

# Quantifying the Similarities between Source Code Lexicons

Lauren R. Biggers and Nicholas A. Kraft  
Department of Computer Science  
The University of Alabama  
Tuscaloosa, AL 35487-0290  
{lbiggers|nkraft}@cs.ua.edu

## ABSTRACT

Several recent static analysis techniques automate software understanding activities by extracting textual information from source code and applying information retrieval models to the extracted corpora. These source code retrieval techniques show efficacy, but the literature provides no guidance regarding configuration of their constituent processes. For example, the literature provides conflicting information regarding the benefit of extracting comments and string literals along with identifiers such as method or variable names. In this paper we present an initial investigation into the similarities between three source code lexicons described in the literature: identifiers, comments, and string literals. We address three research questions using a case study of six open source Java projects. The results indicate that methods uniquely contain from 30% to 60% of the projects' terms, whereas the comments uniquely contain from 22% to 45% of the terms. Future work includes analyzing the extent to which comments and string literals introduce domain terms rather than non-domain terms.

## Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—*enhancement, restructuring, reverse engineering, and reengineering*

## General Terms

Documentation, Design

## Keywords

Program comprehension, information retrieval, static analysis, software evolution and maintenance

## 1. INTRODUCTION

Software evolution is the most costly phase of development, contributing up to 80% of total software cost [4, 9].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

49th ACM Southeast Conference, March 24–26, 2011, Kennesaw, GA, USA  
Copyright 2011 ACM 978-1-4503-0686-7/00/0010 ...\$10.00.

Many recent research efforts seek to curb total software cost by reducing developer effort during software evolution. Partial or full automation of software understanding activities such as bug localization [18] and feature location [20] is key, because software understanding activities consume 50% to 90% of developer effort during software evolution [19].

Static analysis techniques can be used to automate software evolution tasks. Traditional static analysis techniques operate on source code models that represent structural information (e.g., potential calls between methods) or semantic information (e.g., potential paths through a program). However, some recent static analysis techniques operate on source code models that represent textual information (e.g., method names and comments). These source code retrieval techniques apply information retrieval (IR) models to corpora built from text embedded in source code.

Figure 1 illustrates the source code retrieval process. Researchers have used this process to investigate the automation of numerous software understanding activities. However, while the source code retrieval process shows efficacy, the effects of varying its subprocesses are not described in the literature. Consider text extraction, in which we collect the text from which a corpus is built. Some collect only identifiers (i.e., names of classes, methods, parameters, etc.), whereas others also collect comments or string literals (hereafter literals). Researchers thus far report only anecdotal evidence to justify the inclusion/exclusion of comments/literals for a given experiment. For instance, Anquetil and Lethbridge [5] assume that “comments are obsolete and misleading,” whereas Liu et al. [17] report that “a significant amount of domain knowledge is embedded in the comments and identifiers present in source code.”

Given the conflicting reports in the literature, we seek to discover a metric to help decide which text to collect during document extraction for a particular software system and software evolution task. Toward this goal, we present an initial investigation into the similarities between three distinct source code lexicons mentioned in the literature: identifiers, comments, and literals. Quantifying the shared and unique properties of these lexicons could aid in characterizing the expected effects of collecting comments or literals. We formulated three research questions to guide our investigation:

- RQ1: *How similar are the three lexicons?*
- RQ2: *How similar are the lexicons for source code elements (e.g., classes or methods) to their corresponding comments and literals?*
- RQ3: *How much unique information does each source code element contribute?*

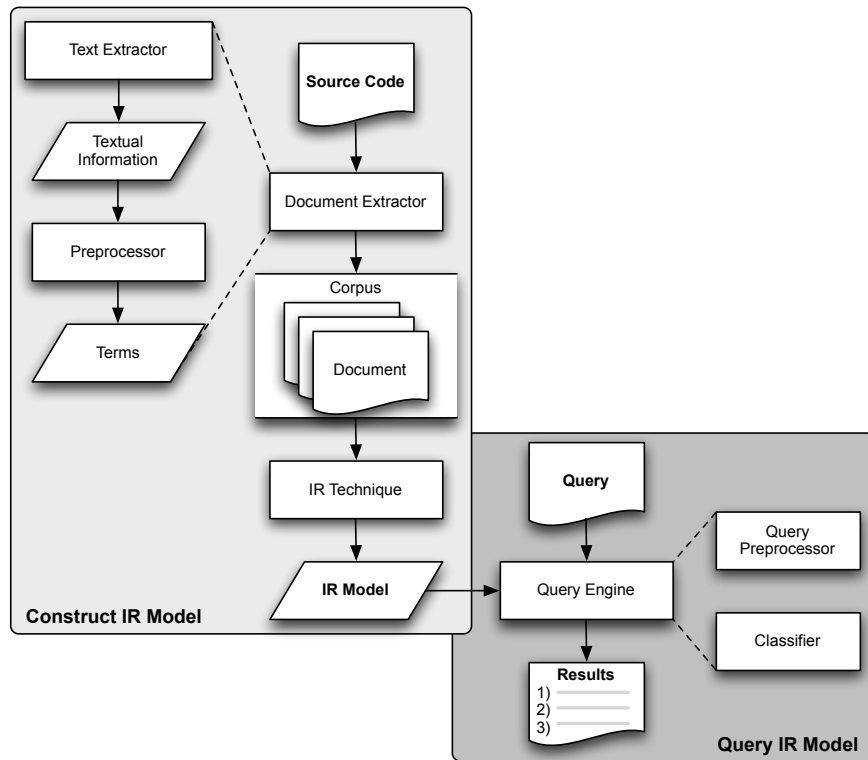


Figure 1: Source Code Retrieval Process

In the next section we provide an overview of our approach. In Section 3 we describe our research method and report the results of our case study. In Section 4 we review the research that relates to our work, and finally, in Section 5 we conclude.

## 2. OVERVIEW

In the case study, we measure the similarities between identifiers, comments, and literals in source code and examine the collected metrics to address our three research questions. To do so, we instantiate the process shown in Figure 1 as follows.

We extract documents at different levels of granularity to answer the three research questions. For *RQ1* we extract three documents (identifiers, comments, and literals) per project. For *RQ2* we extract three documents (again identifiers, comments, and literals) for each file, class, method, and parameter list in each project. Finally, for *RQ3*, we extract five documents (class identifiers, method identifiers, parameter list identifiers, comments, and literals). Note that for *RQ2* and *RQ3*, we uniquely associate each term instance with the document representing the immediately containing source code element. For example, we associate a term instance found in a method with that method’s document, a term instance found in a class (but not in a method) with that class’s document, and so on.

For *RQ2* and *RQ3*, we build two variants of each document that represents a class or a method. In the *non-nested* variant, every class (or method) maps to one document, that is, if class *B* is nested within class *A*, each class maps to a distinct document, and the document for class *A* does not

include the contents of class *B*. In the *nested* variant, only a class in the file scope (or a method in the class scope) maps to a document, that is, if class *B* is nested within class *A*, we create one document (labeled class *A*) that includes the contents of classes *A* and *B*.

Before associating a term with a document, we acquire that term by preprocessing the text extracted from the source code. The preprocessing steps that we apply are: identifier separation, in which tokens that follow common coding style conventions are split (e.g., `input_file`, `inputFile`, and `Input-File` are each split into two terms), case normalization, in which each upper case letter is replaced with the lower case letter, stop word elimination, in which common words such as English language articles (e.g., “an” or “the”) and programming language keywords are removed, and stemming, in which suffixes are stripped (e.g., “jumper”, “jumping”, and “jumps” all become “jump”). In addition, we merge adjacent comments of the same type (i.e., line or block), remove comments that contain copyright notices, and associate the remaining comments with adjacent source code elements.

The resulting documents form a corpus, to which we apply the vector space model (VSM) using a term frequency weighting scheme. That is, each document is mapped to an  $n$ -dimensional vector, where  $n$  is the number of unique terms in the corpus. The entry for each term is the number of times the term appears in the given document. Our query engine compares documents pairwise (as appropriate for each particular research question), and because we compare documents (the terms of which have undergone four preprocessing steps), the query preprocessor is null. We use cosine similarity, which approaches 1.0 as two documents become more similar, as the classifier.

**Table 1: Open Source Java Projects Examined**

Name	URL	Version	NCLOC	CLOC	Files	Classes	Methods	Terms
Ant	ant.apache.org	1.8.1	102K	89K	829	1,244	10,147	13,921
ArgoUML	argouml.tigris.org	0.30.2	30K	20K	164	268	4,613	4,272
CAROL	carol.objectweb.org	2.0.5	10K	10K	154	188	1,104	2,605
dnsjava	dnsjava.org	2.1.1	14K	5K	121	161	1,313	2,402
JDT Core	eclipse.org	3.6.1	290K	120K	1,182	1,478	19,152	24,218
jEdit	jedit.org	4.3.2	100K	43K	483	1,099	6,560	11,021

We use the VSM and cosine similarity due to their popularity in the literature. In particular, we have identified 30 software maintenance and evolution research papers from 1999 to 2009 that use the VSM, and 26 of those papers use cosine similarity as the classifier. See, for example, Antoniol et al. [7], Canfora et al. [10], and Gay et al. [13].

### 3. CASE STUDY

In this section we describe the case study that we conducted to address our three research questions. We describe the data examined then the results.

#### 3.1 Data Examined

Table 1 lists information about the six open source Java projects that form our test suite. For each project we list the name, version, the approximate number of non-commented lines of code (NCLOC), the approximate number of commented lines of code (CLOC), the number of files, the number of classes, the number of methods, and the total number of terms. We used *cloc*<sup>1</sup> version 1.53 to compute NCLOC and CLOC; note that CLOC includes lines from copyright notices. We chose the six projects for their variety of size and domain, and for their frequent use in previous studies.

For *Apache Ant* we consider only files in the `src/main` directory, and for *ArgoUML* we consider only the core model. For *CAROL* and *dnsjava* we consider only files in the `src` and `org` directories, respectively. We consider all Java files in the *JDT Core* source tree, but consider only the files in the `jEdit/org` directory for *jEdit*.

#### 3.2 Results

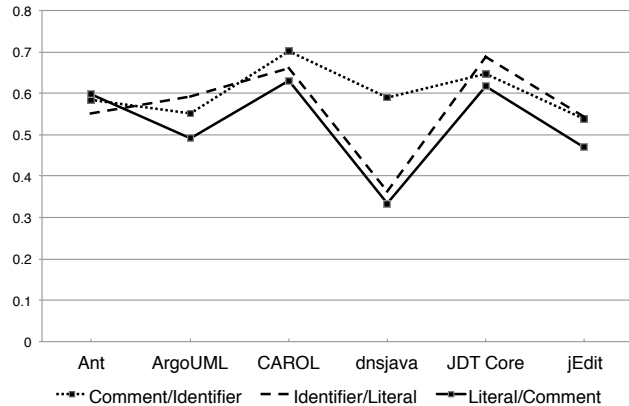
In this section we restate each question and describe the relevant results.

##### 3.2.1 RQ1: How similar are the three lexicons?

To address this research question, we extract three documents (identifiers, comments, and literals) per project. We then compute cosine similarities for the following pairs of lexicons: comment-identifier, identifier-literal, and literal-comment. Figure 2 illustrates the results. For five of the projects (for all except *dnsjava*), the three similarity scores are within a 0.1 range and the three lexicons are generally more similar than not. However, for the *dnsjava* project, the similarity between literals and identifiers or comments is much less than for other projects.

The lack of similarity between literals and other terms in *dnsjava* could indicate different uses of the literals. Literals in source code can serve many purposes, including prompts/dialogs, user error messages, and internal error messages. Further study of literals, including an analysis of the

<sup>1</sup><http://cloc.sourceforge.net>



**Figure 2: Cosine Similarity for Project Documents**

use of domain terms in literals, is needed to understand how literals in *dnsjava* differ from those in other projects.

##### 3.2.2 RQ2: How similar are the lexicons for source code elements to their corresponding comments and literals?

Again, for classes and methods, we build two variants of each document: non-nested and nested (see Section 2). For each of the variants, we compute the average cosine similarity across all instances of a particular source code element. For example, for each method document, we compute the cosine similarity between the identifiers and comments for the method. Note that a method’s comments include comments contained in the method and comments that immediately precede the method. After computing the cosine similarity for each method, we calculate the average of those values. Table 2 lists the results. Unlike in the results for *RQ1*, the lexicons compared for *RQ2* are more dissimilar than similar.

Because we could not identify any work regarding the effects of building non-nested or nested documents, we tested the following hypotheses:

- H0*: There is no difference between the mean of the average cosine similarities across all classes in a nested document and the mean of the average cosine similarities across all classes in a non-nested document.
- H1*: There is a statistically significant difference between the mean of the average cosine similarities across all classes in a nested document and the mean of the average cosine similarities across all classes in a non-nested document.

**Table 2: Average Cosine Similarity**

Vocabulary	Ant	ArgoUML	CAROL	dnsjava	JDT Core	jEdit
File	0.267	0.359	0.168	0.163	0.194	0.233
Class	0.364	0.207	0.308	0.277	0.267	0.174
Nested Class	0.418	0.332	0.330	0.276	0.314	0.307
Method	0.394	0.166	0.331	0.215	0.219	0.274
Nested Method	0.398	0.170	0.334	0.215	0.223	0.281
Parameter List	0.266	0.111	0.210	0.097	0.100	0.121

(a) Comment Lexicon vs. Identifier Lexicon

Vocabulary	Ant	ArgoUML	CAROL	dnsjava	JDT Core	jEdit
File	0.000	0.000	0.000	0.000	0.000	0.000
Class	0.068	0.015	0.035	0.039	0.026	0.024
Nested Class	0.086	0.024	0.040	0.027	0.032	0.048
Method	0.046	0.064	0.072	0.055	0.012	0.070
Nested Method	0.046	0.066	0.070	0.055	0.012	0.071
Parameter List	0.000	0.000	0.000	0.000	0.000	0.000

(b) Literal Lexicon vs. Identifier Lexicon

We computed a paired  $t$  test to compare the average cosine similarity across classes in a nested document to the average cosine similarity across classes in a non-nested document. The analysis produced a significant  $t$  value ( $t_{(11)} = 2.6195$ ,  $p = .024$ ) with  $\alpha = .05$ , so we reject the null hypothesis of no difference. We ran the equivalent paired  $t$  test for methods, and the analysis produced a significant  $t$  value ( $t_{(11)} = 2.6149$ ,  $p = .024$ ). Again, we reject the corresponding null hypothesis of no difference. However, note that the effect of building nested documents rather than non-nested documents is trivial.

### 3.2.3 RQ3: How much unique information does each source code element contribute?

To gain a deeper understanding of how the terms in the comment lexicon are related to the identifier and literal lexicons, we measure the percentage of terms that are unique to the comment lexicon. For example, for Ant’s comment lexicon,  $L$ , we compute two sets:  $U$ , which contains terms that appear only in the comment lexicon, and  $A$ , which contains terms that appear in the comment lexicon and at least one other lexicon. Note that  $L = U \cup A$ . If few unique terms exist, the comments may not be high level enough to contribute meaningful information to the developer. If many unique terms exist, the comments may contribute new domain terms or may be outdated and unclear to the developer.

To complete Figure 3, we computed  $|U|/|A| * 100$  for each  $L$ , a lexicon for one of the six projects. Each value in each column of the figure represents the percentage of terms in that lexicon which are not shared with any other lexicon. We divide the identifier lexicon into lexicons for class identifiers, method identifiers, and parameter list identifiers. We calculated percentages for both non-nested and nested documents, but a paired  $t$  test confirmed no significant difference. Thus, we present data for only non-nested documents.

Figure 3 shows that in comment lexicons a significant number of terms (from 22% to 45%), are unique to the comment lexicon for the given project. For five projects, the method lexicon uniquely contains the largest percentage of terms (from 30% to 57%). Overall, the comment lexicon

contains the second largest percentage of unique terms, and the literal lexicon the third largest (from 6% to 20%).

## 4. RELATED WORK

The combination of the VSM and cosine similarity has been used to help automate software evolution tasks, including traceability tasks [1], localization tasks [22], and clone detection [21]. Moreover, properties of source code lexicons have been studied in the context of software component retrieval [16], program comprehension [2, 6, 11], and quality assessment [3, 15]. The works on program comprehension and quality assessment are most related to our own, because we are interested in how comments and literals affect models built to address these issues.

Abebe et al. [2] present the work most related to ours, a study of the evolution of the source code lexicons for two evolving software projects. They partition each source code lexicon into five sub-vocabularies: class name, attribute name, function name, parameter name, and comment. Next, Abebe et al. analyze the vocabularies to understand how the source code lexicon evolves in an evolving software project. They note that lexicon size and system size tend to evolve in parallel, that the class name vocabulary has the most terms in common with other vocabularies, that the comment vocabulary tends not to reflect changes to the name vocabularies, that new names introduce few terms into the vocabulary, and that frequent terms are associated with domain concepts.

Haiduc and Marcus [14] report the results of a study in which they investigate the provenance of domain terms in source code, focusing on the identifier and comment lexicons. Haiduc and Marcus report that across their test cases, 23% of the domain terms in the source code are unique to the comment lexicon, but only 11% of the domain terms are unique to the identifier lexicon.

Anquetil and Lethbridge [6] seek to define a reliable naming convention for identifiers in legacy software systems. Similarly, Caprile and Tonella [11] aim to improve program comprehension by restructuring program identifiers. They study the use of a standard lexicon and a standard syntax

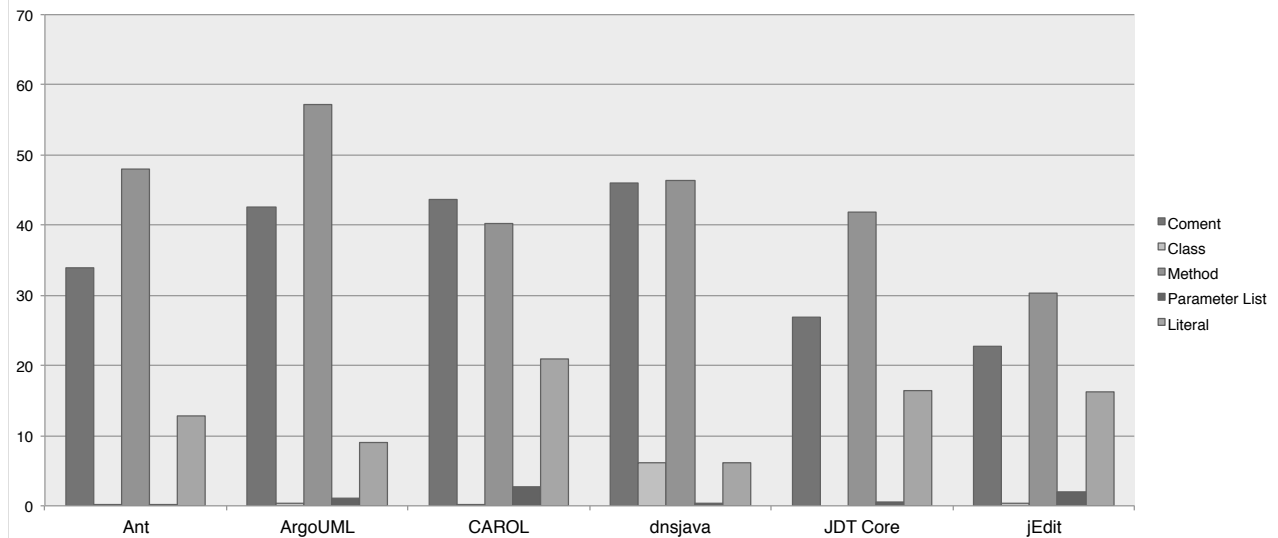


Figure 3: Percentage of Unique Terms in each Lexicon

for identifier structures, using a dictionary-based tool to re-structure program identifiers.

Lawrie et al. [15] develop a tool that considers identifiers and related comments when estimating quality attributes for a given software project. The tool considers the correspondence of comments and code when creating its quality estimate. In similar work Abebe et al. [3] define and detect lexicon bad smells. They develop a tool and describe the results of detecting five different kinds of bad lexical smells. Further, they note that views on what constitutes a lexical bad smell may differ between developers.

Antoniol et al. [8] report the results of a study on the stability of the source code lexicon in comparison to the stability of the source code structure and the frequency of changes to source code elements. The results show that the stability of the lexicon is more stable than changes to structure, that changes to the lexicon were rare during the evolution of a software system, and that the lexicon is found to be more critical to the correspondence between the system and its domain. Antoniol et al. note that the lack of changes may be due to lack of tools, a reluctance of developers to change their mental model, or careful domain analysis.

Fluri et al. [12] study the correlation between comments and changes in source code for three Java projects. Their goals are to determine what types of entities are commented, whether the ratio between source code and comments remains stable, and whether the comments are kept up-to-date. They state that the entity types most often commented are declarations, control and loop structures, special method calls, and variable declarations. Fluri et al. report that for two of the projects, comment changes are triggered by source code changes less than 50% of the time, and that for all three projects, 97% of comment changes occur in the same revision as the source code changes.

## 5. CONCLUSION

Source code retrieval techniques are often used to automate software maintenance and evolution tasks. Some researchers describe comments as a valuable source of domain

knowledge [17], whereas others describe comments as too outdated to be of any use [5]. No metric exists to determine whether to include or exclude comments, so the decision is often based on their own experience or on the experience of previous researchers. In this paper we described a study toward the goal of discovering a metric to help decide which text to collect during document extraction for a particular software system and evolution task.

In our study we investigated the similarities between three distinct source code lexicons mentioned in the literature: identifiers, comments, and literals. We studied six open source Java projects and addressed three research questions. With *RQ1* we sought to quantify the similarity between a project’s comment lexicon and its identifier lexicon (and to quantify the similarities between the identifier/literal lexicons and the literal/comment lexicons). For five of the projects, we found that the three similarity scores were within a 0.1 range and that the three lexicons were more similar than not. Future work includes expanding this experiment to compute these similarities for additional Java projects, as well as for projects in other languages. In addition, we want to perform a study similar to that of Haiduc and Marcus [14] to investigate the overlap of domain terms between the comment/identifier lexicons, the identifier/literal lexicons, and the literal/comment lexicons.

With *RQ2* we sought to measure the similarity between the identifier and comment lexicons for particular source code elements, including the class, method, and parameter list. We found low (less than 0.420) average cosine similarity for all source code elements across all projects. Moreover, we sought to discover whether a difference exists between the mean of the average cosine similarities across all classes in a nested document and the mean of the average cosine similarities across all classes in a non-nested document. We found that a statistically significant, though trivial, difference exists for the projects in our study. Again, future work includes expanding this experiment.

For *RQ3* we investigated the percentages of unique terms in several lexicons, including comment, literal, class identifier, method identifier, and parameter list identifier. We

found that the method identifier lexicon (which includes things such as the method name, local variable names, and the names of called methods), contains the largest percentage of unique terms, followed by comments and literals. Though this result is not surprising, we did not expect the difference between method identifiers and comments to be so small (roughly, less than 10% on average).

Finally, additional future work includes analyzing the extent to which comments and string literals introduce domain terms rather than non-domain terms. This comparison is distinct from that of Haiduc and Marcus [14] in that we want to measure the relative impact of including comments in extracted documents on domain versus non-domain terms. That is, we want to know the extent to which the comment lexicon emphasizes individual terms in the identifier lexicon and whether those emphasized terms are domain terms or non-domain terms, whereas Haiduc and Marcus studied only the (boolean) overlap between the domain terms in the comment and identifier lexicons.

## 6. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 0915559.

## 7. REFERENCES

- [1] A. Abadi, M. Nisenson, and Y. Simionovici. A traceability technique for specifications. In *Proceedings of the 16<sup>th</sup> IEEE International Conference on Program Comprehension*, pages 103–112. IEEE Computer Society, June 2008.
- [2] S. Abebe, S. Haiduc, A. Marcus, P. Tonella, and G. Antoniol. Analyzing the evolution of the source code vocabulary. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, pages 189–198, 2009.
- [3] S. Abebe, S. Haiduc, P. Tonella, and A. Marcus. Lexicon bad smells in software. In *Proceedings of the 16th Working Conference on Reverse Engineering*, pages 95–99, 2009.
- [4] G. Alkhatib. The maintenance problem of application software: An empirical analysis. *Journal of Software Maintenance: Research and Practice*, 4(2):83–104, 1992.
- [5] N. Anquetil and T. Lethbridge. File clustering using naming conventions for legacy systems. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 2–, 1997.
- [6] N. Anquetil and T. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 4–, 1998.
- [7] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, October 2002.
- [8] G. Antoniol, Y.-G. Guéhéneuc, E. Merlo, and P. Tonella. Mining the lexicon used by programmers during software evolution. In *Proceedings of the International Conference on Software Maintenance*, pages 14–23, 2007.
- [9] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.
- [10] G. Canfora, L. Cerulo, and M. D. Penta. Identifying changed source code lines from version repositories. In *Proceedings of the 4<sup>th</sup> International Workshop on Mining Software Repositories*, page 14, 2007.
- [11] B. Caprile and P. Tonella. Restructuring program identifier names. In *Proceedings of the International Conference on Software Maintenance (ICSM '00)*, pages 97–, Washington, DC, USA, 2000. IEEE Computer Society.
- [12] B. Fluri, M. Wursch, and H. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Proceedings of the 14th Working Conference on Reverse Engineering*, pages 70–79, 2007.
- [13] G. Gay, S. Haiduc, A. Marcus, and T. Menzies. On the use of relevance feedback in IR-based concept location. In *Proceedings of the International Conference on Software Maintenance*, 2009.
- [14] S. Haiduc and A. Marcus. On the use of domain terms in source code. In *Proceedings of the 16th International Conference on Program Comprehension*, pages 113–122, 2008.
- [15] D. Lawrie, H. Feild, and D. Binkley. Leveraged quality assessment using information retrieval techniques. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 149–158, Washington, DC, USA, 2006. IEEE Computer Society.
- [16] E. Linstead, P. Rigot, S. Bajracharya, C. Lopes, and P. Baldi. Mining internet-scale software repositories. In *Advances in Neural Information Processing Systems*, March 2008.
- [17] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 234–243, 2007.
- [18] S. Lukins, N. Kraft, and L. Etzkorn. Bug localization using latent dirichlet allocation. *Information and Software Technology*, 52(9):972–990, Sept. 2010.
- [19] H. Müller, J. Jahnke, D. Smith, M.-A. Storey, S. Tilley, and K. Wong. Reverse engineering: A roadmap. In *Proceedings of the Future of Software Engineering*, pages 47–60, June 2000.
- [20] D. Poshyvanyk, Y. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, June 2007.
- [21] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th International Conference on Software Engineering*, pages 461–470, 2008.
- [22] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNI AFL: Towards a static noninteractive approach to feature location. *ACM Trans. Softw. Eng. Methodol.*, 15(2):195–226, 2006.