Evaluating a Software Word Usage Model for C++

Sana Malik, Emily Hill, Lori Pollock, and K. Vijay-Shanker

August 17, 2009

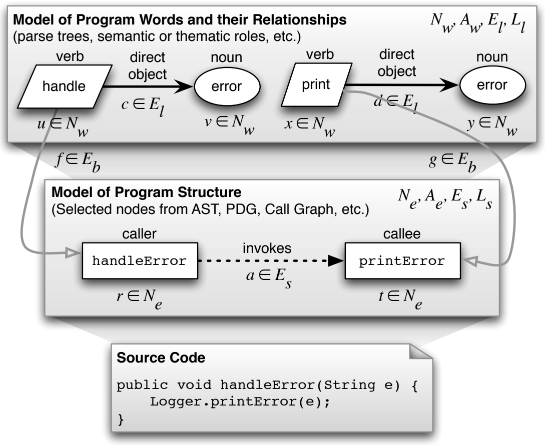
Abstract

Currently, there are many automatic and semi-automatic tools to expedite software maintenance; however, most of these tools rely solely on the structural model of the program, while disregarding any semantic information from the natural language used by the programmer. In previous work towards solving this problem, we develepod a Software Word Usage Model (SWUM) for Java. SWUM enables software engineering tools to apply linguistic relations between words to form a more complete interpretation of the program. Although SWUM is currently defined for Java, we believe that SWUM is capable of representing programs in different programming languages. This paper focuses on investigating the generality and extensibility of SWUM for programming languages beyond Java. The potential structural, semantic, and syntactic modifications of SWUM for other languages were examined, particularly analyzing the differences between Java and C++. We evaluated the effectiveness of the phrases generated from SWUM for C++ code, and modified the SWUM construction algorithm to handle C++ features as needed.

1 Introduction

Throughout the life cycle of an application, between 60-90% of resources are devoted to modifying the application to meet new requirements and to fix faults [2]. With program code growing larger and more complex, software developers need automated software engineering tools to reduce this high maintenance cost. Currently, there are many automatic and semi-automatic tools meant to expedite software maintenance. However, most of these tools rely solely on the structural model of the program, while disregarding any semantic information from the natural language used by the programmer in comments and identifiers.

During software development and maintenance, human programmers read and modify the code that they and others produce, creating code artifacts that are readable as well as runnable [8]. While the programming language syntax and semantics convey the algorithm to be executed, the identifier names and comments express the higher-level conceptual algorithmic steps and domain concepts behind the implementation. For example, from the method name buildQueryForTrace, we can infer that the method's implementation will construct (i.e., build) a query for a trace. The comment provides further elucidation: "Build the sql query string for tracing." Thus, concepts, or ideas, behind the implementation are expressed through words found in comments and identifiers



```
public void handleError(String e) {
    Logger.printError(e);
}
```

Figure 1: SWUM captures program word relationships and links them with program structure.

In previous work [4], we introduced a general Software Word Usage Model (SWUM) that captures the conceptual knowledge of the programmer as expressed in both linguistic information and programming language structure and semantics. SWUM enables software engineering tools to leverage rich linguistic information in the form of phrasal concepts, as opposed to current implementations, which treat a program as a "bag of words" with no relationships. Instead, SWUM uses *phrasal concepts*, concepts expressed as a sequence of words [5], to link related words together. For example, this is particularly beneficial in program search. Consider searching for "add item" in a shopping cart application; a bag of words approach would return irrelevent results that contain either "add" or "item" anywhere within the method, whereas SWUM would

Signature	SWUM Linguistic Model			Example Phrase
	verb	DO	IO	
void Circle::Draw ()	draw	circle		draw circle
circle.MoveTo(new_x, new_y)	move	circle	new x, new y	move circle to new x, new y
library.AddTrack(string aSource)	add	track	a source, library	add track from a source to library
void SetAllSidesTo(int aValue)	set	all sides	a value	set all sides to a value
bool isLink(string aContent)	check	a content		check if a content is link

Table 1: SWUM extracts linguistic information comprised of a verb, direct object, and indirect object to form phrasal concepts. Phrases are derived from the extracted phrasal concept.

rank those results where "add" and "item" are linguistically related in one phrasal concept higher than others.

Although SWUM is capable of representing all programming languages, it is currently implemented only for Java. The potential structural, semantic, and syntactic differences with other languages must be examined to generalize SWUM beyond a single language. In this paper we analyze the differences between Java and C++, modify the SWUM construction algorithm to work for C++, generalize the SWUM model, and present an evaluattion plan.

2 Overview of SWUM

In general, SWUM captures program word relationships and links them with program structure. The basic structural building block of SWUM is an abstract syntax tree (AST). An AST is a representation of the syntactic structure of a program with nodes containing semantic information about the programming language constructs and edges representing hierarchal relations between these constructs [1]. On top of this program structure, SWUM overlays a linguistic layer comprised of verb, direct object, and indirect object relations to form phrasal concepts, as shown in Figure 1.

Formally, a *software word usage model (SWUM)* for a program $\mathcal{P}$ is a tuple consisting of two types of nodes (N) and three sets of labels (L). The two types of nodes are *program element* nodes, which contain syntactic information from the AST, and *word* nodes, which represent the words used in comments and identifiers in the program element nodes. In general, a label $\in L_l$ represents a word relationship, which can be a word dependency or ontological relationship, such as the "is-a" relationship.

To model phrasal concepts, SWUM uses three different types of linguistic edges and labels: word dependencies (different verb phrase structures), "is-a" relationships, and edges inferred from the AST. To construct these edges and correctly label them, knowledge of both English grammar and program structure is necessary. For example, verbs often take both direct and indirect objects. Understanding the fact that verbs take objects prompts us to look for these objects when they are missing from a method name, e.g., formatVersion.compareTo(version). In this example, the verb "compare" takes the direct object "format version" and the indirect object "version". While knowledge of English grammar informs us to look for the missing objects, knowledge of program structure tells us where to look.

3 Taking SWUM from Java to C++

There are various factors we considered in applying SWUM to C++ versus Java. Specifically, there are syntactic, semantic, and stylistic differences found between the two programming languages. For example, we had to investigate inherent language differences such as different keywords between the languages and program structure, as well as user differences such as naming conventions.

3.1 Extracting the Model

In order to see how SWUM can be applied to C++ code, we first had to extract an AST for the C++ program since this is the underlying layer of SWUM. We used the Eclipse C++ Development Tools (CDT) to parse the program and create the AST. The current SWUM implementation for method signatures uses a template that takes as input the method name, return type, parameter names and types, whether it is a boolean function, and whether it is a constructor, as shown in Figure 2. To take SWUM from Java to C++, this information needs to be extracted from the C++ AST.

Implementation To do this, we created a SWUM information collector in the form of an AST visitor that visits each program node. Depending on the node type, the visitor traverses its children to find the information needed for input to SWUM. One major difference between the Java AST and C++ AST is the way the trees ares structured. For the Java AST, all paramters

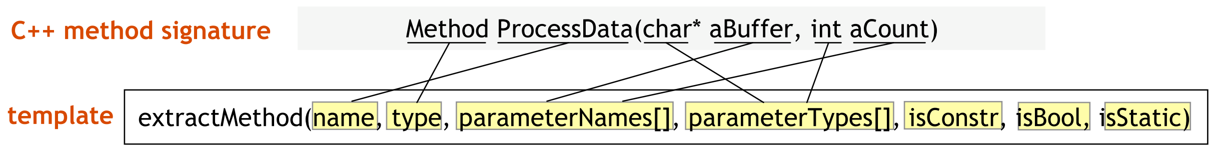

C++ method signature
Method ProcessData(char* aBuffer, int aCount)
template
extractMethod(name, type, parameterNames[], parameterTypes[], isConstr, isBool, isStatic)

Figure 2: The current SWUM implementation for method signatures uses a template that takes as input the method name, return type, parameter names and types, whether it is a constructor, whether it is a boolean function, and whether it is a static function.

and their types were readily available from the Java AST, but there were some challenges in translation for C++. For example, function definition nodes contain information on the name of the function only, but a child node contains information on the function's parameters.

Using this visitor to retrieve raw infrormation, we are able to use the existing SWUM implementation and its methods, and use its output to directly compare results of its performance on C++ versus Java.

3.2 Analyzing Output

The next step is to run the SWUM extractor on the C++ AST. Given the method signature, the SWUM extractor creates a model of the phrase including the verb, direct object, and indirect object. The model output by SWUM's current implementation can be interpreted as a phrase. For example, the method signature void Circle::MoveTo(int newx, int newy) has the verb "move," direct object "circle," and indirect objects "new x" and "new y." This would yeild the phrase "move circle to new x, new y." The phrases derived from SWUM's model enable us to analyze its effectiveness in a concrete manner.

To analyze SWUM on C++, we conducted a manual analysis of the accuracy of the extracted phrases. We determined which constructs were handled correctly by SWUM, in addition to what changes must be implemented to make the phrases more accurate. We iterated this process over a variety of C++ programs, starting from very small and simple programs with very few elements and eventually expanding to programs that covered most of the major features of C++. The programs included the web browser Mozilla with 3,042,083 lines of code, the music player Songbird with 372,731 lines of code, and video client VLC with 600,000 lines of code.

The most notable difference is the structure of classes in Java versus C++. In Java, everything is a class or must be contained within a class. However, in C++ there may be free-floating methods or variables that do not belong to a specific class. This is an issue for SWUM because SWUM uses the containing class as the direct object when available. As seen in Section 2, the direct object is an integral part in modeling linguistic relations in SWUM, so determining the direct object for free-floating C++ programs was a challenge. We found that there were two cases of free-floating method signatures that SWUM could not yet handle: out-of-scope method definitions that included the namespace in the signature and generic functions in the file. Our solution for the former was simple; we simply extracted the scope name from the method name, which were separated by a pair of colons (::). Those functions defined outside any class were more difficult. We had to consider what the best direct object in this case would be as a replacement to the class. We found that often the filename also gave clues about the function or method, so we substituted the name of the file for the declaring type.

Next were the implications that certain keywords have. Certain keywords can have an impact on the extracted model. For instance, SWUM handles the static keyword as a special case because in Java, static means that no class variables are used in the function other than constants. This implies that the method is independent of the class and will not make any changes to the class. For SWUM this means that the class is neither a direct nor indirect object to the function, and that SWUM must look elsehwere to extract this information, such as in the parameters. Since static has the same meaning in both Java and C++, this does not need to be changed in the implementation for C++.

In contrast, the virtual keyword, which indicates that a method in a base class may be overridden by its subclasses, changes how SWUM behaves for Java versus C++. In Java, this is the default for methods and there is no keyword. In C++, however, these functions must be explicitly stated as virtual. This affects how SWUM resolves method invocations.

Constructors are another special case in SWUM and are treated as a "create" or "construct" action. In both Java and C++ these are methods that have the same name as their class and have no explicitly stated return type, which makes them easy to identify. C++, however, has another type of method called a destructor which is essentially the opposite of a constructor. Since Java has automatic garbage collection, this is not needed and has not yet been implemented for SWUM. Based on the fact that it is a background process in Java, we added a case to separate out and disregard all destructors since they do not add any contextual information to the program.

The most similar feature of Java and C++ for SWUM to handle were the return types of methods. The AST of both languages had simple ways of extracting this information from the tree, and once extracted the information had the same implication. The main concern with return types for SWUM is whether or not it is a primitive type. Since C++ and Java have similar primitive types, and the same naming conventions for user-defined classes, it is easy for SWUM to identify whether or not a return type is primitive.

4 Discussion and Reflection

In summary, we propose the following modifications to SWUM for C++:

- Add case for classes defined outside of its class scope.
- Add case for classes outside any class.
- Implement handling for `virtual` keyword
- Ignore destructors

When it comes to method signatures, SWUM for C++ and Java are very similar. The differences between the two are mostly in the underlying program structure and AST, so SWUM is minimally affected. Using an AST visitor as an interface to the SWUM extraction methods makes it easy to abstract away the differences between the two languages. Furthermore, as both languages are object-oriented there is no loss of detail in translating from one language to the other.

5 Evaluation Plan

To evaluate the effectiveness of SWUM on C++ we propose the following plan.

Experimental Design The effectiveness of SWUM on C++ will depend on the accuracy and the completeness of the phrasal concepts it can extrct. We will randomly select 100 method signatures from five programs: 0MQ, an instant messaging client; Chrome, a web browser; Firebird, a database server; NeoMem, a personal organizer; and Xpdf, a PDF viewer. The sizes of these programs are in Table 2. Ten subjects will each recieve 20 method signatures and will manually extract the verb, direct object, and optional indirect object from each signature. We plan to compare the subject extracted phrases with the SWUM extracted phrases in order to determine any areas that may need improvement.

Program	Lines of Code
0MQ	23,159
Chrome	1,700,000
Firebird	50,000
NeoMem	39,382
Xpdf	80,847

Table 2: Source lines of code in programs used in evaluation.

Threats to Validity Obtaining a completely random sample of method signatures for the evaluation would be a threat to validity because some of the signatures may be too similar to accurately represent all features of SWUM. To reduce this risk, we will examine the signatures for diversity so we can ensure all features of SWUM are being exercised.

Although determining part of speech is objective, there may be differences between what the subjects think the verb, direct object, and indirect object are. Because of this, our results may be inaccurate. To reduce this risk, we will give the same signature to multiple subjects.

6 State of the Art

Going beyond the lexical concepts of words to phrasal concepts can yield further improvements in software maintenance tools, and this idea is supported by recent work [3, 13, 14]. Although there has been some work toward representing word relationships in code, to our knowledge, no existing technique automatically captures the lexical concept of a word in conjunction with the context of its surrounding phrase. Some automatic techniques have been developed to capture co-occurring word pairs [10, 11], but co-occurrences do not capture any information about the nature of the relationship between words beyond that the words occur together in the same context. In contrast, the V-DO approach uses <verb, direct object> pairs from method signatures and comments to find actions that cross-cut object-oriented systems [13].

Another potential approach is to capture phrasal concepts in source code with latent semantic analysis (LSA) [7]. However, not only is LSA based on co-occurrences, but the semantic concepts found by LSA do not define phrasal concepts found in text—the concepts are instead represented as a mathematical set that may not correspond to anything expressible in language.

In addition, an approach to automatic generation of domain representations has been suggested for software artifacts, but has not been applied to source code, only documentation [9]. An alternative would be to automatically identify topics in source code [6], and parse the topics to derive word relationships. However, the basic premise of this approach filters the topics based on perceived importance to the system, and thus is incapable of capturing a complete set of phrases for the entire system. Alternatively, reflexion models allow developers to map structural mental models of software artifacts to the source code [12], but the focus is on program structure, rather than linguistic information.

In summary, our work to extract phrasal concepts and word relationships is fundamentally different from prior work. No existing technique is capable of capturing both the textual phrases and the nature of word relationships required to model a variety of phrasal concepts for a given segment of code. With SWUM, we are attempting to capture the full context of a word both within its phrasal concept as well as within the structure of the program.

7 Conclusions and Future Work

As a result of this research, we have developed a C++ information collector that interfaces with the SWUM model extractor. Additionally, the manual analysis of the SWUM extractor on C++ programs has helped develop a list of suggested modifications for SWUM to behave more accurately on C++.

Our next steps are to expand SWUM to include more features of C++ and explore additional programs with varying naming conventions. In the future we would like to investigate other programming languages which SWUM could potentially model; in particular, we would like to choose languages that are vastly different than C++ or Java such as a loosely typed language like Python.

References

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

[2] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[3] 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.

[4] E. Hill, L. Pollock, and K. Vijay-Shanker. Introducing a model of software word usage and its use in searching java source code. In *ICSE '10: the 32nd international conference on Software engineering*, 2010. Under submission.

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

[6] A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: Identifying topics in source code. *Information Systems and Technologies*, 49(3):230–243, 2007.

[7] T. K. Landauer, D. S. McNamara, S. Dennis, and W. Kintsch, editors. *Handbook of Latent Semantic Analysis*. Erlbaum, Mahwah, NJ, USA, 2007.

[8] 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.

[9] J. Lloréns, M. Velasco, A. de Amescua, J. A. Moreiro, and V. Martínez. Automatic generation of domain representations using thesaurus structures. *Journal of the American Society for Information Science and Technology*, 55(10):846–858, 2004.

[10] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, 1991.

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

[12] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, 2001.

[13] 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.

[14] 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*, pages 3–14, 2006.