relacionar ambos os vocabulários extraídos dessas duas fontes para auxiliar desenvolvedores no processo de localização de defeitos no código. De fato, nos últimos anos, pesquisas têm apontado a importância do vocabulário em atividades de manutenção de sistemas [19–21].

Para ilustrar um relatório de defeito cujo vocabulário é similar ao da classe impactada, destacamos o relatório com ID 32823[1], do projeto Eclipse 2.1. Na Figura 3.1, é possível verificar o texto presente no relatório destacado: o título e a descrição do relatório descrevem um defeito que ocorre quando o usuário, em modo de depuração, investiga um método que já foi executado, trazendo à tona um editor específico para código desconhecido. A classe que foi modificada para a correção desse defeito chama-se $StepIntoSelectionHandler$.

[1]Relatório de defeito acessível via `https://bugs.eclipse.org/bugs/show_bug.cgi?id=32823`, verificado em outubro de 2012.

Figura 3.2: Exemplo de Interseção de Vocabulários de Relatório de Defeito e Classe



A Figura 3.2 apresenta um diagrama de Venn contendo o conjunto de termos dos vocabulários extraídos do relatório de defeito e da classe impactada para sua correção. À esquerda do diagrama (em azul), estão representados todos os termos extraídos do relatório e que não estão presentes no vocabulário da classe. Já à direita (em vermelho), há todos os termos da classe que não fazem parte do vocabulário do relatório. Por fim, no meio (em roxo), encontra-se a interseção de ambos os vocabulários. Como é possível perceber, por mais que o relatório de defeitos não tenha uma referência direta à classe a ser modificada, a descrição do defeito contém termos que são similares aos do vocabulário da classe. Assim, ao relacionarmos ambos os vocabulários, a similaridade será alta. De fato, ao analisarmos esse relatório de defeito com a abordagem proposta em nossa pesquisa, a classe $StepIntoSelectionHandler$ foi listada como a mais provável de ser impactada, dentre todas as mais de 7 mil classes do projeto Eclipse 2.1.

Abordagem proposta

Para o desenvolvimento do estudo apresentado neste trabalho, propomos uma abordagem baseada exclusivamente no uso de vocabulários de software e de relatórios de defeitos. Assim, foi possível avaliar o potencial de tais vocabulários em auxiliar a tarefa de sugestão de

classes possivelmente impactadas por relatórios de defeitos. Entretanto, entendemos que, na prática, pode ser mais eficaz utilizar tal abordagem em conjunto com outras técnicas, a fim de melhorar os resultados.

A abordagem proposta é composta dos cinco passos descritos abaixo e ilustrados na Figura 3.3.

Passo 1 - Extração e Tratamento do Vocabulário do Software

Inicialmente, o vocabulário do código fonte é extraído e tratado com técnicas de recuperação da informação, tais como: tokenização, remoção de *stop words* e *stemming*.

Para a extração, utilizamos uma ferramenta desenvolvida em nosso laboratório, chamada *VocabularyTools*[2]. Mais especificamente, esta ferramenta processa todas as classes de um projeto Java, constrói sua Árvore Sintática Abstrata e extrai seu vocabulário (nomes de classes, interfaces, métodos, atributos e constantes). Finalmente, os dados extraídos são armazenados em um arquivo num formato XML que mantém a hierarquia de identificadores do código[3]. Já para o tratamento do vocabulário, utilizamos as técnicas de recuperação da informação apresentadas na Seção 2.2.

Ao final desta etapa, temos um corpus de termos extraídos e tratados referentes ao Vocabulário do Software, agrupados por cada documento. Em nosso caso, os documentos são as classes Java analisadas e o corpus refere-se aos seus termos.

Passo 2 - Indexação de Termos

Os termos extraídos e tratados, provenientes do vocabulário do software são, então, indexados. Para esta tarefa, utilizamos a ferramenta Apache Lucene[4], que é um software de indexação de documentos e busca de termos, mantido pela fundação Apache.

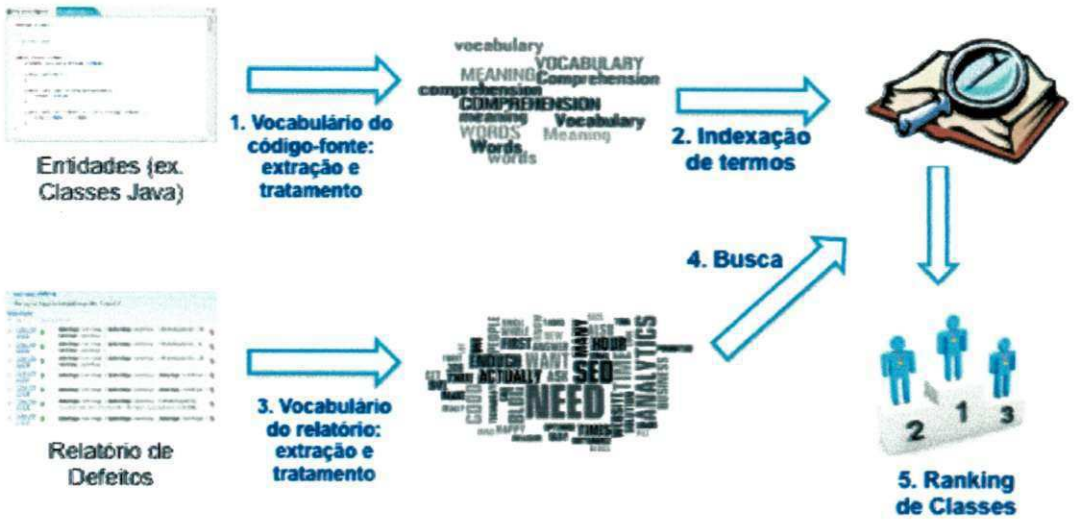
O índice criado armazena dados estatísticos sobre os termos, de modo que uma pesquisa baseada em termos seja feita de forma eficiente. Assim, indexar os termos nos permite buscar eficientemente classes que contenham termos específicos.

[2]Ferramenta disponibilizada juntamente com os dados desse estudo em `http://code.google.com/p/splab-br-analysis/`.

[3]Mais informações sobre como representamos vocabulário em XML podem ser encontradas no Apêndice C.

[4]Ferramenta disponível em `http://lucene.apache.org/`, verificado em outubro de 2012.

Figura 3.3: Abordagem Proposta



É importante notar que tanto o passo anterior quanto o atual são atividades que necessitam ser feitas apenas uma única vez ou, de forma incremental, quando o software for atualizado. Ou seja, a cada atualização do software, pode-se extrair o vocabulário apenas do que foi modificado e acrescentá-lo ao índice que foi criado previamente.

Passo 3 - Extração e Tratamento do Vocabulário do Relatório de Defeito

Para cada relatório de defeito que será analisado, seu vocabulário é extraído e tratado exatamente da mesma forma que é feito no Passo 1. Entretanto, não indexamos o vocabulário dos relatórios de defeitos. Dessa forma, cada relatório é processado de forma independente dos demais. Assim, não dependemos de dados históricos presentes no repositório de defeitos.

Ao final desta etapa, temos um corpus de termos provenientes do vocabulário do relatório de defeito, normalizado da mesma forma que o corpus indexado do vocabulário de software. Para este passo, desenvolvemos uma ferramenta que realiza a extração e o tratamento dos vocabulários dos relatórios de defeitos[5].

Passo 4 - Busca

A partir da extração dos termos do vocabulário do relatório de defeito, é criada uma *query* de busca contendo todos os termos do vocabulário (exceto as *stop words*) unificados com o

[5]Ferramenta também disponibilizada em `http://code.google.com/p/splab-br-analysis/`.

operador lógico OR. Ou seja, a busca será feita nos documentos indexados (no nosso caso, as classes), retornando aqueles que possuem, pelo menos, um dos termos contidos na busca.

O uso do operador lógico AND foi descartado pois, caso o utilizássemos, só seriam retornados documentos que contivessem exatamente todos os termos presentes no relatório de defeito analisado. Como estamos comparando um universo de texto em linguagem natural (proveniente dos relatórios de defeitos) com um universo de texto mais estruturado (identificadores presentes no vocabulário de software), então é extremamente improvável que os termos extraídos de uma dada entidade do código contenha todos os termos presentes no vocabulário do relatório de defeito. Por outro lado, com a utilização do operador OR, retornamos os documentos que tenham, pelo menos, alguns dos termos presentes no vocabulário do relatório. Sendo assim, aqueles termos mais relacionados ao domínio do problema se sobressaem em relação aos outros, que são, provavelmente, provenientes do discurso da linguagem natural.

Passo 5 - Ranking de Classes

Por fim, cada classe Java retornada na busca é ordenada numa lista de ranking segundo a sua similaridade com os termos da *query* de busca utilizada no passo anterior. Classes mais similares são retornadas nas primeiras posições do ranking.

Para calcular a similaridade de cada classe do corpus do vocabulário de software, utilizamos a implementação do algoritmo de Modelo de Espaço Vetorial (em inglês, VSM), disponível na ferramenta Apache Lucene e descrito na Seção 2.2.2. Com este modelo, podemos identificar quão similar cada classe é ao vocabulário do relatório de defeito investigado (a *query* de busca).

É importante notar que outros modelos também podem ser utilizados neste passo para o cálculo da similaridade. Neste trabalho, entretanto, focamos apenas no uso do VSM, pois ele é o mais utilizado na área de Recuperação da Informação.

Capítulo 4

Avaliação

Conforme descrito no Capítulo 1, o objetivo principal desta pesquisa é avaliar o uso de vocabulário de software no processo de localização de defeitos descritos de forma textual em relatórios.

Diante disso, desenvolvemos um estudo empírico para avaliar, por meio de técnicas de recuperação da informação, o uso dos vocabulários extraídos do software e dos relatórios de defeitos, utilizando-os como fatores de auxílio a desenvolvedores que necessitem localizar defeitos. A partir da extração, tratamento e análise dos vocabulários, pudemos ordenar as classes pela similaridade calculada em relação ao texto presente no relatório de defeito analisado.

Neste capítulo, serão apresentados o planejamento do estudo supracitado (Seção 4.1), os resultados obtidos (Seção 4.2) e uma análise sobre as limitações do estudo (Seção 4.3).

4.1 Planejamento do Estudo

Esta seção apresenta o planejamento do estudo que foi desenvolvido como forma de avaliação do uso dos vocabulários de software e de relatórios de defeitos para a localização de classes defeituosas no código.

Iniciamos a seção apresentando a metodologia (Seção 4.1.1) utilizada para realizar o estudo. Além disso, listamos os projetos que foram analisados (Seção 4.1.2) e como eles foram coletados e instrumentados (Seção 4.1.3). Por fim, na Seção 4.1.4, estão descritas as questões de pesquisa que nortearam este estudo, enquanto que na Seção 4.1.5, é explicada a

métrica analisada.

4.1.1 Metodologia

Para a presente avaliação ser conduzida, obtivemos dados de relatórios de defeitos de diferentes projetos, aplicamos a abordagem proposta (detalhada no Capítulo 3) e verificamos a nossa taxa de acerto, que foi calculada a partir do número de classes que eram corretamente retornadas pelo ranking resultante da nossa abordagem.

Para tanto, a partir da extração dos dados contidos nos repositórios de defeitos e código fonte dos projetos, criamos um oráculo, refletindo os dados reais do projeto, contendo quais classes foram modificadas ao se corrigir cada um dos defeitos analisados. Assim, foi possível comparar as classes presentes no oráculo com aquelas presentes no ranking a ser avaliado.

4.1.2 Projetos Analisados

Durante a fase de escolha dos projetos que iriam compor a avaliação, listamos requisitos que precisavam ser atendidos, de modo que fosse possível estudar os relatórios de defeitos dos projetos escolhidos e, ainda, criar um oráculo a partir dos relatórios que já tivessem sido corrigidos.

Segue a lista dos requisitos que qualificaram os projetos para o nosso estudo:

1. **Utilizar sistema de rastreamento de defeitos com banco de dados acessível.** Isso era essencial, já que teríamos que fazer download de todos os dados dos relatórios de defeitos para serem utilizados como entrada para nossa abordagem;

2. **Ter versões em produção.** Os projetos precisavam ter versões já em produção e sendo utilizadas por usuários, de modo que defeitos do sistema já pudessem ter sido encontrados, reportados e corrigidos;

3. **Utilizar repositório de código fonte acessível.** Um repositório de código fonte cujo código fosse disponibilizado, em suas diversas versões, também era necessário. Assim, poderíamos fazer download da versão defeituosa do sistema, bem como analisar o histórico de *commits* para relacionar quais modificações do código referiam-se aos relatórios de defeitos corrigidos;

4. **Seguir diretrizes de *commits* para correção de defeitos.** Também era necessário que os desenvolvedores do projeto seguissem diretrizes para envio de código referente à correção de defeitos reportados. Por exemplo, eles poderiam informar na descrição do *commit* qual defeito está sendo corrigido, por meio do id do seu relatório (ex. "Fix LUCENE-1234");

Os projetos que identificamos que atendiam aos requisitos supracitados e que foram escolhidos para fazer parte da avaliação foram: três versões do Projeto Eclipse e cinco projetos distintos da organização Apache. A Tabela 4.1 apresenta informações sobre os mesmos: uma breve descrição do projeto, as versões que foram analisadas, número de classes (excluindo classes de testes), número de relatórios de defeitos analisados e períodos nos quais os defeitos analisados foram marcados como resolvidos no repositório de defeitos.

Utilizamos projetos de diversos tamanhos, variando de 350 a mais de 10 mil classes. Além disso, ao todo, analisamos mais de 9,5 mil relatórios de defeitos. Isso é mais do que foi estudado por qualquer trabalho relacionado ao nosso, dos quais tivemos conhecimento [3, 6–8, 10, 22–26]. Por fim, para a grande maioria dos projetos analisados, coletamos dados de relatórios de defeitos que foram resolvidos num período de um ano.

4.1.3 Coleta e Instrumentação das Amostras

Conforme descrito na seção anterior, os projetos, para serem utilizados neste estudo, devem ser de código aberto e possuir um repositório acessível de relatórios de defeitos. Portanto, para se coletar suas amostras, basta fazer o download do código referente à versão defeituosa que será analisada, bem como um conjunto de relatórios de defeitos que estejam marcados como resolvidos e referentes às versões estudadas. Por fim, cada relatório de defeito deve ser mapeado com o conjunto de classes que foram modificadas para a sua resolução.

Abaixo, descrevemos o processo de coleta e instrumentação das amostras utilizadas neste experimento.

Download do Código Fonte

Como os projetos eram de código aberto e todos utilizavam repositório de código, tais como CVS ou SVN, baixar o seu código fonte foi uma tarefa simples.

Tabela 4.1: Projetos Utilizados no Estudo

Nome do Projeto	Descrição	Versão	Nº. Classes	Nº. Relatórios Analisados	Resolução dos Defeitos
Apache Hadoop	*Framework* de suporte a processamento distribuído.	0.23.0	535	27	Nov 2010 - Nov 2011
Apache Lucene	Biblioteca de suporte a recuperação da informação.	3.5.0	501	14	Set 2011 - Dez 2011
Apache OpenJPA	Biblioteca de suporte a persistência em Java.	2.1.0	1.226	90	Mar 2010 - Abr 2011
Apache Pivot	Plataforma de desenvolvimento para aplicações online.	2.0.1	566	31	Mar 2011 - Dez 2011
Apache Shiro	*Framework* de segurança para Java.	1.2.0	350	16	Jan 2011 - Jan 2012
Eclipse	Plataforma de desenvolvimento.	2.0	6.413	2.780	Dez 2001 - Dez 2002
		2.1	7.566	2.480	Out 2002 - Set 2003
		3.0	10.331	4.136	Dez 2003 - Dez 2004

Todos os projetos com os quais trabalhamos disponibilizam em suas páginas na Internet um arquivo compactado contendo o código fonte referente a cada versão lançada. Assim, após identificada qual era a versão desejada, bastou fazer seu download na página do projeto e descompactá-la.

Para a escolha da versão, procuramos aquelas que tivessem maior número de relatórios de defeitos resolvidos. Além disso, os *commits* de código no repositório referentes ao período de lançamento e manutenção da versão, deveriam estar acessíveis.

Acesso às Informações dos Repositórios de Código

Além do código fonte, também era necessário que tivéssemos acesso às informações do repositório de código dos projetos. Mais especificamente, precisamos ter uma lista dos *commits* realizados no repositório, suas descrições e quais classes foram modificadas em cada um.

No caso das versões do Eclipse, utilizamos um conjunto de dados sobre defeitos disponibilizado por Zimmermann et al. [27]. Eles extraíram os dados a partir da mineração dos repositórios e os disponibilizaram publicamente[1] para efeitos de pesquisa. Nesse caso, apenas construímos um *parser* para processar os arquivos XML disponibilizados pelos autores.

Desse modo, não precisamos processar diretamente as informações do repositório do Eclipse. Entretanto, para os projetos da organização Apache, buscamos na Internet o endereço do repositório de código que o projeto utiliza e fizemos o download a partir dele. Neste caso, utilizamos o comando exemplificado abaixo para fazer o download, em formato XML, das informações de *commit* a partir do repositório:

```
svn log http://svn.apache.org/repos/asf/openjpa/trunk \
  -r {2010-03-25}:{2011-04-28} --verbose --xml
```

O comando apresentado faz o download, a partir do repositório do projeto OpenJPA, de todos os *commits* de código entre os dias 25/03/2010 (inclusive) e 28/04/2011 (exclusivo).

Por fim, processamos os arquivos XML que foram baixados com um parser que implementamos para este fim.

[1] A URL para acesso aos dados é `www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/`, verificada em outubro de 2012.

Download de Relatórios de Defeitos

Os sistemas de rastreamento de defeitos mais utilizados, tais como Bugzilla ou JIRA, disponibilizam seus relatórios de defeitos em uma forma estruturada, tal como em um arquivo XML. Dessa forma, fazer o download dos relatórios de defeitos compreendeu filtrar os relatórios de defeitos desejados (aqueles resolvidos e referentes às versões que seriam estudadas) e baixá-los.

Para o caso do projeto Eclipse, que utiliza o Bugzilla, fizemos o download por meio do script contido no Apêndice A. Ele se conecta ao repositório do Bugzilla do Eclipse e baixa a versão XML dos relatórios de defeitos desejados.

No caso dos projetos da organização Apache, que utilizam o JIRA, acessamos a interface web de seus repositórios[2] e filtramos os relatórios de defeitos conforme as regras definidas anteriormente.

Se tomarmos como exemplo o projeto Apache OpenJPA, acessamos a URL: `https://issues.apache.org/jira/browse/OpenJPA` e filtramos os relatórios cadastrados no banco de dados, utilizando a seguinte consulta:

```
project = OPENJPA AND issuetype = Bug
  AND fixVersion = ''2.1.0'' AND resolution = Fixed
```

Isso indica que queremos apenas relatórios que contenham descrições de defeitos do projeto OpenJPA, da versão 2.1.0, e que já tivessem sido resolvidos.

Após o filtro, a ferramenta JIRA nos dá a opção de fazer o download de todo o conjunto retornado num formato XML.

Mapeamento de Relatórios de Defeitos para o Código Modificado para a sua Resolução

Mapear cada relatório de defeito às classes que foram impactadas para a sua correção não é uma atividade trivial, já que os relatórios de defeitos – tanto do Bugzilla quanto do JIRA – geralmente não armazenam esse tipo de informação. Sendo assim, precisamos recorrer a trabalhos que fizeram atividades semelhantes para construirmos a nossa própria ferramenta de mapeamento. Ou seja, construirmos o nosso oráculo.

[2] Acessível via uma URL no formato: `https://issues.apache.org/jira/browse/NOME_DO_PROJETO`.

Alguns autores [4, 5, 27] têm feito esse tipo de mapeamento ao minerar repositórios de software, para fins distintos. Basicamente, eles buscam por referências aos IDs dos relatórios de defeitos nas descrições de *commits*. Por exemplo, se numa descrição de *commit*, aparecer descrições como: "Fixed 42233" ou "Bug #53784", os arquivos impactados por esse *commit* são relacionados aos defeitos com ID 42233 e 53784, respectivamente.

Para o caso do projeto Eclipse, nós já tínhamos essa informação mapeada por Zimmermann et al [27]. Já no caso dos projetos da Apache, eles têm uma diretriz que diz que cada *commit* deve começar com o ID do relatório de defeito ao qual ele se refere. Assim, temos nos arquivos dos projetos da Apache, *commits* seguindo o estilo: "*OPENJPA-1550: Set failedObject on RollbackException. Submitted By: Heath Thomann*".

Desse modo, processamos todos os dados baixados, provenientes dos repositórios de código e de defeitos, e mapeamos os relatórios de defeitos às classes Java que foram impactadas para corrigi-los (o oráculo). Aqueles relatórios que não tinham nenhum *commit* no repositório de código referente a eles tiveram de ser desconsiderados, dado que não teríamos informação suficiente para criarmos um oráculo sobre ele.

4.1.4 Questões de Pesquisa

As questões de pesquisa (QP) que buscamos responder com este estudo foram as seguintes:

QP_1: Em geral, quantas classes cada relatório de defeito impacta?

Esta é uma importante questão que emerge de nosso estudo, pois precisamos saber, na prática, quantas classes geralmente os programadores modificam para corrigir um dado defeito.

O conhecimento deste valor pode nos guiar a entender a distribuição do conjunto de impacto típico dos projetos analisados e, consequentemente, o tamanho esperado de uma lista de sugestão de classes possivelmente a serem impactadas.

Para responder a esta pergunta, dado que já temos o mapeamento dos relatórios de defeitos para as classes que foram impactadas para a sua correção, bastou que analisássemos o tamanho do conjunto de impacto de cada relatório.

QP_2: Os defeitos dos sistemas analisados se concentram em apenas um conjunto pequeno de classes?

Ao investigar a resposta da questão de pesquisa anterior, estaremos focando na questão ***quantas** classes são impactadas para cada relatório de defeito?* Entretanto, também temos interesse em avaliar como os defeitos se distribuem por entre as classes que foram modificadas para corrigi-los. Ou seja, queremos agora focar em ***quais** classes são impactadas pelos relatórios de defeitos?*

Esse dado é importante de se averiguar porque, caso haja um conjunto de classes nas quais os defeitos se concentrem, uma técnica que sempre retorne esse conjunto de classes teria uma alta taxa de acerto e, portanto, o desenvolvimento de abordagens diversas para resolver este problema poderia ser considerado como um esforço em vão. Esse caso deve ser comum em sistemas mal estruturados, que contenham classes sem coesão (as chamadas "*God Classes*" [28]), que possam ser o foco principal de defeitos.

Em busca da resposta à esta pergunta, analisamos, para cada classe, quantos relatórios de defeitos as impactam.

QP_3: Qual é a porção do conjunto de classes impactadas que é coberta por um ranking contendo N elementos?

Esta pergunta refere-se à quantidade total de classes impactadas que sugerimos corretamente para um ranking com N elementos, sendo N um número variável. A resposta a esta pergunta nos indica qual é o tamanho da porção do software que precisamos analisar, de modo que alcancemos uma boa taxa de acerto em relação às classes sugeridas num ranking.

Para calcular este valor, aplicamos a nossa abordagem aos oito projetos analisados e comparamos o ranking retornado com o conjunto real de impacto, fornecido pelo oráculo. Assim, é possível calcular a taxa de acerto e, consequentemente, analisar o quanto do software pudemos filtrar sem grandes perdas na resposta, para cada projeto.

QP_4: Qual é a menor posição do ranking na qual é possível encontrar uma classe impactada?

A maioria dos trabalhos relacionados dos quais tivemos conhecimento, calculam suas taxas de acerto de acordo com a primeira entidade corretamente retornada no ranking. Em outras palavras, a taxa de acerto não é calculada considerando todas as classes que foram impactadas por um relatório de defeito, tal como fizemos no cálculo da QP_3. Por outro lado, ela é calculada pela análise apenas da primeira posição do ranking na qual uma classe impactada é corretamente retornada.

Esse cálculo é válido, já que a primeira classe corretamente retornada no ranking significa que 1) ela é uma classe defeituosa; e 2) ela tem o vocabulário mais similar ao do relatório de defeitos, quando comparada às outras classes defeituosas. Assim, podemos imaginar que as classes com vocabulário menos similar mas que também foram impactadas, são aquelas classes modificadas por consequência, após se corrigir o defeito na fonte do problema. Como nosso oráculo é extraído a partir do *commit* único de todas as classes para a correção do problema, não temos como diferenciar a fonte real do defeito das classes impactadas por consequência.

De fato, trabalhos da área de análise de impacto [29–31] mostram que, dada uma classe e uma mudança a ser feita, é possível prever como essa mudança se propagará na rede de dependências de um software. Desse modo, se retornarmos um ranking com menos classes, mas contendo, pelo menos, uma classe impactada, é possível para o desenvolvedor identificar o defeito e começar a corrigi-lo a partir dela.

Então, para responder a esta pergunta, nós fazemos uma análise semelhante à da QP_3, entretanto, apenas consideramos a posição da primeira classe corretamente sugerida no ranking.

QP_5: A análise de comentários de código impacta na melhoria da abordagem?

Em seu trabalho, Haiduc e Marcus [14] citam que ferramentas de suporte à compreensão de software que processam vocabulário de software tendem a ignorar comentários presentes no código fonte e usam apenas identificadores. Muitos dos autores que procedem dessa forma afirmam que os comentários geralmente não estão atualizados e sincronizados com o código

fonte. Entretanto, baseados na sua pesquisa, Haiduc e Marcus defendem a ideia que comentários podem ser mais significativos para o vocabulário do software do que identificadores, refletindo assim o domínio do problema.

Diante dos resultados do trabalho supracitado, não consideramos os comentários *Javadoc* presentes no código como parte integrante do vocabulário do software. As respostas das duas questões de pesquisa anteriores se baseiam nessa metodologia, sem considerarmos comentários. Entretanto, é válido nos questionarmos se os comentários auxiliam ou não na melhoria da nossa abordagem.

Para responder a esta pergunta, aplicamos novamente a nossa abordagem aos oito projetos analisados. Entretanto, dessa vez, ao extrairmos o vocabulário dos sistemas, consideramos os comentários presentes no código fonte. Assim, pudemos comparar os resultados previamente alcançados (sem considerar comentários) com os atuais (considerando comentários).

4.1.5 Métrica Analisada

A métrica que utilizamos para avaliar os nossos resultados foi a **média de cobertura**, que é derivada da cobertura (em inglês, *recall*) calculada para cada ranking, referentes aos relatórios de defeitos. A cobertura é uma métrica bastante utilizada em experimentos da área de Recuperação da Informação. Basicamente, ela representa a proporção de documentos relevantes recuperados, dentre os documentos relevantes existentes na base de dados (em nosso caso, no oráculo). A fórmula a seguir descreve o seu cálculo:

$$cobertura = \frac{|\{classes\ impactadas\} \cap \{classes\ no\ ranking\}|}{|\{classes\ impactadas\}|}$$

$$media\ da\ cobertura = \frac{\sum_{i=1}^{n} cobertura(i)}{n}$$

onde n é o número total de relatórios de defeitos analisados.

Para melhor entendimento, considere o seguinte exemplo:

- O conjunto de impacto de um relatório de defeito com ID $B1$ é $\{C1, C2, C3\}$;

- O conjunto de impacto de um relatório de defeito com ID $B2$ é $\{C1, C4\}$;

Considere que nossa abordagem retornou como um ranking Top 5 de classes para o relatório de defeito com ID $B1$ a seguinte lista: <**C2**, C5, C4, **C3**, C7>; e para o relatório de defeitos com ID $B2$, a seguinte: <C5, C8, **C1**, C9, **C4**>.

Então, a **cobertura** calculada para o *relatório de defeitos B1* é $0,67$ $(2/3)$ e para o *relatório de defeitos B2* é $1,0$ $(2/2)$. Consequentemente, a **média de cobertura** para ambos os rankings é de $(0,67 + 1,0)/2 = 0,83$ (ou 83%).

É importante citar que outra métrica geralmente utilizada juntamente com cobertura é a precisão, que avalia o número de documentos corretamente retornados numa lista. Entretanto, para o caso de listas com tamanho fixo, a precisão não é significativa. Especificamente para a nossa abordagem, a precisão representaria o número de classes corretamente retornadas no ranking. Entretanto, considere como exemplo um ranking sempre contendo 10 elementos, independente do número de classes que foi realmente impactado. Além disso, considere um relatório de defeito analisado, cujo impacto foi de apenas 1 classe. Assim, se utilizarmos precisão como métrica de avaliação, veremos que o melhor resultado possível de se obter seria $1/10$ (ou 10%), mesmo que a cobertura fosse 100%. Ou seja, a precisão seria muito baixa, mesmo que a abordagem tivesse retornado todas as classes impactadas, conforme era esperado. Portanto, precisão não é uma boa métrica para a abordagem proposta neste estudo, pois utilizamos rankings com tamanhos fixos. Assim, ficamos apenas com cobertura para a análise de nossos resultados.

4.2 Resultados e Análise

Nesta seção, apresentamos e discutimos os resultados para cada questão de pesquisa (QP) apresentada na Seção 4.1.4.

QP_1: Em geral, quantas classes cada relatório de defeito impacta?

Um dado importante que podemos extrair desse estudo é a quantidade de classes que são modificadas por um desenvolvedor para corrigir um dado defeito. Esse número nos leva a entender quantas classes, em média, os desenvolvedores dos projetos analisados precisam

descobrir, em meio a todas as classes do projeto, para que se possa corrigir o defeito.

Para responder a esta questão, extraímos os dados do mapeamento previamente feito entre relatórios de defeitos e classes impactadas e produzimos gráficos de setores (Figura 4.1), representando a proporção dos números de classes que foram modificadas para correção de defeitos nos relatórios que analisamos em nosso estudo.

Como é possível perceber, em média, 86% dos relatórios de defeitos impactam de uma a três classes. Para todos os projetos, acima de 78% de seus relatórios impactam até três classes.

Dessa forma, podemos dizer que, no geral, para os projetos estudados, um desenvolvedor tem que identificar para cada relatório de defeito menos de quatro classes a serem modificadas, dentre centenas ou milhares de classes presentes no projeto. Isso representa modificar menos de 1% do número total de classes dos projetos.

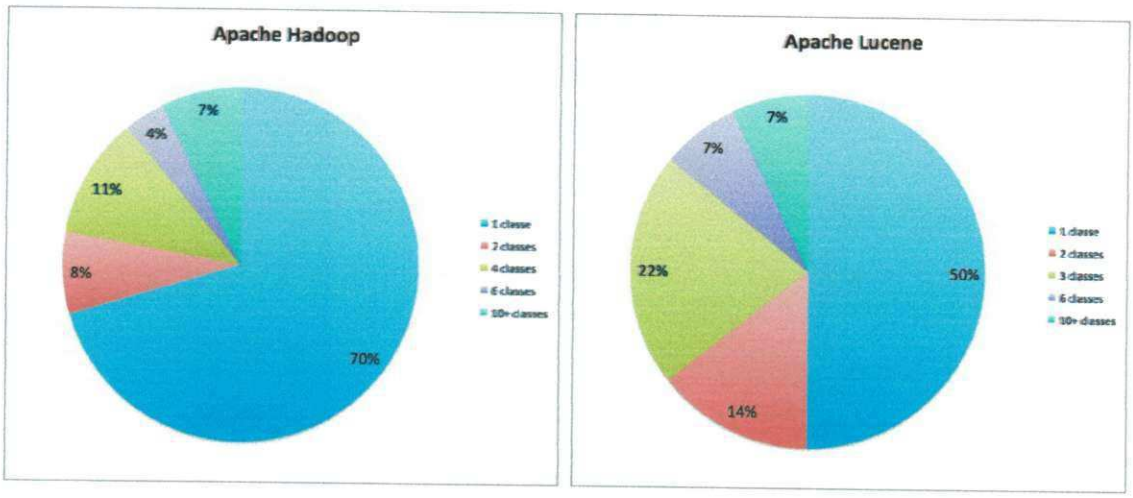
Esses dados nos reforçam a ideia de que técnicas efetivas para identificar o conjunto de impacto são importantes de serem desenvolvidas. Mesmo que a abordagem não seja útil para apontar todas as classes impactadas, ela pode ser considerada útil caso reduza o espaço de busca de classes num projeto, pois isso implica numa redução de esforço por parte do desenvolvedor para corrigir um defeito. Tal melhoria pode nos levar ao aumento do número de defeitos corrigidos por um desenvolvedor.

Vale a pena ressaltar que os dados apresentados referem-se aos relatórios de defeitos que nós processamos. Ou seja, são apenas relatórios contendo descrições de defeitos que foram corrigidos no período analisado. Caso analisássemos relatórios contendo requisição de funcionalidades novas ou refatoramentos, esses números se apresentariam de forma diferente, já que essas atividades, comumente, requerem mudanças em mais classes do projeto. Entretanto, nossa análise é válida para o nosso escopo, já que estamos analisando apenas relatórios identificados como sendo referentes a defeitos.

QP_2: Os defeitos dos sistemas analisados se concentram em apenas um conjunto pequeno de classes?

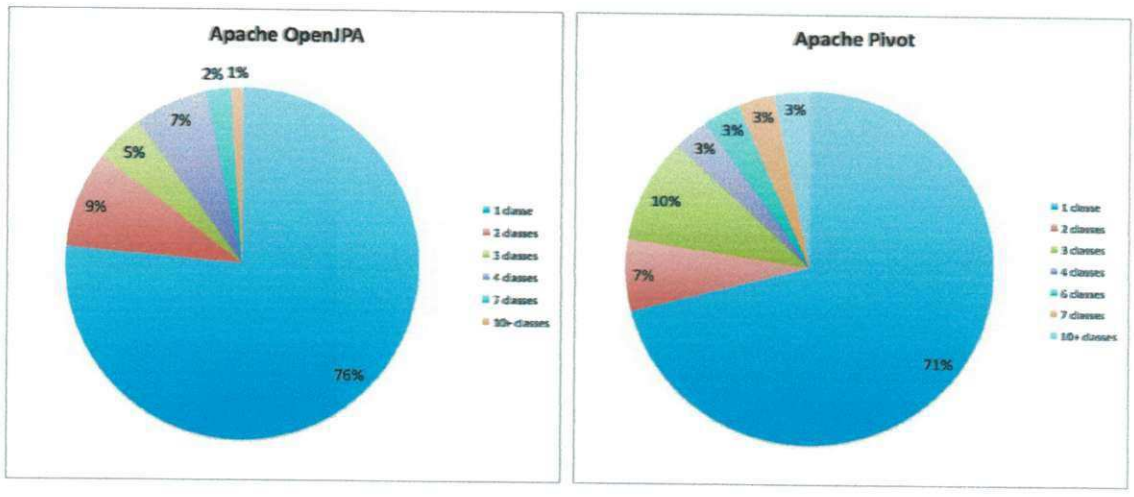
Vimos anteriormente que, geralmente, um número pequeno de classes (até três) é impactado para a correção de cada relatório de defeito. Entretanto, nada foi investigado em relação à di-

Figura 4.1: Proporção do Número de Classes Impactadas por Relatórios de Defeitos



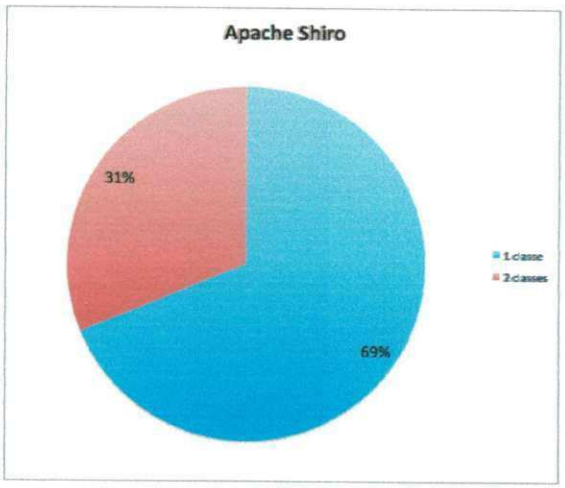
(a) Apache Hadoop

(b) Apache Lucene

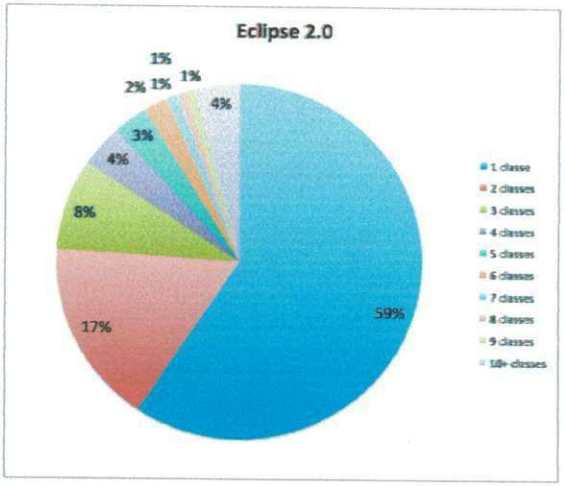


(c) Apache OpenJPA

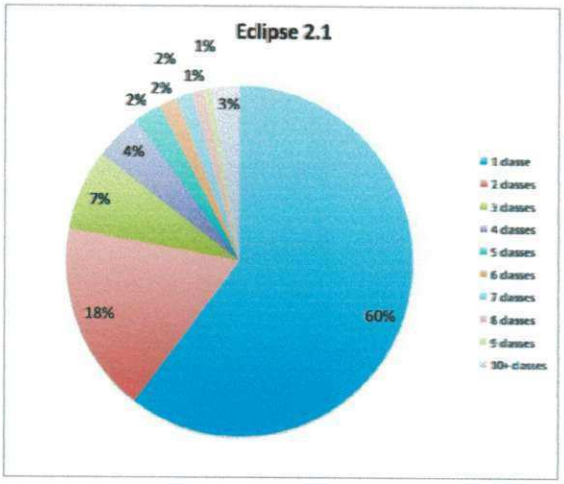
(d) Apache Pivot



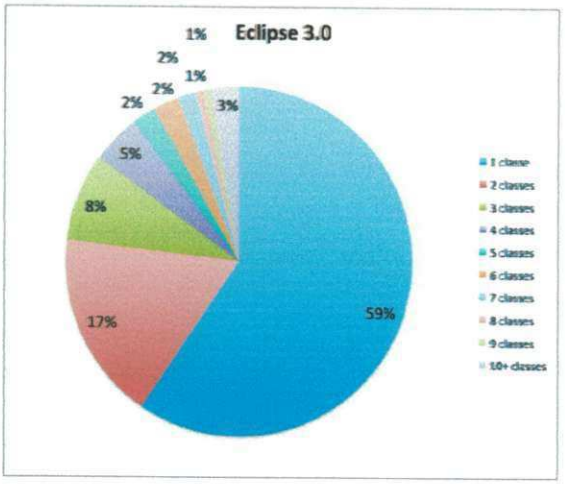
(e) Apache Shiro



(f) Eclipse 2.0



(g) Eclipe 2.1



(h) Eclipse 3.0

versidade dessas classes. Portanto, ao analisarmos a presente questão, visamos responder se há um conjunto específico de classes que são sempre impactadas para correção dos defeitos do projeto.

Para tanto, averiguamos, para cada classe do projeto, quantos relatórios de defeitos as impactaram para a sua resolução. A partir disso, produzimos gráficos de pontos nos quais cada relatório de defeito analisado é representado como uma linha no eixo y, enquanto cada classe é unicamente representada como uma coluna no eixo x. Para cada classe que foi impactada por um relatório de defeito, nós a representamos como um ponto. Assim, caso haja concentrações de pontos em apenas algumas poucas colunas do eixo x, teremos visualmente a indicação de que há, de fato, algumas poucas classes que concentram a maioria dos defeitos do projeto. Entretanto, vimos que isso não é verdade nos oito projeto que analisamos, conforme é possível perceber pela análise da Figura 4.2.

Considere, por exemplo, a Figura 4.2h, que representa o gráfico representando dados do projeto Eclipse 3.0. Como é possível perceber, várias classes diferentes são impactadas ao se resolver os 4.136 relatórios de defeitos analisados, já que diferentes classes (representadas como pontos) foram coloridas para a resolução dos diferentes relatórios de defeitos. O mesmo se aplica para os demais projetos.

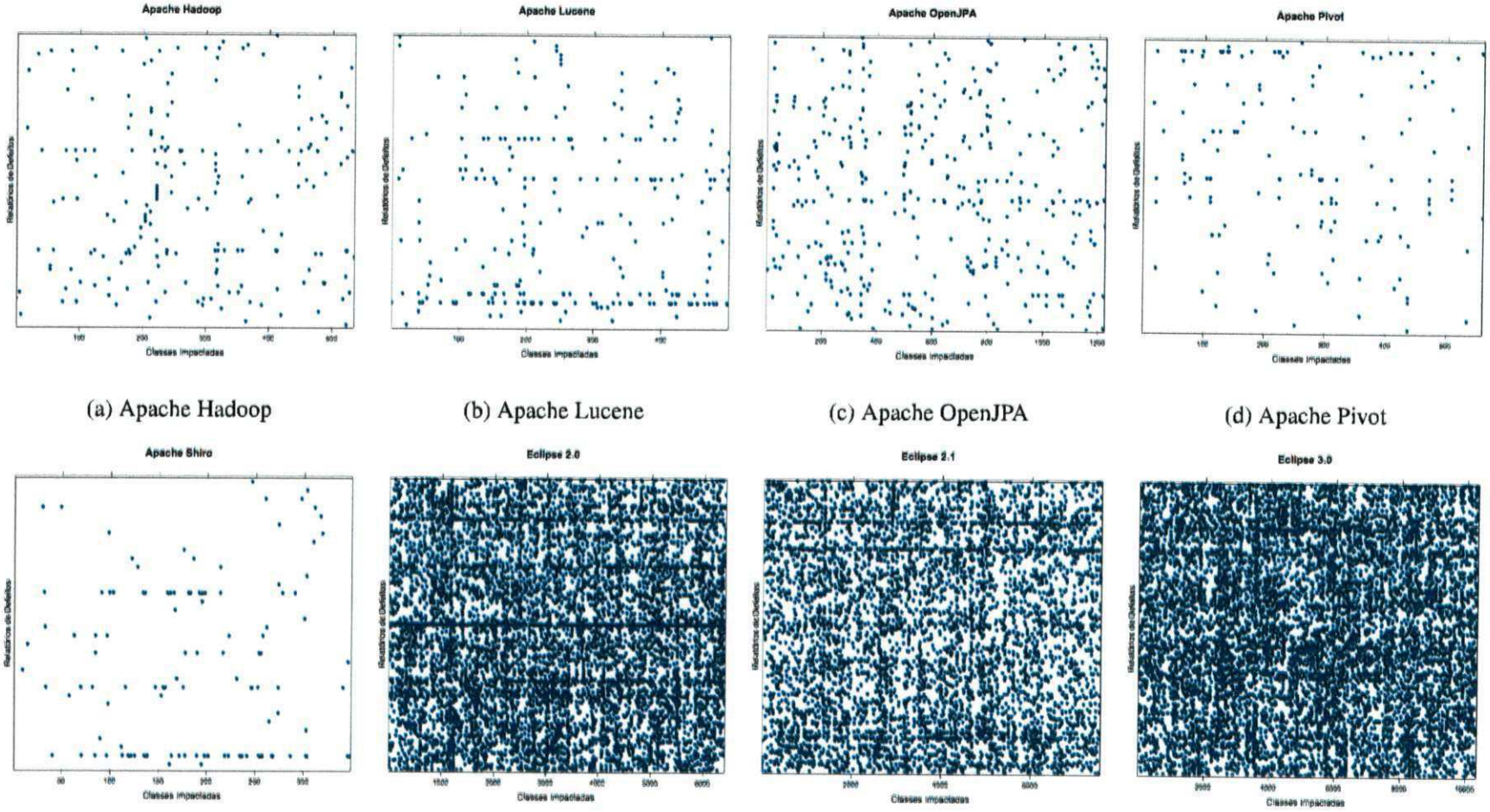
Esse resultado reforça a afirmativa de que a tarefa de encontrar quais classes que serão modificadas para resolução de um relatório de defeito, realmente não é trivial.

Por fim, vale ressaltar que também é possível perceber nas imagens da Figura 4.2 algumas concentrações, ainda que poucas, de pontos alinhados na horizontal. Eles representam relatórios de defeitos que impactam numerosas classes. Por exemplo, o projeto Eclipse 2.0 contém um relatório de defeito que impactou 327 classes distintas. Portanto, há na figura 4.2f uma linha horizontal com 327 pontos, representando este caso. Entretanto, como vimos na análise dos resultados da QP_1, tais casos são raros de ocorrer.

QP_3: Qual é a porção do conjunto de classes impactadas que é coberta por um ranking contendo N elementos?

Esta terceira questão de pesquisa refere-se ao número de classes impactadas que são retornadas corretamente em um ranking contendo N elementos, utilizando a abordagem proposta.

Figura 4.2: Classes Distintas Impactadas no Ranking



(a) Apache Hadoop (b) Apache Lucene (c) Apache OpenJPA (d) Apache Pivot

(e) Apache Shiro (f) Eclipse 2.0 (g) Eclipe 2.1 (h) Eclipse 3.0

Abaixo, apresentaremos gráficos que plotamos que nos dão suporte para responder a esta pergunta.

Gráficos de pontos

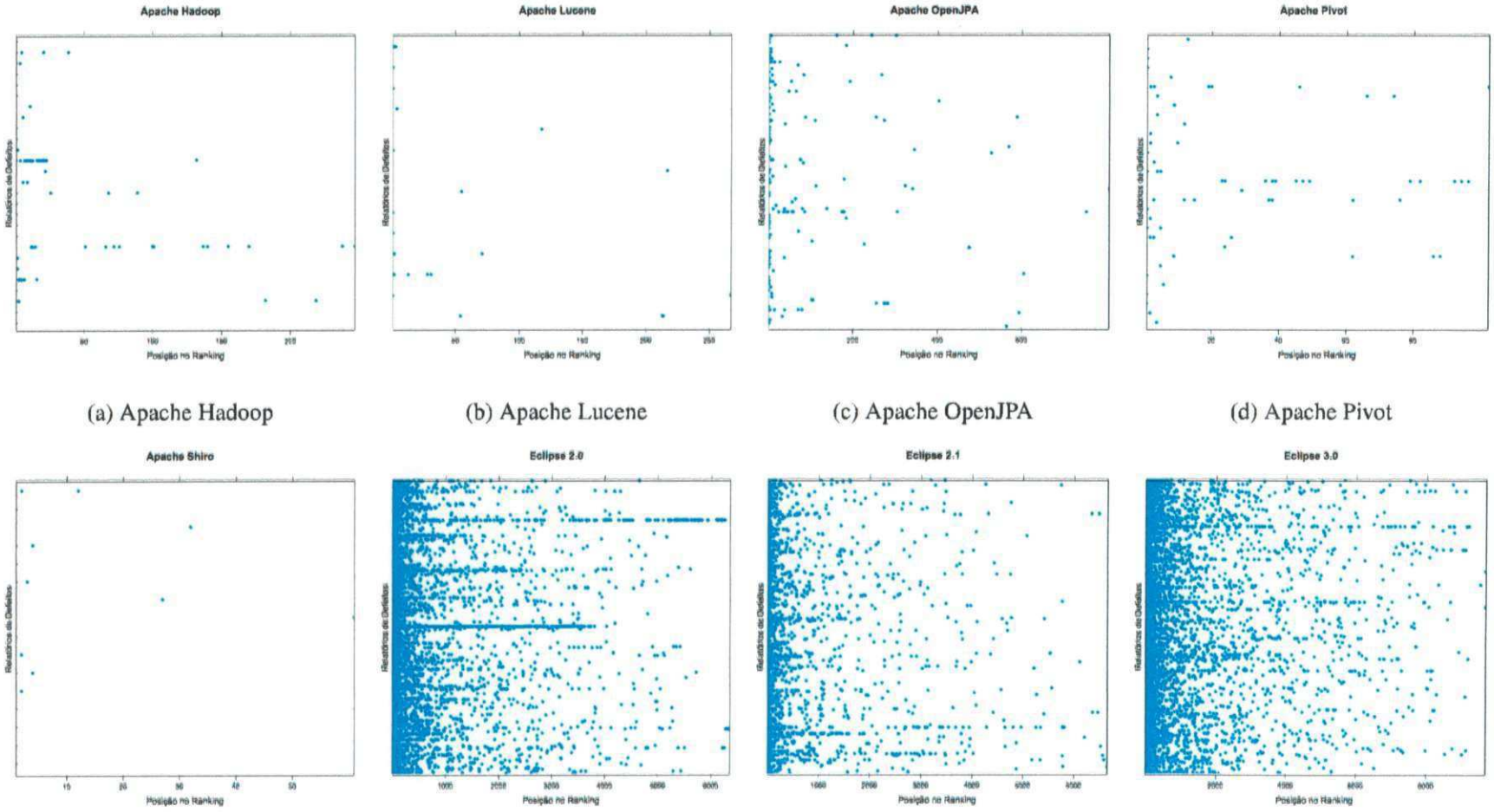
Inicialmente, plotamos outros gráficos de pontos (presentes na Figura 4.3) que demonstram como as classes impactadas são distribuídas no ranking, ao relacionarmos classes e relatórios de defeitos por meio de similaridade de vocabulários. Neles, o eixo y representa cada um dos relatórios de defeitos analisados, enquanto que o eixo x representa a lista de ranking em ordem decrescente de similaridade dos vocabulários. Classes impactadas são representadas como pontos no gráfico. Desse modo, os pontos mais à esquerda são as classes realmente impactadas que foram retornadas nas primeiras posições do ranking. Quanto mais à direita, menor a similaridade do vocabulário das classes com o dos relatórios de defeitos e, consequentemente, mais distante do início do ranking elas se encontram.

Mais uma vez, se analisarmos a Figura 4.3h, que representa o ranking retornado para o projeto Eclipse 3.0, podemos ver que a maioria dos pontos estão situados antes da primeira metade do gráfico, ou seja, primeira metade das posições do ranking. De fato, a posição mediana de todas as classes que foram realmente impactadas para o Eclipse 3.0, é 52 (que representa 0,5% de todas as classes do projeto). Mais especificamente, 25% de todas as classes impactadas estão posicionadas no ranking até a 6ª posição. Além disso, 75% delas foram posicionadas antes da 423ª posição, o que representa apenas 4% de todas as classes do Eclipse 3.0. Ou seja, ao analisarmos apenas vocabulário de classes e relatórios de defeitos, pudemos desconsiderar 96% das classes do Eclipse e, mesmo assim, obtivemos um ranking contendo 75% das classes impactadas pelos relatórios analisados.

Também é possível perceber pelas Figuras 4.3f e 4.3g que o mesmo comportamento se repete para as outras duas versões do Eclipse que foram analisadas (versões 2.0 e 2.1). Isso nos dá confiança de que o resultado persiste por entre diferentes versões do projeto Eclipse e os diferentes relatórios de defeitos de cada versão.

Mesmo com projetos com um número muito menor de classes do que o projeto Eclipse, os resultados se mostram satisfatórios, ao sugerirmos classes possivelmente impactadas por relatórios de defeitos, apenas analisando o fator vocabulário. Por exemplo, o projeto Apache Shiro contém 350 classes, e seu gráfico (Figura 4.3e) mostra que, para os 16 relatórios de

Figura 4.3: Posição das Classes Impactadas no Ranking



(a) Apache Hadoop (b) Apache Lucene (c) Apache OpenJPA (d) Apache Pivot

(e) Apache Shiro (f) Eclipse 2.0 (g) Eclipe 2.1 (h) Eclipse 3.0

defeitos analisados, nossa abordagem posicionou todas as 20 classes impactadas por eles nas primeiras 60 posições do projeto (ou seja, referente a 17% de todas as classes do projeto).

A partir dos resultados apresentados, podemos concluir que é possível utilizar similaridade de vocabulários para desconsiderar a grande maioria das classes que não são impactadas pelos relatórios de defeitos, resultando em apenas uma pequena porção do projeto que precisa ser analisada pelo desenvolvedor. Esse é um resultado importante pois, mesmo que a análise do vocabulário não seja efetiva para apontar precisamente quais são as classes impactadas ao se corrigir um relatório de defeito, ela pode reduzir bastante o total de classes que precisam ser analisadas pelo desenvolvedor para que o defeito seja encontrado e corrigido.

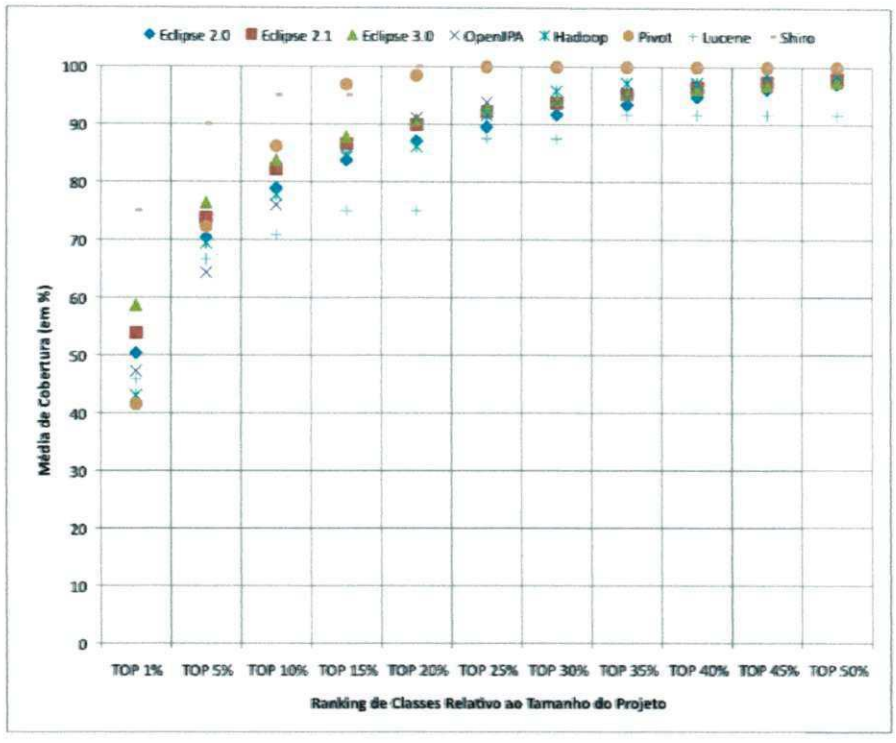
Comparativo com rankings de diversos tamanhos

Outra forma de apresentar resultados que respondam a esta pergunta é construindo um gráfico comparativo dos rankings e suas médias de cobertura para diferentes porções dos projetos. A Figura 4.4 apresenta esta comparação.

Nossa abordagem foi aplicada a cada um dos oito projetos variando a quantidade de retorno de cada ranking (variamos de 1% a 50% do tamanho total dos projetos). A partir disso, calculamos a média de cobertura para ranking. Como é possível perceber, para todos os projetos, os resultados seguiram o mesmo padrão.

Quando pequenas porções do projeto (por ex., 1%) foram consideradas, a média de cobertura alcançada variou de 40% a 75%. Entretanto, rankings com maiores quantidades de sugestões contêm maior quantidade de classes impactadas. Por exemplo, ao considerar similaridade de vocabulários para retornar um ranking com apenas 5% das classes de um projeto, é possível obter em torno de 70% das classes impactadas pelos relatórios de defeitos analisados. Isso significa que é possível, de fato, utilizar similaridade de vocabulários entre código fonte e relatórios de defeitos para descartar em torno de 95% das classes não-impactadas por um relatório de defeito e, ainda assim, obter uma taxa de erro de até 30%. De forma semelhante, se descartarmos 75% das classes não-impactadas, a taxa de erro obtida é de apenas 10%, no máximo.

Figura 4.4: Comparação das Médias de Cobertura para Todos os Projetos Analisados



QP_4: Qual é a menor posição do ranking na qual é possível encontrar uma classe impactada?

Até então, temos respondido às questões de pesquisa buscando entender o quanto nossa abordagem é capaz de identificar todas as classes impactadas pelos relatórios de defeitos. Entretanto, notamos que, caso identifiquemos no ranking pelo menos uma classe impactada pelo relatório de defeitos, isso já pode ser útil para o desenvolvedor, pois ele pode utilizar outras técnicas de predição de propagação de mudanças para descobrir classes que serão também impactadas por consequência. Além disso, faz sentido pensar que nem todas as classes que foram impactadas por um relatório de defeitos (ou seja, classes apontadas pelo oráculo) têm vocabulário similar ao do relatório. Elas podem ter sido, de fato, impactadas por causa da propagação da mudança feita, possivelmente, na classe com vocabulário mais similar ao do relatório (a fonte original do defeito). Sendo assim, mesmo que um ranking não tenha cobertura de 100%, ainda assim ele pode ser útil, pois estará diminuindo o universo de busca do desenvolvedor ao inspecionar um defeito.

Tabela 4.2: Primeiras Posições das Classes Corretamente Sugeridas nos Rankings

Nome do Projeto	**Posição Mín.**	**1º Quartil**	**Mediana**	**3º Quartil**	**Posição Máx.**
Apache Hadoop	1,0	1,0	1,0	2,0	22,0
Apache Lucene	1,0	1,0	1,0	41,5	217,0
Apache OpenJPA	1,0	1,0	4,0	37,5	606,0
Apache Pivot	1,0	1,0	2,0	7,0	29,0
Apache Shiro	1,0	1,0	1,0	2,0	61,0
Eclipse 2.0	1,0	1,0	5,0	40,0	5538,0
Eclipse 2.1	1,0	1,0	8,0	63,7	5767,0
Eclipse 3.0	1,0	1,0	7,0	59,0	8999,0

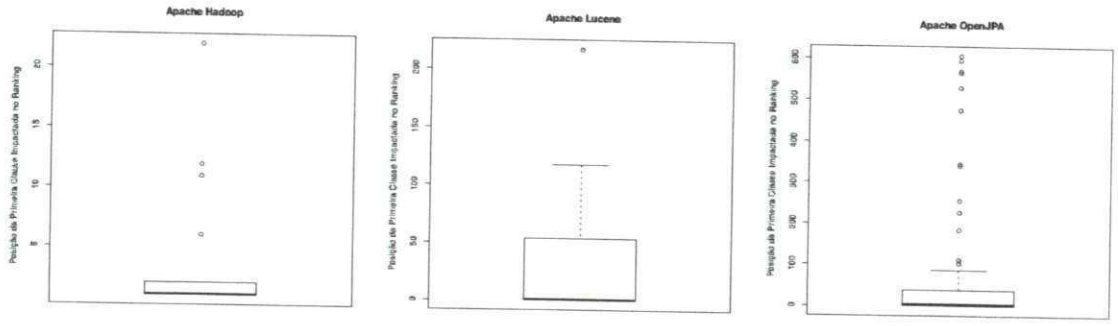
Com esta pergunta, buscamos avaliar em qual posição encontramos a primeira classe corretamente posicionada no ranking. Assim, pudemos avaliar qual é o tamanho do ranking que precisamos retornar para que este contenha a classe com vocabulário mais similar ao do relatório de defeito analisado.

A Tabela 4.2 apresenta os quartis das posições nas quais as primeiras classes corretamente retornadas nos rankings se encontram. Uma outra forma de visualização desses dados está representada na Figura 4.5, a qual contém *boxplots* da distribuição de posições no ranking das primeiras classes que foram corretamente retornadas.

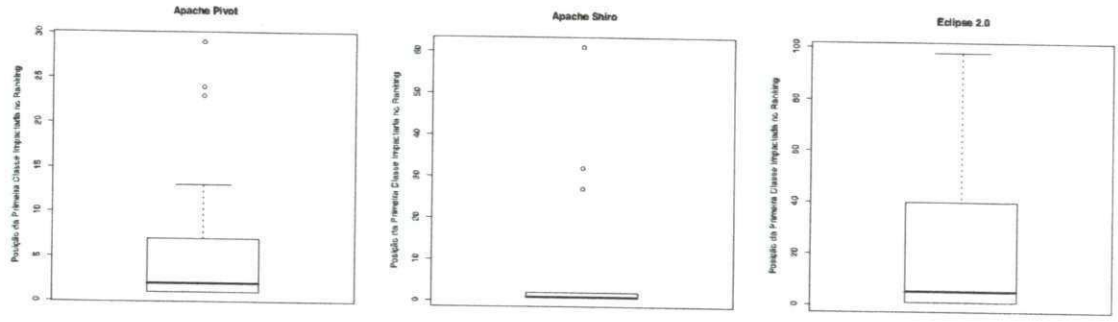
Note que o eixo y dos *boxplots* não representa todas as possíveis posições do ranking (i.e. número total de classes do projeto). Ao contrário, ele representa apenas as posições nas quais foram encontradas as primeiras classes impactadas. Ou seja, apenas uma pequena porção do projeto, já que nossa abordagem é eficaz em filtrar a grande maioria das classes não impactadas. Optamos por plotar os gráficos dessa forma, a fim de que o *boxplot* ficasse mais visível para ser analisado. De forma semelhante, para o caso do projeto Eclipse, não plotamos *outliers*.

Tomando como exemplo o Eclipse 3.0, pode-se perceber que 50% dos rankings apresentam uma classe realmente impactada listada até a sua 7ª posição. Ou seja, mesmo filtrando 99,93% das classes do Eclipse 3.0, ainda assim, é possível encontrar em 50% dos casos, pelo menos, uma classe impactada para cada relatório de defeito que foi analisado. Além disso,

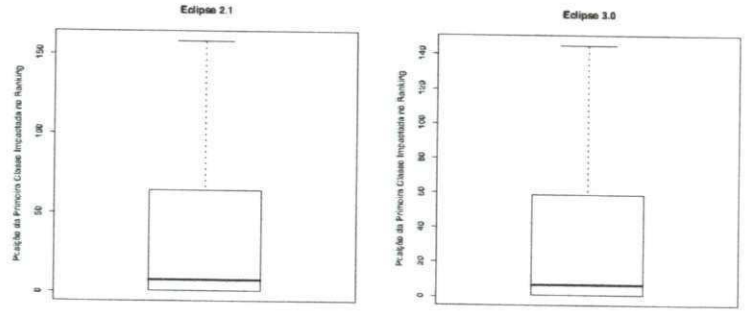
Figura 4.5: Primeiras Posições das Classes Corretamente Sugeridas nos Rankings



(a) Apache Hadoop (b) Apache Lucene (c) Apache OpenJPA



(d) Apache Pivot (e) Apache Shiro (f) Eclipse 2.0



(g) Eclipe 2.1 (h) Eclipse 3.0

75% das listas de ranking do Eclipse 3.0 listam uma classe impactada em até 59 posições (o referente a 0,57% do total de classes do Eclipse).

Se analisarmos os dados de todos os projetos, podemos ver que a primeira classe realmente impactada aparece antes da 9ª posição em 50% dos rankings. Além disso, 75% de todos os rankings analisados contêm pelo menos uma classe corretamente sugerida na posição variando entre menos de 1% a 8% do tamanho do projeto. Em outras palavras, o máximo que precisamos ter é um ranking que desconsidere automaticamente 92% das classes do projeto, apenas analisando similaridade de vocabulários, para encontrarmos 75% dos defeitos descritos em relatórios.

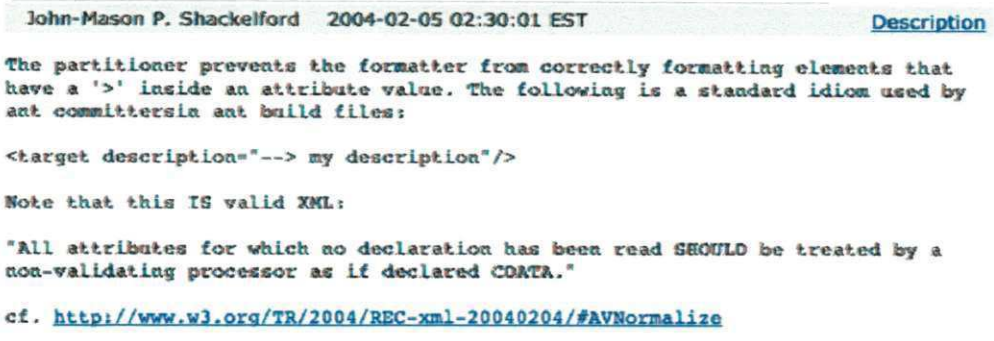
É importante notar também que as posições máximas nas quais aparecem as primeiras classes impactadas são geralmente muito altas, quase totalizando o número de classes do projeto. Ou seja, há casos em que a primeira classe corretamente sugerida aparece no fim do ranking. Isso ocorre porque, de fato, há relatórios de defeitos cujo vocabulário não é similar aos das classes impactadas. Desse modo, fica evidente que a nossa abordagem não é suficiente para encontrar classes defeituosas referentes a *todos* os relatórios de defeitos, mas, pelo menos, para a maioria dos relatórios, ela se mostra eficaz.

Para ilustrar um caso no qual a nossa abordagem posicionou uma classe realmente impactada no fim do ranking (8.999ª posição), analisemos a descrição do relatório da Figura 4.6, cujo ID é 51215[3] e tem como título: "*fix partioner to handle angle brackets in attribute values*" [*sic*]. Ao mapearmos esse relatório com as classes que foram impactadas para a sua correção, pudemos perceber que todo o conjunto do vocabulário da classe impactada é composto pelas seguintes palavras: $\{tag, rule, sequence, detected, end, scanner, token\}$. Ou seja, como é possível perceber, o vocabulário da classe é bem diferente do vocabulário do relatório de defeitos. Portanto, uma técnica que analise apenas vocabulários, realmente não seria capaz de sugerir esta classe como provável a ser impactada pelo relatório analisado.

Por fim, seguindo essa mesma ideia de análise, vimos que, ao retornarmos um ranking com um número fixo de 10 elementos, independente do tamanho do projeto, a nossa abordagem foi capaz de identificar, pelo menos, uma classe impactada entre 54% e 60% das vezes, para as três versões analisadas do projeto Eclipse; e entre 67% e 89%, para os projetos da

[3]Relatório acessível via: `https://bugs.eclipse.org/bugs/show_bug.cgi?id=51215`, verificado em outubro de 2012

Figura 4.6: Exemplo de Relatório de Defeitos do Projeto Eclipse 3.0



Fonte: Repositório aberto de relatórios de defeitos do projeto Eclipse.

Apache. Portanto, como pode ser visto, mesmo retornando um número pequeno de classes, apenas analisando similaridade de vocabulários, os resultados são corretos em mais da metade das vezes.

Em resumo, podemos concluir que a análise de similaridade de vocabulários pode apresentar bons resultados para sugerir um conjunto inicial de impacto ao se resolver um defeito, nos projetos analisados, embora ela não seja 100% eficaz para todos os casos analisados.

QP_5: A análise de comentários de código impacta na melhoria da abordagem?

Ao propor a nossa abordagem, consideramos como vocabulário de software apenas os identificadores escolhidos pelos desenvolvedores para nomear classes, atributos, constantes e métodos. Sendo assim, desconsideramos em nosso estudo os comentários presentes no código.

Entretanto, Haiduc e Marcus [14] defendem que os comentários presentes no código devem ser considerados como parte integrante do vocabulário de software, pois eles podem levar a uma melhor compreensão do código. Dessa forma, desenvolvemos um estudo para verificar se há melhoria em nossa abordagem caso considerássemos comentários presentes no código.

Para isso, aplicamos novamente a nossa abordagem aos oito projetos estudados, dessa vez, considerando os comentários *Javadoc* do código, que foram desconsiderados para as

análises anteriores. Os rankings obtidos foram então analisados segundo a métrica de média de cobertura (tal como na QP_4).

O resultado comparativo pode ser visto na Figura 4.7. Os gráficos nos mostram que, para os projetos da fundação Apache, ao considerarmos comentários *JavaDoc*, temos uma perda de em torno de 5%, em relação à média de cobertura. Entretanto, não há mudança visível para os projetos Eclipse. Ou seja, ao não considerarmos comentários como parte integrante do vocabulário do software, no geral, nossa abordagem funciona melhor ou, no máximo, sem alterações.

De fato, o uso de comentários aumenta consideravelmente o tamanho do conjunto de termos do vocabulário do software. Assim, aqueles identificadores, tais como nomes de classes e métodos, que seriam considerados como parte principal para a compreensão da classe, ficam diluídos por entre dezenas de outros termos oriundos dos comentários.

Logo, concluímos que o uso de comentários como parte integrante do vocabulário do software não é recomendado para a nossa abordagem.

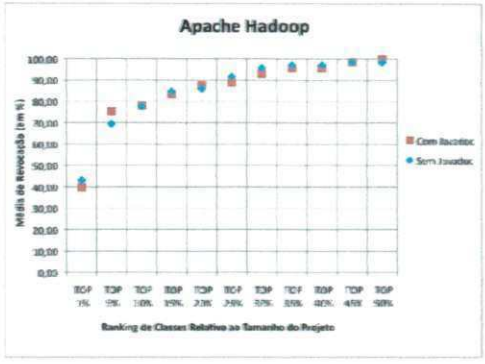
4.3 Limitações do Estudo

Há alguns aspectos que podem influenciar a generalização das nossas observações, ameaçando a validade externa do nosso estudo. Nesta seção, apresentamos e fazemos considerações sobre as ameaças que foram identificadas no decorrer do mesmo.

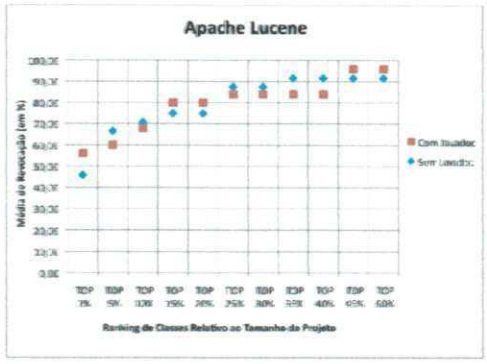
Inicialmente, como avaliamos apenas oito versões de seis projetos distintos, não podemos garantir que nossas observações são generalizáveis para qualquer software. Entretanto, mesmo sendo uma amostra pequena, vale ressaltar que estudamos sistemas reais e com dados reais do repositório de defeitos. Os resultados mostraram que, no geral, obtivemos o mesmo padrão de resultados para todos os projetos. Isso nos reforça a ideia de que os resultados obtidos podem ser esperados para projetos de software semelhantes.

Outra ameaça identificada é que todos os projetos analisados são de código aberto. Assim, não garantimos que as mesmas conclusões possam ser tiradas para projetos proprietários. Além disso, todos os projetos estudados foram programados utilizando a linguagem Java e têm identificadores apenas em inglês. Desse modo, entendemos que são necessários mais estudos que incluam projetos proprietários, estrangeiros e que utilizem outras lingua-

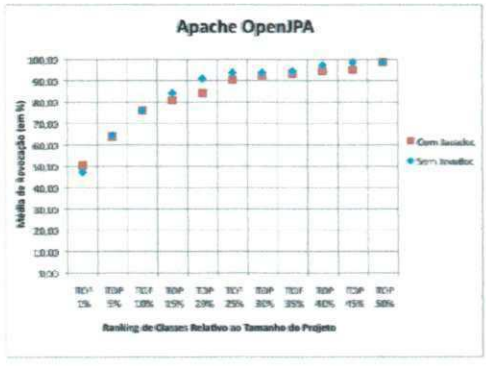
Figura 4.7: Comparativo da Análise com Comentários no Vocabulário de Software



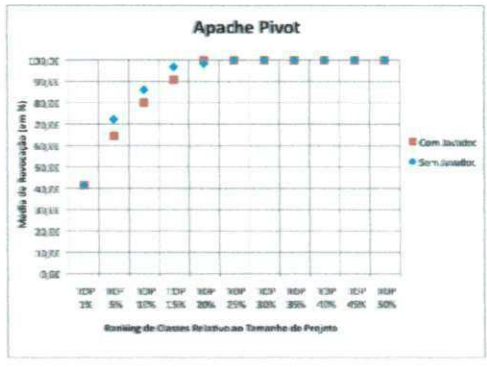
(a) Apache Hadoop



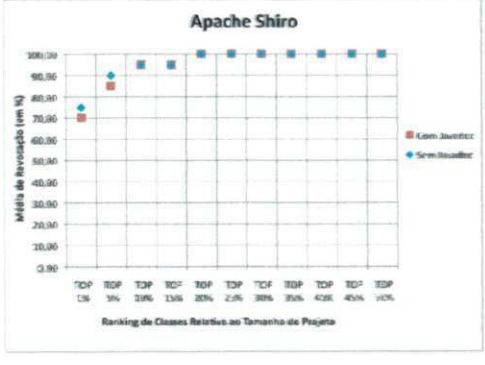
(b) Apache Lucene



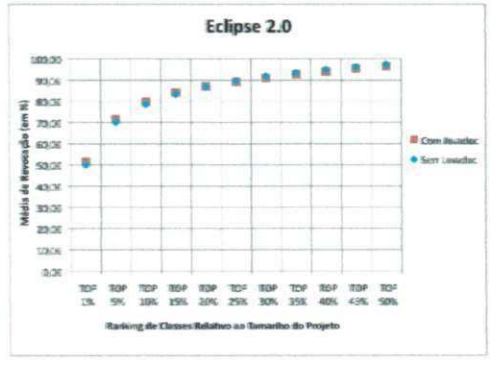
(c) Apache OpenJPA



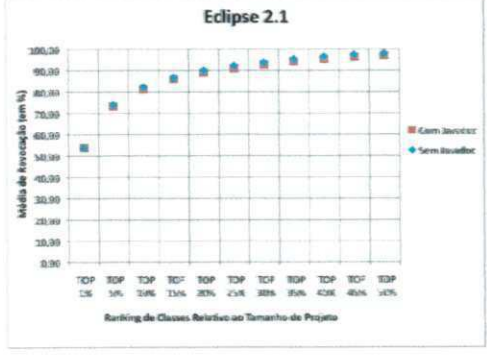
(d) Apache Pivot



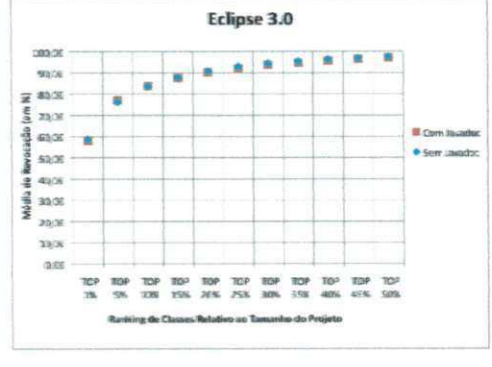
(e) Apache Shiro



(f) Eclipse 2.0



(g) Eclipe 2.1



(h) Eclipse 3.0

gens de programação.

Mais ainda, similar a outros trabalhos que também analisam vocabulário de software, os resultados da nossa abordagem dependem de que o código tenha identificadores significativos, que compreendam o contexto da aplicação. Entretanto, acreditamos que isso não seja uma grande limitação, já que a utilização de bons identificadores está se tornando, nos últimos anos, comum por entre os engenheiros de software [14].

Também podemos citar que o estudo que fizemos envolve projetos cujos usuários são, em sua maioria, da área técnica (geralmente, programadores). Ou seja, mesmo que a maioria dos usuários não conheça o código fonte dos projetos, eles conhecem jargões técnicos que podem ser próximos dos identificadores escolhidos pelos desenvolvedores dos projetos ao implementarem novas funcionalidades e, consequentemente, ao introduzirem os defeitos que são reportados. Dessa forma, os relatórios de defeitos dos projetos estudados podem conter mais termos similares aos das classes defeituosas, do que se analisássemos projetos cujos usuários são leigos da área de tecnologia. É importante que uma investigação similar seja feita para outros tipos de projetos.

Por fim, ao relacionarmos relatórios de defeitos e código fonte impactado, supomos que os desenvolvedores que corrigiram tais defeitos fizeram um único *commit* no repositório de código com todas as classes – e apenas elas – que foram impactadas ao corrigirem aquele defeito. Embora seja uma prática comum por entre os projetos, poderão haver casos nos quais os desenvolvedores não seguiram essa diretriz. De modo a remover esta ameaça, é preciso que o mesmo estudo seja feito em um ambiente mais controlado, no qual desenvolvedores sejam alertados e auditados sobre esta regra ao submeterem *commits* de código.

Capítulo 5

Discussão Geral

Neste capítulo, apresentamos uma discussão geral sobre os resultados obtidos neste estudo, bem como sugestões de trabalhos futuros que podem ser desenvolvidos a partir deste.

5.1 Relação Entre Vocabulários de Software e de Relatórios de Defeitos

Conforme dito anteriormente, os resultados obtidos tornam evidente que vocabulário de software pode ser utilizado, juntamente com o vocabulário extraído de relatórios de defeitos, para ranquear classes possivelmente impactadas pelos relatórios analisados e, consequentemente, desconsiderar aquelas que são pouco prováveis de serem impactadas.

Os bons resultados obtidos refletem a similaridade existente entre os nomes de identificadores escolhidos pelos desenvolvedores e os termos utilizados pelas pessoas que reportam relatórios de defeitos. Essa similaridade ajuda a ressaltar as classes mais relacionadas aos relatórios de defeitos, mesmo que ela não esteja evidente a olho nu, já que, de uma visão leiga, estaríamos comparando texto livre com identificadores desconexos. De fato, o tratamento e indexação dos vocabulários seguindo técnicas de recuperação da informação são a chave para que as entidades sejam relacionáveis entre si, por meio do vocabulário.

Mesmo assim, entendemos que essa relação possa se tornar ainda mais evidente, caso os relatórios de defeitos sejam modificados, de modo que a descrição dos defeitos contenham ainda mais palavras relacionadas ao vocabulário do software. Isso pode ser feito, por

exemplo, pela implementação de um módulo de sugestão automática de *tags* extraídas de termos do código. Desse modo, a pessoa que estivesse reportando um defeito poderia escolher *tags* que fossem relacionadas ao problema relatado, a partir de sugestão das mesmas. Esse é um trabalho que visa melhorar a qualidade dos relatórios de defeitos e que pode ser implementado por meio da modificação de sistemas de rastreamento de defeitos já utilizados na prática, tais como Bugzilla e JIRA.

Quanto mais trabalhos forem desenvolvidos para melhorar a qualidade de relatórios de defeitos reportados, melhores resultados a abordagem proposta neste documento conseguirá alcançar. Mais ainda, quanto mais os desenvolvedores dos projetos tiverem consciência da boa escolha de identificadores, mais o vocabulário do software se aproximará, em termos de similaridade, com o vocabulário contido no repositório de defeitos. Portanto, trabalhos futuros no tocante a essas áreas também são importantes de serem desenvolvidos para a melhoria da nossa abordagem.

5.2 Uso da Abordagem Proposta para Localizar Classes Defeituosas

Ao utilizarmos análise de similaridade de vocabulários para retornarmos um ranking contendo classes possivelmente impactadas por relatórios de defeitos, foi possível desconsiderar grandes porções do software e, ainda assim, obter uma boa taxa de acerto em relação às classes desejadas. Entretanto, não podemos dizer que tal abordagem deva ser utilizada de forma isolada para tal finalidade, dado que, mesmo considerando pequenas porções de um software, ainda assim, o desenvolvedor poderá ter centenas ou milhares de classes para avaliar. Esse fato torna-se ainda mais evidente quando sistemas de grande porte são analisados, tais como fizemos em nosso estudo. Essa foi uma das principais razões por termos desconsiderado sistemas pequenos como parte da avaliação. Desde o início, nosso desejo sempre foi ser o mais justo possível e exercitar nossa abordagem com milhares de classes e centenas de relatórios de defeitos.

Diante do exposto, podemos afirmar que não chegamos a uma técnica ótima, mas ela é um bom ponto de partida para outros estudos e técnicas serem derivados com o objetivo de localizar defeitos sem a utilização de dados históricos. Por outro lado, não temos indí-

cios suficientes que mostrem (ou que neguem) que técnicas que não utilizem algoritmos de aprendizagem – e, portanto, dados históricos – possam ser boas o suficiente para localizarem defeitos apenas por meio da análise de informações estáticas. No decorrer do estudo, tínhamos em mente apenas que não utilizaríamos tais dados históricos, uma vez que muito já foi investigado sobre esse tipo de abordagem para treinar algoritmos. Além disso, nosso desejo foi o de investigar o vocabulário por si, sem a utilização de nada mais que pudesse influenciar a decisão do algoritmo ao ranquear as classes. Por fim, também buscamos evitar o problema de partida a frio (em inglês, *cold start problem*), já que todas as abordagens que foram propostas até hoje e das quais temos conhecimento, sofrem com esse problema de não conseguirem boas taxas de acerto quando não têm dados suficientes sobre as classes ou tipos de relatórios que estão sendo processados.

5.3 Caso Base para Comparação de Resultados

Até onde temos conhecimento, não há trabalhos na literatura que definam um caso base (em inglês, *baseline*) que seja apropriado para comparar resultados de um estudo como o nosso, que sugere classes prováveis de serem impactadas por meio da análise de similaridade com o vocabulário de relatórios de defeitos. Geralmente, os trabalhos relacionados apenas apresentam cobertura ou precisão como métricas avaliadas dos seus resultados, mas não os comparam diretamente com nenhuma outra técnica.

Acreditamos que o melhor caso base para nosso estudo seria uma comparação do tempo gasto por cada desenvolvedor dos projetos analisados para acharem defeitos descritos em relatórios com e sem auxílio da nossa abordagem. Contudo, tal estudo é impraticável, dado que os projetos são de código aberto e os desenvolvedores estão espalhados ao redor do mundo. Entretanto, um estudo similar com projetos locais e menores poderá ser feito em trabalhos futuros. Por exemplo, pode-se avaliar tempo e esforço despendidos por dois grupos de desenvolvedores ao corrigirem defeitos, tendo um grupo utilizado a nossa abordagem.

Uma opção mais simplista de caso base seria a escolha aleatória de um conjunto de classes a serem retornados e depois comparados por nossos resultados. Embora essa escolha fosse válida, tal comparação não seria justa, já que, mesmo a heurística mais ingênua possível, certamente seria melhor do que a utilização de uma seleção aleatória de classes.

Por fim, também pensamos em desenvolver um estudo comparativo tomando os trabalhos relacionados como casos base. Entretanto, não foi possível encontrar dados suficientes para reproduzirmos seus experimentos. Além disso, a grande maioria dos trabalhos relacionados não analisam vocabulário de software como fator. Ao contrário, eles utilizam dados históricos do repositório para sugerir classes possivelmente impactadas.

Diante do exposto, é possível perceber que mais estudos são necessários, de modo que seja possível encontrar um caso base adequado para trabalhos similares ao nosso. Entretanto, acreditamos que nossos resultados podem ser utilizados como casos base para trabalhos futuros que também venham a estudar o uso do vocabulário de software para localização de defeitos.

5.4 Trabalhos Futuros

Mesmo que não seja possível utilizar apenas análise de similaridade de vocabulários para ranquear classes, certamente tal análise pode servir de suporte para tal tarefa. Combinada com outras técnicas ou com algoritmos melhorados, será possível utilizar a abordagem proposta para criar ferramentas que possam auxiliar desenvolvedores na localização de classes defeituosas, partindo de relatórios de defeitos.

Por exemplo, uma forma de tentar melhorar a abordagem é atribuir mais peso no cálculo da similaridade dos termos presentes nos títulos dos relatórios de defeitos. Essa ideia parte do fato de que os títulos dos relatórios são resumos de uma linha do problema relatado. Portanto, eles geralmente contêm as palavras-chaves que descrevem aquele defeito. Ou seja, se os termos dos títulos forem similares aos termos de uma dada classe, a probabilidade é maior que aquela classe seja a desejada. O trabalho de Ko et al. [32] pode servir de guia para um estudo posterior que vise essa mudança na abordagem. Nele, os autores estudaram em torno de 200 mil títulos extraídos de relatórios de defeitos de vários projetos e extraíram padrões interessantes sobre como as pessoas resumem os defeitos.

Outro fator que pode auxiliar bastante no processamento do vocabulário de relatórios de defeitos é a identificação automática de estruturas presentes nos mesmos. Por exemplo, Bettenburg et al. [25] extraíram pilhas de execução e pedaços de código que, muitas vezes, referenciam diretamente classes existentes no código. Assim, uma abordagem que leve em

consideração esses dados, pode identificar automaticamente classes que foram referenciadas pela pessoa que reportou o defeito no relatório.

Além disso, seria interessante um estudo adicional que associe a nossa abordagem a técnicas de predição de propagação de mudanças para verificar se, dada a primeira classe impactada, as demais são realmente encontradas. Assim, poderíamos melhorar a análise da nossa questão de pesquisa QP_4, que analisa o retorno apenas da primeira classe impactada nos rankings.

Por fim, algumas ferramentas podem ser desenvolvidas para auxiliar os desenvolvedores no processo de localização de entidades defeituosas. Por exemplo, é possível desenvolver uma extensão para o Eclipse (ou outro IDE) que indique aos desenvolvedores, para cada classe que está sendo modificada, quais são os relatórios de defeitos que ainda estão em aberto e que possam estar relacionados àquela classe. Assim, o desenvolvedor poderia visualizar facilmente quais os possíveis defeitos já relatados para a classe a qual ele está modificando. Isso poderia diminuir o esforço de localizar classes possivelmente defeituosas e, consequentemente, aumentar a taxa de defeitos corrigidos.

Capítulo 6

Trabalhos Relacionados

Nos últimos anos, diversos estudos têm surgido na literatura envolvendo técnicas de recuperação da informação (RI) aplicadas à manutenção e evolução de software. Binkley e Lawrie [19] apresentaram um catálogo de trabalhos recentes nesta linha de pesquisa. Eles categorizaram cada trabalho com os seguintes focos principais: localização de conceitos ou funcionalidades; predição de defeitos; identificação de desenvolvedores; compreensão de código; análise de impacto; rastreamento de ligações entre código e artefatos; e refatoramento. Segundo as descrições dadas no catálogo para as categorias dos trabalhos, nossa pesquisa pode ser categorizada como sendo da área de análise de impacto, já que utilizamos RI para predizer o provável impacto no código durante a resolução de defeitos, descritos em relatórios.

Durante o nosso levantamento bibliográfico, selecionamos trabalhos que estão relacionados ao processamento de relatórios de defeitos e/ou vocabulário de software. Neste capítulo, apresentamos os trabalhos selecionados que são mais relevantes ao nosso e fazemos uma análise comparativa entre eles e o estudo apresentado neste documento.

6.1 Estudo sobre o Vocabulário de Software

Dentre os trabalhos que estudam vocabulário de software, podemos citar os de Haiduc e Marcus [14], Abebe et al. [33] e Santos et al. [20], que apresentaram estudos de caso com sistemas de software, nos quais foi possível identificar como se dá a evolução do vocabulário de software e suas relações com as funcionalidades do sistema.

Abebe et al. identificaram que, à medida que o software cresce em tamanho, o vocabulário também cresce junto. Ou seja, há um incremento de termos no vocabulário do software sempre que novas funcionalidades são implementadas. Entretanto, também foi descoberto que, durante a manutenção e evolução do software, os desenvolvedores têm a preocupação em reusar termos já existentes sempre que possível. Isso demonstra que os desenvolvedores geralmente se preocupam com a boa manutenção do vocabulário do software.

Além disso, ambos os trabalhos demonstraram que os termos presentes no vocabulário são, de fato, relacionados ao domínio do problema e às funcionalidades implementadas. De fato, Haiduc e Marcus mostraram que os vocabulários de sistemas do mesmo domínio se mantêm consistentes e relevantes. Essas afirmações indicam que o vocabulário de software representa uma fonte significativa de termos do domínio, que podem ser utilizados para diversos propósitos, entre eles a compreensão do software e análise de impacto. Tais estudos são relevantes para o nosso trabalho, já que nossos resultados dependem diretamente da qualidade do vocabulário do software e da sua similaridade com as funcionalidades implementadas.

Por fim, Santos et al. estudaram vocabulários de 39 projetos de código aberto e modelaram o número de termos contidos no vocabulário de software e suas frequências em função do tamanho do projeto. Assim, os autores tornaram possível a geração automática de vocabulários sintéticos, a partir de seus modelos estatísticos. Tais vocabulários podem ser usados em trabalhos futuros como fontes de dados para experimentos.

6.2 Recomendação de Entidades de Código Fonte a Partir de Dados Históricos

No decorrer dos últimos anos, trabalhos foram publicados descrevendo técnicas e ferramentas cujo principal objetivo era a recomendação de entidades de código fonte (por exemplo, classes em Java) a partir do processamento de dados históricos provenientes do repositório de relatórios de defeitos. Nesta seção, apresentamos alguns desses trabalhos e suas semelhanças e diferenças com o nosso.

6.2.1 Pioneirismo na Análise Textual de Relatórios de Defeitos para Análise de Impacto

Segundo nosso levantamento bibliográfico, até onde temos conhecimento, Canfora e Cerulo [3] foram os pioneiros na área de análise de impacto a utilizarem repositório de relatórios de defeitos e recuperação da informação para apontarem entidades do software que são prováveis de serem impactadas. Antes deles, houve um trabalho similar [34] que utilizava recuperação da informação para sugerir documentos que tenham ligação com a requisição de mudança analisada. Entretanto, nenhuma técnica específica foi sugerida para se utilizar de dados de repositório com esta finalidade. Dessa forma, podemos afirmar que Canfora e Cerulo trouxeram para a literatura uma nova abordagem de análise: o processamento textual do repositório de requisições de mudanças (RMs) para recuperar quais são os arquivos de código fonte que seriam impactados por uma mudança.

Em resumo, os autores propõem uma abordagem que retorna o conjunto de arquivos de código fonte que são candidatos a serem impactados por uma RM, utilizando técnicas de recuperação da informação para a indexação e busca no vocabulário das RMs. Para isso, primeiramente, eles indexam todo o vocabulário de RMs passadas, armazenadas no repositório do Bugzilla. Enquanto indexam esse vocabulário, cada termo é associado aos nomes dos arquivos de código fonte impactados pela RM sendo processada. Por fim, com a chegada de uma nova RM no projeto, são recuperadas outras RMs cujo vocabulário seja similar àquela e são retornadas os nomes dos arquivos de código fonte a elas associadas.

Diante do exposto, podemos dizer que a ideia de Canfora e Cerulo é similar à nossa e, de fato, o trabalho desenvolvido por eles serviu de inspiração em diversos aspectos da nossa pesquisa. Entretanto, há uma diferença significativa do seu trabalho com o nosso no tocante à metodologia de análise. Enquanto eles propõem uma abordagem que compara texto unicamente extraído do histórico de RMs do projeto, nós comparamos textos cujas fontes são o código fonte e relatórios de defeitos (para eles, requisições de mudanças – RMs), sem a necessidade de haver um histórico previamente armazenado.

Como forma de validar a abordagem proposta, os autores realizaram experimentos com quatro projetos de código aberto, a saber: KCalc, KPDF, KSpread e Firefox. A quantidade de classes dos projetos varia entre 119 e 1.454; e o número de RMs analisadas varia de 22 a

62. Seus resultados mostram que a precisão da abordagem fica entre 30% e 78%.

Em um outro artigo [22], similar ao anterior, os mesmos autores também apresentam resultados de experimentos feitos para a análise de impacto retornando resultados em nível de granularidade de linhas de código. Seguindo a mesma ideia do trabalho anterior, para cada requisição de mudança (RM) que é indexada, os autores a associam às linhas de código modificadas para implementação da mudança. Na chegada de uma nova RM no projeto, eles fazem uma busca levando em consideração o vocabulário dessa nova RM e recuperam todas as RMs anteriores que têm vocabulário similar. Assim, é possível classificar as linhas de código associadas às RMs retornadas.

A avaliação do trabalho foi feita utilizando os seguintes projetos de código aberto: Gedit, ArgoUML e Firefox. O número de RMs analisadas varia de 116 a 670. Com a análise em nível de linhas de código fonte, os autores obtiveram resultados de precisão 10% melhores, mas com revocação pior.

É importante perceber que eles não fizeram experimentos em projetos grandes, como o Eclipse, que contém mais de quatro mil classes e milhares de RMs. Além disso, foram comparados textos apenas entre RMs. Portanto, é mais intuitivo de que a similaridade encontrada seja maior, dado que ambos o texto indexado e o texto utilizado como busca são provenientes do mesmo tipo de fonte. Por outro lado, nós analisamos textos de dois tipos de fontes diferentes (código fonte e relatórios de defeitos) e os associamos por meio da análise da similaridade.

Por fim, a sua abordagem não leva em consideração vocabulário de software. Ao contrário, apenas é indexado o vocabulário das RMs, que é associado aos arquivos ou às linhas dos arquivos de código fonte modificados. Além disso, a abordagem ainda necessita que haja um histórico de RMs para funcionar. A nossa, ao contrário, não apresenta essa limitação.

6.2.2 Uso de Algoritmos de Aprendizagem

Moin e Khansari [26] também apresentaram uma abordagem para localizar defeitos a partir do processamento do texto de bug reports. Para tanto, eles utilizam uma abordagem idêntica à de Canfora e Cerulo [3], no tocante ao processamento do vocabulário de relatórios de defeitos e associação dos mesmos às entidades do código que foram modificadas por eles. A diferença é que Moin e Khansari utilizam Máquina de Vetores de Suporte (em inglês,

Support Vector Machines ou SVM) [35] ao invés de técnicas de recuperação da informação. Além disso, eles retornam o caminho do pacote que é provável de ser impactado, ao invés do arquivo de código fonte (ou classe, em Java). Por exemplo, uma possível resposta da ferramenta proposta poderia ser: $ui/org/eclipse/jdt/internal/ui$.

A validação foi feita por meio do treinamento e teste do classificador proposto com mais de 2.000 relatórios de defeitos do projeto Eclipse JDT e 23 possíveis pacotes a serem impactados. Os resultados mostram que a revocação e precisão alcançadas ficaram em torno de 98% para o projeto analisado. Entretanto, vale ressaltar que a ferramenta retorna caminhos de pacote e que o universo de escolha era de 23 possíveis caminhos apenas.

Outro trabalho na mesma linha é o BugScout, que é uma ferramenta proposta por Nguyen et al. [6] com o propósito de encontrar arquivos de código fonte que sejam responsáveis pelo defeito descrito em um relatório. A ferramenta proposta extrai o vocabulário do software e o classifica em um conjunto finito de tópicos, utilizando o modelo generativo de Alocação Latente Dirichlet (em inglês, Latent Dirichlet Allocation ou LDA) [36]. Para isso, é preciso que haja um treinamento do algoritmo de aprendizagem com dados históricos para que sejam extraídos os tópicos necessários para categorização. O mesmo processo de extração e categorização é repetido para o vocabulário de relatórios de defeitos.

Experimentos foram realizados com quatro projetos de código aberto. Os resultados mostram que o BugScout recomenda arquivos corretos em até 45% das vezes ao retornar um ranking contendo 10 elementos.

Assim como os trabalhos apresentados na seção anterior, ambas as técnicas propostas que utilizam algoritmos de aprendizagem também necessitam de um histórico de relatórios de defeitos relacionados ao código existente para seu treinamento. Nossa técnica, não depende disso e funciona independente da presença de histórico no repositório. Nós analisamos puramente a similaridade de vocabulários entre relatórios de defeitos e código fonte.

6.2.3 Abordagem Híbrida para Recomendação de Classes Defeituosas

As abordagens propostas por Zhou et al. [7] e Davies et al. [8] têm uma característica híbrida: ambas analisam vocabulários de software e de relatórios de defeitos, juntamente com dados históricos do repositório.

Zhou et al. propuseram uma versão revisada do Modelo de Espaço Vetorial (em inglês,

a sigla ficou *rVSM*), no qual eles levam em consideração o tamanho do documento para otimizar o algoritmo clássico de Modelo de Espaço Vetorial (VSM). Além disso, eles também ajustam os rankings obtidos, adicionando informações de defeitos similares corrigidos no passado. Assim, podemos dizer que a sua abordagem utiliza uma versão modificada do mesmo algoritmo de recuperação da informação que utilizamos mais os dados históricos de defeitos corrigidos no sistema. A ferramenta *BugLocator* foi implementada para automatizar a técnica proposta.

Já Davies et al. utilizaram um classificador implementado na biblioteca de mineração de dados Weka[1], para relacionar relatórios de defeitos e métodos. Eles utilizaram dados históricos de relatórios cujo vocabulário era similar ao relatório que estava sendo analisado. A principal diferença entre o trabalho de Davies et al. e o de Zhou et al., segundo os próprios autores, é que, enquanto este sugere classes que podem ser modificadas, aquele sugere métodos que serão possivelmente impactados.

Nos dois trabalhos, no total, foram feitos experimentos com oito projetos de código aberto (Eclipse, SWT, AspectJ, ZXing, ArgoUML, JabRef, jEdit e myCommander) e um conjunto de 3.851 relatórios de defeitos. Resultados apontam que a técnica de Zhou et. al. é capaz de identificar, de 29% a 40% das vezes, uma classe impactada na primeira posição do ranking. Para um ranking contendo 5 elementos, a técnica é capaz de identificar pelo menos uma classe impactada entre 51% e 67% das vezes; e num ranking contendo 10 elementos, esse número passa ao intervalo de 59% a 81%. Ainda mais, sem utilizar dados históricos mas utilizando a versão modificada do algoritmo de VSM, Zhou et al. apontam que eles conseguem acertar de 55% a 77%, considerando apenas a primeira classe impactada que surge no ranking.

Em relação à avaliação feita por Davies et al., podemos dizer que os autores buscaram apenas comparar seus resultados com uma abordagem proposta anteriormente. Com isso, eles descobriram que a sua técnica, que utiliza classificadores com árvores de decisão, pode aprimorar abordagens anteriormente propostas para localização de defeitos, tendo melhoria estatisticamente significativa em dois dos quatro projetos analisados.

Os trabalhos híbridos apresentados são bem similares ao nosso, embora eles tenham feito experimentos com uma versão modificada do algoritmo que utilizamos e com um outro tipo

[1] http://www.cs.waikato.ac.nz/ml/weka/, verificado em outubro de 2012.

de algoritmo de aprendizagem. Mais uma vez, nosso foco foi o estudo da aplicação mais simples possível de recuperação da informação com vocabulário de software e de relatórios de defeitos, de modo que pudéssemos analisar qual seria a performance de tal análise apenas utilizando ambos os vocabulários, sem nenhuma outra técnica em conjunto.

Ficamos satisfeitos em ver que nossos resultados não se distanciam em grande escala do que foi conseguido pelos trabalhos apresentados e que tiveram o mesmo objetivo, mesmo eles tendo modificado o algoritmo, adicionando mais dados, que estavam fora do escopo da nossa proposta.

6.3 Processamento de Vocabulário de Relatórios de Defeitos para Fins Diversos

Vários trabalhos processam relatórios de defeitos para propósitos distintos. Nesta seção, serão apresentados trabalhos que também processam o texto descrito em relatórios de defeitos, entretanto para outra finalidade. Apesar do foco de seus trabalhos serem diferentes do nosso, eles nos foram importantes pois pudemos aprender com a experiência relatada e adaptar para a finalidade da nossa pesquisa.

6.3.1 Alocação de Desenvolvedores a partir do Vocabulário de Relatórios de Defeitos

Visando diminuir o custo de alocação de desenvolvedores para corrigirem os relatórios de defeitos de projetos, alguns pesquisadores têm trabalhado em abordagens que processam dados históricos de relatórios de defeitos para indicarem automaticamente quais desenvolvedores parecem ser os mais apropriados a resolverem os defeitos relatados.

O trabalho de Matter et al. [23] nos foi bastante relevante. Nele, os autores propuseram um modelo de *expertise* baseado em contribuições do código fonte e vocabulário de relatório de defeitos. Em resumo, eles estudam o vocabulário encontrado nas diferenças (chamadas de *diff*) das contribuições de desenvolvedores ao código fonte, bem como o vocabulário encontrado nos relatórios de defeitos do projeto. A partir disso, eles recomendam desenvolvedores cujo vocabulário de contribuição ao código fonte é similar ao vocabulário do relatório de

defeito.

Os autores avaliaram o trabalho utilizando o projeto Eclipse como estudo de caso. Eles treinaram seu sistema semanalmente com dados do Eclipse durante oito anos, e utilizaram o modelo de *expertise* resultante para associarem desenvolvedores a relatórios de defeitos, analisando a similaridade dos vocabulários processados. Os resultados mostram que eles conseguiram precisão de 33,6% para a primeira posição do ranking e revocação de 54,2% para um ranking contendo 10 sugestões.

É importante notar que os autores utilizam análise de parte do vocabulário do software e do vocabulário de relatórios de defeitos, obtendo uma boa taxa de revocação. Isso nos reforçou a ideia de que comparar ambos os vocabulários é viável e pode nos trazer bons resultados.

Também podemos citar as pesquisas de Anvik et al. [10] e Cubranic e Murphy [24], que, tais como o trabalho de Matter et al. [23], visam reduzir o tempo gasto com a alocação de desenvolvedores para trabalhar em relatórios de defeitos. Ambas as abordagens utilizam algoritmos de máquina de aprendizagem para fazer a recomendação automática de desenvolvedores que estejam relacionados a um relatório de defeito. Além disso, ambos também utilizam dados históricos de repositório de relatórios de defeitos para extrair padrões textuais das descrições cadastradas pelos usuários e desenvolvedores.

Em ambas as pesquisas, foram feitos experimentos com relatórios de defeitos provenientes dos projetos Eclipse e Firefox. Os resultados alcançados variam entre 30% a 64%.

A diferença principal dos trabalhos supracitados para o nosso é que, enquanto eles focam na associação de relatórios de defeitos a desenvolvedores, nós focamos na associação de relatórios de defeitos a código fonte. Além disso, não utilizamos dados históricos do projeto.

6.3.2 Propósitos Diversos

Em 2002, Antoniol et al. [37] apresentaram um estudo sobre como fazer a ligação entre código fonte e a documentação do projeto. Seu estudo focou principalmente no aspecto de processamento de vocabulário e nas suas diversas aplicações.

No ano seguinte, seguindo a mesma ideia, Cubranic e Murphy [38] propuseram a ferramenta *Hipikat*, que recomenda artefatos existentes no projeto que são relevantes para tarefas a serem realizadas. Eles analisam vários artefatos de software, tais como repositório de có-

digo, sistema de rastreamento de defeitos, canais de comunicação (como listas de e-mails) e documentação online. A partir disso, eles usam, dentre outras heurísticas, similaridade de texto para fazer a busca em seu banco de dados de artefatos.

Em suma, a técnica proposta sugere artefatos a partir do código fonte. Nós realizamos a mesma tarefa no sentido oposto: sugerimos entidades do código fonte (por exemplo, classes em Java) a partir de um tipo de artefato específico: relatórios de defeitos.

Apesar dos trabalhos serem voltados a outro tipo de recomendação, suas discussões em torno do processo de tratamento do vocabulário de código foram valiosas para o desenvolvimento da nossa abordagem.

Por fim, Bettenburg et al. [25] implementaram o *infoZilla*, uma ferramenta que detecta e extrai de forma estruturada informações das descrições e comentários presentes em relatórios de defeitos, tais como pilhas de execução e trechos de código fonte. A ferramenta se utiliza de filtros que foram implementados pelos autores, a fim de detectar e extrair as informações desejadas.

Como forma de avaliar a ferramenta, foram processados 800 relatórios de defeitos do projeto Eclipse. Para cada relatório, a ferramenta foi capaz de identificar patches de código, pilhas de execução, trechos de código fonte e enumerações (que podiam significar passos para reproduzir o problema).

A ferramenta apresentada por Bettenburg et al poderá ser útil ao ser combinada com nossa técnica, em trabalhos futuros, de modo que se possa extrair estruturas presentes nos relatórios de defeitos e processar seu vocabulário de forma específica para cada estrutura.

Capítulo 7

Conclusão

Neste trabalho, apresentamos um estudo empírico que realizamos a fim de avaliar se a análise de vocabulários de software e de relatórios de defeitos pode ajudar na tarefa de localização de defeitos no código.

A abordagem proposta abrange a extração, tratamento e indexação do vocabulário de software e a utilização do vocabulário de relatórios de defeitos como *query* de busca no *index* criado. Em resumo, primeiro, extraímos o vocabulário de software e o tratamos e indexamos com técnicas de recuperação da informação. Depois, para cada relatório de defeito analisado, seu vocabulário também é extraído e tratado da mesma forma. Posteriormente, a similaridade de ambos os vocabulários é analisada. Como resultado, nossa abordagem retorna uma lista ranqueada de classes, ordenadas pela sua similaridade com relação ao vocabulário do relatório de defeito.

Realizamos uma avaliação da abordagem utilizando três versões do projeto Eclipse e cinco projetos da fundação Apache. Todos os projetos são implementados principalmente em Java e são amplamente utilizados na indústria de software e na academia. No total, mais de 9 mil relatórios de defeitos foram analisados e comparados com o vocabulário de mais de 27 mil classes provenientes dos oito projetos.

Investigamos cinco questões de pesquisa acerca de diversos tópicos, a saber: número de classes impactadas em geral por relatórios de defeitos; diversidade de classes modificadas para a correção de diferentes defeitos; taxa de acerto de nossa abordagem, a partir da variação dos tamanhos dos rankings; taxa de acerto da abordagem ao analisarmos apenas a primeira classe corretamente retornada no ranking; e o impacto de comentários de código para a

abordagem.

Os resultados mostram que, para os projetos analisados, cada relatório de defeito, em sua maioria, impacta um número pequeno de classes (geralmente, menos de quatro). Entretanto, ao analisarmos todas as classes impactadas para a resolução de todos os relatórios de defeitos, vimos que não existe um conjunto único de classes que é sempre impactado. Ou seja, cada relatório impacta poucas mas variadas classes. Isso significa que desenvolvedores precisam despender um grande esforço para localizar defeitos em um pequeno número de classes, dentre centenas ou milhares delas, presentes no projeto.

Mais ainda, foi possível perceber que similaridade de vocabulários é, de fato, útil para filtrar classes possivelmente impactadas por um relatório de defeitos. Por exemplo, um ranking retornado por nossa abordagem, contendo 15% das classes de um projeto, contém entre 75% a 95% de todas as classes impactadas por um relatório de defeito. Ou seja, ao utilizarmos apenas similaridade de vocabulários, podemos desconsiderar 85% das classes de um software e, ainda assim, obter em torno de 85% das classes defeituosas. Além disso, verificamos que, se considerarmos como *acerto* a recuperação de, pelo menos, uma classe defeituosa no ranking, os resultados são ainda melhores: se tomarmos como exemplo o projeto Eclipse 3.0, podemos desconsiderar 99,93% das classes e, ainda assim, encontrar 50% dos defeitos. Ou, mais ainda, encontraremos 75% dos defeitos ao considerarmos apenas 0,57% das classes do Eclipse.

Também vimos que, para os projetos analisados, ao adicionarmos comentários de código como parte integrante do vocabulário de software, teremos, no geral, uma queda na taxa de acerto de nossa abordagem. Portanto, é melhor para a nossa abordagem se considerarmos apenas identificadores do código como vocabulário de software.

Por fim, não encontramos um caso base apropriado com o qual podemos comparar nossa abordagem. Mais estudos precisam ser desenvolvidos no tocante a este assunto. Entretanto, acreditamos que nosso estudo possa ser utilizado em trabalhos posteriores como caso base para futuras comparações.

7.1 Contribuições

Em resumo, o presente trabalho apresenta as seguintes contribuições:

- Proposta de abordagem para o uso de vocabulário de software na tarefa de encontrar classes que são prováveis de serem impactadas durante a correção de um dado relatório de defeito;
- Relato da avaliação feita com mais de 9 mil relatórios de defeitos, provenientes de seis projetos Java maduros;
- Disponibilização dos resultados da avaliação, que podem ser utilizados como caso base para projetos futuros;
- Disponibilização, de forma estruturada (em arquivos XML), do vocabulário extraído dos seis projetos avaliados;
- Disponibilização do conjunto de dados dos relatórios de defeitos, mapeados para as classes que foram modificadas para a sua correção.

7.2 Considerações Finais

Sabemos que pesquisas sobre vocabulário de software ainda estão no início, quando comparadas a outras áreas. Entretanto, elas já se mostraram promissoras. Nosso estudo é um exemplo de que é possível aplicar análise de similaridade de vocabulário de software na prática. Todavia, esperamos que, à medida que o conhecimento e técnicas sobre vocabulário de software aumentem, nosso trabalho também possa ser expandido, alcançando uma solução ideal para a melhoria do estado da prática. Além disso, nosso estudo deve se beneficiar de outras pesquisas que visam a melhoria da qualidade dos relatórios de defeitos, especialmente a qualidade das descrições de defeitos.

Por fim, acreditamos que nosso trabalho possa ser considerado como passo inicial para pesquisas que usam vocabulário de software para avaliar soluções para o problema de localização de defeitos. Dessa maneira, esperamos ajudar a diminuir o alto custo do processo de manutenção de software.

Referências Bibliográficas

[1] MADHAVJI, N. H.; FERNANDEZ-RAMIL, J.; PERRY, D. *Software evolution and feedback: Theory and practice*. John Wiley & Sons, 2006.

[2] EISENBARTH, T.; KOSCHKE, R.; SIMON, D. Locating features in source code. *IEEE Transactions on Software Engineering*, Piscataway, NJ, USA, v. 29, n. 3, p. 210–224, 2003.

[3] CANFORA, G.; CERULO, L. Impact analysis by mining software and change request repositories. In: Anais do 11° IEEE International Software Metrics Symposium, 2005. METRICS '05. Washington, DC, USA: IEEE Computer Society. p. 29–37.

[4] FISCHER, M.; PINZGER, M.; GALL, H. Analyzing and relating bug report data for feature tracking. In: Anais da 10ª Working Conference on Reverse Engineering, 2003. WCRE '03. Washington, DC, USA: IEEE Computer Society. p. 90–99.

[5] ŚLIWERSKI, J.; ZIMMERMANN, T.; ZELLER, A. When do changes induce fixes? *SIGSOFT Software Engineering Notes*, New York, NY, USA, v. 30, n. 4, p. 1–5, 2005.

[6] NGUYEN, A. T.; NGUYEN, T. T.; AL-KOFAHI, J.; NGUYEN, H. V.; NGUYEN, T. N. A topic-based approach for narrowing the search space of buggy files from a bug report. In: Anais da 26ª IEEE/ACM International Conference on Automated Software Engineering, 2011. ASE '11. Washington, DC, USA: IEEE Computer Society. p. 263–272.

[7] ZHOU, J.; ZHANG, H.; LO, D. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In: Anais da International Conference on Software Engineering, 2012. ICSE 2012. Piscataway, NJ, USA: IEEE Press. p. 14–24.

[8] DAVIES, S.; ROPER, M.; WOOD, M. Using bug report similarity to enhance bug localisation. In: Anais da 19ª Working Conference on Reverse Engineering, 2012. p. 125–134.

[9] SCHEIN, A. I.; POPESCUL, A.; UNGAR, L. H.; PENNOCK, D. M. Methods and metrics for cold-start recommendations. In: Anais da 25ª International ACM SIGIR Conference on Research and Development in Information Retrieval, 2002. SIGIR '02. New York, NY, USA: ACM. p. 253–260.

[10] ANVIK, J.; HIEW, L.; MURPHY, G. C. Who should fix this bug? In: Anais da 28ª International Conference on Software Engineering, 2006. ICSE '06. New York, NY, USA: ACM. p. 361–370.

[11] MANNING, C. D.; RAGHAVAN, P.; SCHüTZE, H. *Introduction to information retrieval*. Cambridge, UK: Cambridge University Press Cambridge, 2008.

[12] PORTER, M. F. Readings in information retrieval. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. Cap. An algorithm for suffix stripping, p. 313–316.

[13] BAEZA-YATES, R. A.; RIBEIRO-NETO, B. *Modern information retrieval*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[14] HAIDUC, S.; MARCUS, A. On the use of domain terms in source code. In: Anais da 16ª IEEE International Conference on Program Comprehension, 2008. ICPC '08. Washington, DC, USA: IEEE Computer Society. p. 113–122.

[15] ENSLEN, E.; HILL, E.; POLLOCK, L.; VIJAY-SHANKER, K. Mining source code to automatically split identifiers for software analysis. In: Anais da 6ª IEEE International Working Conference on Mining Software Repositories, 2009. MSR '09. Washington, DC, USA: IEEE Computer Society. p. 71–80.

[16] BUTLER, S.; WERMELINGER, M.; YU, Y.; SHARP, H. Relating identifier naming flaws and code quality: An empirical study. In: Anais da 16ª Working Conference on Reverse Engineering, 2009. WCRE '09. Washington, DC, USA: IEEE Computer Society. p. 31–35.

[17] SNEED, H. M. Object-oriented cobol recycling. In: Anais da 3ª Working Conference on Reverse Engineering, 1996. WCRE '96. Washington, DC, USA: IEEE Computer Society. p. 169–178.

[18] CAPRILE, B.; TONELLA, P. Nomen est omen: Analyzing the language of function identifiers. In: Anais da 6ª Working Conference on Reverse Engineering, 1999. WCRE '99. Washington, DC, USA: IEEE Computer Society. p. 112–122.

[19] BINKLEY, D.; LAWRIE, D. Information retrieval applications in software maintenance and evolution. *Encyclopedia of Software Engineering*, 2009.

[20] SANTOS, K.; SEREY, D.; FIGUEIREDO, J.; BITTENCOURT, R. Towards a prediction model for source code vocabulary, 2012. Anais do 1º Workshop on The Next Five Years of Text Analysis in Software Maintenance, ocorrido em paralelo ao IEEE ICSM '12.

[21] CAVALCANTI, D.; SANTOS, K.; SEREY, D.; FIGUEIREDO, J. Using software vocabulary to rank classes that are probably impacted by a bug report, 2012. Anais do 1º Workshop on The Next Five Years of Text Analysis in Software Maintenance, ocorrido em paralelo ao IEEE ICSM '12.

[22] CANFORA, G.; CERULO, L. Fine grained indexing of software repositories to support impact analysis. In: Anais do International Workshop on Mining Software Repositories, 2006. MSR '06. New York, NY, USA: ACM. p. 105–111.

[23] MATTER, D.; KUHN, A.; NIERSTRASZ, O. Assigning bug reports using a vocabulary-based expertise model of developers. In: Anais da 6ª IEEE International Working Conference on Mining Software Repositories, 2009. MSR '09. Washington, DC, USA: IEEE Computer Society. p. 131–140.

[24] CUBRANIC, D.; MURPHY, G. C. Automatic bug triage using text categorization. In: Anais da 16ª International Conference on Software Engineering & Knowledge Engineering, 2004. Editors MAURER, F.; RUHE, G. p. 92–97.

[25] BETTENBURG, N.; PREMRAJ, R.; ZIMMERMANN, T.; KIM, S. Extracting structural information from bug reports. In: Anais da 5ª International Working Conference

on Mining Software Repositories, 2008. MSR '08. New York, NY, USA: ACM. p. 27–30.

[26] MOIN, A.; KHANSARI, M. Bug localization using revision log analysis and open bug repository text categorization. In: *Open Source Software: New Horizons*. Springer Berlin Heidelberg, 2010. v. 319 of *IFIP Advances in Information and Communication Technology*, p. 188–199.

[27] ZIMMERMANN, T.; PREMRAJ, R.; ZELLER, A. Predicting defects for eclipse. In: Anais do 3º International Workshop on Predictor Models in Software Engineering, 2007.

[28] RIEL, A. J. *Object-oriented design heuristics*. 1st. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996. Cap. 3.

[29] HASSAN, A. E.; HOLT, R. C. Predicting change propagation in software systems. In: Anais da 20ª IEEE International Conference on Software Maintenance, 2004. ICSM '04. Washington, DC, USA: IEEE Computer Society. p. 284–293.

[30] HAN, J. Supporting impact analysis and change propagation in software engineering environments. In: Anais do 8º International Workshop on Software Technology and Engineering Practice, 1997. STEP '97. Washington, DC, USA: IEEE Computer Society. p. 172–182.

[31] QUEILLE, J.-P.; VOIDROT, J.-F.; WILDE, N.; MUNRO, M. The impact analysis task in software maintenance: a model and a case study. In: Anais da International Conference on Software Maintenance, 1994. p. 234–242.

[32] KO, A. J.; MYERS, B. A.; CHAU, D. H. A linguistic analysis of how people describe software problems. In: Anais do Visual Languages and Human-Centric Computing, 2006. VLHCC '06. Washington, DC, USA: IEEE Computer Society. p. 127–134.

[33] ABEBE, S. L.; HAIDUC, S.; MARCUS, A.; TONELLA, P.; ANTONIOL, G. Analyzing the evolution of the source code vocabulary. In: Anais da European Conference on Software Maintenance and Reengineering, 2009. CSMR '09. Washington, DC, USA: IEEE Computer Society. p. 189–198.

[34] ANTONIOL, G.; CANFORA, G.; CASAZZA, G.; DE LUCIA, A. Identifying the starting impact set of a maintenance request: A case study. In: Anais da Conference on Software Maintenance and Reengineering, 2000. CSMR '00. Washington, DC, USA: IEEE Computer Society. p. 227–230.

[35] CRISTIANINI, N.; SHAWE-TAYLOR, J. *An introduction to support vector machines and other kernel-based learning methods*. New York, NY, USA: Cambridge University Press, 2000.

[36] BLEI, D. M.; NG, A. Y.; JORDAN, M. I. Latent dirichlet allocation. *The Journal of Machine Learning Research*, v. 3, p. 993–1022, Mar. 2003.

[37] ANTONIOL, G.; CANFORA, G.; CASAZZA, G.; DE LUCIA, A.; MERLO, E. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, Piscataway, NJ, USA, v. 28, n. 10, p. 970–983, 2002.

[38] CUBRANIC, D.; MURPHY, G. C. Hipikat: recommending pertinent software development artifacts. In: Anais da 25ª International Conference on Software Engineering, 2003. ICSE '03. Washington, DC, USA: IEEE Computer Society. p. 408–418.

Apêndice A

Script para Download de Relatórios do Bugzilla

O script abaixo (Código Fonte A.1) foi utilizado para o download de relatórios de defeitos do projeto Eclipse, que utiliza o Bugzilla como sistema de rastreamento de defeitos.

Código Fonte A.1: Script para Download de Relatórios do Bugzilla

```
#!/bin/bash

# VARIABLES
URL="https://bugs.eclipse.org/bugs/show_bug.cgi?ctype=xml&id="
IDS=(lista-de-ids-de-relatorios-de-defeitos)
OUTPUT="/home/diegotc/bugzilla/"

# DOWNLOADING...
for (( i = 0 ; i < ${#IDS[@]}; i++ )) do
    wget -q -O $OUTPUT"${IDS[${i}]}".xml" \
      --no-check-certificate $URL"${IDS[${i}]}
done
```

Como é possível perceber, a variável IDS deve conter uma lista com os IDs dos relatórios de defeitos que precisam ser baixados. Ou seja, para cada versão do Eclipse que processamos, adicionamos ao script os IDs dos relatórios baixados.

Apêndice B

Stop Words Utilizadas na Avaliação

Neste apêndice, apresentamos a lista das *stop words* (Seção 2.2) utilizadas para a avaliação da abordagem proposta.

Primeiro, listamos as *stop words* pré-definidas da ferramenta Apache Lucene:

- a
- an
- and
- are
- as
- at
- be
- but
- by
- for
- if
- in
- into
- is
- it
- no
- not
- of
- on
- or
- such
- that
- the
- their
- then
- there
- these
- they
- this
- to
- was
- will
- with

Além das supracitadas, também definimos algumas *stop words* que aparecem muito em

descrições de relatórios de defeitos e não agregam nenhum valor (por exemplo, *NullPointerException*); ou que são específicas para o nosso trabalho, no qual processamos código fonte e relatórios de defeitos do projeto Eclipse e projetos da fundação Apache:

- all
- any
- apache
- boolean
- bug
- char
- class
- double
- duplicate
- eclipse
- ejb
- error
- every
- exception
- final
- float
- foo
- get
- hadoop
- have
- int
- issue
- ivy
- java
- jpa
- lucene
- must
- need
- never
- npe
- null
- open
- openejb
- openjpa
- other
- pivot
- problem
- public
- same
- set
- shiro
- should
- static
- string
- while

Apêndice C

Representação de Vocabulário de Software em Arquivos XML

Em [20], a ferramenta *VocabularyTools* foi desenvolvida para extração, tratamento e análise de vocabulário de software. Dado que o nosso trabalho era na mesma linha de pesquisa e que precisaríamos de uma ferramenta similar para a extração e armazenamento do vocabulário, trabalhamos em conjunto e cooperamos para projetar e implementar a tal ferramenta.

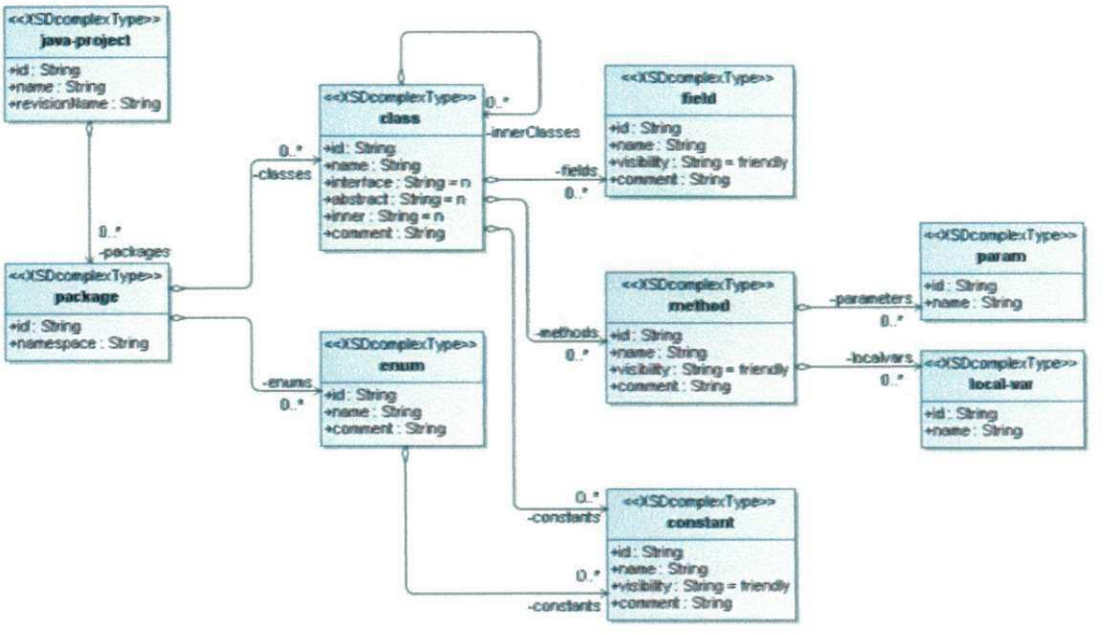
Dentre outras funcionalidades, o *VocabularyTools* utiliza a árvore sintática abstrata do código Java para extrair o seu vocabulário de software e, então, exportá-lo para um formato XML, o qual chamamos de VXL (*Vocabulary XML*).

No VXL, são representadas características importantes do código para a área de vocabulário de software. Como é possível perceber no esquema da Figura C.1, para cada elemento estrutural do projeto, além do nome de seus identificadores, armazenamos comentários e variáveis indicando se as estruturas são internas e/ou abstratas, além de seus comentários e sua visibilidade.

A Figura C.2 apresenta a representação em VXL de um projeto Java. Como é possível perceber, estão representadas uma interface, uma enumeração e uma classe Java, todas em um mesmo pacote, contendo constantes, métodos, parâmetros e variáveis locais.

Portanto, é possível perceber que o VXL foi criado para facilitar o armazenamento de vocabulário de software. Sendo assim, para a sua leitura, basta que se crie um *parser* de XML que processe os atributos pré-definidos.

Figura C.1: Esquema de Representação de Vocabulário em VXL



Fonte: Relatório Técnico de Santos et al. [20].

Figura C.2: Exemplo de Vocabulário Representado em VXL

```
▼<java-project id="name" name="Java Project" revision="1.0">
  ▼<package id="package0001" namespace="java.model">
    ▼<class id="class0001" name="JavaInterface" interface="y" abstract="n" inner="n" comment="">
      ▼<method id="method0001" name="javaMethod" visibility="public" comment="">
          <param id="parameter0001" name="javaParameter"/>
        </method>
      </class>
    ▼<enum id="enum0001" name="JavaEnum" comment="">
        <constant id="attribute0001" name="J" visibility="public" comment=""/>
        <constant id="attribute0002" name="A" visibility="public" comment=""/>
      </enum>
    ▼<class id="class0002" name="JavaClass" interface="n" abstract="n" inner="n" comment="">
      ▼<method id="method0002" name="javaMethod" visibility="public" comment="">
          <param id="parameter0002" name="javaParameter"/>
          <local-var id="locvariable0001" name="javaLocalVariable"/>
        </method>
      </class>
  </package>
</java-project>
```

Fonte: Relatório Técnico de Santos et al. [20].