Lexical Categories for Source Code Identifiers

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

Reem S. AlSuhaibani
Computer Science
Kent State University
Kent, OH 44240 USA
ralsuhai@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*—A set of lexical categories, analogous to part-of-speech categories for English prose, is defined for source-code identifiers. The lexical category for an identifier is determined from its declaration in the source code, syntactic meaning in the programming language, and static program analysis. Current techniques for assigning lexical categories to identifiers use natural-language part-of-speech taggers. However, these NLP approaches assign lexical tags based on how terms are used in English prose. The approach taken here differs in that it uses only source code to determine the lexical category. The approach assigns a lexical category to each identifier and stores this information along with each declaration. srcML is used as the infrastructure to implement the approach and so the lexical information is stored directly in the srcML markup as an additional XML element for each identifier. These lexical-category annotations can then be later used by tools that automatically generate such things as code summarization or documentation. The approach is applied to 50 open source projects and the soundness of the defined lexical categories evaluated. The evaluation shows that at every level of minimum support tested, categorization is consistent at least 79% of the time with an overall consistency (across all supports) of at least 88%. The categories reveal a correlation between how an identifier is named and how it is declared. This provides a syntax-oriented view (as opposed to English part-of-speech view) of developer intent of identifiers.**

***Keywords*—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 in the exploration and comprehension of today's large and complex software. Deissenboeck [3] found that about 70% of the characters within the software systems studied are part of an identifier; hence the saying that about 70% of a software system is its identifiers. With identifiers making up such a large portion of complex systems, it makes sense to theorize that study of these identifiers is of great importance. There has been a large amount of effort put into studying identifiers. For instance, it has been demonstrated that natural-language clues in program identifiers can be used to improve software tools [4]. There is a sizeable body of research that applies natural language processing (NLP) techniques to source code to support various program-comprehension tasks. At the core of many of these is *Part of Speech* (PoS) tagging.

Part of speech tagging is a method of categorizing words with similar grammatical properties. It has been used in a number of software maintenance tasks including identifier naming [5], code summarization [6, 7], reformulation of source-code queries [8], concept location [9], traceability-link recovery [10], and bug fixing [11]. In all of these, the underlying need for PoS is to provide information about what the word(s) that make up the identifier mean in English, so that this meaning can be correlated with the identifier's use in code. The assumption made is that the words that make up the identifier contain information that is relevant to the identifier's meaning within its source-code context. PoS tagging is beneficial to these approaches because it is able to compute a portion of the meaning of a word and this meaning partially correlates to how the word is used in code as part of an identifier. Hence, PoS tagging provides useful information to these techniques. However, source code is not written in English sentences [12]. This means that PoS tagging, based solely on natural language, is limited in its ability to ascertain the meaning of an identifier. That is, the meaning of a word in English does not necessarily translate fully to its meaning in code and may say little about how it is used according to the rules of a programming language.

Here, we present a set of lexical categories for identifiers that are not based on English meaning and usage, but instead on usage in code; specifically, how an identifier is declared.

The assumption made here is that the syntax and placement of an identifier declaration provides a basis for determining meaning. This is in contrast to the assumption made by techniques that use English PoS on source code. After all, using knowledge about the declaration of an identifier, we know how it interacts with other identifiers, how it is used in memory, its visibility, its scope, etc.

As an example, functions that carry out actions (i.e., modify state of the code) are in the verb category. While this is rather simplified compared to the definitions we provide later, it gives insight into the categories. We use relevant, statically computable, information to provide a category that summarizes a set of properties we glean from the identifier's declaration. This is much like how natural language PoS taggers use the context of the word in a sentence; we use the context of the identifier in the program.

How identifiers are assigned categories using properties of their declarations is discussed and it is shown that identifier names imply the category to which they are assigned. The result is a basis for developer's intent with respect to identifier names. Broadly speaking, developer's intent is asking why a developer used a particular algorithm, identifier name, class, etc. Here, we show that repeated declarations using the same identifier name consistently falls within a singular category. Hence, when a developer uses a particular identifier name, this name implies a set of properties on the identifier's declaration.

The results of this work have a number of implications to research concerning naming consistency, recommendations, and classification [13-15]. Categorization of identifiers with respect to their declaration applies readily to discovering when identifier names are being used improperly. For example, if an identifier historically falls within one category in most instances, but it is then used in another category, this may signify a problem. Additionally, it is possible to use the categorizations to recommend names based on declaration and context. Techniques that take advantage of PoS tagging can also benefit since, while PoS tagging is able to compute a meaning for an identifier with respect to English, it does not use the identifier's syntactic context. This means that English PoS and the categories defined in this paper are computing orthogonal information about identifiers that can be used in conjunction.

The contributions of this work are as follows. First a set of new lexical categories for function and variable identifier names are defined based on the programming language syntax. Second, the results of an evaluation show that specific identifier names are correlated with lexical categories across a system. Hence, we show that there is a large amount of consistency between the naming of identifiers and properties of their declaration in source code.

The paper is organized as follows. In the next section we briefly discuss the related work. In Section III we discuss our definitions of lexical categories. In Section IV we compare part of speech to our lexical categories, Section V gives implementation details and the final two sections are the Evaluation, and Conclusions/Future work.

II. Related Work

A number of approaches [13-18] leverage static analysis to similar ends; they seek to use identifiers along with the context of the code surrounding the identifiers to answer some research question(s). Hence, static analysis combined with NLP and/or identifier naming/recommendation systems are both ideas that have been explored in previous research. However, none of these approaches categorize identifiers based purely on the results of fact extraction and static analysis. The idea of relating parts of speech to identifier type and function behavior has been discussed in previous work, as well [14, 19].

In terms of NLP, there have been numerous efforts to improve tagging of part of speech [20-26]. Recently, tagging of identifiers has become very accurate with approaches that can find the correct part of speech tag with an accuracy of 80 to 95% [18, 27]. Part of speech tagging is an active area of research with respect to both assigning a tag and uses of the tags to solve problems. Our approach is different than these in that it completely ignores the English meaning of the words for those identifiers and instead assigns a part-of-speech-like category (lexical category) based on combination of properties discussed in Section III.

The work presented here is a continuation of previous work done by the authors [28]. In this previous work, we present the same categories and some preliminary results on a much smaller set of systems. In this work, we expand our evaluation significantly, tighten the rules for our categories, and give a fuller discussion of their meaning.

Where natural language PoS tagging is related to the theoretical portion of this work, identifier naming and recommendation systems are related to the application portion. Deissenboeck [3] studied how identifiers are named, their consistency, and what makes an identifier name good or bad. He found that words that are synonyms and homonyms are part of what makes it difficult for developers to understand different parts of code due to the fact that there are so many ways of expressing the same meaning with different words in English. His findings motivate the need to investigate whether identifiers that are used outside of their typical context (occurs frequently in one category, then declared outside of that category) make comprehension more difficult for developers.

Comments are the focus of a lot of research in the area of comprehension. However, since comments are typically written in a natural language (English) and often have sentence structure that follows grammatical rules [29-31], we do not process them.

III. Lexical Categories for Identifiers

As mentioned previously, the lexical categories we define are analogous to English language part of speech (PoS). However, we derive the identifier lexical categories based solely on information statically computable from abstract syntax trees rather than information concerning an identifier's meaning or part of speech in a natural language.

Our intent is to determine how an identifier is used in the code and assign a lexical category to it in a consistent manner across projects. At a naive level, if an identifier is used as a method name and represents some action within the system, it is categorized as a verb. Likewise, if an identifier represents some object within the system it is categorized as a noun.

Before we can define rules for the lexical categories, information about identifiers must be defined and collected. We now discuss the informational properties of identifiers that can be statically computed, as these are the basis of the categories we define.

A. Informational Properties of Identifiers

Identifiers have a number of properties that can be statically computed. These properties indicate how an identifier behaves in memory/its properties in memory (memory it refers to or its own memory), access constraints, its type, whether it is complex or primitive, etc. Examining all these properties together grants the ability to make certain assumptions about the nature of the intent and meaning behind the identifier (within the context of the program). The five properties (see examples in Fig. 1) we identify in this work are:

1. *Type*. This includes the type's name, methods on the type, and member data for the type (when applicable).
2. *Behavior* with respect to memory. This includes whether it is an alias, if it can modify memory, if it is statically allocated, etc.
3. Where it is *declared* (e.g., in a function, class, etc.).
4. Where it is *used* in within the program and what other identifiers it interacts with. This is determined via definition-use chain and/or slicing.
5. The *meaning* of the word(s) it is made up of (i.e., natural language part of speech and definition(s)).

Our categories currently only use properties 1, 2, and 3. These three properties are important because they reveal how the identifier can be used in a syntactic statement/expression. Further, these properties are quantifiable in that they are facts with respect to the source code. Hence, we can base assumptions off them as well as use them to create constraints. Incorporating properties 4 and 5 are left to future work.

```
/*Properties of Object:
 *(1) Type: Object, user defined
 *(2) constraints: const, pointer
 *(3) context: function scope
 *(4) Def-use: Defined on line 1, used on line 2
 *(5) English PoS: "Object" is a noun
 */
void foo(){
    const Object* obj = new Object();
    obj.GetData();
}
```

Fig. 1. An example of the properties of an identifier.

We now discuss the first three properties in more detail. Property 1 of an identifier is its type. A type tells us what data an identifier represents as well as methods associated with that data. Most importantly for this work, it tells us whether the identifier is complex or primitive. This allows us to assume that an identifier which represents complex data is not being used purely as a descriptor; it is not just the length of an array or the circumference of a circle. Note that we do not currently differentiate if a type is polymorphic. Hence, being able to compute its base type is enough.

Property 2 of an identifier is how it behaves with respect to memory. This relates to its const-ness, if it is a pointer, reference, etc. It indicates whether an identifier is able to modify the state of the program by changing memory. For example, it can have the ability to modify memory segments it references (if it is a pointer) as well as data held within those memory segments. Understanding how an identifier behaves with respect to memory allows us to assume or constrain that an identifier may not always point to the same location in memory (it is declared as non-const pointer).

Property 3 tells us about the lifetime, visibility/scope of the data represented by the identifier. In languages that support classes and structs, an identifier can be declared in a class/struct, function, namespace, or globally. If it is declared in a class/struct, then the identifier's lifetime is attached to the lifetime of another object and its visibility is the same as the object it is part of. Using this information, we can assume or constrain that an identifier's lifetime is not attached to any other object or that it is visible even in lieu of an object (i.e., a free function).

Each of these three properties is abstract in that they cover a wide range of possible values. Using these in combination allows us to express a wide variety of statically computable facts for identifiers and, hence, gives us the ability to create very precise definitions for the lexical categories for identifiers. The purpose of the category is to determine to what set of properties an identifier correlates. The assumption is that the identifier and set of properties are an implied contract; the developer uses this identifier to imply a certain combination of properties 1, 2, and 3. The correctness of this assumption is explored by examining whether identifiers have the same category in multiple instances. That is, we check to see if the developer consistently uses the same type of definition (with respect to the properties) for a given identifier repeatedly while rarely using the identifier with a different type of definition.

Again, we omit properties 4 and 5 and investigate in future work. To collect property data for identifiers, we use srcML [32]. 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 [33]. Stereotypes categorize methods and functions based on their definitions (we discuss stereotypes in more detail later in the paper). Primarily, our lexical category definitions use stereotypes to determine how a function behaves with respect to its arguments, local variables, and the calling object (if applicable).

B. Definitions for Identifier Lexical Categories

Identifiers in programs are used for a number of constructs: namespace names, class/struct names, function names, and variable names. In this work, we are not concerned with identifiers for namespaces or classes/structs. Identifiers are assigned an identifier category only when they are declared, since at that time the type, specifiers, and name are known. One exception is function identifiers, where the identifier category is assigned when the function is defined, since the assignment is based on the function's stereotype, which is based on the function definition, not just the declaration.

We discuss these categories as a definition and a combination of the properties discussed earlier that an identifier must be constrained to in order to fall within the given category. In some of our definitions, we use the concept of uniqueness. An identifier is *unique* if it is not part of an object composition (i.e., not a class attribute). A non-unique identifier is part of a composition. If an identifier is part of composition, then it is referenced using dot (or arrow) notation in C++ (i.e., `a.b` or `a->b`). The identifier `b` is not unique unless it is attached to a unique identifier `a`. The identifier `a` is unique within its scope. Namespaces are not an issue here, as namespaces do not have types and are not considered by our categories. As we do not compute how objects are destroyed in our analysis, we do not differentiate between aggregation and composition. Composition is used to refer to either situation. Additionally, the term *entity* is used to describe the segment of memory to which an identifier refers.

Our definitions borrow terms from English part of speech and to distinguish them we prepend an **s-** (for source code) to the front of the term. For example, instead of noun, we write s-noun. We have chosen this over creating our own terms to keep the definitions in line with their intention of creating lexical categories akin to part of speech but for source code. As stated above, we aim to describe concisely what an identifier is and how it is used with respect to its context. The definitions are given as a description and a set of constraints. Next to those constraints is the corresponding property number from the previous subsection to clarify which property each constraint corresponds.

Definition proper s-noun: Proper s-nouns are identifiers for unique, non-primitive entities. Proper s-nouns must satisfy the following constraints:

- The identifier must refer to (1):
  a. A first class, user defined object
  b. An array of user defined objects or primitives
- If the identifier is a pointer, then it must be unable to be changed to point at another entity (e.g., a const pointer or reference variable) (2)
- Must be unique (i.e., cannot be a field) (3)

In object oriented programming, proper s-noun identifiers represent one of the primary ways that 1) Data is grouped and moved throughout the system and 2) The state of the system is modified. This categorization is important because of these two facts; identifiers within this categorization embody agglomerations of data developers considered important enough to name and are a big part of how the system's state will change. In Fig. 2, `result` and `n` are proper s-nouns since they have a complex type and are not fields. They are close in definition to s-nouns, which we define next.

Definition s-noun: s-nouns are identifiers for non-unique, non-primitive entities. An s-noun must satisfy the following constraints:

- The identifier must refer to (1):
  a. A first class, user defined object
  b. An array of user defined objects or primitives
- If the identifier is a pointer, then it must be unable to be changed to point at another entity (e.g., a const pointer or reference variable) (2)
- Must be non-unique (i.e., declared in a class) (3)

Identifiers that are s-nouns represent object composition. These identifiers are grouped with other s-nouns in order to embody a larger concept as part of a proper s-noun. Identifiers in this category describe parts of proper s-nouns; identifiers consistently used as s-nouns may hold some meaning (in English) that makes them good candidates for this purpose. Notice that the primary difference between proper s-nouns and s-nouns is that proper s-nouns are unique while s-nouns are not. This difference mirrors their difference in English, where proper nouns tend to name unique entities (people, places, etc) as opposed to nouns, which refer to a class of entities (i.e., city refers to any entity that fits the definition, not any particular city). Further, there is a sense of ownership between proper s-nouns and s-nouns in that s-nouns are only referred to through a proper s-noun. We do not check to see if a given s-noun's entity is garbage collected by the proper s-noun it is part of, so our definition of ownership is a less constrained than in software engineering literature. In Fig. 2, `name` and `yearborn` are both s-nouns due to being fields with complex types.

Definition s-adjectives: s-adjectives are identifiers for unique or non-unique, primitive entities. They must satisify the following constraints:

- Identifier's type is primitive (e.g., int, float, bool) and cannot represent a sequence container, (e.g., array, vector) or any sequence of primitives in memory. (1)
- If the identifier represents a function (note that the type of a function is its return type), then it further satisfies the constraint that it does not apply any modifications to the following (2):
  a. Its aliased, non-primitive parameters (i.e., does not modify references or pointers to non-primitive parameters) or an array of any type.
  b. The calling object (this)
  c. Non-primitive, local variable whose value is then returned (or impacts what is returned). Local variable may not be an array of any type.
- Identifier is not a pointer or reference (2).
- Can be unique or non-unique (3)

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

Fig. 2. An example of applying the lexical categorization.

Identifiers that are s-adjectives represent metadata about some part of the system. This could be the size of a sequence container, the color of a texture, the result of a predicate, etc. Their primary purpose is to convey some data about the data computed by other parts of the system. Hence, s-adjective must have a primitive type, and functions that are s-adjectives must not modify complex entities or arrays. In Fig. 2, `age` and `returnAge` are adjectives; `age` due to its having a primitive type and `returnAge` due to being a function that does not modify state and has a primitive type.

Definition s-pronouns: s-pronouns are identifiers for unique or non-unique, primitive or complex entities. Identifiers that satisfy the following constraints are s-pronouns:

- Must be user-defined or primitive (1)

- Must be pointer or reference to a s-noun or s-proper noun (2)
- Must be able to point at any valid entity (cannot be const) (2)
- Can be unique or non-unique (3)

Identifiers that are s-pronouns embody the ability for an identifier to reference different entities. These identifiers are pointers or reference variables that are distinguished from their s-noun and proper s-noun cousins because they do not need to always refer to the same entity. This distinction is important because changes in context affect them more heavily; they may point to one entity up to a specific line of code and then point to a separate entity afterwards. Identifiers consistently used as s-pronouns emphasize this discrimination to developers better than identifiers used in other categories. They are identical to s-nouns and proper s-nouns in every other way. In Fig. 2, `str` within the argument to the split function is a s-pronoun due to being non-const and a reference variable.

Definition s-verbs: s-verbs are identifiers for unique or non-unique, primitive or complex entities. Identifiers that satisfy the following constraints are s-verbs:

- Type is primitive or complex (1)
- Can be unique or non-unique (method or free function) (3)
- It is the name of a function that applies some modification to at least one of three things (2):
  a. One or more of its non-primitive parameters whose value impacts the returned value or is an alias. If the modified parameter is an array, it may be primitive or non-primitive.
  b. The calling object (`this`).
  c. Non-primitive, local variable whose value is then returned (or impacts what is returned). If the local variable is an array, it may be primitive or non-primitive.
- Its return type, calling object or (aliased) arguments must be modifiable (i.e., it has to perform some useful modification to program state) (2).

Identifiers that are s-verbs are actions that modify the state of the system. Where proper s-nouns and s-pronouns collect and move data, s-verbs act on that data to produce new data. In Fig. 2, `Split and SetName` are s-verbs due to being identifiers for functions that modify the system state (`SetName` modifies `this` and `Split` returns a value with a complex type).

C. Categorizing Method/Function Identifiers with Stereotypes

We now discuss our use of method stereotypes to categorize method identifiers. Stereotypes are always only applied to identifiers for methods, never variables. 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 [34, 35] for a complete discussion of stereotypes. 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 s-verbs. *Creational* methods create or destroy objects of the class. These correspond primarily to s-verbs; they completely construct an object thereby changing the program's state. *Collaborational* methods characterize the communication between objects and how objects are controlled in the system. We primarily mark these as s-verbs, but future work will investigate if there are more complex patterns. Particularly, how s-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 the definitions in this case.

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

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

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

Fig. 3. Example of markup in srcML

Since a method may have more than one stereotype, we discuss how to combine stereotypes to obtain a categorization. The primary purpose of the stereotype is to more accurately differentiate between methods that modify state and methods that do not. The technique benefits from stereotypes since functions and methods require more thorough examination in order to conclude whether or not they modify state. In essence, we differentiate between Structural Accessors and everything else. Anything combined with a Structural Accessor ends up as an s-adjective or a s-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 s-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 categorization and use data about the return type and arguments. This means that if at any point the stereotypes declare a method to be degenerate, we ignore the stereotype and use standard analysis methods to categorize the identifier. If that fails, the identifier will not be categorized.

IV. Implementation in srcML

Our approach for categorizing is implemented in a prototype tool called srcNLP to evaluate the categories. 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 can 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 from the tool *stereocode*, which implements the method stereotype assignment, to determine the constraints of our categories. Once determined, the lexical category is inserted directly into the srcML of the source code in the form of srcML tags with an `nlp` namespace. So if an identifier is a s-noun, it is marked with an `<nlp:s-noun>` tag in srcML.

TABLE II. Counts of identifier categories with no MinSup

Number of Systems	50	
Unique Identifiers	480K	**Percentage**
s-adjective	185K	39%
s-noun	105K	22%
s-verb	78K	16%
s-pronoun	43K	9%
proper s-noun	42K	9%
multiple	27K	6%

TABLE III. Distribution of System Sizes

Size of System (KLOC)	**Number of Systems**
80-600	35
601-1200	10
1201-1800	3
1801-2400	2

The current implementation of srcNLP is very fast, able to completely markup 2.5 million LOC in under four minutes. A small example of the markup is given in Fig. 3.

The implementation is created to serve as a way to evaluate the lexical categories. The evaluation focuses primarily on looking at how consistently identifiers are categorized within each separate system and we only lightly consider the question of how consistent markup is when a word is found within multiple independent systems. Hence, for our implementation, we do very little data preparation on identifier names. The identifier names are made more consistent by converting to lowercase, and removing any non-alphabet symbols from identifier names. This is done because we present information about identifier use between multiple systems. Different systems have different naming conventions, so matching terms requires manipulation of identifier strings. There are a number of well-studied techniques for normalizing identifier names [36-42].

To evaluate the categories, we study a number of open source systems in order to determine whether there is some correlation between the name of an identifier and the category it is placed in. Recall that we define developer's intent with respect to identifier declarations as the implicit relationship between the identifier's name and the category it is in. Identifiers that are consistently placed in the same category have a meaning that correlates to the set of properties for that category. Our hypothesis is that, given multiple declarations of an identifier within a body of code, this identifier will, more often than not, end up in the same category after each unique declaration. If category assignment is consistent for a large number of identifiers, then the category and the identifier have a strong correlation with one another—use of the identifier implies properties characterized by the category it is assigned.

TABLE IV. 50 Systems used in evaluation (KLOC).

Anjuta (237)	Shogun (272)	Blender (1500)
Boost (2400)	Audacity (509)	Clang (650)
Cntk (100)	Calligra (867)	Cppcheck (140)
Easydarwin (615)	Codeblocks (685)	Grpc (181)
Hippodraw (131)	Gameplay (218)	Kdevelop (156)
Llvm (977)	Inkscape (424)	Mozjs (800)
Nedit (95)	Mongoose (95)	Ogre (417)
Opencv (678)	Notepad (174)	Prgrmrs notepad (200)
srcML (297)	Poco (600)	Swift (358)
Apache (519)	Bitcoin (94)	Sxemacs (247)
Bullet3 (467)	Ccv (139)	Synfig (246)
Cocos2d (724)	Codelite (639)	Tensorflow (123)
Folly (119)	Gimp (700)	Tortisesvn (1300)
Httpd (195)	Irrlicht (274)	Quantlib (486)
Mongo (1900)	Monkeystudio (130)	Rocks (188)
Newton (1300)	Octaforge(87)	Scintilla (86)
Openfrmwrks (600)	Ppsspp (400)	

V. Evaluation

We evaluate two ways. First, we examine if identifiers are assigned consistently to the same category when they are drawn from singular systems. Secondly, we examine if category assignment is consistent across systems. While we feel the latter is as reliable-- a developer on one system does not care if his identifiers are being used consistently in another system -- the results are still interesting and the problem is relevant to future work.

In the evaluation, we use various minimum supports (minsup). Minimum support is a threshold that an identifier must exceed in order to be considered frequent. For example, if our minimum support is 2 and we are looking at identifiers that are s-verbs, then only identifiers that are declared at least 2 times and categorized as s-verbs within a given system are considered frequent. All identifiers that occurred less than 2 times are removed from consideration in the results here.

We break the evaluation up into 3 research questions (RQ):

1. How many identifiers are categorized into each identifier category on an intra-system basis?
2. How many identifiers are classified into more than one identifier category on an intra-system basis? At what frequency are identifiers typically given more than one category?
3. What is the distribution of identifiers common between multiple systems?

In order to evaluate this, we examined fifty (see Table IV) open source C and C++ systems across several domains including rendering, image processing, physics engines, compilers, etc. We selected systems based on a few criteria: The system needs to be larger than 80KLOC; we wanted systems with a nice, representative sample of identifiers so that the sample would have a higher chance of reflecting use in other large software systems but we did not want to set the bar so high that it would be difficult to find enough open source systems. In essence, we needed a varied sample both over a multitude of systems and a multitude of usages within those systems. The system needs to be written in C or C++ because the evaluation tool only works on these languages. We also only considered a minimum support of up to 10 because many identifiers are infrequent once we reach that support; going beyond it does not yield significantly different results. Basic data about the systems is provided in Table II and Table III. Table IV presents a list of systems.

In figures 4 and 6, identifiers are presented in the lexical categories. These identifiers always have the same lexical category every time they are declared. Figure 5 presents identifiers that are in multiple categories. An identifier can be placed in the multiple categories if it receives more than one of the other categories defined. For example, take an identifier used in two declarations, in the first declaration it is placed in the s-noun category and in the second declaration it is placed in the s-verb category. Finally, in all cases, only identifier declarations are counted, not usages.

RQ1 How many identifiers are in each category?

To answer RQ1, we determine the distribution of the data across all of our identifier categories. Figure 4 presents the distribution of words that occurred on an intra-system basis. That is, we compute the frequency of occurrence for identifiers within each system separately while categorizing them in one of our categories. For the x-axis, increased minimum support means that an identifier needed to be declared a larger number of times in order to be considered frequent. The y-axis is the total number of words among all systems that are categorized as anything except multiple. Hence, as minimum support is increased, there are fewer identifiers that are declared the required number of times within a single system. In total, there are about 197k identifiers present in the dataset that makes up Figure 4. The most common categories are s-adjectives, s-verbs, and s-nouns. Predictably, as minimum support increases the distribution falls quickly from left to right. In the middle, at minsup 6, we find a total of ~8k identifiers. At the tail end, at minsup 10, there are only around 3k. The most frequently occurring words counted in Figure 4 are: `addoperand`, `loadptr`, `registerelement`, and `getsingleton`.

RQ2 How many identifiers are in more than one category?

Now we move onto RQ2. Figure 5 presents all identifiers placed in multiple categories. It also gives the rate at which those identifiers tended to change. The x-axis is a percentage that represents the identifier's stability. Stability is defined as the identifier's resistance towards changing categories.

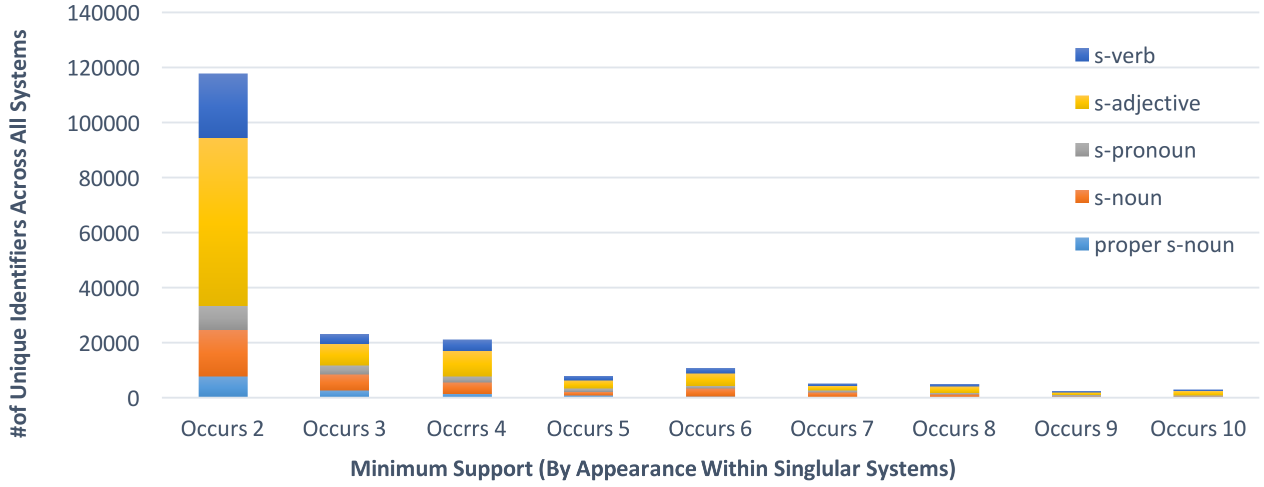

Fig. 4. Distribution of identifiers based on frequency of declaration. Increasing minimum support (x-axis) means that the identifier needs to be declared at least that many times within at least one system to be considered frequent.

Identifiers with a low stability; 10%, for example, are likely to change. Identifiers with high stability; 90%, for example, rarely change. Hence, as we move towards the right on the x-axis, identifiers become more stable. The y-axis shows the number of unique occurrences among all systems. The colors that make up segments of the bars represent minimum support level; how much an identifier has to occur within a single system in order to be considered frequent. There are about 27K identifiers that are placed in multiple categories as opposed to the 197K that are not. Since the number is so low, the trend here is somewhat light except for at lower minsups. At minsup 2, 3, and 4 most identifiers are unstable with a stability < 50%. The higher the minsup, the more stable identifiers tended to be. Even at the higher minsups, most identifiers tend to sit between 50 and 79%, however. In general, the more an identifier is used, the more confident we can be about its categorization when it is in multiple categories. This confidence is only moderately high, but easily suffices due to the comparatively low number of identifiers in multiple categories. Some of the most unstable identifiers present in multiple categories are: `oldbuffer`, `erroraction`, `waittime`, and `tokenpos`. Some of the most stable are: `getvalue`, `tok`, `swap`, and `pop`.

We now examine how the answers to RQ1 and RQ2 relate to one another. In the answer to RQ1 we see the distribution of identifiers categorized consistently. In RQ2 we see the distribution of identifiers categorized inconsistently. The question is, given data from both, how consistent are identifiers in general? Essentially, what we want to see is that identifiers tended to be categorized consistently as opposed to being in multiple categories. Starting with a minimum support of 2, there are 4K identifiers in multiple categories and ~117K identifiers categorized consistently. It is clear that consistency at this minsup is very good. At a minsup of 10 there are 1.4K identifiers in multiple categories and about 3.1k identifiers that are categorized consistently. This means that 31% (1.4K/4.5K)

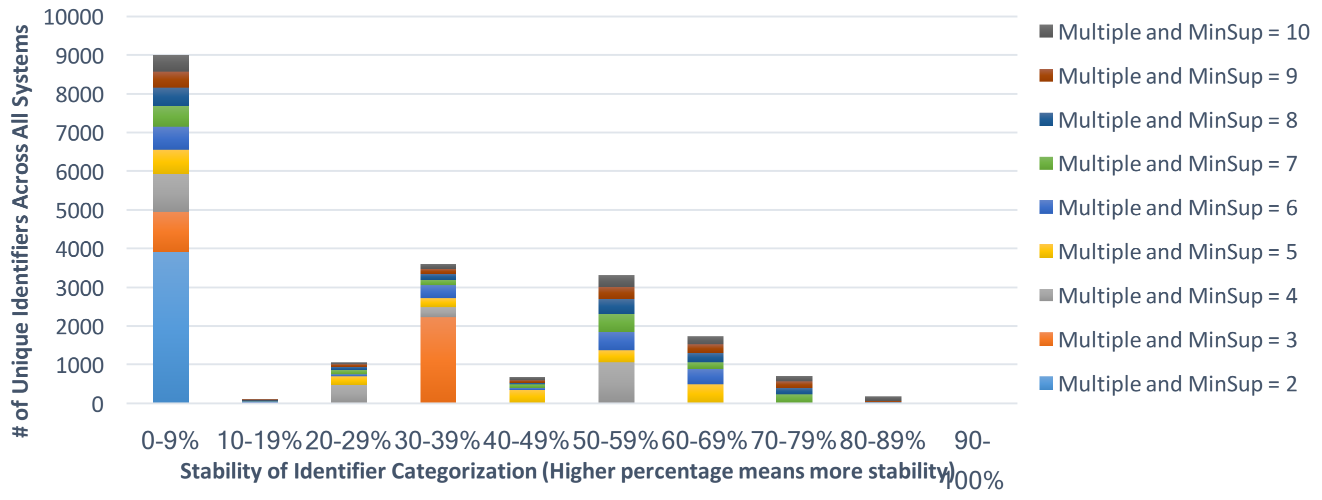

Fig. 5. Count of unique identifiers marked multiple with respect to both stability and frequency of declaration. Higher stability means less proneness to changing category despite identifier having been marked multiple.

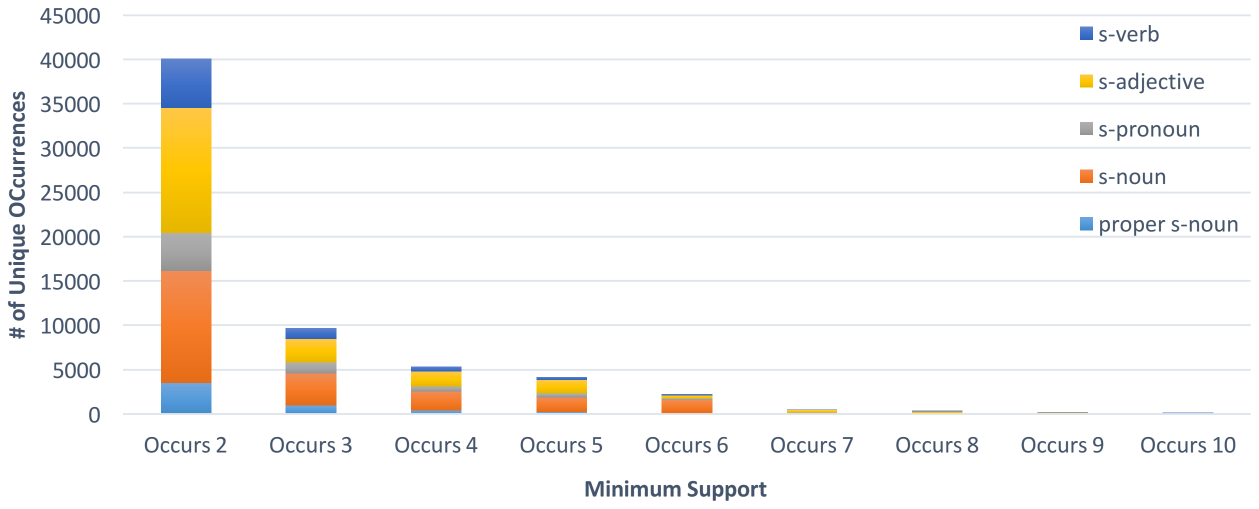

Fig. 6. Distribution of identifiers based on frequency of declaration. Increasing minimum support (x-axis) means that the identifier needs to be declared in at least that many systems to be considered frequent.

of identifiers used in at least 10 different declarations are unstable. Examining Figure 5 we see that of the 1.4K identifiers in multiple categories with a minsup of 10, around 760 are stable 50% of the time or more. Assuming the worst case, around 380 of those 760 are completely stable. This means the number is closer to 22% (1K/4.5K) unstable, therefore 78% are stable. Let us look at a minsup of 5 next; this way we look at both the edges and the middle of the distribution. At a minsup of 5, there are a total of about 8k identifiers that are consistently categorized and 2.2K that are in multiple categories. This means that about 21% (2.2K/10.2K) of identifiers are unstable and 79% are stable. Again, we can look at Figure 5 and see that about 900 of the 2.2K identifiers are stable at least 50% of the time. If we assume that half of those (450) are stable, then the ratio we get is 17% (1.75K/10.2K) unstable and 83% stable. Furthermore, the total number of identifiers that are always categorized consistently at least twice is ~197K (see Figure 4, total for all minsups 2-10) while the total number of identifiers that are in multiple categories is only 27K. The ratio between total identifiers used at least twice and placed in multiple categories is ~12% (27K/224K), meaning that 12% of the total number of identifiers occurring 2 or more times are placed in multiple categories.

RQ3 What is the identifier distribution between systems?

Finally, we answer RQ3. We want to know what the distribution is if instead of only looking for identifier matches within a singular system and counting occurrences, we look at identifiers common to multiple systems and throw out those that happen only within a single system. Figure 6 presents the result. The x-axis is the number of individual systems an identifier has to be matched within before it is considered frequent, e.g., for a minsup of 5, then we have to find an identifier in 5 separate systems in order for it to be considered frequent. The y-axis counts unique identifier declarations across all systems. This distribution falls extremely quickly, as expected. Matching identifiers across systems is much more difficult; some systems append text before or after identifier names, use special abbreviations, etc. There is also the issue of matching synonymous identifiers.

There are ~6K identifiers that occur between at least 5 or 6 systems; after that the distribution is insignificant. Some of the most common words that occurred in multiple systems are: `rotation`, `maxvalue`, `starty`, and `getoffset`.

There are 14 identifiers that occurred in all 50 systems studied: `a`, `length`, `id`, `pos`, `start`, `next`, `str`, `key`, `f`, `x`, `index`, `p`, `left`, `result`.

VI. THREATS

The threats to our approach come in the form of its ability to extend to languages other than C/C++, our use of open source software in our evaluation, and the completeness of the technique as it is presented.

A. Completeness

We do not argue that the set of categories presented are complete. This is partially due to our belief that further research is required to formally define a complete set of categories that uses the categories in this paper and takes into account more statically computable data (e.g., definition-use chains, other language keywords) in order to examine larger patterns. Hence, it is possible that there are more categories than presented. In the current state, however, we take into account many very common types of identifiers and declarations; handling pointers and references, function and variable names, member and non-member variables, methods and free functions, etc. We believe this provides good coverage and that further study may leverage this work to discover more categories or refine the provided categories to include or exclude certain types of identifiers.

B. Open Source Systems as a Sample

We only used open source systems in our sample. It is possible that the data we found is specific to open source software and does not reflect in commercial software. Unfortunately, finding a large data set of commercial software to examine is a much more difficult task. However, even if evaluation on commercial software shows less consistency between identifiers and declarations, it would only motivate the need for further study of the categorizations and commercial software to compare and contrast impact in open source versus impact in commercial.

C. Extensibility to Other Languages

The tool written for the evaluation only works on C/C++ systems. A natural question is whether these lexical categories can be applied to other languages as they are presented in this text. Constraints are purposefully worded to avoid using language keywords. Instead, keywords are relegated to examples. This is because we wanted to emphasize behavior (i.e., saying something "must always refer to x" instead of saying it must be const). Additionally, many languages differ in how they handle things like object declarations. For example, in C# when you declare an object using new, you get a pointer to that object in memory. You cannot change what object the given identifier points to despite it being a pointer. To do this in C++, you need to use the const keyword on the pointer or else it can be changed to point elsewhere. Hence, C#'s default semantics for object declaration using new is more similar to C++'s object declaration using both new and const. This shows how languages can be syntactically similar but semantically different. However, this does not change the category in either case. Despite not having to explicitly use const in C#, the category for this object would be the same as a const pointer to an object in C++. Hence, the ability for the categories to handle differing language syntax primarily hinges on the fact extractor knowing how to handle language differences at categorization time.

Our evaluation is done on C/C++ systems and so the results of the evaluation do not necessarily extend to other languages. We acknowledge this weakness; it will require further study of our categories on systems written in other languages in order to come to any conclusion about the behavior of identifiers in other languages. However, what we present here is certainly a proof of concept as well as proof that these patterns exist at least within C/C++ systems. We feel it will be valuable to extend and study the categories on systems in other languages.

VII. Discussion & Future Work

The answer to RQ1 and RQ2 strongly hints that developers use names purposefully; the name of an identifier more often than not also implies a set of properties on that identifier. That is, the correlation between identifiers and their categorization is consistent. This is apparent in the fact that from the left side of the distribution to the right, the number of consistently categorized identifiers remains high, with 78% being the lowest rate of consistency at a minimum support of 10. Additionally, as stated, the overall consistency for identifiers occurring more than 2 times is 88%. It is our opinion that these findings lend evidence to the notion that the categories are a good foundation for developer's intent with respect to variable and function names. Specifically, the intent is the relationship between the identifier's name and its category; the developer's use of certain identifier names implies a particular category. We believe these categories are a strong step towards creating a more formal, theoretical basis for identifier naming that is based on identifier usage that are natural and programming language-agonistic.

For this reason, there are a few research directions we see for the future of these categories. Combining it with other techniques, such as NLP, may yield strong results for work that relies heavily on analysis of identifiers, such as program comprehension. The reason we make this statement is that NLP looks at identifiers from the perspective of their English meaning. As we argued in the Introduction, this meaning is helpful but not necessarily complete with respect to what the identifier means in code. In essence, we believe our categorizations to be an orthogonal approach to English PoS; they both uncover data that the other technique is unaware of.

Another possible future application is in naming consistency and recommendation. Our evaluation shows that identifiers are used consistently in that they fall into the same category more often than not. Since this is the case, identifier usages that don't match historical usages may indicate poor naming. It would be easy to aware developers of when this happens. Further, it is possible to recommend identifier names to developers in cases where inconsistent use of name may occur or even as the developer is typing.

It is important to note that because an identifier is placed in multiple categories does not necessarily mean that it is categorized incorrectly. All this means is that its use is inconsistent to some extent. Identifiers present in multiple categories are interesting because they imply that these identifiers have more than one meaning to developers. If we think about how [3] concludes that synonyms and homonyms hurt comprehension, then we might theorize that words in multiple categories might have the same negative affect. It requires more study to be certain, but it is a possibility that identifiers in multiple categories may be identifiers that have many meanings and are just heavily overloaded or ambiguous. Future study of identifiers in multiple categories may reveal the nature of these identifiers; whether they are ambiguous/overloaded or something else and if they have a negative effect on comprehension.

Finally, we think it would be very interesting to explore how these categories might help create a database of identifier names similar to WordNet [43]. Recall that if we look between multiple systems as we did in RQ3 and Figure 6, the distribution falls very quickly with increased minsup. This may mean a few things: 1) Identifier usage between multiple systems requires more heavy normalization of identifier names in order to find correlation. 2) Identifier usage between multiple systems is not consistent; developers in differing systems use names in differing ways. This presents a challenge for future work, since construction of a database that categorizes words used as identifier names would be of interest to the community. More work will need to be done to investigate how matching identifiers between systems can be made possible or if it will be helpful. We need to take advantage of common NLP techniques such as stemming, synonym matching, etc. Some work has been done in this area [44], where they present a very interesting technique for extracting and grouping sets of identifiers together using the concept of hyponyms and hypernyms; their equivalent of WordNet's synsets. Using their data structure along with identifier normalizing techniques may enable us to combine our categories with NLP to construct a database to support identifier analysis for software engineering instead of having to rely on WordNet. If this future work is successful, it may become possible to study identifier usages across multiple systems and domains. The ability to limit or broaden the sample of identifiers used in analysis to include one or more than one system could be very valuable and help us identify and analyze patterns that cut across entire domains.

VIII. Conclusion

We have presented a set of lexical categories for source code identifiers. Each category is based on a set of rules that define the properties an identifier's declaration must have to be grouped under the given category. The goal of this work is to understand the relationship between an identifier's name and its declaration in order to ascertain if the developer's use of a given name implies a set of properties with respect to behavior in source code. Our evaluation shows that there is a strong correlation between an identifier's name and its declaration; more often than not, when a particular name is seen multiple times, it is grouped under the same category that it had been every time previously.

For this reason, we think further study of the relationship between identifier name and type is warranted. That is, based on our evaluation, it seems like developers may choose names because they imply a set of properties that are captured by the categories we have presented in this paper. In the future, we plan to research identifiers in multiple categories more rigorously, consider applications to identifier name consistency and recommendation, fully explore properties 4 and 5 (which we omitted in this work), and examine ways in which our categories can be used in conjunction with natural language processing.

References

[1] B. W. Boehm, *Software engineering economics* vol. 197: Prentice-hall Englewood Cliffs (NJ), 1981.

[2] L. Erlikh, "Leveraging legacy system dollars for e-business," *IT professional,* vol. 2, pp. 17-23, 2000.

[3] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal,* vol. 14, pp. 261-282, 2006.

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

[5] D. Binkley, M. Hearn, and D. Lawrie, "Improving identifier informativeness using part of speech information," presented at the Proceedings of the 8th Working Conference on Mining Software Repositories, Waikiki, Honolulu, HI, USA, 2011.

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

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

[8] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of NL-queries for software maintenance and reuse," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, 2009, pp. 232-242.

[9] S. L. Abebe and P. Tonella, "Natural Language Parsing of Program Element Names for Concept Extraction," presented at the Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension, 2010.

[10] G. Capobianco, A. D. Lucia, R. Oliveto, A. Panichella, and S. Panichella, "Improving IR-based traceability recovery via noun-based indexing of software artifacts," *Journal of Software: Evolution and Process,* vol. 25, pp. 743-762, 2013.

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

[12] W. Olney, E. Hill, C. Thurber, and B. Lemma, "Part of Speech Tagging Java Method Names," presented at the The 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME),, Raleigh, NC, 2016.

[13] A. Thies and C. Roth, "Recommending rename refactorings," presented at the Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering, Cape Town, South Africa, 2010.

[14] E. W. Host and B. M. Ostvold, "The Programmer's Lexicon, Volume I: The Verbs," in *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, 2007, pp. 193-202.

[15] E. W. Høst and B. M. Østvold, "Debugging Method Names," in *ECOOP 2009 – Object-Oriented Programming: 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, S. Drossopoulou, Ed., ed Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 294-317.

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

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

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

[19] R. J. Abbott, "Program design by informal English descriptions," *Commun. ACM,* vol. 26, pp. 882-894, 1983.

[20] E. Brill, "A simple rule-based part of speech tagger," presented at the Proceedings of the third conference on Applied natural language processing, Trento, Italy, 1992.

[21] T. Brants, "TnT: a statistical part-of-speech tagger," presented at the Proceedings of the sixth conference on Applied natural language processing, Seattle, Washington, 2000.

[22] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer, "Feature-rich part-of-speech tagging with a cyclic dependency network," presented at the Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1, Edmonton, Canada, 2003.

[23] K. Toutanova and C. D. Manning, "Enriching the knowledge sources used in a maximum entropy part-of-speech tagger," presented at the 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, 2000.

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

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

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

[27] M. Dickinson and W. D. Meurers, "Detecting errors in part-of-speech annotation," presented at the Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics - Volume 1, Budapest, Hungary, 2003.

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

[29] L. H. Etzkorn, C. G. Davis, and L. L. Bowen, "The language of comments in computer software: A sublanguage of English," *Journal of Pragmatics,* vol. 33, pp. 1731-1756, 11// 2001.

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

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

[32] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of

Source Code: A Tool Demonstration," presented at the Proceedings of the 2013 IEEE International Conference on Software Maintenance, 2013.

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

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

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

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

[37] E. Hill, D. Binkley, D. Lawrie, L. Pollock, and K. Vijay-Shanker, "An empirical study of identifier splitting techniques," *Empirical Softw. Engg.,* vol. 19, pp. 1754-1780, 2014.

[38] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," presented at the Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories, 2009.

[39] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. Pollock, *et al.*, "AMAP: automatically mining abbreviation expansions in programs to enhance software maintenance tools," presented at the Proceedings of the 2008 international working conference on Mining software repositories, Leipzig, Germany, 2008.

[40] D. Lawrie, H. Feild, and D. Binkley, "Extracting Meaning from Abbreviated Identifiers," in *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on*, 2007, pp. 213-222.

[41] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," presented at the Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 2014.

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

[43] G. A. Miller, "WordNet: a lexical database for English," *Commun. ACM,* vol. 38, pp. 39-41, 1995.

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