An Approach to Automatically Assess Method Names

Reem S. Alsuhaibani†
Computer Science
Kent State University
Kent Ohio USA
ralsuhai@kent.edu

Christian D. Newman
Computer Science
Rochester Institute of Technology
Rochester New York USA
cnewman@se.rit.edu

Michael J. Decker
Computer Science
Bowling Green State University
Bowling Green Ohio USA
mdecke@bgsu.edu

Michael L. Collard
Computer Science
The University of Akron
Akron Ohio USA
collard@uakron.edu

Jonathan I. Maletic
Computer Science
Kent State University
Kent Ohio USA
jmaletic@kent.edu

ABSTRACT

An approach is presented to automatically assess the quality of method names by providing a score and feedback. The approach implements ten method naming standards to evaluate the names. The naming standards are taken from work that validated the standards via a large survey of software professionals. Natural language processing techniques such as part-of-speech tagging, identifier splitting, and dictionary lookup are required to implement the standards. The approach is evaluated by first manually constructing a large golden set of method names. Each method name is rated by several developers and labeled as conforming to each standard or not. These ratings allow for comparing the results of the approach against expert assessment. Additionally, the approach is applied to several systems and the results are manually inspected for accuracy.

CCS CONCEPTS

•Software and its engineering •Software creation and management •Software post-development issues •Maintaining software

KEYWORDS

Program comprehension, Method names, Method naming standards, Identifier quality.

ACM Reference format:

1 Introduction

Method names are critical to understand the intent and behavior of a method [1]. High-quality names play an important role in increasing productivity [2][3][4][5]. Names are the primary source of information programmers use to acquire knowledge about source code [1]. While there are many ways to improve the comprehension of software, a crucial and simple means is to improve the quality of method names. Well-constructed names save large amounts of time and costs during software maintenance tasks [6][7] which can consume +70% of the software lifecycle [8]. High-quality identifiers can also improve comprehension tasks by around 19% when no abbreviations or letters are used [9]. In general, poor names impair comprehension and make software harder to maintain [10][11][12][13].

Recent work [14] investigating method naming practices surveyed over 1100 professional developers to gauge their views on various naming standards/conventions. The work defines ten method naming standards derived from published literature on the topic of naming along with naming practices from open-source projects. The standards address issues such as the length of the name, grammatical structure, and the use of abbreviations/acronyms. The vast majority of the respondents to the survey [14] agree with and support these proposed standards for naming methods.

The goal of this research is to codify and realize these ten, verified, method-naming standards into an automated approach. The approach supports a quality assessment of a system's method names, produces a score for each method name, and flags names that violate a particular standard with feedback. The advantages of this approach are: 1) It is easy to apply and does not require special models or training, 2) It is based on real human-subject data (mainly professional developers), meaning that the identifiers it flags as volitions have some problems that need to be addressed, 3) The approach is explainable; the tool explains to developers what is wrong with the identifier name so that they can make an informed decision about which problems need to be addressed.

Both researchers and practitioners can leverage the approach to assess the quality of method names of their systems and understand the weakness of specific names. The approach can be invaluable during code reviews; pointing out poorly formed method names that need to be refactored. We also envision the integration of this approach into the IDE to notify developers of the name quality while developing code. It can also be applied to automatically check commits and pushes, and any new method names introduced in a commit can be automatically checked and flagged as necessary. It also has the potential to assist educators in drawing insights about students naming practices to better support their naming habits. Lastly, method names evaluated as high-quality can be selected for software text analysis and machine learning tasks.

The contributions of the work presented here are: 1) The realization of a set of method naming standards into an automated approach, 2) An evaluation of the approach, and 3) A golden set of good/bad method names that other researchers can utilize for investigations on method naming. The realized approach can also be used to evaluate and assess whole systems, thus giving researchers/practitioners a handle on the naming quality of a system in comparison to others.

Our approach is not specific to programing language method names but does focus on languages that support object-oriented development—C++, Java, C#, and others. This is in contrast to other approaches, which implement standards that are language (Java) specific, such as Nominal [15], Checkstyle [16], and Java Coding Standard Checker (JCSC) [17]. In addition, the approach, and method naming standards, apply mainly to application code. Test suites and their associated methods often have different naming practices than production code and likely require different standards [18][19][20] [21].

The paper is organized as follows. The following section provides related work. The method naming standards are described in section 3. Then the approach is discussed in section 4. Section 5 gives an overview of the assessment and scoring. Section 6 presents how we conducted the evaluation. Results and discussion are available in sections 7 and 8.

2 Related Work

The quality of source-code identifiers contributes to the software quality, and software quality ensures software comprehension. When source-code identifiers are readable, they are usually of high quality. A great deal of research discusses identifiers and their impact on program and comprehension [4][22][11][23][24][25] and how they play an important role in supporting the quality of code. As identifier naming has such importance in software research, researchers are still conducting research to find ways to support software developers' comprehension tasks. Maalej et al. [26] run a study to learn more about how developers practice program comprehension on a number of participants. They analyzed the importance of particular types of knowledge for comprehension. One of their main findings is that standardization and the consistent use of naming conventions allow developers to become familiar with an application more quickly, and thus program comprehension activities are easier.

Choosing a good name is not an easy task for developers, and several studies try to assist this problem by providing naming recommendations. A recent experimental study by Feitelson [27] uses a naming model to improve the quality of identifier names. In their study, the model suggests considering three steps for naming; Selecting the concepts to include in the name, choosing the words to represent each concept, and constructing a name using these words. The results show that the study subjects' names are of better quality after considering the model recommendations. Arnaoudova et al. [13, 28] define source-code Linguistic Antipatterns (LAs) and discuss poor practices in naming and choice of identifiers. They created a catalog of 17 types of LAs related to naming inconsistencies and implemented a linguistic anti-pattern detector tool called LAPD. This is followed by a recent identifier appraisal tool called IDEAL [29], which implements the linguistic anti-patterns using srcML.

There are also some empirical studies that investigated naming guidelines adherence. While not specific to method names, Relf [30] investigated 21 identifier naming style guidelines focusing on typography and length of identifiers on some examples from Java and Ada systems. Relf [7] also runs an empirical study investigating whether programmers improve the readability of their source code if they have support from a source-code editor that offers dynamic feedback on their identifier-naming practices. He focused on the effects of identifier-naming style flaws during editing and investigated whether reducing these identifier-naming style flaws improves source-code readability. The results show that there is a statistically significant improvement in readability.

Butler et al. [31] survey the forms of Java reference names and then use the study outcome to investigate naming-convention adherence in Java. This is followed by Nominal, a naming-convention-checking library for Java that allows declarative specification of conventions regarding typography and the use of abbreviations and phrases [15]. To test Nominal, they extract 3.5 million reference fields, formal arguments, and local variable name declarations from several projects to determine their adherence to the Java naming conventions. Their results show that developers largely follow naming conventions, but adherence to specific conventions varies.

3 The Method Naming Standards

We base our assessment approach on a set of method naming standards from a large-scale study [14] [32] that investigates developer perceptions of method name quality. We developed the approach utilizing all the published standards [14] and the associated artifact [32]. Specifically, the study surveys developers to understand their perception of several characteristics of the identifiers used for method names. The work defines ten method-naming standards, given in Table 1, derived from research literature and published coding standards. These standards are not specific to any given language or naming convention. Study results show that developers are in wide agreement that these characteristics are very important in forming high-quality identifier names. Our approach directly leverages this work and implements a means to analyze method names to determine their quality based on the characteristics articulated in that work.

Table 1. Method standards used in the approach. See [14] for complete details.

#	Standard Name	Rules
1	Naming Style	A single standard naming style is used.
2	Grammatical Structure	If there are multiple words, they form a grammatically correct sentence structure.
3	Verb Phrase	It is a verb or a verb phrase.
4	Dictionary Terms	Only natural language dictionary words and/or familiar/domain-relevant terms are used.
5	Full Words	Full words are used rather than a single letter.
6	Idioms and Slang	It does not contain personal expressions, idioms, or slang.
7	Abbreviations	It only contains known or standard abbreviated terms. All abbreviations are well known or part of the problem domain.
8	Acronyms	It only contains standard acronyms. All acronyms are well known or part of the problem domain.
9	Prefix/Suffix	It does not contain a prefix/suffix that is a term from the system. This standard does not apply to languages such as C that do not have namespaces.
10	Length	Maximum number of words is no greater than 7.

4 The Approach

The current approach is implemented in Python. It takes a name and examines it against the method-naming standards. We automatically extract all the method names from a given system using srcML (see srcML.org) [33][34][35]. Currently, the approach works for all the programming languages supported by srcML, i.e., C++, C#, Java. Thus, our approach is not restricted to a particular language naming conventions [36], [37], [31] but more broadly to a variety of languages. We manually investigate all the possible violation cases for each standard and implement it in the assessment approach.

4.1 Naming Style

The naming style standard requires developers to use one of the common lexical naming styles. Our assessment approach supports underscore, camelCase, PascalCase, and kabob-case naming styles. It recognizes any name that does not follow one of these styles. Based on our observations, we classify the violations into four cases:

a) Mixed-Case Violations: Name that mixes styles.
b) Underscore style violation: Name starts with an underscore.
c) Kabob-case violation: Name starts with a dash.
d) No naming style used: example: `strbuffersize()`.

4.2 Grammatical Structure and Verb Phrase

Part-of-speech tagging is essential to understanding the grammatical aspects behind a given method name. The verb phrase and grammatical structure standards are the two method naming standards that require checking the part-of-speech tag sequence of each method name. These standards are based on the concept of grammar patterns, which are shown to be a good way of analyzing the semantics and structure of identifiers, including method names [38]. The reliance on part-of-speech tagging makes the choice of an accurate part-of-speech tagger important. We integrate a recently developed and currently best performing source code PoS tagger called the SCANL ensemble tagger [39] to tag the words (after splitting) that make up a given method name. SCANL's ensemble uses three state-of-the-art part-of-speech taggers: POSSE [40], SWUM [41], and Stanford [42], and the final tagset is composed of the most common PoS tags essential to assess an identifier.

Our approach first investigates if the method name contains a verb for any verb phrase violation. And then, it looks for any grammatical structure violations in the method's name. Specifically inspecting to see grammatical violations such as if the name starts with an English modal verb (e.g., can, should) followed by another verb. For example, `canFind()`. Also, if the name is invalid from a linguistics standpoint. For example, when the method name begins/ends with a preposition, e.g., `sizeTo()`, `MoveTo()`.

4.3 Dictionary Terms

To deal with the dictionary term standard, we leverage WordNet[43], a large lexical database of English words. In the approach, we use WordNet as the base dictionary for checking words that appear in each method name. To check the words that compose the method name, we must first split the name.

For splitting, we use Spiral [44]. Spiral is a Python module that provides several different functions for splitting identifiers found in source code. In the approach, we use the Spiral Ronin splitting algorithm as it has several advanced splitting features. It uses a variety of heuristic rules, English dictionaries, and tables of token frequencies obtained from source-code repositories. Ronin is shown [12] to have a high splitting accuracy compared to other state-of-the-art splitting techniques [45]. Splitting method names is important to assess a method name composing words. Thus, the approach's final assessment accuracy results highly depend on this phase.

Three standards relate to the use of correct dictionary terms (i.e., dictionary terms, abbreviations, and acronyms). WordNet supports our approach in recognizing unknown or unfamiliar abbreviations and acronyms as non-dictionary terms. We developed a way to add terms to deal with domain-dependent abbreviations and acronyms not in WordNet. Examples of these words include SQL, UTF8, unicode, runtime, sharepoint, iterator, and namespace. We also consider expanding known abbreviations into dictionary terms for part of speech tagging purposes.

4.4 Full Words

The *full words standard* only concerns using single letters in method names and numbers instead of the word form (e.g., `int2float()` versus `intTofloat()`). Our approach assesses method names for any occurrences of single letters (i.e., A-Z, a-z, 0-1). Using single-letter names also violates the full words standard. Developers are expected not to use single letters to name methods per this standard. Examples include using j,i, and k to name a method.

4.5 Prefix and Suffix

Prefixes are words used at the beginning of a method name, while suffixes are used at the end. These words can be a term from the system, such as `scintilla_init()` that appears in Notepad++ project, where `scintilla` is a name of an open-source library that provides editing component function for this project. For each project in our evaluation, we manually inspect for any prefix or suffix and create a list of them. The approach then inspects if a given method name starts with a particular prefix or ends with a particular suffix according to the list. This standard does not apply to systems written using a language that does not have a construct to support namespaces, such as C.

4.6 Length

The number of words in a method name plays an important role in communicating the intent of a method. The results of the Alsuhaibani et al. [14] survey of professional developers found that the *length* of a method name should be no more than 7 words. As such, in the approach here, we set 7 as the maximum number of words a method can have after splitting. The assessment approach examines a given method's length and assesses it against this value. If there are more than 7 words, it is a violation of the length standard.

4.7 Idioms and Slang

Sometimes *idioms* or *slang* are used by developers for adding entertainment or cuteness to source-code identifiers [46]. A list of common English idioms and slang that developers tend to use is created. We collected terms from the work of Martin [46] along with other common American slang words and phrases. The approach inspects method names against idioms such as ASAP, cool, clunky, LOL, FYI, OMG, etc. This list can be expanded as new instances are encountered.

5 Assessment and Scoring

The assessment includes a quality score based on adherence to the standards and feedback or comments reporting the flaws found in a name. The score is the total number of standards the method name upholds (i.e., between 10 and 0). A score of 10 means the method name follows all the standards and is high-quality in the aspects that the approach can measure. For seven of the standards, we deduct one point if the method name violates any of the standard's violating conditions. The exception is the dictionary terms, abbreviation, and acronym standards, which all use the same dictionary and analysis. Differentiating between abbreviations and acronyms used in a name is a challenge we encountered, as those standards are all related, so we give a 3-point deduction for any violation related to any of them. We believe that we gave a proper score for each rule, so reasonable names will not get very low scores. For example, method names such as "`relationalExpressionNoIn()`" will not get a poor score because it is only violating the verb phrase and grammatical structure standards. It will still get a reasonable score of 8.

```
Method Name: relationalExpressionNoIn()
Score: 8
Feedback: Check the grammatical structure of the method name; the name
starts or ends with preposition/Add a verb to the method name.
```

The score is primarily designed to highlight the number of violations exemplified by a given method name to help guide developers' attention to potentially low-quality method names. It is not designed to directly measure whether the method name is comprehensible, or incomprehensible. Thus, some low scoring identifiers will be comprehensible, but still have problems that, if solved, may increase its comprehensibility. The score also assumes that the standards are all equally weighted. This, of course, may not be the case. However, there have been no studies to determine if variable weighting is appropriate. We leave this for future work.

6 Evaluation

As stated previously, the approach uses the Spiral identifier splitter, the WordNet Dictionary, and a source code specialized part-of-speech tagger. Common source code abbreviations/acronyms are manually added to the dictionary; so, the approach recognizes them as dictionary words. Also, in our approach, the known abbreviations contain the corresponding expansions of their full form so that the utilized tagger can provide the most accurate part-of-speech tag for our approach assessment.

In the study, we use two different methods for the evaluation. First, we construct a golden set of method names that are labeled as adhering to each standard or not. We use this golden set to develop and evaluate the effectiveness of our automated assessment approach. To show the generalizability and effectiveness of the approach, we select three additional systems from different domains and languages, and we manually evaluate the approach on all 10 rules with a statistically significant sample of method identifiers from each of the systems..

6.1 Development of a Golden Set

We create a golden set as a baseline for the evaluation to develop and preliminarily evaluate the approach's accuracy. The authors performed an intensive manual evaluation of a random subset of method names in the context of the ten method naming standards. To create the golden set, we choose a random project with the following criterion: 1) it should be a widely used and popular open-source project; that has been available for multiple years (+10) with active and continued support; 2) contains a large number of lines of code (i.e., not a small system); 3) has a stable version of the system posted in a repository; 3) has been recently updated (i.e., still being supported). We chose the free source code editor for Microsoft Windows, Notepad++. It was initially released 17 years ago, and it is available in 90 languages.

Golden Set Sample Size: The Notepad++ project contains a total of 6,733 method names. A total of 354 method names is selected using the confidence interval sample size (i.e., the confidence level is 95% and the confidence interval is 5%) for the golden set. A significant amount of effort and time was required to evaluate each method for adherence to the standard (i.e., 354 per standard with ten standards means the total checking effort is 3540 applications of each standard).

Golden Set Evaluation: Each of the authors was assigned a spreadsheet that contains the 354 method names, with columns representing the ten method naming standards: naming style, grammatical structure, verb phrase, dictionary terms, full words, idioms and slang, abbreviations, acronyms, prefix/suffix, and length. In addition, a column contains the source-code file path of the method name-- for the evaluator's reference (e.g., to learn more about the method behavior). Each evaluator is asked to check each name for adherence to the standard, and for each method name, the evaluator marks any violated standards. For example, if a name does not contain a verb, the evaluator marks the verb phrase standard as a violation for that name. Each method name can have zero or more violations. For example, suppose a name violates three method naming standards, where it has 9 words, uses unknown abbreviations, and does not adhere to a naming style. In that case, the evaluator marks all three related standards as violations. All evaluators used Table 1 for the naming rules applicable to each method.

During creating the golden set, evaluators were sometimes unsure about the correct assessment due to poor method naming practice. Thus, after completing the assignment, we organized group meetings to further discuss the evaluation results for each name and each standard that the evaluators were not all in agreement. This procedure required five separate intensive meetings (i.e., approximately three hours per each of the first four standards and an hour for the other standards). This procedure resulted in clear insight into when agreements and disagreements mainly occur per name and rationales for each disagreement. There were also cases where we had no agreement or had an agreement only after lengthy discussion. There was much discussion on the topic of what is considered a dictionary term. This is a crucial component of the evaluation and leads to a better understanding of how to consistently apply this standard.

As an example, we consider well-known abbreviations such as "`info`" in `writeSourceInfo()` as satisfying the dictionary terms standard. In contrast, the unknown abbreviation "`Nsis`" in `classifyWordNsis()` violates the dictionary terms standard as a developer may not be sure about the correct expansion of this term. However, if this abbreviation is part of the system's domain, then it is not a violation for that specific system. We support this through extensions to the dictionary. We also consider misspelled words as non-dictionary words. For example, "attribute" is incorrectly spelled in `getChildElementByAttribut()` and violates the dictionary terms standard. (We will make the golden set publicly available upon the paper acceptance)

Table 2 Summary of the systems used in the manual inspection evaluation

System	Release	Language	KLOC	Total Methods	Sample
Notepad++	7.9.0	C++/C	400K	6733	354
Terminal Image Viewer	1.1.0	C++	50K	901	270
Bio Java	5.4.0	Java	902K	10737	363
Flash Develop	5.3.3	C#	488K	16702	376

Table 3 The two different evaluation perspectives of the confusion matrix when considering each as the positive class. The approach compared to the manually determined results.

	Violation Positive		No Violation Positive	
	Approach	Manual	Approach	Manual
True Positive (TP)	Violation	Violation	Non-Violation	Non-Violation
True Negative (TN)	Non-Violation	Non-Violation	Violation	Violation
False Positive (FP)	Violation	Non-Violation	Non-Violation	Violation
False Negative (FN)	Non-Violation	Violation	Violation	Non-Violation

6.2 Assessing Open-Source Systems

To ensure generalizability, applicability, and validation, we run the approach on three different systems and manually validate a set of samples per system. Table 2 provides details about the systems used for this part of the evaluation (Notepad++ is also included for comparison). Note that we used systems written in three different languages, namely C++, C#, and Java. For each of the three systems, we manually analyzed a random statistically significant sample (i.e., a confidence level of 95% and a confidence interval of 5%). In total, we manually validated 1,363 method names, from four systems (including the golden set), for adherence to the ten method naming standards.

7 Results

This section provides the details of the results. There are two different perspectives one can take when examining the results. First, we can consider a standard violation the positive class (true positive). We term this the violation positive perspective. Ideally, all standard violations should be identified by the approach with no standard non-violating cases identified as violating (false positive) and no standard violating cases identified as non-violating (false negative). This perspective captures this ideal. However, this perspective does not completely evaluate how well the approach assesses standard non-violating cases (e.g., identifying high-quality names for text analysis, machine learning, etc.) For this, we can consider a standard no violation as the positive class (true positive). We term this the no violation positive perspective. In this view, identifying a standard non-violating case as a violation is a false negative, while identifying a standard violating case as a non-violation is a false positive. Both perspectives are important as they show how well the approach wholly performs on no violation cases and violation cases. Table 3 shows these two perspectives of how to evaluate the approach. The first row shows the true positives. These are the method names which both the approach and the manual inspection identify as having a violation in the first perspective. While in the second perspective, both agree that there is no violation.

We present the results of applying the approach on Notepad++ and comparing to the golden set in Section 7.1. The results of the subsequent manual evaluation applying all ten standards to three additional systems is given in Section 7.2-7.4. For all four systems, as the precision, recall, and F1 score are different depending on each perspective, and because of space restrictions, we present the average precision, average recall, and average F1 score of the two perspectives according to [47] (aka macro precision, recall, and F1 score). The true positive, true negative, false positive, and false negative values for each are also reported so that each perspective's metrics can be calculated individually.

7.1 Notepad++(354 sampled methods)

To evaluate the correctness of the approach's results, we compare it to the golden set. For example, for the method name `setItemIconStatus()`, the approach assigns this method a score of 10 with no standard violations reported. To validate this, we compare this result to our golden set to see if the approach produces the same result. In this case, our golden set also shows that this name has no violations. Thus, our approach correctly scored the name. This applies to all the methods names we have for evaluation (see Error! Not a valid bookmark self-reference.). The naming style standard, from the no violation viewpoint, has 338 true-positives, meaning that the approach can correctly identify method names that do not have any naming style violations, per the golden-set evaluation. For the grammatical structure standard, and from violation viewpoint, the approach detects 13 violating methods out of 32 violating cases. For the verb phrase standard, the approach identifies 252 names as adhering to the verb phrase standard, per the golden set, and 51 as non-adhering. However, the approach is not able to detect 29 verb phrase violations in this system. For the dictionary terms standard, the approach detects 82.0% of the total violations in this set. It also detects 5 out of 7 full words violations and 4 of 5 prefix and suffix ones. There are no reported violations for the idioms and slang, and length standards in this system.

Table 4 Notepad++ method name evaluations, and the average of precision, recall, and f score

Violation	TP	TN	FP	FN	Avg Precision	Avg Recall	Avg F1 Score
No Violation	TN	TP	FN	FP			
Naming Style	6	338	6	4	86.9%	79.7%	82.6%
Grammatical Structure	13	320	2	19	90.5%	70.0%	76.0%
Verb Phrase	51	252	22	29	79.8%	77.9%	78.7%
Dictionary Terms (Abbrv. Acronyms)	41	302	2	9	96.2%	90.8%	93.2%
Full Words	5	345	2	2	81.0%	91.4%	85.4%
Idioms/Slang	0	354	0	0	NA	NA	NA
Prefix/Suffix	4	349	0	1	99.9%	90.0%	94.4%
Length	0	354	0	0	NA	NA	NA

Table 5 Terminal Image Viewer method name evaluations, and average of precision, recall, and f score

Violation	TP	TN	FP	FN	Avg Precisio n	Avg Recall	Avg F1 Score
No Violation	TN	TP	FN	FP			
Naming Style	69	187	1	13	96.0%	91.8%	93.6%
Grammatical Structure	16	248	0	6	98.8%	86.4%	91.5%
Verb Phrase	81	157	10	22	88.4%	86.3%	87.1%
Dictionary Terms (Abbrv. Acronyms)	119	140	10	1	95.8%	96.3%	95.9%
Full Words	8	254	0	8	98.5%	75.0%	82.6%
Idioms/Slang	0	270	0	0	NA	NA	NA
Prefix/Suffix	0	270	0	0	NA	NA	NA
Length	0	270	0	0	NA	NA	NA

The average precision of adherence and non-adherence to the naming style standard is 86.9%, with an average recall of 79.7%. There is an average precision of 90.5%, an average recall of 70.0% for the grammatical structure, an average precision of 79.8%, and an average recall of 77.9% for the verb phrase standard. The accuracy for checking dictionary terms standard adherence reached an average precision of 96.2% and an average recall of 90.8%. There is an average precision of 81.0% and average recall of 91.4% for adhering to the full words standard. Additionally, the prefix/suffix standard received an average precision of 99.9% and average recall of 90.0%. Overall observation of this system shows that our approach can achieve decent accuracy (i.e., average precision, recall, and F-score) numbers per standard (+70%) compared against the golden set. These results led us to take another step to verify the results of the approach and ensure that these results generalizable to other systems.

Table 6 BIO JAVA Method name evaluations, and average of precision, recall, and f score

Violation	TP	TN	FP	FN	Avg Precision	Avg Recall	Avg F1 Score
No Violation	TN	TP	FN	FP			
Naming Style	12	342	8	1	79.9%	95.0%	85.7%
Grammatical Structure	7	355	0	1	99.9%	93.8%	96.6%
Verb Phrase	22	313	15	13	77.7%	77.4%	78.4%
Dictionary Terms (Abbrv. Acronyms)	90	268	2	3	98.7%	98.0%	98.2%
Full Words	26	333	4	0	93.3%	99.4%	96.1%
Idioms/Slang	0	363	0	0	NA	NA	NA
Prefix/Suffix	0	363	0	0	NA	NA	NA
Length	6	357	0	0	100.0%	100.0%	100.0%

Table 7 Flash Develop method name evaluations, and the average of precision, recall, and f score

Violation	TP	TN	FP	FN	Avg Precision	Avg Recall	Avg F1 Score
No Violation	TN	TP	FN	FP			
Naming Style	62	308	5	1	96.1%	98.4%	97.2%
Grammatical Structure	20	353	0	3	99.9%	93.5%	96.3%
Verb Phrase	82	274	10	10	92.8%	92.8%	92.8%
Dictionary Terms (Abbrv. Acronyms)	74	281	20	1	89.9%	96.0%	92.0%
Full Words	8	354	2	12	88.4%	69.7%	75.7%
Idioms/ Slang	0	376	0	0	NA	NA	NA
Prefix/Suffix	2	346	0	28	96.3%	53.3%	54.3%
Length	5	371	0	0	100.0%	100.0%	100.0%

7.2 Terminal Image Viewer (270 methods)

We ran our approach on a significant sample of the Terminal Image Viewer system. We inspect the results and observe that the method names of this system appear to be of lower quality than Notepad++. For example, several methods did not adopt any naming style, thus violating this standard. Also, there are several method names missing a verb. We hypothesize this is due to the system being in an early stage of development (release 1.1 with ~170 commits) compared to Notepad++ (release 7.9 with +4K commits). After assessing the method names of this system, our approach can correctly detect several poorly written methods (see Table 5). For example, for the naming style standard, a total of 82 of the examined method names have naming style violations, and our approach correctly detects 69 of these cases (84%). Our approach detects 73% of the methods that violate the grammatical structure standard. For the verb phrase standard, it can detect 79% of the violations. Also, there are 120 method names in violation of the dictionary terms standard, and the approach detects 119 of them (99%). For the full word standard, the approach identifies 50% of the violations. The overall results show that the approach can identify many true positives (violation view), with issues per a naming standard. This system's precision and recall numbers for the standards adherents and non-adherents are also consistent with the previous system.

7.3 BioJava (363 methods)

Overall, the evaluation of BioJava demonstrates that the approach can also recognize Java method names that have violations (TP, violation view) (see Table 6). There are 13 method names that violate the naming style standard, and 12 are correctly reported. There are 8 methods that have violations with the grammar standard, and the approach can correctly recognize 7 of them. The approach found 63% of the methods that violate the verb phrase standard. There are 93 methods that contain non-dictionary terms, and the approach correctly identifies 96.8% of them. There are also 26 method names that violate the full words standard in this system, where the approach recognizes all of them. While there are no violations of using idioms/slang and prefix/suffix on this system, there are 6 length violations that are all identified by our approach. The average precision and recall are +77% for reporting standards adherence/nonadherence in this system.

7.4 Flash Develop (376 methods)

The results on Flash Develop appear in Table 7. True-positives (violation view) detections are overly high per each standard. The approach detects 98% of the method names that violate the naming style, 89% of the ones violating the verb phrase standards, and 87% of the methods with grammatical structure violations. On the other hand, 75 method names contain non-dictionary terms for the dictionary terms standard, and 74 of those are detected (99%). All length violating methods are also detected, and we found no idioms/slang violations. However, the approach only detects 40% of the full words violations and 2 of 30 violations to the prefix/suffix standard. We discuss these more in depth in the discussion section. This aspect of the evaluation shows that the approach is effective on different systems and appears to generalize to different languages, specifically to C++, C#, and Java. However, these languages have the same historical roots and similar general syntax. Additional study is needed to see if the approach generalizes to other, less similar languages.

8 Discussion

This section discusses the quantitative and qualitative results of our evaluation and golden set construction. We go over several nuances that highlight the difficulty of the problem and complications brought about by the tools we rely on. In addition, our manual validation of the false-positive instances shows a repeating pattern that, in most cases, causes the approach to misreport them as issues. These issues are also related to low average precision and recall the approach has.

8.1 Naming Style

During the development of the golden set, a case arose where three evaluators argued that the first letter that comes after the acronym should be capitalized, while two others argued it should not. Our approach flags such method names with a naming style violation, e.g., `ColouriseAPDLDoc()`, `FoldVHDLDoc()`, and `FoldMSSQLDoc()` only if the acronyms are not added to the exception list. If it is added, no violations are reported. This applies to all the similar naming style false positives found in the other three systems. We observe that the naming style is associated with the dictionary terms standard; so, if there is an acronym in a method that is not added to our dictionary, and acronyms usually are all capitalized, the approach flags these method names with two violations, a naming style, and a dictionary terms violation.

The approach is capable of detecting inconsistent or missing naming styles. While not every case of a naming violation will degrade comprehension, it increases the chance of additional cognitive load.

Our evaluation of this standard shows that most of the false-negative cases across the four systems are due to splitting issues. The approach cannot identify a naming style violation in examples like `GetTypesep()`. In this case, the name is supposed to have a capital S to adhere to PascalCase, so `Type` and `sep` can be recognized as two different words per the golden-set evaluation. However, the approach misreports these violations. The same applies to the method name `isLispwordstart()` that appears in Notepad++.

There are some interesting naming style practices in the C# system that we did not observe commonly in the other systems. There appear +20 methods names that have a mixed naming style. Developers of this system tend to use camelCase or a PascalCase with an underscore case. Examples include:

```
addLibraryButton_CheckedChanged()
TreeIcons_Populate()
```

We also observed when developers use prefixes, they tend to use all caps letters style, for example: `mVALUE_int()`, `mSTART_TAG()`, `mIDENTIFIER()`, and `mCOMMENT()`. We are not sure if this practice is due to a naming style guideline provided to the developers of this project, but we are certain that inconsistent naming style does not support method readability according to the literature [14, 32, 48, 49]. Thus, we consider these violations.

8.2 Verb Phrase

Most of the false positives for the verb phrase standard relate to the performance of the tagger. There are some methods that our part-of-speech tagger is not able to annotate correctly. For example, the verbs in the method names:

```
CountCharacterWidthsUTF8()
GetRelativePositionUTF16()
```

that appear in Notepad++ are not correctly recognized by our approach. After investigation, we found the tagger labeled the first words as nouns.

There are some false positive cases where "is" (as the verb) of the method is not recognized as a verb but as a verb modifier. For example: `isIndentToFirstParm()`, `isInListA()`,and `IsMyThread()`. These examples are incorrectly flagged with a verb phrase violation. These false positive cases highly depend on how the tagger interprets the name. Another case where the verb is not recognized is when not using a naming style such as in `fseek()`, where `seek` is a verb, but it was not recognized because the approach reads this name as a full word.

A verb must be present within a method name. Detecting the verb is challenging; some words can be both a noun and a verb.

For the false negatives, the reverse occurs. The approach, in some cases, annotates the nouns as verbs. For example, in the method names `RangeText()` and `CodePageFamily()`, the tagger labels `Range` and `Code` as verbs, while in the golden-set evaluation, evaluators agree those are nouns. The mistake is likely because the words "Range" and "Code" function as verbs in some contexts. More examples are `row()` and `rows()`, and the approach does not flag these words with verb phrase violation as they are tagged as verbs.

8.3 Grammatical Structure

The approach can identify many violations of the grammatical structure standard. For example, cases where method names start with a preposition such as `IntoCreateViewStatement()` or end with a preposition such as `syncFoldStateWith()`, `convertInto()`, `endsWith()`, `sizeTo()`, `activateWindowAt()`, or `MoveTo()`, or also end with a pronoun or a determiner such as in `WeakNotifyThis()`. These method names are flagged by the golden-set evaluators as sub-optimal grammatical structures, as they do not form a complete phrase satisfying the grammatical structure standard definition. They are recognized all as a violation.

During the construction of the golden set, there were some disagreements about method names that end with nouns that could also be interpreted as verbs. For example, the method names `WordListSet()`, `FineTickerCancel()`, `MoveForInsertDelete()`, `InputSymbolScan()`, and `ContractionStateCreate()` have diverging opinions. In our approach they are identified as nouns.

In general, challenges to detecting violations can be classified into the following three cases:

1) The verb comes at the end of the method name, so it is difficult to determine if the name is a verb phrase

or noun phrase. Examples: `FineTickerCancel()`, `SelectionEnd()` and `InputSymbolScan()`.

2) The method name is not clear, and therefore it is not comprehensible. While the original developers most likely understand the name, it is difficult for the non-informed to decipher. Example: `lookingAtHereDocDelim()`.
3) The method name ends with an abbreviation, and it is unclear whether it is a verb or a noun. For example, `numstrcmp()`. In this case, one can question if `cmp` stands for comparison or compare.

Dangling prepositions and determiners are somewhat common patterns in names. However, these are incomplete phrases. Developers need to consider whether using incomplete phrases is appropriate even if their parameters/arguments can complete the phrase. In addition, we found several examples of verbs appearing at the end of method names when placing them at the beginning is better syntax.

There are false-negative cases where adding a preposition in the middle of a name supports better readability. For example, the name `DrawTextNoClip()` can be improved if we add `with` as a preposition: `DrawTextWithNoClip()`. However, currently our approach does not support checking for preposition occurrences at the middle of a phrase. Our general observation for this standard shows that as long as a verb exists in the method name, it is usually understandable to the developer, even if not optimal.

8.4 Dictionary Terms

Most of the false positives for the dictionary terms standard are due to the splitter performance. For example, in cases where a capitalized single letter (i.e., Articles) is followed by a word that starts with a capital letter (e.g., `IsASpace()` from Notepad++), the approach flags it with a dictionary term violation as it evaluates "pace" as a non-dictionary term.

When method names do not adopt any naming style, e.g., `wordchar()`, the approach treats them as one word and flags them as non-dictionary terms. However, in other cases where a method contains a component word such as `whitespace` in `IsNextNonWhitespace()`, the approach recognizes such word as one word and does not flag it as a violation, as also the golden-set evaluators agree that `whitespace` is a full dictionary word.

Developers are highly recommended to use the full form of a word while naming their methods. Unknown abbreviations in source code are common in non-source code contexts, which can cause a misunderstanding not only to other developers but also to software text analysis tools that use English dictionaries.

Some terms in the method names appear in the WordNet dictionary, but they have a different meaning from how they are used in source code. For example, the method names `nbScintillas()`, and `testGetNcsMatrix()`, where the evaluators agree that the abbreviations "nb" and "ncs" are not dictionary terms. However, WordNet contains those abbreviations as full words and defines these words in unrelated contexts. The evaluation results show that these cases are false negatives.

8.5 Full Words

Most of the false negatives of the full words standard are due to the splitting issue at handling single letters. For example, with the method `getLParamFromIndex()`, the approach is not able to recognize the single letter violation as it recognizes LP as a dictionary word. It flags such a method with a naming style and dictionary terms violations, but not a full words violation. Also, there are false negatives where the names are not flagged with full words violations, as developers do not use any naming styles to differentiate between the words and single letters included in their method names. Examples include `fdate()`, `fseek()`, and `fsize()`, which appear in TerminalImageViewer. They all received a score of 5 for violating the naming style, dictionary terms, and verb phrase standards but not for the full words standard. The false positives in this evaluation include:

a) Single letters used to differentiate lists, collections, or groups, such as `isInListA()`.
b) Single letters are used as a name of a programming language, such as `FoldOScriptDoc()`.
c) Single letters used in boolean functions, such as `isADigit()`.

When there is no naming style used to differentiate the constituting of words for a method, it is not only difficult for the developer to read and disassemble *such names, but it can cause more cognitive challenges as well if they have single letters that are not clear what they stand for. Single letter abbreviations have the most potential expansions [50][51][52] and, therefore, are the most difficult for both humans and machines to parse if they do not know the expansion beforehand.*

8.6 Idiom and Slang

There are no false positives and negatives during our evaluation based on our idiom and slang list. We believe that the only case where a false positive can occur is when there is an acronym from the system domain (formally a dictionary word) that is in our idiom and slang list. E.g., the idiom ASAP ("As Soon As Possible").

8.7 Prefix and Suffix

The results show that the approach can correctly identify prefixes/suffixes in the Notepad++ and Flash Develop systems according to our prefixes and suffixes list, with no false positives. However, there are many false negatives when a method name starts or ends with a single letter that is a prefix of a method. In such cases, the approach does not flag the name with a prefix or suffix violation; but flags it with full words violation. For example, the method name: `u_iswalpha()` from Notepad++ and `mCOMMENT()`, `mUSE()`, and `mUNDERSCORE()` from the FlashDevelop system. This naming practice explains why we get a low average recall of 53% in the Flash Develop system.

8.8 Length

Our approach has no issue with identifying any length standard violations. But because of possible errors when splitting, it is possible for it to occur. We set 7 words as the maximum length a method name can have. During the golden-set evaluation, evaluators manually validated the method names for any method exceeding 7 words. Although there are no positive cases in the C++ systems, we did find some length violations in the C# and Java systems. In the C# systems there are a few method names that contain up to ten words, and all are correctly identified with a length violation. These examples received a score of 9:

```
getMetaTagsToKeepOnSameLineAsFunction()
getAdvancedSpacesAfterColonsInDeclarations()
```

In the BioJava system, 6 method names violate the length standard. However, according to the comments provided in the system code, those are all test method names.

Method names need to be descriptive but must also strive to avoid overloading the reader. Prior studies indicate that 7 is at or near the upper bound during reading/comprehending a method name.

Across all the evaluated systems, the dictionary terms and verb phrase standards have the highest number of violations. For the dictionary terms standard, it is mainly due to developers using abbreviations and acronyms that are uncommon or non-dictionary terms. For the verb phrase standard, previous literature emphasizes that including a verb to refer to the action of a method is vital, but according to the results missing a verb in a name is common across systems. We believe increasing the awareness about the importance of this naming aspect among developers will support better-naming practices, resulting in more comprehensible names. The naming style standard also had a high number of violations. This resulted in splitting issues and incorrect identification of a method's constituting words. In general, these three standards are extremely critical in supporting the readability and comprehension of a name.

Broadly, our approach shows that it can successfully detect adherent and non-adherent methods to the ten method naming standards to a great extent. The approach records high average precisions, recalls, and F-scores across the evaluated systems. We believe that these high numbers are not only due to implementing very fine grain method naming characteristics; but also due to using source code specialized tools, i.e., a source code tagger and a splitter which greatly assist in analyzing the different components of a given name.

9 Threats to Validity

Our evaluation is on multiple systems written in different programming languages, and showing consistent results confirms the generalizability of our approach. The approach can be used with any project srcML supports. C++/C, C#, and Java projects in this version of the approach are verified. To maintain validity, the approach is implemented by the first author, and the golden set is constructed by the other authors.

It is possible that the part-of-speech tagger we used in this study could cause some threats to the results. However, the tagger used represents the current state-of-the-art tagger for source-code identifiers, and furthermore, the false-positives and false-negatives were very few. The approach can correctly detect a high number of non-violating cases without issues. We discussed earlier that some of the false-positives in the verb phrase and grammatical structure standards are caused by the tagger performance. There were cases in which some of the words in method names are verified dictionary words, and after checking these words' definitions, they are of a different context than source code. We believe that using a specialized computer science/software engineering dictionary will support our approach by reducing those minor false-negatives. Also, it is possible that the methods tested in this study include a few test methods, although most of them adhered to our standards and received an average quality score above 8 with violations of the length standard. Additionally, while some idioms and slang could exist that we are not familiar with, it is possible to add these phrases.

As the approach is implemented by one of the authors involved with the creation of the golden set, this might constitute a threat. Mitigating this, the approach is applied to and manually evaluated independently against three additional systems.

10 Conclusion

We present an approach for assessing the quality of method names according to the method naming standards formally investigated and evaluated in [14]. The approach implements all ten standards. Further, it is 1) lightweight, only requiring publicly available NLP tools. 2) the scores it produces can be easily used to understand how many flaws an identifier exemplifies, helping highlight identifiers that may be more problematic and 3) it is explainable—the feedback this tool provides is based on well-reasoned, data-driven standards that can help developers make informed decisions about whether their method names should be updated or not. The technique is complementary to other method appraisal approaches such as linguistic antipatterns, inviting the opportunity to combine tools. Our future work involves adding more details about the grammatical structures underneath the method name using the Identifier Name Structure Catalogue [31]. Another avenue of research is to use method stereotype information [53] to fine tune the standards for a particular type of method (getter vs. command).

REFERENCES

[1] F. Deisenbock and M. Pizka, "Concise and Consistent Naming," in 13th International Workshop on Program Comprehension (IWPC'05), St. Louis, MO, USA, 2005, pp. 97–106.

[2] T. M. Pigoski, Practical software maintenance: best practices for managing your software investment. New York: Wiley Computer Pub, 1997.

[3] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Relating Identifier Naming Flaws and Code Quality: An Empirical Study," in 2009 16th Working Conference on Reverse Engineering, Lille, France, 2009, pp. 31–35.

[4] S. Butler, M. Wermelinger, Yijun Yu, and H. Sharp, "Exploring the Influence of Identifier Names on Code Quality: An Empirical Study," in 2010 14th European Conference on Software Maintenance and Reengineering, Madrid, 2010, pp. 156–165.

[5] L. Pollock, K. Vijay-Shanker, E. Hill, G. Sridhara, and D. Shepherd, "Natural Language-Based Software Analyses and Tools for Software Maintenance," in Software Engineering, vol. 7171, A. De Lucia and F. Ferrucci, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 94–125.

[6] E. W. Høst and B. M. Østvold, "Debugging Method Names," in 23rd European Conference on ECOOP 2009 — Object-Oriented Programming, Genoa, Italy, 2009, pp. 294–317.

[7] P. A. Relf, "Tool assisted identifier naming for improved software readability: an empirical study," in 2005 International Symposium on Empirical Software Engineering, 2005., Queensland, Australia, 2005, pp. 52–61.

[8] B. Boehm and V. R. Basili, "Software Defect Reduction Top 10 List," Computer, vol. 34, no. 1, pp. 135–137, Jan. 2001.

[9] J. C. Hofmeister, J. Siegmund, and D. V. Holt, "Shorter identifier names take longer to comprehend," Empirical Software Engineering, vol. 24, no. 1, pp. 417–443, Feb. 2019.

[10] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a Name? A Study of Identifiers," in 14th IEEE International Conference on Program Comprehension (ICPC'06), 2006, pp. 3–12.

[11] 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.

[12] B. Liblit, A. Begel, and E. Sweetser, "Cognitive Perspectives on the Role of Naming in Computer Programs," in Proceedings of the 18th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2006, Brighton, UK, September 7-8, 2006, 2006, p. 11.

[13] V. Arnaoudova, M. Di Penta, and G. Antoniol, "Linguistic antipatterns: what they are and how developers perceive them," Empir Software Eng, vol. 21, no. 1, pp. 104–158, Feb. 2016.

[14] R. S. Alsuhaibani, C. D. Newman, M. J. Decker, M. L. Collard, and J. I. Maletic, "On the Naming of Methods: A Survey of Professional Developers," in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), Madrid, Spain, 2021, pp. 587–599.

[15] S. Butler, M. Wermelinger, and Y. Yu, "Investigating naming convention adherence in Java references," in 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), Bremen, Germany, 2015, pp. 41–50.

[16] "Checkstyle." [Online]. Available: https://checkstyle.sourceforge.io.

[17] "Java Coding Standard Checker (JCSC)." [Online]. Available: https://sourceforge.net/projects/jcsc/.

[18] A. Peruma, E. Hu, J. Chen, E. A. AlOmar, M. W. Mkaouer, and C. D. Newman, "Using Grammar Patterns to Interpret Test Method Name Evolution," in 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), Madrid, Spain, 2021, pp. 335–346.

[19] J. Wu and J. Clause, "A pattern-based approach to detect and improve non-descriptive test names," Journal of Systems and Software, vol. 168, p. 110639, Oct. 2020.

[20] B. Zhang, E. Hill, and J. Clause, "Automatically Generating Test Templates from Test Names (N)," in 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, 2015, pp. 506–511.

[21] B. Zhang, E. Hill, and J. Clause, "Towards automatically generating descriptive names for unit tests," in Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore Singapore, 2016, pp. 625–636.

[22] J. Koenemann and S. P. Robertson, "Expert problem solving strategies for program comprehension," in Proceedings of the SIGCHI conference on Human factors in computing systems Reaching through technology - CHI '91, New Orleans, Louisiana, United States, 1991, pp. 125–130.

[23] T. A. Corbi, "Program understanding: Challenge for the 1990s," IBM Syst. J., vol. 28, no. 2, pp. 294–306, 1989.

[24] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Effective identifier names for comprehension and memory," Innovations in Systems and Software Engineering, vol. 3, no. 4, pp. 303–318, 2007.

[25] S. Fakhoury, Y. Ma, V. Arnaoudova, and O. Adesope, "The effect of poor source code lexicon and readability on developers' cognitive load," in Proceedings of the 26th Conference on Program Comprehension - ICPC '18, Gothenburg, Sweden, 2018, pp. 286–296.

[26] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, "On the Comprehension of Program Comprehension," ACM Trans. Softw. Eng. Methodol., vol. 23, no. 4, pp. 1–37, Sep. 2014.

[27] D. Feitelson, A. Mizrahi, N. Noy, A. Ben Shabat, O. Eliyahu, and R. Sheffer, "How Developers Choose Names," IIEEE Trans. Software Eng., pp. 1–1, 2020.

[28] V. Arnaoudova, M. Di Penta, G. Antoniol, and Y.-G. Gueheneuc, "A New Family of Software Anti-patterns: Linguistic Anti-patterns," in 2013 17th European Conference on Software Maintenance and Reengineering, Genova, 2013, pp. 187–196.

[29] A. Peruma, V. Arnaoudova, and C. Newman, "IDEAL: An Open-Source Identifier Name Appraisal Tool," in 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2021.

[30] P. Relf, "Achieving Software Quality through Source Code Readability." 01-Jan-2004.

[31] S. Butler, M. Wermelinger, and Y. Yu, "A Survey of the Forms of Java Reference Names," in 2015 IEEE 23rd International Conference on Program Comprehension, Florence, Italy, 2015, pp. 196–206.

[32] R. S. Alsuhaibani, C. D. Newman, M. J. Decker, M. L. Collard, and J. I. Maletic, "A Survey on Method Naming Standards: Questions and Responses Artifact," in 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Madrid, ES, 2021, pp. 242–243.

[33] 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.

[34] J. I. Maletic and M. L. Collard, "Exploration, Analysis, and Manipulation of Source Code using srcML," presented at the 37th International Conference on Software Engineering - Volume 2, Florence, Italy, 2015.

[35] Michael L. Collard and Jonathan I. Maletic, "srcML 1.0: Explore, Analyze, and Manipulate Source Code," presented at the 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME), Raleigh, NC, USA, 02-Oct-2016.

[36] J. Gosling, B. Joy, G. L. Jr. Steele, G. Bracha, A. Buckley, and G. L. S. Jr, The Java Language Specification, Java SE 8 Edition, 1 edition. Addison-Wesley Professional, 2014.

[37] A. Vermeulen, The Elements of Java- Style, Reprint edition. Cambridge ; New York: Cambridge University Press, 2000.

[38] C. D. Newman et al., "On the generation, structure, and semantics of grammar patterns in source code identifiers," Journal of Systems and Software, vol. 170, p. 110740, Dec. 2020.

[39] C. D. Newman et al., "An Ensemble Approach for Annotating Source Code Identifiers with Part-of-speech Tags," TSE, vol. Under Review, Jun. 2021.

[40] 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 2013 21st International Conference on Program Comprehension (ICPC), San Francisco, CA, 2013, pp. 3–12.

[41] E. Hill, "A model of software word usage and its use in searching source code," Ph.D. thesis, University of Delaware, 2010.

[42] 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.

[43] G. A. Miller, "WordNet: a lexical database for English," Commun. ACM, vol. 38, no. 11, pp. 39–41, Nov. 1995.

[44] M. Hucka, "Spiral: splitters for identifiers in source code files," The Journal of Open Source Software, 04-Apr-2018. [Online]. Available: https://doi.org/10.21105/joss.00653. [Accessed: 19-Feb-2019].

[45] E. Hill, D. Binkley, D. Lawrie, L. Pollock, and K. Vijay-Shanker, "An empirical study of identifier splitting techniques," Empirical Software Engineering, vol. 19, no. 6, pp. 1754–1780, Dec. 2014.

[46] R. C. Martin, Ed., Clean code: a handbook of agile software craftsmanship. Upper Saddle River, NJ: Prentice Hall, 2009.

[47] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," Information Processing & Management, vol. 45, no. 4, pp. 427–437, Jul. 2009.

[48] D. Binkley, M. Davis, D. Lawrie, and C. Morrell, "To camelcase or under_score," in 2009 IEEE 17th International Conference on Program Comprehension, 2009, pp. 158–167.

[49] B. Sharif and J. I. Maletic, "An Eye Tracking Study on camelCase and under_score Identifier Styles," in 2010 IEEE 18th International Conference on Program Comprehension, Braga, Portugal, 2010, pp. 196–205.

[50] Y. Jiang, H. Liu, Y. Zhang, N. Niu, Y. Zhao, and L. Zhang, “Which abbreviations should be expanded?,” in Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens Greece, 2021, pp. 578–589.
[51] 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.
[52] C. D. Newman, M. J. Decker, R. S. Alsuhaibani, A. Peruma, D. Kaushik, and E. Hill, “An Empirical Study of Abbreviations and Expansions in Software Artifacts,” in 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), Cleveland, OH, USA, 2019, pp. 269–279.
[53] N. Dragan, M. L. Collard, and J. I. Maletic, “Reverse Engineering Method Stereotypes,” in 22nd IEEE International Conference on Software Maintenance (ICSM’06), 2006, pp. 24–34.