An Empirical Study of Abbreviations and Expansions in Software Artifacts

Christian D. Newman
Rochester Institute of Tech
Rochester, New York, USA
cnewman@se.rit.edu

Michael J. Decker
Bowling Green State Univ.
Bowling Green, OH, USA
mdecke@bgsu.edu

Reem S. Alsuhaibani
Kent State University
Prince Sultan Univ.
Kent, OH, USA
ralsuhai@kent.edu

Anthony Peruma, Dishant Kaushik
Rochester Institute of Tech
Rochester, New York, USA
{dxk3597, axp6201}@rit.edu

Emily Hill
Drew University
Drew, NJ, USA
ehill1@drew.edu

Abstract—Expanding abbreviations is an important text normalization technique used for the purpose of either increasing developer comprehension or supporting the application of natural-language-based tools for source code identifiers. This paper closely studies abbreviations and where their expansions occur in different software artifacts. Without abbreviation expansion, developers will spend more time in comprehending the code they need to update, and tools analyzing software may obtain weak or non-generalizable results. There are numerous techniques for expanding abbreviations, most of which struggle to reach an average expansion accuracy of 59-62% on general source code identifiers. In this paper, we reveal some characteristics of abbreviations and their expansions through an empirical study of 861 abbreviation-expansion pairs extracted from 5 open-source systems in addition to analyzing previous literature. We use these characteristics to identify how current approaches may be complementary and how their results should be reported in the future to help maximize both our understanding of how they compare with other expansion techniques and their reproducibility.

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

I. Introduction

Researchers frequently use natural language processing techniques to analyze and normalize source code identifiers for numerous activities including bug detection [1], mining and analyzing software repositories [2]–[5], automated documentation [6], topic modeling [7], feature location [8] and more. Because so many techniques rely on the quality of natural language processing tools, and identifiers contain 70% of the characters in source code [9], it is important that they perform well even in the face of the often imperfect and incomplete language used in source code [10]. However, tools for Natural Language Processing (NLP) of source code need to be more accurate [10]–[12].

One way to increase the accuracy of natural language analysis techniques and improve the comprehensibility of source code is by expanding abbreviations [13], [14]. As one example, many research techniques use Information Retrieval (IR) approaches to help draw conclusions about software, but the vocabulary (including abbreviations) can vary in meaning between software projects and even in documentation written specifically for the codebase [13], [15]. This presents a difficult problem for automated analysis techniques because conclusions which leverage identifier names in multiple software projects will only generalize under the assumption that identifier names between these projects are normalized. Part of this normalization step must involve expanding abbreviations correctly such that abbreviations like *cfg* will be correctly expanded based on context (e.g., to configuration or context-free grammar).

As another consequence, if abbreviations are not handled correctly by software analysis techniques that use natural language data, there is a real risk that these techniques will produce non-replicable results or erroneous data points. Jongeling [12] highlights this problem in a different context; showing that sentiment analysis tools used in software research produce divergent conclusions because they were not trained for the software domain. Equivalently, if it cannot be assumed that vocabulary between source code projects is normalized, then any analysis leveraging that vocabulary is exposed to a large threat.

Abbreviation expansion is also important for developers. Previous research has shown that shorter identifiers are more difficult for developers to comprehend [16]. Additionally, this work compared comprehension of identifiers containing words against identifiers containing letters and/or abbreviations. Their results found that when identifiers contained only words instead of abbreviations or letters, developer comprehension speed increased by 19% on average.

Although expanding some abbreviations like *num* to *number* is not difficult, the general case is non-trivial. For example, the abbreviation *cfg* can have multiple possible expansions in source code, such as *configure*, *configuration*, *control flow graph*, or *context-free grammar*. In order to address these and other related problems, many approaches for abbreviation expansion have been proposed previously. Using the reported accuracy of these expansion tools, we found that the average accuracy of their expansion abilities lie between 59-62% [17]–[19], with the highest reported expansion accuracy being 78% [20] on a smaller test set than the others were evaluated on.

Abbreviation expansion tools have matured over time, but there is significant room for improvement. While reading prior work on abbreviation expansion, we noticed that there is a lot of variation between techniques and how they are reported. Many publications use a different combination of metrics to evaluate their technique, researchers have implemented these techniques using everything from regular expressions to language models to string-matching approximation algorithms, and each technique has types of abbreviations on which its performance is strong and types on which its performance is weak. Further, each technique uses its own mixture of software artifacts from which expansions are gathered.

Table I. Syntax-Based Abbreviation 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

This opens two big questions: **1)** ***What are the characteristics of abbreviations/expansions in software artifacts and how much can these characteristics influence the quality of expansion techniques?*** *and* ***2) Can we use these our data on these characteristics to improve how we conduct and report research on expanding abbreviations in the future?*** Published techniques differ in the quality of their expansion for different types of abbreviations in different types of software artifacts. To begin understanding why, we first have to understand what these characteristics look like and then compare this to the way these techniques are discussed and reported in their original publications. By doing this, we can reflect on how the data (or lack thereof) presented in these publications helps, or hurts our ability to fully understand their strengths and weaknesses. In addition, we can begin understanding what different types of abbreviations and expansions look like in different locations such that future techniques can take advantage of this knowledge to improve on prior literature.

Therefore, in this paper, we take two steps. 1) Using characteristics of abbreviations and expansions derived from previous literature, and a gold set of abbreviations and expansions manually derived by the authors, we study what typical abbreviations and expansions look like in different software artifacts, thereby creating a ground truth. 2) Using the ground truth from the previous step as a baseline, we examine the evaluation methodology used by previous literature to understand how our characteristics influence their quality and determine whether these characteristics are properly accounted for and reported.

Our goal is to analyze what these characteristics look like in general and use this as a baseline to highlight strengths and weaknesses in the way expansion techniques and the software artifacts they use are evaluated. One of the main takeaways from this paper is a set of characteristics that future abbreviation expansion techniques should properly account for and report in order to improve expansion recall/precision as well as contextualize strengths/weaknesses of the approach. To the authors' knowledge, no other existing literature has performed a similar analysis.

II. Abbreviations in Software

Abbreviations are sequences of characters that can be expanded to a larger word or phrase. In this work, we define a software artifact as any by-product of software development including the code or any documentation which describes the code (e.g., behavior, design, architecture) or process used to produce the code. Developers use natural language (e.g., English) to convey meaning to other humans in source code. As a developer writes code, they ultimately construct identifiers, comments, and documentation using 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 unrestricted in what words to use. Some words that developers choose to use are abbreviations.

Expanding abbreviations has been the topic of numerous research papers [17]–[19], [20]. For example, an approach to automatically expand abbreviations might have the following steps: 1) detect abbreviation in an identifier; 2) look for expansion in surrounding code, comment, documentation, or dictionary; 3) expand abbreviation to its natural language word or phrase.

Expanding abbreviations is typically carried out 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. Previous work by Hill, et al. [19] categorizes different types of 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 additional sub-categories. This taxonomy, and examples of each abbreviation type, are shown in Table I. Typically, the accuracy of current approaches to expanding abbreviations varies depending on the type of abbreviation being expanded [17], [19], [21].

III. Experimental Study

What are the characteristics of abbreviations and expansions found in different software artifacts? If we were expanding a domain term, such as JSON, would it be more likely to occur in project documentation than in the source code or comments? Or perhaps acronym expansions occur more frequently in language documentation than any other software artifact. To reduce false-positive expansions and find expansions as quickly as possible, abbreviation expansion techniques should be specialized to look where the expansion is most likely to occur first. These kinds of optimizations can only be done if expansion techniques understand the characteristics of abbreviations and expansions found in different software artifacts.

Table II. System Statistics

System Name	Primary Language(s)	Size (KLOC)	# Unique Abbreviations	# Unique Abbreviation-Expansions	Comment Density (in #comments per 1 KLOC)	Project Document Density (in #words per 1 KLOC)
Wycheproof	Java	9	126	156	134	439
Telegram	Java, C	781	143	164	14	4201
OpenOffice	C++, Java	4462	129	143	161	1410
Enscript	C	59	132	156	77	245
KDevelop	C++	259	189	242	88	7428
Sum/Total		5570	719	861	474	13723

A. Research Methodology

To begin studying these characteristics, we divided our methodology into four stages.

1. Collect a set of projects from which we will manually obtain abbreviations and expansions.
2. Manually split and expand identifiers which contain an abbreviation; verify the split and expansion(s).
3. Partially automated collection of Wycheproof, Telegram, Open Office, Enscript, and KDevelop project documentation using Unix's wget command when required.
4. Partially automated collection of C++ [1], Java [2], and C [3] language documentation using Unix's wget command.
5. Automated search of all forms of documentation and source code for expansions manually collected and verified by authors.

We rely on srcML [22] 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).

B. 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 (for a total of 500; other abbreviation papers generally evaluate with ~200-250), so 500 is in-line with prior work. 3) We looked for 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 sizes of the systems we selected are in Table II. To select abbreviations to include in the study, the annotators chose a file from each 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.

To collect abbreviations from each selected file, 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 author reviewed the expansions of the other two authors. Abbreviations with disagreement over a split or expansion were discussed between the authors. There were no expansions that the authors were unable to come to an agreement on. The manually derived data set is provided and discussed here [23].

C. Sources of possible expansions

We use different software artifacts to find expansions for the abbreviations we collected. These sources are as follows: The source code, comments, project documentation, programming language documentation, a computer science dictionary [24], and an English [25] dictionary— both used in previous literature [17], [19].

For project-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 at [26], 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 since we instructed wget not to leave the domain we originally provided it to avoid crawling unintended websites.

D. 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 (Simple API for XML) parser to collect all required information about identifiers, comments, functions, and classes.

One problem with identifiers/comments in the source code, and words in project/language documents is that words in a multi-word expansion do not necessarily appear adjacent to one another (i.e., no words between them). 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 (i.e., adjacent) to one another. They could appear several words apart within a document. For example, "This reference variable handles all access to the specification data". For this reason, we keep track of the position of all words in the software artifacts so that we can determine when we have

[1] https://en.cppreference.com/w/

[2] https://docs.oracle.com/javase/8/docs/

[3] https://en.cppreference.com/w/c/language

expanded an abbreviation using terms that are adjacent or if we have expanded an abbreviation using terms that are not adjacent.

The final step is to take the expansions and abbreviations that were manually collected and match them in one or more of our software 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 project 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.

IV. Experimental Results

Using the methodology described in the previous section, we answer our research questions (specified below) by examining five systems and measuring three characteristics; two of which (i.e., 1 and 3) are motivated by previous literature on expanding abbreviations and one of which we derive from this study. These characteristics are: 1) Distributions of expansions in different artifacts, 2) adjacency of terms constituting multi-term expansions (e.g., acronyms) in different artifacts, and 3) distribution of expansions for abbreviations of different types in different artifacts. The answers to these Research Questions are later used in Section VI to understand how characteristics of abbreviations/expansions in different artifacts affect techniques that expand abbreviations. They are additionally used to highlight what characteristics future expansion techniques should focus on in both their implementation and in the data reported after evaluation.

First, we provide some statistics on the data we collected for each system. As discussed above, we manually collected and expanded abbreviations for the five systems analyzed in our study. Table II 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 think that comments for some system are a bad source of expansions. However, we must consider the situation where the system has few or 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 project document density per 1 KLOC, presented in Table II. We will refer to this table when such context is required to understand the results.

A. *RQ1: What is the distribution of abbreviation expansions across all artifacts?*

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 III and Table IV. Note that in Table IV, we include counts for when we found the words in an expansion adjacent (i.e., next to) to one another and for when we found them either adjacent or non-adjacent (i.e., anywhere). The values in these tables 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 abbreviation-expansions (Table II) for the corresponding system. For example, the 70.6% for Open Office adjacent comments is obtained by evaluating 101/143. Additionally, we provide the total, mean, median, and standard deviation.

To begin, we will look at Figure 2 and Figure 1 to get a high-level view of the data. Figure 2 shows the distribution of where expansions were found each software artifact. All in all, 3067 non-unique expansions (i.e., one expansion can occur in multiple artifacts) are in this set. Figure 1 shows the distribution of unique expansions (i.e., expansions that were found in only **one** artifact) of which there were 69. Figure 2 shows that the language documentation had the highest number of expansions followed by function body, project documentation, and the English dictionary. If we compare this to Figure 1, we notice that language documentation also contained the most unique expansions. That is, it contained the most expansions that did not appear anywhere else. The next two highest sources of unique expansions were the project's documentation and function names.

The tables give a finer-grain view of the data in the figures; we will analyze these now. We start with Table III, 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:

1. Type/name of declarations (e.g., *String keyMaterial;* where *String* is the type and *keyMaterial* is the name and words in the type and name are expansion candidates.
2. Function parameter type/names.
3. Expressions such as *ctHex = ciphertext;* or *doFinal(test)* where ct, hex, cipher, text, do, final, and test would be expansion candidates.

These categories are strict in that the full expansion needed to be found in the corresponding location (e.g., 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: functions/methods, class name/field, or globals, 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 field/name 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 III show the function category performed best in terms of consistency. 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 (recall that expansions in methods are not counted for classes).

We now look at non-source-code artifacts. These are in Table IV. The data shows that the language documentation has the highest frequency of expansions most consistently. This indicates that the language documentation tended to perform well in all projects big or small, likely because language

Table III. Total Number of Expansions Found per System in Source Code Identifiers.

	Type (declarations)	Type (parameters)	Name (declarations)	Name (parameters)	Name (expr)	Functions and Methods	Class Name/Field	Global
Enscript	53 (34%)	20 (12.8%)	63 (40.4%)	25 (16%)	69 (44.2%)	**75 (48.1%)**	49 (31.4%)	67 (42.9%)
KDevelop	125 (51.7%)	108 (44.6%)	**161 (66.5%)**	121 (50%)	144 (59.5%)	**161 (66.5%)**	120 (49.6%)	157 (64.9%)
Open Office	104 (72.7%)	100 (69.9%)	106 (74.1%)	98 (68.5%)	107 (74.8%)	**108 (75.5%)**	105 (73.4%)	105 (73.4%)
Telegram	99 (60.4%)	75 (45.7%)	112 (68.3%)	103 (62.8%)	112 (68.3%)	**122 (74.4%)**	111 (67.7%)	102 (62.2%)
Wycheproof	30 (19.2%)	16 (10.3%)	29 (18.6%)	12 (7.7%)	13 (8.3%)	**39 (25%)**	17 (10.9%)	0
Total	411	319	445	359	445	**505**	402	434
Mean	82.20	63.80	89.00	71.80	89.00	101.00	80.40	86.80
Median	99.00	75.00	107.00	98.00	107.00	108.00	105.00	102.00
StdDev	39.26	43.57	50.13	49.62	50.13	46.40	45.04	56.80

Table IV. Total Number of Expansions Found per System in Comments, project, language, CS, and English corpora

	Comments (anywhere)	Comments (adjacent)	Project (anywhere)	Project (adjacent)	Language (anywhere)	Language (adjacent)	CS Dict.	English Dict.
Enscript	35 (22.4%)	32 (20.5%)	77 (49.4%)	67 (42.9%)	**141 (90.4%)**	129 (82.7%)	55 (35.3%)	110 (70.5%)
KDevelop	104 (43%)	103 (42.6%)	**225 (93%)**	205 (84.7%)	222 (91.7%)	194 (80.2%)	94 (38.8%)	171 (70.7%)
Open Office	105 (73.4%)	101 (70.6%)	**127 (88.8%)**	124 (86.7%)	121 (84.6%)	113 (79%)	54 (37.8%)	98 (68.5%)
Telegram	108 (65.9%)	104 (63.4%)	126 (76.8%)	108 (65.9%)	**149 (90.9%)**	126 (76.8%)	55 (33.5%)	110 (67.1%)
Wycheproof	38 (24.4%)	37 (23.7%)	45 (28.8%)	39 (25%)	**95 (60.9%)**	69 (44.2%)	32 (20.5%)	51 (32.7%)
Total	390	377	600	543	**728**	631	290	540
Mean	78.00	75.40	120.00	108.60	145.60	126.20	58.00	108.00
Median	104.00	101.00	126.00	108.00	141.00	126.00	55.00	110.00
StdDev	37.93	37.39	68.16	63.45	47.53	44.86	22.39	42.80

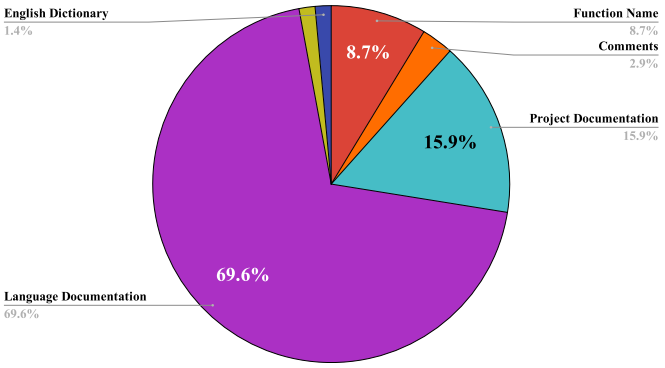

Figure 1. Where Do Abbreviation Expansions Uniquely Occur? (Total of 69)

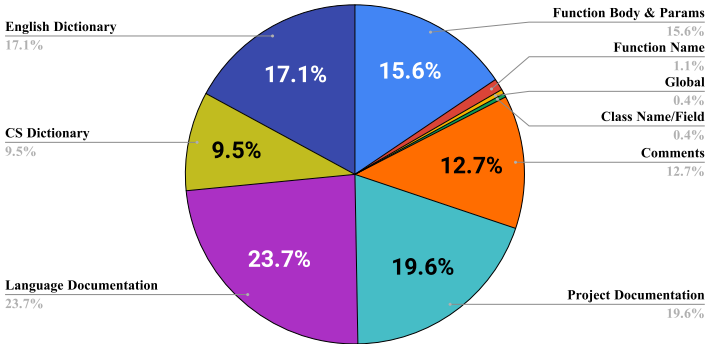

Figure 2. Where Do Abbreviation Expansions Occur? (Total of 3067)

documentation in C, C++, Java, have had a long period of time to mature are of high quality. Turning to the CS and English dictionaries briefly-- while many expansions are available in these dictionaries, they suffer one major drawback: They contain no domain/system information, which is important for expansion [27].

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. Language documentation suffers some of the same drawbacks but to a lesser extent. For example, Java's cryptography library documentation has domain information for cryptography but not project-specific information.

Interestingly, there does not seem to be any clear correlation between comment/project density (Table II) and the number of expansions found in comments or project documents. For example, Telegram has low comment density but more percentage-wise comment expansions than Enscript and KDevelop (Table IV), both of which had higher comment density. This implies that increased comment or projects document density does not necessarily mean more expansions will appear; the number of expansions found may have more to do with specific documentation and commenting practices. More research is required to determine what these practices are.

Table V. Number of Non-Adjacent Multi-Word Expansions

	Type (params)	Type (decls)	Name (decls)	Name (expr)	Name (params)	Total
Enscript	0	3	4	3	1	11
KDevelop	5	5	8	7	7	32
Open Office	9	6	5	6	6	32
Telegram	8	9	9	8	7	41
Wycheproof	6	6	11	2	7	32

Table VI. Total Number of Abbreviations per Category

	Acronym	Dropped	Combo.	Prefix	Total
Wycheproof	41 (38.3%)	22 (20.6%)	3 (2.8%)	**41 (38.3%)**	107
Open Office	22 (16.7%)	34 (25.8%)	4 (3%)	**72 (54.5%)**	132
KDevelop	33 (14%)	60 (25.5%)	4 (1.7%)	**138(58.7%)**	235
Telegram	38 (23.9%)	33 (20.8%)	0 (0%)	**88 (55.3%)**	159
Enscript	23 (15.2%)	46 (30.5%)	1 (0.7%)	**81 (53.6%)**	151

The answer to RQ1 is that the language documentation, project documentation, and source code contain a similar distribution of expansions (23.9%, 19.6%, and 17.5% respectively from Figure 2*) when we are not considering uniqueness. If we consider only expansions that occur in one place, language documentation has the largest share of the distribution at 69.6% (*Figure 1*), with project documentation coming in second place. The distributions in these figures highlight the importance of both source code and external software artifacts for expanding abbreviations and give us an idea of what the typical distribution of expansions looks like across multiple artifacts.*

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

One aspect of finding abbreviation expansions that is not commonly explicitly discussed is the fact that words in an expansion do not always appear 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; there are other words between them. 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 IV and Table V. The only software artifacts where adjacency is an issue are language documentation, project documentation, comments, and source code (e.g., part of an expansion found in type name and other part is found in declaration name). Table IV has data about the frequency of adjacency between terms in expansions in the project documentation, language documentation, and comments. Looking at language and project documentation, most multi-word expansions were adjacent to one another overall. The largest difference was found in Telegram and Wycheproof, where the *anywhere* project documentation column matched 18 (~10%) more expansions than the adjacent in Telegram and the *anywhere* language documentation column matched 25 (~16%) more in Wycheproof. Notably, the effect of adjacency is much less pronounced in comments.

While assuming adjacency will still allow an approach to find most expansions, there are some expansions that may only be reachable by considering non-adjacent words for expansions. Therefore, to get the maximum number of expansions available, especially in project and language documentation, we require a technique that deals with lack of word adjacency. One issue with considering non-adjacent words is how can we tell if two words are related to one another (i.e., part of the same expansion) if they are not adjacent? This is a question that will need to be addressed when expanding using non-adjacent words. Next, we look at Table V, which contains data about multi-word expansions that were non-adjacent to one another in source code. This is similar to the data in Table III but only counts multi-word expansions, where Table III records single-word expansions as well as multi-word. There was a total of 148 expansions found in source code 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 source code if they occurred in the same location (e.g., both in a declaration type, both in a declaration name). Adjacency is generally limited to words occurring on the same line of code (e.g., words that do not appear on the same line but do appear within the same function are **not** considered adjacent).

If we take the number of non-adjacent multi-word expansions and divide by the total number of expansions that were found in source code (148/798; we get 798 by removing expansions from non-source-code artifacts), we find that ~19% of all abbreviation expansions are multi-word and non-adjacent.

The answer to RQ2 is that non-adjacent expansions tend to occur in the source code, project, and language documentation and there is a notable lack of them in comments. From the perspective of the code, 19% of multi-word expansions are non-adjacent. Additionally, in project and language documentation, considering words that are non-adjacent can increase the number of abbreviations you are able to expand by 10-16% in three of the five systems we studied (Table IV). Given this, it is important to understand the effectiveness on a tool on adjacent and non-adjacent expansions as the tools effectiveness on different systems/software artifacts will decay if it is ineffective.

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 I. It simply looks at the form of the abbreviation versus its expansion (i.e., the expansions we manually 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 VI. The results of the categorization are broken down in Table VII, which contains the results for abbreviation types found in the source code, and Table VIII, which presents the results for abbreviation types found in documentation.

Starting with Table VII, prefix abbreviations are the most common everywhere within the source code. Dropped-letter

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

	Type (declarations)	Type (params)	Name (declarations)	Name (params)	Name (expr)	Functions and Methods	Class Name/Field	Global
Prefix	**273 (65.5%)**	**219 (52.5%)**	**314 (75.3%)**	**244 (58.5%)**	**305 (73.1%)**	**335 (80.3%)**	**283 (67.9%)**	**303(72.7%)**
Dropped	92 (51.1%)	70 (38.9%)	104 (57.8%)	82 (45.6%)	99 (55%)	111 (61.7%)	84 (46.7%)	94 (52.2%)
Acronym	46 (29.3%)	30 (19.1%)	53 (33.8%)	33 (21%)	41 (26.1%)	59 (37.6%)	35 (22.3%)	37 (23.6%)
Combo Multi-word	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)

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

	Comments (anywhere)	Project (anywhere)	Language (anywhere)	CS Dict.	English Dict.
Prefix	**275 (65.9%)**	**351 (84.2%)**	**411 (98.6%)**	**193 (46.3%)**	**407 (97.6%)**
Dropped	80 (44.4%)	134 (74.4%)	162 (90%)	61 (33.9%)	116 (64.4%)
Acronym	34 (21.7%)	106 (67.5%)	147 (93.6%)	34 (21.7%)	0 (0%)
Combo Multi-word	1 (8.3%)	9 (75%)	8 (66.7%)	1 (8.3%)	0 (0%)

Table IX. Techniques that Report Overall Accuracy. ** = Technique reports combined split/expansion accuracy

Original Publication	Maximum Reported Overall Accuracy
Lawrie 2011[18]	66%
Lawrie 2007 [28]	64%
Alatawi [20]	78.%
Tidier [29]	48%
****Tris** [30]	86%

Table X. Techniques that Report Accuracy per Abbreviation Type

	Prefix	Dropped	Acronym	Combo	Overall Accuracy
AMAP[19]	79.7%	77%	46.9%	9.2%	63%
LINSEN[17]	86%	77.8%	36%	66.7%	62%

Table XI. Techniques that Report Overall Precision and Recall. ** = Technique reports combined split/expansion accuracy, N/A = Not reported

	Average Precision	Average Recall	F-Measure	Overall Accuracy
****Tris**[30]	95%	91%	92%	86%
Jiang[21]	95%	65%	N/A	N/A
Lingua::IdSplitter [31]	86%	86%	89%	83%

Table XII. Techniques that Report Precision per Abbreviation Type (did not report recall per abbrev type)

	Prefix Precision	Dropped Precision	Acronym Precision
Jiang [21] **Param Name**	97%	79%	96%
Jiang [21] **Param Type**	100%	N/A	98%

abbreviations are second, acronyms are third, and there were 0 combination multi-words-- it is worth noting that we collected extremely few of these; only 12 in total across all systems. The frequency of dropped-letter and prefix abbreviations in the code is expected as is the relatively low number of acronyms and combo-word abbreviations.

The more interesting patterns are found in Table VIII, 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 documents*. Additionally, we find our multi-combination words in this table; with the most showing up in the project and language documents. In fact, the project and language documentation had very similar distributions of this abbreviation type (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 project documents. The second is that most expansions found in source code are single-word, since prefix and dropped-letter abbreviations (which are the single-word categories - Table I) are far more common than the others. This means that when these approaches are using information found in the source code, they should 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.

To answer RQ3: Yes, different types of abbreviations and their expansions are more likely to appear in source code versus documentation. One takeaway for this characteristic is that tools should give higher probability to expansions for types of abbreviations that are more likely to appear given the artifact being searched. For example, one should weight acronyms as more likely in language documentation than source code. This will cut down on false positives when choosing between multiple candidate expansions. Additionally, knowing that certain abbreviation types are more likely to appear in a given artifact ahead of time allows for choosing appropriate expansion techniques to handle increased probability of seeing those abbreviation types and their expansions.

V. Review of Abbreviation Expansion Techniques

We reviewed previous literature on techniques that expand abbreviations. We contextualize our review using the data from

Table XIII. Software Artifacts used in Techniques in Literature Review

	[30]	[21]	[18]	[28]	[20]	[19]	[17]	[29]	[31]
Source code identifiers	✓	✓	✓	✓	✓	✓	✓	✓	✓
Comments			✓	✓		✓	✓	✓	✓
Project Documentation			✓						✓
Language Documentation	✓							✓	
Computer Science Dictionary/training data	✓	✓					✓	✓	✓
English Dictionary/training data	✓		✓	✓	✓		✓	✓	✓

the study above to help us address the goals stated in the Introduction.

Expanding abbreviations has been the topic of numerous research papers [17]–[21], [28]–[31]. We summarize all of these papers in subsection A. We analyzed each technique, looking for: 1) Which artifacts are searched. 2) Whether they report on the adjacency or non-adjacency of multi-term expansions in different artifacts. And 3) the reported effectiveness on different abbreviation types. These relate to the three characteristics highlighted in our experimental results (i.e., RQ1, RQ2, and RQ3) above and we address themin subsections B, C and D.

A. Previous Abbreviation Expansion Techniques

Lawrie et al. [28] proposed an expansion algorithm that uses four lists of potential expansions. They evaluate using 64 identifiers whose abbreviations were manually expanded. Later, Lawrie and Binkley published another expansion technique [18] which extends work in [13] and [28]; improving the abbreviation expansion by using word co-occurrence to determine the most likely expansion. They report an accuracy of up to 66%.

Hill et al. [19] proposed 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 [28].

Corazza et al. [17] 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 [19] on 250 randomly selected abbreviations. Results show that their approach performs better than AMAP on some types of abbreviations, with a reported improvement of about 5% in terms of accuracy.

Guerrouj et al. [29] 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. They additionally show that contextual information significantly impacts identifier expansion [27]. Guerrouj et al. [30] also propose TRIS, an 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. They report that TRIS is more accurate compared to other splitting/expansion approaches [18]. However, they do not report a specific expansion accuracy. Instead, they report splitting accuracy because their technique splits and, if needed, also expands abbreviations.

Alatawi et al. [20] proposed a bigram based inference model that utilizes unigram statistical properties to retrieve the original form of the words in the source code automatically. They evaluate their technique using a randomly selected set of 100 abbreviations and report an accuracy of 78% .

Jiang et al [21] propose a technique for expanding abbreviations in parameters. Their technique works off of the observation that abbreviations in formal parameters can often be expanded by looking at terms contained in its corresponding actual parameter and vice versa. They report an average precision of 95% and an average recall of 65%.

Carvalho et al [31] propose Lingua::IdSplitter, a technique for splitting and expanding identifiers. Their approach takes advantage of an approach they propose, which automatically constructs a custom dictionary constructed from several software artifacts; providing their technique with domain-specific information and expressions that they use to help expand abbreviations and use term frequency similar to AMAP.

B. Software Artifacts used in Technique Evaluation.

In RQ1, we saw that the frequency of expansions is nearly evenly split between language/project documentation and in the function body (Figure 2). Further, we found language documentation is the most likely source that contain expansions that occur nowhere else (Figure 1). The idea that artifacts outside of the code contain important expansions is not new; previous techniques have explored this idea. However, one important question to ask is: Which techniques use which software artifacts? How effective are they on the software artifacts that they use? We begin to answer these questions in Table XIII. On the left-hand side of this table are different types of software artifacts identified through reviewing the literature on expansions. At the top is the citation to the paper reviewed.

Interestingly, some of the most recent techniques use little, if any, information outside of the source code [20] [21] and report the highest accuracy [20] and precision/recall [21] compared to the others. This implies that using more external information does not necessarily translate into a higher-quality expansion technique. This also does not mean that external documentation is not important. While Jiang et. al's technique [21] has 95% precision, its recall averages 65%; meaning more data sources might be required to increase its recall.

We also notice that only three papers [18], [29], [31] report any kind of metric (e.g., accuracy, precision, recall) of their technique at the granularity of the individual software artifacts used by their technique, yet most techniques do use artifacts outside of source code identifiers. Instead, most techniques reported overall accuracy/precision/recall of their approach by combining data from all expansions found in any software

artifact. There is a general lack of information available about how effectively techniques found expansions in different software artifacts. One negative to the lack of data here is that it is difficult to tell which software artifact(s) a technique underperformed on or if that underperformance is due to the technique or the quality of the artifact.

C. Reported effectiveness on non-adjacent expansions

Expansions whose terms appear non-adjacent to one another in text is not reported in previous literature. In fact, it is generally not possible, without an implementation available, to understand whether a technique is effective at finding expansion terms that appear non-adjacent in different software artifacts by just reading the paper associated with the technique. That is, even if a paper reports effectiveness on different abbreviation types, based on our literature review, the papers do not report effectiveness on non-source-code artifacts, and in general, it is then not possible to estimate effectiveness on non-adjacent expansion terms. As shown in our data (i.e., Table III and Table IV) and discussed in RQ2, many multi-term expansions are found adjacent to one another. However, there is a non-trivial number of non-adjacent expansion terms that are missed if a given technique is ineffective (RQ2 - Section IV.B).

D. Reported effectiveness on different abbreviation types

In RQ3, we look at where expansions for different types of abbreviations occur and find that there is a difference in the distribution depending on which artifact we are analyzing. It is important to know how effective a given technique is on different abbreviation types because each abbreviation type has different characteristics. Acronyms, for example, are more likely to appear in project or language documentation (Table VIII). Terms in their expansion may also appear non-adjacent to one another. If a technique reports effectiveness without breaking their evaluation down by abbreviation type, it is very difficult to understand what types of artifacts this technique will be effective on. Additionally, without reporting results at this granularity, it is very difficult to understand how a technique compares to others. For example, does one technique improve on another for particular abbreviation types, or is it complementary?

We analyzed the papers we collected to see how they report the results of their individual evaluations; seeking to understand if they provide this information, or, if not, we wanted to know what information they do provide. To help with this, we grouped techniques by which metrics they use in their evaluation. The first group, shown in Table IX, reports overall accuracy but do not specify accuracy on different abbreviation types [20], [28], [29], [32]. The second group, shown in Table X, reports accuracy with respect to each type of abbreviation (e.g., prefix, acronym) [17], [19]. Finally, the third group, found in Table XI, and Table XII, reports precision/recall/accuracy [21], [30], [31], with one of them additionally reporting precision per abbreviation type [21]. One other technique that uses precision/recall, Tris [30], did not explicitly report precision or recall for abbreviation expansions. Instead, Tris reports the accuracy of their splitting technique which also performs expansion. We were unable to ascertain the accuracy of the expansion part of their technique separate from the splitting technique. For this reason, we report their splitting precision and recall, which may not be fully reflective of expansion accuracy.

Even the most recent publications fall into different groups [17], [20], [21], [31], which implies a difference in perspective when it comes to how these approaches should be evaluated. While each evaluation is valid on its own, the differences cause problems when trying to compare works, particularly in the case where no implementation is available for a given technique, or the technique requires significant re-tooling (which we have found to be unfortunately common). A consistent, holistic set of metrics would help alleviate these problems; creating a standard by which future techniques can be more easily compared even absent of implementation. Besides this advantage, it would also help us understand how different approaches to expansion are complementary and allow users of these tools to pick the one most suited for their data set without having to try them all. Therefore, we make a recommendation for a set of metrics, based on previous literature and the characteristics we study in this paper, that can be used to ease the burden of comparing and understanding the strengths of each technique.

1. Precision, Recall, F1, and Accuracy for each abbreviation type
2. The types of software artifacts and Precision, Recall, F1, and Accuracy for expansions found in each artifact
3. Precision, Recall, F1, and Accuracy for expansions where the terms are non-adjacent.

The reason for these metrics is based on a combination of what previous work reported. That is, we combined the groups described above since these metrics have proven valuable in numerous, similar evaluation tasks and give us a valuable perspective on the data. Additionally, we advocate for reporting these metrics for each artifact type instead of in general when possible. The reason for this is that there is very little data on the effectiveness of expansion techniques on individual artifacts and so it is difficult to understand how the properties of different software artifacts affect the quality of expansion. Finally, reporting these metrics will help support replication. Of course, there may be situations where some of these are not applicable. For example, reporting for individual software artifacts is not possible if your technique uses only source code. However, we feel it is good to have this highlighted such that when future researchers are developing expansion techniques, they can learn from previous literature and make an informed decision.

VI. Discussion

The empirical study provides us with a general view of how characteristics used in previous literature manifest in general software artifacts. These characteristics are as follows: 1) the (unique and non-unique) distribution of expansions in different artifacts, 2) the adjacency of expansion terms in different artifacts, and 3) the distribution of expansions for different abbreviations types in different artifacts. We now present a review of previous literature, aiming to use our data as a baseline to highlight strengths and weaknesses in the way expansion techniques are evaluated. Studying this data helps us identify what aspects of abbreviation expansion require more thorough investigation. We highlight our core findings below.

Word adjacency when expanding abbreviation in different software artifacts. Word adjacency affects how easy it is to find candidates for multi-word expansions. The further spread apart multiple words are in a corpus (i.e., the more words between them), the harder it is to 1) find those words and 2) the

further apart those words are, the more likely it is that they are unrelated and so should not be used together to form a candidate expansion. We found that 19% of expansions in the source code are multi-word and non-adjacent while 10-16% of expansions in non-source-code artifacts are non-adjacent.

No paper that we studied reported effectiveness on non-adjacent expansions. In some cases, it can be inferred that a technique is likely ineffective on non-adjacent terms. For example, AMAP [19] makes heavy use of regular expressions to find expansion candidates; the regular expressions discussed in their paper were not designed for non-adjacent terms. It is not always easy to infer this, however; motivating our suggestion that future techniques report precision, recall, f1, and accuracy on this characteristic. Additionally, it is an open question how related multiple, non-adjacent terms are to one another as the distance between them in the text grows.

Density of different expansion types in different types of software artifacts. Expansions for different types of abbreviations were more likely to occur in different types of software artifacts. While prefix and dropped remain common in most locations in our study, acronyms and, though there were few, combo type abbreviation expansions were much more highly likely to appear in non-source-code artifacts. This indicates that expansion techniques that are more effective on acronyms will be more successful in these types of artifacts and underscores the need for more techniques to report effectiveness at the granularity of abbreviation expansion type. While we find that a number of techniques do report this data, there is still disagreement on what metrics to use. Some use only accuracy, others used precision/recall, and there was a group that did not report at this granularity at all. Our recommendation, based on our study, is to encourage reporting precision, recall, f1, and accuracy at the granularity of each different abbreviation type.

Effectiveness of including different software artifacts. Related to the previous characteristics is how much more effective different software artifacts made the technique. Based on our literature review, more software artifacts does not always mean higher quality expansions, as some of the techniques reporting the highest accuracy/precision use few external sources [20], [21]. This does not mean that including more artifacts is bad, but that more insight on how much different artifacts influence the effectiveness of individual techniques would be valuable; it would show us how different ways of expanding abbreviations are sensitive to different inputs. It would also add to our understanding of how different techniques contrast or synergize with one another.

Having completed our discussion, we now have answers to the two questions stated in the Introduction:

1) *What are the characteristics of abbreviations/expansions in software artifacts and how much can these characteristics influence the quality of expansion techniques?* The answers are in RQs 1-3 where we discussed how much of an affect each characteristic has on the reachability of expansions (e.g., some expansions only appear in specific artifacts).

2) Can we use our data on these characteristics to improve how we conduct and report research on expanding abbreviations in the future? In Sections V and VI we show that, despite the characteristics we discussed having a notable impact on what expansions a technique will be able to find, they are not always reported and even when they are, there is disagreement in the types of metrics to use for evaluation even among similar studies. Our data shows that these characteristics should be reported in order to provide a holistic view of expansion techniques and the artifacts they use.

VII. Threats to Validity

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. 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 large set of abbreviations is very time consuming.

In our data set, we expand some abbreviations that may not be considered worth expanding because their abbreviation is more well-known than their expansion (i.e., URL). One might question if these are worth expanding. From our perspective, this is a good question, but deciding whether an abbreviation should be expanded is not the goal of this paper and is also likely subjective depending on the specific use-case of abbreviation expansion. Therefore, we do not see it as a significant threat. While the data set was manually curated, we had to automatically search for expansion matches in the systems that we studied. It is possible that our automatic splitting techniques missed some expansions or caused some false negatives (i.e., the split was wrong). There is also a small chance that srcML's mark-up was incorrect in a few cases, which may have caused us to miss a small number of expansions.

VIII. Conclusions & Future Work

In this paper, we presented an empirical study of abbreviations and expansions in different software artifacts. We manually collected and expanded 861 unique abbreviation-expansions from five different open source systems. We then used these manually expanded abbreviations to study three characteristics derived from previous literature. Data from this study was used to understand how different characteristics affect the number of expandable abbreviations and to contextualize our literature review which analyzed how prior research evaluates abbreviation expansion techniques and how evaluation methods can be improved in the future.

Our hope is that this work will help spur the field to report more about the software artifacts they study as well as provide more granular data on the characteristics we discuss above. In the future, we would like to study potential synergy between differing expansion techniques as well as the use of online sources with query mechanisms (e.g., Wikipedia) for finding expansion candidates.

IX. Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant No. 1850412.

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] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman, "An Empirical Investigation of How and Why Developers Rename Identifiers," in *Proceedings of the 2Nd International Workshop on Refactoring*, New York, NY, USA, 2018, pp. 26–33.

[4] V. Arnaoudova, L. M. Eshkevari, M. D. Penta, R. Oliveto, G. Antoniol, and Y. Gueheneuc, "REPENT: Analyzing the Nature of Identifier Renamings," *IEEE Trans. Softw. Eng.*, vol. 40, no. 5, pp. 502–532, May 2014.

[5] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman, "Contextualizing Rename Decisions using Refactorings and Commit Messages," in *Proceedings of the 19th IEEE International Working Conference on Source Code Analysis and Manipulation*, New York, NY, USA, 2019.

[6] 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.

[7] 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.

[8] 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.

[9] F. Deissenbock and M. Pizka, "Concise and consistent naming [software system identifier naming]," in *13th International Workshop on Program Comprehension (IWPC'05)*, 2005, pp. 97–106.

[10] 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.

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

[12] 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.

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

[14] 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.

[15] D. Lawrie, H. Feild, and D. Binkley, "Quantifying identifier quality: An analysis of trends," *Empir. Softw. Eng.*, vol. 12, pp. 359–388, 2007.

[16] J. Hofmeister, J. Siegmund, and D. V. Holt, "Shorter identifier names take longer to comprehend," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 217–227.

[17] 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.

[18] 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.

[19] 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.

[20] A. Alatawi, W. Xu, and J. Yan, "The Expansion of Source Code Abbreviations Using a Language Model," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, 2018, vol. 02, pp. 370–375.

[21] Y. Jiang, H. Liu, J. Qi Zhu, and L. Zhang, "Automatic and Accurate Expansion of Abbreviations in Parameters," *IEEE Trans. Softw. Eng.*, vol. PP, pp. 1–1, 2018.

[22] 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.

[23] C. Newman, M. J. Decker, R. AlSuhaibani, D. Kaushik, A. Peruma, and E. Hill, "An Open Dataset of Abbreviations and Expansions," in *35th IEEE International Conference on Software Maintenance and Evolution*, Sept 30th, p. 11.

[24] "Computer Dictionary of Information Technology." [Online]. Available: http://www.computer-dictionary-online.org/. [Accessed: 18-Jul-2019].

[25] "Ispell." [Online]. Available: http://www.gnu.org/software/ispell/ispell.html. [Accessed: 18-Jul-2019].

[26] "Telegram APIs." [Online]. Available: https://core.telegram.org/. [Accessed: 18-Jul-2019].

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] N. R. Carvalho, J. J. Almeida, P. R. Henriques, and M. J. Varanda, "From source code identifiers to natural language terms," *J. Syst. Softw.*, vol. 100, pp. 117–128, 2015.

[32] 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.