Heuristic-Based Part-of-Speech Tagging of Source Code Identifiers and Comments

Reem S. AlSuhaibani
Computer Science
Kent State University
Kent, OH 44240 USA
ralsuhai@kent.edu

Christian D. Newman
Computer Science
Kent State University
Kent, OH 44240 USA
cnewman@kent.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, 44240 USA
jmaletic@kent.edu

***Abstract*—An approach for using heuristics and static program analysis information to markup part-of-speech for program identifiers is presented. It does not use a natural language part-of-speech tagger for identifiers within the code. A set of heuristics is defined akin to natural language usage of identifiers usage in code. Additionally, method stereotype information, which is automatically derived, is used in the tagging process. The approach is built using the srcML infrastructure and adds part-of-speech information directly into the srcML markup.**

***Index Terms*— Natural Language Processing, part-of-speech tagging, identifier analysis, program comprehension.**

I. Introduction

With 60-90% of software life cycle resources spent on program maintenance [1, 2], there is a critical need for advanced tools that help exploration and comprehension of today's large and complex software. To reduce the cost of this software maintenance, it has been demonstrated that natural-language clues in program identifiers can be used to improve software tools [3]. There have been a number of attempts to apply natural language processing (NLP) techniques to source code to support various program comprehension tasks. In the work presented here we are particularly interested in determining the part-of-speech of identifiers (i.e., names of functions, types, variables, etc.) in source code. We view this as a separate problem from determining the part-of-speech of comments. Comments are typically written in a natural language (English) and often have sentence structure that follows grammatical rules [4-6].

Part of Speech (PoS) taggers, for natural language, leverage large amounts of knowledge about English words and their usage in sentences. Thus, they work fine for typical English prose but lacking sentence structure, these methods break down. Additionally, the manner in which programmers use an identifier in a program is very different than how a writer uses a word in a sentence. The grammar is different and the semantics of the identifiers are typically specialized for the domain of software. While we do not deny that there is some correspondence between an identifier and its English counterpart, drawing a direct comparison is clearly flawed.

As such, we feel a more appropriate approach is to define the part-of-speech in terms of how an identifier is used in the code rather than how it would be used in prose. Similar techniques have been used by others [7-9]. For example, we could simply mark all function names as verbs and variables/objects as nouns. Names of functions typically describe an action (on an object or parameter). Likewise, variables are typically nouns that describe an object in the domain.

Of course, this heuristic is overly simplistic and does not take into consideration a wealth of relevant information that can be derived (statically) from the context of an identifier within the source code. Much like how natural language PoS taggers use the context of the word in a sentence, we must use the context of the identifier in the program.

Given this viewpoint, we propose a set of heuristics that define the part of speech of identifiers in source code. That is, we have taken terms from part of speech (e.g., noun, verb, etc.) and defined them, using heuristics, in the context of source code. Thus instead of assigning PoS to identifiers based on the word's usage in English prose, we assign it based on its use in source code. The goal of this work is to produce a specialized PoS tagger for source code. This would be used in conjunction with a NLP PoS tagger for the internal comments.

The paper is organized as follows. In next section we briefly discuss the related work on using PoS taggers for source code. In section III we discuss our approach and define our heuristics. In section IV gives implementation details followed by some preliminary.

II. Related Work

There are a number of tools and techniques for part-of-speech tagging from the NLP community [10-16]. The work we do here is related to any part-of-speech tagging technology. Since the aim is to increase accuracy within and across systems, it is more closely related to [9, 17], which heavily focus on increasing the accuracy of PoS techniques. The best current approaches find the correct tag with an accuracy of 80% to 95%. Their results are quite good, however these taggers take the English usage of these words to provide which part-of-speech they belong to, while our hypothesis is that this may not be the most effective way of looking at the problem. Our work differs from theirs primarily in that we are taking the perspective of how identifiers are used in source code and providing a part-of-speech based on that. We believe that, with further research, this may reveal more latent meaning both within and across multiple systems for identifier usage.

III. Approach for PoS Tagging

Coming up with heuristics in the spirit of English part-of-speech terms involved the same activities as in linguistics. Our focus is to determine how an identifier expresses the intent of the entity it represents. So if an identifier is being characterized as a verb, that identifier must represent some sort of action. If an identifier is being tagged as a noun, it must represent some sort of object in the system. To accomplish this, we need some information about how an identifier is defined, how it is used, and its context.

To get this information, we use srcML [18]. srcML is a markup language that wraps source code with AST information. Hence, it allows us to examine statically computable information about source code. Alongside srcML, we use stereotypes [19]. Stereotypes give the user information about how a function is being used. Examples of stereotypes are giving in Table 1. Primarily, stereotypes are used by our heuristics to figure out how a function behaves with respect to its arguments, local variables, and the calling object (when applicable). In fact, our technique is related to stereotypes. Where as stereotypes categorize at the method and class level, our technique categorizes at the identifier level.

We define our heuristics by presenting a part-of-speech term and then state its definition with respect to source code as opposed to its use in English. As they are heuristics, there is room for debate on whether they are completely correct or not. This is why we feel it is worth noting again that these heuristics are a first step towards properly marking part of speech in source code with respect to source code. The intention is that, with these base heuristics, more data may be collected and tighter definitions obtained using techniques already employed in the NLP community.

The following is our definitions of part-of-speech terms. We first summarize the motivation behind each definition and then give a list of rules that an identifier must satisfy to be classified under the given term. Variables are assigned a PoS when they are declared, since at that time the type and name is known. One exception is function identifiers (names), where the PoS is assigned when they are defined, since the assignment is based on the stereotype of the definition.

A. Source Code PoS Heuristics

The primary way to collect, relate, and move data in an object-oriented system is to use objects. Objects are represented by identifiers with a unique name. This is analogous to *proper nouns* in English, which represent the names of unique objects; a person's name, a location's name. An identifier is a *proper noun* if it satisfies the following rules:

- It names a first class user-defined object.
- The identifier does not appear as a member of another class (i.e., not part of a composition relationship).

A *noun* that is not a proper noun specifies a set of objects that do not have their own unique identity. These are words like building, food, and apple. In objected-oriented code, these are akin to identifiers that represent objects that make up part of the composition for a larger unique object. An identifier in source code is a *noun* if it satisfies the following rules:

- It names a first class user-defined object.
- The identifier appears as a member of another class (i.e., it is used in a composition relationship; this is where they differs from proper nouns).

Adjectives in English describe nouns; a person's hair color, the age of a city. In source code, these are identifiers whose primary purpose is to convey a characteristic of something; its size/length, whether it is true or false; a radius; a file handle, etc. Primitive types are often used for this purpose, particularly primitives that make up part of a class. For this reason, all identifiers whose types are primitive are considered *adjectives*. An identifier is an *adjective* if it satisfies the following rules:

- The type that the identifier's value represents is primitive (e.g., `int`, `float`, `bool`). Note that the type of a function's identifier is its return type.
- If the identifier represents a function, then it further satisfies the constraint that it does not apply any modifications to any of:
    - Its aliased arguments (i.e., does not modify any references or pointers)
    - The calling object (i.e., `this`)
- The identifier does not represent a pointer, reference, const reference or an array.

```
class Person{
public:
  float returnAge()                     //adjective
      {return curyear - yearborn;}
  void SetName(std::string n)           //verb
      {name = n;}
private:
  int age;                              //adjective
  std::string name;                     //noun
  date yearborn;                        //noun
};
vector<string> split                    //verb
         (const string& str){           //pronoun
  vector<string> result;                //proper noun
  ...
  return result;
}
```

Figure 1. An example of applying heuristics.

Pronouns are used as references nouns or proper nouns, e.g., the word "she" can refer to any female person. In source code, these are akin to reference variables and pointers. An identifier is a *pronoun* if it satisfies the following rules:

- It names a pointer or reference to a user-defined object or primitive value.
- It is not `const` (it can be pointed at something else)

Verbs represent actions in English; you run, you kick a ball, you play a game, etc. In source code, verbs modify the state of the system. An identifier is a *verb* if it satisfies the following:

- It is the name of a function that applies some modification to at least one of three things:
    - One or more of its arguments
    - The calling object (`this`)
    - Locally variable whose value is then returned.

- It is not `const` in both its return type and the calling object (i.e., it has to perform some useful modification).

An example of how these heuristics can be applied to source code is given in Figure 1. Notice that only identifiers are tagged; types are not tagged and class names are not tagged (though, in the future, we are considering making them nouns).

B. PoS and Method Stereotypes

We now discuss how we use method stereotypes to further refine the PoS for identifiers. Method stereotypes categorize methods based on their role in a given system. They are not based on the name of the method, but on static analysis of the code in the method. We refer the reader to [20, 21] for a complete definition. The list of stereotypes is provided in Table I. We now briefly discuss each category.

Structural methods provide and support the structure of the class. For example, accessors read an object's state while mutators change it. The identifier for a structural method corresponds primarily with adjectives because these types of methods are asking about some characteristic of the object they are part of (isEmpty, getName, etc). In the case of a mutator, however, they can also be verbs. *Creational* methods create or destroy objects of the class. These correspond primarily to verbs; they completely construct an object thereby changing the progam's state. *Collaborational* methods characterize the communication between objects and how objects are controlled in the system. We primarily mark these as verbs, but future work will investigate if there are more complex patterns. Particularly, how verbs are applied to their intended target (the subject, which could be an argument, the calling object, etc). *Degenerate* are methods give us little information about. We fall back on purely applying our heuristics in this case.

Since a method may have more than one stereotype, a small finite state machine implements the rules for how to assign a tag based on stereotypes. The approach is naïve and requires further research to be refined, but the implementation uses stereotypes to differentiate between Structural Accessors and everything else. Anything combined with a Structural Accessor ends up as an adjective or a noun (since state is not modified by the method but may be modified outside of the method in the case where a member is returned and not `const`). All other combinations are currently verbs. We are very conservative about this; if the stereotype makes it unclear what is going on we fall back on our rules for general identifier mark-up.

C. PoS Tagging of Comments

The heuristics deal with source code identifiers and do not directly apply to comments. srcML marks comments with a tag but does not do any other parsing of the comments (e.g., into sentences). Therefore we use the *Natural Language Toolkit* (NLTK) [22, 23]. When a comment is encountered in the code, we characterize it using NLTK and each word in the comment is given PoS tag. Identifiers from the code that are used in comments are tagged with the code PoS. This is possible since we are able to take multiple passes over the code, and make identifier PoS consistent in comments to their usage in the code.

TABLE I. TAXONOMY OF METHOD STEREOTYPES AND THEIR CORRESPONDING PART OF SPEECH.

Stereotype Category	Stereotype	Description	Part-of-Speech
Structural *Accessor*	get	Returns a data member.	Adjective/noun depending on return type
	predicate	Returns Boolean value that is not a data member.	Adjective
	property	Returns info about data members.	Adjective
	void-accessor	Returns information via a parameter.	Adjective
Structural *Mutator*	set	Sets a data member.	Verb
	command	Performs a complex change to the object's state.	Verb
	non-void-command		Verb
Creational	constructor, copy-const, destructor, factory	Creates and/or destroys objects.	Verb
Collaborational	collaborator	Works with objects (parameter, local or return value).	Verb
	controller	Changes only an external object's state (not *this*).	Verb
Degenerate	incidental	Does not read/change the object's state.	N/A
	Empty	Has no statements.	N/A

D. Discussion

Given the heuristics, it is clear our approach cannot be validated in the same way other taggers are; there is no real 'correct' since we are not basing our definitions on English. To validate, we will need to see whether our approach finds a consistent tag for a given identifier within the identifier's original system and across systems. We talk about the former in this paper but the latter is future work. If it is the case that a word is marked consistently within its given system as well as across systems, then our heuristics are more likely revealing something latent about how language is used in software and can be called correct.

Further, once validated, the data obtained using this approach can be used to apply techniques from the natural language processing domain in order to fine-tune what part of speech is assigned to an identifier. The advantage, given our heuristics, is that these models are based on how identifiers are used in code. It is our belief that more latent knowledge about how identifiers are used in code will be unveiled. We feel tagging will even be consistent for a given identifier between different systems because it is being used in a consistent manner that our heuristics uncover.

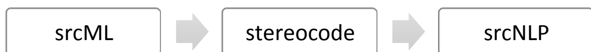
IV. IMPLEMENTATION IN SRCML

Our heuristics have been implemented in a tool called srcNLP. srcNLP uses srcML and a libxml2 SAX parser in order to compute data about identifiers. Since srcML wraps identifiers with abstract syntax, we are able to determine the type of any identifier (as long as the type is statically computable). Furthermore, we can determine if identifiers or functions

are `const`, if identifiers are aliases, and so on. Essentially, we are able to keep track of a large amount of metadata for any given identifier (whether a variable, object, or function name) in any given system. srcNLP uses this information and data gathered from the tool *stereocode*, which implements the method stereotype assignment, to determine the constraints of our heuristics. Once determined, the PoS is inserted directly into the srcML of the source code in the form of srcML tags with an `nlp` namespace. That is, if an identifier is a noun, then it is marked with an `<nlp:noun>` tag in srcML. The current implementation of srcNLP is very fast, able to completely mark up a 2.5 million lines in under four minutes. This does not include the time required for marking up comments since that part has not yet been integrated into the tool. An example of the markup is below.

```
<decl_stmt><decl>
  <type><name>unsigned</name><name>char</name></type>
  <name><nlp:adjective>p</nlp:adjective></name>
</decl></decl_stmt>
```

The workflow for our approach is below. We apply srcML to the code base, and then use stereocode to determine the stereotypes of all the methods. This is the input to srcNLP (i.e., a srcML document with stereotype information). srcNLP then adds markup for PoS.



We currently do elementary data preparation on the names of identifiers; we make upper case into lower case and we remove any non-alphabet symbols from identifier names. We believe it is likely that splitting identifiers [24-26], abbreviation expansion for abbreviated Identifiers [27, 28], and some handling of identifier naming conventions [29] would greatly increase the accuracy of our approach since we could look at what words make up the identifier instead of looking at what is likely an agglomeration of multiple terms. Synonyms [30] are also an issue that will likely need to be addressed.

V. Preliminary Results

There are a number of ways we intend to evaluate our results. For the time being, we examine the data is to see if the heuristics are consistent within a system. So we counted up how many words fell under each separate category (adjective, pronoun, etc). We then pruned out any word that appears only once. This way the data to reflect words that were assigned to the same part of speech at least twice and gives at least a modicum of confidence about consistency of usage. If we find an identifier was given more than one part of speech within the system, we called it a mismatch. A mismatch means that, despite us giving a word the same part of speech at least twice, we still assigned it to a different part of speech somewhere else in the same system. If an identifier is not given more than one part of speech then we leave it in the bucket that corresponds to the part of speech it was characterized. In the end, we get a count of how many times each identifier was seen and its PoS.

This data is shown in Table 1. On the left, we give the part of speech label. The numbers that follow are counts for each part of speech according to system.

Table 1 Number of verified identifiers for each system according to part of speech with precentages. The mismatch row at the bottom represents the number of words in each system that were given more than one part of speech tag hence lowering the accuracy for our tool.

	Blender	Brclad	Cali	Inkscape	Ogre	Avg
Verb	1292 (12%)	1411 (12%)	9768 (45%)	1320 (27%)	1109 (40%)	29%
Adj.	4117 (39%)	4278 (36%)	3707 (17%)	1104 (22%)	448 (16%)	26%
Pronoun	2647 (25%)	2608 (22%)	2159 (10%)	998 (20%)	480 (17%)	17%
Prop. Noun	1594 (15%)	2340 (20%)	3129 (14%)	786 (16%)	508 (18%)	16%
Mis match	838 (8%)	1040 (9%)	1927 (9%)	516 (11%)	147 (5%)	9%
Noun	71 (1%)	102 (1%)	1088 (5%)	186 (4%)	79 (3%)	3%

The data shows that developer use of identifiers in each system were fairly consistent given our threshold of at least two. Mismatches only happened between 8-11% of the time with an average of 9%. The only part of speech that was very low is nouns. This makes some sense seeing as how our heuristic states that nouns occur only in class declarations since they are part of object compositions. Likewise, a large number of verbs and adjectives are reasonable; verbs represent functions with some potential side effect. These are everywhere in OOP. Likewise, adjectives represent functions that return data about objects or variables that hold data about objects (or arrays) that are also very common. In essence, the data reflects that naming is mostly consistent as well as a fairly typical breakdown of types of identifiers within a system.

VI. Future Work

Future work will involve a more full evaluation of the heuristics. The first step that needs to be taken is that we require a large number of systems categorized by domain. This will allow us to study if the use of identifiers between systems of similar application domain is consistent using our heuristics. Secondly, we will apply various filtering techniques (splitting, abbreviation expansion, etc.) in order to remove threats to the validity of our approach.

Provided our evaluation shines a positive light on our heuristics, we will do two things. The first is to create a database of words and their typical usages to be used by the research community; similar to WordNet's use in the NLP community. This is similar to work done in [31]. Using this database, we will be able to record a large number of words from varying systems and their typical usages. This will allow for the creation of models based on typical word usage with respect to source code and would be made available to the research community as a whole. One way we may be able to do this is to use the aforementioned database to find typical word usages and create a mathematical model. Another way would be to investigate more fully the relationship between stereotypes and our PoS heuristics. We will also investigate higher-order patterns like verb-direct object pairs [32]. In the end our hope is that this mark-up could help to support research in identifier naming [33], code summarization [34, 35], and bug fixing [36].

REFERENCES

[1] Boehm, B. W., *Software engineering economics*, Prentice-hall Englewood Cliffs (NJ), 1981.

[2] Erlikh, L., "Leveraging legacy system dollars for e-business", *IT professional*, vol. 2, no. 3, 2000, pp. 17-23.

[3] Shepherd, D., Pollock, L., and Vijay-Shanker, K., "Case study: supplementing program analysis with natural language analysis to improve a reverse engineering task", in Proceedings of Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, 2007, pp. 49-54.

[4] Etzkorn, L. H., Davis, C. G., and Bowen, L. L., "The language of comments in computer software: A sublanguage of English", *Journal of Pragmatics*, vol. 33, no. 11, 11// 2001, pp. 1731-1756.

[5] Etzkorn, L. H. and Davis, C. G., "A documentation-related approach to object-oriented program understanding", in Proceedings of Program Comprehension, 1994. Proceedings., IEEE Third Workshop on, 14-15 Nov 1994 1994, pp. 39-45.

[6] Vinz, B. L. and Etzkorn, L. H., "Comments as a Sublanguage: A Study of Comment Grammar and Purpose", in Proceedings of Software Engineering Research and Practice, 2008, pp. 17-23.

[7] Pollock, L., Vijay-Shanker, K., Shepherd, D., Hill, E., Fry, Z. P., and Maloor, K., "Introducing natural language program analysis", in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. San Diego, California, USA: ACM, 2007, pp. 15-16.

[8] Shepherd, D., Fry, Z. P., Hill, E., Pollock, L., and Vijay-Shanker, K., "Using natural language program analysis to locate and understand action-oriented concerns", in *Proceedings of the 6th international conference on Aspect-oriented software development*. Vancouver, British Columbia, Canada: ACM, 2007, pp. 212-224.

[9] Gupta, S., Malik, S., Pollock, L., and Vijay-Shanker, K., "Part-of-speech tagging of program identifiers for improved text-based software engineering tools", in Proceedings of Program Comprehension (ICPC), 2013 IEEE 21st International Conference on, 20-21 May 2013 2013, pp. 3-12.

[10] Brill, E., "A simple rule-based part of speech tagger", in *Proceedings of the third conference on Applied natural language processing*. Trento, Italy: Association for Computational Linguistics, 1992, pp. 152-155.

[11] Brants, T., "TnT: a statistical part-of-speech tagger", in *Proceedings of the sixth conference on Applied natural language processing*. Seattle, Washington: Association for Computational Linguistics, 2000, pp. 224-231.

[12] Toutanova, K., Klein, D., Manning, C. D., and Singer, Y., "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*. Edmonton, Canada: Association for Computational Linguistics, 2003, pp. 173-180.

[13] Toutanova, K. and Manning, C. D., "Enriching the knowledge sources used in a maximum entropy part-of-speech tagger", in *Proceedings of the 2000 Joint SIGDAT conference on Empirical methods in natural language processing and very large corpora: held in conjunction with the 38th Annual Meeting of the Association for Computational Linguistics - Volume 13*. Hong Kong: Association for Computational Linguistics, 2000, pp. 63-70.

[14] Schmid, H., "Treetagger| a language independent part-of-speech tagger", *Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart*, vol. 43, 1995, pp. 28.

[15] Ratnaparkhi, A., "A maximum entropy model for part-of-speech tagging", in Proceedings of Proceedings of the conference on empirical methods in natural language processing, 1996, pp. 133-142.

[16] Giménez, J. and Marquez, L., "Fast and accurate part-of-speech tagging: The SVM approach revisited", *Recent Advances in Natural Language Processing III* 2004, pp. 153-162.

[17] Dickinson, M. and Meurers, W. D., "Detecting errors in part-of-speech annotation", in *Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics - Volume 1*. Budapest, Hungary: Association for Computational Linguistics, 2003, pp. 107-114.

[18] Collard, M. L., Decker, M. J., and Maletic, J. I., "srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration", in *Proceedings of the 2013 IEEE International Conference on Software Maintenance*: IEEE Computer Society, 2013, pp. 516-519.

[19] Alhindawi, N., Dragan, N., Collard, M. L., and Maletic, J. I., "Improving Feature Location by Enhancing Source Code with Stereotypes", in Proceedings of Software Maintenance (ICSM), 2013 29th IEEE International Conference on, 22-28 Sept. 2013 2013, pp. 300-309.

[20] Dragan, N., Collard, M. L., and Maletic, J., "Reverse engineering method stereotypes", in Proceedings of Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on, 2006, pp. 24-34.

[21] Dragan, N., Collard, M. L., and Maletic, J., "Automatic identification of class stereotypes", in Proceedings of Software Maintenance (ICSM), 2010 IEEE International Conference on, 2010, pp. 1-10.

[22] Loper, E. and Bird, S., "NLTK: The natural language toolkit", in Proceedings of Proceedings of the ACL-02 Workshop on Effective tools and methodologies for teaching natural language processing and computational linguistics-Volume 1, 2002, pp. 63-70.

[23] Bird, S., "NLTK: the natural language toolkit", in Proceedings of Proceedings of the COLING/ACL on Interactive presentation sessions, 2006, pp. 69-72.

[24] Binkley, D., Lawrie, D., Pollock, L., Hill, E., and Vijay-Shanker, K., "A dataset for evaluating identifier splitters", in Proceedings of Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on, 18-19 May 2013 2013, pp. 401-404.

[25] Hill, E., Binkley, D., Lawrie, D., Pollock, L., and Vijay-Shanker, K., "An empirical study of identifier splitting techniques", *Empirical Softw. Engg.*, vol. 19, no. 6, 2014, pp. 1754-1780.

[26] Enslen, E., Hill, E., Pollock, L., and Vijay-Shanker, K., "Mining source code to automatically split identifiers for software analysis", in *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*: IEEE Computer Society, 2009, pp. 71-80.

[27] Hill, E., Fry, Z. P., Boyd, H., Sridhara, G., Novikova, Y., Pollock, L., and Vijay-Shanker, K., "AMAP: automatically mining abbreviation expansions in programs to enhance software maintenance tools", in *Proceedings of the 2008 international working conference on Mining software repositories*. Leipzig, Germany: ACM, 2008, pp. 79-88.

[28] Lawrie, D., Feild, H., and Binkley, D., "Extracting Meaning from Abbreviated Identifiers", in Proceedings of Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on, Sept. 30 2007-Oct. 1 2007 2007, pp. 213-222.

[29] Allamanis, M., Barr, E. T., Bird, C., and Sutton, C., "Learning natural coding conventions", in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Hong Kong, China: ACM, 2014, pp. 281-293.

[30] Haiduc, S. and Marcus, A., "On the Use of Domain Terms in Source Code", in Proceedings of Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on, 10-13 June 2008 2008, pp. 113-122.

[31] Falleri, J. R., Huchard, M., Lafourcade, M., Nebut, C., Prince, V., and Dao, M., "Automatic Extraction of a WordNet-Like Identifier Network from Software", in Proceedings of Program Comprehension (ICPC), 2010 IEEE 18th International Conference on, June 30 2010-July 2 2010 2010, pp. 4-13.

[32] Fry, Z. P., Shepherd, D., Hill, E., Pollock, L., and Vijay-Shanker, K., "Analysing source code: looking for useful verb-direct object pairs in all the right places", *Software, IET*, vol. 2, no. 1, 2008, pp. 27-36.

[33] Binkley, D., Hearn, M., and Lawrie, D., "Improving identifier informativeness using part of speech information", in *Proceedings of the 8th Working Conference on Mining Software Repositories*. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 203-206.

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

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

[36] Yuan, T. and Lo, D., "A comparative study on the effectiveness of part-of-speech tagging techniques on bug reports", in Proceedings of Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on, 2-6 March 2015 2015, pp. 570-574.