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

Investigação sobre Uso de Vocabulário de Código Fonte para Identificação de Especialistas

Katyusco de Farias Santos

Tese 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 Doutor em Ciência da Computação.

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

Orientadores
Dr. Dalton Dario Serey Guerrero
Dr. Jorge César Abrantes de Figueiredo

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

S237i Santos, Katyusco de Farias.
Investigação sobre uso de vocabulário de código fonte para identificação de especialistas / Katyusco de Farias Santos. – Campina Grande, 2015.
137f. : il. color.

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

"Orientação: Prof. Dr. Dalton Dario Serey Guerrero. Prof. Dr. Jorge César Abrantes de Figueiredo".
Referências.

1. Vocabulário de Software. 2. Especialista de Código. 3. Código Fonte. I. Guerrero, Dalton Dario Serey. II. Figueiredo, Jorge César Abrantes de. III. Título.

CDU 004.4(043)

"INVESTIGAÇÃO SOBRE USO DE VOCABULÁRIO DE CÓDIGO FONTE PARA IDENTIFICAÇÃO DE ESPECIALISTAS"

KATYUSCO DE FARIAS SANTOS

TESE APROVADA EM 10/03/2015

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

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

UIRA KULESZA, Dr., UFRN
Examinador(a)

LEANDRO BALBY MARINHO, Dr., UFCG
Examinador(a)

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

ROBERTO ALMEIDA BITTENCOURT, Dr., UEFS
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Identificadores e comentários de um código fonte constituem o vocabulário de software. Pesquisas apontam vocabulários como uma fonte valorosa de informação sobre o projeto. Para entender a natureza e o potencial dos vocabulários, desenvolvemos um ferramental capaz de extraí-los a partir de código fonte. Explorando os dados estatisticamente, identificamos duas propriedades de vocabulários: tamanho, expresso como função de potência de *LOC* (*Lines-Of-Code*); e a repetição de seus termos, que se ajusta a uma distribuição *log-normal*. Vocabulários, bem como suas propriedades e operações foram formalizadas baseadas no conceito de *multisets*. O ferramental de extração e a formalização viabilizaram cooperações científicas sobre a utilidade de vocabulários em atividades de manutenção. Esse conhecimento acumulado revelou que vocabulário pouco foi explorado como insumo à modelagem de conhecimento de código. Desenvolvemos então uma abordagem para identificar especialistas de código cujo conhecimento é definido pela similaridade existente entre vocabulários das entidades e dos desenvolvedores. Comparamos a precisão e cobertura da nossa abordagem com de duas outras: baseada em *commits* e baseada em percentual de *LOC* modificadas. Os resultados apontam que para indicar um único especialista, *top-1*, a nossa abordagem tem uma precisão menor, entre 29.9% e 10% que as abordagens de *baseline*. Já para indicar mais de um desenvolvedor especialista, até *top-3*, a nossa abordagem tem uma acurácia melhor de até 18.7% em relação as de *baseline*. Identificamos também que o conhecimento definido por similaridade quando combinado com um modelo baseado em autoria aumenta a capacidade de identificar especialistas, no R^2 do modelo, em mais de 4 pontos percentuais. Concluímos que além de poder ser utilizado de forma isolada para modelar conhecimento de código e assim identificar especialistas, o vocabulário pode ser um componente adicional a modelos de conhecimento baseados em autoria e propriedade, já que capturam aspectos diferentes dos existentes nesse modelos.

Abstract

Identifiers and comments from a source code are the software vocabulary. Research point vocabularies as a valuable source of information about the project. To understand we developed a tool that extract them from source code. Exploring the data statistically, we identify two vocabularies properties: vocabulary size, that is a power function of *LOC* (*Lines-Of-Code*) and the repetition of vocabulary terms that fits a *log-normal* distribution. Vocabulary as well as their properties and operations were formalized based on the concept of *multisets*. Extraction tool and formalization made possible scientific cooperation on usage of vocabulary in maintenance activities. This accumulated knowledge has shown that vocabulary was not explored as an input to code knowledge. Then we developed a code experts identification approach whose knowledge is defined by existing similarity between entities and developers vocabularies. We compared precision and recall with two baseline approaches: based on *commits* and based on percentage of modified *LOC*. The results show that to indicate a single specialist, *top-1*, our approach has a lower precision, between 29.9% and 10%, than *baseline* approaches. More than one specialist-developer, up to *top-3*, our approach has better accuracy of up to 18.7% over those of the *baselines*. We also identify that the knowledge defined by similarity when combined with an authorship model enhances the ability to identify experts, R^2 of the model, by more than 4 points. We conclude that vocabulary can be solely used to expertise, and thus identify experts. In addition, vocabulary can be an additional component for models based on authorship and ownership, since it captures different aspects from ones existing in these models.

Agradecimentos

Agradeço a todos que direta ou indiretamente me incentivaram a chegar no estágio de escrever e defender esta Tese, um livro complicado segundo minha filha de 7 anos.

Em especial, ao meu núcleo familiar: meus filhos Laís, e Artur de um ano e meio, e a minha esposa Elaine. Também, ao meus pais, irmã, tios e primos que sempre me "cobravam" dizendo: "Quero lhe ver Doutor!!!".

Dalton Serey e Jorge Abrantes, meus orientadores, obrigado pelo suporte e compreensão, além dos ensinamentos críticos-científicos.

Claro que sem aquela ajuda do grupo de trabalho, talvez esta Tese não tivesse nem sido iniciada ou nem fizesse nem sentido. São muitos nomes, e eu não quero correr o risco de esquecer alguém. Então agradeço à todos do SPLab pelas parcerias, discussões, sugestões.

Por último, o meu mais importante agradecimento: a Deus, inteligência suprema e causa primária de tudo. É.... inexoravelmente a gente chega lá!

Conteúdo

Lista de Símbolos

AC - *ACceptances*

AST - *Abstract Syntax Tree*

CFV - *Characteristic Frequency Vector*

DL - *DeLiveries*

DOA - *Degree-Of-Authorship*

DOE - *Degree-Of-Eloquence*

DOI - *Degree-Of-Interest*

DOK - *Degree-Of-Knowledge*

FA - *First Authorship*

IR - *Information Retrieval*

LOC - *Lines Of Code*

SPLab - *Software Practices Laboratory* - Laboratório de Práticas de Software

SVN - *SubVersion*

VCS - *Version Control System*

VXL - *Vocabulary eXtended Language*

XML - *eXtended Markup Language*

ePol - Sistema de Informações da Polícia Federal Brasileira

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Capítulo 1

Introdução

1.1 Contextualização

Estudos sobre identificadores e comentários, que constituem o vocabulário de software, apontam que eles capturam aspectos subjetivos e inerentes de quem os codifica ou os escreve, tais como: quantidade mínima e máxima de palavras para constituir um identificador, predileção por usar determinados termos ou palavras. Esses aspectos são diferentes daqueles que são tradicionalmente capturados pelas análises estáticas (baseadas em elementos e relações estruturais) e dinâmicas (rastreando a execução do sistema) [Lawrie et al., 2006; Arnaoudova et al., 2010]. O vocabulário do software representa cerca de 70% do código fonte [Deissenboeck and Pizka, 2006], onde seus identificadores em parte, são nomeados de acordo com conceitos, abstrações e funcionalidades que representam perante o sistema [Abebe et al., 2009]. Além disso, o vocabulário pode ser utilizado para avaliar a qualidade do código [Host and Ostvold, 2007; Butler et al., 2010], fornece indícios sobre as preferências pessoais [Haiduc and Marcus, 2008] e reflete o mapa mental dos desenvolvedores [Guerrouj, 2010].

De acordo com a literatura, vocabulário é uma fonte valorosa de informação sobre o projeto (processo e produto) que deveria ser utilizada para auxiliar *stakeholders* durante as atividades de desenvolvimento e manutenção de sistemas [Binkley and Lawrie, 2009; Binkley and Lawrie, 2011]. Para tentar entender a natureza dos vocabulários, em outras palavras, a origem da sua formação e como se revelam as informações por eles carregadas, adotamos inicialmente uma abordagem estatística.

Para isso, executamos estudos estatísticos exploratórios sobre os vocabulários de uma amostra composta de 39 códigos fontes de projetos Java *open source*. Esses estudos nos permitiram identificar e entender duas propriedades: tamanho de vocabulário e a ocorrência de seus termos ao longo do código. Relacionamos tamanho de código fonte, em *LOC* (*Lines-Of-Code*), com cada uma das propriedades de vocabulário, e as expressamos através de modelos de regressão.

Em particular, as evidências estatísticas indicaram que tanto o quantitativo de termos distintos, únicos, quanto o total de termos, são modelados como uma função de potência sobre o tamanho em *LOC* do sistema. Já as repetidas ocorrências de cada termo de um vocabulário é regida por uma distribuição Cauda Longa do tipo log-normal, onde tanto sua média, μ, como o seu desvio padrão, σ, também são expressos em função do *LOC* do sistema.

Fizemos uso de uma formalização com intuito de caracterizar nossa concepção e definir sem ambiguidades vocabulários, suas propriedades e operações. Também, para facilitar a nossa expressividade e promover a reprodução de nossos experimentos por outros pesquisadores. Parte dos nossos resultados estão publicados em:

- *Understanding the Occurrence of Vocabulary Terms in Java Code* publicado no WMSWM (*Workshop* de Manutenção de Software Moderno) do SBQS (Simpósio Brasileiro de Qualidade de Software) de 2011, Curitiba - Brasil [Santos et al., 2011]. Apêndice B.

- *Towards a Prediction Model for Source Code Vocabulary*. *Workshop Next Five Years of Text Analysis in Software Maintenance (TAinSM)*, evento satélite ao *International Conference Software Maintenance (ICSM 2012)* [Santos et al., 2012], Trento - Itália. Apêndice C.

Para dar suporte à execução desses estudos empíricos, desenvolvemos o *Vocabulary Tool*, ferramental capaz de extrair, a partir de código fonte java, identificadores, comentários e *javadoc* (vide Apêndice A). Como essas informações são armazenadas num formato baseado em XML (*eXtended Markup Language*), elas podem ser facilmente utilizadas para realizar outros estudos sobre vocabulário de sistemas, ou combinadas com outras medidas de processo e de produto.

O ferramental e o conhecimento adquirido sobre vocabulários nos permitiu experimentar

linhas de investigações já conhecidas na literatura que se utilizavam do potencial de vocabulário. Esse fato é comprovado pelos estudos que fizeram uso do ferramental e com os quais colaboramos. Um primeiro estudo, indexou o vocabulário extraído do código fonte e sobre ele pesquisou o conteúdo de relatórios de defeitos (*bugs*), com o objetivo de localizar *bugs* no código. Parte dos resultados dessa investigação encontra-se em:

- *Using Software Vocabulary to Rank Classes that are Probably Impacted by a Bug Report*. *TAinSM-ICSM 2012* [Cavalcanti et al., 2012], Trento - Itália. Apêndice C.

Uma outra colaboração se deu no uso de técnicas de recuperação de informação e agrupamentos hierárquicos para mostrar como os domínios de conceitos estão dispostos em entidades de código conceitualmente coesas, *i.e.* que compartilham termos entre seus vocabulários, e como evoluem entre versões de sistemas. Detalhes sobre esse trabalho está em:

- *TopicViewer : Evaluating Remodularizations Using Semantic Clustering*. *CBSoft2015: Teoria e Prática* [Santos et al., 2013], Brasília - Brasil. Apêndice D.

A sinergia entre a compreensão sobre a natureza dos vocabulários, através de exploração estatística, e a colaboração com os trabalhos acima citados, endossou o potencial de vocabulário para auxiliar nas mais diversas atividades de manutenção de sistemas [Pollock et al., 2013]. Contudo, na literatura revisada, relatada na Seção 6.1, pouco têm sido exploradas as informações contidas nos vocabulários de código com o objetivo de recuperar automaticamente desenvolvedores especialistas, até onde temos conhecimento. Nesta Tese então, focamos em investigar vocabulário como insumo a modelos de conhecimento sobre o código fonte.

1.2 Motivação

Possibilitar que desenvolvedores encontrem de forma eficiente (rápida e precisa) os pontos no código fonte em que devem atuar para executar uma tarefa de codificação é um desafio constante nos projetos de sistemas. Mas, para que a mudança seja concretizada com o menor esforço (custo e tempo) é preciso também definir quem, entre os membros de uma equipe, é o desenvolvedor mais adequado para realizá-la.

Os custos com manutenção representam entre 70% e 90% do total de um projeto de software [Bennett and Rajlich, 2000; Boehm and Basili, 2001; Buse and Weimer, 2010]. Na manutenção, entender o código toma de 50% [Bennett and Rajlich, 2000] até 78% do tempo do programador, enquanto que corrigir *bugs* representa 20% e escrever código apenas 2%, segundo Hallman [Hallam, 2006]. Identificar, de forma precisa, os membros de uma equipe de desenvolvimento que são *experts* (especialistas no código) em cada parte do sistema favorece a compreensão do código que é uma atividade que não só depende de propriedades do código, mas também de uma documentação atualizada e consistente [Moreno, 2014], e da experiência de quem a executa [Feigenspan et al., 2011].

Durante o desenvolvimento de um projeto, trechos de código criados por um desenvolvedor podem ser modificados por outros desenvolvedores contribuintes. As contribuições, mudanças realizadas, sobre um código fonte podem ser tantas que o especialista que antes era o autor do código, após as mudanças, passa ser aquele que foi um dos contribuintes [Fritz et al., 2010]. É papel então das estratégias de identificação de especialistas auxiliar os responsáveis por alocar tarefas indicar entre os desenvolvedores aquele que é o atual especialista sobre um dado trecho de código.

Outra motivação para identificar especialistas é evitar que nos projetos, quer tenham baixa ou alta rotatividade de desenvolvedores, existam partes de código não dominadas pela atual equipe [Cavalcanti et al., 2014]. Realizar tarefas de manutenção em trechos de código sem especialistas podem tomar um tempo inaceitável para clientes, gerando até multas para fornecedores de sistemas. Verificar com uma abordagem que identifica especialistas se para cada parte do projeto, módulos, classes ou métodos, há pelo menos um desenvolvedor da equipe atual que tem sobre a parte conhecimento, aumenta a probabilidade de que uma tarefa de manutenção, independentemente de sua localização no código, será realizada num custo de esforço previsível.

Projetos onde partes do código são conhecidas por um único desenvolvedor correm o risco de se tornarem inviáveis financeiramente ou até mesmo descontinuados caso o desenvolvedor se torne indisponível, por exemplo mude de emprego ou entre de licença. Do mesmo modo, empresas contratantes de projetos de sistemas que evitem que seus código fontes sejam dominados apenas por colaboradores de um único fornecedor de serviços de desenvolvimento reduzem o risco de terem que aceitar renovações de contratos de desenvol-

vimento com valores sobretaxados. Empresas que sejam convictas que são as únicas capazes de dar manutenção em sistemas de clientes tem o potencial de cobrar a esses preços maiores que os praticados no mercado já que sabem da inexistência de concorrência.

Conhecer os especialistas de um sistema possibilita também que, para uma dada mudança já realizada em uma classe, se possa definir quem são os demais membros da equipe de desenvolvimento capazes de executar uma revisão sobre o código da mudança. Permite também que desenvolvedores que precisem se familiarizar com um código ainda desconhecido saibam a quem se dirigir no caso de dificuldades. Ou ainda, possibilita que os desenvolvedores tenham conhecimento dos seus pares com quem devem colaborar durante o ciclo de vida de construção de sistemas [Minto and Murphy, 2007].

1.3 Definição do Problema

É natural que trechos de código criados inicialmente por um desenvolvedor, no decorrer do projeto, recebam contribuições de código oriundas de outros desenvolvedores colaboradores [Fritz et al., 2010]. São mudanças que para serem realizadas precisam que pelo menos o vocabulário no seu entorno, seja lido e compreendido [Buse and Weimer, 2010]. Uma mudança pode implicar também em transformações sobre o próprio vocabulário: desde a repetição de um termo já existente num outro trecho do código, até a substituição de todas as ocorrências de um dado termo para um novo, mais adequado segundo as concepções do colaborador que executa a mudança (*e.g.*: um *renaming*).

As abordagens mais utilizadas para identificar especialistas têm como princípio minerar informações de autoria e propriedade dos gerenciadores de código (*VCS - Version Control System*). Por exemplo, uma delas consiste na extração do número de *commits* realizados sobre os arquivos de código fonte, ponderando por cada um dos membros da equipe de desenvolvimento [Mockus and Herbsleb, 2002; Hattori and Lanza, 2009]. Uma segunda abordagem define o especialista de uma entidade em função do percentual de LOC (*Lines of Code*) modificadas por cada membro da equipe [Girba et al., 2005; Rahman and Devanbu, 2011]. De forma imprecisa ambas abordagens atribuem créditos de conhecimento a desenvolvedores que apenas indentam um código fonte sem se sequer o terem lido. Esse tipo de erro se agrava quando lembramos que a funcionalidade de mu-

danças no estilo de indentação, de forma automática, é facilmente encontrada na maioria das ferramentas de desenvolvimento atuais. Contudo, mudanças na indentação não afetam o vocabulário.

Da perspectiva das abordagens baseadas em autoria e propriedade, toda entidade tem sempre seu respectivo especialista, mesmo que ela tenha sido manipulada apenas uma vez: na sua criação pelo seu autor. Se por algum motivo, o autor de uma dada entidade desfalca a equipe, essa entidade estaria órfã [Bittencourt et al., 2010], *i.e.* sem um especialista na equipe de desenvolvimento por ela responsável. Da perspectiva de vocabulário, quanto mais as entidades de um sistema compartilham termos em comum, mais conceitualmente similares elas são [Kuhn et al., 2007; Corley et al., 2012; Santos et al., 2013] Logo, uma entidade órfã poderia ser adotada por um outro desenvolvedor (especialista de outra entidade) cuja similaridade de seu vocabulário fosse o próximo possível ao vocabulário da entidade órfã. Essa mesma estratégia poderia ser utilizada para recomendar desenvolvedores a atuarem em entidades em que eles não são os maiores especialistas, para por exemplo balancear carga de tarefas de manutenção;

Analisando como se comportam identificadores e comentários em cenários como esses descritos nos parágrafos anteriores, identificamos que vocabulários capturam conhecimento de código diferentes daqueles capturados por outras abordagens que se utilizam de informações de autoria e propriedade.

Então, definida nos fontes a localização, uma entidade, onde deve acontecer uma tarefa de codificação, mudança ou revisão de código, utilizando vocabulário de software, o nosso problema é identificar entre os desenvolvedores de um projeto aqueles que sejam os mais adequados para concretizarem a tarefa, os especialistas da entidade. Além disso, tentamos melhorar a precisão e a cobertura da identificação em relação a outras abordagens. as

1.4 Solução Proposta

Cada desenvolvedor tem um vocabulário próprio, denominado de **vocabulário do desenvolvedor**, constituído pelo acúmulo dos termos por ele manipulado (adicionado, substituído ou removido) ao longo de suas contribuições sobre o projeto [Matter et al., 2009]. Cada entidade de código tem também seu vocabulário próprio, **vocabulário de entidade**, que é constituído

pelos termos adicionados pelos desenvolvedores que contribuíram, em algum momento, com o código fonte da entidade. Diferente do vocabulário do desenvolvedor, para o vocabulário da entidade é relevante apenas o seu estado atual. Porque, de modo simplificado, para se realizar uma mudança sobre uma entidade exige-se compreendê-la apenas como ela se encontra num dado instante do tempo, e não em cada uma de suas versões que compõem a sua história.

Desenvolvemos então, uma abordagem inovadora, até onde temos conhecimento, baseada na similaridade entre os vocabulários de código fonte e de desenvolvedores para identificar especialistas de entidades de código. De forma breve, a abordagem varre o repositório de código fonte e a cada modificação detectada acumula, como vocabulário de cada desenvolvedor, as contribuições realizadas por cada um deles no vocabulário do código fonte. Para uma dada versão de código fonte do projeto, extraímos seu vocabulário de código fonte, e o granularizamos por entidade de código, classes e interfaces. Identificar o especialista consiste em encontrar o desenvolvedor cujo vocabulário tem a maior similaridade com o vocabulário de uma dada entidade sobre a qual uma tarefa de codificação foi previamente localizada. Adicionalmente, para entidades órfãs, a abordagem usa a similaridade para recomendar entre os desenvolvedores aquele que tem com maior potencial para mais facilmente adotar a entidade.

1.5 Questões de Pesquisa

É preciso averiguar na prática o potencial da abordagem proposta na identificação de especialistas considerando cenários corriqueiros de atribuição de conhecimento. Formulamos então as seguintes questões de:

QP_1: **Nossa abordagem consegue identificar especialistas de código?**

QP_2: **Comparadas com as 2 abordagens de *baseline*, a precisão e a cobertura da nossa abordagem são melhores?**

Para responder a questões de pesquisa, comparamos a abordagem proposta com duas outras técnicas de identificação de especialistas já utilizadas na literatura: por *commits* [Mockus and Herbsleb, 2002] e por percentual de *LOC* modificadas [Girba et al., 2005]. Para isso, construímos um oráculo de especialistas para uma amostra de entidades de um projeto real

que utilizamos como estudo de caso: o ePol[1]. A acurácia das abordagens foi medida em termos de precisão e de cobertura em relação ao oráculo construído. Com uma precisão de até 0.3214 para a identificação de um único especialista, e com precisão de até 0.1905 e cobertura de até 0.3478 num cenário com até 3 especialistas recuperados, comprovamos que a abordagem proposta de fato identifica especialistas de código. E, mais especificamente, os resultados comparativos concluem que para indicar um único especialista, *top-1*, as abordagens de *baseline* oferecem apenas entre 29,9% e 10% a mais de precisão. Já para indicar mais de um desenvolvedor especialista, até *top-3*, a nossa abordagem tem uma acurácia melhor de até 18.7% em relação as de *baseline*.

Em decorrência de serem respondidas QP_1 e QP_2, novas questões de pesquisa surgem, agora sobre o modelo de conhecimento definido através da similaridade:

QP_3: **A similaridade entre vocabulários de entidades e de desenvolvedores carrega aspectos inerentes para identificar especialistas não capturados pelos elementos de autoria e propriedade utilizados em abordagens tradicionais?**

QP_4: **Qual o percentual da contribuição do modelo de conhecimento definido pela similaridade de vocabulários na identificação de especialistas?**

Para respondê-las, adicionamos ao modelo *Degree-Of-Knowledge - DOK* [Fritz et al., 2010] a componente de conhecimento utilizado pela nossa abordagem: similaridade entre vocabulários. Submetemos a amostra de entidades que compõem o Oráculo de especialistas tanto ao modelo *DOK* com suas componentes de autoria originais, quanto ao adicionado com a similaridade entre vocabulários, e encontramos seus respectivos modelos de regressão. Todos os modelos encontrados para *DOK* com suas componentes de autoria originais, obtiveram Coeficiente de Determinação, R^2, menores que os dos seus respectivos modelos de *DOK* acrescentados da similaridade entre vocabulários, revelando assim que a similaridade carrega aspectos próprios de conhecimento de código diferente dos existes em autoria. Explorados e comparados estatisticamente apenas os modelos válidos, quantificamos que o conhecimento de código capturado de vocabulários incrementa em mais de 4 pontos percentuais a capacidade dos modelos em identificar especialistas.

[1]Sistema de Informações da Polícia Federal Brasileira

1.6 Organização do Documento

O restante do documento que compõe esta Tese está organizado da seguinte forma. No Capítulo 2, relatamos conceitos sobre vocabulário e conhecimento de código encontrados na literatura que julgamos necessários para fundamentar o leitor nos demais capítulos. A natureza dos vocabulários e uma formalização para eles, são contribuições nossas decorrentes de experimentos próprios e de colaborações que são relatados no Capítulo 3. No Capítulo 4, detalhamos uma abordagem inovadora para identificação de especialistas de código baseada na similaridade aferida entre vocabulários de desenvolvedores e de entidades de código.

Já no Capítulo 5 detalhamos dois experimentos com foco na avaliação. O objetivo do primeiro deles, Seção 5.1, é mensurar a acurácia da nossa abordagem de identificação de especialistas e a comparamos com duas outras de *baseline*: baseada no número de *commits* e baseada no percentual de *LOC* modificadas. Já no segundo experimento, Seção 5.2, buscamos identificar se conhecimento de código mapeado através de vocabulários capturam aspectos diferentes dos existentes nos modelos baseados em autoria e propriedade. Consideramos como trabalhos relacionados, no Capítulo 6, aqueles que abordam o entendimento e o uso de vocabulário, e os que tratam de conhecimento de código. Finalmente, no Capítulo 7 apresentamos nossas conclusões e discussões sobre nossas investigações, além de uma série de desdobramentos para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo são apresentados os conceitos fundamentais, presentes na literatura, sobre vocabulário e sobre conhecimento de código com o objetivo de facilitar o entendimento do restante deste documento de Tese.

2.1 Conceitos Fundamentais

Software é um artefato conceitual imaterial concebido para automatizar tarefas difíceis e de custo elevado, em esforço e tempo, para serem executadas manualmente por seres humanos. Conceitos e relações de um software são expressos primeiro como código fonte, que é então convertido para uma representação de baixo nível conhecido como código executável. Para expressar suas ideias, em código fonte, os desenvolvedores se utilizam de uma linguagem de programação que assim como uma linguagem natural em texto, é representada por cadeias de símbolos (caracteres, *tokens*, sentenças) que servem como meio de comunicação [Kokol and Podgorelec, 1999].

2.1.1 Níveis de Comunicação para Codificação

Um código fonte tem dois níveis de comunicação [Kuhn et al., 2007], são eles:

- Nível **homem-máquina**, que é realizada através das instruções de programas que são escritas de acordo com rigorosas regras gramaticais formais (gramáticas livres de contexto e regulares). Nas instruções estão inseridas as palavras chaves e reservadas, e os

operadores da linguagem;

- Nível **homem-homem**, que ocorre por meio dos nomes dados aos elementos estruturais e do texto presente nos comentários. Ambos, são definidos durante o processo de programação, são dependentes às limitações impostas pela linguagem, mas sem amarrações à dimensão formal das regras gramaticais;

Em ambos os níveis de comunicação, um humano é o criador da informação a ser assimilada por um receptor: uma máquina ou um outro humano. No nível homem-máquina os desenvolvedores selecionam e codificam as instruções para fazer as máquinas executarem uma determinada tarefa, um algoritmo. Nessa perspectiva o software é um produto que produz resultados visíveis. É também o nível homem-máquina que dá sustentação a principal diferença entre as linguagens de programação e as linguagens naturais: a ambiguidade. Textos escritos em linguagem natural facilmente são ambíguos, enquanto que programas de computador não [Kokol and Podgorelec, 1999; Sipser, 2007].

Já na comunicação homem-homem, os desenvolvedores são os autores das cadeias de caracteres, denominados de **identificadores**, que são utilizadas para definir, referenciar e manipular tanto elementos estruturais básicos (*e.g.*: atributos, variáveis locais, nomes de parâmetros e de métodos), como estruturas mais complexas. Por sua vez, os comentários são textos escritos pelos desenvolvedores em linguagem natural, com o objetivo de documentar explicitamente os elementos de um código fonte.

As estruturas mais complexas são aquelas que abstraem um tipo ou que encapsulam outras estruturas estáticas (básicas ou também complexas), e que denominamos de **Entidades de Código**. No contexto da Tese, são consideradas entidades[1] classes e interfaces. Uma entidade além de estar associada ao identificador que a nomeia, têm sob seu escopo estático os identificadores e os comentários dos seus elementos, básicos e outras entidades, nela contidos. Nessa visão, uma entidade é um *container*, um repositório, onde está contida toda codificação nível homem-homem que hierarquicamente está sobre seu escopo.

Na perspectiva homem-homem, o software é equiparável à uma expressão artística de uma ideia. Assim como palavras de um texto revelam informações sobre o autor, os identificadores e comentários de um código fonte refletem conceitos ou ideias definidas pelos

[1]No paradigma da programação Orientado a Objetos (OO) em Java, podem ser consideradas entidades: pacotes, arquivos, classes, interfaces, enumerações e métodos.

desenvolvedores [Haiduc and Marcus, 2008] de acordo com o seu entendimento sobre o projeto do sistema, suas habilidades em programação e suas preferências pessoais [Shneiderman, 1980].

2.1.2 Conceituação de Vocabulário (Léxico) de Código Fonte

Informalmente um vocabulário, também denominado de **léxico** [Host and Ostvold, 2007; Antoniol et al., 2007], de código fonte compreende todo o conteúdo do código fonte codificado no nível homem-homem. Contudo, essa concepção não é um consenso na comunidade.

Para Bugrística e colaboradores [Biggerstaff et al., 1993] que foram um dos primeiros a propor uma definição, o léxico de um software deve ser encarado como uma estrutura de elementos a ser decomposta. Se visto como uma decomposição conceitual, o léxico L, é a união de um conjunto de termos de domínio de problemas humano L_H, com um conjunto de termos de programação L_P, logo é definido por: $L = L_H \cup L_P$. Se visto como uma decomposição operacional o vocabulário é a união de um conjunto de termos de documentação, L_{Doc}, com um conjunto de termos do código fonte L_{Src}, logo é dado por: $L = L_{Doc} \cup L_{Src}$.

Haidê e Marcus [Haiduc and Marcus, 2008] foram mais afundo e propuseram uma decomposição que insere na decomposição operacional também a conceitual. Assim, o léxico de código fonte deve ser definido por: $L_{Src} = L_{Src_H} \cup L_{Src_P}$, onde: L_{Src_H} é o conjunto de termos do fonte associados a linguagem humana, e L_{Src_P} é o conjunto de termos do fonte associados a programação.

Já a decomposição proposta por Abebé e colegas [Abebe et al., 2009] aponta que o vocabulário do código fonte, consiste da união de cinco diferentes vocabulários: de nomes de classe (*CV*), de nomes de atributos (*AVA*), de nomes de funções (*FEV*), de nomes de parâmetros (*PV*) e de comentários (*COV*). Cada um desses vocabulários é um conjunto de palavras distintas, únicas.

Cada pesquisador utiliza, na prática, sua própria definição de vocabulário ou aquela lhe for mais conveniente, e para descrever os vazios não cobertos pela definição escolhida combina com linguagem natural. Como consequência, essa combinação muitas vezes propicia interpretações dúbias e dificulta a reprodução dos estudos.

2.1.3 Importância do Vocabulário de Código Fonte

Independente das definições e decomposições dadas para vocabulário, os estudos concordam que um vocabulário é constituído de cadeias de caracteres. Por vezes, as cadeias de caracteres não formam palavras que tenham sentido em língua qualquer, nesses casos as denominamos de ***tokens***. Mas, mesmo nessas situações *tokens* podem representar conceitos ou ideias definidas pelos desenvolvedores, por exemplo no uso de abreviações (*e.g.*: depto, abbr), ou acrônimos, ou como uma sequência *ad-hoc* de caracteres.

Várias pesquisas apontam que os identificadores estão entre as mais importantes fontes de informações para entender o código fonte [Caprile and Tonella, 2000; Deissenboeck and Pizka, 2006]. Haidê e Marcus [Haiduc and Marcus, 2008] estudaram vários programas *open source* e descobriram que cerca de 40% das palavras inerentes ao domínio de problema de cada sistema analisado foram usadas também nos seus respectivos códigos-fonte.

Não existem restrições para definir um identificador, exceto pelo fato de ter que se obedecer às regras de formação impostas pelas linguagens de programação. Inclusive em sistemas legados, era possível encontrar procedimentos e dados nomeados arbitrariamente, muitas vezes com nomes de namoradas ou jogadores favoritos dos programadores [Sneed, 1996].

Hoje em dia, Java por exemplo impõe que identificadores não podem iniciar com dígito ou caracteres especiais, tais como "@", "#", "%", nem podem ser uma das palavras reservadas da linguagem. Também não permite espaços entre palavras de um mesmo identificador. Contudo, para expressar o que um elemento do código representa em conceito ou ideia, é possível utilizar mais de um *token* para nomear os identificadores compostos (em inglês, *multi-word identifier*) [Pollock et al., 2013]. Para concatenar os *tokens*, pode-se fazer uso de um caractere especial (*e.g.*: hífen, "-"; sublinha, "_"; cifrão, "$"), de dígitos, ou ainda de um estilo de notação.

Um **estilo de notação** é uma convenção, definida e difundida por uma comunidade usuária de uma dada linguagem de programação, cujas regras são utilizadas para nomear identificadores. Por exemplo, para linguagem Java sua comunidade de desenvolvedores tem a padronização proposta pela *Sun/Oracle Java Code Conventions* [Sun Microsystems, 2012]. Segundo Binkley e colaboradores [Binkley et al., 2009], os dois estilos de notação mais populares são *camelcase* e *underscore*. No *camelcase* as palavras ou *tokens* que compõem um identificador tem sua primeira letra em maiúscula e as demais em minúscula. Já na notação

underscore as palavras ou *tokens* são concatenados com o símbolo de sublinha "_".

De acordo com Enslen e colegas [Enslen et al., 2009], em cerca de 88% dos identificadores compostos seus *tokens* são concatenados com *camelcase*. Um outro levantamento, feito por Butler e colaboradores, [Butler et al., 2011] corrobora com a predominância da notação *camelcase*, em código Java, e reporta que em cerca de 9% dos identificadores compostos os *tokens* ou são concatenados sem qualquer caractere especial, ou separados por dígitos.

Assim como os identificadores, as palavras que compõem as sentenças escritas em linguagem natural dos comentários também são cadeias de caracteres. Comentários, apesar de fazerem parte do código fonte, não são convertidos em código executável. No entanto, eles documentam funcionalidades, algoritmos, arquitetura, e dados que manipulam as unidades de código [Haouari et al., 2011], seguem a evolução do código homem-máquina [Fluri et al., 2007], e são contribuições dos desenvolvedores assim como também são os identificadores.

De fato, nos últimos anos, observou-se uma melhoria na qualidade dos identificadores utilizados no código fonte, bem como no uso de comentários como documentação interna [Haouari et al., 2011]. 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.

2.1.4 *Information Retrieval (IR)* sobre Vocabulários

Em pesquisas sobre vocabulário ou que dele façam uso, classes e interfaces são representadas como documentos de natureza não estruturada. É a área de recuperação da informação (*Information Retrieval - IR*) que define conceitos e fornece técnicas para extrair informações de um coleção de documentos, no nosso caso um vocabulário [D. Manning et al., 2008].

Para reduzir o espaço de busca e custo computacional, *IR* preconiza a realização de um processamento prévio sobre as informações não estruturadas [Binkley, 2007; Binkley and Lawrie, 2009]. No contexto deste estudo, sobre cada elemento do vocabulário são executadas operações típicas de *IR* [D. Manning et al., 2008]:

1. **Tokenização** - separação dos *tokens* que compõem os identificadores compostos;

2. **Normalização de *Tokens*** - unificação de *tokens* com diferenças superficiais, por exemplo *Class* ou *class*;

3. **Remoção de *Stop Words*** - exclusão de palavras cujas ocorrências são extremamente comuns por aparecerem em quase todas as entidades (*e.g.*: "*is*", "*of*", "*the*", "*a*"). Removê-los é importante porque eles agregam pouco valor numa consulta para seleção de documentos;

4. ***Stemming*** - redução de um *token*, quando uma palavra, a seu radical usando alguma heurística. Nos nossos estudos utilizamos o algoritmo de Porter [Porter, 1980] implementado pela biblioteca Apache Lucene [Apache Foundation, b] que reduz palavras da língua inglesa. Por exemplo, *connected*, *connecting* possuem o mesmo radical: *connect*.

Palavras ou *tokens*, após serem submetidos a um processamento de *IR*, passam a ser denominados de **termos**. Para ilustrar o processo de extração de termos tomemos como exemplo o identificador "isDataBaseConnected", codificado em Java. Primeiramente, ele é separado, em função da notação *camelcase*, em quatro *tokens*. Cada *token* é normalizado, nesse caso convertendo todos os caracteres para minúsculos. O resultado são os *tokens* "*is*", "*data*", "*base*" e "*connected*". Em seguida, a *stopword* "*is*" é removida. Finalmente, o *stemming* é executado quando o sufixo "**ed**" é removido do *token* "connected". Assim os termos resultantes do identificador "*isDataBaseConnected*" são: "*data*", "*base*" e "*connect*".

Um termo tanto pode ser diferente como idêntico a qualquer outro já extraído do vocabulário. Termos idênticos são aqueles que possuem a mesma sequência de caracteres. Na realidade, é natural existirem várias ocorrências de termos idênticos ao longo de um código fonte de um sistema. Essas repetições acontecem ou por ser necessário que em outros trechos do código aquele mesmo elemento seja acessado, ou porque outros elementos são nomeados com os mesmos termos para indicar as mesmas ideias ou conceitos [Haiduc and Marcus, 2008; Abebe et al., 2009].

Assim, no contexto desta Tese, o **vocabulário do código fonte** consiste no conjunto de termos distintos, únicos, que compreendem os identificadores ou que estão presentes no texto dos comentários de um código fonte [Abebe et al., 2009].

2.2 *Expertise* (Conhecimento de Código)

Em grandes e complexos projetos de sistemas de software é inerente a necessidade de que suas implementações sejam divididas entre os vários membros da equipe de desenvolvimento. Mesmo seguindo processos de desenvolvimento, manter atualizada a informação de quem é o indivíduo mais adequado para realizar uma determinada tarefa de codificação, não é uma questão simples.

Há conhecimentos que podem ser atestados de forma direta através de um diploma de certificação. Por exemplo, um programador iniciante com certificação Java pela *Sun/Oracle* está habilitado para fazer parte de uma equipe de desenvolvimento de um sistema que esteja sendo codificado em Java. Contudo, esse mesmo programador muito provavelmente não será o mais adequado para corrigir um defeito (*bug*) relatado sobre uma dada entidade do sistema, já que ele não terá conhecimento sobre os conceitos, abstrações e comportamento da entidade, e funcionalidades do sistema.

De acordo com Mockus e colegas [Mockus and Herbsleb, 2002] ***expertise*** é definida como a habilidade do desenvolvedor de ser especialista sobre um código fonte, e, se interpretada quantitativamente, reflete o grau de capacidade que o desenvolvedor tem para executar uma certa tarefa de codificação. Neste documento utilizaremos a terminologia **conhecimento de código** e **especialista de código** como alternativas aos termos *expertise* e *expert*, respectivamente. E, melhor é um especialista de código quanto mais rápido ele consiga executar uma tarefa, com o mínimo de esforço e produzindo resultados (código) de qualidade.

2.2.1 Medidas de *Expertise*

Expertise ou conhecimento de código como definida, é difícil de ser mensurada diretamente. É o seu efeito sobre o processo e sobre o produto que importa. A literatura então, estuda *expertise* de forma indireta através da observação e medição de uma diversidade de medidas sobre processo e produto.

Contudo, dois conceitos são básicos na modelagem de *expertise*: **Autoria** (*Authorship*) e **Propriedade** ou **Posse** (*Ownership*). No contexto da nossa pesquisa, assumimos a definição dada por Corley e colegas [Corley et al., 2012]. Autoria ocorre quando um desenvolvedor

escreve, é autor, de um pedaço de código. Já a propriedade ou posse ocorre quando um desenvolvedor insere um novo código ou concretiza uma mudança no código de um repositório. Autoria não implica em posse e *vice-versa*.

De acordo com Cavalcanti [Cavalcanti, 2012] entre 50% e 75% das tarefas de codificação para consertar defeitos, apenas uma única classe é impactada. Essa é uma das justificativas para nesta Tese estarmos focados em encontrar especialistas que sejam os conhecedores mais experientes em unidades de trecho de código fonte cuja granularidade seja entidades, classes e interfaces.

As abordagens se baseiam na premissa de que desenvolvedores que contribuíram substancialmente, no passado, com mudanças em partes específicas do código fonte são os desenvolvedores adequados para realizar as atuais e futuras mudanças sobre aquele código [Kagdi et al., 2008]. E mais, o desenvolvedor que mais contribuiu com mudanças numa parte específica do código, será considerado o que mais tem conhecimento sobre aquele código.

É dos sistemas de gerenciamento de mudanças, *VCS*, que as técnicas extraem os dados de contribuições do passado para serem utilizadas como insumos às medidas de *expertise* para código fonte. A seguir, apresentamos os principais tipos de contribuições consideradas nos estudos para identificar *expertise*.

Contribuições por Desenvolvedores

A frequência e a extensão das mudanças em arquivos concretizadas pelos desenvolvedores podem ser utilizadas também como uma medida de contribuição. Um desenvolvedor que concretiza alterações nos arquivos tem (ou adquire) conhecimento sobre esses arquivos. Quanto maior a concentração do número de *commits* em alguns arquivos por um dado desenvolvedor, maior a probabilidade do seu conhecimento estar sobre aqueles arquivos, são os especialistas com conhecimento em profundidade. Por outro lado, especialista com conhecimento em amplitude é um desenvolvedor que concretiza *commits* espalhados, ao longo de muitos arquivos, o que indica que seu conhecimento sobre o sistema é também espalhado, e não específico à poucos arquivos. Esse tipo de informação tem alavancado estudos sobre o papel e a importância dos desenvolvedores de núcleo e de periferia no modelo de desenvolvimento *open source* [Mockus et al., 2002; Crowston and Howison, 2006; Terceiro et al., 2010].

Contribuições por Arquivos

É dado pelo número de arquivos, entre um grande número de *commits*, que são atualizados exclusivamente por apenas um único desenvolvedor e por mais nenhum outro. Essa densidade de arquivos dá a ideia da importância de um dado desenvolvedor para o projeto do sistema. Geralmente as contribuições por arquivos são utilizadas em conjunto com outras medidas de *expertise*, por exemplo com contribuições por desenvolvedores. Arquivos que, por um longo período de tempo, tenham sido modificados apenas por um único desenvolvedor, muito provavelmente terão seus códigos fontes dominados por apenas esse desenvolvedor.

Contribuições por *Commits*

Um sistema de software ao longo do seu ciclo de vida sofre uma enorme quantidade de *commits*[2]. A relação entre um desenvolvedor e os arquivos para um dado *commit* é de um para um. No entanto, um desenvolvedor pode contribuir com múltiplos *commits* para um mesmo arquivo. Também, múltiplos desenvolvedores podem mudar o mesmo arquivo através de *commits* distintos. Assim, *commits* possibilitam analisar as contribuições dos desenvolvedores quando realizadas de forma exclusiva sobre arquivos e também quando executadas de forma compartilhada [Kagdi et al., 2008]. A contribuição de um desenvolvedor para uma parte do código, é dada pelo total de *commits* por ele realizado sobre o referido código [Mockus and Herbsleb, 2002; Fritz et al., 2010].

Contribuições por Atividade

Para fins de mensurar *expertise*, atividade é a quantidade de dias do desenvolvimento em que houve mudanças de código concretizadas [Kagdi et al., 2008]. A atividade de um desenvolvedor é o percentual de dias em que ele submeteu pelo menos um *commit* em relação ao total de dias em que houve *commits*. É também uma medida complementar as outras medidas de *expertise* [Kagdi et al., 2008; Terceiro et al., 2012].

[2] mudanças concretizadas no repositório do código fonte

Contribuições por Número de Linhas

Essa medida baseia-se no número de linhas adicionadas, removidas ou alteradas do código fonte. Entre os desenvolvedores, aquele que possuir o maior percentual de linhas modificadas é considerado o especialista [Girba et al., 2005]. A granularidade dessa medida pode ser no nível de arquivo, classe, módulo, ou sistema. Essa medida supri algumas situações em que o conhecimento de código quando medido através das contribuições por *commits*, não conseguem capturar. Por exemplo, o *commit* decorrente apenas da correção de tabulações da endentação do código de um algoritmo, incrementa erroneamente o conhecimento do desenvolvedor que o corrigiu [dos Santos et al., 2012]. Em função disso, os *VCS* mais modernos tais como *SVN*, *Mercurial*, fornecem suporte à extração das linhas modificadas por desenvolvedor, e as pesquisas mais recentes tem dado preferência a representar conhecimento através desse tipo de contribuição.

Abordagens para identificar especialistas podem utilizar mais de uma medida de contribuição para capturar o conhecimento de código. Essa combinação de medidas é um dos motivos à existência de uma variada taxonomia na literatura para nomear *expertise*. Um exemplo, é o modelo grau de conhecimento, em inglês *Degree-Of-Knowledge (DOK)*, proposto por Fritz *et al.*.

2.2.2 Modelo *Degree-Of-Knowledge (DOK)*

O *Degree-Of-Knowledge (DOK)*, proposto por Fritz *et al.* [Fritz et al., 2010] busca identificar especialistas, valorando a combinação dos componentes de autoria, *Degree-Of-Authorship (DOA)*, e a componente de interação sobre o código, *Degree-Of-Interest (DOI)*. O *DOA* reflete o conhecimento adquirido pelos desenvolvedores decorrentes das mudanças por eles concretizadas sobre os elementos de código, já o *DOI* aponta o interesse imediato dos desenvolvedores sobre um dado elemento.

Os dados do *DOA* são extraídos do sistema de versionamento, *VCS*, enquanto que os dados do *DOI*, que são as seleções e as edições realizadas pelos desenvolvedores sobre os elementos, são capturados através do ambiente de desenvolvimento. *DOA* aumenta quando se cria elementos (*First Authorship - FA*) e quando se entrega modificações do elemento criado (*DeLiveries - DL*), e diminuí quando se aceita modificações de outros desenvolvedores

sobre um elemento criado (*ACceptances - AC*). O *DOI* aumenta quando o desenvolvedor interage com o elemento e diminuí quando outros elementos sofrem interações.

DOK consiste em modelar o fluxo e o contra-fluxo de múltiplos desenvolvedores modificando um mesmo elemento. Logo, diferentes desenvolvedores podem ter diferentes valores de *DOK* para um mesmo elemento. *DOK* então, combina linearmente *DOA* e *DOI*, da seguinte forma:

$$DOK = \alpha_{FA} * FA + \alpha_{DL} * DL + \alpha_{AC} * AC + \beta_{DOI} * DOI \tag{2.1}$$

onde: α e β, são pesos para os fatores de *DOA* e de *DOI* respectivamente.

Fritz e colaboradores [Fritz et al., 2010] num estudo empírico, coletaram os dados de autoria *FA*, *DL*, *AC* e os dados de interação, para 40 elementos de código, escolhidos aleatoriamente, de um dado projeto. Com um *survey* sobre seus 7 desenvolvedores, obtiveram o grau de conhecimento, o valor de *DOK*, que cada um deles julgou ter sobre os elementos, considerando a seguinte escala: 1, aponta que o desenvolvedor não conhece o elemento; 3, indica que o desenvolvedor tem um certo conhecimento sobre o elemento, onde para reproduzi-lo é preciso que o desenvolvedor faça consultas ao código original do elemento; 5 significa que o desenvolvedor consegue reproduzir o código sem consultá-lo. Em seguida, os dados foram submetidos a uma regressão multilinear de *DOK* em função de *DOA* e *DOI*, ($DOK \sim DOA + DOI$) onde o ajuste definiu os pesos para α e β da Eq. 2.1.

Esse modelo de predição explica 25% (Coeficiente de Determinação, $R^2 = 0.25$) da variabilidade dos valores de *DOK*. Apesar do *p-value* para todo modelo e dos *p-value* para as variáveis preditoras *FA*, *DL*, *AC* e *DOI* indicarem uma significância estatística, esse valor de R^2 demonstra que ainda falta 75% da variabilidade de *DOK* para ser explicado. Em outras palavras, há outros fatores que precisariam ser considerados para indicar, com uma precisão maior, quem é o especialista de um dado elemento de código [Fritz et al., 2010]. Esse percentual não explicado pelo *DOK*, segundo Fritz, em parte, é devido a influência sobre o modelo tanto do processo de desenvolvimento (fase atual, relação de propriedade entre desenvolvedores e elementos do código), quanto do sistema de software (tipo, tamanho, complexidade) [Fritz et al., 2010].

Tal fato corrobora com uma conclusão comum a que chegam os estudos e pesquisas que

buscam identificar especialistas: é preciso avançar e contemplar outros aspectos não utilizados em suas respectivas abordagens [Mockus and Herbsleb, 2002; Girba et al., 2005; Hattori and Lanza, 2009; Matter et al., 2009; Rahman and Devanbu, 2011]. E, independente das medidas de contribuições ou do tipo da abordagem utilizadas, a informação sobre especialistas deve estar facilmente acessível a desenvolvedores, testadores, e gerentes, para ajudá-los a rapidamente localizar os melhores indivíduos para executar cada tarefa de codificação [Mockus and Herbsleb, 2002].

Capítulo 3

Vocabulários de Software

Neste capítulo, na Seção 3.1, apresentamos uma representação formal para a nossa concepção sobre vocabulários, bem como para suas propriedades e operações. Na Seção 3.2, apresentamos os *surveys* experimentais que realizamos para compreender, do ponto de vista quantitativo, as origens da formação dos vocabulários de código fonte.

3.1 Caracterização do Vocabulário de Código Fonte

Na literatura consultada, como descrito na Seção 2.1.2, cada pesquisador acaba utilizando sua própria decomposição de vocabulário, o que resulta também em definições que lhes forem mais conveniente. E como não há, até onde temos conhecimento, nenhuma definição formal amplamente divulgada e aceita pela comunidade, faz-se uso da linguagem natural escrita para descrever os vazios não cobertos pela definição de vocabulário escolhida. Para preencher essa lacuna, que muitas vezes propicia interpretações dúbias e dificulta a reprodução dos estudos, é que apresentamos na Seção 3.1.1 uma formalização para vocabulários como uma das contribuições desta Tese.

3.1.1 Formalização de Vocabulário

Independente das diferentes decomposições encontradas na literatura, os estudos concordam que um vocabulário é constituído de cadeias de caracteres, *termos*, e que estes podem repetidas vezes ocorrer ao longo de um código fonte.

Partindo dessa concordância, nosso entendimento é que o conceito de *multiset*[1] fornece teoria e abstração apropriadas à representação de vocabulário de código.

Definimos, um vocabulário de código como um *multiset* de *strings*, *i.e.* uma aplicação de $V : \mathbb{S} \rightarrow \mathbb{N}$ que mapeia *strings*, $\mathbb{S}$, em um número natural $\mathbb{N}$. Para qualquer *termo* t, $V(t)$ denota o número de ocorrências de um *termo* t no vocabulário V. Se $V(t) > 0$ dizemos que t é um *termo* do vocabulário.

Com essa concepção, podemos definir o vocabulário para trechos de código quer seja um método, uma classe, ou todo um sistema, como também para uma codificação a ser realizada por um desenvolvedor. Para uma classe, por exemplo, seu vocabulário contém os identificadores e os comentários de todos seus elementos, básicos e/ou outras entidades, que hierarquicamente estão sobre seu escopo estrutural. E a união dos vocabulários de cada entidade que compõem um sistema constitui o vocabulário do código fonte de todo sistema.

Há diferentes notações para *multisets*. Adotamos a notação baseada em somas formais, por julgarmos ser a mais adequada ao contexto de vocabulários, já que propicia a expressividade da multiplicidade associada ao conteúdo dos termos. Nela, cada elemento expressa o número de ocorrências o de um *termo* t na forma $o't$.

$$V = o_1\,'t_1 + o_2\,'t_2 +, \ldots, +o_n\,'t_n \tag{3.1}$$

Como exemplo, considere o trecho de código a seguir, na Listagem 3.1.

Código Fonte 3.1: Trecho de Código

```
1 public static boolean meth(){
2     boolean connected=false;
3     MyConnection c = new MyConnection();
4     connected = c.login();
5     return(connected);
6 }
```

De acordo com a nossa definição e a notação de somas formais, o vocabulário do código fonte da Listagem 3.1 é expresso pelo seguinte *multiset* de termos:

$$V_1 = 3'connected + 2'c + 2'MyConnection + 1'login + 1'meth$$

[1]Um *multiset* é um conjunto de objetos onde os elementos podem ocorrer mais de uma vez [Blizard, 1988]. Cada elemento de um *multiset* é definido como um par (A, m), onde A é um conjunto qualquer, e $m : A \rightarrow \mathbb{N}$ a função multiplicidade que associa a cada elemento de A um número natural, maior que zero, $\mathbb{N} = \{1, 2, 3, ...\}$.

3.1.2 Operações sobre Vocabulários

Processamentos típicos de *IR* e outras operações podem ser, facilmente e de forma não ambígua, especificadas como funções sobre vocabulários.

Tomemos, como exemplo, uma operação de tokenização que divide termos de acordo com o estilo de notação *camelcase* (cc). É uma função definida por $cc : \mathbb{V} \rightarrow \mathbb{V}$ que mapeia cada *termo* $t \in V$ do domínio[2] de cc para um conjunto de termos $\{t_1, t_2, \ldots, t_n\} \in V$, onde cada termo t_i é uma palavra ou *token* que compõe o termo original t. Por exemplo, o vocabulário V_1, é mapeado pela operação de cc, para o vocabulário V_2, dado por:

$$V_2 = cc(V_1) \quad = \quad 3'connected + 2'c + 2'My + 2'Connection + 1'login + 1'meth$$

De forma similar, definimos em termos de *multisets* outras funções sobre vocabulário cujos detalhes podem ser consultados no relatório técnico sobre a *Decomposição do Modelo de Vocabulário de Software baseada na Visão Modular Estrutural da Arquitetura*, [Santos, 2012]. Entre elas, as de *IR* exemplificadas a seguir:

- tokenização, também para notação *underscore*, us;
- normalização, conversão de caracteres para minúsculo, clc. Aplicado sobre o vocabulário V_2, resulta em:

$$V_3 = clc(V_2) \quad = \quad 3'connected + 2'c + 2'my + 2'connection + 1'login + 1'meth;$$

- *stemming*, st. Sobre V_3, temos:

$$V_4 = st(V_3) \quad = \quad 5'connect + 2'c + 2'my + +1'login + 1'meth;$$

[2] $\mathbb{V}$ denota o conjunto de todos os vocabulários possíveis.

- remoção de *stop words*, swr. Para as *Stop Words*, $SW = \{an, is, my, of\}$, temos:

$$V_5 = swr(V_4) = 5'connect + 2'c + 1'login + 1'meth;$$

- remoção de termos com tamanho inferior a um *threshold* em número de caracteres, $trt_{threshold}$. Para $threshold = 2$, sobre vocabulário V_5, temos:

$$V_6 = trt_2(V_5) = 5'connect + 1'login + 1'meth.$$

3.1.3 Propriedades do Vocabulário

Durante nossos estudos empíricos [Santos et al., 2010; Santos et al., 2012] para compreender a natureza dos vocabulários, identificamos duas propriedades observáveis em quaisquer vocabulários: seu tamanho e a frequência com que os seus termos ocorrem. Ambas, são extraídas após a aplicação das operações de *IR* sobre vocabulários.

Tamanho de um Vocabulário

A propriedade de tamanho captura o quantitativo de termos que compõe um vocabulário através de duas métricas.

A primeira é o total de termos (*Total of Terms - TT*) de um vocabulário ($TT(V)$). É soma das multiplicidades, de todos os termos t de um vocabulário, e é definido por:

$$TT(V) = \sum_{i=1}^{n} V(t_i), \forall t_i \in T \tag{3.2}$$

A segunda métrica é o número de termos distintos (*Distinct Terms - DT*) de um vocabulário, ($DT(V)$). É o quantitativo de termos que possuem multiplicidade de pelos menos um. Definimos por:

$$DT(V) = |\{t_1, t_2, \ldots, t_n\}|, \forall t_i \in T, V(t_i) > 0 \tag{3.3}$$

Considerando o vocabulário V_6, como exemplo, temos: $TT(V_6) = 7 \ \ e \ \ DT(V_6) = 3$.

Vetor de Frequência Característico

Vetor de Frequência Característico, em inglês *Characteristic Frequency Vector - CFV*, busca mapear como se dá a distribuição das ocorrências dos termos distintos em um vocabulário. Ele consiste num vetor com todas as multiplicidades dos elementos em ordem decrescente de valor. Os *CFV* são abstrações que nos permitem caracterizar vocabulário desconsiderando o conteúdo dos termos em si.

$$CFV(V) = (V(t_1), V(t_2), \ldots, V(t_{DT(V)})),\ \ \forall i\ \ V(t_i) \geq V(t_{(i+1)}),\ \ 1 < (i+1) < DT(V)$$

Por exemplo, o vetor de frequência característico de V_6 é: $CVF(V_6) = (5, 1, 1)$.

É preciso observar que o vetor de frequência característico do vocabulário é ligeiramente diferente do (*document vector model*) utilizados nas técnicas de *IR* [D. Manning et al., 2008]. O vetor de frequência característico definido neste estudo é um vetor ordenado, em que desconsideramos a relação de indexação entre multiplicidades e termos. O uso desta abstração justificou-se porque não estávamos interessado na recuperação de documentos com base na relevância de determinados termos dentro deles. Em vez disso, estávamos focados em modelar e a caracterizar os vocabulários.

3.2 Compreensão sobre a Natureza dos Vocabulários

Pesquisas sugerem que a maneira como os identificadores e comentários são usados no código fonte influencia na qualidade e na compreensão de um software. São, assim, uma fonte de informação para ferramentas e técnicas que auxiliam nas atividades de desenvolvimento [Binkley and Lawrie, 2011] e manutenção de sistemas [Binkley and Lawrie, 2009].

Extraídas de forma coerente as informações oriundas do vocabulário de código podem revelar aspectos diferentes dos capturados pelas medidas tradicionais [Arnaoudova et al., 2010] (*e.g.*: métricas CK [McCabe, 1976]) e que impactam no software enquanto produto e processo.

Para entender a origem da formação dos vocabulários como um fenômeno de influência humana, adotamos inicialmente uma abordagem estatística explorativa.

3.2.1 Relacionando Tamanho do Sistema com Vocabulário

Os termos que compõem um vocabulário estão espalhados ao longo das entidades que constituem o código fonte de um projeto quer seja para indicar as mesmas ideias ou conceitos, quer seja para acessar os elementos estruturais em pontos distintos de código que devem se relacionar [Haiduc and Marcus, 2008; Abebe et al., 2009]. Conjecturamos então que o tamanho do sistema tem algum nível de relação com as métricas que definimos para medir tamanho de vocabulários: Total de Termos, e número de Termos Distintos. Para investigar e determinar essa relação realizamos um experimento cuja metodologia, ilustrada na Figura 3.1, foi dividida em 4 etapas: 1) definição da amostra, 2) extração de vocabulários, 3) cálculo das métricas de vocabulário e 4) identificação de propriedades.

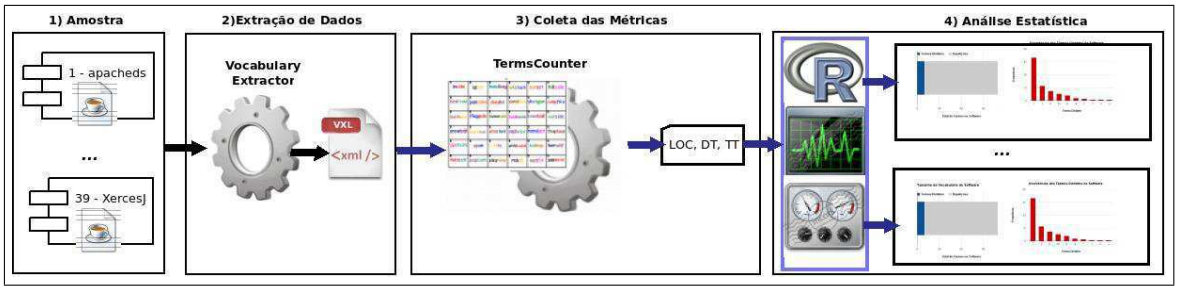


Figura 3.1: Etapas da metodologia utilizada.

Definição da Amostra

Dada a natureza dos participantes, projetos Java *open source*, o universo amostral do qual fazem parte é de difícil contagem. Quando depositados em repositórios de código *open source* pelos seus criadores e mantenedores, os projetos nem sempre são rotulados corretamente: de acordo com o domínio do problema aos quais estão relacionados, nem muito menos de acordo com as tecnologias e linguagens de programação que foram utilizadas no seu desenvolvimento. Tal fato não permite a correta contagem de toda a população de pro-

jetos Java *open source* e consequentemente inviabiliza uso de técnicas de seleção amostral para obtenção de uma amostra representativa [Wasserman, 2004], [Travassos, 2011]. Por isso, utilizamos um conjunto de critérios como guia para a seleção, por conveniência, dos projetos a fazerem parte da amostra:

1. o sistema deveria ser *open source*;
2. os identificadores deveriam estar codificados em língua inglesa;
3. o estilo de notação dos identificadores deveria seguir ou o estilo *camelcase*, ou o *underscore* ou os dois;
4. o projeto deveria ter uma comunidade de desenvolvedores ativa;
5. o projeto deveria ter um grande número de *downloads* recentes na sua categoria[3] e ter obtido mais recomendações positivas que negativas.

Também por conveniência eles deveriam ser de diferentes domínios de problemas e abranger uma faixa representativa de tamanhos em *LOC*.

Selecionamos 39 projetos Java *open source* (15 do *SourceForge* [sou, 2009] e 24 do Apache Foundation [Apache Foundation, a]), o que nos levou a processar perto de 4 milhões de *LOC* contendo cerca 64 mil entidades, entre classes e interfaces, totalizando mais de 2,3 milhões de termos. A Tabela 3.1 apresenta todos os projetos que compuseram a amostra, com suas respectivas versões, tamanho em *LOC*, total de entidades, quantidade de termos distintos e totais de seus vocabulários, e *URL* originais de onde seus códigos foram obtidos via *download*.

Extração de Dados de Vocabulário

Para varrer um código fonte java e dele extrair suas informações de vocabulário, concebemos e desenvolvemos o ferramental *Vocabulary Tool*[4] cuja descrição detalhada pode ser consul-

[3]Repositório de Projetos permitem a categorização dos projetos hospedados geralmente em função da especificação feita pelos desenvolvedores do projeto.

[4]Ferramental disponível no site http:/www.softwarevocabulary.org, na página *Download* da seção *Vocabulary Tool*.

Tabela 3.1: Lista dos projetos Java *open source* da amostra, e respectivas informações.

Índice	Projeto	Versão	LOC	Total Entidades	Termos Distintos	Total de Termos	URL
1	apacheds	1.5.7	154,906	1,224	1,801	65,036	http://svn.apache.org/repos/asf/directory/apacheds/tags/
2	axis2	1.5.3	285,979	2,651	2,586	129,620	http://svn.apache.org/repos/asf/axis/axis2/java/core/tags/
3	beehive	v1.0.2	198,914	2,191	1,922	77,069	http://svn.apache.org/repos/asf/beehive/tags/
4	cxf	2.3.1	360,758	4,011	3,264	167,618	http://svn.apache.org/repos/asf/cxf/tags/
5	derby	10.7.1.1	607,710	2,860	5,226	256,279	http://svn.apache.org/repos/asf/db/derby/code/tags/
6	easymock	3.0 r205	11,954	196	396	5,974	http://sourceforge.net/projects/easymock/
7	findbugs	1.1.1	73,537	921	1,638	39,728	http://sourceforge.net/projects/findbugs/
8	geronimo	3.0M1	194,718	2,697	2,279	114,195	https://svn.apache.org/repos/asf/geronimo/server/tags/
9	Hadoop Common	0.21.0	71,771	903	1,743	34,967	http://svn.apache.org/repos/asf/hadoop/common/tags/
10	Hadoop Mapreduce	0.21.0	172,077	1,822	2,429	78,362	http://svn.apache.org/repos/asf/hadoop/mapreduce/tags/
11	ivy	2.1.0	54,739	568	1,040	28,535	https://svn.apache.org/repos/asf/ant/ivy/core/tags/
12	jabref	2.0b r3397	43,906	532	1,296	19,040	http://sourceforge.net/projects/jabref/
13	jackrabbit	2.2.1	269,938	2,766	2,155	119,298	http://svn.apache.org/repos/asf/jackrabbit/tags/
14	james-server	3.0M2	40,166	512	1,055	20,503	http://svn.apache.org/repos/asf/james/server/tags/
15	jedit	4.3pre9 r8692	99,656	812	1,955	37,186	http://sourceforge.net/projects/jedit/
16	jfreechart	1.0.8 r2273	127,526	944	1,160	62,373	http://sourceforge.net/projects/jfreechart/
17	jgnash	2.5 r2456	44,950	499	1,092	19,208	http://sourceforge.net/projects/jgnash/
18	jGroups	2.12	66,871	558	1,349	31,360	http://sourceforge.net/projects/javagroups/
19	junit	4.5	11,997	393	582	7,004	http://sourceforge.net/projects/junit/
20	jvlt	1.1.4 r592	17,363	291	555	9,451	http://sourceforge.net/projects/jvlt/
21	lucene	3.0.3	171,369	1,759	2,297	73,400	http://svn.apache.org/repos/asf/lucene/java/tags/
22	myfaces-core	2.0.3	101,475	1,196	1,155	46,926	https://svn.apache.org/repos/asf/myfaces/core/tags/
23	myfaces-tomahawk	1.1.10	128,922	1,097	1,373	48,245	https://svn.apache.org/repos/asf/myfaces/tomahawk/tags/
24	openJPA	2.0.1	354,996	3,653	2,851	186,412	https://svn.apache.org/repos/asf/openjpa/tags/
25	pdfsam	2.2.1	26,058	312	812	16,009	http://sourceforge.net/projects/pdfsam/
26	pjirc	2.2.1	10,114	133	585	4,736	http://sourceforge.net/projects/pjirc/
27	pmd	4.2.5 r7178	60,062	817	1,464	27,731	http://sourceforge.net/projects/pmd/
28	robocode	1.7.2.2 r3549	52,267	679	1,350	29,230	http://sourceforge.net/projects/robocode/develop/
29	roller	4.1M1	60,484	633	1,060	28,089	http://svn.apache.org/repos/asf/roller/tags/
30	shindig	2.0.2	78,682	1,007	1,565	48,446	http://svn.apache.org/repos/asf/shindig/tags/
31	solr	1.4.1	79,436	970	1,718	35,459	http://svn.apache.org/repos/asf/lucene/solr/tags/
32	STRUTS2	2.1.1	144,242	1,986	1,894	78,134	https://svn.apache.org/repos/asf/struts/struts2/tags/
33	SweetHome3D	3.0	64,876	367	901	31,529	http://sourceforge.net/projects/sweethome3d/
34	tapestry	4.1.6	109,757	1,770	1,472	64,820	http://svn.apache.org/repos/asf/tapestry/tapestry4/tags/
35	uimaj	2.3.1	171,448	1,824	2,181	95,613	http://svn.apache.org/repos/asf/uima/uimaj/tags/
36	VilloNanny	2.4	14,731	108	610	5,503	http://sourceforge.net/projects/villonanny/
37	wicket	1.4.0	138,512	2,272	1,616	61,575	http://svn.apache.org/repos/asf/wicket/tags/
38	xalan-J	2.7.1	171,488	1,004	1,886	85,736	http://svn.apache.org/repos/asf/xalan/java/tags/
39	Xerces-J	2.9.1	131,310	897	1,621	58,150	http://svn.apache.org/repos/asf/xerces/java/tags/
Totais			**3,996,045**	**49,835**	**63,934**	**2,348,549**	

tada no Apêndice A. Resumidamente, para realizar a extração das informações, o código fonte analisado é completamente mapeado para uma estrutura sintática em árvore, *Abstract Syntax Tree - AST*, equivalente. Em seguida, percorre-se a *AST* através de seus nodos, e deles se extrai os elementos estruturais de interesse. Desse modo, para todas as entidades do código são coletados seus identificadores, seus comentários de linha e de bloco, e seus *javadoc*. Toda informação é armazenada de forma semi-estruturada utilizando a meta-linguagem *XML* (*eXtended Markup Language*), num arquivo de extensão *VXL* (*Vocabulary eXtended Language*), onde a hierarquia estabelecida no código fonte para os elementos estruturais é preservada no armazenamento de identificadores, comentários e *javadoc*.

O armazenamento em *VXL* possibilita que outras ferramentas, de posse do esquema *XML Schema Document - XSD*, possam acessar as informações de vocabulário. Já a manutenção

da hierarquia das entidades, possibilita que estudos outros sobre vocabulário, mesmo aqueles que precisem se relacionar com dados estruturais, possam fazer uso do nosso ferramental. Repositórios de arquivos *VXL* extraídos de códigos fontes de projetos *open source* por exemplo, podem ser incrementalmente construídos e disponibilizados para comunidade científica que se interessa em vocabulários de código.

Cálculo das Métricas de Vocabulários

Esta etapa processa cada vocabulário, e sobre o vocabulário resultante computa-se suas medidas de tamanho e seu vetor de frequência característico. O processamento sobre um vocabulário consiste em aplicar os seguintes operadores:

1. de tokenização, tanto para notação *camelcase*, cc, quanto para *underscore*, us;
2. de conversão para minúsculo, clc;
3. de extração de radical, st;
4. de remoção de *stop words*, swr;
5. de remoção de termos com tamanho inferior a 2, em número de caracteres, trt_1. Para fins das nossas análises, consideramos apenas termos com tamanho igual ou maior que 2 caracteres.

Só após o processamento de um vocabulário é que suas medidas são calculadas. Para ilustrar, considere as operações definidas na Seção 3.1.2, para cada vocabulário V da nossa amostra, o processamento é dado por: $V' = trt_1(swr(clc(cc(us(V)))))$. O cálculo da medida de tamanho de V é: $TT(V')\ e\ DT(V')$, e seu vetor de frequência característico é dado por: $CFV(V')$.

Análise Estatística

A exploração estatística nos revelou que as medidas de tamanho *LOC* e número de entidades #E tem multicolinearidade[5] quando utilizadas para modelar o quantitativo de termos e as suas ocorrências em um vocabulário. Contudo, fortes evidências apontaram *LOC* como a mais adequada.

Alguns modelos multivariados, utilizando *LOC* e #E como preditores foram avaliados, e suas principais estatísticas R^2, e os *p-values* de #E e LOC estão apresentadas na Tabela 3.2. Em todos eles, LOC tem um impacto mais significativo sobre o ajuste do modelo do que #E. Em três dos ajustes concebidos para explicar #DT, #E não é significativo, *p-value* maior do que 0.1 [Nisbet et al., 2009], [Everitt and Hothorn, 2010].

Tabela 3.2: Estatísticas de modelos multivariados de LOC e #E para *#DT* e *TT*.

Preditada	**Regressão**	**Modelo**	**Transformação**	R^2	***p-value* de *E***	***p-value* de LOC**
DT	Multi-linear	$DT = a + b * E + c * LOC$	$DT \sim E + LOC$	89.68%	0.91	6.18e-11
	Multi Power	$DT = a * E^b * LOC^c$	$log(DT) \sim log(E) + log(LOC)$	88.60%	0.42	1.13e-5
	Polynomial	$DT = a + b * LOC + c * E^2$	$DT \sim LOC + E^2$	90.07%	0.24	2.60e-13
	Polynomial	$DT = a + b * E + c * LOC^2$	$DT \sim E + LOC^2$	86.65%	1.11e-5	6.72e-9
TT	Multi-linear	$TT = a + b * E + c * LOC$	$TT \sim E + LOC$	98.13%	0.0003530	$<$ 2e-16
	Multi Power	$TT = a * E^b * LOC^c$	$log(TT) \sim log(E) + log(LOC)$	98.53%	0.003360	2.29e-16
	Polynomial	$TT = a + b * LOC + C * E^2$	$TT \sim LOC + E^2$	98.05%	0.0007680	$<$ 2e-16
	Polynomial	$TT = a + b * E + c * LOC^2$	$TT \sim E + LOC^2$	96.04%	1.68e-14	2.61e-14

Como esperado, identifica-se uma forte correlação[6] positiva, entre *LOC* e o total de termos, *TT*, de um sistema com $\rho = 0.98648$. Este resultado confirma a noção intuitiva que quanto maior é o programa, mais termos ele tem. Não tão intuitivo, também se encontra uma forte correlação positiva entre *LOC* e o número de termos distintos *DT*, com $\rho = 0.94697$. Tal fato confirma que quanto maior for o programa, mais termos distintos ele é propenso a ter. Isto pode estar relacionado com o fato de que cada parte de código tenda a implementar um único aspecto ou *feature* de um sistema que provavelmente tem seu próprio subconjunto de termos distintos. Esta conjectura, no entanto, precisa ser testada.

Sumarizamos a influência do tamanho dos sistemas em *LOC*, no tamanho de seus vocabulários, através de modelos que fossem capazes de explicar essa relação inclusive de forma

[5]Multicolinearidade consiste em um problema comum em regressões, onde as variáveis independentes possuem relações lineares exatas ou aproximadamente exatas.

[6]Utilizamos Correlação de *Spearman* já que os dados não seguem uma distribuição normal.

Tabela 3.3: Regressão Linear para *DT* e *TT*.

Preditado	Regressão	Preditora	Model	Transformação	R^2	*p-value*	resíduos
DT	**Linear**	**LOC**	$DT = a + b * LOC$	$DT \sim LOC$	**89.67%**	< **2.2e-16**	inválidos
	Linear	E	$DT = a + b * E$	$DT \sim E$	65.64%	4.16e-10	inválidos
	Exponencial	LOC	$DT = a * b^{LOC}$	$log(DT) \sim LOC$	68.38%	8.77e-11	válidos
	Exponencial	E	$DT = a * b^{E}$	$log(DT) \sim E$	65.25%	5.13e-10	válidos
	Power	**LOC**	$DT = a * LOC^{b}$	$log(DT) \sim log(LOC)$	**88.39%**	< **2.2e-16**	**válidos**
	Power	E	$DT = a * E^{b}$	$log(DT) \sim log(E)$	80.39%	1.18e-14	válidos
TT	**Linear**	**LOC**	$TT = a + b * LOC$	$TT \sim LOC$	**97.32%**	< **2.2e-16**	inválidos
	Linear	E	$TT = a + b * E$	$TT \sim E$	79.78%	2.08e-14	inválidos
	Exponencial	LOC	$TT = a * b^{LOC}$	$log(TT) \sim LOC$	68.75%	7.03e-11	válidos
	Exponencial	E	$TT = a * b^{E}$	$log(TT) \sim E$	72.47%	6.55e-12	válidos
	Power	**LOC**	$TT = a * LOC^{b}$	$log(TT) \sim log(LOC)$	**98.13%**	< **2.2e-16**	**válidos**
	Power	E	$TT = a * E^{b}$	$log(TT) \sim log(E)$	90.24%	< 2.2e-16	válidos

preditiva. Para selecionar entre vários ajustes (linear, exponencial, potência, hiperbólico, polinomial) candidatos, com o auxílio da *suite R*[7] fizemos uso da técnica *Stepwise Regression*[8,9] [Pitt and Myung, 2002; Wasserman, 2004; Larsen, 2012], considerando os seguintes critérios: 1) Análise de Resíduos, cujas características não inviabilizem o modelo [John M. et al., 1983]; 2) R^2-*Adjusted* [Jain, 1991], com $R^2 \geq 0.7$; 3) *t-Test* dos coeficientes, onde o *p-value* deve ser significativo, *i.e.* $p \leq 0.01$ [Larsen, 2012];

Além disso todas as regressões simples tomando *LOC* como preditor, também foram computados tendo #E como preditor, como pode ser observado na Tabela 3.3. Analisando o R^2 (Coeficiente de Determinação) e os erros residuais gerados quando da regressão de cada um dos modelos candidatos, identificamos a predominância do maior R^2 ou para os modelos linear ou para o de potência. No entanto, mesmo nos casos em que o R^2 para o modelo linear é maior, o de potência é também significativo, mas a análise sobre os erros residuais das regressões lineares indicam características (falta linearidade entre preditor e preditado, ausência de normalidade nos erros, presença de *outliers*) que inviabilizam o modelo [John M. et al., 1983].

Especificamente identificamos que tanto o quantitativo de termos distintos, únicos,

[7] O *R* é uma *suite* estatística que inclui aplicação e linguagem de programação, para tratamento, exploração e teste sobre dados [GNU,].

[8] Técnica de seleção de modelos que de forma semi-automático e sucessivamente adiciona ou remove variáveis preditoras do modelo de acordo com critérios estatísticos definidos [Nau,].

[9] Indicada esse estudo por ser uma análise exploratória, que conta com o auxílio de uma ferramenta estatística computacional [Hjorth, 1989] e onde os modelos analisados possuem apenas uma variável preditora.

quanto o total de termos, são modelados como uma função de potência sobre o tamanho em *LOC* do sistema, como descritas nas equações a seguir:

$$DT = \lfloor 4.759011602 * LOC^{0.50324} \rfloor \tag{3.4}$$

$$TT = \lfloor 0.650710784 * LOC^{0.97309} \rfloor \tag{3.5}$$

onde: $\lfloor x \rfloor$, função piso que converte o número real x no maior número inteiro menor ou igual a x.

3.2.2 Ocorrência de Termos

Já para entender se as repetidas ocorrências de cada termo de um vocabulário poderia ser modelado como fenômeno comportamental dos desenvolvedores, testamos distribuições estatísticas cuja a probabilidade de ocorrer uma variável aleatória se assemelhasse as características de ocorrência dos termos.

Quando postos num gráfico de dispersão, em escala logaritmíca, os pontos formados pelo ranque de termos e pelas repetidas ocorrências de cada termo, para cada um dos 39 projetos, como apresentado na Figura 3.2, não determinam claramente uma reta. Logo, não é possível afirmar que os dados seguem ou não uma distribuição Cauda Longa [Zhang, 2009], [Jain, 1991], [Feitelson, 2009]. Nesse caso, utilizamos testes de aderência a distribuições conhecidas [Johnson et al., 1994; Dalgaard, 2008], e mais especificamente o teste estatístico *Vuong*, desenvolvido por Clauset [Clauset et al., 2009], para detectar a existência ou não de distribuições Caudas Longa, e se detectadas classificar o seu tipo.

Baseado nas evidências estatísticas, a frequência dos termos ao longo de um vocabulário é regida por uma distribuição Cauda Longa do tipo log-normal com uma média μ e um desvio padrão σ, de forma que a frequência do *i*-ésimo termo é dada por:

$$F(term^{ith}) = \lfloor f(i, \mu, \sigma) * TT \rfloor \tag{3.6}$$

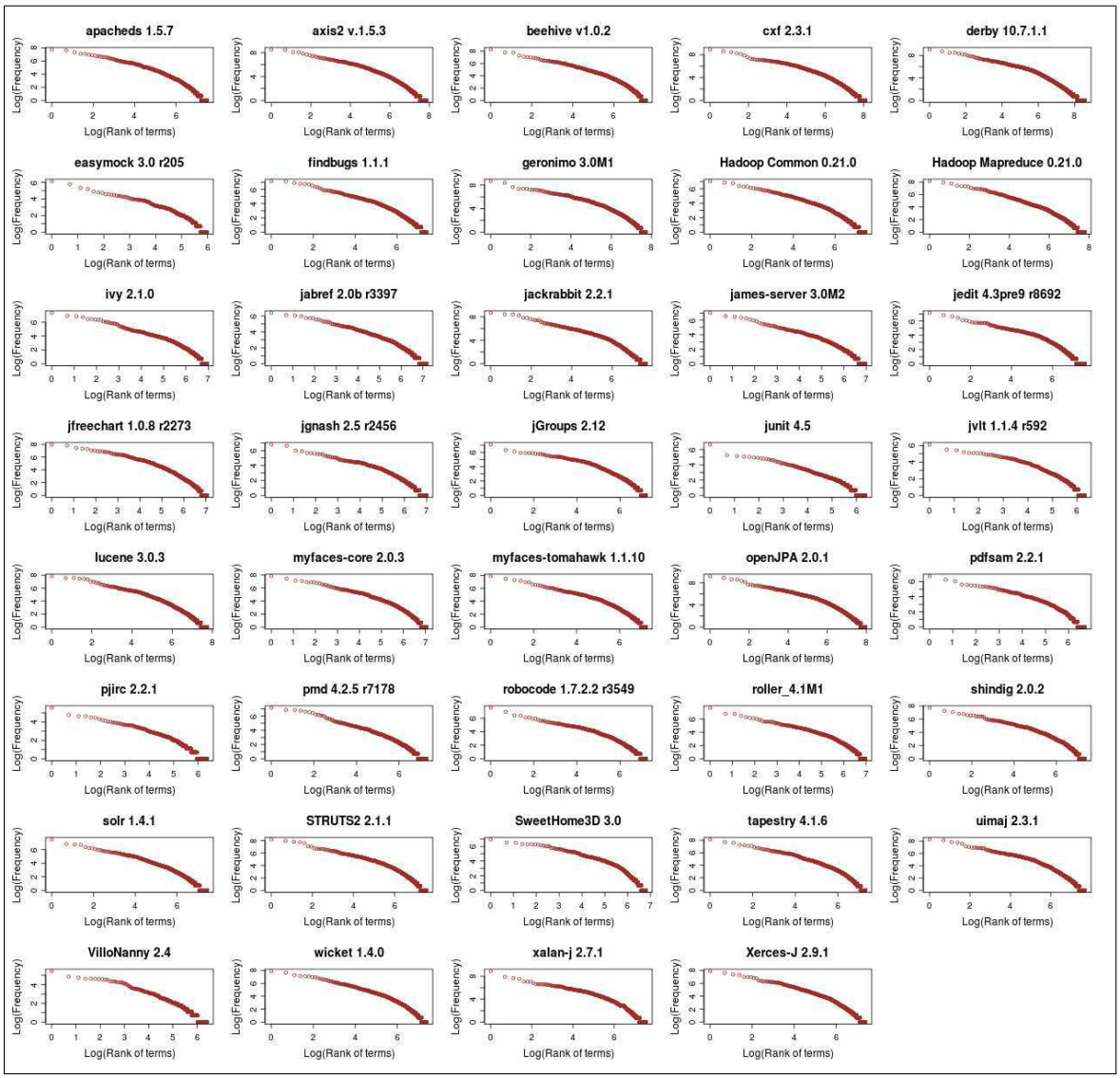


Figura 3.2: Gráfico de dispersão *log (frequência de termos)* por *log (classificação do termos)* para os vocabulários de cada um dos 39 projetos.

onde: $f(i, \mu, \sigma)$ é a log-normal que computa a probabilidade do aparecimento do i-ésimo termo; i é a posição no ranque do i-ésimo termo em ordem decrescente de frequência, $i = 1^o, 2^o, ..., DT^o$.

Essa nossa estratégia para compreender vocabulários bem como os resultados estão no artigo *Towards a Prediction Model for Source Code Vocabulary* publicado no *Workshop Next Five Years of Text Analysis in Software Maintenance (TAinSM)*, evento satélite ao *International Conference Software Maintenance (ICSM 2012)* [Santos et al., 2012], Trento - Itália. Apêndice C. Mais detalhes, como por exemplo todas estatísticas envolvidas e os refinamen-

tos sucessivos até se chegar no modelo final, encontram-se no relatório técnico *Relacionando Medida de Tamanho de Código Fonte LOC com a Natureza do Vocabulário Léxico de Projetos Java Open Source* [Santos, 2011].

3.3 Conclusões

Os modelos são recursos metodológicos e instrumentos de abstração destinados à aquisição de novos conhecimentos, representação e compreensão da realidade [Booch et al., 2000], [Travassos, 2011]. O modelo ideal para representar vocabulário de software deveria ser capaz de capturar aspectos sociais, onde estão inseridos os desenvolvedores, bem como suas preferências individuais. Nesse caso, os métodos qualitativos seriam os mais adequados [Zelkowitz and Wallace, 1998], [Singer et al., 2002], apesar da dificuldade de sua aplicação e da complexidade, principalmente na realidade da Engenharia de Software [Seaman, 1999], [Singer et al., 2002].

Também os modelos tem como uma de suas principais funções a exploração e a redução da complexidade de fenômenos [Sayão, 2001], [Wazlawick, 2010]. É factível então, o uso de métodos científicos alternativos [Kitchenham et al., 2002] que mesmo reduzindo a proximidade do modelo à realidade do vocabulário, ainda assim possibilite sua compreensão.

A estratégia de exploração estatística sobre vocabulários por nós utilizada demonstrou produzir conhecimento. Como resultado, identificamos relações entre as propriedades do vocabulário com o tamanho do código fonte do qual eles são extraídos. Sumarizamos esse conhecimento adquirido em modelos que podem ser utilizados como insumos para abordagens que auxiliem nas atividades de manutenção de sistemas através informações contidas em vocabulário [Binkley and Lawrie, 2009; Binkley and Lawrie, 2011; Pollock et al., 2013].

Essas informações, por sua vez, agora podem ser expressas sem ambiguidades devido a formalização sobre vocabulário como um conceito, com operações e propriedades. Também, com essa concepção formal, além de definir vocabulário de um trecho de código, nos é per-

mitido falar de vocabulário de um método, de uma classe, ou de um sistema como um todo, ou de um desenvolvedor, ou de uma tarefa, ou de um período de atividade no projeto. Também podemos definir vocabulários de outros artefatos de software que não sejam código. Por exemplo, podemos falar do vocabulário de *user stories*, de casos de uso, de documentos de requisito, de *bug reports*, de *change requests* e até de *emails* trocados entre desenvolvedores.

De acordo com literatura revisada e no melhor do nosso conhecimento, quase nada tem sido explorado acerca das informações contidas nos vocabulários de código com o objetivo de recuperar automaticamente desenvolvedores especialistas. Assim, no restante desta Tese focamos em investigar vocabulário como insumo a modelos de conhecimento sobre o código fonte.

Capítulo 4

Abordagem baseada em Vocabulário para Identificação de Especialistas

Aqui neste capítulo apresentamos uma descrição detalhada da abordagem, por nós proposta, que é baseada na similaridade entre os vocabulários das entidades de código e dos desenvolvedores para de forma automática identificar especialistas de código fonte.

4.1 Motivação

Conhecimento é um conceito subjetivo, difícil de ser capturado com medidas diretas, onde o que importa são os efeitos que seus modelos conseguem capturar sobre o processo e/ou sobre o produto. Diversos trabalhos que propõem abordagens para tentar automática e corretamente identificar especialistas [Mockus and Herbsleb, 2002; Girba et al., 2005; Fritz et al., 2010], sugerem também que é preciso aprimorar os modelos utilizados para representar o conhecimento dos desenvolvedores sobre um código. Pois, existem cenários conhecidos em que modelos baseados em autoria e propriedade são falhos ou insuficientes.

Por exemplo, suponhamos um desenvolvedor D_1 que a cada hora trabalhada registre no *VCS* do projeto, através de *commits*, o código implementado, e que no final do dia tenha como resultado a classe Calculadora, como descrito na Listagem 4.1, onde o método *soma* reflete a ausência de indentação no estilo de programação também dos demais métodos.

Código Fonte 4.1: Calculadora exemplo: métodos não indentados.

```
1  public class Calculadora {
2  public int soma (int valor) {
3  int retorno = 0;
4  while (valor >= 0) {
5  retorno += valor;
6  valor--;
7  }
8  return(retorno);
9  }
10 public int diminui(int valor) {
11 ...
12 }
13 public int multiplica(int valor) {
14 ...
15 }
16 public int divide(int valor) {
17 ...
18 }
19 }
```

Consideremos agora que um desenvolvedor D_2 ao tentar fazer uso da classe Calculadora, ajustou a indentação de todos seus métodos ao seu estilo de programação, e registrou essa mudança no *VCS* com um único *commit*. O novo código da Calculadora, com seus métodos indentados, está descrito na Listagem 4.2.

Código Fonte 4.2: Calculadora exemplo: métodos indentados.

```
1  public class Calculadora {
2      public int soma (int valor)
3      {
4          int retorno = 0;
5          while (valor >= 0) {
6              retorno += valor;
7              valor--;
8          }
9          return(retorno);
10     }
11     public int diminui(int valor)
12     {
13         ...
14     }
15     public int multiplica(int valor)
16     {
17     ...
18     }
19     public int divide(int valor)
20     {
21     ...
22     }
23 }
```

Nas ferramentas de desenvolvimento atuais é fácil existir a funcionalidade de ajuste de indentação de código o que torna comum a mudança executada pelo desenvolvedor D_2 sobre o código criado por D_1. Nesse cenário, a identificação de especialistas tanto com base no percentual de *LOC* modificadas por desenvolvedor quanto a baseada na quantidade de *commits* também por desenvolvedor, erroneamente atribuem créditos de conhecimento a D_2 sem que ele tenha lido o código. No entanto, o vocabulário da entidade Calculadora, $V(Calculadora)$, se manteve, mesmo depois das mudanças na indentação e que em notação de somas formais é:

$$V(Calculadora) = valor'7 + retorno'3 + Calculadora'1$$
$$+soma'1 + diminui'1 + multiplica'1 + divide'1$$

Para uma entidade fazer parte de um sistema de software é preciso que ela tenha sido inserida no *VCS* do projeto em algum momento por um dado desenvolvedor. Da perspectiva das abordagens baseadas no número de *commits* e em percentual de *LOC* modificadas, toda entidade tem sempre seu respectivo especialista, mesmo que ela tenha sido manipulada apenas uma vez: na sua criação pelo seu autor. Não é raro termos projetos com alta rotatividade na equipe de desenvolvimento. Se por algum motivo, o autor de uma dada entidade desfalca a equipe, essa entidade estaria órfã [Bittencourt et al., 2010] de especialista. Além disso, as abordagens baseadas em autoria e propriedade não têm subsídios para indicar um outro desenvolvedor que assumisse a entidade órfã.

Da perspectiva de vocabulário, quanto mais as entidades de um sistema compartilham termos em comum, mais conceitualmente similares elas são [Kuhn et al., 2007; Corley et al., 2012; Santos et al., 2013] Logo, utilizando similaridade de vocabulários, é factível indicar entre os membros da equipe aquele que teria o maior potencial de adquirir conhecimento sobre uma dada entidade, em outras palavras, aquele cuja similaridade de seu vocabulário fosse a mais próxima possível ao vocabulário da entidade. Esse característica seria muito útil

em pelo menos duas situações:

- Indicar desenvolvedores para adotar entidades órfãs. Nesse caso, poderia ser utilizada em complemento às abordagens baseadas apenas em dados de autoria e de propriedade;

- Balancear a carga de atividades de manutenção entre desenvolvedores. Nesse caso, seria utilizada para recomendar desenvolvedores para atuarem em entidades em que eles não são os maiores especialistas. Por exemplo, em momentos em que há uma demanda de atividades de manutenção sobre entidades cujos respectivos especialistas estão nos seus limites de carga, a recomendação correta de desenvolvedores outros capazes de mais facilmente assimilar as entidades pode minimizar a carga sobre os especialistas originais.

Observando cenários como descrito anteriormente, e analisando como neles se comportam os identificadores e comentários, conjecturamos que vocabulários tem sua relevância para constituir um modelo de conhecimento para identificação de especialistas. Tomando como premissa que vocabulário carrega conhecimento de desenvolvedores, quanto mais um desenvolvedor manipule o vocabulário de uma entidade, mais ele domina os conceitos, abstrações e funcionalidades que ela representa. Consequentemente, mais familiaridade com o código fonte da entidade o desenvolvedor tem.

4.2 Visão Geral

Para identificar quem são os especialistas em cada parte do código fonte de um projeto, desenvolvemos, como parte desta Tese, uma abordagem que utiliza como modelo de conhecimento a similaridade entre o vocabulário presente nas entidades de código e o vocabulário manipulado por cada um daqueles que ao longo da evolução do projeto participou como integrante da equipe de desenvolvimento. Até onde temos conhecimento, essa abordagem é inovadora e para sua melhor compreensão, a Figura 4.1 apresenta o fluxo dos passos que compõem a abordagem, onde cada um deles é responsável por:

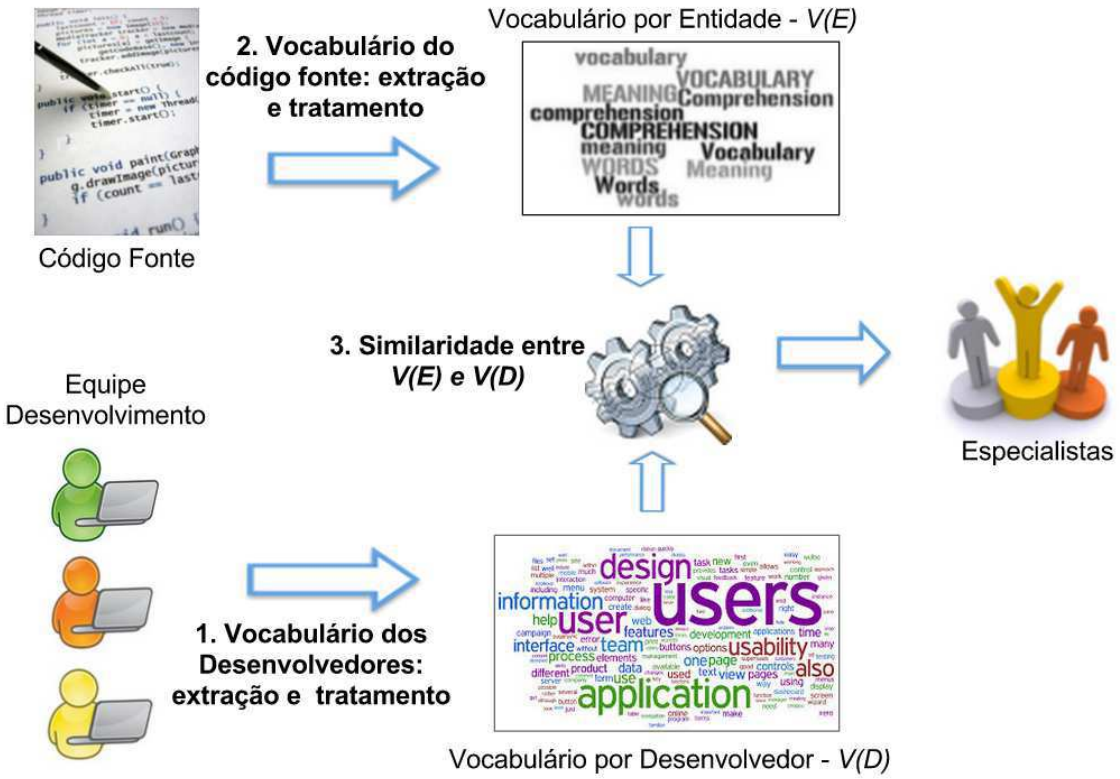


Figura 4.1: Abordagem para Identificação de Especialistas baseada na Similaridade de Vocabulário.

1. coletar as contribuições dadas por cada um dos desenvolvedores sobre o vocabulário. Consiste em percorrer todo o repositório alimentando o modelo de conhecimento, baseado no vocabulário dos desenvolvedores, com os dados extraídos de cada modificação;

2. extrair o vocabulário das entidades para a versão do código fonte de interesse [1]; extrair seu vocabulário, e o granularizá-lo por entidade de *design*;

3. por fim, para uma dada entidade da versão de interesse, computar a similaridade do seu vocabulário com o vocabulário de cada um dos desenvolvedores. Aquele que possuir o maior valor de similaridade é considerado o desenvolvedor mais adequado para manipular a entidade, *i. e.*, o especialista do código da entidade.

[1] em inglês chamada de *target version* que é a versão do código fonte, sobre a qual será realizada uma tarefa de manutenção.

4.3 Extração dos Vocabulários de Entidades

Após seu processamento pelo ferramental *Vocabulary Tool*, o vocabulário do código fonte de um sistema é representado por uma uma matriz de frequência Termo-Entidade, **TE**, onde cada célula contém o número de ocorrências do *termo* t_i na entidade E_j. Na matriz TE apresentada na Tabela 4.1 a seguir, fazem parte do vocabulário da entidade E_1 os termos t_1, t_2, t_3: t_1 ocorre duas vezes, t_2 e t_3 uma vez cada, totalizando 4 ocorrências de termos.

Tabela 4.1: Exemplo de matriz de frequência Termo-Entidade, TE.

Termos	Entidades				
	E_1	E_2	E_3	E_4	E_5
t_1	2	0	1	0	0
t_2	1	2	0	0	0
t_3	1	0	1	0	1
t_4	0	0	1	1	0
t_5	0	0	0	0	1

Os valores das células de toda uma coluna da matriz TE, *i.e.* de uma dada entidade, representam as multiplicidades dos termos que compreendem o vocabulário daquela entidade. Em notação de somas formais, como descrito na Seção 3.1.1, o vocabulário de E_1 é dada por $V(E_1) = 2't_1 + 1't_2 + 1't_3$, seu tamanho é dado por $TT(V(E_1)) = V(t_1) + V(t_2) + V(t_3) = 4$ e por $DT = |(t_1, t_2, t_3)| = 3$. Já o vetor de frequência característico de E_1 é definido por $CFV(V(E_1)) = (2, 1, 1)$.

4.4 Geração do Vocabulário de Desenvolvedores

O vocabulário de um desenvolvedor é resultante das suas contribuições sobre o vocabulário do código [Matter et al., 2009]. Cada contribuição é capturada pela diferença (*diff*) entre os vocabulários do código fonte anterior e do posterior a um dado *commit*, realizado por um desenvolvedor. O *diff* entre vocabulários é calculado em função dos termos adicionados ou removidos após o *commit*. E assim como um vocabulário de entidade, seu resultado é uma matriz de frequência Termo-Entidade. É responsabilidade da aplicação *Extractor of Developer Vocabulary* gerar o vocabulário de cada desenvolvedor.

O vocabulário de cada desenvolvedor, $Voc(D)$, é também representado por uma matriz

de frequência TE^D, onde o expoente D identifica o desenvolvedor. Para uma matriz TE usaremos a seguinte estratégia. Inicialmente as matrizes TE dos desenvolvedores é vazia, mas a medida que cada *commit* é detectado, identifica-se o desenvolvedor responsável, e extraí-se apenas os arquivos que foram modificados em relação à versão do sistema anterior ao *commit*. Com esses arquivos gera-se o vocabulário de um *commit* que é constituído apenas das entidades modificadas. Esse vocabulário é finalmente acumulado ao vocabulário do desenvolvedor responsável, aquele que executou o *commit*, atualizando assim sua matriz TE. Em notação de somas formais, como descrito na Seção 3.1.1, o vocabulário de D é dada por $V(D) = \sum_{i=1}^{n} V(E_i)'t_1 + \sum_{i=1}^{n} V(E_i)'t_2 + \ldots \sum_{i=1}^{n} V(E_i)'t_{DT}$, onde cada frequência de um termo t é a soma das frequências de t em cada entidade E da matriz TE do desenvolvedor.

A geração do vocabulário de desenvolvedores não contabiliza, para efeito de conhecimento de código, nem mudanças no número de linhas das instruções de programas de múltiplas linhas nem ajustes no estilo de indentação de código. Assim, diferentemente das abordagens existentes que extraem informações dos *VCS*, a nossa abordagem por concepção impede que atribuições de conhecimento decorrentes desses cenários aconteçam. Por exemplo, a maioria das ferramentas de desenvolvimento atuais tem opções de edição de texto para ajustar automaticamente a formatação do código fonte. Considerando as duas abordagens que usamos como *baseline*, se esse recurso for utilizado e o código alterado sofrer um *commit*, o crédito de conhecimento sobre o código seria erroneamente atribuído a quem tiver feito uso da opção de edição.

Na Figura 4.2 são apresentados exemplos de momentos de contribuições de desenvolvedores sobre o vocabulário do projeto: da Versão 0, $Voc(V_0)$, até a Versão 3, $Voc(V_3)$. Enquanto não houver qualquer contribuição de código concretizada no repositório, o vocabulário do sistema encontra-se vazio $Voc(V_0) = \emptyset$, assim como também estão vazias as matrizes de frequência TE^{D_1} e TE^{D_2} que representam respectivamente os vocabulários dos desenvolvedores D_1 e D_2.

Quando o projeto se encontra na Versão V_1, seu vocabulário $Voc(V_1)$ é resultante do *commit* C_1, realizado pelo desenvolvedor D_2. O $Voc(D_2)$, antes vazio, é agora $Voc(D_2) =$

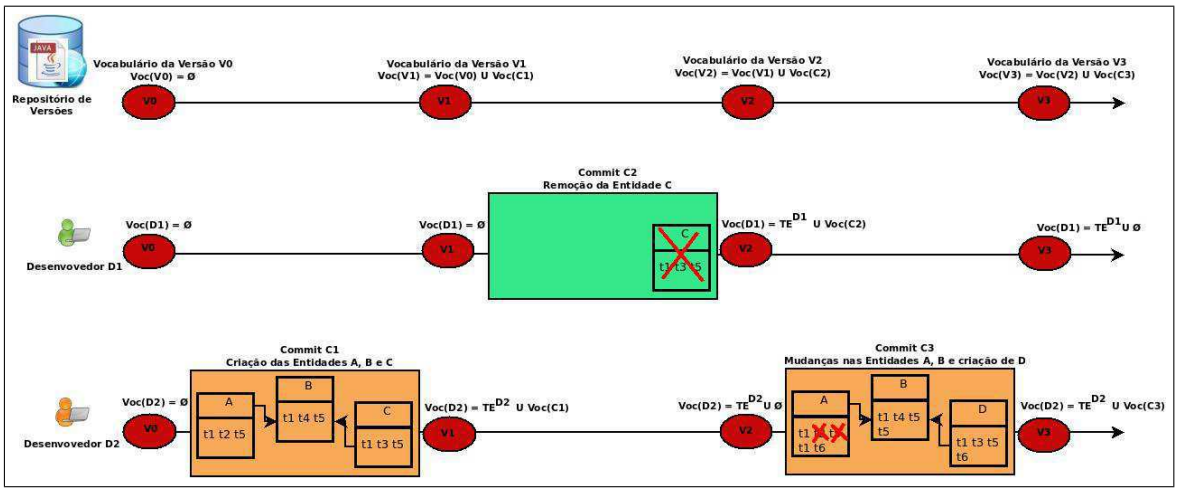


Figura 4.2: Geração do Vocabulário de Desenvolvedores.

$TE^{D_2} \cup Voc(C_1) = Voc(C_1)$. Quando na Versão V_2, o vocabulário do Desenvolvedor D_2 não se altera, mas o vocabulário de D_1 que era vazio passa a ser $Voc(D_1) = TE^{D_1} \cup Voc(C_2) = Voc(C_2)$. Já na Versão V_3 do projeto, o $Voc(D_1)$ permanece inalterado, enquanto que o $Voc(D_2)$ passou a ser $Voc(D_2)' = Voc(D_2) \cup Voc(C_3) = Voc(C_1) \cup Voc(C_3)$.

4.5 Relacionando Vocabulário de Desenvolvedores com de Entidades

É através da similaridade entre os vocabulários dos desenvolvedores e das entidades que são apontados os especialistas. 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 [Baeza-Yates and Ribeiro-Neto, 1999]:

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

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

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.

Para todo desenvolvedor D contribuinte com o projeto P, calcula-se a similaridade do cosseno [D. Manning et al., 2008] do seu vocabulário com o vocabulário de cada uma das entidades presentes na matriz TE_P. O resultado é então uma matriz de similaridade Entidade-Desenvolvedor do projeto P, ED_P, como exemplificado na Tabela 4.2 definida a seguir. ED_P contém i linhas que representam o total de entidades da versão de interesse, e j colunas que representam o número de desenvolvedores contribuintes.

Tabela 4.2: Matriz de similaridade do cosseno entre vocabulários de Entidades e de Desenvolvedores para um Projeto P.

Entidades	Desenvolvedores			
	D_1	$D_{...}$	D_{j-1}	D_j
E_1	$simCos(E_1, D_1)$	$simCos(E_1, D_{...})$	$simCos(E_1, D_{j-1})$	$simCos(E_1, D_j)$
E_2	$simCos(E_2, D_1)$	$simCos(E_2, D_{...})$	$simCos(E_2, D_{j-1})$	$simCos(E_2, D_j)$
$E_{...}$	...	...	...	...
E_i	$simCos(E_i, D_1)$	$simCos(E_i, D_{...})$	$simCos(E_i, D_{j-1})$	$simCos(E_i, D_j)$

Para uma dada entidade E_i, o desenvolvedor D_j que apresentar o maior valor de similaridade do seu vocabulário com o vocabulário de E_i, é aquele recomendado como desenvolvedor especialista para E_i.

Em nossos experimentos para construção da nossa abordagem de identificação de especialistas mensuramos além do cosseno outras medidas de similaridade: distância máxima, distância euclidiana, manhattan e canberra. A Tabela 4.3, apresenta os valores de similaridade em ordem crescente para os cenários com *Top-1*, *Top-2* e *Top-3* especialistas, considerando 2 meses de acúmulo de vocabulários de desenvolvedores. Como observado, a similaridade

do cosseno atingiu os maiores valores para todos os cenários, o que justificou a sua escolha para ser utilizada na nossa abordagem.

Tabela 4.3: Comparativo entre medidas de similaridade para os *Top-1*, *Top-2* e *Top-3* especialistas considerando 2 meses de acúmlo de vocabulário.

Medidas de Similaridade	*top-k*	2 meses de vocabulário	
		precisão	cobertura
Cosseno	1	0.2500	0.2500
Distância Máxima		0.0714	0.0714
Distância Euclidiana		0.0714	0.0714
Manhattan		0.0714	0.0714
Canberra		0.0357	0.0357
Cosseno	2	0.1964	0.2750
Distância Euclidiana		0.1964	0.2750
Canberra		0.1786	0.2500
Distância Máxima		0.1786	0.2500
Manhattan		0.1786	0.2500
Cosseno	3	0.1905	0.3478
Distância Euclidiana		0.1905	0.3478
Distância Máxima		0.1786	0.3261
Canberra		0.1667	0.3043
Manhattan		0.1667	0.3043

É possível, ainda, recomendar não apenas um único especialista, mas sim os k desenvolvedores que possuem conhecimento sobre E_i. Saber os *top-k* é um fator importante em situações onde exigir uma recomendação exata do maior especialistas não faz tanto sentido [Kagdi and Poshyvanyk, 2009; Morales-Ramirez et al., 2014]. Por exemplo, quando se quer identificar um desenvolvedor que possa fazer uma revisão de código, ou que possa fazer uma mudança sobre uma Entidade no caso do seu maior especialista se encontrar indisponível.

Uma outra vantagem dessa nossa abordagem em relação as baseadas em autoria e propriedade, é seu potencial para recomendar especialistas para entidades órfãs de um sistema. Neste cenário, seria factível identificar entre os membros da equipe atual aquele cujas contribuições sobre os vocabulários de outras entidades poderia compreender rapidamente o código da entidade órfã, e sobre a qual uma dada tarefa de manutenção deveria ser executada.

4.6 Vocabulário Adicionado a Modelos Existentes

Como descrito na Seção 2.2.1 é preciso combinar medidas de conhecimento para tentar avançar na acurácia das abordagens que identificam especialistas. Por exemplo, o modelo *Degree-Of-Knowledge (DOK)*, descrito na Seção 2.2.2 combina componentes de autoria através do modelo *Degree-Of-Authorship (DOA)* com a componente de interação (navegação) sobre o código, o modelo *Degree-Of-Interest (DOI)*. Para computar o *DOI*, e consequemtemente o *DOK*, é necessário instrumentar o processo de codificação com um ferramental capaz de capturar as seleções e edições dos desenvolvedores, o que para realidade de muitos projetos torna-se inviável.

Por ter seu modelo de conhecimento definido em função apenas da similaridade do cosseno, é factível adicionar a outros modelos de conhecimento de código a contribuição de vocabulários na identificação de especialistas.

Como definido na Seção 4.4 cada desenvolvedor tem seu vocabulário próprio constituído pelo acúmulo dos termos de cada contribuição sobre o código das entidades do sistema por ele realizada, ao longo de sua participação no projeto. Nesta Tese, denominamos grau de eloquência, em inglês *Degree-Of-Eloquence (DOE)* para a esse acúmulo de termos, e, quanto mais similar o vocabulário de um desenvolvedor é de um vocabulário de uma entidade, maior é o seu *DOE*.

Em outras palavras, podemos entender o *DOE* como a competência de um desenvolvedor em entender os conceitos, abstrações e funcionalidades através dos termos que compõem o vocabulário de uma entidade. O *DOE* de um desenvolvedor sobre um vocabulário sofre variações em função de mudanças ocorridas nos vocabulários das entidades:

- **Aumento da Eloquência**: Acontece a medida que um desenvolvedor autor manipula termos do vocabulário de uma entidade. É o fluxo positivo de eloquência;

- **Diminuição da Eloquência**: Ocorre quando outros desenvolvedores modificam o vocabulário da entidade, podendo causar uma diminuição na similaridade entre o vocabulário do desenvolvedor autor com o da referida entidade. É o fluxo contrário

(contra-fluxo) da eloquência.

Podemos então, adicionar a modelos de conhecimento de código já existentes a componente *DOE*. Adicionando o *DOE* ao modelo *DOK* original, temos uma componente que captura medidas diferentes das capturadas por *DOA* e *DOI*. Tomando a Equação 2.1 apresentada na Seção 2.2.2, o *DOK* acrescido de *DOE* é dado por:

$$DOK = \alpha_{FA} * FA + \alpha_{DL} * DL + \alpha_{AC} * AC + \beta_{DOI} * DOI + \gamma_{DOE} * DOE \quad (4.1)$$

onde: α, β_{DOI} e γ_{DOE}, são pesos para os fatores de *DOA*, *DOI*, e de *DOE* respectivamente.

Mesmo descartando *DOI* devido a inviabilidade de sua coleta, ainda assim temos um modelo que combina outros componentes além dos elementos que modelam autoria, que seria dado por:

$$DOE = \alpha_{FA} * FA + \alpha_{DL} * DL + \alpha_{AC} * AC + \gamma_{DOE} * DOE \quad (4.2)$$

onde: os coeficientes α são os pesos para os fatores de *DOA* e o γ_{DOE} é o peso para *DOE*.

4.7 Conclusões

Analisando a natureza e o comportamento dos vocabulários nos cenários onde abordagens de identificação de especialistas tradicionais atribuem erroneamente créditos de conhecimento, identificamos que vocabulários não endossam os mesmos erros. Apesar de prematuro, sem uma real avaliação, isso nos faz concluir que também para identificação de especialistas vocabulário captura conhecimento de código de uma perspectiva diferente de como os fazem os modelos já utilizados na prática.

Utilizando apenas a similaridade entre vocabulários definimos um modelo de conhecimento que indica os especialistas de entidades de código, e que adicionalmente é capaz de recomendar entre os desenvolvedores da equipe aquele que com mais facilidade adotaria entidades órfãs de especialista. Esse poder de recomendação, já é por si só uma grande van-

tagem frente as abordagens baseadas em autoria e propriedade que, por concepção, não têm de onde obter informação para realizar uma recomendação num cenário equivalente.

Contudo, é preciso averiguar na prática o potencial da abordagem proposta na identificação de especialistas considerando cenários corriqueiros de atribuição de conhecimento. É neste sentido que no próximo capítulo, 5, avaliamos a abordagem baseada em vocabulário, onde também apresentamos suas limitações, Seção 5.1.4.

Capítulo 5

Avaliação

No Capítulo 4, usamos a similaridade entre vocabulários de entidades e de desenvolvedores como modelo de conhecimento com potencial de identificar especialistas, e para recomendar desenvolvedores a entidades órfãs. Agora neste Capítulo, apresentamos dois estudos experimentais. No primeiro, medimos a acurácia da abordagem por nós proposta. Já o segundo tem como meta identificar o percentual de contribuição do modelo de conhecimento definido pela similaridade na identificação de especialistas.

5.1 Acurácia da Abordagem

Este primeiro experimento objetiva avaliar a acurácia da abordagem proposta em termos de precisão e cobertura, em relação a duas outras abordagens: uma baseada no número de *commits* realizados por cada um dos desenvolvedores [Mockus and Herbsleb, 2002; Hattori and Lanza, 2009], e a outra baseada no percentual de *LOC* modificadas por cada desenvolvedor [Girba et al., 2005; Rahman and Devanbu, 2011]

5.1.1 Questões de Pesquisa

As questões de pesquisa (QP) que buscamos responder através deste experimento foram as seguintes:

QP_1: **Nossa abordagem consegue identificar especialistas de código?**

Está é uma questão que automaticamente surge quando se propõe algo novo. Precisamos saber na prática se nossa abordagem é capaz de identificar especialistas.

QP_2: **Comparadas com as 2 abordagens de *baseline*, a precisão e a cobertura da nossa abordagem são melhores?**

Esta é uma importante questão que emerge como consequência da QP_1. Pois, não é suficiente saber se abordagem proposta identifica especialistas. É fundamental comparar seus resultados obtidos com os obtidos através de outras abordagem que servirão de *baseline*, e nos possibilitará atestar se a abordagem baseada na similaridade entre vocabulários é ou não a mais adequada para identificar especialistas de código.

5.1.2 Metodologia

A metodologia de execução deste experimento que foi divida nas seguintes etapas: 1) Seleção amostral; 2) Construção do Oráculo de especialistas por entidade; e, 3) Instrumentação e coleta de medidas;

Seleção Amostral

A versão do projeto ePol a que tivemos acesso foi a 0.5b liberada em 18 de fevereiro de 2014. Para essa versão a equipe desenvolvedora era constituída de doze programadores e um líder do projeto. Já em termos de tamanho, essa versão contém 907 entidades Java (entre classes e interfaces) codificadas ao longo de 96 *KLOC*, e, seu vocabulário de software é composto de 2.751 termos distintos. Os vocabulários de entidades foram extraídos versão 0.5b enquanto que os vocabulários do desenvolvedores forma coletados das versões que antecedem essa versão. Porém, geral o sistema ePol atende aos seguintes critérios:

- possui mais de 85% do código fonte desenvolvido em Java;
- tem identificadores codificados em português, e em estilo *camelcase* e/ou em *underscore*; comentários e *javadoc* escritos em português;

- faz uso de uma política de *commits*, individualizada por desenvolvedor;
- possibilitou acesso aos fontes e à equipe de desenvolvimento para a construção de Oráculo de especialistas por entidade.

Apesar de na prática entidades de testes também receberem manutenção, descartamos da amostra as 436 entidades cujas responsabilidades no fonte do projeto ePol é de apenas testar o código das 471 restantes. Tal fato se deu já que nesta Tese priorizamos definir os desenvolvedores mais adequados para realizarem tarefas de manutenção apenas sobre as funcionalidades disponíveis aos usuários do sistema.

Do ponto de vista de custo experimental, não é viável para um projeto real e em andamento alocar toda uma equipe de desenvolvimento para responder questionários e participar de reuniões com o objetivo de chegar ao consenso de especialistas para uma população de 471 indivíduos, nem muito menos de 907 caso as entidades de testes fossem consideradas. Assim, para não comprometer a produtividade da equipe de desenvolvimento, e minimizar os custos de tempo e de esforço na construção de um Oráculo, nos valemos da inferência estatística [Barbetta et al., 2008] sobre uma amostra de 50 ($n = 50$) entidades o que representa mais de 10% da população das 471 entidades funcionais.

Objetivando ter uma amostra mais representativa possível à realidade e em prol da qualidade da inferência, categorizamos a população em estratos ou grupos. Para isso, primeiramente computamos para toda população de entidades as 3 seguintes métricas:

1. FanIn. Aponta o número de classes que referenciam a entidade [Couto et al., 2012]. Também conhecida como *Efferent Coupling*;
2. LOC. Número de Linhas de Código que compõe a entidade;
3. CC (Complexidade Ciclomática). Quantidade de caminhos linearmente independentes existente no código fonte da entidade [McCabe, 1976].

Em seguida, os valores para cada métrica computada obtidos de todas as entidades foram divididos nos percentis 33º, 66ºe 100º. Como resultado, cada uma das 471 entidades pertence a um dos 27 $(3\ percentis^{(3\ \text{metricas})})$ grupos.

Definimos um grupo G_g, como um tripla $<F_f, L_l, CC_c>$ de métricas de entidades, onde $f, l, c \in \{33º, 66º, 100º\}$ percentis. E, a função de distribuição de entidades por grupo , para população ePol, é dada pela frequência relativa de entidades da população ($m = 471$) que pertence a cada um destes grupos, dada então por:

$$dfd(G_g) = \frac{\#\{E_e\}}{m}$$

onde $E_{e=\{1,...,m\}} \in G_{g=\{1,..,27\}}$ e m é a população de entidades.

A seleção de entidades para fazer parte de um dos grupos foi aleatória e sem reposição. Após ser sorteada, identificava-se o grupo ao qual a entidade pertencia e a submetíamos ao seguinte critério de descarte: se a proporção das entidades que devem estar na amostra de um grupo fosse igual a 0, a entidade era descartada e um novo sorteio seria realizado. Caso contrário, a entidade passaria a fazer parte da amostra e a proporção de entidades para aquele grupo era diminuída de um. O sorteio e o descarte foi realizado até que se completou o tamanho definido para a amostra, ($n = 50$) entidades.

Construção do Oráculo de Especialista por Entidade

Um Oráculo é a autoridade do conhecimento. Para o contexto desta Tese, no Oráculo estão identificados os especialistas para cada uma das entidades, classes ou interfaces, da amostra.

Devido à contínua evolução dos sistemas de software associado a rotatividade e a quantidade de desenvolvedores que codificam um mesmo projeto, é sempre um desafio decidir quem seria o membro mais adequado para executar uma tarefa de manutenção em uma determinada entidade de código. Situações como alta carga de trabalho, indisponibilidade por doença ou férias, pode ter feito com que uma tarefa de manutenção sobre uma dada entidade tenha sido executada por um outro desenvolvedor ao invés de por aquele mais conhecedor da entidade [Cavalcanti et al., 2014]. Apesar dos *logs* dos *VCS*, bem como os relatórios de *Bugs* corrigidos e de requisições de mudanças solucionadas fornecerem indicações satisfatórias, é o conhecimento que está espalhado entre todos os membros da equipe que realmente aponta

o desenvolvedor mais adequado para realizar uma tarefa de manutenção [Servant, 2013].

Para superar o desafio de construir um Oráculo de especialistas por entidade que pudesse refletir, da forma mais fidedigna possível, o conhecimento da equipe sobre o código, adotamos a seguinte estratégia:

1. Toda a equipe corrente de desenvolvimento, incluindo o gerente do projeto, foi previamente convocada para explicar a necessidade da criação do Oráculo em prol deste estudo. O objetivo era evitar que eventualmente os desenvolvedores criassem qualquer juízo de valor deturpado, por exemplo que estaríamos tentando aferir a produtividade de cada um.

2. Em reunião e na presença de todos os membros da equipe o código-fonte de cada entidade da amostra era apresentado, e questionava-se qual dos membros seria o mais adequado para realizar uma tarefa de manutenção sobre a entidade.

3. Voluntariamente, qualquer desenvolvedor expressava verbalmente quem entre os membros ele julgava ser o especialista da entidade que inclusive poderia ser ele mesmo. Em seguida, os membros restantes aderiam ou não a indicação. Em caso de divergência, quando novas indicações de especialistas eram levantadas, discussões de argumentos foram realizadas até que um consenso fosse alcançado.

Para dar suporte a essa estratégia construímos um ferramental para aplicação de um *survey* sobre entidades de código fonte Java. Foi através desse ferramental que cada uma das 50 entidades da amostra foi apresentada a todos os membros corrente da equipe de desenvolvimento do ePol. Além do código fonte da entidade, a ferramenta contextualiza a localização da entidade dentro projeto através da apresentação da navegação sobre o código fonte das entidades que fazem referência (*callers*) e as que são referenciadas (*callees*) pela entidade em foco.

A execução do *survey* com a equipe de desenvolvimento foi previamente agendada para mitigar ausência de membros, com horários de início e fim estabelecidos, e executado na

presença de pesquisadores para mediar possíveis conflitos entre os membros. Além disso, todo o áudio da reunião foi gravada para que uma eventual auditoria pudesse ser realizada.

No final do *survey*, em 28 das 50 entidades inqueridas a equipe chegou a um consenso e estabeleceu quem seria pelo menos um especialista para cada entidade. Para as 22 demais a equipe, também de forma consensual, concordou que nenhum dos desenvolvedores correntes seria o mais adequado para realizar sobre cada uma delas tarefas de manutenção, tornando essas entidades órfãs.

Da perspectiva da nossa abordagem, baseada em similaridade, essas 22 entidades órfãs poderiam ser mantidas no Oráculo, pois como descrito na Seção 4.5, elas poderiam ser adotadas por algum desenvolvedor corrente da equipe cujo vocabulário mais se aproximasse dos respectivos vocabulários das órfãs. Contudo, como as abordagens que usamos como *baseline* não têm insumos para recomendar entre desenvolvedores correntes às entidades órfãs, decidimos por descartá-las. Além disso, esse descarte garante o princípio de isonomia quando na aferição de precisão e de cobertura na identificação especialistas.

O Oráculo ficou então constituído de 28 entidades, 5.94% das 471 entidades da população que estatisticamente continua sendo representativa [Barbetta et al., 2008]. A sumarização dos valores de conhecimento de cada desenvolvedor sobre uma entidade do Oráculo resulta na definição dos especialistas relevantes por entidade. A Tabela 5.1, apresenta a tabulação de um Oráculo resultante fictício. Para E_1 apenas M é o especialista, para E_2 P e M são considerados especialistas, e assim continua até a última entidade E_n para qual M, P e N são os especialistas indicados.

Tabela 5.1: Oráculo exemplo de entidades com seus respectivos especialistas.

Entidades	Especialistas
E_1	M
E_2	P, M
$E_{...}$	P, M
E_n	M, P, N

Instrumentação e Coleta de Medidas

Para coletar informações de acurácia da abordagem proposta e dela extrair seu percentual de participação na identificação de especialistas, instrumentamos este experimento com as seguintes ferramentas:

- **Identificador de Especialista através de Similaridade de Vocabulários**. Consistiu na implementação da identificação de especialistas através da similaridade entre vocabulários de entidades e de desenvolvedores como descrito no Capítulo 4. Para tanto, fizemos uso de ferramental de apoio já existente, o *Vocabulary Tool*, e o desenvolvimento do *Extractor of Developer Vocabulary* para gerar o vocabulário de cada desenvolvedor. O *Vocabulary Tool* foi configurado para: extrair os identificadores que nomeiam classes, interfaces, enumerações, atributos, métodos, parâmetros e variáveis locais; capturar os textos de comentários e de *javadocs*; tratar identificadores codificados tanto em notação *camelcase* como *underscore*; e extrair o radical de palavras escritas em língua portuguesa desconsiderando termos formados por menos de 3 caracteres.

- **Identificador de Especialistas através das Abordagens *Baseline***: As 2 abordagens, por *commits* e por percentual de *LOC* modificadas, que servirão de *baseline* estão implementadas no ferramental *AuthorshipExtractor*.[1]

Após construção dos insumos ferramentais submetemos a amostra de entidades do projeto ePol a cada um dos 3 tratamentos possíveis para o fator *Tipo de Abordagem*: similaridade entre vocabulários, número de *commits*, percentual de *LOC* modificados computando para cada um deles sua acurácia em termos de precisão e cobertura.

Há situações em que se necessita da identificação de possíveis especialistas, os *top-k*, e não apenas da recomendação exata do maior especialista. Por exemplo, quando se quer

[1]Ferramental desenvolvido pelo SPlab, que extrai informações dos gerenciadores de versões de código, Mercurial e SVN, e identifica os desenvolvedores que tenham contribuído com o código fonte de um dado projeto: tanto pelo número de *commits* realizados quanto pelo percentual de *LOC* modificadas por desenvolvedor [dos Santos et al., 2012].

um desenvolvedor para revisar um código, ou para fazer uma mudança sobre uma entidade quando na indisponibilidade de seu maior especialista. O sucesso ocorre se algum dos especialistas relevantes no Oráculo estiverem entre os *top-k* recuperados pela abordagem[2].

A acurácia de uma abordagem, um tratamento, é dada em função do conjunto formado pelos pares Entidade, Desenvolvedor Recuperado, (E_i, DR_j), recuperados pela abordagem, e pelo conjunto constituído pelos pares Entidade, Desenvolvedor Especialista, (E_i, DE_k), relevantes, *i.e.* definidos no Oráculo. A seguir estão definidas, respectivamente, as fórmulas para o cômputo da precisão[3] e da cobertura[4] de uma abordagem de identificação de especialistas.

$$precisão_{Abordagem} = \frac{\#\Big|\overbrace{\big((E_1, DR_1), \ldots, (E_i, DR_j)\big)}^{Relevantes} \cap \overbrace{\big((E_1, DE_1), \ldots, (E_i, DE_k)\big)}^{Recuperados}\Big|}{\#\Big|\underbrace{\big((E_1, DE_1), \ldots, (E_i, DE_k)\big)}_{Recuperados}\Big|} \tag{5.1}$$

$$cobertura_{Abordagem} = \frac{\#\Big|\overbrace{\big((E_1, DR_1), \ldots, (E_i, DR_j)\big)}^{Relevantes} \cap \overbrace{\big((E_1, DE_1), \ldots, (E_i, DE_k)\big)}^{Recuperados}\Big|}{\#\Big|\underbrace{\big((E_1, DR_1), \ldots, (E_i, DR_j)\big)}_{Relevantes}\Big|} \tag{5.2}$$

onde:

- $i = \{1, 2, \ldots, tamanho\ da\ amostra\ de\ entidades\}$;
- $j = \{1, 2, \ldots, total\ de\ desenvovedores\ recuperados\}$; e,
- $k = \{1, 2, \ldots, total\ de\ desenvolvedores\ especialistas\}$.

[2]Abordagens que recomendam uma lista de especialistas como [Kagdi et al., 2008] sugerem que *top-3* é um tamanho de lista razoável para iniciar as medições. Mas, o valor de *k* deve ser confirmado ao longo de experimentos considerando valores próximos a $k = 3$, tais como $k = 2$, $k = 4$ e $k = 5$.

[3]Fração dos resultados recuperados (identificados) pela técnica que são relevantes.

[4]Também chamada de revocação, é a fração de resultados relevantes que são recuperados.

Quando a abordagem recomenda um único especialista, o cálculo da precisão e da cobertura, é realizado considerando $k = 1$, *top-1*.

5.1.3 Resultados e Análise

Para uma entidade fazer parte de um sistema de software é preciso que ela tenha sido inserida no *VCS* do projeto em algum momento por um dado desenvolvedor. Assim, da perspectiva das abordagens baseadas no número de *commits* e em percentual de *LOC* modificadas, toda entidade tem sempre seu respectivo especialista, mesmo que ela tenha sido manipulada apenas uma vez: na sua criação pelo seu autor. No caso da nossa abordagem, nem sempre uma entidade terá um especialista associado. Dependendo do período de desenvolvimento observado, pode ser que nenhum vocabulário de desenvolvedor compartilhe termos com vocabulários de entidades para uma determinada versão de interesse.

Até a data do lançamento de uma versão de um projeto de software uma equipe de desenvolvimento pode sobre o projeto executar tarefas de codificação. Tomando o ePol como estudo de caso, definimos como versão de interesse sua *release* 0.5b, lançada em 18 de fevereiro de 2014, e geramos o vocabulário de cada um dos seus desenvolvedores considerando 4 diferentes períodos tempo: 2, 4, 6 e 9 meses anteriores ao dia em que a *release* 0.5b do ePol foi liberada para operação.

Dado que utilizamos uma estratégia aleatória para definir as entidades que compõem o Oráculo, Seção 5.1.2, não temos garantias de que elas tenham sido ou não manipuladas durante os 4 períodos de desenvolvimento que avaliamos. Contudo, para as duas abordagens de *baseline*, esse período de tempo é indiferente.

Computamos a acurácia das abordagens considerando não apenas um único especialista por entidade (*top-1*) onde apenas a medida de precisão é suficiente. Já para situações em que se necessita que mais de um especialista (*top-2*) e (*top-3*) sejam recuperados foram medidas tanto a precisão quanto a cobertura. Essa avaliação é necessária nas situações em que por indisponibilidade do maior especialistas para realizar uma tarefa de manutenção, o seu sucessor é que deve ser indicado.

Observando os resultados alcançados pela abordagem baseada em vocabulário considerando um único especialista, *top-1*, apresentado na Tabela 5.2, identificamos que à medida que mais tempo de contribuições são utilizadas para constituir os vocabulários de desenvolvedores, melhor é a precisão da nossa abordagem. Quando os vocabulários de desenvolvedores são extraídos para os períodos de 2 e 4 meses de contribuições, a precisão é de 0.25. Para 6 meses de vocabulários de desenvolvedores a precisão passa para 0.2857, representando 14.28% de aumento. E quando considerados os 9 meses de contribuições a precisão atinge 0.3214, percentualmente uma melhoria de 28.56% em relação a precisão obtida com 2 e 4 meses de vocabulários. No nosso ponto de vista, esse incremento concretiza a ideia do grau de eloquência *DOE* em que quanto mais os desenvolvedores manipulam os vocabulários das entidades mais termos eles compartilham com entidades específicas de código.

Tabela 5.2: Comparação entre abordagens para os *Top-1* especialistas.

Acurácia para *Top-1*	*Commit*	*%LOC*	Vocabulário			
			2 meses	4 meses	6 meses	9 meses
Precisão	0.3571	0.3571	0.2500	0.2500	0.2857	0.3214

No nosso entendimento, esse aumento de precisão em função do tamanho do vocabulário dos desenvolvedores, já é suficiente para atestar que vocabulários são capazes de modelar conhecimento. E assim responder positivamente nossa primeira questão de pesquisa QP_1: **Nossa abordagem consegue identificar especialistas de código?**. Além disso, é importante ressaltar que utilizando um modelo de conhecimento definido apenas em função de uma única medida entre vocabulários, a similaridade, nossa abordagem atingiu 0.3214 de precisão máxima para a identificação de um único especialista. No cenário em que um único especialista, *top-1*, é indicado, a diferença entre a precisão das abordagens de *baseline* e da baseada em vocabulário é de no máximo 0.1071 e no mínimo 0.0357 pontos percentuais, vide Tabela 5.2, em pro as de *baseline*. Esse valores representam que a nossa abordagem atinge uma precisão menor, entre 29.9% e 10%, que as de *baseline*.

Quando 2 especialistas, *top-2*, devem ser indicados, as abordagens de *baseline* continuam melhores, tanto em precisão quanto em cobertura, que a abordagem baseada em vocabulário, vide Tabela 5.3. Contudo, observa-se uma redução da diferença entre a acurácia da aborda-

gem por número de *commits* e da baseada em vocabulário, em relação ao cenário em que um único especialista, top-1, é indicado. No máximo a diferença em termos de precisão pro número de *commits* é de no máximo 0.0536 e no mínimo de 0.0179 pontos percentuais, já em relação a cobertura a diferença pro número de *commits* é de no máximo 0.075 e de no mínimo 0.025. Comparadas as abordagens por percentual de *LOC* e a baseada em vocabulário, para a precisão a diferença máxima é de 0.714 e a mínima é 0.0357. Já em termos de cobertura, a diferença pro números de *commit* é de no máximo 0.10 e no mínimo de 0.05 ponto percentual.

Tabela 5.3: Comparação entre abordagens para os *Top-2* especialistas.

Acurácia para *Top-2*	*Commit*	*%LOC*	Vocabulário			
			2 meses	4 meses	6 meses	9 meses
Precisão	0.2143	0.2321	0.1964	0.1607	0.1964	0.1964
Cobertura	0.3000	0.3250	0.2750	0.2250	0.2750	0.2750

Nos cenários em que até 3 especialistas, *top-3*, são indicados pelas abordagens, a Tabela 5.4 apresenta que quando o período considerado para gerar o vocabulário dos desenvolvedores aumenta, a acurácia da abordagem também aumenta, sobressai os valores alcançados pelas duas abordagens de *baseline*, exceto para o período de 4 meses de vocabulários de desenvolvedores.

Tabela 5.4: Comparação entre abordagens para os *Top-3* especialistas.

Acurácia para *Top-3*	*Commit*	*%LOC*	Vocabulário			
			2 meses	4 meses	6 meses	9 meses
Precisão	0.1548	0.1667	0.1905	0.1548	0.1786	0.1667
Cobertura	0.2826	0.3043	0.3478	0.2826	0.3261	0.3343

A diferença da precisão entre a abordagem baseada em vocabulário e baseada em *commits* é no máximo de 0.0357 e no mínimo 0 (zero), já em termos de cobertura a diferença pro abordagem baseada em vocabulário é de no máximo 0.0652 e de no mínimo 0 (zero). Esse valores representam um ganho máximo de 18.7% de precisão e de cobertura da nossa abordagem em relação a baseada em *commits*. A diferença de precisão entre abordagem baseada em vocabulário e baseada em percentual de *LOC* modificadas é no máximo é de 0.0238 e mínima de -0.0119 (pro abordagem percentual de *LOC* modificadas), já em termos de co-

bertura a diferença pro abordagem baseada em vocabulário é de no máximo 0.0435 e de no mínimo -0.0217 (pro abordagem percentual de *LOC* modificadas). Tais valores, representam um ganho máximo de 12.5% de precisão e de cobertura da nossa abordagem em relação a baseada percentual de *LOC* modificadas.

Ainda considerando cenários de indicação de mais de um especialistas, *top-2* e *top-3*, uma observação interessante é que enquanto a cobertura das abordagens de *baseline* diminuem a medida que aumenta o número de especialistas considerados, a cobertura da nossa abordagem aumenta. Isso reflete que para os casos onde outros desenvolvedores devem ser acionados, por indisponibilidade dos especialistas principal, a nossa abordagem consegue melhores recomendações.

Diante da análise realizada, a resposta a nossa questão de pesquisa QP_2: **Comparadas com as 2 abordagens de *baseline*, a precisão e a cobertura da nossa abordagem são melhores?** é que, para indicar um único especialista, as abordagens de *baseline* oferecem entre 10% e 29.9% a mais de precisão. Já para indicar mais de um desenvolvedor especialista a nossa abordagem atinge até 18.7% a mais de precisão e de cobertura em relação a abordagem baseada em número de *commits*, e até 12.5% a mais de precisão e cobertura em relação a abordagem baseada no percentual de *LOC* modificadas.

5.1.4 Conclusões

Acreditamos que os resultados poderiam ser melhores caso o estudo de caso pudesse ter sido realizado sobre um projeto de software codificado em inglês. Nesse caso, teria sido utilizado o algoritmo Porter [Porter, 1980] cuja taxa de acerto para extração de radicais na língua inglesa é acima de 90% [Flores et al., 2010]. Para extração em português, entre os algoritmos PTStemmer, Savoy [Savoy, 2006] e Orengo [Orengo and Huyck, 2001], utilizamos o último de acordo com nossas investigações, sobre uma base de identificadores extraídos de versões do sistema ePol, obteve a maior taxa de acerto para extração, 67% [Quintans et al., 2010].

Uma parte significante do vocabulário é formada tanto por acrônimos quanto pela concatenação de termos que não conseguem ser identificados pelos mecanismos de tokeniza-

ção baseados em convenções (*e.g*: *camelcase*). Logo a utilização de algoritmos que fazem uso de dicionários para realizar normalização de termos que em alguns casos implica na expansão ou até mesmo troca dos termos originais[Lawrie and Binkley, 2011; Guerrouj, 2013] poderia aprimorar os resultados.

As limitações tanto do algoritmo de extração de radicais quanto o de tokenização são fragilidades de construto conhecidas [Flores et al., 2010] e que pretendemos mitigar em trabalhos futuros.

A despeito dos resultados, vocabulários carregam informações de conhecimento diferente das capturadas pelas abordagens que usamos como *baseline*. Nesse caso, um caminho promissor para avançar nas técnicas de identificação de especialistas é mesclar num único modelo de conhecimento dados de autoria e de propriedade com dados extraídos de vocabulário. É nessa direção que planejamos e executamos o estudo detalhado na próxima Seção 5.2.

Limitações da Abordagem

Da mesma forma que o conhecimento utilizado pelas abordagens existentes para identificar especialistas tem limitações que podem resultar em indicações errôneas, como o exemplo apresentado na Seção 4.1, também na nossa abordagem identificamos algumas.

Uma das limitações se dá quando do uso de recursos das ferramentas de desenvolvimento que modificam automaticamente o vocabulário das entidades, tais como:

- Criação de *gets* e *sets* para os atributos das entidades;
- Geração de cabeçalhos, em *javadocs*, para entidades e seus métodos;
- Renomeção dos identificadores de atributos ou de variáveis locais. Se atributo, sua mudança repercutirá em todos métodos da entidade, e se variável local em todo o método que o engloba.

Esses cenários, acima listados, também não estão corretamente mapeados no conhecimento pelas abordagens tradicionais, como as duas que nesta Tese utilizamos como baseline.

Dependendo da janela ou período de tempo analisado de vocabulários de desenvolvedores, pode ser que para uma dada entidade nossa abordagem não identifique especialistas entre os desenvolvedores correntes do projeto. Essa seria a situação em que o vocabulário da entidade não alcançou um similaridade mínima, no nosso caso 0.2 de acordo com padrões da literatura [Kuhn et al., 2007; Santos et al., 2013]. Por exemplo, a entidade foi manipulada por desenvolvedores antes do início da janela de tempo considerada e não mais faziam parte da equipe antes do início da janela. Para as abordagens de *baseline*, o conhecimento para qualquer entidade apontará um especialista, mesmo ele não mais fazendo parte da equipe, já que em algum momento ela foi inserida no *VCS*.

5.1.5 Ameaças à Validade

Interna

A priori não identificamos situações em que os tratamentos aplicados ao fator Tipo de Abordagem sofram influências das variáveis dependentes precisão e cobertura.

Externa

A validade externa desse estudo é limitada pelas seguintes restrições:

- Amostra de entidades de código extraída de um único sistema, o ePol, codificado na sua maior parte em Java;
- A granularidade de entidades, enquanto uma unidade de *design*, está restrita ao nível de classes e interfaces;
- Vocabulário das entidades está limitado aos termos presentes nos identificadores de classes, interfaces, atributos, métodos, parâmetros e de variáveis locais; e o conteúdo dos *javadocs* e dos comentários de linha e de bloco. São desconsiderados *strings* literais e invocação de métodos, por exemplo;

- Os desenvolvedores participantes apesar de terem experiência em programação Java por mais de 3 anos, são estudantes de graduação em Ciência da Computação. Apesar de termos explicitamente informado os motivos científico-experimentais da investigação, as necessidades econômicas dos participantes podem os ter induzido: ou a serem presunçosos para garantirem sua permanência no projeto, ou modestos para minimizar suas demandas de atividades.

É evidente a impossibilidade da extrapolação dos resultados para sistemas desenvolvidos no mundo Java comercial e, consequentemente na generalização dos resultados. Processos de desenvolvimento, regras e estilos de programação de empresas provavelmente interferem ou induzem a forma como os termos são formados. Contudo, em caráter especifico, os resultados apontarão a precisão e a cobertura do uso de similaridade entre vocabulários de entidades e de desenvolvedores para identificação de especialistas sobre as entidades do código do ePol.

De Construto

Quanto a validade de construção para nossa abordagem, destacamos:

- As funções que manipulam vocabulários são implementadas considerando que os identificadores contém cadeias de caracteres concatenadas através de notação *camelcase*, *underscore*, ou ambas;

- Os termos que compõem um vocabulário são resultantes de uma radicalização de palavras através do algoritmo Orengo para código escrito em português cuja taxa de acerto é baixa quando comparado com o Porter utilizado para o inglês;

- A construção dos vocabulários dos desenvolvedores baseia-se na contribuição dada ao vocabulário do código fonte por cada um dos desenvolvedores. A identificação correta dessa contribuição depende do uso de um identificador, *Id*, único por desenvolvedor no ato de rotular os *commits*, concretização de uma mudança no repositório. Por exemplo, se um membro da equipe possuir mais de um identificador, a acurácia dos resultados

estão comprometidos. Para mitigar essa ameaça pode-se, antes de iniciar o estudo, extrair todos os *Id* de desenvolvedores e submetê-los a equipe de desenvolvimento. Identificadas inconsistências, uma função de mapeamento de 1 para N (de um *Id* único para N possíveis) deve ser aplicada;

Com relação as ameaças de construção para as 2 abordagens que servirão de *baseline*, destacamos:

- Implementação incorreta no ferramental *AuthorshipExtractor*. Para minimizar foram realizados testes manuais para averiguar a corretude do ferramental;
- Limitações inerentes dos algoritmos e heurísticas utilizadas em cada uma das 2 abordagens.

Relacionado a todos os tratamentos aplicados ao fator Tipo de Abordagem está a construção do Oráculo e a sua fidedignidade à realidade dos especialistas do projeto ePol. A presença do gerente do projeto durante o *survey* sobre as entidades, bem como a gravação de todo o áudio da reunião para eventuais auditorias, são aliados à veracidade do Oráculo. Contudo, não podemos garantir a ausência do efeito manada (groupthink) [Levesque et al., 2001], onde relatos ou ações iniciais quer sejam positivos ou negativos, impulsionam no mesmo sentido os relatos ou ações seguintes.

5.2 Contribuição dos Vocabulários para Conhecimento de Código

Neste segundo experimento temos como meta mensurar, em termos percentuais, quanto os vocabulários representam do conhecimento sobre um código fonte, e mais: se eles revelam aspectos do conhecimento diferentes dos capturados pelas abordagens de identificação de especialistas baseadas nas informações de autoria e propriedade.

5.2.1 Questões de Pesquisa

As questões de pesquisa (QP) que buscamos responder através deste experimento são decorrentes das respostas obtidas para QP_1 e QP_2, Seção 5.1.1. São elas:

QP_3: **A similaridade entre vocabulários de entidades e de desenvolvedores carrega aspectos inerentes para identificar especialistas não capturados pelos elementos de autoria do *DOA*: *FA (First Authorship)*, *DL (DeLiveries)* e *AC (ACceptances)*?**

Para responder essa questão buscaremos identificar um ou mais modelos para *DOA* acrescido da componente *DOE* que sejam estatisticamente válidos.

QP_4: **Qual o percentual da contribuição do modelo de conhecimento definido pela similaridade de vocabulários na identificação de especialistas?**

Nesse caso, identificando que o modelo para *DOA* acrescido da componente *DOE* fornece estatísticas melhores que o *DOA* desconsiderando *DOE*, podemos concluir que vocabulário tem influência no conhecimento, e explica capacidade do modelo de conhecimento definido por *DOE* (diferenças entre os R^2 dos modelos) de identificar especialistas.

5.2.2 Metodologia

A estratégia adotada para alcançarmos as metas do experimento foca em averiguar se o modelo de conhecimento definido pela similaridade entre vocabulários de entidades e de desenvolvedores quando adicionados a modelos baseado em autoria de código aumentam ou não a capacidade de identificar especialistas.

Como ilustrado na Figura 5.1 esse estudo foi divido nas seguintes etapas: 1) Seleção Amostral; 2) Coleta de Medidas de Autoria e de Similaridade; 3) Geração de modelos para *DOA*, e para *DOA* acrescido de *DOE*.

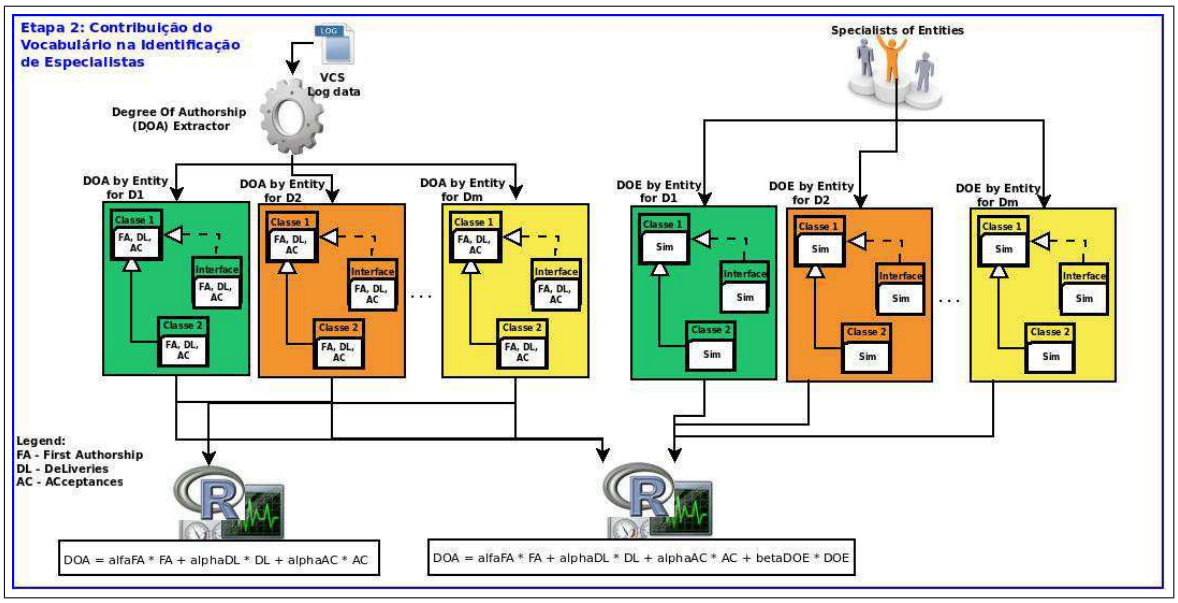


Figura 5.1: Etapas para aferir a contribuição de similaridade.

Seleção Amostral

A amostra é constituída das entidades que compõem o Oráculo de especialistas por entidade construído para estudo de caso sobre o projeto ePol e que por sua vez foram escolhidos por atenderem aos critérios já estabelecidos na Seção 5.1.2.

Coleta de Medidas de Autoria e de Similaridade

1. **Coleta de Medidas de Autoria**. Para cada tupla Entidade-Desenvolvedor, pertencente ao projeto ePol mas antecedente a sua versão 0.5b, extraímos os eventos de criação (*FA*) e de mudança (*DL*) de uma entidade, realizadas pelo desenvolvedor criador; e os eventos de mudanças (*AC*) ocorridos sobre a mesma entidade, mas realizados por outros desenvolvedores; Os eventos são extraídos dos *commits* registrados nos *logs* do *VCS* do projeto ePol, e são quantificados para todos os desenvolvedores que contribuíram com o projeto. Os desenvolvedores adquirem ou perdem conhecimento de autoria de acordo as seguintes regras por tipo de evento:

    - ***First Authorship - FA***. Desenvolvedores criadores de entidades adquirem o valor

de conhecimento 1.0 (um) ponto;

- ***DeLiveries - DL***. Para as modificações sobre entidades realizados pelos desenvolvedores autores, eles adquirem 0.5 (meio) ponto;
- ***ACceptances - AC***. Para as modificações sobre entidades realizadas pelos demais desenvolvedores, os não autores, os desenvolvedores autores perdem 0.1 (um décimo) de ponto do conhecimento.

O resultado é uma *M*atriz tridimensional de *E*ntidades por *D*esenvolvedores por Elementos do *DOA*, *MEDDOA*, vide Figura 5.2. O Valor de *DOA* para cada tupla Entidade-Desenvolvedor é dada por: $DOA(E_i, D_j) = MEDDOA[E_i, D_j, FA] + MEDDOA[E_i, D_j, DL] + MEDDOA[E_i, D_j, AC]$.

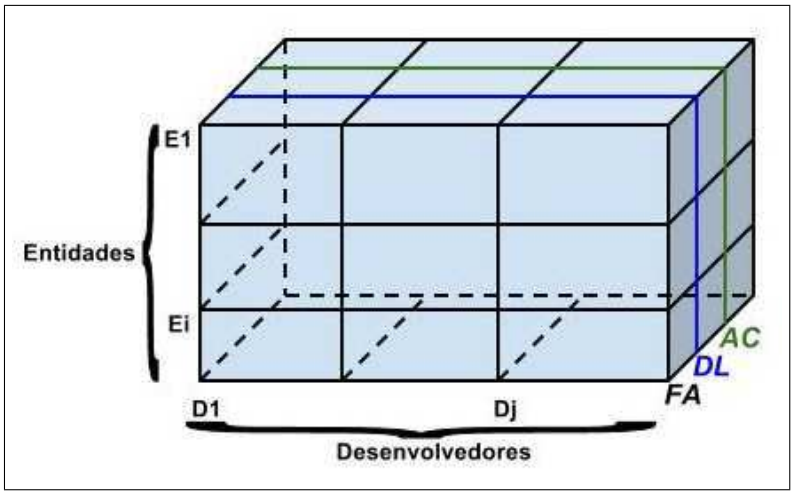


Figura 5.2: *MEDDOA*: Matriz Entidades por Desenvolvedores por Elementos de *DOA*.

2. **Coleta de Medidas de Similaridade**. Para cada tupla <Entidade-Desenvolvedor>, do projeto ePol, versão 0.5b, extraímos o vocabulário dos desenvolvedores e o vocabulário das entidades de código. O valor de *DOE* é a similaridade do cosseno entre vocabulário de uma dada entidade com o vocabulário de um dado desenvolvedor. É a ferramenta que calcula a similaridade entre vocabulários que geramos uma matriz de similaridade Entidade-Desenvolvedor, como exemplificada na Tabela 4.2 da Seção 4.5, que conterá todos os possíveis valores de *DOE*.

Geração do modelo para *DOA* apenas com seus componentes. A lista dos especialistas para as entidades que constituem o Oráculo do ePol foram definidos em função do consenso entre os membros da equipe, e não por valores auto-definidos pelos desenvolvedores como no modelo *DOK* original. Por este motivo, o valor de *DOA* extraído para cada par <Entidade, Desenvolvedor> do Oráculo do ePol, é definido em função da posição p ocupada pelo desenvolvedor D no lista de especialistas da cada *j*-ésima entidade E, através da seguinte expressão:

$$DOA_{E_j}^{D_p} = 100 - (p-1) * vde \tag{5.3}$$

onde, $p = \{1, 2, \ldots, ne\}$, dado que valor de ne indica o número total de especialistas indicados pelo Oráculo para a entidade E_j. Já vde é o valor de decremento de *DOA* entre desenvolvedores adjacentes na lista de especialistas de uma dada entidade E_j.

No projeto ePol, até a release 0.5b, apesar do Oráculo ter sido gerado com base no consenso dos 12 desenvolvedores que naquele momento formavam a equipe de desenvolvimento, os *logs* do *VCS* onde o código do ePol é gerenciado listava mais 23 diferentes desenvolvedores que já codificara alguma parte do projeto. Dessa forma, entre correntes e antigos, 35 desenvolvedores poderiam fazer parte das listas de especialistas para cada uma das entidades presentes no Oráculo. Para normalizar o valor de conhecimento de autoria*DOA* de uma tupla <Entidade, Desenvolvedor> qualquer do Oráculo, consideramos que seu valor máximo é 100, logo esse será o valor de *DOA* para o desenvolvedor ocupante da primeira posição na lista de especialistas, $p = 1$. A partir do segundo desenvolvedor da lista os valores de *DOA* sofrem acumulativamente um valor de decréscimo vde, dada por:

$$vde = DOA\ máximo = 100/ne \tag{5.4}$$

Para melhor compreensão dos valores atribuídos a *DOA*, imaginemos uma Entidade E que de acordo com o Oráculo tenha 4 desenvolvedores, $ne = 4$, como seus especialistas. O vde é dado por: $vde = 100/4 = 25$.

Aplicando a expressão 5.3, o valor de *DOA* para cada um dos especialistas é: $DOA_E^{D_1} = 100 - (1-1) * 25 = 100$

$DOA_E^{D_2} = 100 - (2-1) * 25 = 75$

$DOA_E^{D_3} = 100 - (3-1) * 25 = 50$

$DOA_E^{D_4} = 100 - (4-1) * 25 = 25$

Realizamos uma análise exploratória, baseada na técnica de regressão *Stepwise* [Boehm et al., 2005; Shmueli, 2010], com o objetivo de definir um modelo que estatisticamente explique os valores de *DOA* obtidos do Oráculo, em função dos seus componentes *FA*, *DL* e *AC* cujos valores foram coletados do repositório *VCS* do projeto como descrito.

Geração do modelo para *DOA* acrescido de *DOE*. Para cada par <Entidade, Desenvolvedor> agregamos seus valores de *DOE*, aos seus respectivos valores *FA*, *DL* e *AC*. Assim, definimos *DOA* como também dependente de DOE, como apresentado na Eq. 4.2.

Em seguida, assim como fizemos para *DOA* apenas em função *FA*, *DL* e *AC*, novamente aplicamos a técnica de regressão *Stepwise*. Nesse caso, o objetivo foi encontrar um modelo que estatisticamente explique os de *DOA* obtidos do Oráculo, para um dado par <Entidade, Desenvolvedor> não só em função dos seus componentes *FA*, *DL* e *AC* coletados do repositório, mas também em função da similaridade existente entre os vocabulários da entidade e do desenvolvedor, *i.e.* o *DOE*.

5.2.3 Resultados e Análise

Para esse experimento não há valores de medições resultante à submissão de diferentes tratamentos, mas sim modelos de regressão ajustados. De acordo com a técnica de refinamento regressão *Stepwise* [Nisbet et al., 2009; Shmueli, 2010], as seguintes estatísticas devem ser utilizadas para comparar os modelos: Coeficiente de Determinação[5] - R^2, o grau de significância, *p-value* do modelo e o *p-value* de cada um dos coeficientes do fatores, normalidade e dispersão dos erros residuais.

[5]O coeficiente de determinação, R^2, de um modelo revela de forma sumarizada o percentual da variabilidade dos dados que o modelo consegue explicar em função das variáveis observadas (as dependentes) [Jain, 1991]. De forma complementar, $(1 - R^2)$, revela o percentual da variabilidade dos dados não explicados pelo modelo.

A matriz de correlação de *Pearson*, descrita na Tabela 5.5, revela que entre *DOA* obtido do Oráculo e seus elementos coletados do repositório, *AC* é o único que apresenta uma forte correlação, 0.9288. Já os elementos *FA* e *DL*, tem fraca correlação, respectivamente 0.2132 e 0.1493.

Tabela 5.5: Matriz de Correlação de *Pearson* para valores de *DOA* e seus elementos de grau de autoria.

	FA	*DL*	*AC*	*DOA*
FA	1.0000000	-0.5252257	0.1980295	0.2132007
DL	-0.5252257	1.0000000	0.1690166	0.1493047
AC	0.1980295	0.1690166	1.0000000	0.9288407
DOA	0.2132007	0.1493047	0.9288407	1.0000000

Esses valores de correlação são indícios de que apenas as modificações realizadas pelos autores de entidades, *AC*, são suficientes para definir o conhecimento de autoria *DOA* obtido do Oráculo.

Contudo, é apenas com análise exploratória sobre modelos de regressão para o conhecimento de autoria *DOA* contido no Oráculo que conseguimos mensurar quanto cada tipo de evento de autoria representa do conhecimento de código. A Tabela 5.6 a seguir, apresenta um resumo de características estatísticas de modelos multilineares ajustados ao *DOA* obtido do Oráculo, em função dos eventos coletados do repositório do projeto. Para cada modelo, é apresentada a fórmula de regressão, seu Coeficiente de Determinação - R^2, e o valor do *p-value* do modelo (deve ser menor que 0.05 para ser válido).

Tabela 5.6: Estatísticas de Modelos Preditivos de *DOA* em função de seus elementos: *FA*, *DL* e *AC*.

Índice	Regressão	R^2	*p-value*
1	$DOA \sim FA$	0.0454545454545455	0.464272993911742
2	$DOA \sim DL$	0.0222918843608499	0.610448519397615
3	$DOA \sim AC$	0.862745098039216	1.60589513772778e-06
4	$DOA \sim FA + DL$	0.13973063973064	0.437008116700285
5	$DOA \sim FA + AC$	0.863636363636364	1.74117393178119e-05
6	$DOA \sim DL + AC$	0.862805890820507	1.80030070908373e-05
7	$DOA \sim FA + DL + AC$	0.863771043771044	0.000119577301073815

Os três primeiros modelos, índices 1, 2 e 3, ajustam o grau de autoria *DOA* em função de apenas um único elemento. Suas estatísticas evidenciam que, no projeto ePol, tanto o ato de criar entidades *FA*, como as modificações realizadas pelos criadores sobre as entidades *DL*,

pouco explicam quem detém o conhecimento sobre uma entidade. O modelo de *DOA* como apenas função linear de *FA* explica 4.54% da autoria, e em função de *DL* explica menos ainda, apenas 2.22%. Já as modificações realizadas sobre as entidades por outros desenvolvedores *AC*, diferentes de seus criadores, explicam 86.27% da identificação de especialistas através do *DOA*.

O modelo *DOA* em função de *FA* e de *DL*, índice 4, consegue explicar 13.97% dos dados, porém o *p-value* do modelo o invalida (*p-value* > 0.1). Já o pequeno incremento no poder de predição para os modelos descritos nas linhas 5, 6 e 7, respectivamente com R^2 de 86.36%, 86.28% e 86.37% mesmo tendo apenas o coeficiente de *AC* como significativo, podem ser considerados válidos já que seus *p-values* de modelo também são significativos.

De acordo com o maior valor de R^2 (86.37%) dos modelos válidos para *DOA*, na tabela 5.6, para pelo menos 13% dos dados de autoria não há nos modelos explicação. Também, mesmo que individualmente *FA* e *DL* pouco expliquem *DOA*, desprezá-los seria negar que a criação de entidades e suas modificações realizadas pelos seus criadores proporciona aquisição de conhecimento sobre essas entidades [Fritz et al., 2010]. Entre os estudados, o modelo 7 é o que melhor conglomera os três elementos que compõem o grau de autoria *DOA* aliado a evidências estatísticas de um modelo válido.

DOA ajustado de forma linear apenas em função de *DOE*, modelo 8 da tabela 5.7, explica 7.34% da autoria, percentual que mesmo não sendo alto, é maior quando comparado com aos valores de R^2 para *DOA* ajustado apenas em função de *FA*, 4.54%, ou apenas em função de DL, 2.22%. Mas, quando *DOE* é acrescido como mais uma variável preditora aos modelos de 1 ao 7, da tabela 5.6, todos os respectivos coeficientes de determinação R^2 tiveram um incremento em seus valores.

Os modelos 1, 2 e 4, da tabela 5.6, mesmos com seus respectivos *p-values* não significativos, com o acréscimo de *DOE* passaram a ser descritos respectivamente pelos modelos 8, 9 e 12, da tabela 5.7, cujos poderes de explicação do conhecimento de autoria, no mínimo, mais que dobraram. Do modelo 1 da tabela 5.6 para o modelo 9 da tabela 5.7, o R^2 subiu de 4.54% para 12.82%, do modelo 2 para o 10, o R^2 passou de 2.22% para 13.05%, e do

Tabela 5.7: Estatísticas de Modelos Preditivos de *DOA* em função de seus elementos e de *DOE*.

Índice	Regressão	R^2	*p-value*
8	$DOA \sim DOE$	0.073451654775299	0.531411408597936
9	$DOA \sim FA + DOE$	0.1282903649308298	0.601465878749191
10	$DOA \sim DL + DOE$	0.1305554745167992	0.593292927899763
11	$DOA \sim AC + DOE$	0.903394018703085	1.75826136984574e-05
12	$DOA \sim FA + DL + DOE$	0.36691285126329	0.24649079109983
13	$DOA \sim FA + AC + DOE$	0.904545535150184	0.00011625900018468
14	$DOA \sim DL + AC + DOE$	0.903406022904436	0.000121167089696869
15	$DOA \sim FA + DL + AC + DOE$	0.906136748478243	0.000575388051264534

modelo 4 para o 12, o R^2 saiu de 13.97% e atingiu 36.69%.

Já o incremento no poder de explicação sobre grau de autoria *DOA* que tiveram os modelos válidos da tabela 5.6, o 3, 5, 6 e 7, descritos na tabela 5.7 respectivamente pelos modelos 11, 13, 14 e 15, o aumento foi de pouco mais 4 pontos percentuais, em média. Do modelo 3 da tabela 5.6 para o modelo 11 da tabela 5.7, o R^2 subiu de 86.27% para 90.33%, do modelo 5 para o 13, o R^2 passou de 86.36% para 90.45%, do modelo 6 para o 14, o R^2 saiu de 86.28% e atingiu 90.34%, e do modelo 7 para o 15, o R^2 saiu de 86.37% e atingiu 90.61%.

5.2.4 Conclusões

Diante dos modelos de conhecimento válidos que combinam componentes de *DOA* e *DOE*, Tabela 5.7, podemos responder positivamente a questão de pesquisa QP_3: **A similaridade entre vocabulários de entidades e de desenvolvedores carrega aspectos inerentes para identificar especialistas não capturados pelos elementos de autoria do *DOA*: *FA (First Authorship)*, *DL (DeLiveries)* e *AC (ACceptances)*?**

Já a questão de pesquisa QP_4: **Qual o percentual da contribuição do modelo de conhecimento definido pela similaridade de vocabulários na identificação de especialistas?** foi respondida quando da comprovação do incremento de mais de 4 pontos percentuais na capacidade de identificar especialistas quando adicionados aos modelos de conhecimento o *DOE*.

É factível então combinar modelos de conhecimento baseados em vocabulário a modelos baseados em autoria e propriedade inclusive com os já consolidados na prática com o

objetivo de aprimorar as abordagens de identificação de especialistas. Pois, de acordo com nossos resultados pelo menos uma parcela dos conhecimentos modelos por vocabulário são diferentes dos capturados pelos modelos tradicionais de autoria.

5.2.5 Ameaças à Validade

As ameaças consideradas no estudo que avalia a acurácia da abordagem de identificação de especialistas de entidade utilizando vocabulário, Seção 5.1.5 também são válidas nesse experimento. Destacamos, adicionalmente como ameaça à validade do construto, a coleta e o processamento dos eventos de autoria: *FA*, *DL* e *AC*. Ressaltamos ainda que a generalização das nossas conclusões estão restritas ao nosso estudo de caso, e que as estatísticas utilizadas consideram valores significativos aqueles com $p\text{-}value < 0.05$.

Capítulo 6

Trabalhos Relacionados

Devido esta tese focar na investigação do uso do vocabulário de código fonte como insumo para identificar especialistas de sistemas, nossa revisão bibliográfica tem duas principais temáticas: 1) vocabulário de sistemas, e 2) especialistas de código.

Assim, neste capítulo, relatamos a literatura que nos auxiliou e nos instigou a desenvolver esta tese, bem como suas diferenças e suas semelhanças com trabalhos relacionados.

6.1 Sobre Vocabulário de Sistemas

Pesquisas vem apontando a importância do vocabulário em atividades de manutenção de sistemas [Binkley and Lawrie, 2009]. Estudos empíricos concluíram que a qualidade dos identificadores, quando eles não refletem os requisitos funcionais por exemplo, afeta na compreensão do software [Lawrie et al., 2006; Host and Ostvold, 2007].

Haiduc e Marcus [Haiduc and Marcus, 2008] analisaram 6 bibliotecas (4 em Java e 2 em C + +) do mesmo domínio de problema, e concluíram que os identificadores são escolhidos de acordo com as preferências pessoais e experiências dos desenvolvedores, bem como em função do jargões do domínio. Com base em várias versões de 2 sistemas de código aberto escritos em C++, Abebe e colaboradores [Abebe et al., 2009] mostraram que os identificadores estão relacionados com conceitos do domínio do problema. Butler e colegas, analisando 8 projetos de código aberto Java, identificaram que a compreensão e manutenção de software

são degradados quando identificadores não seguem boas regras da convenção *e.g.*: número excessivo de palavras, anomalias de capitalização [Butler et al., 2010].

Outro conjunto de pesquisas quantificou vocabulários de sistemas e sobre eles realizou análises estatísticas exploratórias com intuito de entender sua natureza e características.

Linstead e colegas [Linstead et al., 2009], por exemplo, revelaram uma distribuição estatística *power law* para as frequências dos termos de acordo com suas respectivas classes gramaticais (substantivos, verbos, adjetivos, etc). Este estudo também revelou que as palavras mais frequentes nas entidades do tipo classe são verbos, enquanto que os outras elementos são dominadas por substantivos. Diferente de quando compreendemos a natureza dos vocabulários através de modelos onde cada sistema tem seu próprio corpo de termos 3.2, os vocabulários dos 12.151 projetos *open source* foram agrupados num imenso e único corpo composto de identificadores de entidades, métodos e campos.

Anslow e Tempero [Anslow et al., 2008] extraíram palavras apenas dos identificadores de classe, da versão 1.1 à 1.6 da API Java, e também de 91 projetos *open source*. O objetivo foi acompanhar a evolução das frequências das palavras desse identificadores. Zhang [Zhang, 2009] conduziu um estudo empírico sobre os *tokens* léxicos (termos) de 24 sistemas *open source* do mundo real, mas que diferente do nosso estudo onde temos apenas sistemas codificadas em Java, ele incluiu sistemas em C++ e em C. Ele identificou a existência constante de distribuições do tipo *power law* nos programas de computador, tanto quando os termos são observados em trechos de sistemas como quando diferentes versões de um mesmo sistema são consideradas. A conservação de propriedades do vocabulário dos sistemas que identificamos quando exploramos, de forma isolada, o vocabulário de cada uma das entidades [Santos, 2012], corrobora com as estatísticas encontradas por Zhang.

Arnaoudova e colaboradores [Arnaoudova et al., 2010] também extraíram os termos a partir do código fonte, mas as frequências desses foram usadas para definir medidas de entropia e de cobertura de contexto para termos. Já Eshkevari [Eshkevari, 2010] mostrou que entidades de código que contém termos com altos valores dessas medidas estão mais propensas à ocorrência de falhas (*bugs*).

Medidas sobre o léxico de código fonte foram definidas por Biggers e colegas et al. [Biggers et al., 2011] com o objetivo de informar a melhor técnica de *Information Retrieval* que deve ser utilizado sobre um código fonte para uma atividade específica compreensão de software. Já o grupo de Haouari *et. al* [Haouari et al., 2011] analisou comentários de programas tanto de forma quantitativa quanto qualitativamente, e propôs uma taxonomia para os diversos tipos de comentários. Romano e colaboradores [Romano et al., 2011] propuseram uma abordagem que aproveita as informações léxicas do código fonte em conjunto com técnicas *fuzzy* de agrupamento para reduzir o número de padrões de projeto catalogados erroneamente por técnicas que utilizam informação estrutural.

Mais recentemente, uma corrente de estudiosos [Pollock et al., 2013]t em utilizado o termo linguagem natural do código fonte para lidar com o que chamamos de vocabulário de software. Butler e colegas [Butler et al., 2011] apontam que em 9% dos identificadores em código java, os seus termos não conseguem ser extraídos pelos algoritmos de tokenização baseados nos estilos de notação triviais (*e.g.*: *camelcase*, *underscore*), e propõe um algoritmo que leva em consideração a existência de dígitos e abreviações comumentes encontradas na linguagem natural do código fonte em análise. Na visão de Lawrie e Binkley o próximo passo para possibilitar um melhor tratamento da linguagem natural encontrada, é além de extrair *tokens* dos identificadores é expandi-los em palavras de dicionários que se assemelhem as palavras encontradas no vocabulário dos artefatos de software [Lawrie and Binkley, 2011]. A expansão de *tokens* é uma vertente seguida também por Guerrouj, mas adiciona ao uso de dicionários técnicas de reconhecimento de voz, por exemplo termos que tiveram uma vogal omitida podem ainda ser identificados [Guerrouj, 2013].

Esse caminho de extrair mais significado dos termos, deve contribuir com as pesquisas mais recentes que buscam utilizar vocabulário ou de parte dele no apoio à construção de técnicas que auxiliem na manutenção de sistemas [Binkley and Lawrie, 2009]. Inclusive vislumbramos que essa estratégia deve implementada na nossa abordagem de identificação de especialistas e verificada sua influência na acurácia. Rastkar [Rastkar et al., 2011] propôs uma abordagem de sumarização, a partir das informações estruturais e da linguagem natural

presente no código fonte, para descrever tanto de que se trata um dado *concern* analisado, como ele é implementado através de vários elementos ou níveis de abstração de um sistema. Ainda nesse caminho, Moreno [Moreno, 2014] faz uso de técnicas de linguagem natural que a partir de identificadores de variáveis e de instruções de código geram frases que sumarizam classes e conjuntos de mudanças. Sobre as classes, a sumarização, de acordo com resultados, auxiliam na compreensão do código abstraindo suas partes complexas. Já a sumarização de conjuntos de mudanças auxilia na listagem de mudanças ocorridas entre versões de um mesmo sistema.

Durante nossa imersões sobre uso de vocabulário, colaboramos com um estudo que investigou a nossa suspeita de que se desenvolvedores, ao lerem relatórios de defeitos (*bugs*), localizam no código os defeitos relatados, é porque deve existir algum nível de similaridade entre as os conteúdos das descrições dos defeitos e o vocabulário das entidade de código [Cavalcanti et al., 2012]. Os nossos resultados demonstraram que apesar de vocabulário não poder ser utilizado como fonte única de sugestão de classes impactadas, ele é, de fato, útil para diminuir o espaço de busca dos desenvolvedores. Mais recentemente, Saha e amigos [Saha et al., 2013], também buscaram localizar *bugs* usando vocabulários de código, mas incorporaram ao modelo de *IR* a ponderação das ocorrências de identificadores de classe, de métodos e de variáveis, e comentários, em dois campos dos relatórios de *bugs*: descrição e sumário.

A investigação sobre vocabulário de código quer seja do ponto de vista quantitativo ou do qualitativo, segundo as pesquisas aqui relatadas, revelam aspectos tanto sobre o processo de desenvolvimento [Binkley and Lawrie, 2011] quanto do produto resultante, o sistema, diferentes das capturadas por medidas tradicionais. Contudo, no melhor do nosso conhecimento, nenhum deles tratou vocabulário de software como insumo para mapear conhecimento na busca da identificação de especialistas como o fizemos nesta Tese. Um trabalho que potencializa a aquisição de conhecimento sobre código utilizando vocabulário é o desenvolvido mais recentemente por Inozemtseva e colaboradores [Inozemtseva et al., 2014]. Eles usam os identificadores para integrar diferentes repositórios de consultas, tais como *e-mails*, lis-

tas requisições de mudanças, perguntas do *Stack Overflow*[1], para facilitar a construção do entendimento sobre elementos de código relevantes quando na execução de tarefas de desenvolvimento.

6.2 Sobre Especialistas de Código

Existe uma série de trabalhos que desenvolveram conceitos e técnicas para identificar quem entre os desenvolvedores são os mais adequados para manipular um dado trecho de código. Inclusive, são utilizadas geralmente mais de uma medida para modelar e capturar *expertise*. Essa combinação de medidas é um dos motivos para a existência de uma variada taxonomia, na literatura, para nomear *expertise*. Contudo, Minto e colegas [Minto and Murphy, 2007] realizaram um levantamento bibliográfico sobre a temática e classificaram em três tipos, as abordagens para recomendar e identificar especialistas de projetos de software: as baseadas em heurísticas, as baseadas em redes sociais e as baseadas em máquinas de aprendizado.

As mais disseminadas são as abordagens baseadas em heurísticas que são aplicadas sobre os dados coletados a partir do histórico do desenvolvimento. Uma das mais simples é a modelada pela ideia de que especialista é aquele desenvolvedor que foi o último a modificar um dado código. Foi a utilizada na abordagem *Expertise Recommender* proposta por [McDonald and Ackerman, 2000] que é uma arquitetura proposta para armazenar histórico de mudanças, e delas identificar especialistas para solucionar problemas similares.

Entre as heurísticas, a mais utilizada é a que define que a contribuição de um desenvolvedor sobre uma parte do código, é dada pelo total de *commits* por ele realizado sobre o referido código. Por exemplo, o *Expertise Browser* proposto por Mockus e Herbsleb [Mockus and Herbsleb, 2002], recomenda os especialistas com base nas unidades básicas, átomos, de experiências (*EA*) que são extraídas do repositório de versões e computadas para cada autor a cada revisão de arquivo. Um *EA* é uma mudança relevante num arquivo, num módulo ou numa funcionalidade, e a experiência de um desenvolvedor é dada pela quantidade de *EAs*

[1] É um site gratuito com perguntas e respostas destinadas a profissionais da programação. http://http://stackoverflow.com/

por ele realizadas.

A abordagem proposta por Kagdi e colegas [Kagdi et al., 2008] recomenda uma lista classificada de especialistas para um dado arquivo fonte. A modelagem do conhecimento de código considera além do número de *commits*, a atividade sobre o código que é o total de dias em que houve alguma contribuição, e a data mais recente de atividade. Com essa combinação de medidas pode-se caracterizar: especialistas em profundidade, onde ocorre a concentração de *commits* em poucos arquivos; especialistas em largura, onde a concentração dos *commits* se dá em muitos arquivos; e especialistas exclusivos, em que apenas um desenvolvedor é quem realiza os *commits*.

O modelo *Degree-Of-Knowledge - DOK*, introduzido por Fritz e colaboradores [Fritz et al., 2010], em sua componente de autoria, *Degree-of-Authorship - DOA*, faz uso do número de *commits*. Quando para uma dada entidade, os *commits* indicam a sua criação (*First Authorship - FA*) ou modificações (*DeLiveries - DL*) sobre ela, o *DOA* do desenvolvedor responsável aumenta. Já quando nos *commits* se identifica que outros desenvolvedores modificaram (*ACceptances - AC*) a dada entidade, é diminuído o *DOA* do desenvolvedor responsável. A avaliação realizada sobre *DOK* nos inspirou no planejamento do experimento, definido na Seção 5.2, para mensurarmos a influência do uso do vocabulário na identificação de especialistas.

Para considerar situações não capturadas pelo conhecimento de código baseado no total de *commits* por desenvolvedor, Girba e colegas [Girba et al., 2005], consideraram como heurística uma informação de granularidade mais fina, modelando especialistas em função do percentual de linhas que cada desenvolvedor tenha modificado em cada arquivo. Visualizam os resultados em um mapa de autoria através do qual caracterizam comportamentos padrões dos desenvolvedores na evolução de código.

Hattori e Lanza [Hattori and Lanza, 2009] implementaram e avaliaram um repositório inovador, *Syde*, que grava toda mudança, inclusive as edições sobre os arquivos de código, realizadas por qualquer desenvolvedor. Os *logs* do *Syde* podem ser minerados como os repositórios tradicionais (*e.g.*: SVN) e os desenvolvedores não precisam parar de codificar

para submeter seu código. Nessa abordagem, para se recomendar os especialistas de código considera-se não só a quantidade de linhas modificadas, mas também as modificações decorrentes das edições sobre o código realizadas pelos desenvolvedores.

O conhecimento do código fonte expresso pelo percentual de linhas modificadas foi também utilizado por Rahman e Devanbu [Rahman and Devanbu, 2011] para relacionar o nível de especialidade do desenvolvedor com a qualidade do software em termos de defeitos (*bugs*). Do repositório de versão são extraídos os *implicated codes*, trechos de código que depois de modificados corrigem um *bug* conhecido. De cada linha dos trechos, utilizando o comando *blame* do *git*, são obtidos autor, revisão, data da última mudança. Os resultados apontam que os *implicated codes* estão mais associados aos códigos manipulados por um único desenvolvedor, especialistas em profundidade, do que aos generalistas, especialistas em largura.

As abordagens baseadas em redes sociais recomendam especialistas em função das relações entre os desenvolvedores extraídas dos sistemas de desenvolvimento. Um exemplo é o *Netexpert* [Ramon Sangüesa Solé and Serra, 2001]. Ele replica o processo de construção de uma rede de conhecimento no nível de comunidade de desenvolvedores, onde as interligações ocorrem em função das reputações dos membros. A reputação de um membro aumentam a medida que seus conhecimentos sobre o código são solicitados e ratificados pela comunidade.

Bird e colaboradores [Bird et al., 2006] construíram uma rede social baseada nas mensagens enviadas e respondidas na lista de e-mails do *Postgres*[2]. Os resultados apontam que as medidas de centralidade tem valores significativamente maiores para desenvolvedores do que para não desenvolvedores. Inclusive os dois maiores contribuintes do código fonte, considerando o número de *commits*, são também os dois participantes mais centrais da rede social.

Já as abordagens baseadas em máquinas de aprendizado focam em definir os especialistas com base em técnicas de categorização de texto. Anvik e colegas [Anvik et al., 2006]

[2]Sistema Gerenciador de Banco de Dados Objeto-Relacional, de código aberto, popular. www.postgresql.org

usam um período das informações contidas no repositório de *bugs* de um projeto para aprender padrões que indiquem quem deve resolver que tipo de *bug*. Para delegar requisições de mudanças a desenvolvedores, Cavalcanti e outros [Cavalcanti et al., 2014] integraram regras básicas de *experts* e o modelo de *IR* com aprendizado supervisionado. As regras são usadas para que critérios e restrições inerentes das organizações sejam seguidos (*e.g*: em módulos críticos apenas os desenvolvedores x, y podem atuar), já o *IR* é usado para encontrar desenvolvedores que tenham solucionados requisições de mudanças similares no passado.

Minto e Murphy [Minto and Murphy, 2007] produziram a ferramenta *Emergent Expertise Locator (EEL)* cujo objetivo é ajudar desenvolvedores a encontrar pares que possam auxiliá-los a resolver um problema (*e.g.*: correção de um *bug*, inclusão de nova funcionalidade). A abordagem utilizada pela ferramenta baseia-se em minerar o histórico de arquivos em busca daqueles modificados conjuntamente e, identificando quem foram os responsáveis pelas mudanças.

ExpertiseNet é um modelo de grafo relacional e evolucionário proposto por Song e colaboradores [Song et al., 2005] para modelagem de conhecimento de artigos. Consiste em analisar e extrair tanto do texto quanto das citações dos artigos a informação relacional e evolutiva referente ao conhecimento de um autor. Representados os perfis dos autores como grafos, e minerando padrões sobre eles, identifica-se autores com o mesmo interesse ou aqueles com um mesmo padrão de evolução de conhecimento.

Considerando o desenvolvimento projetos de forma de distribuída, Morales-Ramirez e outros [Morales-Ramirez et al., 2014] extraíram de discussões por *e-mails* as informações de conteúdo (stakeholders, tópico, termos e suas relações) e as intenções (perguntar, responder, refinar resposta) dos especialistas sobre as discussões. Propuseram uma abordagem que modela essas informações numa rede Markov para extrair um ranque de especialistas sobre um tópico.

Um dos trabalhos que mais nos instigou a levantar as questões de pesquisa que trata esta tese, foi o realizado por Matter e colaboradores [Matter et al., 2009]. Nele, a similaridade entre os vocabulários dos desenvolvedores e o conteúdo dos textos dos *bugs reports* é uti-

lizada para automaticamente associar *bugs*, a serem corrigidos, a um dado desenvolvedor. Já a nossa abordagem, extraímos o vocabulário das entidades (classes e interfaces) de uma versão de interesse projeto, e através da sua similaridade com o vocabulário dos desenvolvedores apontamos o especialista para realizar uma mudança ou uma revisão de código, cuja localização já é conhecida.

Esses trabalhos estudados sobre conhecimento de código, ratificam o que foi na descrito na Seção 2.2.1: para ser revelado, o conhecimento é medido de forma indireta, pelos seus efeitos. O construto para representá-lo precisa capturar as suas principais causas que nem sempre são visíveis nem palpáveis, nem muito menos conhecidas. Contudo, propor e avaliar novos construtos para conhecimento é fundamental para aprimorarmos as técnicas de identificação de especialistas. E, localizar o desenvolvedor mais adequado, implica por exemplo, em minimizar o custo (de tempo e de esforço) de realizar manutenção em um sistema.

Capítulo 7

Considerações Finais

Nos parágrafos a seguir, relatamos as nossas contribuições no entendimento sobre vocabulário de software e sua utilização no avanço para melhoria de modelos de conhecimento de código. Também tecemos nossas conclusões sobre o uso da nossa abordagem e sobre os resultados por ela alcançados com o uso de vocabulários na identificação de especialistas de entidades Java de código. Em seguida, descrevemos possíveis trabalhos futuros como desdobramentos desta tese.

7.1 Contribuições e Conclusões

O levantamento bibliográfico sobre vocabulário e áreas correlatas, por nós realizado possibilitou identificar o estado-da-arte sobre a importância e a utilidade do vocabulário nas atividades de desenvolvimento e manutenção de sistemas.

Para compreender a origem da formação dos vocabulários e como se revelam as informações por eles carregadas, planejamos e executamos estudos estatísticos exploratórios que permitiram identificar e entender duas de suas propriedades: tamanho de vocabulário e a ocorrência de seus termos ao longo do código. Em particular, as evidências estatísticas indicaram que tanto o quantitativo de termos distintos, únicos, quanto o total de termos, são modelados como uma função de potência sobre o tamanho em *LOC* do sistema. Já as repetidas ocorrências de cada termo de um vocabulário são regidas por uma distribuição Cauda

Longa do tipo log-normal, onde tanto sua média, μ, como o seu desvio padrão, σ, também são expressos em função do *LOC* do sistema.

Utilizando o conceito *multisets* construímos uma formalização com intuito de caracterizar nossa concepção e definir sem ambiguidades operações sobre vocabulários, bem como para facilitar a expressividade e promover a reprodução dos nossos experimentos por outros pesquisadores [Santos et al., 2011; Santos et al., 2012].

Para realizar esses estudos empíricos, desenvolvemos o *Vocabulary Tool*, ferramental capaz de extrair, a partir de código fonte java, identificadores, comentários e *javadoc* (vide Apêndice A). Esse ferramental foi utilizado em outras investigações que fazem uso de vocabulário e com as quais colaboramos. Um primeiro estudo, indexou o vocabulário extraído do código fonte e sobre ele pesquisou o conteúdo de relatórios de defeitos (*bugs*), com o objetivo de localizar *bugs* no código. Parte dos resultados dessa investigação encontra-se em: *Using Software Vocabulary to Rank Classes that are Probably Impacted by a Bug Report*. *TAinSM-ICSM 2012* [Cavalcanti et al., 2012], Trento - Itália. Apêndice C.

Um outro estudo que utilizou o *Vocabulary Tool* e contou com a nossa colaboração foi: *TopicViewer : Evaluating Remodularizations Using Semantic Clustering*. *CBSoft2015: Teoria e Prática* [Santos et al., 2013], Brasília - Brasil. Apêndice D. Neste utilizamos técnicas de recuperação de informação e agrupamentos hierárquicos sobre entidades de código conceitualmente coesas para propor mudanças na arquitetura de sistemas.

A exploração sobre vocabulário, nos permitiu encontrar uma lacuna onde vocabulário, até a escrita desta Tese, ainda não tinha sido explorado: como modelo de conhecimento de código para identificar especialistas. Como contribuição nossa concebemos e implementamos, uma abordagem de identificação de especialistas que utiliza como modelo de conhecimento a similaridade entre o vocabulário presente nas entidades de código e o vocabulário manipulado por cada um daqueles que ao longo da evolução do projeto participou como integrante da equipe de desenvolvimento.

Extraímos de forma estratificada e aleatória uma amostra de entidades representativa para um projeto real de software, o ePol. Com o apoio de um ferramental de software, subme-

temos a amostra a um *survey* com os integrantes correntes da equipe de desenvolvimento gerando assim um Oráculo, base de referência de especialistas por entidade.

Avaliamos a abordagem proposta e a comparamos com duas outras técnicas de identificação de especialistas bem difundidas na literatura e utilizadas na prática: por *commits* [Mockus and Herbsleb, 2002] e por percentual de *LOC* modificadas [Girba et al., 2005]. A acurácia das abordagens foi medida em termos de precisão e de cobertura em relação ao Oráculo construído. Comprovamos que a abordagem proposta identifica especialistas de código, já que para a identificação de um único especialista a abordagem alcança uma precisão de até 0.3214, e num cenário para indicar até 3 especialistas precisão alcançada é de até 0.1905 enquanto que cobertura é de até 0.3478. Quando comparados considerando os mesmos cenários, no geral, os resultados comparativos concluem que para indicar um único especialista, *top-1*, as abordagens de *baseline* oferecem apenas cerca de 10% a mais de precisão. Já para indicar mais de um desenvolvedor especialista, até *top-3*, a nossa abordagem tem uma acurácia melhor de até 23% em relação as de *baseline*. Todos os resultados em detalhes podem ser consultados na Seção 5.1.3

Esse resultado foi aceito para publicação como artigo completo:

- *Using Developers Contributions on Software Vocabularies to Identify Expertst. 12th International Conference on Information Technology: New Generations - ITNG 2015*, Las Vegas - USA. Apêndice E.

Adicionamos ao modelo *Degree-Of-Knowledge - DOK* [Fritz et al., 2010] a componente de conhecimento utilizado pela nossa abordagem: similaridade entre vocabulários. Submetemos a amostra de entidades que compõem o oráculo de especialistas tanto ao modelo *Degree-Of-Knowledge - DOK* com suas componentes de autoria originais, quanto ao adicionado com a similaridade entre vocabulários, e encontramos seus respectivos modelos de regressão. Todos os modelos encontrados para *DOK* com suas componentes de autoria originais, obtiveram Coeficiente de Determinação, R^2, menores que os dos seus respectivos modelos de *DOK* acrescentados da similaridade entre vocabulários, revelando assim que a similaridade carrega aspectos de conhecimento de código diferente dos existes em autoria. Entre as dife-

renças destacamos a comparação entre os modelos válidos de $DOA \sim FA + DL + AC$ com $R^2 = 0.8637$ e ele mesmo acrescido de similaridade, o $DOA \sim FA + DL + AC + DOE$ com $R^2 = 0.9061$. Todos os modelos de regressão utilizados bem como suas estatísticas estão descritas na Seção 5.2.3. Explorados e comparados estatisticamente todos os modelos válidos, quantificamos que o conhecimento de código capturado de vocabulários incrementa em mais de 4 pontos percentuais a capacidade dos modelos em identificar especialistas.

De forma geral concluímos que além de poder ser utilizado de forma isolada para modelar de conhecimento de código e assim identificar especialistas, o vocabulário pode ser um componente adicional a modelos de conhecimento baseados em autoria e propriedade, já que capturam aspectos próprios e diferentes dos existentes nesses modelos.

7.2 Trabalhos Futuros

Toda a investigação descrita nesta Tese, bem como a identificação de ameças a sua validade e também as conclusões a que chegamos sobre uso de vocabulário para mapear conhecimento de código, nos remete a uma série de novas investigações. As que julgamos prioritárias com seus respectivos descritivos seguem:

- Reproduzir os estudos realizados nesta Tese para outros projetos reais, incluindo projetos que estejam codificados, comentados e documentados em língua inglesa. O grande desafio para realizar este estudo é o acesso ao código fonte e a toda equipe de desenvolvimento para geração de oráculos;

- Aferir qual a taxa de acerto da abordagem baseada em vocabulário sobre recomendação de desenvolvedores para adotarem classes órfãs.

- Investigar se a similaridade entre os vocabulários dos agrupamentos semânticos de classe e dos desenvolvedores pode ser utilizada para identificar especialistas de domínios de conceitos. A motivação é que em pelo menos 25% dos casos, as tarefas de codificação transpassam além dos limites do código de uma única entidade [Caval-

canti, 2012], estando atreladas às funcionalidades ou às subtarefas desempenhadas por um sistema, as *features* [Eisenbarth et al., 2003].

- Utilizar o oráculo construído para o sistema ePol em outras investigações. Por exemplo, entidades que no oráculo foram classificadas como órfãs pela equipe atual, poderiam ter seus vocabulários comparados com o de outras entidades para identificar outras possíveis órfãs. Outro exemplo, seria o uso do oráculo como *benchmark* para avaliar a acurácia de quaisquer outras abordagens desenvolvidas para identificação de especialistas;

- Aprimoramento nos algoritmos de extração de radicais de palavras escritas em Português. Os algoritmos por nós investigados foram inicialmente desenvolvidos para realizar extração sobre palavras de textos literais e científicos e não, sobre programas onde acentos, cedilhas, conjunções verbais, entre outras restrições não são consideradas. Acreditamos que adequações sobre esses algoritmos possam ser realizadas para melhorar suas taxa de acerto no contexto de desenvolvimento de programas;

- Associar medidas e propriedades de vocabulários a outras medidas tradicionais. O ferramental *Vocabulary Tool* fornece informações sobre o vocabulário de projetos, em fomato XML-like, que podem ser facilmente absorvidas em outras ferramentas. Considerando que os vocabulários carregam informações de processo e de produto diferentes das providas pelas medidas tradicionais, entendemos que estudos, na área de manutenção de sistemas por exemplo, que não fazem uso de vocabulários podem tomar outros rumos caso passem a utilizá-los;

- Combinar o conhecimento de código mapeado através de vocabulários de software, com o conhecimento utilizado por outras abordagens de identificação de especialistas: baseadas em autoria e propriedade, baseadas em mineração de outros repositórios como fóruns, lista de *e-mails* e artefatos de documentação, ou ainda alguma baseada em máquina de aprendizado;

- Investigar as diferentes abordagens de identificação de especialistas em função da taxa de concordância em especialistas (percentual das entidades de um software nas quais as abordagens comparadas indicam os mesmos especialistas) entre elas. Além de ser uma alternativa a medição da acurácia e consequente da criação de oráculos de especialistas, taxa de discordância (1 - taxa de concordância) poderá revelar novos cenários ainda não mapeados em conhecimento de código.

Bibliografia

[sou, 2009] (2009). SourceForge. http://sourceforge.net.

[Abebe et al., 2009] Abebe, S. L., Haiduc, S., Marcus, A., Tonella, P., and Antoniol, G. (2009). Analyzing the Evolution of the Source Code Vocabulary. *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, pages 189–198.

[Anslow et al., 2008] Anslow, C., Noble, J., Marshall, S., and Tempero, E. (2008). Visualizing the word structure of Java class names. In *Proc. OOPSLA (Companion)*, number 133, pages 777–778.

[Antoniol et al., 2007] Antoniol, G., Gueheneuc, Y.-G., Merlo, E., and Tonella, P. (2007). Mining the Lexicon Used by Programmers during Sofware Evolution. *2007 IEEE International Conference on Software Maintenance*, pages 14–23.

[Anvik et al., 2006] Anvik, J., Hiew, L., and Murphy, G. C. (2006). Who should fix this bug? *Proceeding of the 28th international conference on Software engineering - ICSE '06*, page 361.

[Apache Foundation, a] Apache Foundation. Apache Foundation. http://www.apache.org/.

[Apache Foundation, b] Apache Foundation. Lucene Project. http://lucene.apache.org/.

[Arnaoudova et al., 2010] Arnaoudova, V., Eshkevari, L., Oliveto, R., Guéhéneuc, Y., and Antoniol, G. (2010). Physical and conceptual identifier dispersion: Measures and relation to fault proneness. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–5. IEEE.

[Baeza-Yates and Ribeiro-Neto, 1999] Baeza-Yates, R. and Ribeiro-Neto, B. (1999). *Modern information retrieval*. ACM Press.

[Barbetta et al., 2008] Barbetta, P., Reis, M., and Bornia, A. (2008). *Estatística para Curso de Engenharia e Informática*. São Paulo - Atlas.

[Bennett and Rajlich, 2000] Bennett, K. and Rajlich, V. (2000). Software Maintenance and Evolution: A Roadmap. *ICSE '00 Proceedings of the Conference on The Future of Software Engineering*, pages 73 – 87.

[Biggers et al., 2011] Biggers, L., Eddy, B., Kraft, N., and Etzkorn, L. (2011). Toward a Metrics Suite for Source Code Lexicons. In *Conference On Software Maintenance*.

[Biggerstaff et al., 1993] Biggerstaff, T., Mitbander, B., and Webster, D. (1993). The concept assignment problem in program understanding. *Proceedings of 1993 15th International Conference on Software Engineering*, pages 482–498.

[Binkley, 2007] Binkley, D. (2007). Source Code Analysis: A Road Map. *Future of Software Engineering (FOSE '07)*, pages 104–119.

[Binkley et al., 2009] Binkley, D., Davis, M., Lawrie, D., and Morrell, C. (2009). To CamelCase or Under_score. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 158–167. IEEE.

[Binkley and Lawrie, 2009] Binkley, D. and Lawrie, D. (2009). Information Retrieval Applications in Software Maintenance and Evolution. *Encyclopedia of Software Engineering*, pages 1–29.

[Binkley and Lawrie, 2011] Binkley, D. and Lawrie, D. (2011). Information Retrieval Applications in Software Development.

[Bird et al., 2006] Bird, C., Gourley, A., and Swaminathan, A. (2006). Mining Email Social Networks in Postgres. *MSR '06 Proceedings of the 2006 International Workshop on Mining Software Repositories*, page 185.

[Bittencourt et al., 2010] Bittencourt, R. A., Santos, G. J. S., Guerrero, D., and Murphy, G. C. (2010). Improving Automated Mapping in Reflexion Models using Information Retrieval Techniques. *17th Working Conference on Reverse Engineering*.

[Blizard, 1988] Blizard, W. D. (1988). Multiset theory. *Notre Dame Journal of formal logic*, 30(1):36–66.

[Boehm and Basili, 2001] Boehm, B. and Basili, V. R. (2001). Software Defect Reduction Top 10 List. *Software Manangement Magazine*, pages 135–137.

[Boehm et al., 2005] Boehm, B., Rombach, H. D., and Zelkowitz, M. V. (2005). *Foundations of Empirical Software Engineering - The Legacy of Victor R. Basili*. Springer-Verlag Berlin Heidelberg, Berlin Heidelberg.

[Booch et al., 2000] Booch, G., Rumbaugh, J., and Jacobson, I. (2000). *UML - Guia do Usuário*. Editora Campus.

[Buse and Weimer, 2010] Buse, R. P. L. and Weimer, W. R. (2010). Learning a Metric for Code Readability. *IEEE Transactions on Software Engineering*, 36(4):546–558.

[Butler et al., 2010] Butler, S., Wermelinger, M., Yu, Y., and Sharp, H. (2010). Exploring the Influence of Identifier Names on Code Quality: an empirical study. *oro.open.ac.uk*.

[Butler et al., 2011] Butler, S., Wermelinger, M., Yu, Y., and Sharp, H. (2011). Improving the tokenisation of identifier names. *25th European Conference on Object-Oriented Programming*.

[Caprile and Tonella, 2000] Caprile, B. and Tonella, P. (2000). Restructuring Program Identifier Names. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 97–107. IEEE.

[Cavalcanti et al., 2012] Cavalcanti, D., Santos, K., Serey, D., and Figueiredo, J. (2012). Using Software Vocabulary to Rank Classes that are Probably Impacted by a Bug Re-

port. In *1st Workshop on the Next Five Years of Text Analysis in Software Maintenace - ICSM2012*, pages 0–4.

[Cavalcanti, 2012] Cavalcanti, D. T. (2012). Uso de Vocabulários para Analisar o Impacto de Relatórios de Defeitos a Código-Fonte. Technical report, UFCG - Universidade Federal de Campina Grande.

[Cavalcanti et al., 2014] Cavalcanti, Y. a. C., Machado, I. D. C., da Mota Silveira Neto, P. A., de Almeida, E. S., and de Lemos Meira, S. R. (2014). Combining Rule-based and Information Retrieval Techniques to assign Software Change Requests. In *Proceedings of The 29th IEEE/ACM International Conference on Automated Software Engineering (ASE'2014)*, pages 325–330, cavalcanti.2014.

[Clauset et al., 2009] Clauset, A., Shalizi, C., and Newman, M. (2009). Power-law distributions in empirical data. *SIAM*, 51(4):661–703.

[Corley et al., 2012] Corley, C., Kammer, E., and Kraft, N. (2012). Examining the Relationship between Ownership and Topics in Source Code. Technical report, The University of Alabama, Tuscaloosa, Alabama 35487-0290 USA.

[Couto et al., 2012] Couto, C., Silva, C., Valente, M. T., Bigonha, R., and Anquetil, N. (2012). Uncovering causal relationships between software metrics and bugs. In *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pages 223–232.

[Crowston and Howison, 2006] Crowston, K. and Howison, J. (2006). Core and Periphery in Free/Libre and Open Source Software Team Communications. *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06)*, 00(C):118a–118a.

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

[Dalgaard, 2008] Dalgaard, P. (2008). *Introductory Statistics with R*. Springer, second edition.

[Deissenboeck and Pizka, 2006] Deissenboeck, F. and Pizka, M. (2006). Concise and consistent naming. *Software Quality Journal*, 14(3):261–282.

[dos Santos et al., 2012] dos Santos, E. F. G., Guerrero, D. D. S., and Monteiro, J. a. A. B. (2012). Coleta de Dados sobre os Desenvolvedores e o Código do e-Pol. Technical report, Departamento de Sistemas e Compuatção - DSC/UFCG, Campina Grande.

[Eisenbarth et al., 2003] Eisenbarth, T., Koschke, R., and Simon, D. (2003). Locating Features in Source Code. *IEEE Transactions on Software Engineering*, pages 210–224.

[Enslen et al., 2009] Enslen, E., Hill, E., Pollock, L., and Vijay-Shanker, K. (2009). Mining Cource Code to Automatically Split Identifiers for Software Analysis. *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 71–80.

[Eshkevari, 2010] Eshkevari, L. M. (2010). Linguistic Driven Refactoring of Source Code Identifiers. *2010 17th Working Conference on Reverse Engineering*, pages 297–300.

[Everitt and Hothorn, 2010] Everitt, B. S. and Hothorn, I. (2010). *A Handbook of Statistical Analyses Using R*. Chapman & Hall/CRC.

[Feigenspan et al., 2011] Feigenspan, J., Apel, S., Liebig, J., and Kästner, C. (2011). Exploring Software Measures to Assess Program Comprehension. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*.

[Feitelson, 2009] Feitelson, D. G. (2009). *Workload Modeling for Computer Systems Performance Evaluation*. The Hebrew University of Jerusalem, Jerusalem, Israel, version 0. edition.

[Flores et al., 2010] Flores, F. N., Moreira, V. P., and Heuser, C. a. (2010). Assessing the impact of stemming accuracy on information retrieval. *Lecture Notes in Computer Sci-*

ence (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 6001 LNAI:11–20.

[Fluri et al., 2007] Fluri, B., Wursch, M., and Gall, H. C. (2007). Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes. *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 70–79.

[Fritz et al., 2010] Fritz, T., Ou, J., Murphy, G. C., and Murphy-Hill, E. (2010). A Degree-of-Knowledge Model to Capture Source Code Familiarity. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, page 385.

[Girba et al., 2005] Girba, T., Kuhn, A., Seeberger, M., and Ducasse, S. (2005). How Developers Drive Software Evolution. *Proceedings of the 2005 Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*.

[GNU,] GNU, F. S. F. R-Project. http://www.r-project.org.

[Guerrouj, 2010] Guerrouj, L. (2010). Automatic Derivation of Concepts Based on the Analysis of Source Code Identifiers. *2010 17th Working Conference on Reverse Engineering*, pages 301–304.

[Guerrouj, 2013] Guerrouj, L. (2013). Normalizing source code vocabulary to support program comprehension and software quality. *ICSE '13 Proceedings of the 2013 International Conference on Software Engineering*, pages 1385–1388.

[Haiduc and Marcus, 2008] Haiduc, S. and Marcus, A. (2008). On the Use of Domain Terms in Source Code. *The 16th IEEE International Conference on Program Comprehension*, 0:113–122.

[Hallam, 2006] Hallam, P. (2006). What Do Programmers Really Do Anyway? In *Microsoft Developer Network (MSDN). http://blogs.msdn.com/b/peterhal/archive/2006/01/04/509302.aspx*.

[Haouari et al., 2011] Haouari, D., Sahraoui, H., and Philippe, L. (2011). How Good is your Comment? A study of Comments in Java Programs. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*.

[Hattori and Lanza, 2009] Hattori, L. and Lanza, M. (2009). Mining the History of Synchronous Changes to Refine Code Ownership. *Mining Software Repositories, 2009. MSR'*, pages 141–150.

[Hjorth, 1989] Hjorth, U. (1989). On model selection in the computer age. *Journal of Statistical Planning and Inference*, 23(1):101–115.

[Host and Ostvold, 2007] Host, E. W. and Ostvold, B. M. (2007). The Programmer's Lexicon, Volume I: The Verbs. *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, I:193–202.

[Inozemtseva et al., 2014] Inozemtseva, L., Subramanian, S., and Holmes, R. (2014). Integrating Software Project Resources Using Source Code Identifiers Categories and Subject Descriptors. pages 400–403.

[Jain, 1991] Jain, R. (1991). *The Art of Computer Systems Performance Analysis: techniques for experimental design, measurement, simulation, and modeling*. Wiley New York.

[John M. et al., 1983] John M., C., William S., C., Kleiner, B., and Tukey, P. A. (1983). *Graphical Methods For Data Analysis*. Duxbury Press.

[Johnson et al., 1994] Johnson, N. L., Kotz, S., and Balakrishnan, N. (1994). *Continuous Univariate Distributions (volume 1)*. Wiley-InterScience, 2nd edition.

[Kagdi et al., 2008] Kagdi, H., Hammad, M., and Maletic, J. (2008). Who Can help Me with this Source Code Change? *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 157–166.

[Kagdi and Poshyvanyk, 2009] Kagdi, H. and Poshyvanyk, D. (2009). Who can help me

with this change request? In *2009 IEEE 17th International Conference on Program Comprehension*, volume 9, pages 273–277. IEEE, IEEE.

[Kitchenham et al., 2002] Kitchenham, B., Pfleeger, S., Pickard, L., Jones, P., Hoaglin, D., El-Emam, K., and Rosenberg, J. (2002). Preliminary Guidelines for Empirical Research in Software Engineering. *Engineering, IEEE*, (January).

[Kokol and Podgorelec, 1999] Kokol, P. and Podgorelec, V. (1999). Computer and Natural Language Texts - A Comparison Based on Long-Range Correlations. *Journal of the American Society for Information Science*, 50(December 1999):1295–1301.

[Kuhn et al., 2007] Kuhn, A., Ducasse, S., and Gîrba, T. (2007). Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243.

[Larsen, 2012] Larsen, P. V. (2012). Module 8: Selecting regression models. http://statmaster.sdu.dk/courses/st111/module08/index.html.

[Lawrie and Binkley, 2011] Lawrie, D. and Binkley, D. (2011). Expanding Identifiers to Normalize Source Code Vocabulary. *2011 27th IEEE International Conference on Software Maintenance (ICSM)*.

[Lawrie et al., 2006] Lawrie, D., Feild, H., and Binkley, D. (2006). Quantifying identifier quality: an analysis of trends. *Empirical Software Engineering*, 12(4):359–388.

[Levesque et al., 2001] Levesque, L. L., Wilson, J. M., and Wholey, D. R. (2001). Cognitive divergence and shared mental models in software development project teams. *Journal of Organizational Behavior*, 22(2):135–144.

[Linstead et al., 2009] Linstead, E., Hughes, L., Lopes, C., and Baldi, P. (2009). Exploring Java software vocabulary: A search and mining perspective. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 29–32. IEEE Computer Society.

[Matter et al., 2009] Matter, D., Kuhn, A., and Nierstrasz, O. (2009). Assigning bug reports using a vocabulary-based expertise model of developers. *6th IEEE International Working Conference on Mining Software Repositories*, pages 131–140.

[McCabe, 1976] McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320.

[McDonald and Ackerman, 2000] McDonald, D. W. and Ackerman, M. S. (2000). Expertise recommender: a flexible recommendation system and architecture. In *CSCW '00 Proceedings of the 2000 ACM conference on Computer supported cooperative work*, volume Philadelph of *CSCW '00*, pages 231–240, New York, New York, USA. ACM Press.

[Minto and Murphy, 2007] Minto, S. and Murphy, G. C. (2007). Recommending emergent teams. In *Proceedings - ICSE 2007 Workshops: Fourth International Workshop on Mining Software Repositories, MSR 2007*, number Section 5 in MSR '07, pages 5–5. Ieee.

[Mockus et al., 2002] Mockus, A., Fielding, R., and Herbsleb, J. (2002). Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(3):346.

[Mockus and Herbsleb, 2002] Mockus, A. and Herbsleb, J. (2002). Expertise Browser: a quantitative approach to identifying expertise. *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 503–512.

[Morales-Ramirez et al., 2014] Morales-Ramirez, I., Vergne, M., Morandini, M., Siena, A., Perini, A., and Susi, A. (2014). Who is the expert? combining intention and knowledge of online discussants in collaborative RE tasks. *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*, pages 452–455.

[Moreno, 2014] Moreno, L. (2014). Summarization of complex software artifacts. *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*, pages 654–657.

[Nau,] Nau, R. F. Forecasting Course.

[Nisbet et al., 2009] Nisbet, R., Elder, J., and Miner, G. (2009). *Handbook of Statistical Analysis and Data Mining Applications*. Elsevier Ltd.

[Orengo and Huyck, 2001] Orengo, V. and Huyck, C. (2001). A stemming algorithm for the Portuguese language. In *Proceedings of the 8th International Symposium on String Processing and Information Retrieval (SPIRE*, volume 2001, pages 186–193. Ieee.

[Pitt and Myung, 2002] Pitt, M. A. and Myung, I. J. (2002). When a good fit can be bad. *Trends in Cognitive Sciences*, 6(10):421–425.

[Pollock et al., 2013] Pollock, L., Vijay-Shanker, K., Hill, E., Sridhara, G., and Shepherd, D. (2013). Natural language-based software analyses and tools for software maintenance. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7171 LNCS, pages 94–125.

[Porter, 1980] Porter, M. (1980). An algorithm for suffix stripping. *Program: electronic library and information systems*, 40(3):211–218.

[Quintans et al., 2010] Quintans, C., Santos, K., and Guerrero, D. (2010). Avaliação e Melhorias de Algoritmos para Extração de Radicais de Identificadores Codificados em Português. Technical report, Departamento de Sistemas e Computação – Universidade Federal de Campina Grande – PB – Brasil.

[Rahman and Devanbu, 2011] Rahman, F. and Devanbu, P. (2011). Ownership, experience and defects: A fine-grained study of Authorship. *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 491.

[Ramon Sangüesa Solé and Serra, 2001] Ramon Sangüesa Solé and Serra, J. M. P. (2001). Netexpert: A multiagent system for expertise location. *Proceedings of the IJCAI-01*, pages 85–93.

[Rastkar et al., 2011] Rastkar, S., Murphy, G. C., and Bradley, A. W. J. (2011). Generating natural language summaries for crosscutting source code concerns. In *IEEE International Conference on Software Maintenance, ICSM*, number Section II, pages 103–112.

[Romano et al., 2011] Romano, S., Scanniello, G., Risi, M., and Gravino, C. (2011). Clustering and lexical information support for the recovery of design pattern in source code. *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 500–503.

[Saha et al., 2013] Saha, R. K., Lease, M., Khurshid, S., and Perry, D. E. (2013). Improving bug localization using structured information retrieval. *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–355.

[Santos et al., 2013] Santos, G. J. D. S., Santos, K. D. F., Valente, M. T., Guerrero, D. D. S., and Anquetil, N. (2013). TopicViewer : Evaluating Remodularizations Using Semantic Clustering. In *The Brazilian Conference on Software: Theory and Practice*, Brasilia.

[Santos, 2012] Santos, K. D. F. (2012). Decomposição do Modelo de Vocabulário de Software baseada na Visão Modular Estrutural da Arquitetura. Technical report, UFCG - Universidade Federal de Campina Grande, Campina Grande.

[Santos et al., 2010] Santos, K. D. F., Dario, D., Guerrero, S., and Jorge, C. (2010). An Empirical Study on the Quantitative and on the Occurrence of Vocabulary Terms in Java Code. Technical report, UFCG - Universidade Federal de Campina Grande, Campina Grande.

[Santos et al., 2012] Santos, K. d. F., Guerrero, D. D. S., de Figueiredo, J. C. A., and Bittencourt, R. A. (2012). Towards a Prediction Model for Source Code Vocabulary. In *1st Workshop on the Next Five Years of Text Analysis in Software Maintenace - ICSM2012*, pages 0–4.

[Santos, 2011] Santos, K. F. (2011). Relacionando Medida de Tamanho de Código Fonte

LOC com a Natureza do Vocabulário Léxico de Projetos Java Open Source. Technical report, Campina Grande.

[Santos et al., 2011] Santos, K. F., Guerrero, D. D. S., and Figueiredo, J. C. A. D. (2011). Understanding the Occurrence of Vocabulary Terms in Java Code. In *VIII Workshop de Manutenção de Software Moderna (WMSWM) - X Simpósio Brasileiro de Qualidade de Software*, Curitiba.

[Savoy, 2006] Savoy, J. (2006). Light Stemming Approaches for the French, Portuguese, German and Hungarian Languages. *Proceeding SAC '06 Proceedings of the 2006 ACM symposium on Applied computing*, pages 1031–1035.

[Sayão, 2001] Sayão, L. F. (2001). Modelos teóricos em ciência da informação - abstração e método científico. *SciELO Brasil*, 30(1):82–91.

[Seaman, 1999] Seaman, C. B. (1999). Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572.

[Servant, 2013] Servant, F. (2013). Supporting Bug Investigation Using History Analysis. *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 754–757.

[Shmueli, 2010] Shmueli, G. (2010). To Explain or to Predict? *Statistical Science*, 25(3):289–310.

[Shneiderman, 1980] Shneiderman, B. (1980). *Software Psychology*. Winthrop, Cambridge, Massachusetts.

[Singer et al., 2002] Singer, J., Storey, M.-a., and Damian, D. (2002). Selecting Empirical Methods for Software Engineering Research. *Guide to Advanced Empirical Software Engineering*, pages 285–311.

[Sipser, 2007] Sipser, M. (2007). *Introdução à Teoria da Computação*. Thomson.

[Sneed, 1996] Sneed, H. (1996). Object-oriented COBOL recycling. *Proceedings of WCRE '96: 4rd Working Conference on Reverse Engineering*.

[Song et al., 2005] Song, X., Tseng, B., Lin, C., and Sun, M. (2005). ExpertiseNet: Relational and Evolutionary Expert Modeling. *User Modeling 2005*.

[Sun Microsystems, 2012] Sun Microsystems (2012). Java Code Conventions. http://java.sun.com/docs/codeconv/CodeConventions.pdf.

[Terceiro et al., 2012] Terceiro, A., Mendonça, M., Chavez, C., and Cruzes, D. S. (2012). Understanding Structural Complexity Evolution: A Quantitative Analysis. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 85–94. IEEE.

[Terceiro et al., 2010] Terceiro, A., Rios, L. R., and Chavez, C. (2010). An Empirical Study on the Structural Complexity Introduced by Core and Peripheral Developers in Free Software Projects. In *2010 Brazilian Symposium on Software Engineering*, pages 21–29. IEEE.

[Travassos, 2011] Travassos, G. H. (2011). Experimentação em Engenharia de Software: Fundamentos e Conceitos. In *SBQS - X Simpósio Brasileiro de Qualidade de Software*, Paraná- Curitiba.

[Wasserman, 2004] Wasserman, L. (2004). *All of Statistics: A Concise Course in Statistical Inference*. Springer, second edition.

[Wazlawick, 2010] Wazlawick, R. (2010). Reflections about Research in Computer Science regarding the Classification of Sciences and the Scientific Method. 6:3–9.

[Zelkowitz and Wallace, 1998] Zelkowitz, M. and Wallace, D. (1998). Experimental models for validating technology. *Computer*, 31(5):23–31.

[Zhang, 2009] Zhang, H. (2009). Discovering power laws in computer programs. *Information Processing & Management*, 45(4):477–483.

Apêndice A

Ferramental Desenvolvido

Este capítulo objetiva descrever o ferramental por nós desenvolvido para investigar vocabulário de software desenvolvidos em Java, o *Vocabulary Tool*. Serão apresentados os artefatos já produzidos neste trabalho, assim como a interação entre eles.

A.1 *Vocabulary Tool*

O *Vocabulary Tool* consiste num conjunto de artefatos de software para extração a partir do código fonte, manipulação e análise de vocabulário de projetos Java. É composto de duas principais aplicações, o *Vocabulary Extractor* e o *Terms Counter*; e definido um formato de armazenamento para as informações de vocabulário. A Figura A.1 a seguir, apresenta uma visão geral dos seus artefatos de software e, como se dá o processamento sobre um código fonte Java.

A.1.1 *Vocabulary Extractor*

O *Vocabulary Extractor* extrai informações de vocabulário a partir de um código fonte java. Ele lê um fonte java, e fazendo uso da API da biblioteca *Java Development Tools - JDT*[1] do Eclipse, mapeia completamente o respectivo código para uma estrutura sintática em árvore,

[1] A biblioteca *Java Development Tools - JDT* do Eclipse possui um conjunto de operações que permitem a obtenção da *AST* de um arquivo fonte java

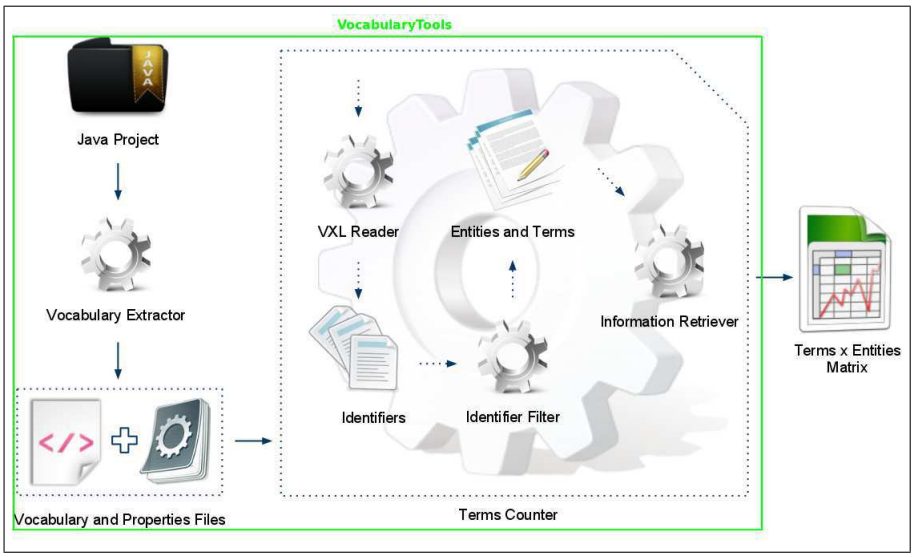


Figura A.1: Artefatos de Software que compõem o Vocabulary Tool.

(*Abstract Syntax Tree - AST*), equivalente. A listagem A.1 a seguir, exemplifica a obtenção da *AST* para um dado fonte.

Código Fonte A.1: Obtenção da AST de um arquivo fonte java.

```
package org.splabs.vocabulary.vxl.browsers;
import java.util.*;
import org.eclipse.jdt.core.dom.*;
import org.splabs.vocabulary.vxl.processors.*;
public abstract class CompilationUnitParser {
  protected static CompilationUnit astNode;
  public static StringBuffer parse(char[] sourceCode, String fileName, String pathRoot) {
    CompilationUnitProcessor.setSourceCode(new String(sourceCode));
    ASTParser parser = ASTParser.newParser(AST.JLS3);
    parser.setSource(sourceCode);
    // setting java compilationUnit
    Map<String, String> options = JavaCore.getOptions();
    parser.setCompilerOptions(options);
    astNode = (CompilationUnit) parser.createAST(new NullProgressMonitor());
    return (new CompilationUnitProcessor(astNode, fileName, getPackage(pathRoot, fileName))
        ).getVxlFragment();
  }
  private static void setCompilationUnitComments(CompilationUnit compilationUnit, String
      sourceCode) {
    for(Comment element : (List<Comment>) compilationUnit.getCommentList())
    if(element instanceof BlockComment || element instanceof LineComment) {
      CompilationUnitProcessor.addCommentUnit(new CommentUnit(element, sourceCode));
    }
  }
}
```

Em seguida, o *Vocabulary Extractor* percorre os nodos da *AST*, extraindo os elementos estruturais de interesse. Desses elementos, com o uso da API do *JDT* o *Vocabulary Extractor* coleta as informações de vocabulário. Para cada tipo de elemento estrutural, além dos seus

identificadores, adicionalmente outras informações são também extraídas e armazenadas:

- para pacote, o seu *namespace*;
- para classe, se é ou não abstrata, se é ou não *interface*, se é interna ou não a outra classe, seus comentários e *javadoc*;
- para atributo, sua visibilidade e seu *javadoc*;
- para método, sua visibilidade, seus comentários e seu *javadoc*;
- para enumeração, suas constantes, seus comentários e seu *javadoc*;

O *Vocabulary Extractor* tem a fucionalidade de ler código interno de métodos. Nesse caso, são extraídos identificadores utilizados nos comandos java: expressões, instanciações de objetos, invocações de métodos, conteúdo das constantes de *strings* literais. Já as informações de vocabulário que são armazenados para variáveis locais, para constante literal e para invocação de método, são seus respectivos totais de ocorrências;

A.1.2 *Vocabulary eXtended Language - VXL*

Toda a informação de vocabulário extraída pelo *Vocabulary Extractor* é armazenada num arquivo cujo formato é baseado na linguagem *XML* (*eXtended Markup Language*), denominado de *Vocabulary eXtended Language - VXL*. A listagem A.2, a seguir, exibe o arquivo em formato *VXL* gerado a partir do código A.1.

Código Fonte A.2: Exemplo de VXL gerado a partir de um projeto Java.

```
<?xml version="1.0" encoding="UTF-8"?>
<java-project id="default" name="CompilationUnitParser" revision="default">
  <pkg name="/CompilationUnitParser.java:org.splabs.vocabulary.vxl.browsers">
    <class name="/CompilationUnitParser.java:org.splabs.vocabulary.vxl.browsers.
        CompilationUnitParser" intfc="n" abs="y" inn="n" sloc="18" jdoc="">
      <field name="astNode" access="prot" jdoc=""/>
      <mth name="/CompilationUnitParser.java:org.splabs.vocabulary.vxl.browsers.
          CompilationUnitParser.parse(char[], String, String)" access="pub" jdoc="">
        <comm cntt="setting java compilationUnit"/>
        <param name="sourceCode"/>
        <param name="fileName"/>
        <param name="pathRoot"/>
        <lvar name="ASTParser" count="1"/>
        <lvar name="CompilationUnitProcessor" count="1"/>
```

```
        <lvar name="JavaCore" count="1"/>
        <lvar name="astNode" count="2"/>
        <lvar name="fileName" count="1"/>
        <lvar name="options" count="2"/>
        <lvar name="parser" count="4"/>
        <lvar name="sourceCode" count="2"/>
        <mthinv name="createAST" count="1"/>
        <mthinv name="getOptions" count="1"/>
        <mthinv name="getVxlFragment" count="1"/>
        <mthinv name="newParser" count="1"/>
        <mthinv name="setCompilerOptions" count="1"/>
        <mthinv name="setSource" count="1"/>
        <mthinv name="setSourceCode" count="1"/>
      </mth>
      <mth name="/CompilationUnitParser.java:org.splabs.vocabulary.vxl.browsers.
          CompilationUnitParser.setCompilationUnitComments(CompilationUnit, String)" access
          ="priv" jdoc="">
        <param name="compilationUnit"/>
        <param name="sourceCode"/>
      </mth>
    </class>
  </pkg>
</java-project>
```

A organização interna das informações do vocabulário são gravadas no *VXL* baseado num esquema *XSD* (*XML Schema Document*). Para fins dos nossos estudos, esse esquema foi proposto com o intuito de: manter a hierarquia dos identificadores de acordo com o escopo do qual fazem parte na estrutura do código fonte, e de armazenar as características de cada tipo de elemento estrutural. A Figura A.2, a seguir, mostra o diagrama.

O armazenamento em *VXL* possibilita que outras ferramentas, de posse do esquema *XSD - XML Schema Document*, possam acessar as informações de vocabulário. Já a manutenção da hierarquia das entidades, possibilita que estudos outros sobre vocabulário, mesmo aqueles que precisem se relacionar com dados estruturais, possam fazer uso do nosso ferramental. Repositórios de arquivos *VXL* extraídos de códigos fontes de projetos *open source* por exemplo, podem ser construídos e disponibilizados para comunidade científica.

A.1.3 *Terms Counter*

A segunda etapa é realizada pelo *Terms Counter*, que gerencia as funcionalidades dos componentes utilitários *VXL Reader*, *Identifier Filter*, *Information Retriever (IR)*, com o objetivo de gerar como saída uma matriz de frequência de termos por entidades. A seguir, resumidamente, estão descritos cada um desse componentes.

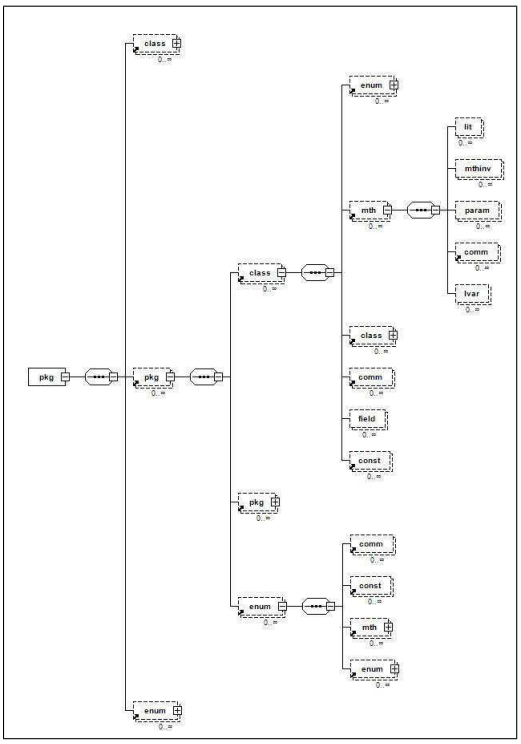


Figura A.2: Esquema XSD do VXL.

VXL Reader

Realiza a leitura do arquivo no formato *VXL* que contém o vocabulário do código fonte. Através de seu arquivo de propriedades, pode-se configurar quais os elementos do vocabulário (classe, interface, atributos, etc) devem ser carregados para processamento. É possível também configurar entre pacotes, ou classes e interfaces, ou métodos quais deles serão considerados *containers* de vocabulários, entidades.

De acordo com interesse onde os identificadores dos elementos de código, incluindo javadoc e comentários configurados pelo usuário no arquivo de propriedades são carregados

Identifier Filter

Sequência de filtros responsáveis pelo processo de normalização dos termos que compõem um vocabulário. Os filtros disponíveis são:

1. *Underscore*: Separação de identificadores compostos, em função da existência *underscores*, "_", em *tokens*;

2. *CamelCase*: Separação de identificadores compostos, em função da mudança da sequência de caracteres minúsculos para maisúculos ou vice e versa, em *tokens*;

3. *Stop Words*: Remoção de palavras com pouca relevância;

4. *Stemming*: Extração do radical de palavras;

5. *Limit Term Length*: Eliminaçaõ de termos com tamanho menor que um limite.

A sequência da aplicação dos filtros é configurada através de seu arquivo de propriedades. Uma sequência típica está representada na Figura A.3.

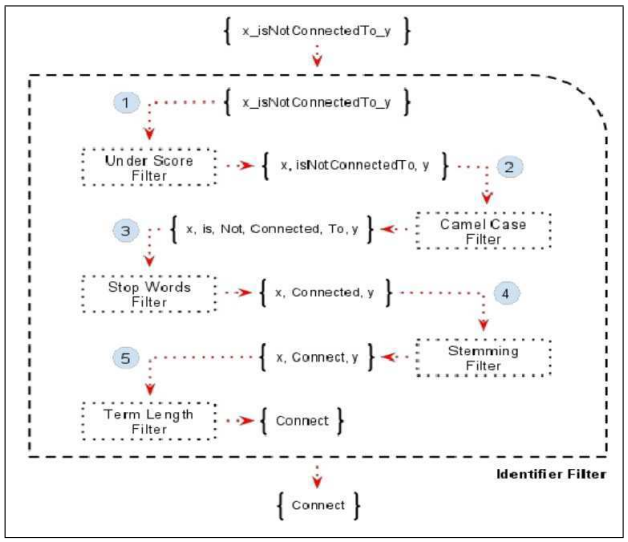


Figura A.3: Execução típica dos filtros do Identifier Filter.

Information Retriver - IR

Realiza a contagem das frequências de termos de acordo com as principais técnicas de *Information Retrieval* [D. Manning et al., 2008]:

1. *Term Frequency*: dá ao termo o valor igual à quantidade de vezes em que ele se repetiu no documento, na entidade;

2. *Inverse Document Frequency*: tende a atribuir um valor alto para termos raros e valores mais baixos para os mais frequentes;

3. *Term Frequency* and *Inverse Document Frequency*: os valores das frequências são obtidos por meio da multiplicação dos valores de *Term Frequency* de cada par termo-entidade e o *Inverse Document Frequency* do termo.

Apêndice B

Artigo publicado no WMSWM-SBQS2011

WMSWM (Worshop de Manutenção de Software Moderno) do SBQS (Simpósio Brasileiro de Qualidade de Software) em Junho de 2011, Curitiba, Brasil.

Understanding the Occurrence of Vocabulary Terms in Java Code

Katyusco de F. Santos[1], Dalton D. S. Guerrero[1] and Jorge C. Abrantes de Figueiredo[1]

[1]Software Practices Lab - Department of Systems and Computing
UFCG - Federal University of Campina Grande
Campina Grande, PB - Brazil

katyusco@copin.ufcg.edu.br, {dalton,abrantes}@dsc.ufcg.edu.br

Abstract. *This empirical study aims to characterize the relation between the number of vocabulary terms and the design size of a system, as well as, the occurrence of these terms along the code. We analyzed the Java code of 15 open source projects from SourceForge and proposed an analytic-statistical model for occurrence of terms and their frequency over code. We concluded that both the number of terms and their frequencies can be expressed as function of the number of software entities (classes and interfaces). It was verified that: while the number of distinct terms is a linear function of the number of entities, the total number of terms is a power function; furthermore the frequencies of terms along the code could be modeled by a Power Law distribution of Zipf type.*

1. Introduction

In object oriented paradigm identifiers are used to name static structures of a source code. They naming structures that encapsulate others (Entities), such as, classes and interfaces; and simple ones, such as attributes and parameters. They play an important role in program concepts, and choosing meaningful names is an important decision to facilitate program comprehension and maintenance understandable [Haiduc and Marcus 2008], [Host and Ostvold 2007] and [Butler et al. 2010].

Good strategies to evaluate systems must combine different measures from diverse perspectives [Lanza and Radu 2006]. Recently studies, suggested that information extracted from software vocabulary (*e.g.:* entropy) reveals other aspects not captured by traditional structural metrics [Arnaoudova et al. 2010]. These aspects can help to understand and to mitigate the software aging phenomenon [Parnas 1994]. Besides, they could give stakeholders insights on how to make better tools and techniques to improve software development process.

In this context, we carried out an empirical study on 15 Java open source projects. The goal was to characterize vocabulary size and to identify potential relationship between its elements (terms quantitative) and design size (number of entities: classes and interfaces) of a system. We processed 4.5 thousand of entities, extracted more than 11.6 thousand of different terms and nearly 463.5 thousand of total terms. The study concluded there is a typical relation between vocabulary size and design size of Java open source systems: while the amount of different terms is linear function of number of entities, the total of terms is a power function; furthermore the frequencies of terms along the code could be modeled by a Zipf distribution (a Pareto type [Downey 2005]).

The rest of this article presents defnitions in Section 2. Section 3 describe research questions and experiment design. Experiment execution is presented in Section 4. Section 5 details statistical analysis. Results interpretation and threat analysis are showed in Section 6. Section 7 reports related works. And, conclusions are described in Section 8.

2. Basic Definitions

The vocabulary of source code is a set of distinct (unique) terms that appears in the identifiers [Abebe et al. 2009]. We refer to elements of each vocabulary as terms, since many of them are not proper words from a spoken language. Identifiers have their terms extracted by a normalization process that in the context of this study, is comprised by: first, separating tokens from composed identifiers based in camel case notation style. Then, they are submitted to Stop words removal [1] and Stemming[2].

The identifier "isDataBaseConnected" of **House** class, an entity *(E)*, presented in Listing 1 is splitted into four lowercase tokens: "is", "data", "base" and "connected". Removing the stopword "is", just "data", "base" and "connected" are considered. When stemming is executed "data", "base" and "connect" are the resulting terms.

Listing 1. House Java Class

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

A term may be different from all others already extracted, or identical to any of them. Identical terms are those having the same sequence of characters regardless of identifiers the terms were extracted from. For the purpose of this research we use two measures to characterize vocabulary size: number of **Distinct Terms *(DT)*** and number **Total Terms *(TT)***. While the *(DT)* accumulates the quantity of unique terms, the *(TT)* accounts the occurrence frequency (repetitions) of each one. In the House entity the terms "house", "set" and "get" appeared just one time; "db" two times; "data" and "base" repeated three times each one; and "connect" four times. So, the vocabulary size of the entity House has seven *(DT)*, and fifteen *(TT)*.

3. Design of the Study

This study trying to answer two questions: is the size of software vocabulary a function of system design size? Do term repetitions along a code follow a pattern? We conducted a case study on 15 Java open source systems where hypotheses were formulated and tested.

3.1. Research Questions and Hypotheses

Since we characterized vocabulary size as the quantities of *(DT)* and of *(TT)*, and design size as the number of *(E)*, two Research Questions (**RQ**) were derived from the first question, while from the second only one.

RQ-1: Do systems with large design size, also have great number of *DT* to name identifiers? Thus, we formulated the hypothesis - I (*H-I*) as follows:

[1]Language-word that has no significance meaning, such as "the", "is", and "of".

[2]Process for reducing inflected words to their stem, *e.g.*: "connect**ing**" become "connect".

Table 1. List of software projects.

Index	Project	Version	Entities	Total Terms	Distinct Terms	Download site
1	JUnit	4.5 compact	23	1,176	131	http://sourceforge.net/projects/junit/
2	VilloNanny	1.0.0	25	846	175	http://sourceforge.net/projects/villonanny/
3	EasyMock	2.4	63	1,463	168	http://sourceforge.net/projects/easymock/
4	PDF SaM	1.0.1	68	2,386	310	http://sourceforge.net/projects/pdfsam/
5	PJirc	2.2.1	133	4,181	431	http://sourceforge.net/projects/pjirc/
6	SweetHome3D	1.3.1	100	11,969	549	http://sourceforge.net/projects/sweethome3d/
7	Jvlt	1.1.1	235	8,529	451	http://sourceforge.net/projects/jvlt/
8	Jedit	4.2	234	15,969	1,036	http://sourceforge.net/projects/jedit/
9	Robocode	1.6.0.1	250	18,012	800	http://sourceforge.net/projects/robocode/
10	Jgnash	1.11.7	319	19,025	911	http://sourceforge.net/projects/jgnash/
11	JabRef	2.4.b2	462	22,315	1,402	http://sourceforge.net/projects/jabref/
12	JfreeChart	1.0.10	546	115,007	778	http://sourceforge.net/projects/jfreechart/
13	JavaGroups	2.6.3 GA	555	40,876	1,378	http://sourceforge.net/projects/javagroups/
14	PMD	4.2.3	569	76,376	1,228	http://sourceforge.net/projects/pmd/
15	FindBugs	1.3.5	969	125,627	1,863	http://sourceforge.net/projects/findbugs/
Totals			**4,551**	**463,484**	**11,611**	

•*H*-I_0 (null): The number of *DT* in a system is not a function of its number of *E*;
•*H*-I_1 (alternative): The number of *DT* in a system is a function of its number of *E*.

RQ-2: The greater the design size of system, the greater its quantity of *TT*? The hypothesis-II (*H-II)* was formulated as:
•*H*-II_0: The number of *TT* in a system is not function of its number of *E*;
•*H*-II_1: The number of *TT* in a system is function of its number of *E*.

RQ-3: Does the frequency of terms that belongs to a system code follow a pattern, a statistical distribution? Thus, the following hypothesis - III (*H-III*):
•*H*-III_0: The frequency of terms has a different statistical distribution for each system;
•*H*-III_1: The frequency of terms follows a standard statistical distribution regardless of the software system.

3.2. Design of Experiment

The experiment was divided into three steps: selecting Java open source projects (subjects); extracting quantitative about vocabulary terms and about entities; finally, performing data analysis[3].

Selecting Subjects. Table 1 shows 15 Java open source projects from SourceForge[4] that were selected and downloaded. To be included in the sample, the projects should meet the following criteria: 1)systems should be open source and have their jar file available; 2)identifiers coded in English and in camel case notation style; There should be systems with different sizes and purposes Finally, each selected project should have an active developer community, but, not necessarily the most popular.

Extracting Term and Entities Quantities. *Terms Counter* tool scanned projects' bytecode, and extracted identifiers of classes, interfaces, attributes and methods. Over each identifier it applies the normalization process defined in Section 2 to isolate terms. Then, it produced term quantities of system vocabulary and number of design entities. Table 1 presents values of *Entities*, of *(DT)* and *(TT)* for all processed projects.

[3]In http://www.gmf.ufcg.edu.br/∼katyusco/experiments are the details to reproduce this experiment.
[4]Available in http://www.sourceforge.net

Table 2. Fit Ajustment for Distinct and Total Terms as a Function of Entities.

Predicted	Regression	Predictor	Model	Transformation	R^2	*p-value for the Model*
Distinct Terms	**Linear**	**E**	$DT = a + b * E$	$DT \sim E$	**83.51%**	**1.915e-06**
	Exponential	E	$DT = a * b^E$	$log(DT) \sim E$	68.85%	1.297e-04
	Power	E	$DT = a * E^b$	$log(DT) \sim log(E)$	79.16%	1.817e-04
Total Terms	Linear	E	$TT = a + b * E$	$TT \sim E$	79.51%	8.008e-06
	Exponential	E	$TT = a * b^E$	$log(TT) \sim E$	79.29%	8.601e-06
	Power	**E**	$TT = a * E^b$	$log(TT) \sim log(E)$	**90.61%**	**4.749e-08**

Performing Data Analysis. Scripts in R language [Everitt and Hothorn 2010] generated descriptive statistics (mean, median, standard deviation, coefficients of correlation), and plotted charts and histograms. Several linear regressions, were conducted to detect potential relationship between number of *E* and term quantities (*DT* and *TT*). Every candidate model were analyzed according to its: Coefficient of Determination (R^2)[5] [Barbetta et al. 2008]; *p-value* of *T-test*[6] and of *F-test*[7]; and graphical analysis of residual error [Jain 1991]. The analysis process was refined until the relations converge to an acceptable model or until evidences revealed a nonplausible models.

4. Experiment Execution

The experiment execution consists to pass the subjects, the jar file of systems, successfully, through each of its design steps (Section 3.2). No human preparation is necessary.

5. Statistical Analysis

This experiment analyzed the bytecode of 15 Java systems and processed more than 4.5 thousand of design entities. Besides, it extracted nearly 463.5 thousand of total terms and more than 11.6 thousand of distinct terms (See the Totals in Table 1).

5.1. Relationship between Terms Quantitatives and Number of Entities.

The correlation between *E* and *DT* is **0.9138**, and between *E* and *TT* is **0.8917**. Both values are positive and suggest that design size of a system is related to quantity of terms of its vocabulary.

Relationship between *DT* and *E*. Linear, exponencial and power regression were fit to model the relation between *DT* and *E* and their statistics results can be compared in Table 2. The linear model best estimates *DT* as a function of *E*, and is given by:

$$\mathbf{DT = 232.434 + E * 1.785} \quad (1)$$

Relationship between *TT* and E. Table 2 also presents the statistics of regressions to fit the relation between *E* and *TT*. The power model best predicts *TT* on *E*:

$$\mathbf{TT = 10.493 * E^{1.34}} \quad (2)$$

5.2. Pattern of Term Frequencies

The term frequencies for each system have the same behavior: many terms repeat a few and little terms repeat a lot. When terms are sorted from the most frequent to the least

[5]Represents the explained data percentage by the model.
[6]Indicates the significance for the regression coefficients.
[7]Indicates the significance of the model as a whole.

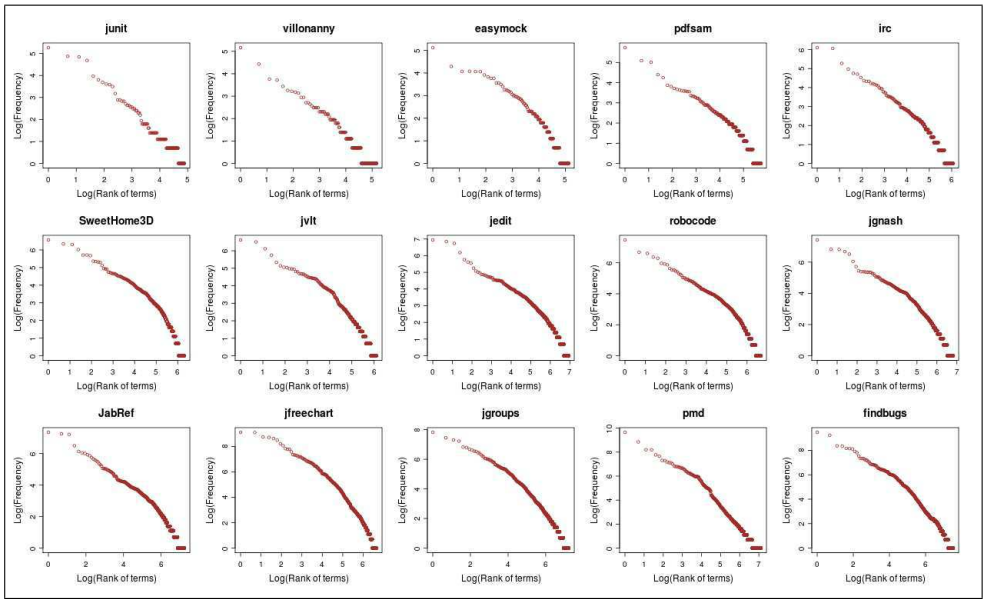


Figure 1. Graphic *log (Frequency)* per *log (Rank of terms)* of all 15 systems.

one, and plotted on the logarithm scale graph, they describe a curvature (See Figure 1). hindering statistical distribution idetification [Feitelson 2009]. The Vuong statistc test[8] asserted that 14 of 15 (93.33%) of systems were classified as having a Pareto distribuition, and just 1 (6.67%), **PMD** project, as having Log-normal one.

The values of Pareto[9] constants, C and b ,are presented in Table 3. All R^2 are above 88%, even for PMD that is the highest, 98.06%. Since all b values are negative (-2.011, -1.061), a Zipf statistical distribuition[10] describes the pattern of term frequencies, F, for all observed systems:

$$\mathbf{F = C * (r^{-b}) \Rightarrow F = C/(r^{b\_Zipf})}, \tag{3}$$

where: C is a constant, b_Zipf is the positive value of b Pareto tail index, and r represents the position term in the rank $(r = 1, 2, ..., n)$. To illustrate, take $b\_Zipf = 1$ for a ranking of $n = 50$ terms. So, $F_1 = \frac{C}{(1^1)}$, $F_2 = \frac{C}{(2^1)}$,..., $F_{50} = \frac{C}{(50^1)}$. Frequency of the first term is twice of second, $F_1 = 2 * F_2$, frequency of 5th term is ten times of 50th, $F_5 = 10 * F_{50}$.

5.3. Relationship between Zipf Coefficients and Terms Quantitatives

Constant C has a high positive correlation with *TT*, **0.8860**, that is best modeled by a power function. While the tail index b_Zipf has a high negative correlation with *TT*, **-0.8896**, whose hyperbolic model is the best. Table 4 shows the statistics for both models.

Substituting respectively C and b_Zipf from equation 3 by its repectively models (column Equation of Table 4) terms frequency of a system follows a pattern, a Zipf statistical distribution, defined by:

$$\mathbf{F = \frac{(0.00417 * TT^{1.586})}{(r^{TT/(-1905.42 - 0.54 * TT)})}} \tag{4}$$

[8]Based on *maximum likelihood estimation* and implemented in R scripts by [Clauset et al. 2009].
[9]$F = C * (r^b)$, where: C is a constant, b is the tail index, and r is the term position in the ranking.
[10]Belongs to Pareto family.

Table 3. Values of C and b, *CI(95%)*, R^2 and Pareto equation for all 15 systems.

Project	C	*CI(95%)* for C		b	*CI(95%)* for b		R^2	Equation
		Lower	**Upper**		**Lower**	**Upper**		
JUnit	355.97	342.68	369.76	-1.189	-1.212	-1.166	97.55%	$F = 355.97 * (r^{-1.189})$
VilloNanny	203.38	195.88	211.17	-1.061	-1.088	-1.033	95.09%	$F = 203.38 * (r^{-1.061})$
EasyMock	633.10	588.20	681.430	-1.229	-1.269	-1.188	90.12%	$F = 633.10 * (r^{-1.229})$
PDF SaM	759.74	726.49	794.50	-1.130	-1.156	-1.104	92.14%	$F = 759.74 * (r^{-1.130})$
irc	1,882.86	1,823.18	1,944.50	-1.240	-1.258	-1.221	94.69%	$F = 1,882.86 * (r^{-1.240})$
SweetHome3D	14,964.93	14,145.39	15,831.955	-1.461	-1.485	-1.438	88.77%	$F = 14,964.933 * (r^{-1.461})$
Jvlt	8,090.58	7,741.77	8,455.11	-1.436	-1.456	-1.416	93.31%	$F = 8,090.58 * (r^{-1.436})$
Jedit	11,571.72	11,224.72	11,929.44	-1.303	-1.318	-1.289	91.90%	$F = 11,571.72 * (r^{-1.303})$
Robocode	23,601.08	22,692.03	24,546.54	-1.462	-1.479	-1.445	91.46%	$F = 23,601.08 * (r^{-1.462})$
Jgnash	24,095.66	23,366.46	24,847.62	-1.455	-1.469	-1.441	93.44%	$F = 24,095.66 * (r^{-1.455})$
JabRef	21,599.64	21,129.42	22,080.33	-1.366	-1.377	-1.355	93.77%	$F = 21,599.64 * (r^{-1.366})$
JfreeChart	993,156.74	948,428.02	1,039,994.90	-2.011	-2.025	-1.996	93.93%	$F = 993,156.74 * (r^{-2.011})$
JavaGroups	84,787.31	82,912.87	86,704.13	-1.549	-1.559	-1.540	95.23%	$F = 84,787.31 * (r^{-1.549})$
PMD	**152,321.77**	**150,116.27**	**154,559.68**	**-1.719**	**-1.726**	**-1.712**	**98.05%**	$\mathbf{F = 152,321.77 * (r^{-1.719})}$
FindBugs	565,907.02	554,841.62	577,193.11	-1.740	-1.748	-1.732	95.96%	$F = 565,907.02 * (r^{-1.740})$

Table 4. Statistics for Zipf Constants

Predicted	Regression	Predited	Model	Equation	R^2	*p-value* for the model
C	power	TT	$C = a * TT^b$	$C = 0.00417 * TT^{1.586}$	98.22%	5.801e-13
b_Zipf	hyperbolic	TT	$b\_Zipf = TT/(a + b * TT)$	$b\_Zipf = TT/(-1905.42 - 0.54 * TT)$	98.79%	4.772e-14

6. Interpretations

The models for *DT* and *TT* as a function of *E* reject the null hypothesis of RQ-1 and of RQ-2 respectively promote a initial understanding of the relation between vocabulary size and design size. Moreover the Zipf distribuition answered the RQ-3 pointing a typical behavior of the term occurrences along a system code.

Vocabulary understanding is a new aspect to be investigate that can contribute with state-of-the-art of software development process. Monitoring properties of the terms during a software development suggest a new perspective to analyze software. And, associating vocabulary understanding with traditional techniques of dynamic and static analysis, and metrics can improve activities of software maintenance.

Having prior knowledge of the key terms that encode a software project, feature location tools could use such terms as seeds to make better search with more accurate results. For impact analysis new approaches to define the scope and precision of the impact of changes can be associated with measures of vocabulary (*e.g.:* entropy, cover context [Arnaoudova et al. 2010]). Identifying patterns of behaviors in terms of modules can promote changes in clustering algorithms or even new algorithms, carrying out benefits to architectural recovery.

6.1. Threat Analysis

All results are restricted to classes and interfaces as types of entities; Neither terms of comments, methods parameters, local variables nor javadoc were considered. The open source characteristic of the sample, development processes, rules and styles of corporations programming prevent the generality of results

7. Related Works

Several researches have stressed the importance of software vocabulary to facilitate maintenance of programs. Lawrie [Lawrie et al. 2006] conducted an empirical study about vocabulary stating that the quality of identifiers affects software comprehension. Based on several versions of 2 C++ open systems, Abebe [Abebe et al. 2009] showed that identifiers are related to problem domain concepts. Haiduc [Haiduc and Marcus 2008] analyzed 6 libraries (Java, C++) of the same kind, and concluded that identifiers are chosen in accordance with stakeholders' personal preferences and experiences as well as problem domain jargons. According to Host, identifiers that do not reflect functional requirements affect software understanding [Host and Ostvold 2007]. Software understanding and maintenance are degraded when identifiers do not follow good convention rules (e.g.: number of words), said Butler [Butler et al. 2010] studying 8 Java open source projects.

Linstead [Linstead et al. 2009] grouped vocabularies of 12,151 open source projects into a unique large corpus comprised of entity identifiers (along with methods and fields). A statistical analysis revealed a power law distribution on terms according to their grammatical class (nouns, verbs, adjectives, etc). Anslon [Anslow and Tempero 2008] extracted words from class identifiers, from version 1.1 to 1.6 of Java API in order to follow the evolution of word classes and frequencies. Zhang [Zhang 2009], carried out an empirical study on lexical tokens of 24 real-world open source softwares (Java, C++ and C) and observed that power law regularities exist in computer programs.

Arnaudova *et. al.* [Arnaoudova et al. 2010] defined the measures entropy and context coverage for vocabulary terms and related then to fault proneness. And, [Eshkevari 2010] showed that entities containg terms with high values for these measures are more fault-prone.

8. Conclusions

Previous studies have related software vocabularies with descriptive statistics [Anslow and Tempero 2008] and distributions [Linstead et al. 2009], [Zhang 2009]. But, in the best of our knowledge, none of them towards to identify models to term quantities (vocabulary size) and occurrences (vocabulary behavior) as a function of entities (design size). We identified that: while the number of distinct terms is a linear function of the number of entities, the total terms is a power one; and the term frequencies along a code is modeled by a *Zipf* statistical distribution. These initial results cannot be statistically generalized, but they seem to point to a typical nature of vocabulary of Java open systems. New perspective analysis of software systems can be revealed combining the vocabulary understanding with traditional metrics. And as consequence, this fact can give clues to improve software maintenance activities, tools and techniques.

9. Acknowledgments

This research has been partially supported by MCT/CNPq-14/2009 project, through grant number 483249/2009-2.

References

Abebe, S. L., Haiduc, S., Marcus, A., Tonella, P., and Antoniol, G. (2009). Analyzing the Evolution of the Source Code Vocabulary. *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, pages 189–198.

Anslow, C. and Tempero, E. (2008). Visualizing the Word Structure of Java Class Names. *Conference on Object Oriented Programming Systems Languages and Applications*, (133):777–778.

Arnaoudova, V., Eshkevari, L., and Oliveto, R. (2010). Physical and conceptual identifier dispersion: Measures and relation to fault proneness. *(ICSM), 2010 IEEE*.

Barbetta, P., Reis, M., and Bornia, A. (2008). *Estatística para Curso de Engenharia e Informática*. São Paulo - Atlas.

Butler, S., Wermelinger, M., Yu, Y., and Sharp, H. (2010). Exploring the Influence of Identifier Names on Code Quality: an empirical study. *oro.open.ac.uk*.

Clauset, A., Shalizi, C., and Newman, M. (2009). Power-law distributions in empirical data. *SIAM*, 51(4):661–703.

Downey, A. B. (2005). Lognormal and Pareto distributions in the Internet. *Computer Communications*, 28(7):790–801.

Eshkevari, L. M. (2010). Linguistic driven refactoring of source code identifiers. *Reverse Engineering, Working Conference on*, 0:297–300.

Everitt, B. S. and Hothorn, I. (2010). *A Handbook of Statistical Analyses Using R*. Chapman & Hall/CRC.

Feitelson, D. G. (2009). *Workload Modeling for Computer Systems Performance Evaluation*.

Haiduc, S. and Marcus, A. (2008). On the use of domain terms in source code. *The 16th IEEE International Conference on Program Comprehension*, 0:113–122.

Host, E. W. and Ostvold, B. M. (2007). The Programmer's Lexicon, Volume I: The Verbs. *Seventh IEEE SCAM)*, I:193–202.

Jain, R. (1991). *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. Wiley New York.

Lanza, M. and Radu, M. (2006). *Objects-Oriented Metrics in Practice*. Springer, Berlin.

Lawrie, D., Feild, H., and Binkley, D. (2006). Quantifying identifier quality: an analysis of trends. *Empirical Software Engineering*, 12(4):359–388.

Linstead, E., Hughes, L., Lopes, C., and Baldi, P. (2009). Exploring Java software vocabulary: A search and mining perspective. In *ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 29–32. IEEE.

Parnas, D. (1994). Software aging. *Proceedings of 16th International Conference on Software Engineering*, pages 279–287.

Zhang, H. (2009). Discovering power laws in computer programs. *Information Processing & Management*, 45(4):477–483.

Apêndice C

Artigos publicados no TAinSM-ICSM2012

TAinSM (Workshop on The Next Five Years of Text Analysis in Software Maintenance) do ICSM (International Conference on Software Maintenance) em Setembro de 2012, Trento, Itália.

Towards a Prediction Model for Source Code Vocabulary

Katyusco de F. Santos
Federal Institute of Pernambuco
Dept. of Systems, Processes and Control
Email: katyusco@recife.ifpe.edu.br

Dalton D. S. Guerrero
and Jorge C. A. de Figueiredo
Federal University of Campina Grande
Email:{dalton, abrantes}@dsc.ufcg.edu.br

Roberto A. Bittencourt
State University of Feira de Santana
Department of Exact Sciences
Email: roberto@uefs.br

***Abstract*—In many research fields, producing synthetic data from probabilistic models to evaluate tools, methods and classification techniques is a common and well accepted practice. In a previous work, we have been able, for instance, to produce synthetic, yet realistic, software designs that allowed us to evaluate and compare architecture recovery algorithms. A few, but relevant, algorithms based on text analysis, however, could not be evaluated using such synthetic dataset. In this paper, we present our initial results concerning the development of probabilistic models that can be used to synthesize realistic software vocabularies, i.e. collections of names and identifiers of software systems. We propose a formal definition of vocabularies based on multisets and present three probabilistic models that we derived from the analysis of the vocabularies of 39 open source software systems. We believe these results can be useful not only for experimenters, but also for other researchers in the field of software text analysis.**

***Index Terms*—Software vocabulary; Empirical research; Statistical modeling; Java open source**

I. Introduction

In a previous study, we have been able to design a controlled and randomized experiment to evaluate and compare how different algorithms for architectural recovery perform under the very same conditions. Such a study required that we synthesized a large amount of artificial software designs that were as similar as possible to real software designs, from a statistical point of view [1], [2].

The approach proved to be useful and successful at some extent. However, a few, but relevant, of the algorithms could not be included in the study. The probabilistic models we developed at the time could synthesize software-realistic design graphs, i.e. dependency graphs, but not natural language found in identifiers and comments. As a consequence, no algorithm for architectural recovery based on text analysis, as opposed to being based on structural dependencies, could be included in the study.

The results we present here have been drawn from our initial studies on synthesising software-realistic vocabularies[1]. The goals of this study have been to characterize and to investigate general properties of real software vocabularies. In what follows, we present our results so far. First, we present our formal definition of vocabularies as multisets over strings and how we characterize some of their properties. Second, we present a statistical analysis of the vocabularies of a set of 39 well known open source software systems. And, third, we present three statistical models for the properties of vocabularies discussed previously, namely for the vocabulary size and for the frequency of terms in a software vocabulary, both as functions of LOC (Lines of Code). In particular, the model for the frequency of terms captures the fact that terms of a software vocabulary have, typically, a long tailed distribution (more specifically, a log-normal distribution). That is, while a few terms repeat very often, the vast majority appear a very small number of times.

While we aim at producing models that allow us to synthesize software vocabularies for experimental purposes, we believe that the formal definition we propose and the statistical characterization of vocabularies we have developed can be useful for other researchers in the field of software text analysis. As is usual in similar studies, no strict claims can be made about generalizations of the statistical results out of our dataset. Nonetheless, certainly a relevant number of software systems are well represented in such a dataset and can be fairly compared to them.

[1]In this paper we specifically focus on source code vocabularies, i.e. names and identifiers from the source code of a software system. Despite that, all concepts and methods apply to vocabularies of other artifacts.

II. Formal Definition of Vocabulary

Informally, a software vocabulary comprises the strings used to name or identify software systems entities. Having a formal definition of vocabularies, however, helps making the understanding and communication more effective and less ambiguous, whenever needed. In this section, we briefly present our formalization of vocabularies.

A. Vocabulary definition

We define a software vocabulary as a multiset of strings, i.e. an application $V : \mathbb{S} \to \mathbb{N}$ that maps strings to natural numbers. Elements of a vocabulary are called *terms*. For any term t, $V(t)$ denotes the number of occurrences of the term t in the vocabulary V. If $V(t) > 0$ we say that t is a term of the vocabulary.

As an example, consider the excerpt of java code in Listing 1. According to our definition, the vocabulary can be expressed as the following multiset of terms:

$$V = 4'SampleClass + 2'sa + 2'sc + 1'me + 1'mt$$

Listing 1. Excerpt of Code

```
public static int mt(){
    SampleClass sa = new SampleClass();
    SampleClass sc = new SampleClass();
    sa.me(sc);
}
```

Observe we can adopt existing multiset notations for vocabularies. Above, we have used formal sums, in which each sum term expresses the number of occurrences n of one vocabulary term t in the form $n's$. Other notations can be convenient for other purposes, as well.

B. Vocabulary operations

In practice, one typically needs to process vocabularies prior to performing actual analysis. Typical information retrieval (IR) processing operations, as well as other vocabulary operations, can be easily and unambiguously specified as functions over vocabularies. Take, as an example, a tokenization operation that splits terms according to a given formation rule (camel case, for instance). This operation can easily be specified as a function[2] $cc : \mathbb{V} \rightarrow \mathbb{V}$, that maps each pair $(t, m) \in V$ to a set of terms $\{(t_1, m), (t_2, m), \ldots, (t_n, m)\}$, where each term t_i is one of the words that composes the original term t. For instance, the vocabulary V_1 above maps, by such operation, to

$$cc(V) = 4'Sample + 4'Class + 2'sa + 2'sc + 1'me + 1'mt$$

C. Vocabulary properties

For our simulation and synthesizing purposes, size metrics are relevant. For that reason, we have defined two metrics related to the size of vocabularies. The first is the total number of terms ($TT(V)$) of a vocabulary. It is defined as the size of the vocabulary multiset, i.e. the sum of the multiplicities of all terms in the vocabulary. The second one is the number of distinct terms ($DT(V)$) which is the number of terms whose multiplicity is at least one[3]. For our example above, we have

$$TT(V) = 10 \quad and \quad DT(V) = 5$$

A third relevant property of a vocabulary is what we call its *Characteristic Frequency Vector*. It is simply the vector of all multiplicities in decreasing order. Characteristic frequency vectors are relevant because they help us characterize vocabularies, abstracting the terms themselves. As an example, the characteristic frequency vector of the vocabulary V above is

$$CVF(V) = (4, 2, 2, 1, 1)$$

Observe that the characteristic frequency vector concept is slightly different from the one used in the so called document vector model in IR techniques. The characteristic frequency vector defined here is a sorted vector, in which we drop the relation between multiplicities and terms. This abstraction is possible because we are not concerned with retrieving documents based on the relevance of specific terms within them. Instead, we are concerned with modeling and characterizing vocabularies.

[2]We denote the set of all vocabularies by $\mathbb{V}$.

[3]It can also be defined as the cardinality of the underlying set of the vocabulary. Recall that for some definitions of multisets, the underlying set is the smallest set of elements from which the multiset can be derived.

III. Empirical Study on Software Vocabularies

In the previous section we developed a set of definitions and concepts about vocabularies in a general and abstract sense. In this section, we dive deep into concrete, real software vocabularies. We have extracted and analyzed the vocabularies of 39 real software systems. Because our aim is to develop a model that allows us to synthesize artificial vocabularies that resembles real ones, we focused on their properties. Specifically, we studied how their total number of terms $TT(V)$, number of distinct terms $DT(V)$ and characteristic frequency vectors $CFV(V)$ relate to the size of the code — we used the lines of code (LOC) metric.

The study was divided into 4 steps: A) obtaining a sample of software systems; B) extracting the basic dataset from the softwares in the sample, viz. LOC and the vocabularies themselves; C) calculating the size metrics from the vocabularies, viz. total terms, distict terms and the characteristic frequency vectors; and D) performing the statistical data analysis itself.

A. Obtaining the sample

We selected and downloaded 39 Java open source projects, being 15 from SourceForge and 24 from Apache. Table I lists for each project its version, LOC, Disticnt Terms and Total of Terms.

TABLE I
Code Size and Vocabulary Size of Projects.

Sample		Source Size	Vocabulary Size	
Project	**Version**	**LOC**	**Distinct Terms**	**Total of Terms**
apacheds	1.5.7	154,906	1,801	65,036
axis2	1.5.3	285,979	2,586	129,620
beehive	v1.0.2	198,914	1,922	77,069
cxf	2.3.1	360,758	3,264	167,618
derby	10.7.1.1	607,710	5,226	256,279
easymock	3.0 r205	11,954	396	5,974
findbugs	1.1.1	73,537	1,638	3,9728
geronimo	3.0M1	194,718	2,279	114,195
Hadoop Common	0.21.0	71,771	1,743	3,4967
Hadoop Mapreduce	0.21.0	172,077	2,429	7,8362
ivy	2.1.0	54,739	1,040	28,535
jabref	2.0b r3397	43,906	1,296	19,040
jackrabbit	2.2.1	269,938	2,155	119,298
james-server	3.0M2	40,166	1,055	20,503
jedit	4.3pre9 r8692	99,656	1,955	37,186
freechart	1.0.8 r2273	127,526	1,160	6,2373
jgnash	2.5 r2456	44,950	1,092	19,208
jGroups	2.12	66,871	1,349	31,360
junit	4.5	11,997	582	7,004
jvlt	1.1.4 r592	1,7363	555	9,451
lucene	3.0.3	171,369	2,297	73,400
myfaces-core	2.0.3	101,475	1,155	46,926
myfaces-tomahawk	1.1.10	128,922	1,373	4,8245
openJPA	2.0.1	354,996	2,851	186,412
pdfsam	2.2.1	26,058	812	16,009
pjirc	2.2.1	10,114	585	4,736
pmd	4.2.5 r7178	60,062	14,64	27,731
robocode	1.7.2.2 r3549	52,267	1,350	29,230
roller	4.1M1	60,484	1,060	28,089
shindig	2.0.2	78,682	1,565	48,446
solr	1.4.1	79,436	1,718	35,459
STRUTS2	2.1.1	144,242	1,894	78,134
SweetHome3D	3	64,876	901	31,529
tapestry	4.1.6	109,757	1,472	64,820
uimaj	2.3.1	17,1448	2,181	95,613
VilloNanny	2.4	14,731	610	5,503
wicket	1.4.0	138,512	1,616	61,575
xalan-J	2.7.1	171,488	1,886	85,736
Xerces-J	2.9.1	131,310	1,621	58,150
Totals		**3,996,045**	**63,934**	**2,348,549**

While we did not use any strict sampling method, we followed these guidelines and criteria to select the systems for this study: 1) the systems should be open source; 2) identifiers should be written in English; 3) composed identifiers should use either camel case or underscore notation; 4) the project should have an active developers community; 5) the project should have a large number of recent downloads within its category and receive more positive than negative reviews. Furthermore, we tried to select systems from different problem domains and with a meaningful range of LOC sizes.

B. Extracting the dataset from the sample

Our second step was to extract the basic dataset from the samples to support the analysis. The basic data consists of the LOC and the vocabulary of each system expressed as a multiset. We collected them using *VocabularyExtractor* [4] tool and stored them VXL[5] files for further processing.

C. Calculating the size metrics for the vocabularies

Our third step was to process each vocabulary and derive their size measurements. The processing consisted in applying the operators for tokenization (both for camel case cc and for underscore us notations), stemming st, stop words removal swr and short terms removal str (for the sake of analysis, we removed all terms with only one character). Finally, we calculated the measurements. That is, for each vocabulary V in the sample, we calculated the processed vocabulary $V' = str(swr(cc(us(V))))$ and then $TT(V')$, $DT(V')$ and $CFV(V')$. All the data was stored in CSV files. For that, we used *TermsCounter* tool. Table I presents each system and the size measurements of the corresponding vocabulary.

D. Performing Data Analysis

An exploratory data analysis [3], [4] was performed to detect trend relationships between software size (LOC) and vocabulary size measurements (*DT* and *TT*). Given the possibility of a relationship, some regression models such as linear, multi-linear and nonlinear [5] were considered as candidates to explain vocabulary size as a function of code size. To select the best fit we applied the Adjusted R^2 and F-Statistics model selection techniques [5]. To model the occurrences of distinct terms in vocabularies, we used model fit test [5] and standard statistical tests [6], [4]. Finally, we used the Vuong statistical test developed by Clauset [7] to detect and classify long tail distributions of the character frequency vectors.

IV. Results and Derived Models

Overall, almost 4 million lines of code, 2.34 million terms and 64,394 distinct terms were considered in this study (see row Totals of Table I). The vocabularies themselves are too large to display in this paper, but they are available in our website.

[4]Please refer to http://bit.ly/softwarevocabularies to find more details on this study, including all necessary data and tools to reproduce the results.

[5]An XML-based file format that maintains the identifiers hierarchy from code.

A. Modeling the vocabulary size

As expected, we identified a strong positive correlation between LOC and the total number of terms TT of a system, with $\rho = 0.98648$. This result confirms the intuitive notion that the longer the program is, the more terms it has. Perhaps not so intuitive, we also found a strong positive correlation between LOC and the number of distinct terms DT, with $\rho = 0.94697$. This result confirms that the larger the program, the more distinct terms it is likely to have. This is probably related to the fact that each piece of code of a system is very likely to implement a unique aspect or feature of the system and, thus, it is likely that it has its own distinctive subset of terms. This conjecture, however, needs to be tested.

Regression. After assessing the correlations above, we focused on finding suitable regression models. Table II presents the candidate models we have considered both for TT and DT as functions of LOC. Based on R^2, the p-value and residual analysis, we concluded that for our dataset the power model is the best fit.

TABLE II
COMPARISON OF MODELS FOR *DT* AND *TT* AS FUNCTION OF LOC.

Predicted	Model	Model	R^2	*p-value*	residual
DT	Linear	$DT = a + b * LOC$	89.67%	$<$ 2.2e-16	invalid
	Exponential	$DT = a * b^{LOC}$	68.38%	8.77e-11	valid
	Power	$DT = a * LOC^b$	**88.39%**	$<$ **2.2e-16**	**valid**
TT	Linear	$TT = a + b * LOC$	97.32%	$<$ 2.2e-16	invalid
	Exponential	$TT = a * b^{LOC}$	68.75%	7.03e-11	valid
	Power	$TT = a * LOC^b$	**98.13%**	$<$ **2.2e-16**	**valid**

The final power regression models for the two variables are:

$$DT = \lfloor 4.759011602 * LOC^{0.50324} \rfloor \quad (1)$$

$$TT = \lfloor 0.650710784 * LOC^{0.97309} \rfloor \quad (2)$$

B. Modeling the characteristic frequency vector

We analyzed the characteristic frequency vectors of the 39 vocabularies and confirmed they all follow a long tail distribution [8], [9]. Figure 1 presents scatter plots with both axis in logarithmic scale for the rank of terms per frequency for some vocabularies in our sample. In fact, all other vocabularies follow the same pattern.

A graphical interpretation of the shape of the log-log scatter plots suggests that the distributions are long tailed. We used Vuong's test, as proposed by Clauset *et al.* [7], to detect whether a long tail of type log-normal best fits the data. The results with the Vuong test confirmed that 31 out of the 39 (79.5%) vocabularies have their characteristic frequencies modeled as log-normal distributions (30 of them) or as a pareto distribution (for one case). For 8 of the vocabularies, the Vuong's test did not derived any statistically valid conclusions.

We concluded that as a first approximation we can safely use a log-normal density function with a mean μ and a standard deviation σ can be used to model term frequencies in a vocabulary. By this model, the frequency of a *ith*-term in a characteristic frequency vector is given by:

$$F(term^{ith}) = \lfloor f(i, \mu, \sigma) * TT \rfloor \quad (3)$$

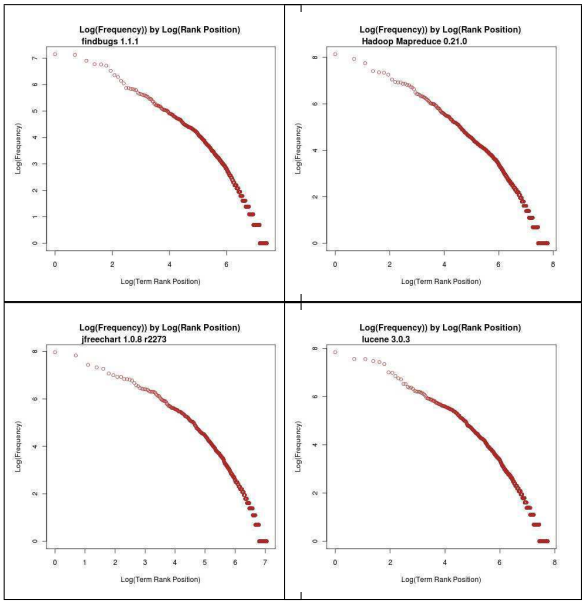


Fig. 1. Scatter plots in logarithmic scale for the rank of terms per frequency for: FindBugs, Hadoop MapReduce, JFreeChart and Lucene.

where: $f(i, \mu, \sigma)$ is a log-normal that computes the appearance probability of the *ith*-term; i is the position of the *ith*-term in the decreasing frequency rank, $i = 1^{st}, 2^{nd}, ..., DT^{th}$.

The estimators $\bar{x}$ for μ and s for σ can be computed taking the natural logarithm of the log-normal observed values (the frequencies of each term of a vocabulary) [10]. Over these estimators we evaluated several fits. The hyperbolic adjustment was the best to model both s and $\bar{x}$ as a function of LOC. Table III presents the models as well as their coefficient of determination R^2 and *p-value*.

TABLE III
HYPERBOLIC ADJUSTMENT FOR $\bar{x}$ AND s AS A FUNCTION OF LOC.

Hyperbolic Model	R^2	***p-value***
$s = \frac{LOC}{(5.849+0.543*LOC)}$	99.77%	$<$ 2.2e-16
$\bar{x} = \frac{LOC}{(-1.784+0.5576*LOC)}$	98.00%	$<$ 2.2e-16

The parameters, μ and σ in Equation 3, are respectively replaced by $\bar{x}$ and s, both modeled as a function of LOC.

The models presented here allow us to predict the size and the frequencies of terms of vocabularies. While we used them to synthesize vocabularies to produce data for an experimental settings, we believe they can be useful for other researchers in this area and for other purposes too. Combined, the models for the number of distinct terms and the one for characteristic frequency vectors allow us to artificially produce vocabularies that are software-realistic, when we take our sample as a reference.

V. RELATED WORK

Previous research has stressed the importance of software vocabulary to facilitate program maintenance. Abebe *et al.* [11] showed that identifiers are related to problem domain concepts. Haiduc and Marcus [12] concluded that identifiers are chosen in accordance with stakeholders' personal preferences and experience. According to Host and Ostvold [13], identifiers that do not reflect functional requirements affect software understanding. Butler *et al.* found that software understanding and maintenance are degraded when identifiers do not follow good convention rules [14]. Binkley and Lawrie, in [15] and [16] describe the use of IR on projetc's natural language to addressing software engineers problems.

There are some research trying to compute metrics based on systems vocabulary. Arnaoudova *et al.* [17] defined entropy and context coverage measures for vocabulary terms, and try to related them to fault proneness Eshkevari [18]. Lexicon measures was defined by Biggers *et al.* aiming to inform the best IR technique might be used for a particular software understanding activity [19].

Some research has also conducted exploratory statistical analysis on software vocabulary. Anslow and Tempero [20] extracted word frequencies from class identifiers of several Java API versions to follow the evolution. Power law regularities in computer programs was detected both by Linstead *et al.* [21] that grouped vocabularies of 12,151 open projects into a unique corpus, and by Zhang [22] that explored lexical tokens of 24 open source systems. Haouari *et. al.* [23] explored the quantitative distribution of comments over the program constructs.

In none of the aforementioned studies no association between vocabulary and source code size was investigated. Thus, a typical behavior of vocabulary might not be modeled as function of LOC size, as well as we did in ours. Furthermore, we applied Vuong statistic test that gave us accurate evidences to assert that log-normal distribution best fits term frequencies in software vocabulary.

VI. CONCLUSIONS AND FUTURE WORK

We achieved an initial model of software vocabulary. Being properly extended, this model will allow the generation of synthetic, realistic software vocabularies. In our case, these vocabularies are added to models that synthesize software-realistic design [1]. This fact will enable the creation of a benchmark for assessing architectural recovery algorithms that either use structural information, or vocabularies, or both. We also believe these vocabularies can be useful for other purposes for researchers in the area of text analysis.

Future research directions we expect to explore include i) studying the decomposition of the vocabulary over the design entities; ii) identifying how the software vocabularies evolve over time; and iii) investigating whether the models identified can be generalized at some extent or whether different models must be developed for different domains or somehow related software systems.

Success in these quests would enable the generation of tools capable of synthesising realistic software vocabularies, for size and occurrences of terms, that resembles to the vocabulary of real software systems.

ACKNOWLEDGMENT

We would like to thank SPLab/UFCG members for their contributions, and IFPE for licensing the first author. This work is sponsored both by CAPES and by ePol/SETEC/DPF.

REFERENCES

[1] R. Souza, D. Guerrero, and J. Figueiredo, "Modular Network Models for Class Dependencies in Software," *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, 2010.

[2] R. Bittencourt and D. Guerrero, "Comparison of Graph Clustering Algorithms for Recovering Software Architecture Module Views," *European Conference on Software re Maintenance and Reengineering*, pp. 251–254, 2009.

[3] C. John M., C. William S., B. Kleiner, and P. A. Tukey, *Graphical Methods For Data Analysis*. Duxbury Press, 1983.

[4] B. S. Everitt and I. Hothorn, *A Handbook of Statistical Analyses Using R*. Chapman & Hall/CRC, 2010.

[5] R. Jain, *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. Wiley New York, 1991.

[6] C. Wohlin, M. Höst, P. Runeson, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering: an introduction*. Kluwer Academic Pub, 2000.

[7] A. Clauset, C. Shalizi, and M. Newman, "Power-law distributions in empirical data," *SIAM*, vol. 51, no. 4, pp. 661–703, 2009.

[8] A. B. Downey, "Lognormal and Pareto distributions in the Internet," *Computer Communications*, vol. 28, no. 7, pp. 790–801, May 2005.

[9] D. G. Feitelson, *Workload Modeling for Computer Systems Performance Evaluation*, 2009.

[10] N. L. Johnson, S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions (volume 1)*, 2nd ed. Wiley-InterScience, 1994.

[11] S. L. Abebe, S. Haiduc, A. Marcus, P. Tonella, and G. Antoniol, "Analyzing the Evolution of the Source Code Vocabulary," *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, pp. 189–198, 2009.

[12] S. Haiduc and A. Marcus, "On the use of domain terms in source code," *The 16th IEEE International Conference on Program Comprehension*, vol. 0, pp. 113–122, 2008.

[13] E. W. Host and B. M. Ostvold, "The Programmer's Lexicon, Volume I: The Verbs," *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, vol. I, pp. 193–202, Sep. 2007.

[14] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Exploring the Influence of Identifier Names on Code Quality: an empirical study," *oro.open.ac.uk*, 2010.

[15] D. Binkley and D. Lawrie, "Information Retrieval Applications in Software Maintenance and Evolution," *Encyclopedia of Software Engineering*, pp. 1–29, 2009.

[16] ——, "Information Retrieval Applications in Software Development," *Encyclopedia of Software Engineering*, pp. 1–37, 2010.

[17] V. Arnaoudova, L. Eshkevari, R. Oliveto, Y. Guéhéneuc, and G. Antoniol, "Physical and conceptual identifier dispersion: Measures and relation to fault proneness," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–5.

[18] L. M. Eshkevari, "Linguistic Driven Refactoring of Source Code Identifiers," *2010 17th Working Conference on Reverse Engineering*, pp. 297–300, Oct. 2010.

[19] L. Biggers, B. Eddy, N. Kraft, and L. Etzkorn, "Toward a Metrics Suite for Source Code Lexicons," *Conference On Software Maintenance*, 2011.

[20] C. Anslow and E. Tempero, "Visualizing the Word Structure of Java Class Names," *Conference on Object Oriented Programming Systems Languages and Applications*, no. 133, pp. 777–778, 2008.

[21] E. Linstead, L. Hughes, C. Lopes, and P. Baldi, "Exploring Java software vocabulary: A search and mining perspective," in *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*. IEEE Computer Society, 2009, pp. 29–32.

[22] H. Zhang, "Discovering power laws in computer programs," *Information Processing & Management*, vol. 45, no. 4, pp. 477–483, 2009.

[23] D. Haouari and H. Sahraoui, "How Good is your Comment? A study of Comments in Java Programs," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2011.

Using Software Vocabulary to Rank Classes that are Probably Impacted by a Bug Report

Diego Cavalcanti, Katyusco Santos, Dalton Serey and Jorge Figueiredo
Software Practices Laboratory (SPLab)
Federal University of Campina Grande - Brazil
{*diegot,katyusco,dalton,abrantes*}*@computacao.ufcg.edu.br*

***Abstract*—Locating and fixing bugs described in bug reports are time consuming tasks that depend on developer's knowledge about the software architecture. Hence, the success of such tasks is directly depending on choosing the right developer. In our work, we perform an empirical study 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, the top-25% list of entities has about 90% of the target entities. We concluded that while vocabularies cannot be the sole source of information, they can certainly improve results if combined with other techniques.**

***Keywords*-bug location; software vocabulary; bug reports.**

I. Introduction

Bugs are often introduced in software systems. Consequently, locating and fixing bugs are routine tasks in software development. In fact, a large part of software development processes is spent on understanding, locating and fixing bugs. A previous study [1] has shown that developers usually spend more time looking for code entities (e.g. classes) than actually fixing them.

Typically, large projects organize bug information in bug reports – documents in which users and developers register pre-defined information about a problematic software system, and a free-form text that describes the encountered bugs. One common problem in relying on bug reports to locate the code entities that are responsible for the faulty code is that, due to the fact that bug reports can be written by non-developer users, there is no direct mapping from bug reports to code entities. This way, developers usually have to rely solely on the free-form text contained in a bug report in order to understand, locate, and fix a given bug.

For large projects with several modules, the task of locating code entities related to bugs has a significant cost [2], especially because determining the code entities where the bug is located does not scale as software grows.

There have been some attempts to cope with this problem by using past data extracted from code repository [3], [4], [5], [6], [7], [8]. However, these approaches cannot be used neither with new projects nor new artifacts since there are no past data about recently created pieces of code. For instance, bug reports related to new features might use a completely new vocabulary from the past projects' bug data set, which can compromise those approaches, once they rely on suggestions using historical data. So, an approach that is independent from the history of source code and bug repository is desired.

In our work, we conducted an empirical study to evaluate whether the automated analysis of bug reports and software vocabularies can be helpful in the task of locating bugs. We extract software and bug reports vocabularies and use Information Retrieval to rank the Java classes that are more likely to be impacted on fixing a given bug.

We analyzed eight versions of mature open-source projects (three Eclipse versions and five Apache projects) and found interesting results. For example, considering vocabulary similarity, a Top 25% ranking of classes contains about 90% of impacted classes. In other words, by analyzing only software vocabulary and bug reports description, a developer can reduce her search space in 75% and still has an error rate less than 10%.

In summary, our study has the following contributions:

1) we show that considering software vocabulary as a factor is helpful to narrow down the task of finding entities which are more likely to be impacted to fix a given bug report;
2) we concluded from evaluation performed in a sample of six mature Java projects, that the vocabulary analysis of source code and bug reports are good descriptors to find impacted classes;
3) we made available, in a structured format, the extracted software vocabulary data and bug data set of eight versions from six mature projects.

II. Bug reports

Bug repositories (also called Bug Tracking Systems) provide a database of problems reported about a software. The term *open bug repository* refers to systems in which

any user can login and save information about bugs found in software. There are a number of bug repositories used in open-source projects (e.g. Bugzilla, GNATS and JIRA).

Generally, bug reports from different bug tracking systems have similar structure. Each bug report is represented by a unique id and is commonly composed by some sections, such as pre-defined fields, free-form text and attachments [9].

Pre-defined fields store information about the faulty software (e.g. product, component, version and used platform), and about the bug report itself (e.g., status, priority, target milestone, developer assigned to solve it, keywords and timestamp). Reporters supply most data when the report is filled. The remaining are automatically generated or supplied by project manager.

The free-form text includes the title of report, a description and comments. The title is commonly a one-line summary of the bug. While the text is free-form, reporters are asked to provide the following information: a detailed description of the bug, steps to reproduce it and any other kind of information that can help developers to identify and solve the bug. The additional comments represent discussions about possible approaches to solve the bug and pointers to other bug reports that can contain more information about the problem or that appear to be duplicated.

Finally, attachments are allowed in order to add non-textual information (e.g. screenshot of error) to the bug report.

III. Empirical Study Design

As stated in Section I, in our work, we assessed whether comparing bug reports and source code vocabulary can be helpful for bug localization.

This section presents the projects that were evaluated, describes the data collection process and study setup, presents our data analysis, and shows how one can find the tools that were used in our study.

A. Projects

We analyzed eight open-source projects (3 versions of Eclipse and 5 Apache projects). All of them developed mainly using Java.

We chose by convenience projects that contain released versions in production, so that they would have a significant number of fixed bugs. Furthermore, the projects should had guidelines about fixing bugs and committing them, such as specifying on commit messages which bug is being fixed (e.g. "Fix LUCENE-1234"). It permits us to build an oracle which links bug reports and impacted classes from repository commits. That said, we chose Apache and Eclipse projects because we found that, besides being mature, they also have such guidelines.

Table I presents information about the versions of projects we used, such as: number of classes (excluding test classes), number of bug reports we evaluated, and the period of time the bug reports were marked as resolved in the bug repository.

B. Data Collection and Study Setup

In order to perform the study, we collected the source code of each project, and a set of fixed bug reports, each one mapped to the entities impacted for its solution. We considered only fixed bug reports that were related to the studied versions.

Since we evaluated only open-source projects, downloading source code was trivial – we only checked out the code of the desired versions from project's repository.

Likewise, most bug tracking systems allow users to filter and export their data set in a structured document format such as XML. However, mapping each bug report to impacted classes is not as easy, since the reports usually do not store this kind of information. Some authors have done this kind of mapping by mining software repositories [4], [5], [6]. They search for references to bug reports in commit messages (e.g., Fixed 42233 and bug #53784). We used this same approach to guide us on this task.

More specifically, in case of Eclipse versions, we used a bug data set provided by Schröter et al [6]. They extracted the data from mining repositories and made it available for other researches usage. On the other hand, with regard to Apache projects, we manually extracted the bug data set by filtering it in the JIRA environment, which is available through a link like: https://issues.apache.org/jira/browse/PROJECT-NAME. For example, considering the Apache Lucene project, we filtered it on https://issues.apache.org/jira/browse/LUCENE with the following query:

```
project = LUCENE AND issuetype = Bug AND
fixVersion = ''3.5'' AND resolution = Fixed
```

Finally, before extracting the vocabulary we removed from project all test classes and third-party libraries. It guarantees that we would analyze and rank only potential faulty classes.

C. Methodology

The study was conducted according to the five steps presented in Figure 1, as follows:

First of all (*Step 1*), we extract the vocabulary of source code and apply IR algorithms to treat terms. To extract the project's vocabulary we use the VocabularyExtractor tool[1]. More specifically, the tool processes all classes in a Java project, constructs their Abstract Syntax Tree (AST), and extracts vocabulary (names of classes, interfaces, methods, fields and constants) and classes' documentation (e.g. *javadoc*). Finally, the extracted data is stored in an XML-based file format that we call VXL and that maintains the identifiers hierarchy from code.

[1]Tool developed by our research group and available together with this study's data (Section III-E).

Table I
EVALUATED PROJECT VERSIONS

Project name	version	# of classes	# of bug reports	Period of bug reports
Apache Hadoop	0.23.0	535	27	Nov 2010 - Nov 2011
Apache Lucene	3.5.0	501	14	Sep 2011 - Dec 2011
Apache OpenJPA	2.1.0	1,226	90	Mar 2010 - Apr 2011
Apache Pivot	2.0.1	566	31	Mar 2011 - Dec 2011
Apache Shiro	1.2.0	350	16	Jan 2011 - Jan 2012
Eclipse	2.0	6,413	2,780	Dec 2001 - Dec 2002
Eclipse	2.1	7,566	2,480	Oct 2002 - Sep 2003
Eclipse	3.0	10,331	4,136	Dec 2003 - Dec 2004

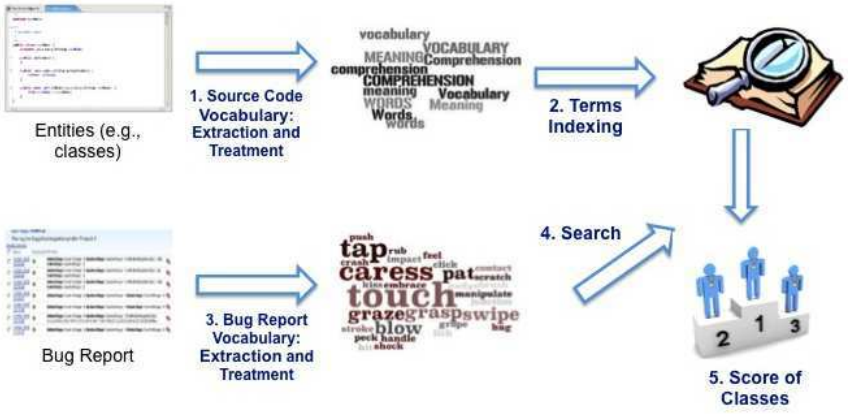


Figure 1. Our methodology

In *Step 2*, we index the terms from classes, using the search engine library Apache Lucene [2]. The index stores statistics about terms in order to make term-based search more efficient. Thus, indexing terms let us to efficiently search over the entities vocabulary.

Step 3 aims at extracting the vocabulary of bug report and treating it with IR techniques. However, we do not index bug report's vocabulary. Instead, we use it as a query to search over the code's entities the ones that contain the bug report's terms in their vocabularies (*Step 4*).

Finally, in *Step 5*, we score source code classes and return a ranking of them to the user. For that purpose, we use Vector Space Model (VSM) [14] of Information Retrieval to assess how similar each code entity is to the bug report vocabulary (the query). In VSM, documents (in our case, classes) and queries (bug report descriptions) are represented as weighted vectors in a multi-dimensional space, where each distinct index term is a dimension, and weights are TF-IDF values.

D. Research Questions

Our study aimed at clarifying the following research questions (RQ):

- RQ_1: **How many classes does each bug report impact?** This question is important to measure, in practice, how many classes programmers should figure out in order to fix a given bug. It can guide us to know the distribution of the impact set and, consequently, the expected size of a list of impacted classes.
- RQ_2: **What portion of the impacted set is below the top-N part of the ranking based on vocabulary similarity?** We have approached this question because a possible use of a vocabulary similarity ranking would be to discard all entities ranked below a given threshold. So it is interesting to know what portion would be lost.
- RQ_3: **What is the potential of vocabulary similarity to improve the state-of-the-art?** As far as we know, there is no baseline defined in the literature to evaluate the effort to find impacted classes by using such a ranking. So we decided to compare our results with a synthetic baseline.

E. Reproducibility

The tools and data used in the study can be downloaded at http://code.google.com/p/splab-br-analysis/.

IV. RESULTS

For the sake of brevity, we will not present the details of our results. However, in summary our results demonstrated that 1) in general, a developer has to identify for each bug report less than four classes to change among hundreds or thousands of classes from a project. It enforces that effective techniques to help to identify the start impacting

[2]http://lucene.apache.org/

set are needed (refer to RQ_1); 2) one can use vocabulary similarity to filter out most non-impacted classes, with only a small portion of the project. It is worth noticing that even though vocabulary is not effective to precisely point all impacted classes, it can greatly reduce the size of classes that a developer needs to analyze in order to fix a bug. For example, if one uses vocabulary similarity to retrieve a Top 25% of project's classes, she will get about 90% of impacted classes presented in ranking. That means we can indeed use vocabulary similarity to discard about 75% of non-impacted classes by each bug report (refer to RQ_2); and 3) further studies are needed to be developed in order to construct a baseline that can be used to compare results for future studies about software vocabulary applied to bug impact. The synthetic baseline we used in our studies is not representative enough for comparison of results (refer to RQ_3).

Even the fact that the rankings generated by our study present good results, they may not be the sole source of information to suggest impacted classes, since, most times, they contain several classes. It is even more evident to our subjects (Eclipse and Apache projects) since they have hundreds or thousands of classes. That means that even a portion of 1% of project classes corresponds to a ranking containing several classes that should be manually analyzed by a developer. Therefore, one may want to use other techniques (e.g. machine learning algorithms) that when combined with ours can be effective to achieve smaller sets.

However, even though it is not possible to solely use vocabulary to rank classes, it certainly can be used as a support to filter them. Combined with other techniques, it makes possible to create tools in order to help developers on locating classes that are more likely to be modified for a bug fix. Such a tool could be an IDE extension (e.g. Eclipse's plugin) that when a developer opens a class to edit it, she could easily visualize bug reports that are potentially related to that class. We guess that this tool could decrease the effort of localizing potentially faulty classes, what implies on an increase of the rate of bug report resolution.

Moreover, our approach is interesting in the sense that it does not need past historical data. Thus, it is independent of the maturity of project and the history of its repository. This is due to the usage of software vocabulary.

We know that researches about software vocabulary are still in their early days when comparing with other areas. However, they have already shown to be promising. Our study is an example that one can apply software vocabulary's similarity analysis in practice. Nevertheless, we hope that, as soon as the knowledge about software vocabulary expands, our work can be expanded as well, approaching an ideal solution to the improvement of state-of-practice. Besides that, this study must be continued together with other researches that aim at improving quality of bug reports, especially quality of bug descriptions.

Finally, we are sure that our study can be useful as an initial step to researches that use software vocabulary as a factor to assess solutions for the problem of bug localization. In doing so, we hope to help to decrease the expensive process of software maintenance.

V. Conclusion

In our study, we evaluated whether the automated analysis of software and bug reports vocabularies is helpful on locating bugs.

We conducted the analysis on three versions of Eclipse and five Apache projects. All of them written mainly in Java and still in development. At total, we automatically analyzed 8,022 bug reports and related them to their impacted classes.

In summary, we extracted software and bug reports vocabularies, treated them using information retrieval techniques, and analyzed their similarity. As output, we retrieved a list of classes ranked by their similarity with bug reports.

The results showed us that each bug report impacts only a few number of classes (mostly, less than four) among hundreds or thousands of them. It means that developers need to make a great effort, or use proper techniques, to analyze several classes to find out only few ones to change.

Moreover, we could see that vocabulary similarity analysis is indeed helpful to filter out classes that are not impacted by a bug report. For instance, a ranking with the Top 15% of classes, contains from 80% to 95% of all classes that were impacted by a bug report. In other words, by using only vocabulary similarity, one can filter out 85% of software classes and still have about 90% of the impacted classes.

Besides the good results we had, we also defined a synthetic baseline to compare our approach with. In most cases, our results were better than that baseline by retrieving more impacted classes in its ranking. So, the comparison showed us that vocabulary is a potential candidate to improve the state-of-art in the area of bug localization. However, we still need to find a more representative baseline to compare our results with.

Finally, this study enforces that software vocabulary analysis is a promising area and also can be used to improve the state-of-practice. In doing so, we intend to continue working on it, by combining our approach to other techniques, in order to improve the retrieved ranking. Then, when we are able to obtain good results with a small ranking, we will be ready to develop a tool to help developers on bug fixing in a practical way.

Acknowledgment

The authors appreciate the valuable discussions with members of Evolution Group from Software Practices Laboratory/UFCG. This research was supported by CAPES (Brazilian's Coordination for the Improvement of Higher Level Personnel).

REFERENCES

[1] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for ides," in *Proceedings of the AOSD '05*, 2005.

[2] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Transactions on Software Engineering*, 2003.

[3] G. Canfora and L. Cerulo, "Impact analysis by mining software and change request repositories," *IEEE International Symposium on Software Metrics*, 2005.

[4] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of MSR '05*, 2005.

[5] M. Fischer, M. Pinzger, and H. Gall, "Analyzing and relating bug report data for feature tracking," in *Proceedings of WCRE '03*, 2003.

[6] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering (Promise 2007)*, 2007.

[7] A. Nguyen, T. Nguyen, J. Al-Kofahi, H. Nguyen, and T. Nguyen, "A topic-based approach for narrowing the search space of buggy files from a bug report," in *Proceedings of ASE '09*, 2011.

[8] J. Zhou, Z. H., and L. D., "Where Should the Bugs Be Fixed?" in *Proceedings of ICSE '12*, 2012.

[9] J. Anvik, L. Hiew, and G. Murphy, "Who should fix this bug?" in *Proceedings of ICSE '06*, 2006.

[10] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge, UK: Cambridge University Press Cambridge, 2008.

[11] M. Porter *et al.*, "An algorithm for suffix stripping," 1980.

[12] S. Haiduc and A. Marcus, "On the use of domain terms in source code," in *Proceedings of ICPC '08*, 2008.

[13] S. L. Abebe, S. Haiduc, A. Marcus, P. Tonella, and G. Antoniol, "Analyzing the evolution of the source code vocabulary," in *Proceedings of CSMR '09*, 2009.

[14] G. Salton, A. Wong, and C. Yang, "A vector space model for information retrieval," *Journal of the American Society for information Science*, vol. 18, no. 11, 1975.

[15] G. Canfora and L. Cerulo, "Fine grained indexing of software repositories to support impact analysis," in *Proceedings of MSR '06*, 2006.

[16] A. Moin and M. Khansari, "Bug localization using revision log analysis and open bug repository text categorization," in *Open Source Software: New Horizons*, 2010.

[17] A. J. Ko, B. A. Myers, and D. H. Chau, "A linguistic analysis of how people describe software problems," *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2006.

[18] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," in *Proceedings of MSR '08*, 2008.

[19] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers," in *Proceedings of MSR '09*, 2009.

[20] D. Čubranić and G. Murphy, "Hipikat: Recommending pertinent software development artifacts," in *Proceedings of ICSE '03*, 2003.

[21] N. Bettenburg, S. Just, A. Schröter, C. Weiß, R. Premraj, and T. Zimmermann, "Quality of bug reports in Eclipse," in *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, 2007.

Apêndice D

Artigo publicado no CBSoft2013: Teoria e Prática

CBSoft2013 (Conferência Brasileira de Software: Teoria e Prática) em Setembro de 2013, Brasília, Brasil.

TopicViewer: Evaluating Remodularizations Using Semantic Clustering

Gustavo Jansen de S. Santos[1], Katyusco de F. Santos[2], Marco Tulio Valente[1], Dalton D. S. Guerrero[3], Nicolas Anquetil[4]

[1]Federal University of Minas Gerais

[2]Federal Institute of Pernambuco

[3]Federal University of Campina Grande

[4]RMoD Team, INRIA

gustavojss@dcc.ufmg.br, katyusco@recife.ifpe.edu.br, mtov@dcc.ufmg.br,

dalton@dsc.ufcg.edu.br, nicolas.anquetil@inria.fr

Abstract. *Software visualization techniques have been proposed to improve program comprehension, as large systems get difficult to understand and maintain. In this paper, we describe the design and main features of* `TopicViewer`*, a tool that uses Information Retrieval techniques and Hierarchical Clustering to show how domain concepts are disposed across one system's architecture as it evolves. Also, we describe an application of this tool during the remodularization of* `JHotDraw`*.*

1. Introduction

As a software system gets larger and more complex, required modifications and improvements get more difficult to implement. It is estimated that up to 60% of software maintenance efforts is spent on understanding the code to be modified [Abran *et al.* 2001] . Most of this knowledge is often spread in external documentation, like task reports.

Many approaches have been proposed to make program comprehension more effective. In general, the main goal is to extract a mental, high-level model of the software, based on documents or by the reverse engineering the source code. On the other hand, to help the interpretation of many documents described in natural language, Information Retrieval (IR) techniques can be used as support to search and acquire similar documents, given a set of information queries.

In order to assist program comprehension tasks, this paper presents `TopicViewer`, a tool that supports a methodology proposed by Kuhn *et al.* [2007] . It relies on IR and Hierarchical Clustering to extract groups of similar documents and retrieve a set of frequent words from these groups, which represent domain concepts in the software. We performed an adaptation of this methodology to make possible its application in any sources of textual information, like bug reports or high-level documentation, and also to support the analysis of how domain concepts evolve after a remodularization.

The remainder of this paper is structured as follows. In Section 2 we present the design and implementation of the `TopicViewer` tool, including the original methodology and our experience with `JHotDraw`'s source code. Section 3 discusses related tools and Section 4 presents our final remarks.

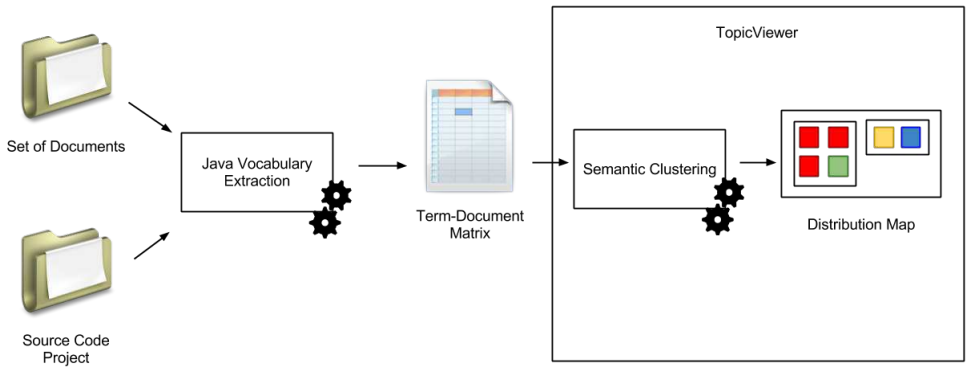


Figure 1. The `TopicViewer` engine

2. The `TopicViewer` Tool

`TopicViewer` is a tool that provide visualizations in terms of clusters of classes that represent common semantic concerns. Basically, classes that manipulate the same concerns (e.g. data structures, file management) are tipically grouped in the same cluster.

For this purpose, `TopicViewer` requires a representation of the set of documents as a term-document matrix, which each cell `(t,d)` contains the relevance of term `t` in the document `d`. In this work, a software entity is a container of inner entities or identifiers, like packages, classes, files or methods. Moreover, although we focus on source code text extraction, our approach can be applied to bug reports and other textual documents. This matrix must be preprocessed, as the text extraction fully depends on the document's structure. However, an interface of this matrix is provided in order to handle systems with large number of terms and documents.

As presented in Figure 1, the result of `TopicViewer` process is a Distribution Map [Ducasse *et al.* 2006]: a visualization in which documents are organized in folders and packages, and colored according to the group of similar classes to which they belong. The organization of these documents is important in the visualization, in order to analyze the distribution of domain concepts over time.

We present the original methodology in Subsection 2.1. Then, we introduce our adaptations to assess software evolution (Subsection 2.2). Next, we present the design and implementation of `TopicViewer`, and finally a case study using source code extraction and analysis of a `JHotDraw` remodularization (Subsections 2.3 and 2.4).

2.1. Semantic Clustering

Semantic Clustering is a technique proposed by Kuhn *et al.* to group similar classes of a system, according to their vocabularies [2007]. According to the technique, every class is represented as a document, and terms are obtained by extracting and filtering identifiers and comments. Moreover, terms are later weighted with *tf-idf* function, in order to punish words that appear in many documents of the vocabulary. The resulting term-document matrix is then processed by Latent Semantic Indexing (LSI), an IR technique that reduces the matrix in number of terms with minimal loss of information [Baeza-Yates and Ribeiro-Neto 2011].

With the reduced matrix, each class is retrieved as a vector, and the similarity of a pair of classes is calculated by the cosine of the smallest angle formed by their representing vectors. Next, the technique relies on an agglomerative Hierarchical Clustering to group a fixed number of clusters or until no pair of clusters has similarity greater than a given threshold. Finally, queries are used to retrieve a set of words representing the meaning of each group (called semantic cluster) and the Distribution Map is generated.

Ducasse *et al.* also proposed two metrics to measure concentration and spreading of a semantic cluster [2006]. The *Spread* of a cluster is calculated as the number of parts (e.g. folders or packages) covered by the cluster. Moreover, *Focus* measures how a cluster dominates, i.e., covers most of the classes of the parts in which they appear. We also rely on an *Internal Cohesion* metric to assess the quality of a package as the mean similarity between their classes, pairwise [Marcus and Poshyvanyk 2005]. In contrast to structural metrics that consider static dependencies between software artifacts, these metrics express a conceptual relationship between these artifacts, as denoted by common terms in their vocabularies.

2.2. Remodularization Support

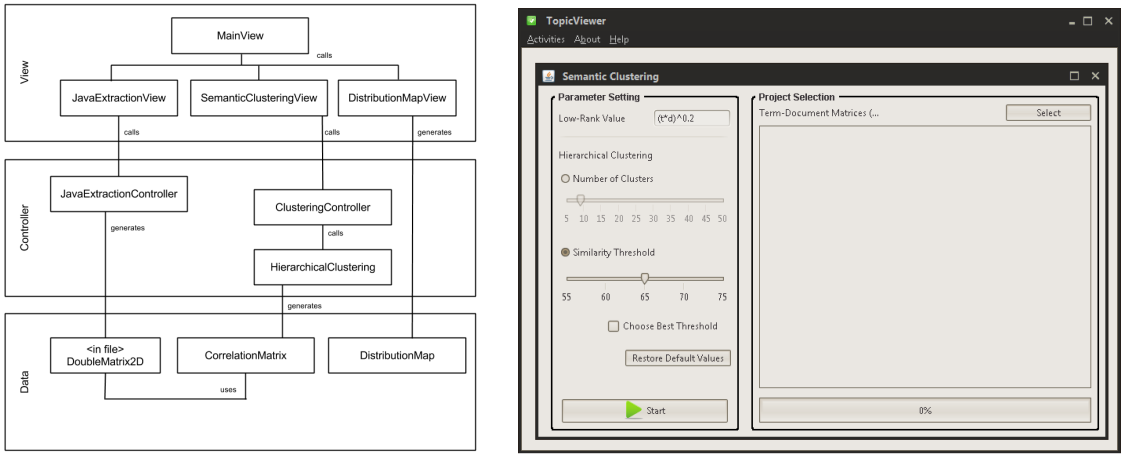
The original Semantic Clustering technique supports the extraction of clusters for a single version of a given system. Since we also want to evaluate how the semantic clusters evolve over time, we adapted this methodology to use two versions of the same system. For this purpose, we calculate the semantic clusters for the first version, and each cluster is represented by a vector that represents the sum of all vectors of this cluster's documents. Then, for the next version, each class vector is mapped to the previous version vocabulary, since their set of terms may differ. Finally, we calculate the cosine similarity between this mapped class and all cluster vectors from the first version, and we assign the class to the cluster with the highest similarity.

The result is a representation of two Distribution Maps. New packages or classes are displayed in both versions, but as empty entities in the first one. Similarly, removed packages or classes are displayed as empty entities in the second version. It is important to observe how the disposition of domain concepts evolves between versions. In other words, we expect that the classes of a package are more internally similar after maintenance tasks, and so the concentration of domain concepts.

2.3. Internal Architecture and Interface

`TopicViewer` is a desktop application implemented in Java. To use the tool, developers must follow four steps: (i) *Set Workspace*, i.e., a folder in which all output data is stored; (ii) *Java Vocabulary Extraction*, to extract vocabulary from Java source code, (iii) *Semantic Clustering*, in order to generate semantic clusters and Distribution Map given a term-document matrix, and (iv) *Distribution Map Viewer*, to interact with the map structure described in Subsection 2.1.

Our tool reuses IR functions from `VocabularyTools` [Santos *et al.* 2012], a set of tools for extraction, filtering and LSI operations in Java, developed by the Software Practices Lab of Federal University of Campina Grande (UFCG). The current `TopicViewer` implementation follows a Model-View-Controller architectural pattern with three main modules, as presented in Figure 2a:



(a) Architecture (b) User Interface

Figure 2. `TopicViewer` Architecture and User Interface

- *Data*: Representation of the main data structures of the tool. It includes matrix storage and operations in file, representation of a Distribution Map and utility functions to save these data structures and other information in files.
- *Controller* contains the main algorithms: the agglomerative Hierarchical Clustering, semantic topics extraction, as well as an interface for `VocabularyTools` operations.
- *View*: A set of Swing windows providing an user interface.

The interface for Semantic Clustering is shown in Figure 2b. The developer can choose (in the left panel) which clustering stop criteria to use, as a fixed number of clusters or by a similarity threshold. Next, the developer can choose a set of term-document matrices to cluster (right panel) and monitor the execution progress on the bottom bar.

2.4. Application

In this section we detail our experience with `TopicViewer` in evaluating a remodularization of `JHotDraw`, a framework for technical drawing editors[1]. We focused on the remodularization from version 7.3.1 to 7.4.1 in this paper. We used the `TopicViewer` tool to extract the semantic clusters and to generate the comparative Distribution Map, as displayed in Figure 3.

We observed that a lot of packages were created in this remodularization, as they are empty in the first map. Most of them have their internal classes very similar, and the classes are mapped to a common cluster. To assist our analysis, we provided an interaction with *Distribution Map View* so the user can visualize quality metric values for both versions (see Section 2.1), as comparison means to assess maintenance quality.

In this remodularization, the packages *org.jhotdraw.app.action* and *org.jhotdraw.draw* have been split up into sub-packages, resulting in the creation of 16 packages. For example, the *app.action* package had internal cohesion of 0.89 in the 7.3.1 version; after the remodularization, it had 0.94 and the average cohesion of the

[1] http://www.jhotdraw.org/

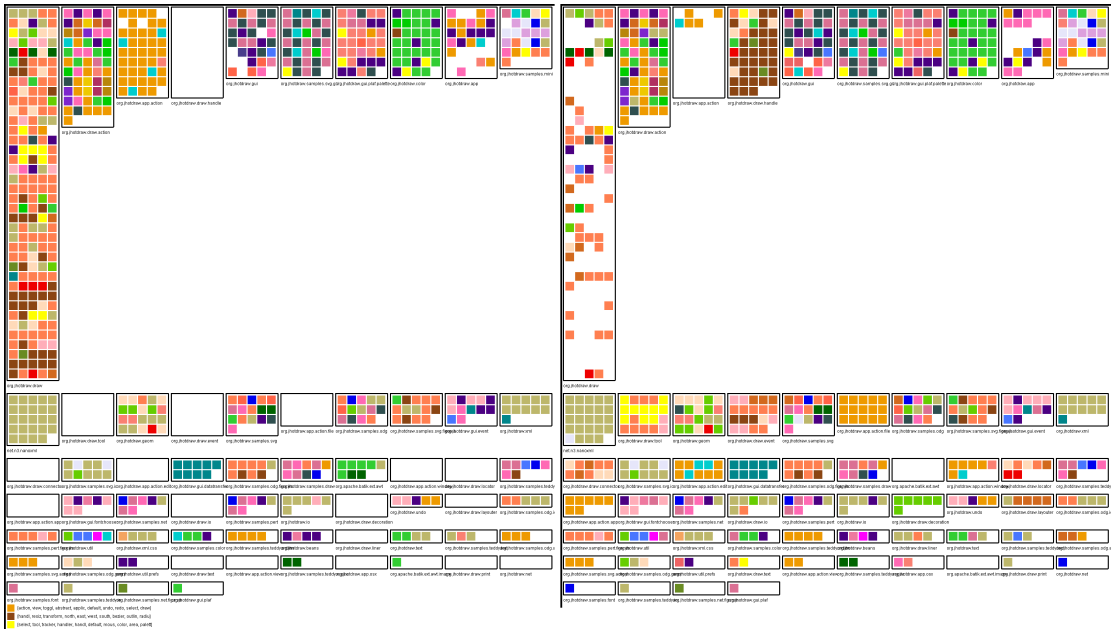


Figure 3. Comparative Distribution Map for `JHotDraw-7.3.1 - 7.4.1` Remodularization. A colored version of this image with better resolution is available at http://tinyurl.com/c4efph8

splitted packages is 0,98. It means that, internally, their classes are more cohesive in their respective new packages. Moreover, analyzing the semantic topic that dominated that package (marked in orange), both metrics Spread and Focus increased, which means that this cluster covers a greater number of packages, and the concern attached to it dominates the new packages after the remodularization effort.

As the latter refactored package, *draw*, its internal cohesion decreased a from 0.58 to 0.57. Nevertheless, the average cohesion of the eleven created packages from this refactoring is 0,70, which means that their classes are more cohesive. For the semantic clusters, both values of Spread and Focus increased.

For the semantic clusters, the same happens for the most dominating clusters, marked in yellow and dark chocolate, in which both Spread and Focus values increased. In general, we observed that semantic clusters usually share the same behavior in this case study: they are spread in a higher number of packages, and also dominate more classes in each package, increasing their Focus.

3. Related Tools

`TopicXP` is an Eclipse plugin that uses Latent Dirichlet Allocation and term extraction from identifiers and comments in source code [Savage *et al.* 2010]. The semantic topics can be shown in different views in which the user can inspect the relationship between them and excerpts in the code. `CodeTopics` is also an Eclipse plugin that uses information from high-level artifacts, i.e., textual documents provided by developers, and shows a mapping between these artifacts and the source code [Gethers *et al.* 2011].

At some extent, `Hapax` is the most similar tool to `TopicViewer`, and it was developed as a plugin for the `Moose` platform [Kuhn *et al.* 2007]. It performs text extrac-

tion from identifiers and comments, and returns a Distribution Map as result. Nonetheless, `TopicViewer` provides a comparative view to assist software evolution assessment, under a semantic point of view.

4. Final Remarks

Program comprehension often requires retrieving a set of external documents, most of them in natural language. Usually, it is difficult to match the semantics of these documents and the intention (or meaning) of the source code. Various approaches combine Information Retrieval and Software Visualization techniques to overcome these difficulties, as a means to assist the understanding and assess the conceptual quality of a code.

We developed `TopicViewer`, a tool that uses Latent Semantic Indexing to extract groups of similar classes, according to their vocabularies. We applied our tool to the remodularization of a real system, and we could highlight the changes, as well as assess this remodularization in terms of concept concentration and conceptual internal quality of the refactored packages.

`TopicViewer` is publicly available at http://code.google.com/p/topic-viewer/.

Acknowledgments:

Our research is supported by CAPES, FAPEMIG, and CNPq.

References

Abran, A., Bourque, P., Dupuis, R., and Moore, J. W., editors (2001). *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, Piscataway, NJ, USA.

Baeza-Yates, R. A. and Ribeiro-Neto, B. A. (2011). *Modern Information Retrieval - The concepts and technology behind search, Second edition*. Pearson Education Ltd., Harlow, England.

Ducasse, S., Gîrba, T., and Kuhn, A. (2006). Distribution map. In *22nd IEEE International Conference on Software Maintenance*, pages 203–212.

Gethers, M., Savage, T., Di Penta, M., Oliveto, R., Poshyvanyk, D., and De Lucia, A. (2011). Codetopics: which topic am I coding now? In *33rd International Conference on Software Engineering*, pages 1034–1036.

Kuhn, A., Ducasse, S., and Gîrba, T. (2007). Semantic clustering: Identifying topics in source code. *Information & Software Technology*, 49(3):230–243.

Marcus, A. and Poshyvanyk, D. (2005). The conceptual cohesion of classes. In *21st IEEE International Conference on Software Maintenance*, pages 133–142.

Santos, K. D. F., Guerrero, D. D. S., Figueiredo, J., and Bittencourt, R. A. (2012). Towards a prediction model for source code vocabulary. In *1st International Workshop on the Next Five Years of Text Analysis in Software Maintenance, International Conference on Software Maintenance*.

Savage, T., Dit, B., Gethers, M., and Poshyvanyk, D. (2010). TopicXP: Exploring topics in source code using latent dirichlet allocation. In *26th IEEE International Conference on Software Maintenance*, pages 1–6.

Apêndice E

Artigo completo aceito para publicação no ITNG2015

ITNG (International Conference on Information Technology : New Generations) em Abril de 2015, Las Vegas, USA.

Using Developers Contributions on Software Vocabularies to Identify Experts

Katyusco de F. Santos
Federal Institute of Paraíba
Campus Campina Grande
Academic Dept. of Information Technology
Email: katyusco.santos@ifpb.edu.br

Dalton D. S. Guerrero
and Jorge C. A. de Figueiredo
Federal University of Campina Grande
Department of Systems and Computer
Email:{dalton, abrantes}@dsc.ufcg.edu.br

***Abstract*—Developers choose identifiers to name entities during software coding. While these names are lexically restricted by the language, they reflect the understanding of the developer on the requirements that the entity is devoted for. In this paper, we analyze the use of such vocabularies to identify experts on code entities. For a real software development, e-Pol (Management Information System for Federal Police of Brazil), we observed around 30% of its code entities has more than 0.3 of similarity with at least one developer vocabulary.**

We propose an approach to catch this potential expertise that vocabularies carries on. Also, we built an oracle of source code entities per developer that allowed us to assess our approach accuracy compared with two others ones: based on commit and based on percentage of modified *Lines of Codes*. One advantage of our approach is to disregard changes in formatting or indentation of source code as acts of expertise acquisition.

We achieve an accuracy ranging from 0.16 to 0.32 depending on the assumed period of developers' contributions and the *top-k* experts we are interested on. These results confirm similarity between vocabularies might be explored to point out code experts. Moreover, for orphaned entities, expertise approach based on vocabularies can recommend among current team members one whose vocabulary is closest to the entity.

***Index Terms*—software vocabulary, expertise, oracle of experts**

I. Introduction

A constant challenge in software development is to support team leaders to find efficiently (quick and accurate) points in the source code where developers should perform maintenance tasks. But, for a task be implemented with minimal effort (cost and time) it is also necessary to define who among the members of a team is the most appropriate developer to do it.

Maintenance costs represent between 70% and 90% of a software project [1], [2]. During maintenance tasks, code understanding takes 50% [1] to 78% of programmer time, while 20% is to fix bugs and write code just 2%, [3]. Identify the right members of a development team that are experts for each of entities project aims to source code comprehension that is an activity that not only depends on code properties, but also on person experience [4]. Also, code snippets created by a developer can be modified by another one. The contributions, made changes, can exchange the code snippet expert from the original author to the largest contribute developer [5].

The most used approaches to identify experts are based on mining logs records from Version Control System (VCS). For instance, extracting the number of commits [6], [7], or computing percentage of modified *Lines of Code* (*%LoC*) [8], [9]. However, these known approaches endorse it is necessary to add other different aspects they already use to improve their accuracy.

Studies about identifiers and comments used to code software systems, show that they capture different aspects of those who are traditionally captured by static analysis (based on structural relationships) and by dynamic ones (to trace system executions) [10]. Identifiers in part, are named according to concepts, abstractions and representing features of a software system [11], and when summed to comments they represent about 70% of the source code [12]. Furthermore, they can be used to assess code quality [13] and reflect mental map of developers about a project [14]. Source code vocabulary can be used to narrow down the bug-locating task according the results we obtained an empirical experiments we have collaborated [15].

In summary, studies on software vocabularies report they are a valuable source of information about the developers and the system itself, however in the best we know, they have not been explored to model code expertise. Besides, both before performing changes as during new code implementation developers act on vocabulary of entities. For these reasons we question ourselves whether the vocabulary used by developers to name identifiers and to write comment and javadocs texts catch project expertise able to point out entities experts.

To evolve whether developers and source vocabularies can be used to identify experts, first we must be sure they are related. This way we define our first Research Questions (***RQ-1***): *Is there similarity between source code of entities and developers vocabularies?*

Since there is similarity between them, is worth additional effort to measure the accuracy of this relationship. So, we define the second Research Question, (***RQ-2***): *Is similarity can be used to identify source code entities experts?*

In this paper we propose a novel approach to identify experts of Java source entities based on similarity between developers vocabulary and system software vocabulary. We also present the approach accuracy, in terms of precision and recall, compared with two others baselines ones: based on commit and based on *%LoC*.

II. Background

A. Software Vocabulary

Program instructions are written according to strict formal grammar rules to avoid ambiguity, and part of them are formed by reserved words and operators of a language [16].

Developers are the authors of characters sequences, called identifiers, which are used to define, to reference and to manipulate both basic structural elements (*e.g.*: attributes, local variables, methods and parameters names), and more complex ones. In turn, comments are texts also written by developers using natural language with the goal to document explicitly source code elements. Naming structural elements and defining comments contents occur during programming process, and although they are limited to compiler restrictions, they are not tied to the formal dimension of grammar rules [16]. So, developers make use of those mechanisms to transfer their own ideas to and perceptions about software projects

In previous studies [17] we have developed a formal definition of vocabularies to aid the understanding and communication more effective and less ambiguous. Here we briefly present our formalization of vocabularies.

1) Vocabulary definition: We define a software vocabulary as a multiset of strings, i.e. an application $V : \mathbb{S} \to \mathbb{N}$ that maps strings to natural numbers. Elements of a vocabulary are called *terms*. For any term t, $V(t)$ denotes the number of occurrences of the term t in the vocabulary V. If $V(t) > 0$ we say that t is a term of the vocabulary.

As an example, consider the excerpt of java code in Listing 1. According to our definition, the vocabulary can be expressed as the following multiset of terms:

$$V = 4'SampleClass + 2'sa + 2'sc + 1'me + 1'mt$$

Listing 1. Excerpt of Code

```
public static int mt(){
    SampleClass sa = new SampleClass();
    SampleClass sc = new SampleClass();
    sa.me(sc);
}
```

Observe we can adopt existing multiset notations for vocabularies. Above, we have used formal sums, in which each sum term expresses the number of occurrences n of one vocabulary term t in the form $n's$. Other notations can be convenient for other purposes, as well.

2) Vocabulary Properties: For our purposes of expertise identification, size of vocabularies are relevant. For that reason, we have defined two metrics. The first is the total number of terms of a vocabulary, $TT(V)$, i.e. the sum of the multiplicities of all terms in the vocabulary. The second one is the number of distinct terms, $DT(V)$, which is the number of terms whose multiplicity is at least one. For our example above, V, we have

$$TT(V) = 10 \quad and \quad DT(V) = 5$$

3) Vocabulary operations: In practice, one typically needs to process vocabularies prior to performing actual analysis. Typical information retrieval (IR) processing operations, as well as other vocabulary operations, can be easily and unambiguously specified as functions over vocabularies. Take, as an example, a tokenization operation that splits terms according to a given formation rule (camel case, for instance). This operation can easily be specified as a function[1] $cc : \mathbb{V} \to \mathbb{V}$, that maps each pair $(t, m) \in V$ to a set of terms $\{(t_1, m_1), \ldots, (t_n, m_n)\}$, where each term t_i is one of the words that composes the original term t and m_i its respective number of occurrences. For instance, the vocabulary V_1 above maps, by such operation, to

$$cc(V) = 4'Sample + 4'Class + 2'sa + 2'sc + 1'me + 1'mt$$

4) Vocabulary of Source Code: An **entity code** is a complex static structure that encapsulates others one (basic or not). In this study, we have considered entities just classes and interfaces. Associated to an entity there are the identifier that naming it, its own comments and javadocs, as well the identifiers, comments and javadocs of its inner structures. Then, an entity is a vocabulary container of code snippet who is hierarchically on the entity's scope.

The vocabularies union of each entitiy that makes part of a system defines the vocabulary of this software system. We have modeled a software vocabulary as a Term-Entity matrix, TE, where each cell (T_i, E_j) is term frequency of t_i in entity E_j. In formal sums, vocabulary of E_1 is exemplified in matrix TE above, Table I, and is given by: $V(E_1) = 2't_1 + 1't_2 + 1't_3$.

TABLE I
Term-Entity, *TE*, example matrix.

Terms	Entities				
	E_1	E_2	E_3	E_4	E_5
t_1	2	0	1	0	0
t_2	1	2	0	0	0
t_3	1	0	1	0	1
t_4	0	0	1	1	0
t_5	0	0	0	0	1

5) Vocabulary of Developers: The vocabulary of a developer, D, is a result of the developer contributions on source code vocabulary [18]. Each contribution is captured by difference (*diff*) between a previous vocabulary of a given commit and a subsequent one. Like source code vocabulary, Term-Entity matrix represents a contribution. Each new contribution is accumulated to the previous one. Thus, the vocabulary of a developer is a result matrix, TE^D, that means the accumulation of all contributions of developer D.

B. Expertise

Expertise is defined as the ability of the developer to be an expert on given source code, and, if interpreted quantitatively reflects ability degree of developers have to perform a encoding task [6]. Expertise is difficult to measure directly because depends on how it affects development process and software product. The literature then studying expertise indirectly through observation of a variety of process and product measures.

In this study we are interested on assessing the usage of software vocabulary for experts identification. For this, we

[1] We denote the set of all vocabularies by $\mathbb{V}$.

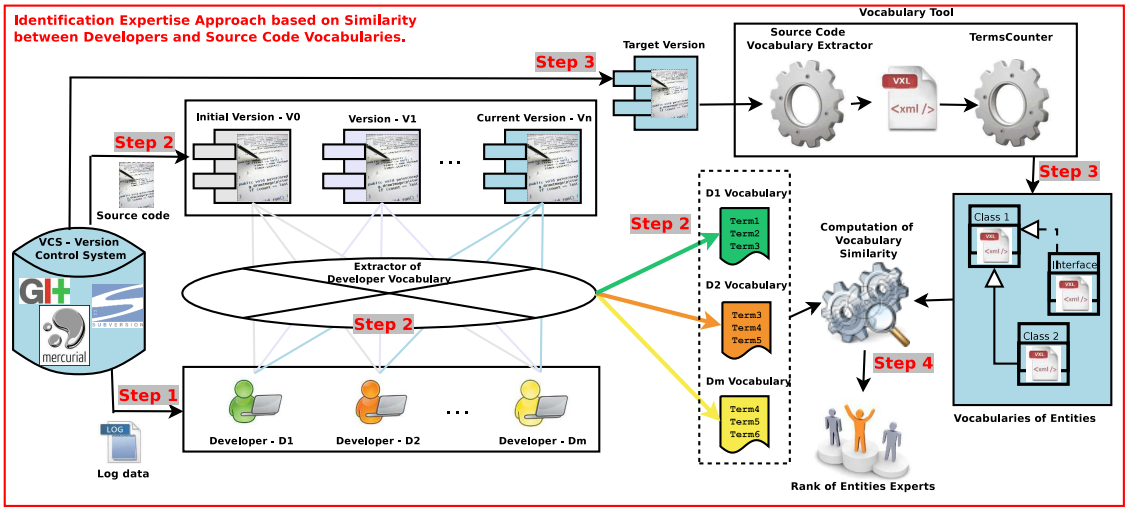


Fig. 1. Identification Expertise Approach Based on Similarity Vocabularies.

compare it with two most used approaches to identify experts. The first one is based on mining logs from *Version Control System* (*VCS*) to compute the number of commits performed by each member of development team on source code files [6], [7]. The second one also scans *VCS* logs, but defines an entity specialist according the percentage of *Lines of Code* (*%LoC*) modified by each team member [8], [9].

III. Our Approach - Based on Vocabulary Similarity

It is common for code snippets created by an author to be modified (code contributions or changes) by other developers, during projects life-cycle [5]. The vocabulary surrounding these code first must be read and understood by a developer responsible for. A change may also imply transformations on vocabulary: since an adding of term which different piece of code already has, to a replacement of all occurrences of a given term to a new one. An example for the latter is a full renaming refactory of a identifier that happens according concepts who performs the change.

Each developer has its own vocabulary that is accumulation of terms manipulated (added, removed or replaced) by a developer during its contributions on project. Each source entity also has its own vocabulary, but it differs from developers vocabularies. Because, in simplified way, a developer who will perform a change requires to understand an entity just like it is at specific target version, and not in versions before.

In the best of our knowledge the supposed similarity relationship between developers and source code vocabularies has not yet been investigated focusing on experts identification. Thus, we propose a strategy to explore this issue. Figure 1, diagrams four steps that we considered necessary to compute similarity between vocabularies, as follows:

1) extracts from *VCS* logs, all developers ID (identifiers) who have contributed to source project;
2) scans *VCS* commits extracting developers contributions. Each contribution is accumulated to its respective developer vocabulary as describe in Section II-A5;
3) extracts vocabulary of source code grained by entity for a target version[2];
4) computes the similarity among vocabularies of all target entities with vocabularies of all developers.

Whoever has the highest similarity value is considered the developer specialist of code entity.

IV. Experimental Design

A. Case Study Design

Trying to answer *RQ-1* and eventually *RQ-2,* we propose a case study that was divided into the following steps: 1) selecting Java project and sampling entities; 2) building oracle of entities experts; and 3) measuring expertise.

1) Selecting Java Project and Sampling Entities: As software projects evolve, so can change their specialists. Knowledge about expert of code spreads among developers. Factors captured by current expertise approaches are not enough to define best suited developer to perform a given task coding [19]. Beyond having access to source code, we must conduct meetings and interviews with stakeholders to construct oracles that reliably reflect who are the experts of a software project. Depending on team size and how its members are geographically distributed, building of oracle is an unaffordable cost. For this, we perform a case study on ePol[3] project. The ePol version we had access was 0.5b released on February 18, 2014. On that time developer team consisted of twelve programmers and one project leader. In terms of size, ePol has 907 Java entities (classes and interfaces) in more than 96*KLOC*, and its software vocabulary is comprised of 2751 distinct terms.

[2]source code on which a task coding will be performed.

[3]ePol is the Management Information System of Brazil Federal Police. It is under development at Software Practices Laboratory (SPLab) of Federal University of Campina Grande, Brazil.

Because we are interested on the best suited developer to realize tasks maintenance on system functionalities, we discard 436 entities whose role in ePol code is to test the other remaining 471.

2) Building Oracle of Entities Experts: It is not economically feasible to allocate a development team to answer questionnaires, discuss consensus in meetings, for a population of 471 individuals, *i.e.* entities. Therefore, we have random extracted a sample comprised of 50 entities ($n = 50$) representing more than 10% of population. Although sampling is random it follows the same entities distribution in the population according to percentiles 33º, 66ºand 100ºof three entities metrics: FanIn, LOC and Cyclomatic Complexity. It makes sample more representative. Combining each percentile for each metric, resulting in 27 ($3^{3metric}$) groups of entities. Then, each of the 471 entities belongs to one of these groups. The distribution function of ePol entities is the relative frequency of these groups for population of entities ($m = 471$).

A group G_g is defined as a tuple of entities metrics $<F_f, L_l, CC_c>$, where $f, l, c \in \{33º, 66º, 100º\}$ percentiles. The distribution function density, *dfd*, is given by:

$$dfd(G_g) = \frac{\#\{E_e\}}{m}$$

where $E_{e=\{1,...,m\}} \in G_{g=\{1,..,27\}}$ and m is the population of entities.

Due to continuous evolution of software systems associated with large amount of developers coding the same project, development teams spend some considerable time to decide who is the most suitable member to perform a maintenance task on a given code entity. Despite log information of *VCS* and of Bug Report provide useful tips, the experiential knowledge about the expertise of each developer is what really point out most appropriate developer [19]. Moreover, this experimental knowledge is spread over all team members.

To overcome the challenge of building an expert-by-entity oracle, as reliable as possible, we submit our entities sample to development team. Source code of each entity, its callers and callees were presented to all team members at same time during a predefined meeting. Voluntarily, any developer have pointed among team members out who she thought was the expert of the analyzed entity that even could be herself. Then, remaining members adhered or not the indication. In case of disputes, new indications emerged until consensus expert was reached to.

In 28 of the 50 entities, the team have took consensus and pointed for each entity at least one expert. However, there were 22 entities for which the team did not appoint any experts. These ones were discarded to compound the oracle.

3) Measuring Expertise: To process ePol vocabulary we use VocabularyTools[4] that for this study purpose we configured it to be able: to extract identifiers that name classes, interfaces, enumerations, attributes, methods, parameters, and local variables; to collect comments and javadocs texts; to split identifiers coded both in camelcase as in underscore notation style; to extract root of words written in Portuguese language; and, to discard terms which had less than 3 characters.

[4]It is an open source suite tool that is able to extract and process software vocabulary of Java projects. More details in www.softwarevocabulary.org.

The result of processing on source code vocabulary is represented by a Term-Entity matrix as we exemplified in Table I and described in Section II-A4. The vocabulary of each developer is also represented by a frequency matrix TE^D as was defined in Section II-A5.

Experts are pointed out according to similarity value between developers and entities vocabularies . For every developer D that contributes with a project P, we calculate the cosine similarity between its vocabulary with the vocabulary of each entities present in the matrix TE_P. Thus, a similarity matrix Entity-Developer of project P, ED_P, is derived and exemplified by Table II. ED_P contains i lines where each one represent an entity of target version of project, and columns j indicates the number of contributing developers.

TABLE II
SIMILARITY MATRIX FOR ENTITIES AND DEVELOPERS VOCABULARIES OF A PROJECT P, ED_P.

Entities	Developers			
	D_1	$D_{...}$	D_{j-1}	D_j
E_1	$sim(E_1, D_1)$	$sim(E_1, D_{...})$	$sim(E_1, D_{j-1})$	$sim(E_1, D_j)$
E_2	$sim(E_2, D_1)$	$sim(E_2, D_{...})$	$sim(E_2, D_{j-1})$	$sim(E_2, D_j)$
$E_{...}$	...	...	...	...
E_i	$sim(E_i, D_1)$	$sim(E_i, D_{...})$	$sim(E_i, D_{j-1})$	$sim(E_i, D_j)$

For a given entity E_i, the developer D_j who has the greatest similarity value is recommended as the expert developer to E_i. Besides, it also possible to pointed not just a single expert but the k developers who have knowledge about E_i. It makes sense in situations where the principal expert is not available, for instance.

V. RESULTS AND DISCUSSION

Developer team has performed maintenance tasks on source code of ePol versions until the target one 0.5b. We have collected developers vocabularies of four development periods: two, four, six and nine months before the day when version 0.5b of ePol were released. We compute cosine-similarity of each developer vocabulary with source code vocabulary of target ePol version. Then, we have counted the total of entities whose similarity values were greater than 0.3, 0.4 and greater than 0.5. The graphic drawn in Figure 2 summarizes the results we have achieved. For every similarity limit we identified the same behavior: on first two months regardless of entities number the similarity between vocabularies is about 2.5% greater than the next two periods: four and six months. We believe this decreasing reflects the vocabulary decay phenomenon where newest words are weighted slightly more than older ones [18]. However for the four periods we observe, in average, the percentage of entities whose vocabulary has more than 0.5 of similarity with developers vocabularies is 20.63%, for more than 0.4 and more 0.3 the percentage increases for 25.47% and 29.38% respectively.

After we have confirm similarity relationship between entities and developers vocabularies, we used our approach, Sec-

tion III, to identify entities experts. We compared it, in terms of precision an recall, with two others traditional approaches: based on number of commits, and based on *%LoC*.

To be part of a project an entity had been inserted in *VCS* at some point by a given developer. Thus, from the perspective of both based on commit and based on *%LoC* approaches every entity always has its respective expert, even it has been manipulated only once: in its creation by its author. In case of our approach, not always an entity will have an expert associated with. Depending on the developmental period has been observed none developer vocabulary will share terms with entities vocabulary for a given target version.

Since a random strategy was used to define the sample that compound the oracle, Section IV-A2, we have no guarantee that each sampling entities had been manipulated during developmental period we chose to assess our approach. Because this, we have measured the accuracy of our approach regarding the same four different periods of developers vocabularies we used to observe similarity relationship behavior. Besides, we have computed accuracy considering both just one expert by entity (*top-1*) as two (*top-2*) and three (*top-3*) experts.

The oracle we have built allow us to measure approaches accuracy, and consequently to compare them. Table III contains precision and recall values we have achieved for each experimented approach. As expected, the accuracy for based on commit and based on *%LoC* has kept same values regardless parameters *top-k* and number of months have been changed.

In case of our approach, its accuracy depend on parameters values. First, taking *top-1* expert, as more contributions are considered to constitute vocabularies of developers, better the accuracy of our approach. When developers vocabularies are comprised by just two months of contributions precision and recall values is 0.25, but accumulation of nine months of contributions produces a precision and recall of 0.3214.

Scenarios in which more than one expert is considered, in general, we find that: as the period considered to generate the vocabulary of developers increases, the accuracy of our approach is close to values reached by the two baseline approaches. In our view this trend indicates that similarity of vocabularies is a promising way to identify experts.

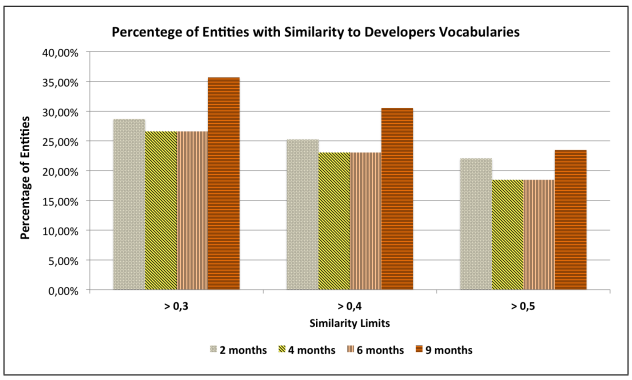


Fig. 2. Percentege of Entities with Similarity to Developers Vocabularies.

Generation of developers vocabulary does not count nor wrapping long statements lines or indentation adjustments. Disregard these types of formatting settings prevents improper expertise attribution by our approach. For example, most of development tools have text editing options to automatically adjust a source code formatting. Considering the two approaches we use as baseline, if this feature is performed and change code is committed, the expertise credit would be mistakenly attributed for whom have selected the feature on.

One other advantage of using similarity of vocabularies is its potential to recommend experts for orphaned entities of a system. In this scenario, would identify among the current team members one whose contributions on the vocabularies of other entities could quickly understand the code of orphan entity over which a given maintenance task would be executed.

A. Threats to Validity

Construct Validity: Vocabulary operations consider that identifiers are coded in camelcase or underscore notation style, and that comments and javadocs are written in Portuguese. Thus, our findings are limited by accuracy of those algorithms. The meeting with development team to build the oracle was scheduled to avoid lack of members, set its start end time to ensure its completion, and performed in presence of researchers to mediate possible conflicts among members. Besides, all meeting audio was recorded for eventually later audit of any ePol project stakeholder. However, we can not guarantee that groupthink phenomenon has not influenced on oracle result information.

External Validity: The apparently project-specific influences on the usage of software vocabularies to identify experts, suggest that, general conclusions may not be derived from our findings, although our approach seems to be promising to improve state-of-practice. Caution is necessary when applying our proposal to other projects.

VI. Related Work

Researches have stressed the importance of software vocabulary to facilitate program maintenance. Identifiers are chosen in accordance with stakeholders' personal preferences and experience [20], and related to problem domain concepts [11] to promote software understanding. When identifiers do not follow good convention rules [13] maintenance tasks are degraded.

In addition to the *VCS*, other repositories are mined to map expertise. For instance, mining *e-mails* exchanged among developers whose subject were related to code changes or bug reported indicates experts [6]. Learning machines techniques have also been used. Taking slots of bug repository information to learn patterns that indicate who should solve a kind of bug [21]. The *Degree-Of-Knowledge - DOK* model combines authority information with edition interactions performed by all others developers for point out who are the *experts* for a given source entity [5].

The aforementioned studies endorse its necessary to add other different aspects they already use to improve their accuracy. Besides, in the best of our knowledge in none of them

TABLE III
APPROACHES COMPARISON FOR *Top-k* EXPERTS CONSIDERING 9 MONTHS OF DEVELOPMENT.

Approach Based on	*top-k*	2 months		4 months		6 months		9 months	
		precision	recall	precision	recall	precision	recall	precision	recall
Commit	1	0.3571	0.3571	0.3571	0.3571	0.3571	0.3571	0.3571	0.3571
	2	0.2142	0.2142	0.2142	0.2142	0.2142	0.2142	0.2142	0.2142
	3	0.1547	0.1547	0.1547	0.1547	0.1547	0.1547	0.1547	0.1547
%LoC	1	0.3571	0.3571	0.3571	0.3571	0.3571	0.3571	0.3571	0.3571
	2	0.2321	0.2321	0.2321	0.2321	0.2321	0.2321	0.2321	0.2321
	3	0.1667	0.1667	0.1667	0.1667	0.1667	0.1667	0.1667	0.1667
Cosine-Similarity (Ours)	1	0.2500	0.2500	0.2500	0.2500	0.2857	0.2857	0.3214	0.3214
	2	0.1964	0.1964	0.1607	0.1607	0.1964	0.1964	0.1964	0.1964
	3	0.1905	0.1905	0.1548	0.1548	0.1786	0.1786	0.1667	0.1667

no association between developers and software vocabularies was used to model code expertise. The closest of our study was developed by Matter and colleagues [18]. In it, the similarity between the vocabularies of developers and content of bug reports is used to assign bugs to be fixed by one developer. In our proposal, we extract the vocabulary of source entities (classes and interfaces) and through their similarity with developers vocabulary we pointed the expert who will make a change or a code review whose location is known.

VII. CONCLUSIONS AND FUTURE WORK

Literature on software vocabulary provides evidences of developers' skills, personal preferences, as well as mental map of a project, and carries design aspects not captured by traditional structural metrics. The results we have achieved in this study increases the list of vocabulary usage: identifying code experts. Based on the accuracy we found, its obvious our approach based on vocabularies similarity must be refined. For instance, considering decay factor for old terms of a vocabulary and for inactivity time of a given developer.

By itself vocabulary usage does not seams to be sufficient to produce a revolutionary technique to automatic identify code experts. However, we must point out if additional cost of vocabularies usage is affordable or not. For this, we are conducting a new study to assess whether vocabularies capture different aspects of expertise than those captured by commonly used approaches.

We are extending the concept of vocabulary for groups of entities that share terms. We intend to investigate whether similarity between entities clusters and developers vocabularies point concerns experts.

REFERENCES

[1] K. Bennett and V. Rajlich, "Software Maintenance and Evolution: A Roadmap," *ICSE '00 Proceedings of the Conference on The Future of Software Engineering*, pp. 73 – 87, 2000.

[2] R. P. L. Buse and W. R. Weimer, "Learning a Metric for Code Readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, Jul. 2010.

[3] P. Hallam, "What Do Programmers Really Do Anyway?" in *The Microsoft Developer Network (MSDN). http://blogs.msdn.com/b/peterhal/archive/2006/01/04/509302.aspx*, 2006.

[4] J. Feigenspan, S. Apel, J. Liebig, and C. Kästner, "Exploring Software Measures to Assess Program Comprehension," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2011.

[5] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill, "A Degree-of-Knowledge Model to Capture Source Code Familiarity," *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, p. 385, 2010.

[6] A. Mockus and J. Herbsleb, "Expertise Browser: a quantitative approach to identifying expertise," *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pp. 503–512, 2002.

[7] L. Hattori and M. Lanza, "Mining the History of Synchronous Changes to Refine Code Ownership," *Mining Software Repositories, 2009. MSR'*, pp. 141–150, 2009.

[8] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse, "How Developers Drive Software Evolution," *Proceedings of the 2005 Eighth International Workshop on Principles of Software Evolution (IWPSE05)*, 2005.

[9] F. Rahman and P. Devanbu, "Ownership, experience and defects: A fine-grained study of Authorship," *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, p. 491, 2011.

[10] D. Lawrie, H. Feild, and D. Binkley, "Quantifying identifier quality: an analysis of trends," *Empirical Software Engineering*, vol. 12, no. 4, pp. 359–388, Dec. 2006.

[11] S. L. Abebe, S. Haiduc, A. Marcus, P. Tonella, and G. Antoniol, "Analyzing the Evolution of the Source Code Vocabulary," *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, pp. 189–198, 2009.

[12] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, Sep. 2006.

[13] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Exploring the Influence of Identifier Names on Code Quality: an empirical study," *oro.open.ac.uk*, 2010.

[14] L. Guerrouj, "Automatic Derivation of Concepts Based on the Analysis of Source Code Identifiers," *2010 17th Working Conference on Reverse Engineering*, pp. 301–304, Oct. 2010.

[15] D. Cavalcanti, K. Santos, D. Serey, and J. Figueiredo, "Using Software Vocabulary to Rank Classes that are Probably Impacted by a Bug Report," in *1st Workshop on the Next Five Years of Text Analysis in Software Maintenace - ICSM2012*, 2012, pp. 0–4.

[16] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic clustering: Identifying topics in source code," *Information and Software Technology*, vol. 49, no. 3, pp. 230–243, Mar. 2007.

[17] K. d. F. Santos, D. D. S. Guerrero, J. C. A. de Figueiredo, and R. A. Bittencourt, "Towards a Prediction Model for Source Code Vocabulary," in *1st Workshop on the Next Five Years of Text Analysis in Software Maintenace - ICSM2012*, 2012, pp. 0–4.

[18] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers," *6th IEEE International Working Conference on Mining Software Repositories*, pp. 131–140, May 2009.

[19] F. Servant, "Supporting bug investigation using history analysis," *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 754–757, Nov. 2013.

[20] S. Haiduc and A. Marcus, "On the Use of Domain Terms in Source Code," *The 16th IEEE International Conference on Program Comprehension*, vol. 0, pp. 113–122, 2008.

[21] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" *28th international conference on Software engineering - ICSE*, p. 361, 2006.