Specifying and Evaluating a Software Word Usage Model

Emily Hill, Christian Newman, Giriprasad Sridhara
Patrick Francis, Lori Pollock, K. Vijay-Shanker, and David Shepherd

Abstract—Textual information has become a growing source of data for a variety of software maintenance tools, including program search and navigation, traceability link recovery, and code reuse. To maximize the benefits of textual information, there is a growing need for researchers and software engineering (SE) tool developers to be experts not just in SE concepts, but also in natural language and text processing techniques to effectively create new SE tools. In this paper, we introduce a novel Software Word Usage Model (SWUM) that captures the conceptual knowledge of a programmer expressed through both natural language information and programming language structure and semantics. In contrast to existing ”bag of words” text-based approaches, we take a transformative step in automatic analysis for SE tools by accounting for how words occur together in code, rather than just counting frequencies. We have developed and implemented a set of heuristics to automatically extract SWUM from object oriented code written in C++ and Java, which we evaluate for accuracy with a user study. In addition to evaluating accuracy, we evaluate the model using two concrete applications: method summarization and part of speech tagging. We further evaluate the model by showcasing its generality to other SE tasks.

Index Terms—Software word usage model, program comprehension, software maintenance, natural language processing for SE, code summarization.

1 Introduction

TEXTUAL information has become a growing source of information for a variety of software maintenance tools. Examples include tools that support program navigation and search [1], [2], automatic recovery of traceability links between software artifacts [3], [4], and code reuse [5], [6]. Most earlier textual approaches treated a program as a “bag of words” [7], i.e., words are viewed as independent occurrences with no relationships. We use the term *lexical concept* to describe a concept evoked by a single word, and *phrasal concept* to describe a concept expressed as a sequence of words [8]. Although most existing approaches use lexical concepts, phrasal concepts can enable much richer and more accurate sources of textual information to improve existing SE tools such as feature location [9] or enable completely new techniques such as automatic summary comment generators [10], [11], [12], [13], [14].

Thus, to maximize the benefits of textual information with phrasal concepts, there is a growing need for researchers and software engineering (SE) tool developers to be experts not just in SE concepts, but also in natural language and text processing techniques to effectively create new SE tools. A program model that integrates both program structure and linguistic information would enable researchers and practitioners to more easily leverage this information. In this paper, we introduce a novel **Software Word Usage Model (SWUM)** that not only captures the occurrences of words in code, but also their linguistic and structural relationships. SWUM captures the conceptual knowledge of a programmer expressed through both natural language information and programming language structure and semantics, in the form of phrasal concepts. In contrast to existing lexical approaches, we take a transformative step in automatic analysis for software engineering tools by accounting for *how words occur and semantically relate to one another in code*, rather than just counting their frequencies.

A model is useless without a concrete reference implementation. We have developed a set of heuristics to automatically extract SWUM from OO code written in C++ and Java. We evaluate the accuracy of the implementation instance of the model with a user study. In addition to evaluating the accuracy of these heuristics, we evaluate the model by using it in two concrete applications: method summarization and part of speech tagging. We further evaluate the model by showcasing its generality in other SE tasks.

In previous work [15], we represented phrasal concepts for verb phrases and noun phrases using textual phrases. In this paper, we generalize beyond a textual representation of phrasal concepts to a general model of phrase structure, with applications beyond query reformulation. Specifically, we provide details on the design, construction and evaluation of SWUM, that goes far beyond the brief overviews of SWUM that have appeared in previous papers [9], [12], [13].

In summary, this paper provides significant contribu-

- *E. Hill is with the Department of Mathematics and Computer Science, Drew University, Madison, NJ, USA. E-mail: emhill@drew.edu*
- *C. Newman is with the Department of Software Engineering, Rochester Institute of Technology, Rochester, NY, USA. E-mail: cnewman@se.rit.edu*
- *G. Sridhara is with IBM, Bangalore, India. Email: girisrid@in.ibm.com*
- *P. Francis is with Uservoice, Raleigh-Durham, NC, USA*
- *D. Shepherd is with Virginia Commonwealth University,Virginia, USA USA. E-mail: shepherdd@vcu.edu*
- *L. Pollock and K. Vijay-Shanker are with the Department of Computer and Information Sciences, University of Delaware, Newark, DE, USA. E-mail: {pollock, vijay}@cis.udel.edu*

This material is based upon work supported by the National Science Foundation Grant No. CCF-0702401 and CCF-0915803.

Method	LOC	Summaries	
butFlip_		Developer:	–
actionPerformed()	19	VSM:	color start end flip foreground
MegaMek 0.32.0		SWUM:	draw ruler to board view1
exportToSVG(String)		Developer:	Exports the plan objects to a given SVG file.
SweetHome3D 2.3	36	VSM:	svg export output file plan
		SWUM:	export plan component to svg
handleDoubleClick		Developer:	Hook method which can be overriden by subclasses to provide
(MouseEvent)			specialised behaviour in the event of a double click.
JHotDraw 7.0.8	39	VSM:	v handle pos outer figure
		SWUM:	`if handle is not null`
			`   track double click`
			`else`
			`   handle view to drawing and initialize p`
			`   if figure tool  is not null`
			`      handle mouse pressed, given mouse event`
			`   else`
			`       clear selection`
			`       add drawing view to selection`

TABLE 1
Example method summaries generated with lexical (VSM) versus phrasal (SWUM) concepts

tions beyond our previous work:

- a general software word usage model, SWUM, that captures the semantics of both the program structure and natural language in source code
- a set of heuristics to automatically extract SWUM from object-oriented code, along with an open source implementation to automatically extract SWUM from Java and C++ code
- an evaluation of SWUM's accuracy in identifying phrasal concepts in Java and C++ code examples
- an evaluation of SWUM's use in a concrete application: automatic method summarization
- further potential uses of SWUM to aid a wider variety of software engineering tasks.

This paper is organized as follows. Section 2 presents a motivating example where we explain the need for SWUM's ability to annotate phrase structure. Section 3 discusses phrasal concepts and how we build phrasal representations. Section 4 discusses three layers which, together, make up the core of SWUM. Section 5 describes how we build SWUM's three layers using a set of rules. In Section 6 we look at some of the current implementations of SWUM. We evaluate SWUM in Section 7, discuss related work in Section 8, and conclude in Section 9.

2 Motivating Example

Phrasal concepts (i.e., concepts expressed as a sequence of words) enable a rich source of textual information that software engineering (SE) tools can leverage to improve accuracy and communicate more effectively with a human developer. For example, consider the task of automatically generating comments that summarize methods [10], [11], [13], [12]. Few software projects adequately document code to reduce future maintenance costs [16], [17], despite the utility of comments for understanding software [18], [19], [20]. Thus, automatically generated method summaries can potentially help developers understand code more quickly.

Approaches to automatically generating method summaries have been proposed that use either lexical or phrasal concepts. Summaries using lexical concepts are generated by applying the Vector Space Model (VSM) to find words that occur more frequently in the method to be summarized than are commonly found in methods for that particular program [11]. The output is a list of words, typically 5 or 10, that are specific to the method.

In contrast, method summaries using phrasal concepts are generated by identifying important high-level actions taken by the method's implementation and then generating short phrases describing these actions [13]. SWUM's phrasal concepts are not only used in the heuristics to identify important actions, but are critical to generating readable phrases.

Table 1 gives 3 example method summaries that are automatically generated using lexical concepts (VSM) versus phrasal concepts (SWUM). Because the lexical model lacks any relationships between words, the VSM approach is limited to listing the n most highly relevant words. In contrast, SWUM's phrasal concepts can be used to construct phrases and sentence fragments that are easier to comprehend. This approach to method summarization would be impossible without a model of code that connects linguistic and program structure information, such as SWUM.

3 Representing Phrasal Concepts

A phrasal concept is analogous to a parse tree of an English phrase. English phrases are modeled by parse trees consisting of phrase structure nodes, such as verb phrases, noun phrases, or prepositional phrases. A *noun phrase* (NP) is a sequence of noun modifiers, such as nouns and adjectives, followed by a noun, and optionally followed by other modifiers or prepositional phrases [21].[1] A *verb phrase* (VP) is a verb followed by an NP, and does not usually include the subject of the verb. A *prepositional phrase* (PP) is a preposition plus an NP, and can be part of a VP or NP.

Figure 1 presents an example NP, VP, and VP with PP for three method name identifiers. The phrase structure nodes are NP, VP, and PP, while the other nodes indicate parts

1. In English, NPs are defined to begin with a determiner such as "a" or "the". Since determiners are rarely used in identifiers in software, we relax this requirement.

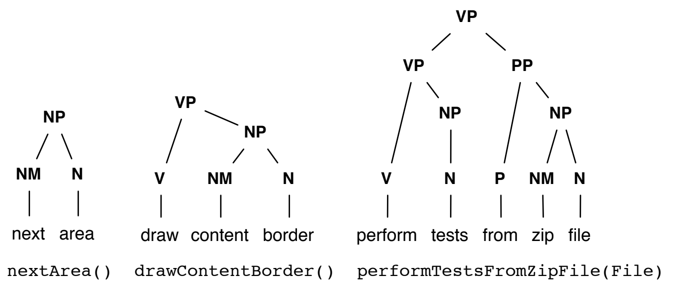


Fig. 1. Example English parse trees and phrase structure for 3 method names.

of speech. The leaf nodes are the individual words split from within the identifier. Each word in the identifier is assigned a part of speech, which can then be used to derive the identifier's phrase structure.

An important building block of phrase structure is part of speech (POS) information. Any word in a phrase can be assigned a part of speech, called a POS *tag*. A wide variety of possible POS tag sets exist for natural language [21]. For the purposes of identify phrasal concepts in source code, we simplify this variety into a much smaller subset:

- **Noun (N)** Nouns occur in the head position of a noun phrase (NP), where the *head* is the final word in the phrase. Head words are important because they typically represent the central theme of the NP [22]. For example, consider the identifier DataSourceChangeListener. Although every word in the identifier is a noun, the central concept of this NP is captured by the head word, "listener".
- **Noun Modifier (NM)** A noun modifier is a noun occurring to the left of the head position in an NP, or an adjective. For example, in DefaultEventIndex, both the adjective "default" and the noun "event" are noun modifiers.
- **Verb (V)** The verb label is used to annotate all verbs, regardless of form (e.g., root, third person, past participle, etc.)
- **Verb Modifier (VM)** Verb modifiers are adverbs that describe the nature of the action taking place, like "quickly", "completely", or "simply".
- **Verb Particle (VPR)** Verb particles are prepositions found in phrasal verbs like "set up", "look for", and "write down".
- **Preposition (P)** Prepositions such as "to", "in", "from", and "for" are labelled as P unless they appear as verb particles.
- **Conjunction (CJ)** Conjunctions connect multiple phrases together. Examples include "and", "or", "but", "if", and "unless".
- **Determiner (DT)** Determiners typically describe nouns. Common determiners include "the", "a", "any", and "all".
- **Pronoun (PR)** Pronouns like "my", "it", or "this" can occur in identifiers to further describe nouns.
- **Digit (D)** Sequences of digits are treated as words.

Figure 1 shows example POS tags above each identifier word. This set of POS tags has been sufficient for our current tasks of generating method summaries and searching source code. In future work, we anticipate adding more detailed tags, such as splitting noun modifiers into adjectives and noun-adjuncts, as they become necessary to solve additional software engineering challenges. From this phrase structure, semantic role information can be derived. (e.g., action, theme, and secondary arguments). The semantic roles give additional information about how the various phrasal concepts relate.

4 A Software Word Usage Model

Intuitively, SWUM models phrasal concepts that capture both the programming language (PL) and the natural language (NL) syntax and semantics found in source code. SWUM captures this information in three layered, interconnected subgraphs, as shown in Figure 2. At the top level we have the program structure (SWUM$_{structure}$), with traditional structural relationships such as invocation, def-use, etc. At the lowest level we have the context-independent word information (SWUM$_{words}$), similar to a bag of words representation. Linking these two representations together is our novel layer of phrasal concepts (SWUM$_{phrases}$), which capture both PL and NL semantics.

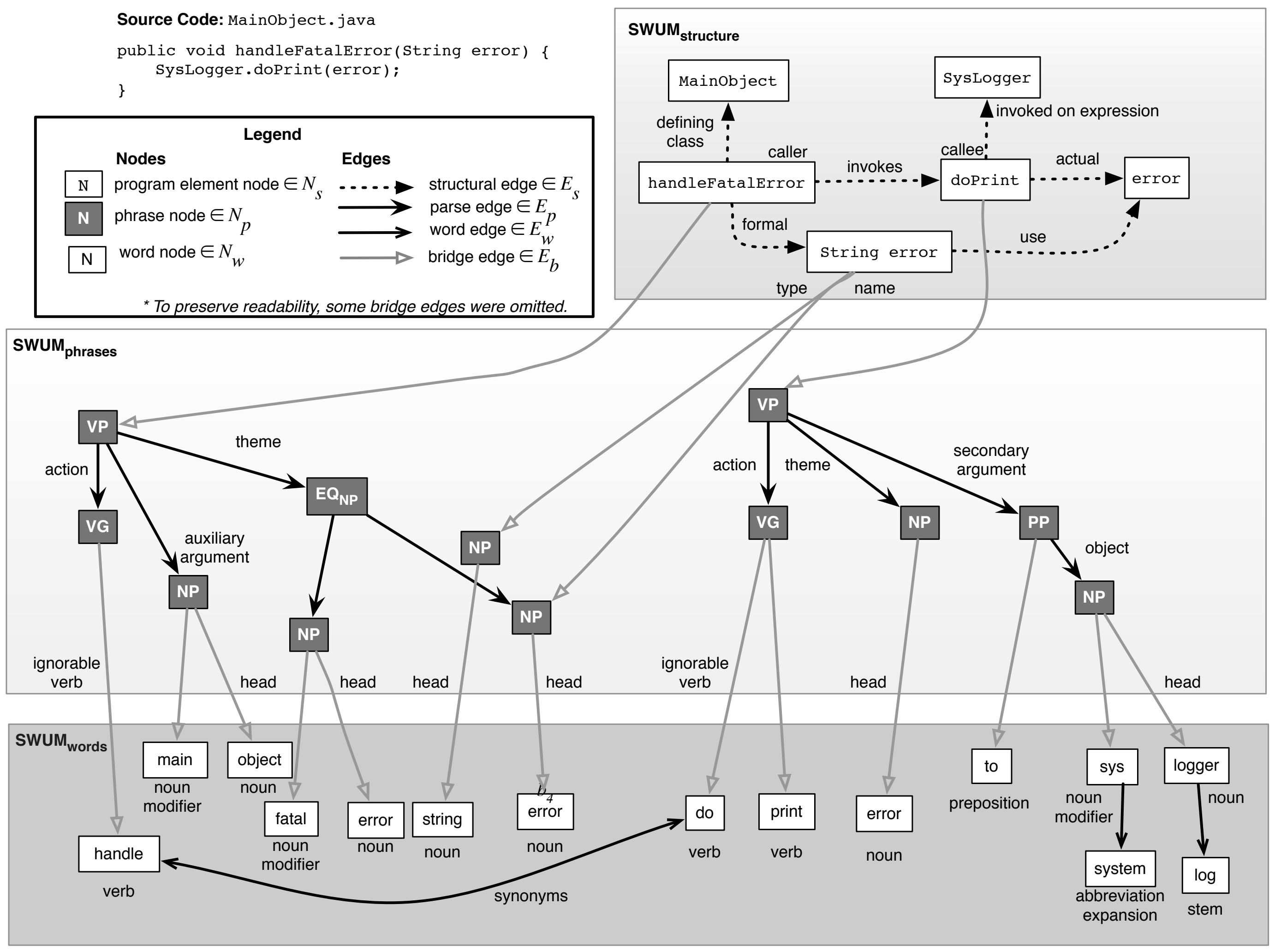
These layers are connected by bridge edges to facilitate combined analyses leveraging both NL and PL semantic information. Source code identifiers and declarations in SWUM$_{structure}$ are linked to their associated VP and NP phrasal concepts in the SWUM$_{phrases}$ layer. Similarly, phrase structure nodes in SWUM$_{phrases}$ are linked to their constituent words in the SWUM$_{words}$ layer. The following subsections detail the three layers of SWUM.

4.1 Modeling Program Structure with SWUM$_{structure}$

Each identifier declaration and use in the code segment being modeled becomes a program element node in SWUM$_{structure}$, and each structural relationship becomes an edge. For example in Figure 2, there is a method invocation edge from program element node handleFatalError to doPrint, a formal declaration edge from handleFatalError to String error, and a formal use edge from String error to error.

Nodes are annotated with structural role information, such as the "caller" and "callee" annotations on handleFatalError and doPrint, respectively. Edges are labelled by the kind of structural relationship such as "invokes", "actual", "formal", "defining class", and "call". In general, a label can be a control or data dependence, call edge, inheritance edge, AST edge, or a lightweight structural model relationship [23].

Although the example in Figure 2 includes nodes at the method declaration level, program element nodes can be created for an identifier at any level. Nodes can be created for classes, source files, or even packages, depending on the needs of the target software engineering tool utilizing SWUM. Program element nodes also may be annotated with higher-level semantic roles. For example, variable declarations may be annotated by usage pattern roles such as loop iterators or temporaries [24]. Alternately, field, method, and type declarations can be annotated by the role they play in the larger software system. For example, method declarations can be annotated by whether



Source Code: `MainObject.java`

```
public void handleFatalError(String error) {
    SysLogger.doPrint(error);
}
```

Fig. 2. An example 3-layer SWUM graph for the handleFatalError method.

they take action that changes program state, return useful data, or perform computations that do not change the program state [25], [26]. Similarly, fields can be annotated by whether they are used as a constant or global variable (e.g., COLOR_CYAN), contain a property of a class (e.g., ReportElement.transparent), or contain information that is part-of a class (e.g., HTMLTable.title). Type declarations might be annotated by the role they play in the overall system—whether they encapsulate a data object (e.g., TPSReport), encapsulate an action (e.g., ReportCompiler), play a role in an implemented design pattern (e.g., DocumentBuilderFactory), or other abstract class role [27], [28], [29].

4.2 Modeling Program Words with SWUM$_{words}$

There exists a word node in SWUM$_{words}$ for every word occurrence in the code sample. The word nodes are derived by splitting the identifiers into their constituent word tokens based on conservative camel casing or other identifier splitting techniques [30]. Word nodes can be annotated with lexical features, such as parts of speech, word meanings, relative frequency, or other lexical semantics [21]. Words from comments can also be included as word nodes.

Context-independent word relationships are used to create edges between word nodes in SWUM$_{words}$. For example,

$$
\begin{array}{rcl}
\mathrm{NP} & \rightarrow & \mathrm{NM}^*\ \mathrm{N} \\
\mathrm{PP} & \rightarrow & \mathrm{P}\ \mathrm{NP} \\
\mathrm{VG} & \rightarrow & (\mathrm{VI}|\mathrm{VM})^*\ (\mathrm{V}|\mathrm{VI})\ \mathrm{VPR}? \\
\mathrm{VP} & \rightarrow & (\mathrm{VG}\ |\ \mathrm{EQ}_{VG})\ (\mathrm{NP}\ |\ \mathrm{EQ}_{NP})\ \mathrm{PP}? \\
\mathrm{EQ} & \rightarrow & \mathrm{EQ}_{NP}\ |\ \mathrm{EQ}_{VG} \\
\mathrm{EQ}_{NP} & \rightarrow & \mathrm{NP}+ \\
\mathrm{EQ}_{VG} & \rightarrow & \mathrm{VG}+
\end{array}
$$

Fig. 3. Grammar for SWUM's phrase structure.

in Figure 2, the words "do" and "handle" are frequently used as synonyms in source code, represented by the word edge labeled "synonym". Additional word nodes *not* in the code may be added to create some of these relationships. For instance, the stem, or root form, of "logger" is not present in our code example. The word node "log" is added to SWUM$_{words}$ with a word edge labeled "stem". These word relationships are useful for applications such as code search.

4.3 Modeling Phrase Structure with SWUM$_{phrases}$

SWUM$_{phrases}$ contains the key components of SWUM used to represent phrasal concepts and capture the semantic knowledge of the programmer in terms of program structure and natural language syntax and semantics. To capture NL syntax and semantics in SWUM$_{phrases}$, we use phrase structure information inspired by natural language parsing techniques [21]. We have created additional phrase structure nodes to account for semantic information unique to source code.

The phrase structure information allows us to capture semantic role relationships like action, theme, and argument [21]. These semantic roles capture the higher level concepts of the code, and can be used to construct phrasal concepts. SWUM contains 6 phrase structure node types:

- **Noun Phrase (NP)** Noun phrases consist of a sequence of noun modifiers followed by a noun (NP $\rightarrow$ NM* N). A bridge edge exists from the NP to every word node in its phrase. For example, in Figure 2, the NP node representing the subphrase "Fatal Error" from the method name handleFatalError has a bridge edge to the word node for "fatal" labeled noun modifier and word node for "error" labeled noun.
- **Prepositional Phrase (PP)** A prepositional phrase consists of a preposition followed by an NP (PP $\rightarrow$ P NP). A PP has a bridge edge to its preposition as well as a parse edge to its NP object. In Figure 2, the VP for doPrint connects to the PP "to sys logger", which has a bridge edge to the word node representing "to" (inferred during SWUM construction and not explicitly in the code) and a parse edge to the NP node representing "sys logger".
- **Verb Group (VG)** We introduce verb groups to group action words together as one conceptual phrase unit. Verb groups can consist of a sequence of verbs, verb modifiers, and an optional verb particle (VG $\rightarrow$ (VI|VM)* (V|VI) VPR?). For example, in Figure 2, a VG node represents the verb group "do print", which together form the action of the method SysLogger.doPrint. A bridge edge maps the VG node to the word nodes of its constituent words, in this case, "do" and "print". We use verb groups to simplify the recursive VP parse structures common in more traditional NL parsing [21], which recursively define a VP $\rightarrow$ V (VP | NP). Verb groups provide a phrase structure parallel to NPs, which enables both VGs and NPs to be similarly processed by software engineering tools like code search.
- **Verb Phrase (VP)** Verb phrases consist of a verb group and its arguments such as the theme (i.e., direct object), secondary arguments (i.e., indirect objects), the subject, or other auxiliary arguments (VP $\rightarrow$ VG (NP | EQ) PP?). A parse edge $\in E_p$ exists from the VP to each of these semantic roles, if present. In Figure 2, both method names handleFatalError and doPrint map to VP nodes. Because method names frequently encapsulate actions, methods often map to VPs in SWUM. In Figure 2, the VP for sysLogger.doPrint maps to a verb group action, its theme, and the class as an auxiliary argument.
- **Equivalence (EQ)** Equivalence nodes capture relationships between noun phrases or verb groups that can be used interchangeably (EQ $\rightarrow$ NP+ | VG+). An equivalence node will either connect only NPs or only VGs. In Figure 2, the theme fatal error in handleFatalError further describes the method's formal parameter, String error. An equivalence node is used to semantically join the two noun phrases of these program elements together. For example, in Figure 2, the theme for handleFatalError is represented as an EQ$_{NP}$ between "fatal error" and "error".
- **Conjunctive Phrase (CP)** Conjunctive phrases unite two other phrase nodes via a conjunction, such as in lockAndInitHandle or rotateLeftOrRight. Solutions exist in the NLP community for handling many conjunctive phrase structures [31], [32], [33], and we leave handling conjunctive phrases for future work.

Figure 3 shows the grammar represented by the phrase structure nodes in SWUM$_{phrases}$.

4.4 Bridging SWUM layers

Bridge edges from program elements in SWUM$_{structure}$ to SWUM$_{phrases}$ phrase structure nodes can only have NP or VP nodes as their destination, since these are the topmost level phrase structure nodes in the parse trees we use. Program elements such as variable declarations with name and type information (i.e., formals, local variables, and fields), contain two bridge edges, one for the name phrase and one for the type phrase.

Bridge edges from phrase nodes to word nodes are labeled with information about the word occurrence that is related to its use in the phrase. For instance, a noun that is in the head position of a noun phrase will have the label "head" on the bridge edge leading from the phrase to that head noun.

5 Automatically Building SWUM$_{phrases}$

In this section, we describe our approach to automatically build SWUM$_{phrases}$ by extracting the necessary information from the source code. We leave discussion of automatically constructing the SWUM$_{structure}$ and SWUM$_{words}$ layers to the implementation section (Section 6), since the approach can significantly vary depending on the amount and type of information needed by the target software engineering problem. For instance, structural information derived from intra- versus inter-procedural program analysis, taking advantage of additional lexical semantic information such as synonyms or abbreviation expansion, etc.

We construct SWUM$_{phrases}$ using rules derived from naming conventions used across thousands of Java programs. While our analysis was performed on Java, we have since created an implementation of SWUM for C and C++, and our evaluation includes these languages. We first present our methodology for rule development, followed by our phrase structure construction algorithm, which includes our approach to part-of-speech tagging.

5.1 Developing SWUM$_{phrases}$ Construction Rules

If humans can consistently perform some task, such as annotating VPs and NPs in a source code identifier, then a heuristic set of rules can be derived to approximate it. The challenge is to derive this rule set in a timely manner with reasonable cost and acceptable accuracy. To develop our SWUM$_{phrases}$ construction rules we adapted traditional statistical learning and data mining techniques into a supervised exploratory data analysis approach by inserting a human developer (the first author) into the learning process. The benefit of this approach is that a human can learn and generalize rules with much less data than would be required by machine learning techniques, and can do both feature selection and rule learning simultaneously. The risk is the introduction of bias. Because SWUM$_{phrases}$ is the first attempt to extract phrasal concepts from source code identifiers, we consider this risk acceptable to demonstrate feasibility of the approach.

We followed a grounded theory methodology [34] to derive the rules. This started with a large set of identifiers. We analyzed method and field names from a set of over 9,000 open source Java programs. Starting from a random sample of 10,000 method signatures, we partitioned the data set and attempted to build a rule for each partition. Thus, a rule consists of a way to identify the partition where the rule should be applied as well as how to parse the names in that partition.

At each iteration of the learning process, we went through the following steps:

1) Create a hypothesis rule ($rule_h$) to analyze a partition (i.e., subset) of the data
2) Evaluate the accuracy of the current approach ($Rules \cup rule_h$) on the random sample by analyzing how the results differ from the prior iteration ($Rules$).
3) If the $Rules \cup rule_h$ approach is not yet accurate enough for the target application (e.g., method summarization), then the learner partitions the set into cases where the new approach worked correctly and where it did not. At this point the learner must create a new hypothesis rule ($rule_i$) by:
    a) Determining an automatic way to identify the correct and incorrect partitions
    b) Determining an approach to parse names in the incorrect partition
4) If the $Rules \cup rule_h$ approach is accurate enough for the target software engineering tasks (e.g., method summarization), or if the remaining mistakes appear to be poor examples to generalize from, add $rule_h$ to $Rules$ and analyze the next partition. If there are not any partitions left, the learner browses the entire set for mistakes.
5) Repeat until an acceptable level of accuracy is achieved or run out of time.

For example, we started with the assumption that all method names are in the same partition and that every method name begins with a verb. Upon applying this rule to our sample, we observed that some names start with a noun phrase or preposition. This led to two new rules in addition to starting with a verb: starting with a preposition and starting with an NP. Using the partitions instead of a pre-annotated data set allowed us to quickly view a much larger sample of the data set, by only analyzing differences between learning iterations. We continued refining our construction rules until we were satisfied with the level of accuracy for our current target applications of search and method summarization.

Our methodology drives our set of rules to analyze the "low-hanging fruit" of the analysis task. This approach is not scalable in general, as the number of rules and interactions between them can become too complicated for a human to reliably keep track of. Future work includes systematically refining the heuristic rule set with more traditional machine learning approaches [35].

5.2 Constructing SWUM$_{phrases}$

SWUM$_{phrases}$ consists of phrase structure nodes (e.g., NP and VP) and parse edges representing semantic roles of VPs such as the action and theme. The challenge in constructing SWUM$_{phrases}$ is in (1) accurately identifying the parts of speech of words in identifiers, and (2) identifying the action, theme, and secondary arguments for VPs.

To accurately identify the parts of speech of words and determine if a name should be classified as a VP or NP, we use two sources of information: the *lexical concept* of the word itself, and the *position* of the word in the PL syntax of the program as well as within the NL syntax of the identifier. For example, if the method name begins with a noun, determiner, or adjective (e.g, newStruct() or all_interfaces()) we assume the name is an NP. Similarly, if the name ends with a past tense verb (e.g., actionPerformed()), we also assume the name is an NP. In most other cases, we assume that the method name is a VP. Section 6 describes more details of how we implemented part-of-speech tagging.

The next step is to identify the action, theme, and secondary arguments for VPs. Because methods encapsulate actions in a program, we assume that all methods represent VPs, regardless of whether the actual name is an NP or VP. However, determining the action depends on whether the name is classified as an NP or VP; if the name is an NP, a default action is inferred, otherwise the action is the VP's verb. The theme and other arguments are inferred from the name, parameters, declaring class, and return type of the method signature.

Table 2 shows our rules for extracting phrase structure nodes. The input is a method declaration of split identifiers including the name, declaring class, return type, and formals. Prior to applying these rules, preamble is processed following Algorithm 1. Equivalent nodes are processed using Algorithm 2.

5.2.1 Tagging Method Names and Identifying the Action

5.2.1.1 Identifying Preamble: A developer may use a preamble of leading sequences of letters to differentiate different types of methods which take similar actions, or just prepend information describing the method's action. For example, consider the methods BrowserLauncher.ICStart() and Permute.gsl_permute_ushort(), with preambles "IC" and "GSL", respectively. We tag any such leading sequences of letters that appear to carry little content as *preamble*. Because our verb tagger assumes method names begin with a verb, tagging preamble allows our verb tagger to correctly identify the verbs in the method names (e.g., "start" and

TABLE 2
Rules for extracting the action, theme, secondary args, and auxiliary args for a given method. Input is a method declaration of split identifiers including the name, declaring class, return type, and formals. Preamble is assumed to be processed prior to this step following Algorithm 1. Equivalent nodes are processed using Algorithm 2.

Rule	Action	Theme	Secondary Args	Aux. Args
Base verb (default)	all leading verbs $\in \{V, VM, VPR, VI\}$	First non-empty phrase in: rest of name, first formal, class	N/A	Remaining formals, return type, & class
Base verb (preposition)	all leading verbs $\in \{V, VM, VPR, VI\}$	First non-empty phrase in: rest of name prior to preposition, class	Preposition + first unused non-empty phrase in: rest of name, first formal, class	Remaining formals, return type, & class
Base verb (checker)	Default base verb action becomes condition	Default base verb theme becomes condition	Subject = class	Remaining formals
General **Event handler** **Starts with preposition** (on, before, after) **Noun phrase** (void)	handle	Name	N/A	Formals, return type, & class
Starts with preposition (to, from) **Starts with preposition** (default)	convert –	–	Name	Formals, return type, & class
Noun phrase (non-void)	get	Name	N/A	Formals, return type, & class
Constructor	create, construct	Name	N/A	Formals
All preamble	–	Name	N/A	Formals, return type, & class

TABLE 3
Phrase structure examples extracted for each rule. Set notation ({}) indicates equivalence.

Rule	Example (class, return type, name, formals)	Action	Theme	Second. Args	Aux. Args
Base verb (default)	ThreadLocalConnection void commit(String name)	commit	string name		thread local connection
Base verb (preposition)	SerializeTools int parseStringForDateInformation(int information, String stringValue, Calendar cal, int fromI)	parse	{string value, string}	for {date information, int information}	calendar cal, int from I, int, serialize tools
Base verb (preposition)	RestrictionImpl MinCardinalityRestriction convertToMinCardinalityRestriction(int cardinality)	convert	restriction impl	to min cardinality restriction	int cardinality
NP (void)	Anonymous void onFailure(Throwable caught)	handle	on failure		throwable caught
NP (non-void)	BasicStatistics double factorial(int a, int b)	get	factorial		int a, int b, basic statistics
Start w/prep (on, before, after)	OPASplash void onDestroy()	handle	–	on destroy	OPA splash
Start w/prep (to, from)	TPM_KEY_PARMS void fromBytes(byte source, int offset)	convert	–	from bytes	byte source, int offset, TPM KEY PARMS
Start w/prep (default)	ArraysUtils Set asSet(Object a)	–	–	as set	object a, arrays utils
General	ProcessListener void errorOutput(ProcessEvent processEvent)	handle	error output		process event, process listener
Event handler	JoinAction void messageReceived(IRCMessageEvent e)	handle	message received		irc message event e, join action
Constructor	StoryTest(String name)	create, construct	story test		string name
All preamble	Parser boolean jj_3R_188()	–	jj 3 R 188		Parser
Base verb (preamble)	BrowserLauncher int ICStart(int instance, int signature)	start	int instance		int signature, browser launcher
Base verb (preamble)	LicenseEJB void ejbCreate()	create	license EJB		–

Algorithm 1 $parsePreamble(name)$

```
1: Input: split method name, $name = w_0, w_1, ..., w_n$,
   dictionary of two-letter words ($dictionary$),
   and two frequency lists from a corpus of method names, $onlyFreq$ and
   $firstFreq$
2: Output: Parsed preamble and name = $(preamble, name)$
3:
4: $preamble = ""$
5: $i = 0, i_{last} = -1$
6:
7: while $name \neq \emptyset \wedge i_{last} \neq i$ do
8:     $i_{last} = i$
9:     while $tag(w_i) = DIGIT$ do
10:        $i++$
11:    end while
12:    if $length(w_i) = 1 \vee$
       $(length(w_i) = 2 \wedge w_i \notin dictionary) \vee$
       $(length(w_i) < 5 \wedge !contains(w_i, get) \wedge !contains(w_i, set) \wedge$
       $!isPotentialVerb(w_i) \wedge onlyFreq(w_i) = 0 \wedge firstFreq > 10)$
       then
13:        $preamble = preamble + dequeue(name)$
14:        $i++$
15:    end if
16: end while
17: return $(preamble, name)$
```

"permute"). We do not discard the preamble, but assign it a special POS tag (PRE) and associate it with the action's VG to be conservative with respect to errors.

Our approach to identifying preamble is described in Algorithm 1. It takes as input the method name and a list of two-letter dictionary words likely to occur in software ($dictionary$). These words include "am", "my", and "on", but do not include words from other domains like "io", from greek mythology and astronomy, which is more commonly used as an abbreviation in code to refer to "input and output". We also include a list of positional word frequencies. In our training sample of over 9,000 open source Java programs, we counted the number of times a word occurred in the beginning, middle, or end of a method name, as well as the number of times the word occurred as the *only* word in the name. Our preamble function takes as input two of these frequency lists: $onlyFreq$ and $firstFreq$, respectively.

5.2.1.2 Special Method Names: Programmers follow different conventions in the way they name methods, based on the kinds of methods being named. The accuracy of SWUM is affected by automatically identifying different kinds of method names and parsing them accordingly. Here, we describe the main kinds of method names and how they are parsed for SWUM.

General names can include event-driven methods like actionPerformed(), keyPressed(), or Thread.run(). We detect these methods by looking for names like main or run, and by looking for past tense (end with -ed) or present participle (end with -ing) verbs at the end of the method name. In addition, methods with formal parameters of a type ending in "event" are frequently event handlers, as are method names beginning with the prepositions "on", "before", or "after". Lastly, NP method names that are void are assumed to be general (e.g., void method Handler.characters(String characters)). We assume a method name is an NP if it begins with a noun, determiner, pronoun, or adjective, and cannot be a verb. If the method name is an NP and has non-void return type, we assume the method is a getter.

In some cases, an entire method name may be labelled as preamble. For example, consider the automatically generated method Parser.jj_3R_188(). In such cases, we assume SWUM has made a mistake, and assume the name is the theme, marking the method as having a general name. This indicates that the method name probably has little to do with the method's actions, and enables SE tools using SWUM information to handle these methods differently.

Next, we detect whether the method is a boolean checker like isVisible or containsKey. If the method name begins with a verb in third person singular form, or is a modal verb like "can", "must", or "should", we assume the method is a boolean checker. Checkers are a special category of method names that do not conform to the action-theme VP model. Checkers need a subject that indicates what is to be checked (i.e., *what* is visible, *what* contains a key). Thus, checkers contain a subject, usually the class, and the method name becomes a VP condition mapping to the action and theme.

If the method name begins with a preposition, the action and theme are unknown and we treat the name as a secondary argument. If the preposition follows a known naming convention, like "from" or "to", we can infer an action. For example, if the method starts with a common preposition like "to", as in toString(), we infer the verb group (VG) "convert". In contrast, for less commonly used prepositions, like inSectorNorthWest(Vector), we leave the VG empty.

5.2.2 Identifying VP Theme and Arguments

If we have not identified the method name as any other type, we assume the name starts with a verb in base form. The first step in parsing a base verb name is to identify the verb group. In addition to the verb, the name may contain verb modifiers, verb particles, and additional verbs in the form of ignorable verbs. For example, the method name testMakeProductPrice() contains two verbs in its VG: "test" (VI) and "make" (V). Depending on the SE tool, "test" may or may not be "ignorable", but tagging "test" as VI enables us to identify another verb, "make". If we were not able to identify ignorable verbs, we would only tag "test" as the verb, and "make" would be improperly tagged as a noun modifier (NM) in the theme.

To determine if the first word in the method name is an ignorable verb (VI) followed by an actual verb (V), we utilize the positional frequency lists we first used to tag preamble, described in Section 5.2.1.1. If the second word in the name is more likely to occur as the first or only word in a name rather than occur in the middle or end of a name, we assume the first word in the method name is VI.

Once we have identified the verb group, we identify the theme, secondary arguments, and auxiliary arguments. The first step is to check if the name contains a preposition and populate the auxiliary arguments from the formals. For most names we add any formals of non-boolean type to the auxiliary args. Boolean-typed variables are typically used as flags controlling the flow of execution, and are unlikely to be the theme or secondary argument. The exceptions are names beginning with verbs that can take boolean arguments, like "check", "assert", "contains", "add", "print", "append", "push", or "set". For these boolean-argument verbs, all formal parameters are added to the auxiliary args.

If the method name does not contain a preposition, we look for the theme in the rest of the name, the auxiliary args, or the class. If the theme is in the name, we also look for equivalent formal parameters in the auxiliary args, presented in Algorithm 2. We consider an NP auxiliary arg to be equivalent to the NP theme if their head parameters are the same, unless the auxiliary argument NP ends with an ignorable noun, in which case we check the penultimate word in the NP for head overlap.

If the method name contains a preposition, we look for a secondary argument in addition to the theme. Similar to our phrase generation approach for query reformulation [15], we look for the theme first in the name, and then in the class. We look for the secondary argument in the name, and then in the auxiliary args. If the theme or secondary argument are in the name, we check these arguments for overlap in the auxiliary args.

Algorithm 2 $checkHeadOverlap(theme, auxArgs)$

```
1: Input: set of phrase structure nodes ($theme, auxArgs$)
2: Output: possibly modified phrase structure nodes ($theme, auxArgs$)
3:
4: for all $arg \in auxArgs$ do
5:     if $head(theme) = head(arg) \vee$
       $(tag(head(arg)) = NI \wedge head(theme) = head_{-1}(arg))$ then
6:         $theme = EQ(theme, arg)$
7:         $remove(arg, auxArgs)$
8:     end if
9: end for
10: return ($theme, auxArgs$)
```

5.2.3 Beyond Method Declarations

To build a SWUM for a method invocation, we start with the SWUM from its declaration. Based on the argument position of the formal parameters, the actual parameters and invoked-on-expression are added to the model as EQ nodes with the formal parameters and declaring class, respectively.

At some method call sites, multiple method invocations may be chained together as composed and nested method invocations. We consider a *composed* method invocation to be a sequence of method calls where each method call is invoked on the previous method invocation expression. For example, from the composed invocation pServer.getTagName.equals("server"), we can infer that "equals" compares a server with a tag name. We consider a *nested* method invocation to be a sequence of method calls where a method call uses one or more method invocation expressions as actual parameters. For example, from the nested invocation server.addItem(server.getURLFromString(id)), we can infer that this particular "add item" invocation takes a "url" as an argument. In addition to being linked in the program structure, composed and nested method invocations are linked together in SWUM via their phrase structure nodes, since the arguments of one method become the arguments of another. In the example above, the theme for getURLFromString(id) would become one of the auxiliary arguments to the VP for addItem.

Our construction algorithm for SWUM currently supports variables, types, method and field signatures, and method invocations. We support logical and arithmetic expressions by creating an EQ node linking to the NPs for every variable used in the expression. Assignment statements are modeled with a bridge edge to the SWUM of the left hand side, and a separate bridge edge to the right hand side. Support for other language constructs, such as conditional statements and loops, is the subject of continuing research [13].

Additional nodes and edges may be added to capture return information. For example, if the method is a getter, its return type is considered equivalent to its theme. Within a method declaration, SWUMs for any return statements are linked with an EQ node to the return type. This return information can provide valuable insights into the actual types of variables and how general methods are used in the source code. For example, consider the nested invocation:

JConfig.getCanonicalFile(JConfig.queryConfiguration("savefile"))

Based on its declaration, we know that queryConfiguration has a string return type and returns a variable retVal (not shown). We also know from the declaration of getCanonicalFile that the name of its formal parameter is "fname". Thus, since the invocation of queryConfiguration is nested inside the call to getCanonicalFile, we can also infer that this particular invocation of queryConfiguration returns the concept of an "fname". For this example, queryConfiguration would contain three equivalent arguments for the return type in $auxArgs$: the declared return type, "string"; the returned expression, "ret val"; and the inferred return, "fname".

5.3 Developing SWUM Construction Rules for other Programming Languages

Although the rules were developed specifically for Java, we have found that they apply well to other object-oriented languages such as C++, as will be discussed in Section 7. Nevertheless, the performance of the rules may be further improved by small tweaks for differences in language semantics and usage conventions. For example, method names more frequently contain prefixes in C++ than in Java, and the rules may need adjustments to better handle these cases. We have also found that the existing rules work well for C, but other non-object-oriented languages may require their own custom rules.

For other types of languages, the naming convention rules may be quite different. The overall theory of SWUM and what nodes need to be constructed to capture phrasal concepts remains the same, but the construction rules may require significant adjustments. New rules can be constructed based on the rule development methodology outlined in Section 5.1. Depending on the level of accuracy required by the target software engineering tool, this can be a relatively straightforward process for someone proficient and experienced in the language. Language proficiency can be bootstrapped by studying many examples from other programmers.

6 Current SWUM Implementations

There are currently two implementations of SWUM, one targeting Java and one targeting C and C++, implemented with different underlying technologies. The SWUM implementation for Java is an Eclipse plug-in, taking advantage of Eclipse's program structure information and AST for Java source code. The C++ implementation preprocesses source code using srcML [36] and the SrcML.NET framework [37] (https://github.com/abb-iss/SrcML.NET) for program structure information.

Implementation details of statement-level SWUM analysis within method bodies are left to the target software engineering applications, such as method summarization [13]. Because statement-level information is more expensive to calculate, depending on the required level of precision, we also leave optimizing access to that information up to the target applications.

6.1 Program Structure (SWUM$_{structure}$)

Algorithms for constructing program structure models (SWUM$_{structure}$) are well documented [38], [39]. The Java version of SWUM utilizes Eclipse's call hierarchy for calling relationships, and relies on Eclipse's program element reference and declaration search mechanisms. Additional structural information, such as whether a method writes or reads a field, are detected with light-weight lexical approximations using regular expressions or by traversing the AST. Although not used in modeling methods, SWUM utilizes calling relationships, field def-use information, and declaration-use edges for both variables (local, formal, field) and methods in support of target applications, such as comment generation [13].

6.2 Program Words (SWUM$_{words}$)

The overall accuracy of SWUM depends on being able to accurately extract word nodes from identifiers and comments. In most cases, simply splitting on camel case words (e.g., ASTVisitor, getInfoByName) or non-alphabetic characters (e.g., model_type, object2byte) will suffice. This straight-forward splitting has been successfully employed by tools which utilize lexical concepts [4], [1], [15], [40]. However, sometimes there is no discernible delineation between words, especially in cases where two words together indicate a particular concept (e.g., notype, USERLIB). In such cases, alternative word splitting techniques can be employed [41], [42]. In addition, word node extraction is complicated by the use of abbreviations, which can be expanded as necessary to properly extract lexical features [43]. Synonym support is not currently implemented, since accurate synonym information for the domain of software does not yet exist [44]. This is an area for future work.

Part of speech (POS) information for individual words is crucial to extracting accurate phrasal concepts. Although POS taggers for written English texts boast accuracy of 95-97% [21], the problem of identifying the correct POS for a given word in software is different from English. In software, sentences are often written in imperative form and nouns are frequently used as verbs. Consider the common English noun "fire", which is frequently used as a verb in software (e.g., "fire action event"), but not always. The word "fire" may be a noun (e.g., "fire started") or even an adjective in gaming software (e.g., "car is in a fire state"). Further, existing taggers perform poorly on unknown vocabulary [21]. This is a particular problem for software, where terminology frequently evolves as new technologies are developed and new domains encountered.

To address this problem, we use an alternative to traditional English POS taggers: the morphological parser PC-KIMMO [45], which uses the structure of the word itself to suggest potential POS tags. For example, words ending in "-ed" are often verb past participles. Thus, PC-KIMMO can make informed POS guesses even for new vocabulary. The drawback is that PC-KIMMO outputs *every possible* POS—the final decision is left to the user. We use standard naming conventions [46], [25] (e.g., the fact that method names frequently begin with verbs, and variables are NPs) to disambiguate the POS for a given word node.

We also used PC-KIMMO to create a word stemmer specific to the domain of software [47]. We select between possible parses by using word frequency information from a corpus of over 9,000 open source Java programs. Using this technique, we identified stems for 21,329 words found in the Java corpus.

6.3 Practical Analysis of Time and Space

Our unoptimized research prototype for Java, run on a 2.7 GHz Intel Core i7 with 16 GB of RAM, takes just 2.8 seconds to analyze signatures from a 74K LOC program (containing 3,462 fields and 4,712 methods) and analyzed 1.5 million LOC in just under 90 seconds (64,694 fields and 130,363 methods). On a more modest machine with 4 GB of RAM and a 3GHz Intel Pentium D830 Smithfield, the Java implementation took 11.3 seconds to analyze signatures from a 74K LOC program and analyzed 1.5 million LOC in just under 11 min.

The C++ implementation was run on a 2.67 GHz Intel Core i5 with 6 GB of RAM. Including the time for srcML to process the files, our implementation for C++ took 26.1 seconds to analyze signatures from a 70K LOC program (containing 2,593 fields and 1,258 methods) and roughly 9 minutes to analyze a 2.1 million LOC program (containing 48,831 fields and 76,681 methods). If the srcML is already present, this drops to just 7.5 seconds to analyze the 70K LOC program and 2 minutes for the 2.1 million LOC program.

In terms of memory space, the Java implementation used 152 MB of RAM to store the SWUM representation for all the program elements (method and field declarations) for the 74K LOC program, and 470 MB for the 1.5 million LOC program. The C++ implementation used 68 MB for the 70K LOC program and 394 MB for the 2.1 million LOC program.

Analysis of body statements as well as signatures will require additional time, and will be heavily dependent on the speed of the structural analysis. Although we have presented times for analyzing an entire program, SWUM analysis can be run on-demand. Demand-driven construction is especially useful when more expensive analyses are required, as for comment generation [13].

7 Evaluation

We evaluate SWUM's accuracy in two ways. First, by asking programmers to annotate a verb (V), direct object (DO), and optionally an indirect object (IO) for a set of methods. By comparing the syntactic roles identified by humans with the semantic roles identified by the SWUM tool, we can judge SWUM's overall accuracy at identifying phrasal concepts. The second way we evaluate SWUM's accuracy is by analyzing its effectiveness at annotating words with their correct part of speech tag. Finally, we end our evaluation by exploring the generalizability of SWUM's model to natural language tasks for software engineering. To do this, we look at how SWUM has been applied in prior research to improve code search, summarization, query completion, and more. We answer the following research questions.

RQ1: How accurate are the SWUM extraction heuristics for OO code written in Java and C++?
RQ2: How effective is the model for the concrete application of method summarization?
RQ3: How generalizable is the model in applying to other software engineering tasks?

We explore each of these research questions in the following subsections.

7.1 RQ1: SWUM Accuracy

We will start with its accuracy on phrasal concepts. To elicit as accurate annotations as possible, we gave our participants access to method bodies and comments. The downside is that inaccurate results could be due to mistakes in SWUM's construction *or* because the method is poorly named and the participants' use different information based on the method's body and comments. When analyzing our results, we have attempted to separate these two sources of errors.

7.1.1 Phrasal Concept Accuracy Study Design

We asked 12 human annotators (6 Java programmers and 6 C++ programmers) to provide the verb, direct object, and optional indirect objects for 150 methods (75 Java, 75 C++), and compared these with the equivalent semantic roles (action, theme, secondary argument) identified by SWUM.

We randomly selected 75 method declarations from 5 open source Java programs: the Java 1.5 API implementation; Vuze, a bit torrent client; Freemind, a mind map application; Gantt, an open source interface for using Gantt Charts; and HSQLDB, a relational database engine. In addition, we randomly selected 75 method declarations from 5 open source C++ programs: Classic Shell, which provides extended interface functionality for Windows 7 and Vista; Ophcrack, a Windows password cracker; VLC, an open source media player; FileZilla, a cross-platform GUI FTP client; and Inkscape, a vector graphics editor.

Subjects were either given all Java or all C++ examples, depending on their prior programming language experience. We grouped the examples for each program into 3 blocks of 5 methods each. Annotators were randomly assigned to blocks for their programming language such that each participant had 1 block from each program in the language, every block was annotated by 2 participants, and no more than 2 blocks were annotated by the same pair of participants. We collected a total of 300 (150 Vs and 150 DOs) annotations for 150 methods, for a total of 900 individual annotations including verb, direct object, and optional indirect object information.

To evaluate accuracy, we need a measure of agreement when comparing SWUM's output to the human subjects'. We define two phrases to be the same if they match exactly, have the same head word, have almost identical verbs (e.g., calc and calculate, get and return), or contain the same words and clearly refer to the same concept (e.g., "leave attribute of menu" vs. "leave attribute of menu structure").

7.1.2 Preliminary Data Characteristics

In general, the C++ annotators had less agreement with SWUM than the Java annotators. When calculating agreement for verbs (Vs) and direct objects (DOs) for Java, 82% of the matches were exact. In contrast for C++, just 60% of the V and DO matches were exact out of 300 observations. In general, the Java annotators were in agreement for 88% of verbs and direct objects, exactly matching 60%, whereas the C++ annotators only agreed on 79% of Vs and DOs, exactly matching 43%. Java annotators agreed 60% of the time as to whether the IO was *necessary* as a secondary argument in the phrasal concept, however, they did not agree on the IO *description* for 58% of the methods. Following the same trend, the C++ annotators agreed 45% of the time as to whether the IO was necessary as a secondary argument, but disagreed on the IO for 65% of the methods.

7.1.3 Results and Discussion

To calculate SWUM's accuracy, we take the manually annotated data described above and compare SWUM's automatically-generated annotations using only the manual annotations where there was agreement between annotators. We begin by comparing SWUM's Vs and DOs to each subject's annotations independently, finding that SWUM's overall accuracy is 77% for Java and 73% for C++. However, for java, there are 28/75 methods for which the subjects did not agree, and 23/75 where the subjects used information from outside the method signature (with overlap). Meaning that even human annotators required additional, external information in all but 5 situations.

We focus our analysis on inaccuracies due to SWUM's construction rules, rather than ambiguity in subject agreement or method naming. Given this criteria, we observed 2 dominant situations for Java and 1 dominant situation for C++. For Java, SWUM had trouble identifying the DO for 5 (out of 150) methods . For example, both the DO and IO are found in the parameters for insertBefore(Node newChild, Node refChild), but SWUM assumed the DO was in the class name. Second, 5 (out of 150) methods were parsed incorrectly either during the identifier splitting or POS tagging phases. For example, in jumpLeft, 'left' is incorrectly tagged as a past participle verb, when it is in fact a noun modifier. Even more challenging to parse is the method cellPaint, where the DO precedes the verb.

For C++, SWUM made 4 (out of 150) verb mistakes. Three of the verb mistakes were in methods where the last word in the name is a verb, and were parsed as noun phrases and ending with past participle verbs. In both of these situations, a default verb, "handle" is used, which was not selected by the annotators. For example, for avoid_conn_transformed, both annotators agreed that the verb is avoid, not handle, and similarly for sp_metadata_update, both annotators agreed the verb is update. In the final mistake, fsm_reset_bforce, SWUM' s POS tagger was unable to identify "fsm" as preamble before the true verb, reset. In both the Java and C++ cases, more accurate POS tagging could minimize these errors [48].As with the Java examples, more accurate POS tagging could minimize these errors [48].

SWUM conservatively identifies IOs only when the method name contains a preposition, leaving us with 4/75 examples in Java and 5/75 examples in C++; there are very few examples from which to draw conclusions about IO accuracy. For C++, SWUM agreed with at least 1 annotator for all 5 examples, and both annotators for 3 of them. In Java, SWUM agreed with the subjects for 3 of the examples. For the last example, SWUM's poor IO identification was the result of the DO mistake in insertBefore.

7.1.4 A Second Look at Phrasal Concept Accuracy

In the prior study, we observed that pairs of programmers did not agree on the IO for 62% of the methods. Given this variability, we hypothesize that there may be many equally acceptable IOs. To investigate this question, we performed an additional evaluation. We used SWUM's phrasal concepts to automatically generate phrases containing a V, DO, and IO. If SWUM did not identify a secondary argument to be used as an IO, we selected the most likely candidate from the list of auxiliary arguments. We then generated phrases for an additional set of 20 randomly selected Java methods, and asked 4 human annotators familiar with Java to rate the correctness of each phrase. If a phrase was incorrect in any way, we asked the annotators to present an alternate phrase.

Overall, we found that all 4 of our subjects agreed with SWUM's generated phrases for 11/20 methods. When we break down the phrases into V, DO, and IO (yielding 240 individual annotations, 60 per subject), we found SWUM's accuracy to be 88% overall, 84% for verbs, 95% for DOs, and 88% for IOs. As with our initial study, disagreement between SWUM and the subjects could be due to inaccuracies in SWUM's extraction rules or poor method naming. When we remove examples with poor names, SWUM's accuracy becomes 93% for verbs, 97% for DOs, and 93% for IOs. Methods with poor naming could in future be identified and renamed by automatic techniques [49], [50]. As with the previous study, we found inaccuracies were due to poor identification of arguments (DOs and IOs) and poor parsing.

Given their greater accuracy and the fact that we doubled the number of annotators for each method, we believe these results demonstrate that even though human annotators may disagree in *constructing* phrasal concepts, they have much higher agreement in *recognizing* them.

7.1.5 Part of Speech Accuracy Study Design

In addition to studying SWUM's accuracy with phrasal concepts, we also study its accuracy with respect to annotating words with their appropriate part of speech. We collected 267 function identifiers (confidence level 95%, confidence interval 6) from a set of 428k identifiers selected from 20 open source systems. These identifiers were first split using the heuristic_split algorithm found in the spiral [51] python library and then manually examined to correct any poor splitting choices. One person annotated each split identifier with a part of speech tag for each word within the identifier and a second person reviewed each annotation and either agreed or disagreed with the original annotator.

After agreeing on the correct annotation for each identifier, all split 267 identifiers were fed to three part of speech taggers: SWUM, Stanford [52], and Posse [53]. Posse is a part of speech tagger specifically designed to tag source code identifiers and is a state-of-the-art technique for part of speech tagging in this domain. Stanford is a part of speech tagger designed for general part of speech tagging of natural language texts. Thus, we compare to a state-of-the-art technique for our domain (software) and a state-of-the-art technique for general natural language text.

We automatically compared the output from each tagger to the set of annotations manually supplied by humans to calculate how much agreement there was between the two sets. This comparison was done by scanning the output the humans and tools in order (i.e., left to right) and checking to see if they annotated each word in a given identifier the same way.

7.1.6 Part of Speech Accuracy Study Results

The results of our study are shown in Table 4. SWUM has the highest accuracy (i.e., compare to the human annotations) in the following categories: Noun Modifier (NM, 85.16%), Preamble (PRE, 12.50%), and Determiner (DT, 100%). SWUM and Stanford annotated the same number of Digits (D, 85.71%) correctly compared to the human annotations. SWUM had the second-highest agreement in the following categories: Verb (V) and Preposition (P). In these cases, SWUM and Posse had very similar accuracy while Stanford had much higher accuracy on prepositions (92.86%) and somewhat higher accuracy on verbs (79.74%).

SWUM was least accurate (compared to other techniques) on Noun (N), Plural Noun (NPL), Verb Modifier (VM), and Conjunction (CJ). SWUM was able to correctly annotated 74.47% of nouns, but both Posse and Stanford were more accurate at 80.85% and 91.05% respectively. All taggers had a categories which they were unable to annotate at all. For SWUM and Posse, this was: noun plural, verb modifier, and conjunction. For Posse and Stanford: preamble. For Posse alone: digit.

Our results highlight a couple things. First, SWUM has high accuracy on the most common tag in our set: Noun Modifiers; achieving 85% agreement with the human annotations. Additionally, while SWUM is still not highly accurate on Preamble annotations, it does detect them more effectively than the other two part of speech taggers. Second, our results highlight the complementarity of SWUM with other part of speech techniques. That is, SWUM is strong where other approaches are not and vice versa; SWUM can combine with other techniques to increase overall accuracy. This has positive implications for future work in part of speech tagging and customizing natural language techniques to the SE domain; a problem in need of much more attention [54].

7.1.7 Threats to validity

In our study, we selected examples from a variety of Java and C++ programs from different domains, evaluating SWUM's accuracy on 170 unique methods. However, our results may not generalize to all Java/C++ programs or methods written in other languages. In an attempt to elicit as accurate annotations as possible by giving subjects access to method bodies and comments, we introduced a threat to internal validity since incorrect results are not only caused by SWUM's inaccuracies but also by poorly named methods. We have attempted to minimize this threat by reporting on the accuracy of our results as a whole, as well as for instances where incorrectness does not appear to stem from

TABLE 4
Frequency of Per-tag Agreement between Human Annotations and Tool Annotations

Part of Speech	Human Annotations	SWUM	% SWUM Agreement w/Humans	Posse	% Posse Agreement w/Humans	Stanford	% Stanford Agreement w/ Humans
NM	256	218	**85.16%**	63	24.61%	33	12.89%
N	235	175	74.47%	190	80.85%	214	**91.06%**
V	232	172	74.14%	165	71.12%	185	**79.74%**
NPL	49	0	0.00%	0	0.00%	40	**81.63%**
P	42	28	66.67%	27	64.29%	39	**92.86%**
PRE	24	3	**12.50%**	0	0.00%	0	0.00%
VM	9	0	0.00%	0	0.00%	5	**55.56%**
CJ	7	0	0.00%	0	0.00%	2	**28.57%**
D	7	6	**85.71%**	0	0.00%	6	**85.71%**
DT	5	5	**100.00%**	1	20.00%	2	40.00%

SWUM's construction rules. Finally, determining agreement between semantic roles has the potential to introduce bias. We have tried to minimize this threat as much as possible by using rules that can be replicated and by presenting how infrequently they apply to our data set.

7.2 RQ2: SWUM Usage

We now highlight some of the usages of SWUM's novel phrasal concepts layer to help the reader understand the need for this higher-order natural language information. SWUM has been used in the development of a novel technique to automatically summarize Java methods [13], [12]. Given the signature and body of a method, our automatic comment generator identifies the content for the summary and generates natural language text that summarizes the method's overall actions.

There are three main components to automatically generating method summary comments: (1) selecting the content to be included in the summary comment, (2) lexicalizing and generating the natural language text to express the content, and, (3) combining and smoothing the generated text. For content selection, SWUM is used to exploit the structural relationships based on the linguistic information of the theme and secondary arguments. For example, one kind of statement in the method body that is considered to be important content for the summary is a statement that contains a method call with the same action as the method being analyzed. For text generation, the semantics captured by action-theme relationships in SWUM are used in conjunction with the actual words to generate intelligible phrases that accurately represent the code. Lexicalization uses SWUM to construct noun phrases that represent variables, using both type and variable name information. Text generation is improved by identifying theme equivalences, which occur when the theme in a method name overlaps its parameters.

Below we evaluate the use of SWUM in method summarization. With no existing automatic comment generators for comparison, 13 programmers were asked to judge the generated comments [13]. Figures 4 and 5 show these results. For clarity, an s_unit is the same as a Java statement, except when the statement is a control flow statement; then, the s unit is the control flow expression with one of the if, while, for or switch keywords.

Figure 4 shows the Accuracy, Content Adequacy, and Conciseness of individual phrases constructed using s_units. In terms of **accuracy**, of the 144 accuracy judgements over all 48 s_units, 127 rated the generated comment accurate with only 1 judgement giving a rating of very inaccurate. For **Content Adequacy**, 138 of 144 judgements the comments were not missing any important information for understanding, with 110 of those indicating no missing information. More significantly, there were no comments for which there was a majority opinion that important information was missing for understanding. For **Coniseness**, There was no s unit where a majority of evaluators said the comment had too much unnecessary information. Instead, 46 of the comments were ranked by majority opinion as having no unnecessary information with only 2 of the 48 comments with the majority of evaluators saying it had some unnecessary information.

Figure 5 shows the Accuracy, Content Adequacy, and Conciseness of method summaries, constructed by taking s_units and compiling them together to summarize the method. In terms of **Accuracy**, Of the 8 methods, 7 had accurate summaries according to the majority of evaluators per method. Not one among the 24 individual responses suggested that we generated ”very inaccurate” text. Only in 1 of the 8 methods did a majority believe the summary text was a ”little inaccurate”. For **Content Adequacy**, 6 of the 8 summaries were judged to not have missed important information by the majority. In fact, 17 of the 24 individual responses suggest that the generated summaries did not miss important information. For **Conciseness**, There was only 1/8 methods for which a majority of the evaluators felt that the generated summary was very verbose. 13 of the 24 individual responses suggest that the generated summary was concise.

The results from this study suggest that the SWUM-based summary comment generation technique can generate accurate text descriptions while tolerating some missing, but relatively unimportant information. The reason this summarization technique can generate high-quality summaries is that it uses SWUM to identify the *action*, *theme*, and *secondary arguments* for any given method; each of which is critical for understanding how a method behaves and what entities are associated with this behavior.

7.3 RQ3: SWUM Applications

7.3.1 Code Search

SWUM phrase structure can be used to improve source code search. Given a query, the relevance of a program element can be weighted by what semantic role the word appears in. Specifically, a word appearing in the action or theme roles would be more relevant to the query than if the word appeared as a secondary or auxiliary argument. Likewise, a word's proximity to the head position of a VG or NP could be taken into account. Relevance scores could be damped based on how far away a noun is from the head position of the NP, with ignorable V and N tags not causing dampening.

We evaluated two variations of this SWUM-based search approach and compared its effectiveness with 3 state of the art search techniques on 8 search tasks [9]. The two SWUM-based approaches only varied in their thresholds for determining relevance: SWUM10 considers the top 10 results as relevant, and SWUMT uses the mean relevance score of the top 20 results as a threshold. We ranked the approaches from 1–5 based on their overall effectiveness for each concern in terms of F Measure, with 1 representing the most effective technique. SWUMT is the most highly ranked technique with an average rank of 2.38 and a standard deviation (std) of 1.18. Its closest competitor, a search based on Google Desktop Search (GES) [55], has an average of 2.75 (std 1.19), SWUM10 an average of 2.88 (std 1.64), AOIG-based FindConcept [40] an average of 3.00 (std 0.93), and a regular-expression search similar to grep has an average of 3.50 (std 1.41). From these results we can see that although SWUMT and GES are the best overall search techniques in the study [9], SWUMT is consistently ranked more highly.

Accuracy			**Content Adequacy**			**Conciseness**		
Response Category	**S**	**R**	**Response Category**	**S**	**R**	**Response Category**	**S**	**R**
Accurate	44	127	Adequate	41	110	Concise	46	128
Slightly Inaccurate	4	16	Misses Some	7	28	Slightly Verbose	2	15
Very Inaccurate	0	1	Misses Important	0	6	Very Verbose	0	1

Fig. 4. Human judgements of individual phrases. S = Majority opinion (# s_units). R = # Responses.

7.3.2 Query Completion

In addition, SWUM has been leveraged to support query completion in the Sando search tool for Visual Studio [56]. Sando is an open-source code search tool based on information retrieval search technology. Sometimes, a given query word will result in a large number of potential matches, which are ranked based on the IR technique. To help the user focus their search on ranking higher the locations that are more relevant to their intended search, based on words used in the code, the SWUM representation of the application can be used to suggest more query terms.

Accuracy			**Content Adequacy**			**Conciseness**		
Response Category	**M**	**R**	**Response Category**	**M**	**R**	**Response Category**	**M**	**R**
Accurate	7	18	Adequate	4	11	Concise	3	13
Slightly Inaccurate	1	6	Misses Some	2	6	Slightly Verbose	4	6
Very Inaccurate	0	0	Misses Important	2	7	Very Verbose	1	5

Fig. 5. Human judgements of method summaries. M = Majority opinion (# methods). R = # Responses.

To suggest new terms, the query completion reports back all corresponding actions and themes associated with the original query words. If the term occurs as an action node in SWUM, then its theme is returned with the action as a new set of query terms. Similarly, if the query term occurs as a theme node in SWUM, its corresponding action is reported along with the original term as a set of query terms. While both contextual search and this query completion use SWUM, they differ in what is being accomplished for the user. This query completion is built as part of the query reformulation on top of any existing search and presentation of search results, and suggests additional terms, with a ranked list of suggested sets of query terms. The user can choose a new set of query words from the ranked list of query term sets. Query completion has not yet been systematically evaluated, although early users report that the results are helpful.

7.3.3 Additional Potential Uses of SWUM

Beyond the case studies, we believe SWUM can be used to create improved human-oriented software models and tools. Here, we mention a few others.

Program Comprehension. Because of its emphasis on representing both the programming language and natural language semantics found in software, SWUM is a natural fit for program comprehension tools. In addition to comment generation, SWUM can also be used to improve keyword programming [57], detect poorly named methods [49], [58], refactor identifiers [59], or to generate patch documentation [60], [61]. Finally, the statement-level linguistic relationships that can be inferred using SWUM can be considered similar to abstract types, which have been used for both program comprehension and test generation [62], [63].

Debugging. Ko and Myers developed a debugging system based on question and answering [64]. They developed an interface that tracks output-affecting state variables through a trace, and enables the developer to select from a set of NL questions relevant to the output. SWUM could take this idea further by generating more informative questions which are specific to the underlying code, rather than relying on the current strictly predefined questions. In addition, SWUM could enable development of a true question and answering debugging system that allows the developer to enter an arbitrary question expressed in natural language.

Text Mining of Source Code. SWUM can be used to develop customized NL analyses for software through text mining of source code. The node annotations and linguistic relationships in SWUM provide a natural framework for mining finer-grained information beyond simple co-occurrences. SWUM can be used to help mine verb and phrase meanings [65], [66], populate a software ontology [67], generate word relation databases for software [44], or even mine commonly occurring verb-preposition relationships, which can lead to improved construction algorithms for SWUM in the future.

8 Related Work

Our Sofware Word Usage Model (SWUM) is a generalization of the Action-Oriented Identifier Graph (AOIG) [68]. Like the AOIG, SWUM places emphasis on accurately modeling the actions in code by modeling method declarations. However, the AOIG only models verbs and their themes, or direct objects, and does not support the full range of phrase structures found in SWUM. Because SWUM can represent the full range of phrase structures, it can be used in software engineering tools that are not action-oriented, and provides greater accuracy in verb arguments for software engineering tools that are action-oriented.

Beyond AOIG, other researchers have investigated naming conventions and the semantics of method signatures. Liblit, et al. identified common morphological patterns and naming conventions [25], which became the starting point for our rule development. Caprile and Tonella developed a language of function identifiers that shares similarities with our phrase structure node rules [69], and applied it to an identifier restructuring tool [59]. The main differences between Caprile and Tonella's grammar and SWUM's grammar are (1) our phrase structure nodes and grammar closely mimic well-known parsing structures from NLP, and can thus readily take advantage of parsing solutions found in that field; (2) we have developed an algorithm to automatically parse identifiers according to our grammar; and (3) our rules cover a broader set of identifiers and were developed using a much larger code base (18 million LOC in Java versus 230 KLOC in C). Because Caprile and Tonella's grammar was developed exclusively on C code, the similarities between their grammar and SWUM's provide further evidence that the construction rules built for Java can translate to other languages.

Method names have also been analyzed in terms of their most common meanings [70], [66]. Høst and Østvold created a dictionary of commonly occurring method name patterns along with some of their most common attributes in terms of implementation semantics [66]. Although the primary aim of this work is to help developers choose better names, the parsing rules and implementation attributes are similar in spirit to SWUM's grammar and role information for program elements. In fact, SWUM already takes advantage of some of their structural attributes, such as reading or writing fields. Patterns observed in the dictionary could be integrated into SWUM's naming convention rules in the future.

A number of papers explore how developers express behavior through natural language. Butler [71] studied class identifier names and lexical inheritance; analyzing the effect that interfaces or inheritance has on the name of a given class. For example, a class may inherit from a super class or implement a particular interface. Sometimes this class will incorporate words from the interface name or inherited class in its name. His study builds off of work by Singer and Kirkham [72], who identified a naming pattern for class names and studies how different class' names correlate with micro patterns. Amongst Butler's findings, he identifies a number of naming patterns for class names and extends these patterns to identify where inherited names and interface names appear in the pattern. The same author also studies Java field, argument, and variable naming structures [73].

Binkley et al [74] study grammar patterns for attribute names in classes. They come up with four rules for how to write attribute names. Liblit et al. [25] discusses naming in several programming languages and makes observations about how natural language influences the use of words in these languages. Schankin et al. [75] focus on investigating the impact of more informative identifiers on code comprehension. Their findings show an advantage of descriptive identifiers over non-descriptive ones. Hofmeister et al [76] compared comprehension of identifiers containing words against identifiers containing letters and/or abbreviations. Their results show that when identifiers contained only words instead of abbreviations or letters, developer comprehension speed increased by 19% on average.

Several recent studies explore how names evolve and how this evolution is related to behavior. Many of these techniques would benefit from the model SWUM provides. Arnoudova et al. [77] present an approach to analyze and classify identifier renamings. The authors show the impact of proper naming on minimizing software development effort and find that 68% of developers think recommending identifier names would be useful. They also defined a catalog of linguistic anti-patterns [78]. Liu et al.[79] proposed an approach recommends a batch of rename operations to code elements closely related to the rename. They also studied the relationship between argument and parameter names to detect naming anomalies and suggest renames [80]. Peruma et al. [81] studied how terms in an identifier change and contextualized these changes by analyzing commit messages using a topic modeler. They later extend this work to include refactorings that occur with renames [82].

SWUM addresses some of the problems highlighted in recent studies, which advocate for improved models which can bridge the gap between source code behavior/semantics and natural language [83], [84], [**?**], [85]. In addition, SWUM may support efforts to create metrics which more accurately measure readability and comprehension by clarifying the connection between terms found in method names and these methods' behavior behavior.

9 Conclusion

Source code represents a strong coupling of both natural language and program structure. Comprehension of source code requires a strong understanding of how these aspects influence this coupling. Therefore, we have presented SWUM, which integrates both program structure and linguistic information into a single model which can help us formalize our understanding of their relationship. We have shown that SWUM is applicable to, and improves, the output of tools which must leverage static program data and linguistic information. Additionally, we have shown that SWUM is complimentary to part of speech tagging tools for source code and could be used to help create more accurate part of speech output.

In the future, we intend to continue improving SWUM's accuracy on detecting both phrasal concepts. We will increase the number of part of speech tags supported by SWUM and improve SWUMs accuracy on these part of speech tags using the data that we have collected as part of this work. We also plan on releasing an improved version of SWUM to the public to allow for its integration into more research tools in addition to using it to solve relevant issues in software maintenance, such as identifier name recommendation, document generation, and further modeling of natural language artifacts with program structure.

ACKNOWLEDGMENTS

The authors would like to thank Sana Malik, Oana Tudor, Amy Siu, Eric Enslen, Stephen Siegel, and the anonymous reviewers for their useful help and comments on this paper.

REFERENCES

[1] E. Hill, L. Pollock, and K. Vijay-Shanker, "Exploring the neighborhood with Dora to expedite software maintenance," in *ASE '07: Proceedings of the 22nd IEEE International Conference on Automated Software Engineering (ASE'07)*. Washington, DC, USA: IEEE Computer Society, November 2007, pp. 14–23.

[2] D. Poshyvanyk and A. Marcus, "Combining formal concept analysis with information retrieval for concept location in source code," in *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 37–48.

[3] A. Abadi, M. Nisenson, and Y. Simionovici, "A traceability technique for specifications," *ICPC '08: Proceedings of the 16th IEEE International Conference on Program Comprehension*, pp. 103–112, June 2008.

[4] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970–983, 2002.

[5] Y. S. Maarek, D. M. Berry, and G. E. Kaiser, "An information retrieval approach for automatically constructing software libraries," *IEEE Transactions on Software Engineering*, vol. 17, no. 8, pp. 800–813, 1991.

[6] A. Michail and D. Notkin, "Assessing software libraries by browsing similar classes, functions and relationships," in *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1999, pp. 463–472.

[7] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008.

[8] R. Jackendoff, *Semantic Structures*. Cambridge, MA: MIT Press, 1990.

[9] E. Hill, L. Pollock, and K. Vijay-Shanker, "Improving source code search with natural language phrasal representations of method signatures," in *ASE '11: Proceedings of the 26th IEEE International Conference on Automated Software Engineering, short paper*. Washington, DC, USA: IEEE Computer Society, November 2011, pp. 524–527.

[10] B. P. Eddy, J. A. Robinson, N. A. Kraft, and J. C. Carver, "Evaluating source code summarization techniques: Replication and expansion," in *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, june 2013, pp. 13–22.

[11] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, ser. WCRE '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 35–44. [Online]. Available: http://dx.doi.org/10.1109/WCRE.2010.13

[12] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *ICSE '11: Proceedings of the 33rd ACM/IEEE International Conference on Software Engineering*. New York, NY, USA: ACM, 2011, pp. 101–110.

[13] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *ASE '10: Proceedings of the 25th IEEE International Conference on Automated Software Engineering (ASE'10)*, September, 2010.

[14] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, May 2013, pp. 23–32.

[15] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of NL-queries for software maintenance and reuse," in *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 232–242.

[16] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *SIGDOC '05: Proceedings of the 23rd annual international conference on Design of communication*. New York, NY, USA: ACM, 2005, pp. 68–75.

[17] M. Kajko-Mattsson, "A survey of documentation practice within corrective maintenance," *Empirical Softw. Engg.*, vol. 10, no. 1, pp. 31–55, Jan. 2005. [Online]. Available: http://dx.doi.org/10.1023/B:LIDA.0000048322.42751.ca

[18] A. A. Takang, P. A. Grubb, and R. D. Macredie, "The effects of comments and identifier names on program comprehensibility: an experimental investigation," *J. Prog. Lang.*, vol. 4, no. 3, pp. 143–167, 1996.

[19] T. Tenny, "Program readability: Procedures versus comments," *IEEE Trans. Softw. Eng.*, vol. 14, no. 9, pp. 1271–1279, Sep. 1988. [Online]. Available: http://dx.doi.org/10.1109/32.6171

[20] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The effect of modularization and comments on program comprehension," in *ICSE '81: Proceedings of the 5th International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 1981, pp. 215–223.

[21] C. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing*. Cambridge, MA, USA: MIT Press, May 1999.

[22] D. Jurafsky and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.

[23] G. C. Murphy and D. Notkin, "Lightweight lexical source model extraction," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 3, pp. 262–292, 1996.

[24] P. Byckling, P. Gerdt, and J. Sajaniemi, "Roles of variables in object-oriented programming," in *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, 2005, pp. 350–355.

[25] 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*, 2006.

[26] N. Dragan, M. Collard, and J. Maletic, "Reverse engineering method stereotypes," in *Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on*, 24-27 2006, pp. 24 –34.

[27] J. Y. Gil and I. Maman, "Micro patterns in java code," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM, 2005, pp. 97–116.

[28] J. Singer and C. Kirkham, "Exploiting the correspondence between micro patterns and class names," in *SCAM '08: Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2008, pp. 67 –76.

[29] M. Marin, L. Moonen, and A. van Deursen, "Documenting typical crosscutting concerns," in *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, 28-31 2007, pp. 31 –40.

[30] E. Hill, D. Binkley, D. Lawrie, L. Pollock, and K. Vijay-Shanker, "An empirical study of identifier splitting techniques," *Empirical Software Engineering*, 2013, 10.1007/s10664-013-9261-0. [Online]. Available: http://dx.doi.org/10.1007/s10664-011-9174-8

[31] D. Kawahara and S. Kurohashi, "Coordination disambiguation without any similarities," in *COLING '08: Proceedings of the 22nd International Conference on Computational Linguistics*. Morristown, NJ, USA: Association for Computational Linguistics, 2008, pp. 425–432.

[32] D. W. Kosy, “Parsing conjunctions deterministically,” in *Proceedings of the 24th annual meeting on Association for Computational Linguistics*. Morristown, NJ, USA: Association for Computational Linguistics, 1986, pp. 78–84.

[33] A. Okumura and K. Muraki, “Symmetric pattern matching analysis for english coordinate structures,” in *Proceedings of the 4th Conference on Applied Natural Language Processing*. Morristown, NJ, USA: Association for Computational Linguistics, 1994, pp. 41–46.

[34] P. Y. Martin and B. A. Turner, “Grounded theory and organizational research,” *The Journal of Applied Behavioral Science*, vol. 22, no. 2, pp. 141–157, 1986. [Online]. Available: https://doi.org/10.1177/002188638602200207

[35] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, “Learning natural coding conventions,” in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 281–293. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635883

[36] M. Collard and J. I. Maletic, “srcml: A document-oriented xml representation of source code.” [Online]. Available: http://www.sdml.info/projects/srcml/

[37] V. Augustine, “Automating adaptive maintenance changes with srcml and linq,” in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE ’12. New York, NY, USA: ACM, 2012, pp. 9:1–9:2. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393604

[38] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[39] J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The program dependence graph and its use in optimization,” *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.

[40] 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,” in *AOSD ’07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, 2007, pp. 212–224.

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

[42] H. Feild, D. Binkley, and D. Lawrie, “An empirical comparison of techniques for extracting concept abbreviations from identifiers,” in *Proceedings of IASTED International Conference on Software Engineering and Applications (SEA’06)*, Nov. 2006.

[43] 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 5th International Working Conference on Mining Software Repositories*. Washington, DC, USA: IEEE Computer Society, 2008.

[44] 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 16th IEEE International Conference on Program Comprehension*. IEEE, June 2008.

[45] E. L. Antworth, *PC-KIMMO: a two-level processor for morphological analysis*. Dallas, TX: Summer Institute of Linguistics: Occasional Publications in Academic Computing No. 16., 1990, http://www.sil.org/pckimmo/.

[46] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java Language Specification*, 3rd ed. Prentice Hall, June 2005.

[47] A. Wiese, V. Ho, and E. Hill, “A comparison of stemmers on source code identifiers for software search,” in *Proceedings of the 2011 16th IEEE International Conference on Software Maintenance and Reengineering*, ser. ICSM ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 496–499.

[48] M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker, “Automatically mining software-based, semantically-similar words from comment-code mappings,” in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR ’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 377–386. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487155

[49] E. W. Høst and B. M. Østvold, “Debugging method names,” in *ECOOP ’09: Proceedings of the 23rd European Conference on Object-Oriented Programming*, 2009.

[50] S. Mirghasemi, J. J. Barton, and C. Petitpierre, “Naming anonymous javascript functions,” in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, ser. SPLASH ’11. New York, NY, USA: ACM, 2011, pp. 277–288. [Online]. Available: http://doi.acm.org/10.1145/2048147.2048222

[51] M. Hucka, “Spiral: splitters for identifiers in source code files,” *Journal of Open Source Software*, vol. 3, p. 653, 04 2018.

[52] K. Toutanova and C. D. Manning, “Enriching the knowledge sources used in a maximum entropy part-of-speech tagger,” in *Proceedings of the 2000 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora: Held in Conjunction with the 38th Annual Meeting of the Association for Computational Linguistics - Volume 13*, ser. EMNLP ’00. Stroudsburg, PA, USA: Association for Computational Linguistics, 2000, pp. 63–70. [Online]. Available: https://doi.org/10.3115/1117794.1117802

[53] 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.

[54] D. Binkley, D. Lawrie, and C. Morrell, “The need for software specific natural language techniques,” *Empirical Softw. Engg.*, vol. 23, no. 4, pp. 2398–2425, Aug. 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9566-5

[55] D. Poshyvanyk, M. Petrenko, A. Marcus, X. Xie, and D. Liu, “Source code exploration with Google,” in *ICSM ’06: Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM ’06)*, 2006, pp. 334–338.

[56] D. Shepherd, K. Damevski, B. Ropski, and T. Fritz, “Sando: an extensible local code search framework,” in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE ’12. New York, NY, USA: ACM, 2012, pp. 15:1–15:2.

[57] G. Little and R. C. Miller, “Keyword programming in java,” *Automated Software Engineering*, vol. 16, no. 1, pp. 145–192, 2009.

[58] D. Lawrie, H. Feild, and D. Binkley, “An empirical study of rules for well-formed identifiers,” *Journal of Software Maintenance and Evolution*, vol. 19, no. 4, pp. 205–229, 2007.

[59] B. Caprile and P. Tonella, “Restructuring program identifier names,” in *ICSM ’00: Proceedings of the International Conference on Software Maintenance (ICSM’00)*. Washington, DC, USA: IEEE Computer Society, 2000, p. 97.

[60] R. P. Buse and W. R. Weimer, “Automatic documentation inference for exceptions,” in *ISSTA ’08: Proceedings of the 2008 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2008, pp. 273–282.

[61] R. P. Buse and W. Weimer, “Automatically documenting program changes,” in *ASE ’10: Proceedings of the 25th IEEE International Conference on Automated Software Engineering (ASE’10)*, 2010.

[62] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst, “Dynamic inference of abstract types,” in *ISSTA ’06: Proceedings of the 2006 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM Press, 2006, pp. 255–265.

[63] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, “Feedback-directed random test generation,” in *ICSE’07, Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, May 23–25, 2007, pp. 75–84.

[64] A. J. Ko and B. A. Myers, “Debugging reinvented: asking and answering why and why not questions about program behavior,” in *ICSE ’08: Proceedings of the 30th International Conference on Software Engineering*. New York, NY, USA: ACM, 2008, pp. 301–310.

[65] E. W. Høst and B. M. Østvold, “The programmer’s lexicon, volume I: The verbs,” in *SCAM ’07: Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 193–202.

[66] ——, “The java programmer’s phrase book,” in *Proceedings of the 1st International Conference on Software Language Engineering*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 322–341.

[67] R. Witte, Q. Li, Y. Zhang, and J. Rilling, “Text mining and software engineering: An integrated source code and document analysis approach,” *IET Software*, vol. 2, no. 1, pp. 3–16, February 2008.

[68] D. Shepherd, L. Pollock, and K. Vijay-Shanker, “Towards supporting on-demand virtual remodularization using program graphs,” in *AOSD ’06: Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, 2006, pp. 3–14.

[69] B. Caprile and P. Tonella, "Nomen est omen: Analyzing the language of function identifiers," in *WCRE '99: Proceedings of the 6th Working Conference on Reverse Engineering*, 1999, pp. 112–122.

[70] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Control*, vol. 14, no. 3, pp. 261–282, 2006.

[71] S. Butler, "Mining java class identifier naming conventions," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 1641–1643. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337509

[72] J. Singer and C. Kirkham, "Exploiting the correspondence between micro patterns and class names," in *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, Sep. 2008, pp. 67–76.

[73] S. Butler, M. Wermelinger, and Y. Yu, "A survey of the forms of java reference names," in *2015 IEEE 23rd International Conference on Program Comprehension*, May 2015, pp. 196–206.

[74] 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

[75] A. Schankin, A. Berger, D. V. Holt, J. C. Hofmeister, T. Riedel, and M. Beigl, "Descriptive compound identifier names improve source code comprehension," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: ACM, 2018, pp. 31–40. [Online]. Available: http://doi.acm.org/10.1145/3196321.3196332

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

[77] V. Arnaoudova, L. M. Eshkevari, M. D. Penta, R. Oliveto, G. Antoniol, and Y.-G. Gueheneuc, "Repent: Analyzing the nature of identifier renamings," *IEEE Trans. Softw. Eng.*, vol. 40, no. 5, pp. 502–532, May 2014. [Online]. Available: https://doi.org/10.1109/TSE.2014.2312942

[78] V. Arnaoudova, M. Di Penta, G. Antoniol, and Y. Guéhéneuc, "A new family of software anti-patterns: Linguistic anti-patterns," in *2013 17th European Conference on Software Maintenance and Reengineering*, March 2013, pp. 187–196.

[79] H. Liu, Q. Liu, Y. Liu, and Z. Wang, "Identifying renaming opportunities by expanding conducted rename refactorings," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 887–900, 2015.

[80] H. Liu, Q. Liu, C.-A. Staicu, M. Pradel, and Y. Luo, "Nomen est omen: Exploring and exploiting similarities between argument and parameter names," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 1063–1073.

[81] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman, "An empirical investigation of how and why developers rename identifiers," in *International Workshop on Refactoring 2018*, 2018. [Online]. Available: http://doi.acm.org/10.1145/3242163.3242169

[82] ——, "Contextualizing rename decisions using refactorings and commit messages," in *Proceedings of the 19th IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2019.

[83] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-V?squez, D. Poshyvanyk, and R. Oliveto, "Automatically assessing code understandability," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[84] S. Scalabrino, M. Linares-Vásquez, R. Oliveto, and D. Poshyvanyk, "A comprehensive model for code readability," *Journal of Software: Evolution and Process*, vol. 30, no. 6, p. e1958, 2018, e1958 smr.1958. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1958

[85] M. P. Robillard, A. Marcus, C. Treude, G. Bavota, O. Chaparro, N. Ernst, M. A. Gerosa, M. Godfrey, M. Lanza, M. Linares-Vásquez, G. C. Murphy, L. Moreno, D. Shepherd, and E. Wong, "On-demand developer documentation," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2017, pp. 479–483.