Part-of-Speech Tagging of Program Identifiers for Improved Text-based Software Engineering Tools

Samir Gupta
Computer and Information Sciences
University of Delaware
Newark, DE 19716 USA
sgupta@cis.udel.edu

Sana Malik
Department of Computer Science
University of Maryland
College Park, MD 20742 USA
maliks@cs.umd.edu

Lori Pollock and K. Vijay-Shanker
Computer and Information Sciences
University of Delaware
Newark, DE 19716 USA
{pollock, vijay}@cis.udel.edu

***Abstract*—To aid program comprehension, programmers choose identifiers for methods, classes, fields and other program elements primarily by following naming conventions in software. These software "naming conventions" follow systematic patterns which can convey deep natural language clues that can be leveraged by software engineering tools. For example, they can be used to increase the accuracy of software search tools, improve the ability of program navigation tools to recommend related methods, and raise the accuracy of other program analyses. After splitting multi-word names into their component words, the next step to extracting accurate natural language information is tagging each word with its part of speech (POS) and then chunking the name into natural language phrases. State-of-the-art approaches, most of which rely on "traditional POS taggers" trained on natural language documents, do not capture the syntactic structure of program elements.**

In this paper, we present a POS tagger and syntactic chunker for source code names that takes into account programmers' naming conventions to understand the regular, systematic ways a program element is named. We studied the naming conventions used in Object Oriented Programming and identified different grammatical constructions that characterize a large number of program identifiers. This study then informed the design of our POS tagger and chunker. Our evaluation results show a significant improvement in accuracy(11%-20%) of POS tagging of identifiers, over the current approaches. With this improved accuracy, both automated software engineering tools and developers will be able to better capture and understand the information available in code.

***Index Terms*—Program understanding, comprehension, part-of-speech, natural language processing, identifiers**

I. Introduction

Comprising 70% of source code [1], program identifiers are a fundamental source of information to understand a software system. Because programmers choose program names to express the concepts of the domain of their software, this natural language component of the program provides the reader with insights into developers' intent. It has already been demonstrated that many tools that help program understanding for software maintenance and evolution rely on, or can benefit from, analyzing the natural language embedded in identifier names and comments (e.g., concept location, comment generation, code refactoring) [2]–[5].

This material is based upon work supported by the National Science Foundation under Grant No. CCF 0915803.

Automated analysis of program identifiers begins with splitting the identifier into its constituent words and abbreviations to tokenize the name. Many text-based tools for software engineering then use part-of-speech (POS) taggers, which identify POS of a word and tag it as a noun, verb, preposition, etc. and then chunk (or parse) the tagged words into grammatical phrases to help distinguish the semantics of the component words. However, off-the-shelf POS taggers have difficulties tagging words found in software, especially source code, even when applying appropriate templates [3], [5], [6]. And, studies of using off-the-shelf text analysis techniques on software without customization show that the inaccuracies inhibit the effectiveness of the client software tools [7].

We informally surveyed software engineering researchers developing text-based software engineering tools and found that 58% of them use a POS tagger. For example, Abebe and Tonella [3] parse program identifiers to create an ontology used for concept extraction from source code. This extracted domain knowledge has been shown to help in program comprehension [8] and query reformulation in source code exploration [9]. POS tagging is also useful for mining semantically related words from source code. Falleri et al. [6] use POS tag information to extract WordNet-like [10] lexical views. Synonym information can improve code search [11] and improve automatically generated recommendations for program exploration from a starting point [12].

Sridhara et al. [4] leverage the natural language information generated by the Software Word Usage Model (SWUM) [13] to identify the action, theme and secondary arguments of a method name to automatically generate leading comments that describe a method's intent. SWUM requires POS tagging to identify the word usage information. Code refactoring can also benefit from POS tagging. For instance, Binkley et al. [5] identified rules to improve identifier names as an example application of POS tagging of source code. Verb-direct object pairs have been used to locate and understand action-oriented concerns [2], and require identifying the verbs and their associated direct objects in method signatures. Other examples of using POS tagging include improving software search and exploration [13] and increasing the accuracy of traceability recovery [14], [15].

Automated POS tagging and parsing in software is compli-

cated by several programmer behaviors and the way that POS taggers typically work. POS taggers for English documents are built using machine learning techniques that depend on the likelihood of a tag given a single word and the tags of its surrounding context – data that is trained on typical natural language text such as the Wall Street Journal or newswire training data [16]. These taggers work well on newswire and similar artifacts; however, their accuracy reduces as the input moves farther away from the highly structured sentences found in traditional newswire articles.

Word usage in the domain of software can be very different from general natural language documents. For example, the words 'fire', 'handle', 'set', and 'test' are overwhelmingly used in the verb sense in software, but are typically identified as nouns in natural language documents. Programmers invent new non-dictionary words and their own grammatical structures when naming program elements. For instance, they form new adjectives by adding "able" to a verb (e.g., "give" becomes "givable"). In addition, source code identifiers and comments follow very different grammar, may omit conjunctions and prepositions, and are typically written as imperative directives rarely encountered in traditional natural language training data. Thus, traditional POS taggers and parsers for natural language fail to accurately capture the lexical structure of program identifiers.

Some researchers have approached the problem by retraining existing natural language POS taggers for field names [5], but accuracy is 15-20% less than tagging natural language texts (under 80%). Others create sentences from identifiers and then run an existing tagger. Abebe and Tonella [3] transform each identifier into one of a set of template sentences, depending on whether it is a class, method, and attribute name under certain constraints and then input the sentences into a natural language parser for POS tagging and parsing. These template generated sentences guide the parser to treat identifiers in a specific way, which may work for typical cases but may be too constraining for other scenarios. Falleri et. al. [6] use TreeTagger [17], trained on English text to extract POS information from identifiers.

In this paper, we present a POS tagger and chunker, *POSSE* (POS tagger for Software Engineering), that is customized to software, and developed after considerable analysis of programmer conventions in identifier naming from our extensive work on natural language analysis in the development of SWUM [13], verb-DO pair identification [18], and the accurate analysis needed for comment generation [4]. Over the years, we have studied a diverse set of Java signatures and naming conventions by analyzing the most frequently occurring identifiers in a set of 9,000 open source Java programs downloaded from sourceforge.net. This set of programs contains over 18 million signatures, with 3.5 million unique names consisting of over 200 thousand unique words [13]. Developers follow patterns while naming identifiers in order to make their code readable. Leveraging these conventions, *POSSE* takes program identifiers as input and outputs their tagged part of speech and chunked syntactic phrases. *POSSE* is targeted towards object-oriented language identifiers including class, method and attributes. To evaluate our approach, we compared *POSSE*'s output over a test set of 310 Java and C++ Program Identifiers to a human annotated gold set. We measured accuracy of *POSSE* and compared it with two implementations based on current approaches of POS tagging of identifiers [3], [6]. Our results show an improvement of 11-20% over the current approaches.

The main contributions of this paper are:

- Analysis of programming conventions in naming classes, methods and attributes,
- An algorithm and implemented tool for POS tagging and syntactic chunking of program identifiers,
- Development of an annotated gold set that can be used to evaluate identifier POS tagging tools, and
- Evaluation and comparison with existing POS tagging approaches. The evaluation shows an increase of 11% to 20% in accuracy of POS tagging over the current approaches.

II. Challenges through examples

Table I shows examples that demonstrate some of the challenges in POS tagging of program identifiers, specifically method names which present the most problems for an automatic system. The first example illustrates the impact of abbreviations. The method name "concatTokens" should be tagged as "concat" (verb) and "tokens" (noun) with concat as the action verb. Since "concat" is a short form for "concatenation", WordNet [10] does not identify it as word; abbreviation expansion using techniques such as AMAP [19] are needed before POS tagging. After expansion, this example follows a straightforward (verb, direct object) convention for method naming, where the verb is the first word in the method name.

In the second example, the method is composed only of a single word. Naive approaches will tag these words as verbs, while sometimes, they should be nouns, such as in the example "spaces". A POS tagger can capture these situations by leveraging the typical POS of certain words in software.

The third example contains two words that could both be tagged as verbs or nouns in text documents or software, making it difficult to identify a single tag for each word. However, some words are predominately either verb or noun in software, especially in particular positions in the method name. In this example, "fire" in the software domain predominantly behaves as a verb, but plays a strong noun role in the natural language domain. A POS tagger for software should leverage the strong POS roles of words such as "fire" in software.

The fourth and fifth examples show how the main action verb is not necessarily the first word in the method name. The verb "set" occurs in the last position, and the direct object precedes it. It is still in a sense a verb phrase but it does not follow the grammatical construction of verb phrases typical in natural language text. An interesting scenario occurs in example 5 where "text" acts as an adverb modifying the main action verb "remove", indicating that "textRemove" is a type of remove operation. The last example indicates the issues

TABLE I
EXAMPLE METHOD NAME SCENARIOS

Method Signature	Actual Part of Speech	Automation Challenges
`String concatTokens(IIIZ)`	concat:*verb*; tokens:*noun*	Difficult to identify concat as verb since 'concat' is an abbreviation for concatenate. 'concat' is not recognized by WordNet as a word.
boolean `equals(QObject;)` `String spaces(I)`	equals:*verb* spaces:*noun*	A single word method name can be a noun. Difficult to identify spaces as a noun since the stemmed part can be both a verb and noun
void `fireCopy(QString;)`	fire:*verb*; copy:*noun*	Difficult to identify 'fire' as verb since fire is predominantly a verb in English text but acts as a verb in almost all cases in S/W domain
void `dynamicMultipleValuesSet(Z)`	dynamic:*adjective*; multiple:*adjective*; values:*noun*; set:*verb*	Difficult to identify 'set' in method signature as verb since it is at signature's end
void `textRemoveTabs2()`	text:*noun*; remove:*verb*; tabs:*noun*; 2:*noun*	Difficult to identify 'remove' in method signature as verb since it is preceded by a term which can be a verb in the first position
void `downloadManagerAdded(QDManager;)`	download:*noun*; manager:*noun*; added:*verb (past particle)*	Difficult to identify 'download' as an adjective and 'added' as event (past participle)

associated with conventional naming of methods that handle events. These types of events generally have a past participle at the end of the method name. The challenge is to identify such cases and not wrongly classify "download" as the verb. Identifying "download manager" as a single entity is important in handling such cases. *POSSE* is able to tag all the above examples with the correct POS tags.

III. PART OF SPEECH TAGGING

The main task of the *POSSE* POS tagger for software is to assign part of speech tags to each word of multi-word program identifiers, namely method, attribute and class names and perform syntactic chunking of the tagged words. We didn't consider POS tagging of method parameters and return type of identifiers explicitly, as they fall in the class of attribute and class names respectively.

The POS tags comprise a set of 12 tags:

- Noun
- Verb
  - Base verb (baseV)
  - Third-person singular verb (3PS)
  - Verb ending in "ing" (VBG)
  - Past tense verb (VBD)
  - Past participle (VBN)
- Adverb (adv)
- Adjective (adj)
- Closed list (fixed)
  - Article (art)
  - quantifier (quant)
  - pronoun (pro)
  - Preposition (prep)

In chunking, or parsing, the identifiers, the component words are chunked into syntactic phrases. For example, the method identifier *setCurrentBalance* is identified as a Verb Group (VG) followed by Noun Phrase, where "CurrentBalance" is identified as a Noun Phrase (NP) and "Current" as a Noun Modifier (NM). Specifically, we show the POS tagging and chunker output for "setCurrentBalance" where VG is verb group, NM is noun modifier and NP is noun phrase.

Input:	void setCurrentBalance()
POS Tagging:	set *(baseV)* current *(adj)* balance *(noun)*
Chunker Output:	*[set]:VG [[current]:NM balance]:NP]*

A. Study of Conventions in Program Element Names

Using a combination of manual analysis and scripts over millions of program identifiers from 11546 Java programs taken from 20 open source projects across multiple domains, we have studied programmer conventions in word usage and grammatical structures of program identifiers for methods, classes, and attributes. Our POS tagger was designed by first categorizing the different grammatical structures we observed. This section summarizes our observations for method, class and attribute names, focusing primarily on method names which have the most variety of forms.

Method Names: Most method names can be characterized as one of the following:

1) *Leading Verb*. Method names beginning with a verb are typically verb phrases and have an explicit action. We further classify method names with a leading verb according to the type of the leading verb.
   - *baseV*: Methods starting with a base form of verb (baseV) behave like "imperative sentences". For example, *delete* and *insert* are examples of methods with an explicit main action verb, but the object is not mentioned in the method name. In other cases, the object of the main action verb is indicated in the method name, such as *insertElement, deleteItem* and *showDialog*.
   - *3PS, VBG, VBD*: These method names begin with third-person singular (3PS), verbs ending in "ing" and "ed", respectively. Examples of method names beginning with 3PS are *isList, isOpen, hasElement* where the verbs are in a closed list, and *contains, exists, equals, creates* where the verb is an arbitrary verb in 3PS form. These kinds of methods typically function as "predicate" statements and having a

boolean return type. Leading verbs that are VBG or VBD are usually verbs behaving in the adjectival sense. For example, *serializedData, synchronizedList, pendingRelayMessage* are functions that typically return some object. In these cases, determining the appropriate POS tag to assign to the first term is difficult in the sense that it can be interpreted both as a derived verb (VBG/VBD) or an adjective. The Penn Tree Bank [20] provides guidelines on how to handle confusing and problematic cases among which are examples to distinguish between adjective and VBG.

2) *Reactive Names*. This category of methods represents event handlers, where the actual method name does not express the action performed by the method but rather suggests the context under which the method's action should be performed. For example, in method names such as *onOkPressed, actionPerformed, mouseReleased, connectionClosed*, the main intended action is "handle some event" where the event is written in the method name. Some names can also to interpreted to refer to the "state of an object". For example, *messageDeleted* may refer to "handle the event of the message being deleted" or may ask the question"has the message been deleted?". Typically, methods which describe a state or ask a question are of type boolean, while event-handling methods are non-boolean.
3) *Implicit Action Verb*. Methods fall into this category when they do not specify the specific action being performed. These methods can be further classified as "getters" and "converters":
   - *Getters:* These are typically Noun Phrases that may be constructors or methods where the implicit action verb is "get" or "compute" and they return or modify data objects. Examples include *squareRoot, elementAt, endColumnNumber, synchronizedList*.
   - *Converters:* Methods with names that have the implicit action verb "convert " belong to this class. These are typically of the form *(NP)'to'NP*. Examples include *toString, listToArray, littleToBigEndian, viewToModel*.
4) *Non-Leading Action Verb*. In some cases, the action verb is explicitly expressed in the method name but is not in the typical "verb position in method names", which is the leading word. Consider the methods *lruRemove* and *dynamicMultiValueSet*. The main action verb is "remove" and "set" respectively, which occur at the end of the method name. Methods like *textRemoveTabs* and *tagDefineShape* have the action verb in the middle of the method name. A noun phrase preceding the action verb suggests a specialization of the action denoted by the verb. The noun phrase can be viewed as behaving in adverbial sense. That is, "lruRemove" can be viewed as a special kind of "remove".

Attributes and Class Names. We did not observe as much variety in names of attributes and class names as method names. Class names are a user-defined data type and are typically noun phrases. Similarly, attributes represent some kind of data and are also typically noun phrases. We found that most boolean type attributes are 'predicates' and ask a question whose answer (value) is true or false. These attribute names generally are characterized as Verb Phrases. Examples of boolean type attribute names which act as a VP are *isDescending, isPrimaryKey, displayingEvents, createTempTable*.

B. Overview of Approach

Figure 1 presents the phases of *POSSE*. We first split the identifier into its component words using the Samurai identifier splitter [21]. We loosely call them "words" as most are words, however, some may be abbreviations or acronyms.

The *POSSE* approach is broadly divided into two phases:

1. Tagging all possible part-of-speech of each word in an identifier.
2. Using POS tag sets for each word and set of grammatical structures identified from programming conventions, chunk the identifier into its lexical components and choose a particular POS for each word.

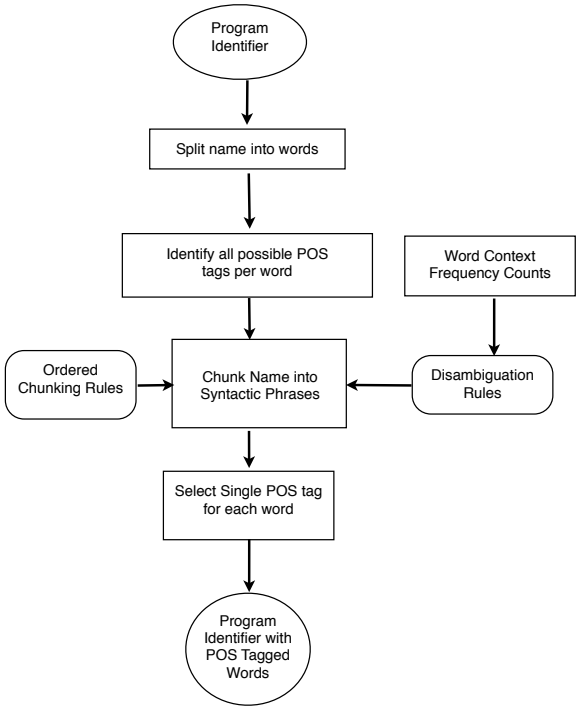

Fig. 1. The *POSSE* Approach to POS Tagging of Program Identifiers

We use WordNet and morphological rules to assign all possible POS tags to each word. For example, the word 'request' in the method name 'decodeRequest()' can be both a noun and a verb as identified by WordNet. The tags are assigned from the list of 12 POS tags presented earlier.

After identifying all possible POS tags for each word in the identifier, we use context to select the appropriate POS tag from the set for the particular instance of the words and group words into meaningful lexical phrases. Specifically, we use the POS tags of the surrounding words and the position of the word within the method identifier. The context is considered in terms of the type of lexical phrase structure being considered.

Based on our study of programmer conventions, we developed a set of grammatical constructions (i.e., chunking rules) which are derived from combinations of one or more of the basic lexical phrases. We determine which grammatical construction best matches the program identifier under analysis. We grouped the grammatical constructions under three sets which are tried in a default order, which depends on the kind of identifier is being considered (method/class/attribute). Because words typically have multiple parts of speech, it is sometimes possible to apply more than one grammar construction to a identifier. For example, the method name *squareRoot* can be parsed as 'square' as the action verb and 'root' as its object, thus implying a Verb Group(VG) followed by its argument. Alternatively, it could be parsed as Noun Phrase where 'square' is a Noun Modifier modifying the noun "root". Thus we might need to override the default order for which we developed certain disambiguation rules.

To resolve ambiguities, we use "Word Context Frequency Counts", which we call the *NVScore* for words. The *NVScore* of a word is based on the probability of the word being a noun or verb in software. It is used to override default order of identifying appropriate grammatical construction for an identifier, which will affect the choice of POS tag for words within the name.

C. Algorithm Details

We present the details of identifying all possible POS tags for a given word in a name, chunking the name into basic lexical phrases, and selecting the grammatical construction, which leads to trivial identification of the appropriate POS tag for each word in the name.

Finding All Possible POS Tags: Given a program identifier split into words, this phase outputs a set of possible POS tags for each word, ignoring context of the word within the method name. Each word in the identifier is treated independently while assigning all possible POS tags for it.

1) For each word in the identifier, call WordNet to return all possible POS tags. For example, consider the word 'request' in the method name "decodeRequest()". WordNet identifies "decode" as a noun and "request" as both noun and verb, so output of this phase is: *decode(baseV), request(noun,baseV)*.
2) WordNet may not always be able to identify a POS tag for a word that occurs in an identifier. These cases occur when the word being analyzed is not a "true word". Programmers often invent new words by adding certain prefixes/suffixes (re-, de-, un-, -able, etc) to form words like *givable, unmarshall, deserialize, relable* . If WordNet fails to return any POS tags, then to identify the role of a word, we use suffix and prefix information and morphological rules to identify the word's possible POS tags. For example adding '-able' to the verb 'give' forms an adjective 'giveable'. Similarly words with common verb prefixes like 'de-', 're-' are tagged as verbs.

Chunking into Basic Lexical Phrases: The study of naming conventions resulted in a set of categories of method names that can be expressed as a set of grammatical constructions. Each of these involves a combination of basic lexical phrases. Our system uses six of the same lexical phrases that are commonly found in the English language [22]. In this section, we describe each type of phrase, give examples of how they are identified in program identifiers, and present the matching pattern used in the syntactic parser.

Noun Modifier Phrase (NM): A noun modifier phrase is any combination of nouns, adjectives, or determinants that precedes a noun. Consider the attribute name *tablerowheader*. Here, "table row" modifies the noun "header" and is identified as an NM. The lexical pattern for a noun modifier is:

$$\text{NM} \quad \rightarrow (noun|adj|VBN|VBG|quant|pro)^*$$

Noun Phrase (NP): Noun phrases consist of any type of noun (singular, plural, gerund, or pronoun), optionally preceded by a noun modifier phrase (NM). For example, the class name *currentBalance* will be parsed as an NP with "current" being parsed as an NM phrase. We introduce the following syntactic chunking notation: $[[current]_{NM} balance_{NP}]$ Two noun phrases joined by "of" also make a noun phrase, as in "size of array". The lexical pattern for a noun phrase is:

$$\text{NP} \quad \rightarrow \text{NM?}\ noun|VBG|pro$$

Verb Modifier (VM) : Verb Modifiers are adverbs (adv). The lexical pattern is simply:

$$\text{VM} \quad \rightarrow adv$$

Verb Group (VG): A verb group ends with any word from the verb category (baseV, 3PS VBG, VBD) and is optionally preceded by helping verbs or verb modifiers (VM). The lexical pattern is:

$$\text{VG} \quad \rightarrow \text{VM?}\ baseV|VBG|3PS|VBD$$

Past Participle Phrase (PLP): A past participle phrase is a noun phrase followed by a past participle. For example, in the method name *isButtonPressed*, "button pressed" is a past participle phrase with chunked notation: $[[button]_{NP} pressed]_{PLP}$. The lexical pattern is:

$$\text{PPL} \quad \rightarrow VBN|NP(\text{“}is\text{”}|\text{“}has\ been\text{”})VBN$$

Prepositional Phrase (PP): A prepositional phrase is a noun or Past Participle Phrase preceded by a preposition. For example, in the method *movePaneltoFront*, "to front" is a PP. The

chunked notation is $[move]_{VG}[panel]_{NP}[[to][front]_{NM}]_{PP}$. The lexical pattern is:

PP $\rightarrow prep\ NM|PPL$

Selecting the Identifier's Grammatical Construction: The program identifiers tagged with all possible POS tags are tested against a set of possible grammatical constructions involving a combination of the basic lexical phrases in an order we settled on after analyzing method, class and attribute names. If an identifier cannot be of a phrase structure, then the next phrase structure is tested. Because of the different roles of methods, attributes and classes and their naming convention differences, our algorithm for selecting a candidate grammatical construction is different for method, attribute and class names.

After the appropriate grammatical construction is determined for a given identifier, successfully assigning POS tags is a trivial task with exception of Noun Modifiers. The POS tags for the words in the identifier are those tags in the set of all possible tags for each word that fit the grammatical structure selected for the identifier. Determining the tags for words in an NM is more difficult, as some words can be both adjectives and nouns in an NM. For words in a NM phrase which can be both noun and adjective, we assign a noun tag if the word is the last term of the identifier otherwise a adjective tag is assigned.

I. Method Names

Based on our categorizing of different grammatical structures we observed for method names, we grouped the observed grammatical constructions of basic lexical phrases into three sets:

1) VG-starting
2) NM/Noun-starting
3) Reactive and Converter methods

Typically, names start with a Verb Group (VG), such as *sortList*. However, sometimes a method name appears to start with a verb according to WordNet which assigned it a possible tag of verb, when in fact, the first word is really not intended to be a verb in the method name. For example, WordNet provides verb as a possible tag for 'value' in *valueSet*, when in fact it is being used as a Noun Modifier. If we always select the VG-starting constructions for every identifier that started with a word that WordNet returns verb as a possible tag, we would mistag many identifiers and also chunk them incorrectly as verb phrases. This problem occurs because WordNet does not differentiate between the likelihood of a word being a verb or noun in software. For example, it is highly unlikely that the word 'value' will be used as a verb in the programming domain. To enable the POS tagger to leverage the fact that word usage in software differs from usage in English documents, we have developed a score for a word, called the *NVScore*, (Noun-Verb Score), that captures the likelihood of the word being used mostly as noun or mostly as a verb in the programming domain.

Calculation of the NVScore: Based on our analysis of method names, our hypothesis is that if a word appears most frequently over a large corpus of method names as the first word in method names, then it is probably used primarily as a verb in software. Specifically, we consider 'verb' positions in method names to be at the start of the name and single-word method names. Likewise, we found that the last word of a method name is predominantly a "noun position". We do not infer that every word appearing at the end of a method name is a noun. Instead, we believe that a word occurring at the end of many method names as compared to the beginning of method names is a good indicator that it is usually used in the noun sense. We consider 'noun' positions to be at the end of method and attribute names and single-word attribute names.

To establish our *NVScore* for individual words, we collected 81,873 method names and 31,533 attribute names across 20 open source Java projects. We counted the number of times that each word appeared in the noun and verb positions. We define the Noun-Verb Score, *NVScore* for a word to be:

$$NVScore(word) = \frac{freq(word\,at\,verb\,positions)}{freq(word\,at\,noun\,positions)}$$

A word with a higher (lower) NVScore is more likely used in programs in a verb (noun) sense. After several iterations of analysis, we conservatively set 0.9 as the threshold above which we consider a word to be a *strong verb*, and 0.1 as the threshold under which we consider a word to be a *strong noun* in software. For example, using these thresholds, we determine that *document, model, lock, mask* are strong nouns and *fire, parse, add, set* are all strong verbs in software.

Selecting Grammatical Constructions using NVScore: We use NVScore of the first word in a method name to determine if it is considered to be strong noun. If so, we begin selection of the grammatical construction for the identifier under analysis by examining the NM/Noun-starting constructions. For example, in the identifier "squareRoot" where WordNet returns both noun and verb as possible tags for 'square', the NVScore for 'square' is 0.08 indicating a strong noun, and thus we do not consider the constructions in the VG-starting set. If the first word doesn't have a verb tag as a possibility and can be an adjective, noun or from the closed list containing 'on', 'my' and similarly others, then also we do not consider the VG-starting set of constructions. VG-starting set of constructions are also skipped, if the last word is a past participle and the word preceding it is a 'strong noun' as in the case of *downloadManagerStarted*.

The remainder of this section describes the grammatical constructions that comprise the three grammatical constructions sets that we defined for POS tagging and chunking based on programmer conventions.

VG-starting Grammatical Construction Set: This set includes typical cases of method names that start with Verb Groups. After the initial VG, the name can be optionally followed by its arguments. Based on our study of method names, there are seven such grammatical constructions in this group:

1) *VG*: Verb Group with no arguments
2) *VG NP*: Verb Group followed by a Noun Phrase
3) *VG NP PP*: Verb Group followed by a Noun Phrase and Prepositional Phrase
4) *VG PP*: Verb Group followed by Prepositional Phrase
5) *VG NP prep*: Verb Group followed by a Noun Phrase and preposition.
6) *VG prep*: Verb Group followed by a preposition.
7) *VG PPL*: Verb Group followed by a Past Participle Phrase.

The order to select a grammatical construction for a given identifier among these 7 constructions is not important, because they express non-overlapping structures, that is, they are mutually exclusive. The matching of these constructions against the identifier with all possible tags, as well as the chunking into base phrases, is performed by regular expression matching. The identifiers *generateAPIList, buildOptimizer* and *createTable* are some examples that are matched by VG-starting constructions.

NM/Noun-starting Grammatical Construction Set: This set consists of grammatical constructions that begin with a Noun Modifier or Noun. Names such as *dynamicValueSet* and *squareRoot* match these constructions. In *dynamicValueSet*, "dynamicValue" forms an NP, and acts as a Modifier of the verb "set" (the method itself sets some value). In contrast, the entire name "squareRoot" forms an NP, and the method name has an implicit verb (i.e., compute or get). We need to distinguish between cases that have a VG after an NP versus cases where the entire name is an NP. We first check whether the identifier fits the first case of NP VG by checking if it starts with an NP and is followed by word whose NVScore exceeds a threshold indicating it is a verb. Note that the verb can be followed by its arguments as in VG-starting constructions. If the NVScore of the word following the NP doesn't exceed the threshold for being a verb, we chunk the entire name as a NP. *synchronizedList, toggleToolBar, error* are examples that are also detected by constructions in this set and chunked as NP.

Reactive and Converter Method Construction Set: This set contains constructions that correspond to names commonly identifying reactive methods and converter methods. Our study revealed that reactive names come mostly in two forms: either ending in a Past Participle Phrase (PPL) or starting with a Prepositional Phrase (PP). When a name ends with a PPL, the preceding part of the name represents its arguments. Usually this argument is an NP. A typical example is the method *keyPressed* which is an event-handler reacting to the situation of "the key is pressed". Reactive names such as *onFailure* can also start with prepositions. These names are invariably just a prepositional phrase (PP). In addition to the reactive names this group also has constructions involving the preposition "to". A method named *toString* performs the task of converting an object to string. Such converters methods can also include the word "to" in the middle of the name. To recognize such cases, we check for the grammatical construction "NP 'to' NP". *windowClosed*, *onPostRemoveCollection* and *listToArray* are examples that matched by constructions in this set.

II. Attribute and Class Names

As we stated in our study of program identifiers, attribute and class names are generally Noun Phrases, thus we first check if the identifier matches any grammatical constructions in the NM/Noun-starting grammatical construction set. Within that set, we check whether the identifier is an NP before checking for the NP VG construction, when processing attribute and class names. The only exception are attribute names that are boolean type attributes. These attribute names are treated the same as method names, first checking whether the identifier matches a construction in the VG-starting grammatical construction set. An example is the attribute name *boolean sorted*.

IV. Evaluation

We designed our evaluation to answer the following research questions:

RQ1. How accurately does *POSSE* assign POS tags to words in Java identifiers, namely method, attribute and class names?

RQ2. How well does *POSSE* apply to other object oriented programming languages?

RQ3. How well does *POSSE* compare to the 'state-of-the-art'?

We designed an experimental setup in which human annotators participated to create a gold annotated set that was used to test the accuracy of *POSSE*. The annotated test set consisted of method, attribute and class names extracted from Java and C++ programs, which enabled us to address RQ1 and RQ2. We compared results from *POSSE* with two implementations of current approaches to tagging program identifiers with POS information to answer RQ3. In this section, we present the design of the experiment including a brief discussion of our implementation of the "state-of-the-art" approaches that we compared against, results of comparing *POSSE* with these taggers, an analysis of the results, and finally a discussion of threats to validity.

A. Experiment Design

Subjects, Variables and Measures: The subjects in our experiment were identifiers from the two most popular object oriented programming languages, Java and C++. We extracted program identifiers from 20 open source projects written in Java and 6 in C++ across multiple domains. In total, the projects are comprised of approximately 3 million lines of code from 14,989 program files.

The independent variable in our study is the set of POS taggers being compared. In addition to implementing *POSSE*, we implemented two systems based on the current approaches that perform POS tagging on program identifiers. The first system is the *TreeTagger*, which was used by Falleri et. al. [6]. In this system, the identifier is separated into a series of words, which is input to the POS tagger, TreeTagger. This POS tagger, which was developed for general English as well as other languages, is freely available online. The second system, which we will call *TemplateTagger* is based on the system developed by Abebe and Tonella [3]. It uses the templates that are described in [3] to generate phrases for a given

identifier based on the possible POS tags of the first word in the identifier. The possible tags are obtained by looking them up in WordNet. The generated phrases are then input to a broad based coverage parser called Minipar. In their work, rather than just doing POS tagging, the focus is on ontology construction. Thus, in our implementation, *TemplateTagger* we use the POS tags assigned by Minipar when it parses the phrase generated by the templates. This is also similar to the approach of Binkley et. al [5] where phrases are first generated using templates and then POS tags are determined by using an NLP tool that processes these phrases. Thus, *TemplateTagger* implements the only other approach for POS tagging that we are aware of that specializes for the software domain. We measure and compare effectiveness of systems by computing accuracy, defined as the percentage of program identifiers in the test set that are correctly tagged (i.e., every word in the identifier has the correct POS tag).

Methodology: While extracting method names from the Java and C++ programs, we extracted all method, attribute and class names, except method names that were of the type "getWord" and "setWord", where Word is a single term. This purposely excludes a large number of relatively simple cases, thus the results are dominated by these method names. We did not exclude completely the 'get' and 'set' form of method names. For example, getMultiValueHash was not excluded from our experiment. From the Java identifiers, we randomly extracted 100 method names, 60 class names and 50 attribute names which formed the Java program identifier test set. We extracted 50 method names, 25 class names and 25 attributes for C++ test set.

We created a gold set by having humans annotate both test sets. Our annotators consisted of two human subjects, neither of whom are the authors of this paper. One of them has extensive expertise in natural language processing and linguistics, whereas the other has considerable programming knowledge and experience. We asked the annotators to discuss and jointly tag each identifier resolving any differences.

The annotators were given the identifier category (method, class, attribute), the entire identifier signature with return type, the programming language that the identifier belongs to (Java or C++) and the identifier split into its component words. They were asked to tag each word in the identifier with 5 basic part of speech tags: *Noun*, *Verb* (base and all its forms), *adjective*, *adverb* and *closed list*. The closed list is intended for articles, quantifier, pronouns, prepositions and other categories. This reduced tag set was chosen primarily to reduce the burden on the annotators, make the annotations simple and unambiguous and allow for easy comparison of the different tools.

B. Results

Table II shows the accuracy of output from *POSSE*, *TemplateTagger* and *TreeTagger* on the 210 Java identifiers in the test set. Overall, 91.4% of the Java identifiers were correctly tagged by *POSSE* while *TemplateTagger* and *TreeTagger* achieved an accuracy of 75.7% and 77.1%, respectively. Table III shows the percentage of the correctly tagged identifiers from the 100 C++ identifiers in the test set, for the three taggers. Combined (method, class and attribute names), *POSSE* correctly tagged 85% of the C++ identifiers, while *TemplateTagger* and *TemplateTagger* achieved accuracy of 65% and 74%, respectively. Our results indicate an 11%-14% increase in performance over the closest tagger: *TreeTagger* and a 15%-20% increase over the *TemplateTagger*.

Both of these tables also show the separation of results into the three individual identifier categories: methods, attributes and classes. *POSSE* achieved a higher accuracy for all of the identifier types, both in Java and C++. The biggest improvement is in tagging method names (20%-24%), which has much more variety than the other two types of names.

TABLE II
ACCURACY RESULTS FOR JAVA IDENTIFIERS

	POSSE	**TemplateTagger**	**TreeTagger**
Methods	94%	70%	69%
Attributes	88%	88%	84%
Classes	90%	75%	85%
Combined	91.4%	75.7%	77.1%

TABLE III
ACCURACY RESULTS FOR C++ IDENTIFIERS

	POSSE	**TemplateTagger**	**TreeTagger**
Methods	82%	56%	62%
Attributes	96%	88%	92%
Classes	80%	60%	80%
Combined	85%	65%	74%

C. Error Analysis

In this section, we present a sampling of the "errors" made by each of the taggers. By "errors", we imply disagreements between the automated output of the taggers and the developers' annotations.

POSSE: In the case of method names, our approach failed to identify cases where verbs occurred both at the start and the end. For example, in the method name *addDiscriminatorToInsert*, the annotators identified both 'add' and 'insert' as verbs, while *POSSE* only identified 'add' as a verb. None of our grammatical constructions allowed for a verb to both begin and end a method name.

POSSE had difficulty with method names like *logQueued* where both words are ambiguous. *POSSE* tagged "logQueued' as a noun followed by a verb implying "handle the event of the log being queued", while the developers' annotation identified it as a verb followed by an adjective implying "log (register) the activity queued". In the non-boolean attribute name *deployMinesCommand*, *POSSE* was unable to tag 'deploy' as a verb. This is due to the fact *POSSE* gives preference to noun starting construction while processing non-boolean attribute names. *POSSE* correctly identified class names like *naivesBayesSimple* as a Noun Phrase but incorrectly tagged the last term 'simple' as a noun rather than an adjective.

TemplateTagger: One group of errors made by *TemplateTagger* includes incorrectly tagging 'leading verbs' in a

method name as ‘nouns‘ or ‘adjectives’. Examples include *open, testList, updatePrefs*. Minipar overrides constraints put on the ‘templates’ generated for these sentences on such kinds of method names. For example, consider the method name ‘testList’. The *TemplateTagger* generates the sentence ‘subjects test list‘. When this sentence is processed by Minipar, it produces a dependency tree that identifies the entire sentence as an NP and interprets ‘test’ as an adjective.

Some nouns and adjectives in the middle of method names are incorrectly identified as verbs by *TemplateTagger*. Examples include *getReportFrequency* and *initializePoseTransforms* where nouns such as ‘report‘ and ‘transforms’ are tagged as verbs instead. Similar errors are encountered in cases of attribute names (e.g. *pngFormatName*) and class names (e.g. *jftpDataStreamHandler*) where nouns like ‘format’ and ‘stream‘ are tagged as verbs. Despite the appropriate choice of templates, Minipar’s preferences are based on the assumption that these are phrases from standard English.

TreeTagger: Since *TreeTagger* has no component that attempts to specialize to the software domain, as expected, we find the same kind of errors that we found in *TemplateTagger* in *TreeTagger*. For example, in the method name *showLoadDialog*, *TreeTagger* incorrectly tags the verb ‘show’ as a noun. Examples of class names include *complementNaiveBayes* and *querySelect* where the adjective ‘complement‘ and the noun ‘query’ are incorrectly classified as verbs. *TreeTagger* also fails to identify the verb ‘use’ in the boolean attribute name *mUseDiscretization* and tags it as a noun. As indicated by the results, *TreeTagger* performs better than *TemplateTagger* even though it does not attempt to specialize to the software domain. We speculate that this might be due to the fact that *TreeTagger* performs better at POS tagging than Minipar.

D. Threats to Validity

Our study and development were based on Java program identifiers. We thought that the ideas developed could be extended to C++ because of its similarity with Java. In fact, our results indicate *POSSE* performs similarly across Java an C++. This may not be conclusive proof that the results can be extended to any object oriented programming language (e.g., Ruby, JavaScript or Python). Non OOP languages may have different syntax which may not been captured by our tagger.

As with any human-based annotations, there might be some cases where the annotators may not have correctly identified the part of speech tag. Annotators must have good knowledge of how identifiers are named in Java and other OOP languages as well have a good background in understanding POS tagging. To limit this threat, we had an annotator with extensive expertise in natural language processing and linguistics (particularly in the POS tagging task) and a second annotator with extensive programming knowledge sit together, discuss and annotate.

V. Related Work

Several efforts have taken program identifiers as input and generated sentences or phrases to input to a classic POS tagger for English text. Abebe and Tonella [3] applied natural language parsing to sentences generated from the terms in a program element. Depending on the program element being named (class, method, or attribute) and the role (noun/verb) played by the first term in the identifier, different templates are used to wrap the words of the identifier to form sentences, which are input into Minipar, a descendant of Principar [23] to construct parse trees. Multiple sentences may be generated if the first term in a method name is identified by WordNet as both as noun and verb. For example, in the method name “processMail”, ‘process’ is tagged as both a verb and noun by WordNet, and the following sentences are generated: “subjects process mail” & “subjects get process mail”. These sentences are then parsed by Minipar. Based on how these candidate sentences are parsed by Minipar, rules are then applied to pick one of the candidate sentences. The Minpar parsing of the chosen sentence and its tagging of the individual words in the sentence are used construct an ontology.

Binkley et. al. [5] also discuss part of speech tagging but limit it to field names. They followed the Abebe and Tonella approach of using templates. Although they use fewer templates (List, Sentence, Noun and Verb) than Abebe and Tonella, they produce sentences with each template which are then input to a POS tagger developed for the general English domain. Unlike the Abebe and Tonella system, they do not output a unique tag for each word, but rather produce four different tags, one for each template (and the resulting sentence). They leave it for future work to choose a single tag from those output by these four different “taggers”. They use a more recent tagger: Stanford Log-linear POS Tagger [16] than Minipar. Of course, Minipar is a parser and assigns dependency structure as opposed to the Stanford Tagger which only assigns POS information. Falleri et al. [6] simply uses the TreeTagger [17], a tagger trained on English text to perform the POS tagging. This closely relates to the sentence template used by [5] except the use of a different language parser.

Shepherd et al. [24] extracted verb and direct object information from method names by approximating possible POS for terms in method names, giving preference to method names being Verb Phrases by favoring the verb tag for words in the first position. While this works well for methods where the first word is the action verb, it is not comprehensive, as indicated by our study of naming conventions. Caprile and Tonella [25] developed a grammar for C function names. Hill [13] developed the Software Word Usage Model in which a set of identifier grammar rules for Java were developed. Liblit, et al. [26] examined common naming conventions and identified morphological patterns which can serve as a starting point for development of a parsing tool. A phrase book of the most common method patterns was created by Høst and Østvold [27]. Though these rules describe most of the patterns in naming, the rules cannot be applied to tag arbitrary method signatures and is only limited to method names.

Our work is distinguished from the prior work as a specialized POS tagger for the software domain, able to handle the common as well as the less common naming conventions equally well and not limited to method names.

VI. Conclusion

Our study of naming conventions suggests that method names may not follow the typical verb-argument phrase structure. Although a majority of method names start with a Verb Group followed by a Noun Phrase, this may not always be the case. Our approach does not constrain an identifier to satisfy a specific phrase structure, which is used in current approaches to guide English language based POS taggers. For attributes and classes, we let the different categories of constructions decide which phrase structure that the identifier might satisfy depending upon its all possible POS information. Having this flexibility and a method to identify "strong and weak" nouns/verbs in the software domains helps us to accurately assign part of speech information to identifiers. The evaluation indicates an increase of 11% to 20% in accuracy of POS tagging of program identifier sover the current approaches. This improvement is over both programming languages (Java and C++) as well as across each category of identifiers (methods, attributes and classes). We believe this increased accuracy will enhance the performance of other software analysis and maintenance tools which rely on part of speech and parsing of identifier names in source code.

References

[1] F. Deissenbock and M. Pizka, "Concise and consistent naming [software system identifier naming]," in *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, may 2005, pp. 97 – 106.

[2] D. Shepherd, Z. Fry, E. Hill, K. Vijay-Shanker, and L. Pollock, "Using natural language program analysis to locate and understand action-oriented concerns," in *Proceedings of the International Conference on Aspect Oriented Software Development*, Apr 2007.

[3] S. L. Abebe and P. Tonella, "Natural language parsing of program element names for concept extraction," in *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, ser. ICPC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 156–159. [Online]. Available: http://dx.doi.org/10.1109/ICPC.2010.29

[4] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 43–52. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859006

[5] D. Binkley, M. Hearn, and D. Lawrie, "Improving identifier informativeness using part of speech information," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 203–206. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985471

[6] 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*, 30 2010-july 2 2010, pp. 4 –13.

[7] G. Sridhara, E. Hill, L. Pollock, and K. Vijay-Shanker, "Identifying word relations in software: A comparative study of semantic similarity tools," in *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*. IEEE Computer Society, 2008, pp. 123–132.

[8] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *Proceedings of the 10th International Workshop on Program Comprehension*, ser. IWPC '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 271–. [Online]. Available: http://dl.acm.org/citation.cfm?id=580131.857012

[9] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 232–242. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070524

[10] C. Fellbaum, *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[11] J. Yang and L. Tan, "Inferring semantically related words from software context," in *Proceedings of the Working Conference on Mining Software Repositories (MSR'12)*, June 2012.

[12] E. Hill, L. Pollock, and K. Vijay-Shanker, "Exploring the neighborhood with dora to expedite software maintenance," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 14–23. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321637

[13] E. Hill, "Integrating natural language and program structure information to improve software search and exploration," Ph.D. dissertation, Aug 2010.

[14] 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*, 2012. [Online]. Available: http://dx.doi.org/10.1002/smr.1564

[15] G. Capobianco, A. De Lucia, R. Oliveto, A. Panichella, and S. Panichella, "On the role of the nouns in ir-based traceability recovery," in *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, may 2009, pp. 148 –157.

[16] K. Toutanova, D. Klein, C. Manning, and Y. Singer, "Feature-rich part-of-speech tagging with a cyclic dependency network," in *Proceedings of HLT-NAACL 2003*, 2003, pp. 252–259.

[17] H. Schmid, "Probabilistic part-of-speech tagging using decision trees," in *Proceedings of the International Conference on New Methods in Language Processing*, Manchester, United Kingdom, 1994.

[18] Z. P. Fry, D. Shepherd, E. Hill, L. Pollock, and K. Vijay-Shanker, "Analysing source code: looking for useful verb-direct object pairs in all the right places," *IET Software Special Issue on Natural Language in Software Development*, vol. 2, pp. 27–36, Feb 2008.

[19] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. Pollock, and K. Vijay-Shanker, "AMAP: Automatically mining abbreviation expansions in programs to enhance software maintenance tools," in *MSR '08: Proceedings of the Fifth International Working Conference on Mining Software Repositories*. Washington, DC, USA: IEEE Computer Society, 2008.

[20] B. Santorini, "Part-of-speech tagging guidelines for the penn treebank project (3rd revision)," 1990.

[21] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," in *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, ser. MSR '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 71–80. [Online]. Available: http://dx.doi.org/10.1109/MSR.2009.5069482

[22] C. D. Manning and H. Schuetze, *Foundations of Statistical Natural Language Processing*. The MIT Press, 1999.

[23] D. Lin, "Principar: an efficient, broad-coverage, principle-based parser," in *Proceedings of the 15th conference on Computational linguistics - Volume 1*, ser. COLING '94. Stroudsburg, PA, USA: Association for Computational Linguistics, 1994, pp. 482–488. [Online]. Available: http://dx.doi.org/10.3115/991886.991970

[24] D. Shepherd, L. Pollock, and K. Vijay-Shanker, "Towards supporting on-demand virtual remodularization using program graphs," in *Proceedings of the 5th international conference on Aspect-oriented software development*, ser. AOSD '06. New York, NY, USA: ACM, 2006, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/1119655.1119660

[25] B. Caprile and P. Tonella, "Nomen est omen: Analyzing the language of function identifiers," in *Proceedings of the Sixth Working Conference on Reverse Engineering*, ser. WCRE '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 112–122. [Online]. Available: http://dl.acm.org/citation.cfm?id=832306.837072

[26] B. Liblit, A. Begel, and E. Sweetser, "Cognitive perspectives on the role of naming in computer programs," in *Proceedings of the 18th Annual Psychology of Programming Workshop*. Citeseer, 2006.

[27] E. W. Host and B. M. Ostvold, "The programmer's lexicon, volume I: The verbs," in *SCAM '07: Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society, 2007, pp. 193–202.