2011 19th IEEE International Conference on Program Comprehension

Can Better Identifier Splitting Techniques Help Feature Location?

Bogdan Dit[1], Latifa Guerrouj[2], Denys Poshyvanyk[1], Giuliano Antoniol[2]
[1]Department of Computer Science
The College of William and Mary
Williamsburg, Virginia, USA
{bdit, denys}@cs.wm.edu
[2]Department of Computer Science Engineering
École Polytechnique de Montréal
Québec, Canada
{latifa.guerrouj, giuliano.antoniol}@polymtl.ca

Abstract — **The paper presents an exploratory study of two feature location techniques utilizing three strategies for splitting identifiers: CamelCase, Samurai and manual splitting of identifiers. The main research question that we ask in this study is if we had a perfect technique for splitting identifiers, would it still help improve accuracy of feature location techniques applied in different scenarios and settings? In order to answer this research question we investigate two feature location techniques, one based on Information Retrieval and the other one based on the combination of Information Retrieval and dynamic analysis, for locating bugs and features using various configurations of preprocessing strategies on two open-source systems, Rhino and jEdit. The results of an extensive empirical evaluation reveal that feature location techniques using Information Retrieval can benefit from better preprocessing algorithms in some cases, and that their improvement in effectiveness while using manual splitting over state-of-the-art approaches is statistically significant in those cases. However, the results for feature location technique using the combination of Information Retrieval and dynamic analysis do not show any improvement while using manual splitting, indicating that any preprocessing technique will suffice if execution data is available. Overall, our findings outline potential benefits of putting additional research efforts into defining more sophisticated source code preprocessing techniques as they can still be useful in situations where execution information cannot be easily collected.**

Keywords-feature location; information retrieval; dynamic analysis; identifier splitting algorithms

I. INTRODUCTION

Early work on program comprehension and mental models [33, 35] highlighted the significance of textual information to capture and encode programmers' intent and knowledge in software. Recent research efforts have studied how software developers capture and express their intent in natural language embodied in source code. Identifiers used by programmers as names for classes, methods, or attributes in source code or other artifacts contain vital problem domain information [2, 4, 7, 15, 18, 23, 28, 34] and account for approximately more than half the source code in software [7]. These names often serve as a starting point in many program comprehension tasks [4]; thus, it is imperative that these names clearly reflect the concepts that they are supposed to represent, since self-documenting identifiers reduce the time and effort to acquire a basic comprehension level for any maintenance task [2].

The magnitude of a program's lexicon can hardly be underestimated. Identifiers and comments represent an important source of domain information that is used by (semi-) automated techniques to recover traceability links among software artifacts [1, 24] and locate features in source code [8, 21, 25, 27, 29, 30]. Prior work [16, 32] employed a natural language-based representation of source code, based on the conjecture that there is an intrinsic pattern in unstructured textual information, to support a range of program comprehension activities. Due to the large abstraction gap between the domain of a software system and the implementation mechanisms offered by programming languages, the mapping between domain concepts and their implementation in source code is frequently ambiguous, as these concepts are distorted and scattered in the code [28].

The problem of extracting and analyzing the textual information in software artifacts was recognized by the software engineering research community only recently. Information Retrieval (IR) methods were proposed and used effectively to support program comprehension tasks, such as feature (or concept) location and traceability link recovery. These IR-based approaches vary not only in their scope, but also in their underlying indexing mechanisms, corpus generation, or results analysis methods. Identifier splitting is one of the essential ingredients in any feature location or traceability recovery technique [1, 8, 21, 24, 27, 29] , since it helps disambiguate conceptual information encoded in compound (or abbreviated) identifiers. The widely adopted approach is based on the CamelCase splitting algorithm, with more sophisticated strategies, such as Samurai [11] and TIDIER [14], recently proposed in the literature.

In this paper we investigate the impact of three identifier splitting techniques (CamelCase, Samurai and manually built splitting (i.e., Oracle)) on the accuracy of feature location in presence and absence of execution information. The main research question that we ask in this study is *if we had a perfect technique for splitting identifiers, such as a manually built oracle, would it still help improve accuracy of feature location techniques applied in different scenarios and settings*? To answer this research question we investigate two feature location techniques (FLTs), one based on IR and the other one based on the combination of IR and dynamic analysis (IRDyn), for locating bugs and features using different configurations of preprocessing strategies on two open-source systems, Rhino and jEdit. Our findings reveal that feature location techniques using IR can benefit from better preprocessing algorithms, and that their improvement in effectiveness while using manual splitting over state-of-the-art approaches is statistically significant. However, the results of the IRDyn FLT do not show any improvement while using manual splitting, indicating that any

1063-6897/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPC.2011.47

preprocessing technique will suffice if execution data is available.

II. Background on Preprocessing Unstructured Information in Software

In this section we overview some of the existing work in the field of feature location and identifier splitting. In particular, we overview two feature location techniques and three approaches for splitting identifiers that are used in our empirical study.

A. Feature Location in Software

Unstructured textual information in software, found in identifiers and comments encodes important problem domain and design decisions about a software system. This unstructured data lends itself for further analysis using IR techniques that can be leveraged to support feature location in source code. *Feature location* is the activity of finding the source code elements (i.e., methods or classes) that implement a specific feature (e.g., "print page in a text editor" or "add bookmark in a web-browser") [25, 27]. In this work, we rely on two feature location approaches that use IR and a combination of IR and dynamic analysis. While there are several IR techniques that have been successfully applied in the context of feature location, such as the Vector Space Model [8], Latent Semantic Indexing (LSI) [21, 27, 29, 30], and Latent Dirichlet Allocation [22], this empirical study focuses on evaluating LSI for feature location, and the notation IR is used to denote that LSI is the default information retrieval method used in the study. We also provide the details of these feature location approaches and explain the role of identifier splitting techniques in this process. Feature location via LSI follows five main steps: generating a corpus, preprocessing the corpus, indexing the corpus using LSI, formulating a search query and generating similarities and finally, examining the results.

Step one – generating the corpus. The source code of a software system is parsed, and all the information associated with a method (i.e., comments, method declaration, signature and body) will become a document in the system corpus. In other words, we are using a method-level granularity for the corpus, so each method from the source code has a corresponding document in the corpus.

Step two – preprocessing the corpus. The generated corpus is then preprocessed in order to normalize the text contained in the documents. This step includes removing operators, programming language keywords, or special characters. Additionally, compound identifiers are split using the algorithms that are explained in details in subsection II.B, as these algorithms are at the core of this paper. The split identifiers are then stemmed (i.e., reduced to their root form) using the Porter stemmer [26], and finally the words that appear commonly in English (i.e., "a", "the", etc.) are eliminated.

Step three - indexing the corpus using LSI. The preprocessed corpus is transformed into a term-by-document matrix, where each document (i.e., method) from the corpus is represented as a vector of terms (i.e., identifiers). The values of the matrix cells represent the weights of the terms from the documents, which are computed using the term frequency – inverse document frequency (tf-idf) weight. The matrix is then decomposed using Singular Value Decomposition [6] which decrease the dimensionality of the matrix by exploiting statistical co-occurrences of related words across the documents.

Step four – formulating a search query and generating similarities. The software developer chooses a set of words (i.e., a *query*) that describe the feature or bug being sought (e.g., "print page"). The query is converted into a vector-based representation, and the cosine similarity between the query and every document in the reduced space is computed. In other words, the textual similarity between the bug description and every method from the software system is computed.

Step five – examining the results. The list of methods is ranked based on their cosine similarities with the user query. The developer starts investigating the methods in order, from the top of the list (i.e., most relevant methods first). After examining each method the developer decides if that method belongs to the feature of interest or not. If it does, the feature location process terminates. Otherwise, the developer can continue examining other methods, or refine the query based on new information gathered from examining the methods and starting from Step 4 again.

Feature location via LSI and dynamic information has one additional step, which can take place before the Step 4 described earlier.

Step for collecting execution information. The software developer triggers the bug, or exercises the feature by running the software system and executing the steps to reproduce from the description of the feature or bug. This process invokes the methods that are responsible for the bug or feature and these methods are collected in an execution trace. The developer can take advantage of this information by formulating a query (Step 4) and examining the results (Step 5) produced by ranking only the methods found in the execution trace (as opposed to ranking all the methods of the software system). The advantage of using execution information is that it reduces the search space, thus increasing the performance of feature location.

In this paper, we consider the IR and IRDyn FLTs. While previous studies have shown that the IRDyn FLT outperforms its basic version (i.e., IR FLT) [21, 27, 29, 30], the goal of this paper is to study the impact of the preprocessing techniques from Step 2 on the accuracy of feature location.

B. Background on Identifier Splitting Technique

State-of-the-art approaches to split identifiers into separate words are the CamelCase splitter, the Samurai approach proposed by Enslen et al. [11], and the recent TIDIER approach [14].

1) CamelCase Splitting Technique

The de facto splitting algorithm is CamelCase. This simple, fast, and widely used preprocessing algorithm has been previously applied in multiple approaches to feature location and traceability link recovery [1, 21, 24, 25, 27, 29,

30]. This approach splits compound identifiers according to the following rules:

RuleA: Underscore, structure and pointer access, as well as special symbols are replaced with the space character.

RuleB: Identifiers are split where terms are separated using the CamelCase convention. For example, *userId* is split into *user* and *Id* while *setGID* is split into *set* and *GID*.

RuleC: When two or more upper case characters are followed by one or more lower case characters, the identifier is split at the last-but-one upper-case character. For example, *SSLCertificate* is split into *SSL* and *Certificate*.

Sometimes, a space is inserted before and after each sequence of digits. For example, *print_file2device* is split into *print*, *file*, *2*, and *device,* while *cipher128_code* is split into *cipher*, *128*, and *code.* Overall, a CamelCase splitting algorithm cannot split effectively same-case composite words, such as *USERID*, *currentsize*, into separate terms.

2) Samurai Splitting Algorithm

Samurai [11] is an automatic approach to split identifiers into sequences of terms by mining term frequencies in large source code bases. It relies on two assumptions. First, it assumes that a substring composing an identifier is also likely to be used in other parts of the program (or in other programs) alone or as a part of other identifiers. Second, given two possible splits, the split that most likely represents the developer's intent partitions the identifier into terms occurring more often in the program. In other words, central to Samurai is the idea of using two tables of frequencies: one program specific and one mined out of a large corpus of programs, to find the most likely identifier split. Furthermore, the frequency tables are used in conjunction with CamelCase rules. In fact, Samurai algorithm first tries to apply CamelCase split and then ranks possible splits according to its identifiers frequency tables. In this way Samurai overcomes the main limitation of CamelCase, by being able to correctly split same-case identifiers, such as *USERID*, *currentsize,* or mixed-case (e.g., *DEFMASKBit*). Refer to [11] for more details on Samurai and its evaluation.

3) TIDIER: Term IDentifier RecognizER

TIDIER [14] is a novel approach to split program identifiers using high-level and domain concepts captured into multiple dictionaries. The approach is based on a thesaurus of words and abbreviations and uses a modified string-edit distance [20] between terms and words as a proxy for the distance between the terms and the concepts they represent. The main assumption made by TIDIER is the fact that it is possible to mimic developers when creating an identifier relying on a set of transformation rules on terms/words. For example, to create an identifier for a variable that counts the number of software defects, the two words, *number* and *defects*, can be concatenated with or without an underscore, or following the CamelCase convention e.g., defects_number, defectsnumber or defectsNumber. Developers may drop vowels and (or) characters to shorten one or both words of the identifier, thus creating defectsNbr or nbrOfdefects. TIDIER uses contextual information in the form of specialized dictionaries (e.g., acronyms, contractions and domain specific terms) and mimics the process of transforming words via contraction rules; more details can be found in [14]. It is important to emphasize that TIDIER does not perform significantly better than Samurai on Java code and even though TIDIER and Samurai outperform CamelCase, Samurai is much faster than TIDIER. For this reason, TIDIER was only used as a reference in supporting the construction of the Oracle but not in the empirical study or to generate new terms as in [14].

Table 1 The configurations of the two FLTs (i.e., IR and IRDyn) based on the splitting algorithm

Splitting Algorithm	IR FLT	IRDyn FLT
CamelCase (Baseline)	$IR_{CamelCase}$	$IR_{CamelCase}Dyn$
Samurai	$IR_{Samurai}$	$IR_{Samurai}Dyn$
Oracle (Manual Split)	IR_{Oracle}	$IR_{Oracle}Dyn$

III. Empirical Study Design

The goal of this study is to compare accuracy of two FLTs (i.e., IR and IRDyn), when utilizing three identifier splitting algorithms: CamelCase, Samurai and Oracle (i.e., manual splitting of identifiers). This study is done from the perspective of researchers who want to understand if existing approaches for splitting identifiers can improve accuracy of FLTs under different scenarios and settings, including best possible scenario where splitting is done by experts. In addition, we are interested to know if an advanced splitting algorithm would be still useful for enhancing the accuracy of feature location when execution information is used.

The context consists of two Java applications: Rhino and jEdit where the main characteristics are described in Subsection III.C.

A. Variable Selection and Study Design

The main independent variable of our study is the type of splitting algorithm used: CamelCase, Samurai and Oracle (i.e., manually split identifiers).

The second independent variable is the use of dynamic information. Thus, we have two FLTs, and each has three configurations, which depend on the identifier splitting technique (see Table 1). For example, $IR_{CamelCase}$, $IR_{Samurai}$, and IR_{Oracle} are the IR based feature location techniques that use LSI to compute similarities between queries and methods, after applying the CamelCase, the Samurai and the Oracle splitting algorithms on the identifiers from the methods and queries. Similarly $IR_{CamelCase}Dyn$, $IR_{Samurai}Dyn$ and $IR_{Oracle}Dyn$ were defined.

In order to compare which configuration of the FLTs is more accurate than another (i.e., $IR_{CamelCase}$ vs. $IR_{Samurai}$), we considered their *effectiveness measure* [21]. The effectiveness measure is the best rank (i.e., lowest rank) among all the methods from the gold set for a specific feature. Intuitively, the effectiveness measure quantifies the number of methods a developer has to examine from a list of ranked methods returned by the feature location technique, before she is able to locate a relevant method pertaining to the feature. Obviously, a technique that consistently places relevant methods towards the top of the ranked list (i.e., lower ranks) is more effective than a technique that contains relevant methods towards the middle or the bottom of the ranked list (i.e., higher ranks). In this analysis we focus on the scenario of finding just one relevant method, as opposed

to finding all relevant methods from the gold set, for two reasons. First, we are focusing on concept location, rather than impact analysis. Second, once a relevant method has been identified, it is much easier to find other related methods by following program dependencies from the relevant method, or by using other heuristics.

In literature, the identifiers that are split using CamelCase are referred as hard-words, whereas the identifiers split using Samurai or TIDIER are called soft-words. During our analysis, we treat the hard and soft words in the same way and we refer to them as split identifiers.

The dependent variable considered in our study is the effectiveness measure of the FLTs.

We aim at answering the following overarching question: *if we had a perfect technique for splitting identifiers, would it still help improve accuracy of FLTs?* We plan to answer this question by examining these more specific **r**esearch **q**uestions (RQ):

RQ$_1$: Does $IR_{Samurai}$ outperform $IR_{CamelCase}$ in terms of effectiveness?

RQ$_2$: Does $IR_{Samurai}$Dyn outperform $IR_{CamelCase}$Dyn in terms of effectiveness?

RQ$_3$: Does IR_{Oracle} outperform $IR_{CamelCase}$ in terms of effectiveness?

RQ$_4$: Does IR_{Oracle}Dyn outperform $IR_{CamelCase}$Dyn in terms of effectiveness?

Previous work [11, 14] compared the CamelCase, Samurai and TIDIER splitting algorithms in terms of their accuracy for correctly splitting identifiers. However, in our study we are addressing the impact that splitting algorithms have on feature location.

B. *Building an Oracle – "Perfect Splitter"*

The aim of the Oracle is to provide an exact identifier splitting into terms, and possibly mapping acronyms and contractions into terms or English words, thus building a reference dictionary to be used in subsequent feature location phases. Application dictionaries, collected identifiers and terms from comments, may contain thousands of words. Hence, manual verification and split is a tedious and error prone task. To simplify Oracle building we applied a multi-step strategy aiming at minimizing the manual effort. In the following subsections we report details of each step.

Step one – building software application dictionary. We parsed and extracted identifiers and comments from both Rhino and jEdit and created a dictionary for each system. During this step we also built an application specific identifier (or term) frequency table for Samurai. Following this preliminary step, we filtered some dictionary entries to reduce manual validation effort.

Step two – filtering concordant identifier split. For each dictionary entry we ran the CamelCase, Samurai and TIDIER splitters to locate the identifiers for which these three splitting algorithms were in agreement. TIDIER was configured with WordNet [1] dictionary, as well as with acronyms and abbreviations known to the authors. We used the Samurai global frequency table made available by Samurai authors [11], as well as a local frequency table estimated from the software application under analysis (see Step 1). Whenever the three splitting algorithms agreed on the identifier term subdivision, we considered this as a strong indication that the resulting split was actually correct. This assumption divided the dictionary into two sub-dictionaries: one on which the algorithms disagree and one where there is agreement among them. The sub-dictionary where the tools agreed was then manually inspected to make sure that no errors were present. For example, out of about 6,000 dictionary entries (or words) for Rhino, about 2,500 words were split in this phase with a minimum manual effort.

Step three – filtering discordant identifier split. We manually inspected the identifiers for which the three splitting algorithms did not agree, in order to provide the best splitting. Examples of identifiers from the Rhino dictionary are words such as *DToA*, *DCMPG* or *impdep2*. Most of identifiers were manually split in this step (including careful inspection of the source code to understand the exact context of those identifiers), but there was a reduced set where it was unfeasible to assign any evident meaning even after inspecting the source code. For example, about 120 Rhino dictionary entries fell into this category. Examples of such identifiers include short strings (e.g., *DT*, *i3* or *m5*) and cryptic identifiers (e.g., *P754*, *u00A0* or *zzz*).

During the Oracle building process, the authors validated the split identifiers following a consensus approach (i.e., one author proposed an identifier split, which was then verified and validated by a second author). In a few cases, disagreements were discussed among all the authors. We adapted this approach in order to minimize the bias and the risk of producing erroneous results. This decision was motivated by the complexity of identifiers, which capture developers' domain and solution knowledge, experience, personal preference, etc., thus, it is difficult to decode the true meaning of identifiers in some cases.

C. *Systems*

We conducted our evaluation on two open source Java systems, Rhino and jEdit, and we constructed four datasets from these two systems. The first system considered is Rhino[2], an open-source implementation of JavaScript written in Java. Rhino version 1.6R5 has 138 classes, 1,870 methods and 32K lines of code. Rhino implements the specifications of the European Computer Manufacturers Association (ECMA) Script[3]. We constructed two datasets from Rhino.

The first dataset is $Rhino_{Features}$ and contains 241 features extracted from the specifications. Each feature has a textual description that was used as a query in the evaluation. These descriptions correspond to sections of the ECMAScript specifications. Each feature also has a set of methods which are associated with the features (i.e., gold set). The gold sets were constructed using the mappings between the source code and the features, which were made available by Eaddy et al. [9]. These mappings[4] were produced by considering the

[1] http://wordnet.princeton.edu/

[2] http://www.mozilla.org/rhino/

[3] http://www.ecmascript.org/

[4] http://www.cs.columbia.edu/~eaddy/concerntagger/

Table 2 Summary of the four datasets used in the evaluation: name (number of features/issues), source of the queries and gold sets, and the type of execution information

Dataset (Size)	Queries	Gold Sets	Execution Information
$Rhino_{Features}$ (241)	Sections of ECMAScript	Eaddy et al.	Full Execution Traces
$Rhino_{Bugs}$ (143)	Bug title and description	Eaddy et al. (CVS)	N/A
$jEdit_{Features}$ (64)	Feature (or Patch) title and description	SVN	Marked Execution Traces
$jEdit_{Bugs}$ (86)	Bug title and description	SVN	Marked Execution Traces

sections of the ECMAScript specification as features, and associating them with software artifacts using the following prune dependency rule, created by Eaddy et al. [9]: "A program element is relevant to a concern if it should be removed, or otherwise altered, when the concern is pruned." These mappings were used in other research papers, such as [8, 9, 29]. Rhino is distributed with a suite of test cases, and each test case has a correspondence in the ECMAScript specification. We used these test cases to collect full traces for each of the features.

The second dataset collected is $Rhino_{Bugs}$ and contains 143 issue reports (i.e., bugs) that were collected from Bugzilla, the issue tracking system of Rhino[5]. Each bug from Bugzilla has a title and a description, and we used this information as queries in the evaluation. As in the $Rhino_{Features}$ dataset, we used the information made available by Eaddy et al. [9] to associate each bug with a set of methods from Rhino which are responsible for the bug (i.e., the gold set). Eaddy et al. extracted the mappings between bugs and source code by analyzing CVS commits. However, there was no association between the 143 issue reports and the test cases, hence, we did not collect any execution traces for this dataset.

The second system considered is jEdit[6], a popular open-source text editor written in Java. jEdit version 4.3 has 483 classes, 6.4K methods and 109K lines of code. We constructed two datasets from this system.

The first dataset is $jEdit_{Features}$ and consists of 64 issues (34 features and 30 patches) extracted from jEdit's issue tracking system[7]. The second dataset is $jEdit_{Bugs}$ and consist of 86 bug reports. We now describe some steps used for collecting additional information for these two datasets. We used the changes associated with the SVN commits between releases 4.2 and 4.3 to construct the gold sets. In addition, the SVN logs were parsed for issue identifiers which were matched against the issues from the tracking system. Similarly to the $Rhino_{Bugs}$ dataset, the title and description of these issues were used in the evaluation as queries. We used a tracer to generate marked traces, by executing jEdit and following the steps to reproduce from the issue description. For more details about the process of generating this dataset, and for the complete dataset, which includes queries and execution traces, please refer to our online appendix[8].

[5] https://bugzilla.mozilla.org/
[6] http://www.jedit.org/
[7] http://sourceforge.net/tracker/?group_id=588
[8] http://www.cs.wm.edu/semeru/data/icpc11-identifier-splitting/

Table 3 Descriptive statistics from datasets: number of methods in the gold set, number of methods in traces, and number of identifiers from corpora

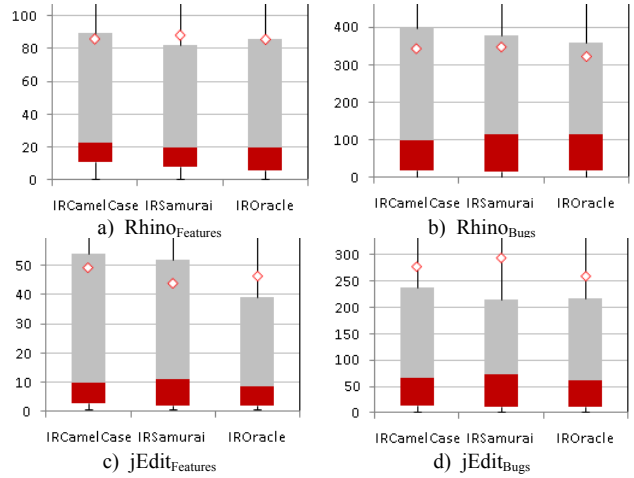
# of …	Measure	$Rhino_{Features}$	$Rhino_{Bugs}$	$jEdit_{Features}$	$jEdit_{Bugs}$
methods in gold set	min	1	1	1	1
	median	4	1	5	2
	average	12.82	2.24	6.3	4.01
	max	280	15	19	41
	st. dev	28.8	2.39	5.33	5.63
	total	3,089	320	403	345
unique methods from traces	min	777	N/A	227	227
	median	917	N/A	1.1K	1.1K
	average	912	N/A	1.1K	1.1K
	max	1.1K	N/A	1.9K	1.9K
	st. dev	54	N/A	310	310
identifiers in the corpus (with queries)	split by CamelCase	3,318 (4,154)	3,318 (4,223)	4,227 (4,361)	4,227 (4,596)
	split by Samurai	2,642 (3,416)	2,642 (3,411)	3,439 (3,552)	3,439 (3,751)
	Split by Oracle	2,030 (2,921)	2,030 (2,718)	2,758 (2,852)	2,758 (3,051)

The four datasets, extracted from Rhino and jEdit, which were used in the evaluation, are summarized in Table 2. We also present additional information about the datasets used in the evaluation in Table 3. First, we present details about the number of methods from the gold sets of each dataset. Each data point (i.e., a feature or a bug) from the $Rhino_{Features}$ dataset has on average 12 methods, whereas the $Rhino_{Bugs}$ dataset has only two methods on average. For jEdit there are on average four to six methods associated with each issue. The features from the $Rhino_{Featuress}$ dataset have many gold set methods in common, hence the total number of methods is much higher than for the other datasets.

Second, we present information about the number of methods extracted from the traces. For both systems, the average number of unique methods extracted from each trace was about one thousand. Third, we present information about the size of the corpora in terms of the number of identifiers, after applying the CamelCase, Samurai and Oracle splitting techniques. As expected, the more accurately we split the identifiers, the more we reduce the number of unique identifiers. For example, the corpus for $Rhino_{Features}$ has 3,318 identifiers after applying the CamelCase splitting technique, and has only 2,030 identifiers after using the Oracle splitting technique. This is explained by the fact that identifiers that could not be split by CamelCase formed an unique identifier, whereas the Oracle split the identifier into two or more (common) terms that already appear in the corpus, hence reducing the number of unique identifiers.

D. Analysis

For each dataset, every FLT will produce a list of ranks (i.e., effectiveness measures) that has the size of the number of features in the dataset. For example, the dataset $Rhino_{Features}$ produces 241 ranks for $IR_{CamelCase}$, 241 ranks for $IR_{Samurai}$ and 241 ranks for IR_{Oracle}, and each of those ranks represents the best position (i.e., lowest rank) of a method from the gold set associated with that feature. These lists of ranks are used as an input for the following comparison techniques: descriptive statistics, side by side comparisons, and statistical tests.



a) $Rhino_{Features}$ b) $Rhino_{Bugs}$ c) $jEdit_{Features}$ d) $jEdit_{Bugs}$

Figure 1 Box plots of the effectiveness measure of the three IR-based FLTs ($IR_{CamelCase}$, $IR_{Samurai}$ and IR_{Oracle}) for the four datasets: a) $Rhino_{Features}$, b) $Rhino_{Bugs}$, c) $jEdit_{Features}$ and d) $jEdit_{Bugs}$

First, we compare the ranks using descriptive statistics, such as minimum, first quartile, median, third quartile, maximum, and average. We present all these descriptive statistics graphically, using box plots (i.e., whisker charts). Although this technique provides a quick and intuitive view of the data, it only presents a high level perspective.

The second comparison technique examines the data in more details and works as follows. Given two lists of ranks produced by two different FLTs, we compare the ranks side by side and we count the number of cases the first technique produces lower ranks than the other, as well as the number of cases the second technique produces lower ranks (i.e., better results) than the other. We report these values as percentages.

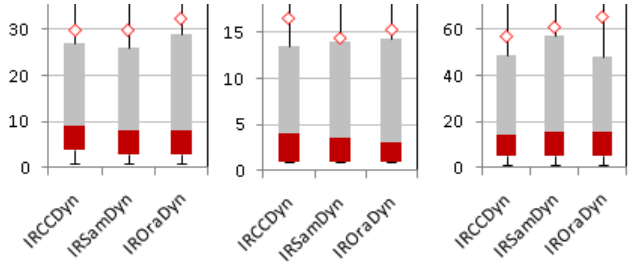
The third comparison of the ranks is a statistical analysis. We use the Wilcoxon signed-rank test [5] to test whether the difference in terms of effectiveness for two measures is statistically significant or not. This test is non-parametric and it takes as an input two lists of ranks produced by two different feature techniques. In the test we used a significance level $\alpha = 0.05$, and the output of the test is a p-value, which can be interpreted as follows. If the p-value is less than α, then the difference in ranks produced by one feature location technique is statistically significantly lower than the ranks produced by the other technique. Otherwise, if the p-value is larger than α, then we conclude that the two techniques produce almost equivalent results.

E. Hypotheses

We formulate several null hypotheses in order to test whether an improved splitting algorithm has a higher effectiveness measure than a simple splitting algorithm. For example:

$H_{0,IRSamurai}$ There is no statistical significant difference in terms of effectiveness between $IR_{Samurai}$ and $IR_{CamelCase}$.

$H_{0,IRSamuraiDyn}$ There is no statistical significant difference in terms of effectiveness between $IR_{Samurai}Dyn$ and $IR_{CamelCase}Dyn$.



a) $Rhino_{Features}$ b) $jEdit_{Features}$ c) $jEdit_{Bugs}$

Figure 2 Box plots of the effectiveness measure of the 3 FLTs ($IR_{CamelCase}Dyn$ ($IR_{CC}Dyn$), $IR_{Samurai}Dyn$ ($IR_{Sam}Dyn$) and $IR_{Oracle}Dyn$ ($IR_{Ora}Dyn$)) for the 3 datasets: a) $Rhino_{Features}$, b) $jEdit_{Features}$ and c) $jEdit_{Bugs}$

We also define several alternative hypotheses for the case when a null hypothesis is rejected with high confidence. These alternative hypotheses state that an improved identifier splitting technique (e.g., Samurai, Oracle) would produce higher effectiveness than the baseline splitting technique (i.e., CamelCase). The following alternative hypotheses correspond to the null hypotheses defined above.

$H_{A,IRSamurai}$ $IR_{Samurai}$ has statistically significantly higher effectiveness than $IR_{CamelCase}$.

$H_{A,IRSamuraiDyn}$ $IR_{Samurai}Dyn$ has statistically significantly higher effectiveness than $IR_{CamelCase}Dyn$.

The corresponding null and alternative hypotheses for the Oracle splitting technique are defined analogously.

IV. Results and Discussion

This section presents the effectiveness measures of the FLTs presented in Table 1, which were applied on the four datasets (see Table 2) extracted from Rhino and jEdit. Please refer to our online appendix for complete data.

Figure 1 presents the box plots of the effectiveness measures of the three IR based FLTs applied on the four datasets. For each dataset, all the instances of the IR feature location technique produce very similar results in terms of lower quartile, median, mean, upper quartile, etc. For example, Figure 1(a) shows that for the $Rhino_{Features}$ dataset, using the CamelCase splitting ($IR_{CamelCase}$) we obtain a median of 23 and an average of 86, and if we use the Oracle splitting (IR_{Oracle}), we obtain a median of 20 and an average of 86. The same small differences between the descriptive statistics measures are observed among all the IR instances, and in all the four datasets.

Similarly to Figure 1, Figure 2 presents the box plots of the effectiveness measure of the three IRDyn FLTs which were applied on the following three datasets: $Rhino_{Features}$ (Figure 2 (a)), $jEdit_{Features}$ (Figure 2 (b)) and $jEdit_{Bugs}$ (Figure 2 (c)). For all the datasets, the three FLTs produce almost identical results, regardless of the technique used for splitting the identifiers. For example, Figure 2(a) shows that for the $Rhino_{Features}$ dataset, using CamelCase splitting ($IR_{CamelCase}Dyn$), the median and average are 9 and 30 respectively, whereas for Oracle splitting ($IR_{Oracle}Dyn$) the median and average are 8 and 32 respectively. The small differences observed on the IR based instances are also observed here. Even more so, for the other datasets, when incorporating dynamic information the differences produced by the feature location techniques seem to be less noticeable

Table 4 Percentages of times the effectiveness of the FLT from the row is higher than $IR_{CamelCase}$, and vice-versa (see percentages from parenthesis)

FLT	$Rhino_{Features}$	$Rhino_{Bugs}$	$jEdit_{Features}$	$jEdit_{Bugs}$
$IR_{Samurai}$	39 (40)	36 (48)	33 (36)	41 (41)
IR_{Oracle}	49 (33)	45 (48)	44 (38)	40 (55)

Table 5 Percentages of times the effectiveness of the FLT from the row is higher than $IR_{CamelCase}Dyn$, and vice-versa (percentages from parenthesis)

FLT	$Rhino_{Features}$	$Rhino_{Bugs}$	$jEdit_{Features}$	$jEdit_{Bugs}$
$IR_{Samurai}Dyn$	33 (36)	N/A	27 (22)	28 (41)
$IR_{Oracle}Dyn$	42 (35)	N/A	34 (22)	35 (50)

than the differences produced by IR-based feature location techniques. This fact may suggest that dynamic information has some influence and the splitting techniques used for identifiers may not be as important. It is also interesting to observe that feature location techniques applied on the datasets that use features as queries (i.e., $Rhino_{Features}$ and $jEdit_{Features}$) have lower effectiveness measures than the feature location techniques applied on the datasets that use bug descriptions as queries. For example, for Rhino, the median effectiveness when using feature descriptions as queries is about 21 (see Figure 1(a)), whereas the median effectiveness when using bug descriptions as queries is about 110 (see Figure 1(b)). The same observation is valid for the jEdit when only textual information is used (see Figure 1(c)(d)) as well as when textual and execution information are combined (see Figure 2(b)(c)).

The results illustrated in Figure 1 and Figure 2 provide only a high level picture of the effectiveness measure. We now present results from a case by case comparison of the effectiveness measure. Table 4 presents the percentage of times an instance of the IR based FLT produces lower ranks than another instance of the IR based FLT. The first cell value represents the percentage of times the FLT from the corresponding row produces lower ranks than $IR_{CamelCase}$, whereas the number in parenthesis represents the percentage of times $IR_{CamelCase}$ produces lower ranks than the technique from the row (in the remaining percentages, the two techniques produce identical ranks). In this case, a higher percentage denotes a more effective technique. Similarly, Table 5 shows the percentage of times the FLT form the row produces better results than $IR_{CamelCase}Dyn$.

We observe from Table 4 that comparing the effectiveness measures of IR_{Oracle} and $IR_{CamelCase}$ side by side, IR_{Oracle} produces lower ranks in 49% of cases, whereas $IR_{CamelCase}$ produces better results in 33% of cases. In the remaining 18% of cases (i.e., 100%-49%-33%) the two techniques produce identical ranks.

Similarly, from Table 5 we observe that when dynamic information is taken into account, for the $Rhino_{Features}$ dataset, $IR_{Oracle}Dyn$ produces lower ranks (i.e., better results) in 42% of cases, whereas $IR_{CamelCase}Dyn$ produces better results in 35% of cases. In the remaining 23% of cases (i.e., 100%-42%-35%) the techniques produce the same results.

It is interesting to observe that for both systems, IR_{Oracle} and $IR_{Oracle}Dyn$ produce a higher percentage of good results than $IR_{CamelCase}$ and $IR_{CamelCase}Dyn$ respectively, when these techniques are applied on the datasets that use features as queries (see columns two and four of the last rows of Table 4 and Table 5). However, when these techniques are applied

Table 6 The p-values of the Wilcoxon signed-rank test for the FLT from the row compared with $IR_{CamelCase}$ (statistical significance values are **bold**).

FLT	$Rhino_{Features}$	$Rhino_{Bugs}$	$jEdit_{Features}$	$jEdit_{Bugs}$
$IR_{Samurai}$	0.692	0.890	0.742	0.479
IR_{Oracle}	**0.005**	0.497	0.202	0.785

Table 7 The p-values of the Wilcoxon signed-rank test for the FLTs from the row compared with $IR_{CamelCase}Dyn$ (there are no stat. significant values)

FLT	$Rhino_{Features}$	$Rhino_{Bugs}$	$jEdit_{Features}$	$jEdit_{Bugs}$
$IR_{Samurai}Dyn$	0.713	N/A	0.307	0.928
$IR_{Oracle}Dyn$	0.265	N/A	0.095	0.937

on the datasets that use bug description as queries, the opposite phenomenon is observed. In other words $IR_{CamelCase}$ and $IR_{CamelCase}Dyn$ produce higher percentage of good results than IR_{Oracle} and $IR_{Oracle}Dyn$ respectively (see columns three and five of the last rows of Table 4 and Table 5).

The effectiveness measures presented as box plots and percentages are statistically analyzed using the Wilcoxon signed-rank test. Table 6 presents the p-values of the Wilcoxon signed-rank test for all the instances of the IR-based FLTs. The results that are statistically significant (i.e., the p-value is lower than $\alpha = 0.05$) are highlighted in bold. The table shows that there is only one instance when the Oracle splitting technique (i.e., IR_{Oracle}) produces results that are statistically significantly better than the technique that uses CamelCase splitting (i.e., $IR_{CamelCase}$). This is for the $Rhino_{Features}$ dataset and the p-value is equal to 0.005. We performed the same analysis between IR_{Oracle} and $IR_{Samurai}$ and the results show that only for the $Rhino_{Features}$ dataset IR_{Oracle} produces results that are statistically significantly better than $IR_{Samurai}$ (p-value=0.009). Refer to our online appendix for the data.

Similarly, Table 7 shows the p-values of the Wilcoxon signed-rank test applied on the effectiveness measures produced by the IRDyn FLTs. The results show that no technique produces statistically better results than any other technique. This observation helps in answering the research questions RQ2 and RQ4, that the splitting technique used is not as important if dynamic information is considered. Refer to our online appendix for the results comparing $IR_{Oracle}Dyn$ and $IR_{Samurai}Dyn$. When dynamic information is involved, no technique produces statistically significant results than the other for any of the datasets.

If we look at the same results (i.e., the effectiveness measure) from three different points of view (i.e., box plots, percentages and statistical analysis), we derive the following conclusions. First, there are instances where a better identifier splitting technique (i.e., Oracle) improves feature location. This has been the case for the Rhino, for the $Rhino_{Features}$ dataset. Second, there are cases when even a perfect identifier splitting technique cannot help in the process of feature location. Such an example is given by the $jEdit_{Features}$ dataset, when the effectiveness measure is improved for a few cases, but the difference is not statistically significant. Moreover, there are instances where the perfect splitting technique can have negative impact on feature location, as it was the case for the $jEdit_{Bugs}$ dataset. In this case, the original CamelCase splitting technique produced better results than the Oracle in terms of percentages (see Table 4), but the difference is still not

Table 8 Examples of splitted identifiers from Rhino using CamelCase and Samurai. The identifiers which are split correctly are highlighted in bold

Original Identifier	CamelCase	Samurai
GETPROP	getprop	**GET PROP**
readadapterobject	readadapterobject	**read adapter object**
SHORTNUMBER	shortnumber	**SHORT NUMBER**
debugAccelerators	**debug accelerators**	debug Ac ce le r at o rs
tolocale	tolocale	tol ocal e
imitating	**imitating**	imi ta ting

statistically significant. Finally, there is one instance, $Rhino_{Features}$, where splitting helps when textual information is used. However, when dynamic information is used, all the splitting techniques produce equivalent results from a statistical point of view.

A. *Qualitative Results*

This section presents some observations after examining the results produced by the splitting techniques and after examining the queries.

One of the problems that we encountered using Samurai was that it tended to split certain types of identifiers into many meaningless terms, some of them having between one-three characters. Examples of identifiers from Rhino, where Samurai split them incorrectly were: *debugAccelerators*, *tolocale*, *imitating*, *imlementation*, etc. Their incorrect Samurai splitting was: *debug Ac ce le r at o rs*, *tol ocal e*, *imi ta ting*, *i ml eme n tat ion* (see Table 8). For these examples, CamelCase performed better, as it correctly split the first identifier (*debug accelerators*), but it left the other ones unaltered. Please refer to our online appendix for more information.

One of the benefits of using Samurai was that it accurately split same-case identifiers composed of multiple words. For these cases, CamelCase left the identifiers unmodified. Examples of such identifiers from Rhino include *SHORTNUMBER*, *readadapterobject*, *GETPROP* which are correctly split by Samurai as *SHORT NUMBER, read adapter object,* and *GET PROP*, and are left unchanged by CamelCase (see Table 8).

However, there were some cryptic identifiers that were almost impossible to split using CamelCase or Samurai. Examples of such identifiers from Rhino include *ldbl*, *njm*, *pun*, *rve*, *wbdry*, etc. In these cases, inferring the meaning from the context in which these identifiers appeared was the only way to split or expand them correctly.

We observed a vocabulary mismatch problem, which produced inconsistencies between the identifiers used in the queries, and the identifiers used in the code.

This problem seemed to be less noticeable for features, and more severe for bugs. For jEdit, the issues that described features often contained terms that were later used in the code as identifiers for classes, methods, variables, etc. For example, jEdit's feature #1608486[9] ("Support 'thick' caret"), contained in its description many identifiers that were also found in the name of the methods (e.g., *thick*, *caret*, *text*, *area*, etc.). For features, their queries were expressive, and more consistent with the source code vocabulary, so they benefitted less from an Oracle splitting. Hence, when using feature descriptions as queries for both Rhino and jEdit, the median effectiveness of the FLTs, regardless of splitting, were about 20 for Rhino (see Figure 1 (a)) and about 10 for jEdit (see Figure 1 (c)).

On the other hand, the vocabulary of the queries extracted from bug reports was less consistent with the source code vocabulary, and a splitting technique, helped bridge this gap. For example, jEdit's bug #1575505[10] ("C+j bug") reported a problem with the "join lines" implementation, yet nowhere in its description were the words *join* or *lines* mentioned. In general, the identifiers from the bug descriptions were less consistent with the code, and this issue was reflected in terms of the effectiveness measures produced by the FLTs, when these bug descriptions were used as queries. For example, in Figure 1 (b) the median effectiveness for Rhino system was about 110 (as opposed to a median of 20 when features were used as queries). Also, Figure 1 (d), shows that the median effectiveness of the techniques that used bugs as queries was around 67, as opposed to 10, which was the median effectiveness when features were used as queries.

Another problem with the queries is that some identifiers were used just for communication between developers, and no matter what splitting technique was used, these identifiers provided no useful information, because they appeared only in the query vocabulary, and did not appear at all in the source code vocabulary. Examples of such identifiers included words that are common in communication, such as *btw* (i.e., by the way), *thanks*, *hate*, *rant*, *greetings*, *fly*, *annoying*, etc., name of developers, *ApeHanger*, *Slava*, *Carlos*, etc.

B. *Threats to Validity*

In this section we present several threats to validity associated with the evaluation.

Threats to *construct validity* concern the relation between the theory and the observation. This threat is mainly due to mistakes in the Oracle and gold set. We cannot guarantee that no errors are present in the Oracle. As the intent of the Oracle is to explain identifier semantics, we cannot guarantee that some identifiers could have been split in different ways by developers that originally created them. This problem is difficult and it relates to guessing the developers' intent. To limit this threat, different sources of information such as comments, source code context, and online documentation were used when producing the Oracle. To minimize the risk on the accuracy of the gold set, we used data produced by other researchers, which was used in previous studies and made available to the research community.

Threats to *internal validity* concern any confounding factors that could have influenced our study results. In particular, these threats are due to the subjectivity of the manual building of the Oracle and to the possible biases introduced by manually splitting identifiers. To limit this threat, the Oracle was produced by a joint work among the authors, using CamelCase, Samurai and TIDIER. In addition,

[9] http://tinyurl.com/4ne9u9v

[10] http://tinyurl.com/64wonla

inconsistencies in splitting/mapping to dictionary words were discussed.

Threats to *conclusion validity* concern the relations between the treatment and the outcome. Proper tests were performed to statistically reject the null hypotheses. In particular, we used a non-parametric test (i.e., Wilcoxon signed-rank test), which does not make any assumptions on the underlying distributions of the data. Furthermore, as the only significant p-value is 0.005 (see Table 6), even with the conservative Bonferroni correction, it will remain significant as the limit in such case is equal to α-value/number of tests (i.e., 0.05 / 3 = 0.01666 < 0.05).

Threats to *external validity* concern the possibility of generalizing our results. To make our results as generalizable as possible, we used two Java applications from two different application domains but we cannot be sure that our findings will be valid for other domains, applications, programming languages or software engineering tasks (i.e., different from feature location). More case studies are needed to confirm the results presented and to verify if indeed, in the general case, dynamic information reduces the gain of more sophisticated identifier split techniques.

V. Related Work

Given the paramount role of source code identifiers in maintenance tasks such as traceability link recovery or feature and concept location, a large body of relevant work is available in this area. We divided this section into the related work on the role of unstructured information in program comprehension and approaches to feature location.

A. The Role of Unstructured Information in Program Comprehension

Takang et al. [34] attempted to determine the informativeness of identifiers. They conducted experiments to compare abbreviated identifiers to full-word identifiers and uncommented code to commented code. Their study results showed that commented programs are more understandable than non-commented programs and that programs containing full-word identifiers are more understandable than those with abbreviated identifiers.

Lawrie et al. [18, 19] have performed an empirical study to assess the quality of source code identifiers. Their study with over 100 programmers indicated that full words as well as recognizable abbreviations lead to better comprehension. Lawrie et al. [17] introduced GenTest, a splitting algorithm which by incorporating vocabulary normalization is able to outperform Samurai.

Binkley et al. [3] have investigated the use of different identifier separators in program comprehension. They found that the CamelCase convention led to better understanding than underscores and, when subjects are properly trained, they performed faster with identifiers in the CamelCase style rather than identifiers built using underscores. Binkley et al.'s study was replicated by Sharif and Maletic [31] using an eye tracking system. Their results indicate that there was no difference in terms of accuracy between the CamelCase and underscore style, and that subjects recognized identifiers that used the underscore notation more quickly.

Caprile and Tonella [4] have analyzed the internal structure of identifiers. Their in-depth analysis showed that identifiers are one of the most important source of information about system concepts, and that the information carried by identifiers is often the starting point for program comprehension.

Deißenböck and Pizka [7] have provided guidelines for the production of high-quality identifiers. With such guidelines, identifiers should contain enough information for an engineer to understand the program concepts.

B. Related Work on Feature Location

Marcus et al. introduced LSI-based feature location technique [25]. This approach was later extended to include the Rocchio algorithm for relevance feedback [12] by allowing developers to reformulate search queries. Grant et al. [13] used Independent Component Analysis for feature location, by separating the features (modeled as input signals) into independent components and estimating the relevance to each source code method. Shepherd et al. [32] proposed an approach to feature location that is based on the program model that captures action-oriented relations between identifiers in a program.

There are several FLTs that use more than one type of information (or underlying analysis). For example, SITIR [21] and PROMESIR [27] both utilize textual and execution information. Eisenbarth et al. [10] proposed a technique that applies formal concept analysis to traces to generate a mapping between features and methods. Cerberus [8] is another hybrid technique which combines static, dynamic and textual analysis. The up-to-date summary of feature location approaches can be found in [29].

VI. Conclusions

Perfecting splitting techniques can improve the accuracy of feature location, easing program comprehension and thus, software evolution. In situations where execution information cannot be collected (e.g., mission critical and time critical applications), the benefits of using advanced splitting techniques can be mostly visible. In fact, by splitting source code identifiers and mapping them to domain concepts, the localisation of entities contributing to implementing some user observable functionality may be easier, which could minimize feature location effort.

In this paper, we presented an exploratory study of two FLTs (i.e., IR and IRDyn) for locating bugs and features, utilizing three strategies for splitting identifiers: CamelCase, Samurai and manual splitting of identifiers. These FLTs and their preprocessing techniques were evaluated on two open-source systems, Rhino and jEdit, and compared in terms of their effectiveness measure.

The results of the IR-based FLT reveal that Samurai and CamelCase produce similar results. However, the IR_{Oracle} outperforms $IR_{CamelCase}$ in terms of the effectiveness measure, on the $Rhino_{Features}$ dataset. This supports our conjecture that when only textual information is available, an improved splitting technique can help improve effectiveness of feature location. The results also show that when both textual and execution information are used, any splitting algorithm will

suffice, as FLTs produce equivalent results. In other words, because execution information helps pruning the search space considerably, the benefit of an advanced splitting algorithm is comparably smaller than the benefit obtained from execution information; hence the splitting algorithm will have little impact on the final results. Overall, our findings outline potential benefits of creating advanced preprocessing techniques as they can be useful in situations where execution information cannot be easily collected.

Future work will extend evaluating feature location techniques on other software systems. In addition, we plan to apply the splitting strategy namely, TIDIER, which is based on the use of contextual-aware dictionaries and specialized knowledge, as well as GenTest, which uses vocabulary normalization.

ACKNOWLEDGMENT

This work is supported by NSF CCF-0916260 and NSF CCF-1016868 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

REFERENCES

[1] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E., "Recovering Traceability Links between Code and Documentation", *IEEE Transactions on Software Engineering*, vol. 28, no. 10, October 2002, pp. 970 - 983.

[2] Antoniol, G., Gueheneuc, Y.-G., Merlo, E., and Tonella, P., "Mining the Lexicon Used by Programmers during Software Evolution", in Proc. of 23rd IEEE ICSM'07, Paris, France, 2007, pp. 14-23.

[3] Binkley, D., Davis, M., Lawrie, D., and Morrell, C., "To CamelCase or Under_score", in Proc. of 17th IEEE ICPC'09, Vancouver, British Columbia, Canada, May 17-19 2009, pp. 158-167.

[4] Caprile, C. and Tonella, P., "Nomen Est Omen: Analyzing the Language of Function Identifiers", in Proc. of 6th IEEE WCRE'99, Atlanta, Georgia, USA, 6-8 October 1999, pp. 112-122.

[5] Conover, W. J., *Practical Nonparametric Statistics*, Third Edition, Wiley, 1998.

[6] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R., "Indexing by Latent Semantic Analysis", *Journal of the American Society for Information Science*, vol. 41, 1990, pp. 391-407.

[7] Deissenboeck, F. and Pizka , M., "Concise and Consistent Naming", *Software Quality Journal*, vol. 14, no. 3, 2006, pp. 261-282

[8] Eaddy, M., Aho, A. V., Antoniol, G., and Guéhéneuc, Y. G., "CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis", in Proc. of 16th IEEE ICPC'08, Amsterdam, The Netherlands, 2008, pp. 53-62.

[9] Eaddy, M., Zimmermann, T., Sherwood, K., Garg, V., Murphy, G., Nagappan, N., and Aho, A. V., "Do Crosscutting Concerns Cause Defects?", *IEEE Transaction on Software Engineering*, vol. 34, no. 4, July-August 2008, pp. 497-515.

[10] Eisenbarth, T., Koschke, R., and Simon, D., "Locating Features in Source Code", *IEEE Transactions on Software Engineering*, vol. 29, no. 3, March 2003, pp. 210 - 224.

[11] Enslen, E., Hill, E., Pollock, L., and Vijay-Shanker, K., "Mining Source Code to Automatically Split Identifiers for Software Analysis", in Proc. of 6th IEEE MSR'09, Vancouver, Canada May 16-17 2009, pp. 71-80.

[12] Gay, G., Haiduc, S., Marcus, M., and Menzies, T., "On the Use of Relevance Feedback in IR-Based Concept Location", in Proc. of 25th IEEE ICSM'09, Edmonton, Canada, September 2009, pp. 351-360.

[13] Grant, S., Cordy, J. R., and Skillicorn, D. B., "Automated Concept Location Using Independent Component Analysis ", in Proc. of 15th WCRE'08, Antwerp, Belgium, October 15-18 2008, pp. 138-142.

[14] Guerrouj, L., Di Penta, M., Antoniol, G., and Guéhéneuc, Y.-G., "TIDIER: An Identifier Splitting Approach using Speech Recognition Techniques", *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, vol. to appear, 2011.

[15] Haiduc, S. and Marcus, A., "On the Use of Domain Terms in Source Code", in Proc. of 16th IEEE ICPC'08, Amsterdam, The Netherlands, June 10-13 2008, pp. 113-122.

[16] Hill, E., Pollock, L., and Vijay-Shanker, K., "Automatically Capturing Source Code Context of NL-Queries for Software Maintenance and Reuse", in Proc. of 31st IEEE/ACM ICSE'09, May 16-24 2009.

[17] Lawrie, D., Binkley, D., and Morrell, C., "Normalizing Source Code Vocabulary", in Proc. of 17th IEEE WCRE'10, Beverly, Massachusetts, USA, October 13-16 2010, pp. 3-12.

[18] Lawrie, D., Morrell, C., Feild, H., and Binkley, D., "What's in a Name? A Study of Identifiers", in Proc. of 14th IEEE ICPC'06, Athens, Greece, June 14-16 2006, pp. 3-12.

[19] Lawrie, D., Morrell, C., Feild, H., and Binkley, D., "Effective Identifier Names for Comprehension and Memory", *Innovations in Systems and Software Engineering*, vol. 3, no. 4, 2007, pp. 303-318.

[20] Levenshtein, V. I., "Binary Codes Capable of Correcting Deletions, Insertions, and Reversals", *Cybernetics and Control Theory*, vol. 10, no. 8, 1966, pp. 707-710.

[21] Liu, D., Marcus, A., Poshyvanyk, D., and Rajlich, V., "Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace", in Proc. of 22nd IEEE/ACM ASE'07, Atlanta, Georgia, November 5-9 2007, pp. 234-243.

[22] Lukins, S. K., Kraft, N. A., and Etzkorn, L. H., "Bug localization using Latent Dirichlet Allocation", *Information and Software Technology*, vol. 52, no. 9, 2010, pp. 972-990.

[23] Maletic, J. I. and Marcus, A., "Supporting Program Comprehension Using Semantic and Structural Information", in Proc. of 23rd ICSE'01, Toronto, Ontario, Canada, May 12-19 2001, pp. 103-112.

[24] Marcus, A., Maletic, J. I., and Sergeyev, A., "Recovery of Traceability Links Between Software Documentation and Source Code", *International Journal of Software Engineering and Knowledge Engineering*, vol. 15, no. 4, October 2005, pp. 811-836.

[25] Marcus, A., Sergeyev, A., Rajlich, V., and Maletic, J., "An Information Retrieval Approach to Concept Location in Source Code", in Proc. of 11th IEEE WCRE'04, Delft, The Netherlands, November 9-12 2004, pp. 214-223.

[26] Porter, M., "An Algorithm for Suffix Stripping", *Program*, vol. 14, no. 3, July 1980, pp. 130-137.

[27] Poshyvanyk, D., Guéhéneuc, Y. G., Marcus, A., Antoniol, G., and Rajlich, V., "Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval", *IEEE Transactions on Software Engineering*, vol. 33, no. 6, June 2007, pp. 420-432.

[28] Ratiu, D. and Deissenboeck, F., "From Reality to Programs and (Not Quite) Back Again", in Proc. of 15th IEEE ICPC'07, Banff, Canada, 2007, pp. 91-102.

[29] Revelle, M., Dit, B., and Poshyvanyk, D., "Using Data Fusion and Web Mining to Support Feature Location in Software", in Proc. of 18th IEEE ICPC'10, Braga, Portugal, June 30 - July 2 2010, pp. 14-23.

[30] Revelle, M. and Poshyvanyk, D., "An Exploratory Study on Assessing Feature Location Techniques", in Proc. of 17th IEEE ICPC'09, Vancouver, British Columbia, Canada, May 17-19 2009, pp. 218-222.

[31] Sharif, B. and Maletic, J. I., "An Eye Tracking Study on camelCase and under_score Identifier Styles", in Proc. of 18th IEEE ICPC'10, Braga, Portugal, June 30 - July 2 2010, pp. 196-205.

[32] Shepherd, D., Fry, Z., Gibson, E., Pollock, L., and Vijay-Shanker, K., "Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns", in Proc. of 6th AOSD'07, 2007, pp. 212-224.

[33] Soloway, E., Bonar, J., and Ehrlich, K., "Cognitive Strategies and Looping Constructs: An Empirical Study", *Communications of the ACM*, vol. 26, no. 11, November 1983.

[34] Takang, A., Grubb, P., and Macredie, R., "The Effects of Comments and Identifier Names on Program Comprehensibility: An Experimental Investigation", *Journal of Programming Languages*, vol. 4, no. 3, 1996, pp. 143-167.

[35] Von Mayrhauser, A. and Vans, A. M., "Program Comprehension During Software Maintenance and Evolution", *Computer*, vol. 28, no. 8, 1995, pp. 44-55.