Where Should We Look? An Empirical Study on the Locations of Abbreviation Expansions in Source Code

***Abstract*—Due to the prevalence of abbreviations in source code, the inclusion of accurate abbreviation expansions can increase the quality of software engineering tools that rely on natural language. Unfortunately, state-of-the-art expansion techniques struggle to reach an average expansion accuracy of 59-62%. To help address this problem, we manually collected more than 900 abbreviations and their expansions. We then empirically studied 5 open source software systems by automatically analyzing the source code source code (e.g., identifiers, comments), system documentation, language documentation, a computer science dictionary, and an English dictionary; looking for where these manually-obtained abbreviations and expansions tended to occur most frequently. This paper has two main contributions: 1) It is the first paper to present an empirical analysis of where abbreviations and expansions occur in source code and natural language software artifacts. 2) The results of this work can be used to specialize information retrieval or matching algorithms used in abbreviation expansion tools to prioritize different software artifacts when searching for candidate expansions.**

Keywords— Program Comprehension, abbreviation expansion, software maintenance, software evolution

I. Introduction

Researchers frequently use natural language processing techniques to analyze source code for numerous activities including bug detection [1], mining and analyzing software repositories [2], automated documentation [3], topic modeling [4], feature location [5] and more. Because so many techniques rely on the quality of natural language processing tools, it is important that they perform well even in the face of the often imperfect and incomplete language used in source code [6]. However, tools for Natural Language Processing (NLP) of source code are still inaccurate [6]–[10]. These inaccuracies have consequences [11] and can have downstream impacts in further natural language analyses like sentiment [11], part of speech tagging [6], [12]–[20], source code summarization [3], [21]–[23], and identifier splitting [5], [24], [25].

One way to increase the accuracy of natural language analyses and improve the comprehensibility of source code is by expanding abbreviations [26]–[28]. For example, the abbreviation 'cfg' can have multiple possible expansions in source code, such as configure, configuration, control flow graph, or context free grammar. Understanding an abbreviation's expansion can affect the accuracy of subsequent natural language analyses like part of speech tagging or sentiment analysis, as well as help developers better understand the meaning of source code identifiers.

Existing techniques that search for abbreviation expansions average 59-62% accuracy [7], [29], [30] . A critical step in improving the accuracy of abbreviation expansion techniques is in understanding *where* different types of abbreviations and their expansions occur, both in terms of the type of software artifact (e.g., system documentation) and the type of location in source code (e.g., type names, function names). Studying these locations and the types of abbreviations they contain will help guide future techniques by 1) revealing the most lucrative locations for expansions, 2) identifying what kind of abbreviations/expansions likely to occur in different artifacts, and 3) providing some clue of how difficult it will be to find/match these expansions.

In this paper, we empirically analyze the frequency at which abbreviation expansions appear in varying types of software artifacts (e.g., source code, documentation, etc.) by examining where abbreviations and their expansions occur in 5 open source software systems. Additionally, to obtain the abbreviations and expansions we use in our study, we collected over 900 abbreviation-expansion pairs across all 5 systems. We address the following research questions:

RQ1: Across all artifacts, where do abbreviation expansions most frequently occur in general? We will use this question to understand what our data says about the location of expansions. Our goal will be to understand where most expansions are located, if any of the locations we study have unique expansions; expansions that are not found in any other location, and whether/how often expansions occur near to their abbreviation. The answer to this question will help future expansion techniques by revealing the most lucrative locations for expansions.

RQ2: Do words that make up abbreviation expansions typically occur adjacent to one another? After looking at the distribution of abbreviations and expansions across multiple software artifacts, we want to get a better understanding of how difficult it would be to find these expansions. Multi-word abbreviation expansions (i.e., expansions that contain two or more words) whose constituent terms frequently appear next to one another (i.e., they are adjacent) are easier to find. We want to know if there is a pattern to where non-adjacent expansions tend to occur; these locations will need more rigorous approaches to determine if an abbreviation represents a multi-word expansion and then determine what that expansion is.

RQ3: Do expansions for abbreviations of varying type occur in some artifacts more often than others? This question will look specifically at different types of abbreviations. This contrasts with RQ1, which looks at expansions exclusively. There are different types of abbreviations; taxonomies have been published in previous work [29], which we will discuss. Some types of abbreviations are harder to detect/expand than others. This question will help us understand where different types of abbreviations are more likely to occur and will help future expansions techniques by informing them of what types of abbreviations are more frequent and should be prioritized in different locations.

RQ4: How can an approach to expanding abbreviations use what we have learned? The last question ties everything from the previous three RQs together. The goal of this question is to put the insights from previous questions together such that we will have a big-picture view of how all of the data in this paper can be synergized to improve future abbreviation expansion techniques.

The results of these research questions can directly impact abbreviation expansion techniques by providing am empirical, data-driven, answer to what tools can expect when trying to find expansions for abbreviations in different locations whether these locations be software artifacts or the source code itself. The rest of the paper is organized as follows: Section II discusses non-dictionary words, Section III discusses our research methodology, Section IV presents the data from our empirical study and answers our research questions, Section V contains related work and Section VI concludes.

Table I. Intent-based Non-Dictionary Categorizations

Category	Definition	Example
Abbreviation	Word that is the shortened form of a larger word or phrase.	See Table II
Alias	Word that represents a concept but is not a single or multi word abbreviation of that concept.	H - Cofactor Q – Prime X – Independent Variable These letters are not shortened versions of the concepts they represent.
Distinguisher	Word which does not represent a concept. Its only purpose is to avoid name-collision at compile-time.	int x, x1, x2; The numbers 1 and 2 are used as distinguishers.

II. Non-Dictionary Words in Software

Abbreviations form a subset of non-dictionary words in source code and software artifacts. A non-dictionary word is a word that has no dictionary definition; they are typically shortened versions of dictionary words (i.e., abbrev is a non-dictionary word that is short for abbreviation, which is a dictionary word). A taxonomy of non-dictionary words has been published previous [29], and the accuracy of current approaches to expanding abbreviations vary depending on the category of the abbreviations they are expanding [7][29]. In this section, we explain this previous taxonomy along with two complementary categories we empirically derived. We also discuss previously-published approaches to abbreviation expansion, to help the reader understand state-of-the-art techniques.

A. *Categorizations of non-dictionary words*

Programming languages use natural language to convey meaning to humans. As a professional developer writes code, they ultimately construct identifiers, comments, and documentation out of words that indicate the role or behavior of a part of the software they are constructing. As the creation of documentation and identifier names is a partially subjective activity, developers are free to choose what words to use. Some of the words developers choose to use are non-dictionary words.

While collecting data for our study, we noticed recurring patterns in the way developers use non-dictionary words. We show a set of categories that summarize these patterns in Table I. There are three categories: Abbreviation, Alias, and Distinguisher. These categories capture the relationship between the syntax (i.e., the form) of the abbreviation and its intent (i.e., what it represents):

- An *abbreviation* is a shortened version of a word or phrase. Abbreviations generally begin with the same letter as the word or phrase it was shortened from. For example, 'AES' is an abbreviation of *Advanced Encryption Standard*.
- An *alias* represents a concept but is not a shortened version of any dictionary word related to that concept. Aliases are syntactically less strict than abbreviations. For example, the mathematical variable 'K' is often used in the Singular Value Decomposition (SVD) technique to refer to the dimensionality of the resulting matrix. In this case, K is an alias because it is not a shortening of any word or phrase that describes the function of K in the context of SVD. Another example comes from the Digital Signature Algorithm (DSA), which uses variables such as R and S to represent components of the encrypted signature. These variables are not shortenings of some larger word, but they represent a clear concept in the context of the DSA.
- A *distinguisher* is a word that is attached to an identifier to keep the compiler from registering a name collision, which is when two names in the same scope are syntactically equivalent. A distinguisher is not a syntactic shortening of any word and does not represent a concept in any form. Its only purpose is to avoid a name collision. An example would be if we have two variables named integer1 and integer2. The numbers provide no additional information about what the variables are doing and are not shortenings of any word. Adding them as a suffix simply allows the program to compile and distinguish the variables from one another.

This is not the first work to categorize abbreviations. Previous work by Hill, et al. [29] categorizes abbreviations based solely on an abbreviation's syntax. They broadly categorize abbreviations as Single Word and Multi Word abbreviations. Each of these breaks into two sub-categories. For single word, these are *prefix* and *dropped letter*. For multi word, these are *acronym* and *combination multi word*. For clarity, we show Hill's abbreviation categories in Table II. The categorizations presented in Table I are complementary to these, as Hill, et al.'s categorizations are a more fine-grained subset of the abbreviation category.

Table II. Syntax-Based Non-Dictionary Word Categorizations

Category	Abbreviation Type	Definition	Example
Single Word	*Prefix*	Abbreviation of a single word that is strictly a prefix of the full word; formed by dropping letters from the end of the full word	Pub → Public Attr → Attribute Abbrev → Abbreviation
	Dropped Letter	Abbreviation of a single word that is formed by dropping letters from anywhere within the full word except the first letter	Cfg →Configure Ln → Line Tty → Teletype
Multi Word	*Acronym*	Abbreviation made from the first letters of multiple words.	Kv → Key value Ip → Internet protocol Vr → Virtual reality
	Combination Multi-word	Abbreviation made by dropping letters from multiple words	Oid → Object Identifier StdDev→Standard Deviation Arg → access rights

B. Expanding non-dictionary words

Expanding non-dictionary words and, in particular, expanding abbreviations, has been the topic of numerous research papers [7], [8], [29]–[31]. Expanding abbreviations is typically done as a part of word preprocessing alongside other steps such as splitting, stemming, tokenization, etc. For activities that involve information retrieval or natural language text analysis, abbreviation expansion adds more information to the corpus; giving these techniques more data to work with. The most recent techniques for abbreviation expansion take advantage of several information sources to try and detect correct expansions.

That is, they use multiple levels of software artifacts. For example, LINSEN [7] uses the following sources to find expansions: 1) Terms extracted from comments of the source file containing the current identifier, 2) terms extracted from comments of all source files in the software system, 3) dictionary of computer science programming terms, 4) an English dictionary. AMAP [29] worked in a very similar way by gradually expanding the search scope from very close to the abbreviation to less contextualized sources such as relative frequency of terms across a large set of open source systems. This approach works very well since some terms are not likely to show up in a comment within the code but are extremely likely to be in a computer science dictionary. Or vice versa; a system-specific abbreviation will more likely show up in a comment than in a computer science dictionary.

We note that the work presented here does not represent an independent expansion technique similar to those discussed above. Instead, results of the empirical analyses presented in this paper can be used to improve existing techniques such as LINSEN and AMAP. To the authors' knowledge, no other work presents an empirical analysis of abbreviations expansions.

III. Research Methdology

What sources of information are most effective for finding abbreviation expansions? Are there sources we should be focusing on more than others? For example, perhaps expansions for acronyms are (e.g., JSON) are more likely to show up in documentation for the system or programming language documentation (e.g., javadocs). In this case, approaches should search these locations for potential expansions first or prioritize attempting to expand an abbreviation as an acronym when searching programming language documentation. As another example, perhaps expansions found in source code require a more specialized searching mechanism that abbreviations in language documentation. If we are going to improve the accuracy of expansion techniques, then it is important to understand what sources of information are best to use and how we should obtain data from those sources to fit our needs.

We begin to answer these questions in this paper by studying the forms and locations of abbreviations and their expansions. Our goal is to record patterns of where expansions and different types of abbreviations occur. These patterns can then be used to improve the way existing techniques search for expansions in natural language and source code text.

Our research approach is separated in to 4 stages. We briefly list these stages and then expand upon them in the following sections. The steps are as follows:

1. Collect appropriate systems
2. Manual splitting of identifiers, manual expansion of the abbreviations they contain, and verification of the split and expansion(s).
3. Partially automated collection of system/language documentation
4. Automatically search all forms of documentation and source code for expansions manually collected and verified by authors.

We rely on srcML [32] for all automated collection, grouping, and preprocessing of identifiers and comments. srcML is a markup language that blends AST information into source code. Thus, it allows us to find identifiers and statically compute where these identifiers occur (e.g., in a class, function).

Table III. System Statistics

System Name	Size (KLOC)	Primary Language(s)
Wycheproof	9	Java
Telegram	781	Java, C
OpenOffice	4462	C++, Java
Enscript	59	C
KDevelop	259	C++

Table IV. Example of Abbreviation Data Set

Identifier	Abbrev:expansion	Proper Split
mdat	(m:media-dat:data)	(m) (dat)
mAddAnimations	(m:member)	(m) add animations
getCurveSpecRef	(Spec:specification -Ref:reference)	get Curve (Spec) (Ref)
wm	(wm:windows manager)	(wm)
quickAckBlock	(ack:acknowledge)	quick (Ack) Block
msTime	(ms:milliseconds)	(ms) time
args	(args:arguments)	(args)
dinf	(d:data-inf:information)	(d) (inf)
updateMinMax	(min:minimum-max:maximum)	update (min) (max)
ic_ab_back	(ic:icon-ab:action bar)	(ic) (ab) back
id	(id:identifier)	(id)

Table V. Sources of abbreviation expansions in study

Artifact	Description
Statement	Expansions that are found within statements (i.e., declaration statements, expression statements, parameter declarations) appearing 1) in a function, 2) in a class, or 3) in global scope.
Functions	Expansions that are fully matched within the scope of a single function
Classes	Expansions that are fully matched within the scope of a single class but were not matched within any of the class' methods (matches made in method are counted in the function category)
Comments	Expansions that are fully matched within the text of a single comment
System Documentation (non- adjacent)	Expansions that are fully matched within the system documentation. When matching, the words do not have to be adjacent (i.e., directly next) to one another.
System Documentation (adjacent)	Expansions that are fully matched within the system documentation. When matching, the words must be adjacent (i.e., directly next) to one another
Language Documentation (non-adjacent)	Expansions that are fully matched within the language documentation. When matching, the words do not have to be adjacent (i.e., directly next) to one another.
Language Documentation (adjacent)	Expansions that are fully matched within the system documentation. When matching, the words must be adjacent (i.e., directly next) to one another
Computer Science Dictionary	A dictionary of words commonly used in computer science literature, obtained from previous literature [7], [29]
English Dictionary	A dictionary of English words obtained from previous literature [7], [29].

A. Collecting Systems and Abbreviated Identifiers

In the first step we pick a set of 5 systems on the following three criteria: 1) written in C++, Java, C#, or C due to our reliance on srcML. 2) They must contain abbreviations. Our goal was to collect at least 100 unique abbreviations per system, so at least 100 needed to be present. 3) We wanted small, medium and large systems (in terms of KLOC) to see how the size/maturity of a system affects the location of its expansions. The size of the systems we selected are in Table III.

After selecting the systems, three of the authors separately scanned the source code manually and collected information on identifiers that contain abbreviations. Whenever an identifier was collected, it was manually split and abbreviations within the identifier were expanded by hand. Each annotator reviewed the expansions of the other two authors. Abbreviations with disagreement over a split or expansion were discussed between the authors. If consensus could not be reached, the abbreviation was removed from the study.

To select abbreviations to include in the study, the annotators chose a file from the system at random and then went from the top of the file to the bottom, collecting all abbreviations they could find before reaching the end of the file. At times, the randomly selected file was exceptionally large. To mitigate the threat of collecting too many abbreviations from a single file, the annotators limited the number of unique abbreviations from any one file to 25. In one case, Wycheproof, the system was small enough that we collected most, if not all abbreviations. An example of the manually derived data set is provided in Table IV. Notice that the data set tells us how to properly split each identifier and the expansions for any abbreviations it contains.

B. Sources of possible expansions

Table V lists the sources of abbreviation expansions considered in this study. We consider words in the source code, comments, system documentation, as well as language dictionaries. The computer science and English dictionaries are the same as used in previous literature [7], [29].

For system level documentation, we used any documentation included as part of the system's main source code repository and any of the online documentation hosted by the system's governing body. For example, Telegram's documentation is a set of API docs available through the webpage, since there were no documents hosted in their repository. In the case where we needed documentation from an online source, we used the unix command *wget* to crawl the webpage for documentation. All documentation available on the page was collected, however, there is a chance that, if some documentation was hosted on a different domain, it was missed.

C. Preprocessing software artifacts and Finding Expansions

Every artifact except the computer science and English dictionaries require varying amounts of preprocessing so they can be used for analysis. The first preprocessing step is to apply standard text normalization techniques: 1) remove all punctuation and special characters, 2) conservatively split on camelCase, under_scores, and numbers, and 3) convert all characters to lower case. As discussed earlier, we use srcML and a specialized (for srcML) version of libxml2's SAX parser to collect all required information about identifiers, comments, functions and classes.

One problem with identifiers/comments in the source code, and words in system/language documents is that words in a multi-word expansion do not necessarily appear consecutively adjacent to one another. Take the following example. Let us say we have an abbreviation named '*SpecRef'*, which expands to Specification Reference. If we want to find the expansion, we must find the word Specification and the word Reference. Naively, we could search for the string "Specification Reference", but there is no reason to assume that they occur right next to one another. They could appear several words apart within a document. For example, "This reference variable handles all access to the specification data".

This problem is accentuated in system/language documentation because these texts can be thousands of pages in length. If one part of an expansion occurs at the beginning of a long collection of text and the second part at the end, it is not likely they are related. For this reason, we keep track of the position of each word inside of system and language documents. As an example, take the following sentence: "Record the position from left to right right to left from position the record". We would take this sentence and generate the following python-like dictionary and lists, where each key's value is the positions it occurred in within the sentence.

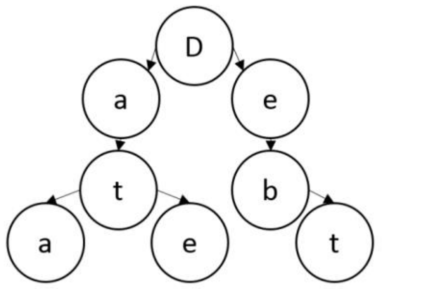
{Record:[1, 14], the:[2,13], position:[3,12], from:[4,11], left:[5,10], to:[6, 9], right:[7,8]}

This sentence has 14 words, *record* is used in the 1st and 14th positions; position in the 3rd and 12th; *from* in the 4th and 11th, etc. This allows us to search for words based on the distance between the positions they appeared in. All words in system and language documents are assigned a position list. The computer science and the English dictionaries do not suffer from this problem, as each individual entry in these dictionaries is unrelated to the next or previous entry except for the dictionary ordering. That is, it makes little sense to count positions. Additionally, this problem can appear in comments and source code, though it is less of a threat; a single comment or function is not typically the same length as a body of system/language documentation. Even so, we use the position list technique for comments, but take a different approach for source code, which we discuss in Section IV.

The final step is to take the expansions and abbreviations that were manually collected and match them in one or more of our eight artifacts. To do this, we use the following workflow:

1. Take an abbreviation and its expansion(s).
2. Scan the entire body of code for the system that corresponds with the current abbreviation/expansion(s) and record where we match the expansion and where we see the abbreviation (e.g., in a method, as part of a type name, etc).
3. Check the system and language documentation position lists and record whether we match the expansion.
4. Check the computer science and English dictionaries and record whether we match the expansion.

To perform the matching, we used prefix trees, which allow us to find words with dissimilar stems (e.g., significant vs. significantly) and Levenstein edit distance [33], which allows us to constrain how far two words can be from one another in terms of edit distance before we consider them not a match. We use



EditDistance("Date", "Data") == 1
EditDistance("Date", "Debt") == 3

Figure 1. Example prefix tree and edit distance

this to avoid the situation where two words have similar prefixes, but their stems make them different words.

For example, if we decide to limit matches to an edit distance of three, then *expand* and *expanding* will be considered matches, but *expand* and *expansion* will not. We give an example of prefix trees and edit distance in Figure 1. In this figure, *date*, *data*, and *debt* are contained in a prefix tree. They all share the same first letter, so the tree is rooted at *D*. The Levenstein edit distance is the number of edit operations it takes to turn one word into the other. To turn *date* into *data*, replace the *e* with an *a*; one operation. *Date* and *debt* require three operations. For this paper, we used an edit distance limit equal to 30% of the size of the expansion we are trying to match. For example, *expanding* is a 9-letter word; 30% of 9 is 3 (rounded up).
Therefore, if we are looking for a match to *expand* and we find *expanding*, we consider it a match because it is within 30% (3 letters) of *expanding*. We chose 30% because it empirically gave us a good tradeoff between positive and negative matches.

Table VI. Number of Abbreviations and Expansions per System

System	# Unique Abbreviations	# Unique Expansions
Telegram	178	260
Wycheproof	160	133
Enscript	169	161
OpenOffice	149	307
KDevelop	231	327
Total	887	1188

IV. Experimental Study

Using the methodology described in the last section, we answer our research questions by examining five systems and reporting on where abbreviation expansions occur most frequently. First, we provide some statistics on the data we collected for each system. As discussed above, we manually collected abbreviations for the five systems analyzed in our study. Table VI shows each system, the number of unique abbreviations and the number of unique expansions.

Because each system may have different amounts of documentation and identifiers to search for abbreviation expansions, we are careful in drawing conclusions from raw numbers of expansions found in each location. For example, if we find very few expansions in comments, we might

Table VII. Total Number of Expansions Found Per System in Source Code Identifiers.

	Type (declarations)	Type (params)	Name (declarations)	Name (params)	Name (expr)	Function	Class	Global
Enscript	55 (34%)	21 (13%)	81 (51%)	25 (16%)	84 (53%)	**96 (61%)**	56 (35%)	84 (53%)
KDevelop	148 (45%)	124 (38%)	**252 (77%)**	167 (51%)	213 (65%)	**253 (77%)**	168 (51%)	237 (72%)
Open Office	172 (56%)	147 (48%)	229 (75%)	167 (55%)	221 (72%)	**245 (80%)**	217 (71%)	199 (65%)
Telegram	83 (37%)	64 (29%)	139 (62%)	113 (50%)	133 (59%)	**168 (75%)**	143 (64%)	124 (55%)
Wycheproof	27 (20%)	12 (9%)	28 (21%)	7 (5%)	15 (11%)	**53 (40%)**	23 (17%)	0
Mean	97.00	73.60	145.80	95.80	133.20	163.00	121.40	129.40
Median	83.00	64.00	139.00	113.00	133.00	168.00	143.00	124.00
StdDev	61.41	60.38	95.29	76.38	87.28	88.65	80.21	92.86
Coefficient of Variation	0.63	0.82	0.65	0.80	0.66	**0.54**	0.66	0.72

Table VIII. Total Number of Expansions Found Per System in Comments, system, language, CS, and English corpora

	Comments	System (non-adjacent)	System (adjacent)	Language (non-adjacent)	Language (adjacent)	CS Dict	English
Enscript	41 (26%)	68 (43%)	62 (39%)	**123 (77%)**	114 (71%)	52 (33%)	96 (60%)
KDevelop	148 (45%)	**210 (64%)**	201 (61%)	207 (63%)	186 (57%)	90 (27%)	164 (50%)
Open Office	**207 (68%)**	117 (38%)	116 (38%)	111 (36%)	103 (34%)	52 (17%)	92 (30%)
Telegram	107 (48%)	122 (54%)	104 (46%)	**145 (65%)**	123 (55%)	56 (25%)	105 (47%)
Wycheproof	71 (54%)	48 (36%)	41 (31%)	**95 (72%)**	73 (55%)	31 (23%)	48 (36%)
Mean	114.80	113.00	104.80	136.20	119.80	56.20	101.00
Median	107	117	104	123	114	52	96
StdDev	65.25	62.76	61.81	43.58	41.53	21.29	41.53
Coefficient of Variation	0.57	0.56	0.59	**0.32**	0.35	0.38	0.41

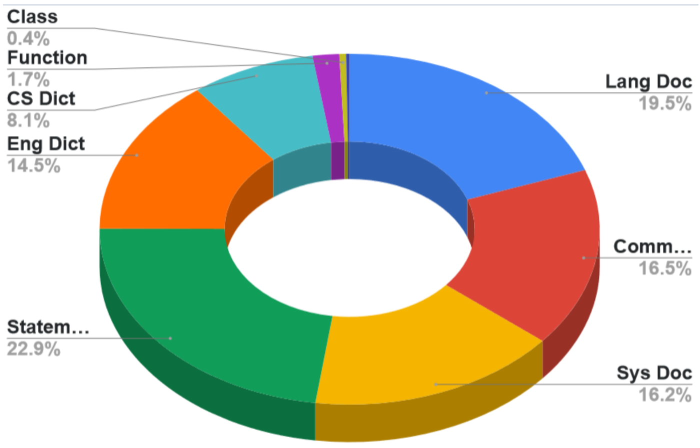


Figure 2. Where Do Abbreviation Expansions Occur? (Total of 3487)

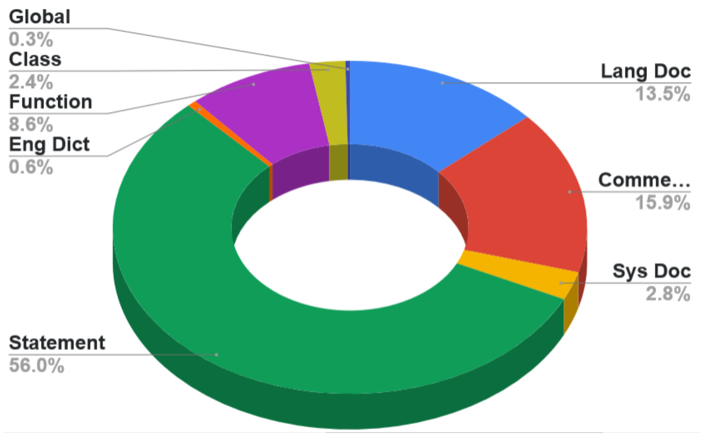


Figure 3 Where Do Abbreviation Expansions Uniquely Occur? (Total of 327)

think that comments for some system are a bad source of expansions. However, we must consider the situation where the system has no comments; this would obviously cause the number of comment-born expansions to be low. This situation only applies to artifacts that vary in size between systems. For this reason, we calculated comment and system doc density per 1 KLOC, presented in Table IX. We will refer to this table when such context is required to understand the results.

A. *Across all artifacts, where do abbreviation expansions most frequently occur in general?*

To answer this question, we look at frequency counts for the number of abbreviations found within each artifact. This data is broken down per system in Table VII and Table VIII.

Table IX. Comment, System, Language document density per 1 KLOC for each system

	Comment Density (in #comments per 1 KLOC)	System Document Density (in #words per 1 KLOC)
Telegram	14	4201
Wycheproof	134	439
Enscript	77	245
OpenOffice	161	1410
KDevelop	88	7428

These values were obtained by recording every location where we found the full expansion. For example, if the acronym *kv* expands to key value, we needed to match both the words *key* and *value* in the same place (e.g., type declaration) for it to count. The percentage in parenthesis next to each value is obtained by dividing the given value by the number of unique expansions (Table VI) for the corresponding system. For example, the 45% for Open Office comments is obtained by evaluating 148/307. Additionally, we provide the mean, median, standard deviation (StdDev), and coefficient of variation (CV).

To begin, we will look at Figure 2 and Figure 3 to get a high-level view of the data. Figure 2 shows the distribution of where expansions were found in the source code. All in all, 3860 non-unique expansions (i.e., one expansion can occur in multiple places) are in this set. Figure 3 shows the distribution of unique expansions (i.e., expansions that were found in only **one** artifact) of which there were 294. Figure 2 shows that the language corpus had the highest number of expansions followed by comments, system documentation, and source code statements. If we compare this to Figure 3, we see something interesting: Statements contained the most unique expansions by a large margin. That is, it contained the most expansions that did not appear anywhere else. The next three best sources were comments, functions (i.e., identifiers that were not in statements are included here), and language documentation.

The tables give a finer-grain view of the data in the figures; we will analyze these now. We start with Table VII, which contains counts for the number of expansions found in different parts of the source code. The first five categories present expansions found in the type/name of declarations (e.g., int x; where *int* is the type and *x* is the name), function parameter type/names, and expressions such as x = x+y or functionCall(x, y).

These categories are strict in that the full expansion needed to be found in the corresponding location (i.e., fully within the type, fully within a declaration name). In some cases, different parts of an expansion appeared in different locations (e.g., one part in a type and one part in a name); these are recorded in the last three categories: function, class, and global, since even if one part of an expansion is in a type, and one part is in a name, the full expansion still occurred within 1) the body of a function, 2) the body of a class or 3) in global scope. Note that if an expansion is found in a method (i.e., a function in a class), it is not recorded as being in a class; the function, class, and global categories are mutually exclusive.

The results in Table VII show the function category performed best in terms of CV (0.54), with declaration types second (0.63), and declaration names third (0.65).This result is not surprising; functions are where most identifiers are found so it is natural that they have a high number of expansions compared to finer levels of granularity (i.e., the first five categories) and even classes/globals.

However, it is worth noting that upwards of 30% of all expansions consistently found in declaration types and declaration names in 4 out of the 5 systems studied. This number increases to >50% of all expansions in 4 out of 5 systems for expression names and globals. The data implies that, when examining just source code, larger systems' (i.e., KDevelop, Open Office, and Telegram) expansions occur throughout the code frequently; upwards of 30% in all but one case (Telegram type params). It is harder to draw conclusions for smaller systems. The one trend that seems to appear is that smaller systems' expansions are more focused around specific locations in the source code. In particular, they were found in functions rather than classes or globals (note that Wycheproof is written in Java; there are no globals).

We now look at non-source-code artifacts. These are in Table VIII. The data shows that the language corpus performs best in terms of its coefficient of variation in either adjacent or non-adjacent (0.32 and 0.35 resp.) relative to other artifacts in this table. This indicates that the language corpus tended to perform well in all systems big or small and we postulate that this is because language documentation in C, C++, Java and likely other population languages have had a long period of time to mature and so their documentation is of high quality.

Arguably, the next closest runners-up are the CS and English dictionaries with a CV of 0.38 and 0.41 respectively. While many expansions are available in these dictionaries, they suffer one major drawback: They contain no domain/system information, which is important for expansion [31]. These dictionaries are necessarily system and domain agnostic (perhaps less-so for the CS dictionary), meaning that a tool that wants to find expansions in these dictionaries may have a harder time choosing between multiple, equally likely expansion candidates. That is, the information surrounding potential expansion candidates can help a tool in choosing which expansion is appropriate and these dictionaries may lack some of that information.

Table X Number of Adjacent Multi-Word Expansions

	Type (parameters)	Type (declarations)	Name (declarations)	Name (expressions)	Name (parameters)	Total
Enscript	1	3	3	3	0	10
KDevelop	3	6	8	3	2	22
Open Office	1	5	4	6	3	19
Telegram	6	9	14	15	12	56
Wycheproof	2	8	3	2	0	15

The system and comment categories are the weakest in Table VIII. However, there is one important note to make. In both tables, when there are outliers, Enscript and Wycheproof tend to be those outliers and are a significant drag on the CV for both comments and system data. They are also the two smallest systems we studied, with Wycheproof at 9 KLOC and Enscript at 59 KLOC. If we remove these systems from consideration, the StdDev for the system (non-adjacent) and comment corpora become: 52.31 and 50.27 respectively. Additionally, their CV become: .35 and .33 respectively. This indicates that these can be high-quality sources for expansions, but they are highly dependent on the quality of documentation and comments within the systems themselves.

Interestingly, there does not seem to be any clear correlation between comment/system density (Table IX) and the number of expansions found in comments or system documents. For example, Telegram has low comment density but more percentage-wise comment expansions than Enscript and KDevelop (Table VIII), both of which had higher comment density. This implies that increased comment or system document density does not mean more expansions are matched; the number of expansions found may have more to do with specific documentation and commenting practices; what these practices are requires further research.

The answer to RQ1 is that the quality of artifacts such as comments and system documentation will determine how much mileage an expansion technique gets out of these sources of information. However, there are a few sources that are consistently high-quality: the source code itself and language documentation. Not only that, but it is the source code and language documentation that contain the most unique abbreviation expansions. These are where approaches should focus when trying to find and filter appropriate expansions.

B. *RQ2: Do words that make up abbreviation expansions typically occur adjacent to one another?*

One important aspect of finding abbreviation expansions is the fact that words in an expansion do not always sit adjacent to one another. For example, the identifier ptHex in wycheproof expands to plaintext hexadecimal. However, the words ‘plaintext’ and ‘hexadecimal’ do not occur next to one another in their expanded forms. The question is whether this happens frequently or not. If it is frequent, then approaches that automatically expand identifiers will need to consider this when trying to find appropriate expansion candidates.

To answer this research question, we will turn our attention to Table VIII and Table X. The only source code artifacts where adjacency is an issue are the language documentation, system documentation, comments, and source code (e.g., part of an expansion found in type and part found in name). Table VIII has data about adjacent and non-adjacent words in the system and language dictionaries. We do not report comments as adjacent and non-adjacent since the numbers were the same for both (i.e., if we found a multi-word expansion in a comment, every word was adjacent). Looking at language and system documentations, most multi-word expansions were adjacent to one another overall. The largest difference was found in telegram and wycheproof, where the non-adjacent column matched 22 (10%) more expansions than the adjacent in telegram and 22 (17%) more in wycheproof.

While assuming adjacency will still allow an approach to find most expansions, it is clear that there are some expansions that may only be reachable by considering non-adjacent words for expansions. There is one difficult issue with considering non-adjacent words, however, and that is: How can we tell if two words are related to one another if they are not adjacent? This is a question that will need to be addressed with non-adjacent expansions. Next, we look at Table X, which contains data about multi-word expansions that were adjacent to one another in source code. This is similar to the data in Table VII but only counts multi-word expansions, where Table VII records single-word expansions as well as multi-word. There was a total of 257 expansions that were made up of multiple words. We define adjacency in source code slightly differently than in free text. We consider words in an expansion adjacent in code if they occurred in the same location (i.e., in the type, in the name).

Looking at the table, the results show that a fair number of multi-word expansions found in source code occur adjacent to one another, but that the majority are non-adjacent. This means that approaches to expanding these will need to pay attention to surrounding statements and expressions in the source code.

The answer to RQ2 is that words in documentation-type sources are more likely to occur adjacent to one another, while words in source code are more likely to be non-adjacent. This difference in the way expansions are found will need to be incorporated into searching strategies; prioritizing non-adjacent matches in source code while doing the opposite in documentation.

C. *RQ3: Do expansions for abbreviations of varying type occur in some artifacts more often than others?*

Different types of abbreviations require different techniques for performing the expansion. Prefix abbreviations are the simplest to expand whereas combination multi-word is the hardest [2]. We created a small program to automatically categorize abbreviations as one of the four categories first introduced in Table II. It simply looks at the form of the abbreviation versus its expansion (i.e., the expansions we empirically obtained) to perform the categorization. We manually checked the results of the categorization to make sure the algorithm worked properly. We present the total number of abbreviations in each category in Table XIII. The results of the categorization are broken down in Table XI, which contains the results for expansions found in the source code, and Table XII, which presents the results for expansions found in documentation.

Starting with Table XI, prefix abbreviations are the most common everywhere within the source code. Dropped-letter abbreviations are second, acronyms are third, and there were 0 combination multi-words and it is worth noting that we collected extremely few of these; only 11 in total across all systems. The only other thing worth noting in this table is that there is a noticeable spike in the number of acronyms in the *function* category. Because these numbers are low in the statement-level

Table XI. Frequency at which Different Types of Abbreviations Occur in Different Source Code Locations

	Type (declarations)	Type (params)	Name (declarations)	Name (params)	Name (expr)	Function	Class	Global
Prefix	**347 (50%)**	**275 (39%)**	**540 (77%)**	**354 (50%)**	**493 (70%)**	**562 (80%)**	**440 (63%)**	**472 (67%)**
Dropped	109 (41%)	84 (32%)	160 (60%)	111 (42%)	150 (56%)	181 (68%)	131 (49%)	140 (53%)
Acronym	27 (16%)	8 (5%)	25 (15%)	12 (7%)	19 (11%)	72 (42%)	36 (21%)	35 (21%)
Combo Multi-word	0	0	0	0	0	0	0	0

Table XII. Frequency at which Different Types of Abbreviations Occur in Different Software Artifacts

	Comments	System (non-adjacent)	Language (non-adjacent)	CS Dict	English
Prefix	406 (58%)	339 (48%)	395 (56%)	187 (27%)	391 (56%)
Dropped	134 (50%)	124 (47%)	149 (56%)	63 (24%)	113 (42%)
Acronym	33 (19%)	94 (55%)	130 (76%)	30 (18%)	0
Combo Multi-word	1 (9%)	8 (73%)	7 (64%)	1 (9%)	0

Table XIII Total Number of Abbreviations in Each Category

Combination Multi-Word	Dropped letter	Prefix	Acronym
11	266	701	170

categories before *function*, this implies that elements such as function names and return types are more likely to contain acronyms.

The more interesting patterns are found in Table XII, where we are looking at documentation. Prefix abbreviations are the most popular, but by a slimmer margin. There are many more acronyms, with the most appearing in the *language document corpus*. Additionally, we find our multi-combination words in this table; with the most showing up in the system and language corpora. In fact, between the system and language corpora, there is almost an even distribution of each type of abbreviation (though, of course, we found very few multi-combination abbreviations).

There are a few takeaways we can glean from this data. The first is that documentation will require more varied methods of matching and filtering candidate abbreviation expansions; especially language and system documents. The second is that the majority of expansions found in source code are single-word, since prefix and dropped-letter abbreviations (which are the single-word categories). This means that when these approaches are using information found in the source code, they might first assume that abbreviations that could go either way (i.e., could be single word or multi-word) are single-word and, if that assumption fails, then investigate multi-word options.

The answer to RQ3 is that, yes, different types of abbreviations are more likely to appear in source code versus documentation. This characteristic means that tools may be able to prioritize expansions for types of abbreviations that are more likely to appear given the artifact being searched, which could both simplify the searching method and increase accuracy.

D. *RQ4: How can an approach to expanding abbreviations use what we have learned?*

In RQ1, we learned that language documentation and functions are the most consistently high-quality places to find abbreviation expansions. Language documentation tends to be high quality because the languages used by the systems in our study are popular, mature, and well-documented. This means that it is always a good source regardless of the quality of the system itself. Functions outperformed other source code locations and artifacts. Despite this being by a slim margin, we also learned that source code statements have the highest number of unique abbreviation expansions; expansions that are found in no other place. This tips the scale in the favor of functions, though it does not mean that comments or system documentation should not be included. The problem with comments and system documentation is that they are more dependent on the maturity of the system and on developer practices. This is true for code, to some extent—however, generally speaking, source code is updated more frequently than documentation. This indicates that language documentation and functions are the most reliable sources.

Using RQ2, we were able to determine that multi-word expansions in source code are less likely to occur adjacent (i.e., in the same source code element) as one another compared to documentation, where they were highly likely to be adjacent. Further, using RQ3, we know that these multi-word expansions are less likely to happen in code than in documentation. We can theorize that, when trying to find expansions in source code, it is better to assume that abbreviations are single-word abbreviations (i.e., dropped or prefix). However, if that fails, then when we try to find a multi-word expansion for the abbreviation, we should keep in mind that the expansion words are more likely to sit non-adjacent to one another. On the other hand, RQ2 and RQ3 indicate that expansions in documentation are more likely to be adjacent to one another and that there is a more even distribution between types of abbreviations.

The answer to RQ4 is that techniques that expand abbreviations can take advantage of the differing characteristics *of source code and documentation to help find expansions that are more likely to be correct and filter out less likely candidates.*

E. Threats to Validity

The primary threats to our experimental setup are in the way we collected data. We selected files to collect abbreviations from at random and went from top to bottom, collecting every abbreviation we saw, while occasionally skipping those we had seen before. It is possible that there were abbreviations we missed due to not recognizing them or simply not seeing them. Another threat is in the form of the systems we selected. We tried to select systems that were not all in the same domain, varied in size, and were written in differing languages. However, all languages we used were still imperative and most support some form of object-oriented programming. For this reason, our results may not extend to systems written in, for example, functional languages. Our sample size is 5 systems. While these systems vary in size, domain, and language, the sample may not generalize. However, we think the number of systems is justified due to the manual component of the study; collecting a set of abbreviations is very time consuming.

V. Related Work

The goal of this paper is an empirical analysis on the nature of abbreviation expansion for the improvement of existing expansion techniques. To the authors' knowledge, no other existing literature has performed a similar analysis. As such, we report here on related work on program comprehension and abbreviation expansion techniques.

Normalizing words in identifiers is a major interest in literature. Previous work shows how normalizing words positively impacts program comprehension [26]–[28]. Yang and Tan [34] have used a technique for finding general semantically related word pairs (rPairs), including abbreviations, from a software context analysis view. They leverage the context of software comprising words in comments and identifiers to get more precise insights about those artifacts.

Guerrouj et al. [25] proposed an approach named TIDIER (Term IDentifier RecognIzER) for recognizing words composing source code identifiers. Part of this tool is used for splitting/expanding identifiers, which TIDIER successfully accomplishes in about 48% of cases studied. The same authors also investigated the effect of contextual information on identifier expansion, in terms of mapping soft words to their corresponding domain concepts. Their results show that contextual information significantly impacts identifier expansion [31]. Lawrie et al. [30] propose an algorithm for normalizing source code vocabulary. It uses a splitter, called GenTest, and is tested on a corpus of 8000 words. They additionally perform expansion using wildcard string matching and phrase finder.

Lawrie et al. [9] also proposed an expansion algorithm that uses four lists of potential expansions: a list of natural-language words extracted from the code, a list of phrases extracted from the code, a list of programming language specific words referred to as a stoplist, and a natural-language dictionary. The quality of the expansion algorithm is evaluated through comparing the tool generated expansion of 64 identifiers with 'by-hand' expansion of the same identifiers. Results shows the importance of context in the expansion process, and co-occurrence models plays a role in choosing the correct expansions.

Hill et al. [29] propose AMAP, a tool for expanding abbreviations. They categorize types of abbreviations found in software and describe the challenges in automatically expanding them. Their approach used the idea of most frequent expansion along with levels of software dictionaries to identify expansions. They evaluated their approach on 250 abbreviations, and the results showed an improvement of 57% in accuracy compared to an approach by Lawrie [9].

Fry [8] developed an algorithm for choosing an abbreviation expansion if more than one expansion possibility is identified. This work is a refinement of work presented in [29]. The results show that they improved expansion accuracy by 23.4% over the state-of-the-art at the time.

Corazza et al. [7] proposed an approach called LINSEN (Linear IdeNtifier Splitting and Expansion) that is used for identifier expansion and splitting. They evaluate their expansion approach against AMAP [29] on 250 randomly selected abbreviations. Results shows that their approach performs better than AMAP on some types of abbreviations, with a reported improvement of about 5% in terms of terms of accuracy.

Lawrie and Binkley [10] extend work in [30] by improving the abbreviation expansion. Here, they use a strategy that determines the most likely expansion by considering word co-occurrence. They report an accuracy of up to 66%

Guerrouj et al. [35] propose TRIS, a Tree-based representation approach which pre-compiles a set of dictionary words into a tree representation and associates a cost to each transformation. It treats the splitting/expansion problem as an optimization problem; optimizing splitting/expansion by treating it as a shortest path problem. Results show that TRIS is more accurate compared to other approaches [10] while remaining efficient in terms of computation time.

VI. Conclusions & Future Work

In this paper, we presented an empirical study of abbreviations in source code. We manually collected and expanded 887 unique abbreviations from five different open source systems. We then used these manually expanded abbreviations to empirically characterize the distribution different types of abbreviations and their expansions in source code and software artifacts. The goal of this study is to highlight these characteristics such that future approaches to expanding abbreviations can use them to become more accurate.

In the future, we plan to use this data to improve on the state of the art for abbreviation expansion. Our goal is to use what we have learned to focus in on collecting expansions from complementary sources of information, with appropriate techniques for the types of abbreviations we expect to find. We will also continue our investigation by analyzing other types of non-dictionary words, such as aliases and distinguishers. While we did not explore these thoroughly in this paper, it is our opinion that identification and, when appropriate, expansion of aliases and distinguishers is of high importance. While this empirical study is not necessarily exhaustive, it highlights clear are trends that the authors believe are worth studying further through application.

References

[1] L. Tan, Y. Zhou, and Y. Padioleau, “aComment: mining annotations from comments and code to detect interrupt related concurrency bugs,” in *Software Engineering (ICSE), 2011 33rd International Conference on*, 2011, pp. 11–20.

[2] G. Bavota, “Mining Unstructured Data in Software Repositories: Current and Future Trends,” in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, 2016, vol. 5, pp. 1–12.

[3] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, “Towards automatically generating summary comments for java methods,” in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 43–52.

[4] R. Rehurek and P. Sojka, “Software framework for topic modelling with large corpora,” in *In Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, 2010.

[5] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol, “Can better identifier splitting techniques help feature location?,” in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, 2011, pp. 11–20.

[6] W. Olney, E. Hill, C. Thurber, and B. Lemma, “Part of speech tagging Java method names,” in *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, 2016, pp. 483–487.

[7] A. Corazza, S. Di Martino, and V. Maggio, “LINSEN: An efficient approach to split identifiers and expand abbreviations,” in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, 2012, pp. 233–242.

[8] Z. Fry, “Improving Automatic Abbreviation Expansion within Source Code to Aid Program Search Tools,” University of Delaware, 2008.

[9] D. Lawrie, H. Feild, and D. Binkley, “Extracting meaning from abbreviated identifiers,” in *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on*, 2007, pp. 213–222.

[10] D. Lawrie and D. Binkley, “Expanding identifiers to normalize source code vocabulary,” in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, 2011, pp. 113–122.

[11] R. Jongeling, P. Sarkar, S. Datta, and A. Serebrenik, “On negative results when using sentiment analysis tools for software engineering research,” *Empir. Softw. Eng.*, vol. 22, no. 5, pp. 2543–2584, 2017.

[12] C. D. Newman, R. S. AlSuhaibani, M. L. Collard, and J. I. Maletic, “Lexical categories for source code identifiers,” in *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, 2017, pp. 228–239.

[13] R. S. AlSuhaibani, C. D. Newman, M. L. Collard, and J. I. Maletic, “Heuristic-based part-of-speech tagging of source code identifiers and comments,” in *Mining Unstructured Data (MUD), 2015 IEEE 5th Workshop on*, 2015, pp. 1–6.

[14] D. Binkley, M. Hearn, and D. Lawrie, “Improving identifier informativeness using part of speech information,” in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 203–206.

[15] S. Gupta, S. Malik, L. Pollock, and K. Vijay-Shanker, “Part-of-speech tagging of program identifiers for improved text-based software engineering tools,” in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, 2013, pp. 3–12.

[16] E. Brill, “A simple rule-based part of speech tagger,” in *Proceedings of the third conference on Applied natural language processing*, 1992, pp. 152–155.

[17] T. Brants, “TnT: a statistical part-of-speech tagger,” in *Proceedings of the sixth conference on Applied natural language processing*, 2000, pp. 224–231.

[18] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer, “Feature-rich part-of-speech tagging with a cyclic dependency network,” in *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, 2003, pp. 173–180.

[19] A. Ratnaparkhi, “A maximum entropy model for part-of-speech tagging,” in *Conference on Empirical Methods in Natural Language Processing*, 1996.

[20] J. Giménez and L. Marquez, “Fast and accurate part-of-speech tagging: The SVM approach revisited,” *Recent Adv. Nat. Lang. Process. III*, pp. 153–162, 2004.

[21] S. Haiduc, J. Aponte, and A. Marcus, “Supporting program comprehension with source code summarization,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, 2010, pp. 223–226.

[22] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, “On the use of automated text summarization techniques for summarizing source code,” in *Reverse Engineering (WCRE), 2010 17th Working Conference on*, 2010, pp. 35–44.

[23] G. Sridhara, L. Pollock, and K. Vijay-Shanker, “Automatically detecting and describing high level actions within methods,” in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 101–110.

[24] E. Hill, D. Binkley, D. Lawrie, L. Pollock, and K. Vijay-Shanker, “An empirical study of identifier splitting techniques,” *Empir. Softw. Eng.*, vol. 19, no. 6, pp. 1754–1780, 2014.

[25] L. Guerrouj, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, “Tidier: an identifier splitting approach using speech recognition techniques,” *J. Softw. Evol. Process*, vol. 25, no. 6, pp. 575–599, 2013.

[26] A. A. Takang, P. A. Grubb, and R. D. Macredie, “The effects of comments and identifier names on program comprehensibility: an experimental investigation,” *J Prog Lang*, vol. 4, no. 3, pp. 143–167, 1996.

[27] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, “Effective identifier names for comprehension and memory,” *Innov. Syst. Softw. Eng.*, vol. 3, no. 4, pp. 303–318, 2007.

[28] L. Guerrouj, “Normalizing source code vocabulary to support program comprehension and software quality,” in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 1385–1388.

[29] E. Hill *et al.*, “AMAP: automatically mining abbreviation expansions in programs to enhance software maintenance tools,” in *Proceedings of the 2008 international working conference on Mining software repositories*, 2008, pp. 79–88.

[30] D. Lawrie, D. Binkley, and C. Morrell, “Normalizing source code vocabulary,” in *Reverse Engineering (WCRE), 2010 17th Working Conference on*, 2010, pp. 3–12.

[31] L. Guerrouj, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, “An experimental investigation on the effects of context on source code identifiers splitting and expansion,” *Empir. Softw. Eng.*, vol. 19, no. 6, pp. 1706–1753, 2014.

[32] M. L. Collard, M. J. Decker, and J. I. Maletic, “srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration,” in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, 2013, pp. 516–519.

[33] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” in *Soviet physics doklady*, 1966, vol. 10, pp. 707–710.

[34] J. Yang and L. Tan, “Inferring semantically related words from software context,” in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, 2012, pp. 161–170.

[35] L. Guerrouj, P. Galinier, Y.-G. Guéhéneuc, G. Antoniol, and M. Di Penta, “Tris: A fast and accurate identifiers splitting and expansion algorithm,” in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, 2012, pp. 103–112.