Natural Language-based Software Analyses and Tools for Software Maintenance

Lori Pollock[1], K. Vijay-Shanker[1],
Emily Hill[2], Giriprasad Sridhara[1], and David Shepherd[3]*

[1] Computer and Information Sciences, University of Delaware, Newark, DE 19716
{pollock,vijay,gsridhar}@cis.udel.edu
[2] Computer Science, Montclair State University, Montclair, NJ 07043
hillem@mail.montclair.edu
[3] ABB Inc., US Corporate Research
davidshepherd@gmail.com

Abstract. Significant portions of software life cycle resources are devoted to program maintenance, which motivates the development of automated techniques and tools to support the tedious, error-prone tasks. Natural language clues from programmers' naming in literals, identifiers, and comments can be leveraged to improve the effectiveness of many software tools. For example, they can be used to increase the accuracy of software search tools, improve the ability of program navigation tools to recommend related methods, and raise the accuracy of other program analyses by providing access to natural language information. This chapter focuses on how to capture, model, and apply the programmers' conceptual knowledge expressed in both linguistic information as well as programming language structure and semantics. We call this kind of analysis Natural Language Program Analysis (NLPA) since it combines natural language processing techniques with program analysis to extract information for analysis of the source program.

Keywords: software maintenance, natural language program analysis, software engineering tools

1 Introduction

Despite decades of knowledge that software engineering techniques can reduce software maintenance costs, focusing on fast initial product releases and leveraging existing legacy systems means that as much as 90% of software life cycle resources are spent on maintenance [22]. Software engineers continually identify and remove bugs, add or modify features, and change code to improve properties such as performance or security. Often, the maintainer is a newcomer to the whole system or the parts of the system relevant to the maintenance task. Before

* The authors' work in this area has been supported by the National Science Foundation Grants No. CCF-0702401 and CCF-0915803.

they can safely make changes, they need to perform tasks that sometimes involve tedious, error-prone collection and analysis of information over these large, complex systems to understand the code adequately for making good decisions.

By automating error-prone tasks that do not require human involvement and presenting that information to maintainers in a useful way, software tools and environments can reduce high maintenance costs. Software development environments now include tools for searching for relevant code segments, navigating the code, providing contextual information at a given program point, and many other kinds of information and predictions that help determine where to make changes, the kinds of changes to make and their potential impact.

Automated program analyses historically build models of the program using the static programming language syntax and semantics and dynamic information from executing the program, and then perform analysis over the model to gather and infer information that is useful for the software maintainer. While the syntax of the program and the associated programming language semantics convey the intended computations to the computer system, programmers often convey the domain concepts through identifier names and comments. This natural language information can be very useful to a wide variety of software development and maintenance tools, including software search, navigation, exploration, debugging, and documentation.

This chapter focuses on how to capture, model, and apply the programmer's conceptual knowledge expressed in both linguistic information rooted in words as well as programming language structure and semantics. We call this kind of analysis, Natural Language Program Analysis (NLPA), since it combines natural language processing techniques with program analysis to extract natural language information from the source program. The underlying premise is that many developers attempt to make their code readable and follow similar patterns in their naming conventions. NLPA leverages these common patterns and naming conventions to automatically extract useful natural language information from the source code.

A number of researchers have studied what these naming conventions are and how they can be leveraged in software engineering tools. Researchers have gained insight into naming convention habits by studying how developers write identifiers and name methods [12, 50, 53] and use domain terms in source code [31], as well as by studying how source code vocabulary in identifiers evolves across multiple versions [2, 3]. These studies have led to guidelines for proper naming [18, 39, 41, 48] and using these conventions to debug poorly named methods [40], as well as identifying synonyms for words used in source code [38] and building dictionaries of verb-object relations [32]. Natural language information has been used by software engineering tools to search source code for concern location [66, 54, 74, 78], find starting points for bug localization [57, 75], automatically recover traceability links between software artifacts [1, 4, 17, 19, 64, 68, 76, 103], assemble and assess software libraries [58, 69], assess code quality [52, 65], and mine topics from source code [46, 55, 67].

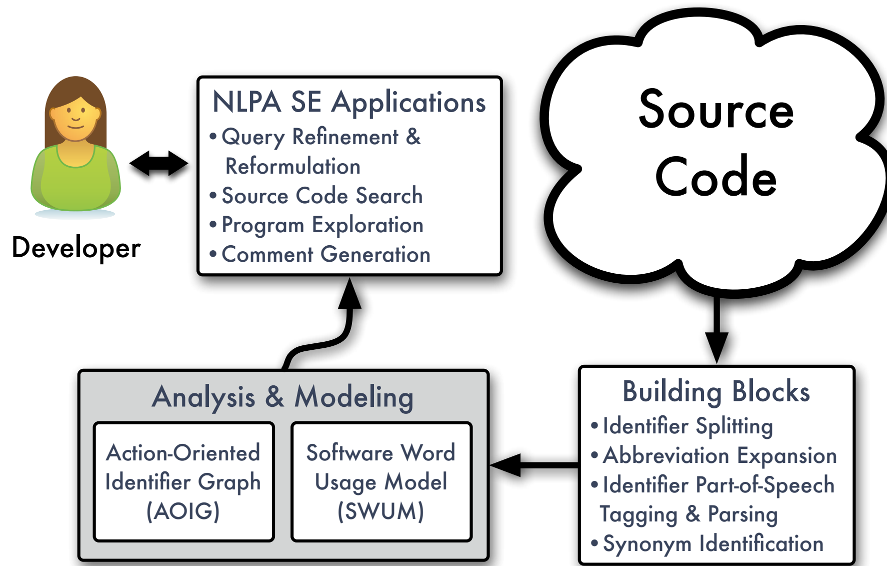

Fig. 1. Chapter Overview

Most existing tools that leverage words in identifiers treat a program as a "bag of words" [63], i.e., words are viewed as independent occurrences with no relationships. Not taking advantage of the natural language semantics captured by the relationships between words can lead to reduced accuracy. For example, consider searching for the query "add item" in a shopping cart application. The presence of "add" and "item" in two separate statements of the same method does not necessarily indicate that the method is performing an "add item" action—the method may be *adding an action* to the system's queue and then *getting the item field* of another object in the system. Ignoring the relationships between words causes irrelevant results to be returned by the search, distracting the user from the relevant results. This suggests that richer semantic representations of natural language information in source code may lead to more accurate software engineering tools.

In this chapter, we provide an overview of NLPA that accounts for *how words occur together in code*, rather than just counting frequencies. NLPA can be used to (a) increase the accuracy of software search tools by providing a natural language description of program artifacts to search, (b) improve the ability of program navigation tools to recommend related procedures through natural language clues, (c) increase the accuracy of other program analyses by providing access to natural language information, and (d) enable automatic comment generation from source code.

Figure 1 depicts an overview of the relationship of the following sections. We begin with the building blocks of NLPA—preprocessing source code identifiers to

enable analysis of the natural language for software engineering tools. We then motivate the focus on analyzing the natural language of source code from the perspective of verbs and actions, and present an overview of a model of software word usage in source code to serve as the underlying model for NLPA-based applications. We describe briefly how NLPA has been used to improve several applications, including search, query reformulation, navigation, and comment generation. We conclude by summarizing the state of NLPA and future directions in preprocessing, analysis, and applications of NLPA.

2 Building Blocks

In this section, we discuss the problems of automatically splitting individual identifiers into component words, automatically expanding abbreviations which are so common in identifiers, automatically tagging the part-of-speech of individual words as used in different identifier contexts, and automatically learning synonyms used in programs to connect similar programmer intent.

2.1 Identifier Splitting

A key first step in analyzing the words that programmers use is to accurately split each identifier into its component words and abbreviations. Programmers often compose identifiers from multiple words and abbreviations, but unlike English writing, program identifiers cannot contain spaces (e.g., ASTVisitorTree, newValidatingXMLInputStream, jLabel6, buildXMLforComposite). Automatic splitting of multi-word identifiers is straightforward when programmers follow conventions such as using non-alphabetic characters (e.g., "_" and numbers) to separate words and abbreviations, or camel-casing (where the first letter of each word is upper case) [12, 18, 49, 53]. However, camel casing is not followed in certain situations, and may be modified to improve readability (e.g., ConvertASCIItoUTF, sizeof, SIMPLETYPENAME).

An identifier-splitting algorithm takes a given set of identifiers as input and outputs the set of substrings partitioning the identifier. An identifier $t = (s_0, s_1, s_3, ...s_n)$, where s_i is a letter, digit, or special character. Most algorithms first separate the id before and after each sequence of special characters and digits, and each substring is then considered as a candidate to be further split. For these alphabetic terms, there are four possible cases to consider in deciding whether to split at a given point between s_i and s_{i+1}:

1. s_i is lower case and s_j is upper case (e.g., getString, setPoint)
2. s_i is upper case and s_j is lower case (e.g., getMAXstring, ASTVisitor)
3. both s_i and s_j are lower case (e.g., notype, databasefield, actionparameters)
4. both s_i and s_j are upper case (e.g., NONNEGATIVEDECIMALTYPE)

Case (1) is the natural place to split for straightforward camel case without abbreviations. Case (2) demonstrates how following strict camel casing can provide incorrect splitting (e.g., get MA Xstring). We call the problem of deciding where to split when there is alternating lower and upper case present, the

mixed-case id splitting problem. We refer to cases (3) and (4) as the *same-case id splitting problem.*

Currently, there exists a small set of identifier-splitting algorithms. Some depend on dictionaries, some exploit the occurrence of component words in other parts of the source code and use frequencies, and some combine this information. The greedy approach [24] is based on a predefined dictionary of words and abbreviations, and splits are determined based on whether the word is found in the dictionary, with longer words preferred. The Samurai [20] approach is based on the premise that strings composing multi-word identifiers in a given program are most likely used elsewhere in the same program, or in other programs. Thus, Samurai mines string frequencies from source code, and builds a program-specific frequency table and a global frequency table from mining a large corpus of programs. The frequency tables are used in the scoring function applied during both mixed-case splitting and same-case splitting.

GenTest [51] focuses on the same-case splitting problem. Given a same-case term, GenTest first generates all possible splittings. Each potential split is scored (i.e., tested) and the highest scoring split is selected. The scoring function uses a set of metrics ranging over term characteristics, dictionaries and information from non-source code artifacts, and information derived from the program itself or corpus of programs. The Dynamic Time Warping approach [59] is based on the observation that programmers build new identifiers by applying a set of transformation rules to words, such as dropping all vowels. Using a dictionary containing words and terms belonging to the application domain or synonymous, the goal is to identify a near optimal matching between substrings of the identifier and words in the dictionary, using an approach inspired by speech recognition.

Enslen, et al.'s [20] empirical study showed that Samurai misses same-case splits identified by the Greedy algorithm but outperforms Greedy overall by making significantly fewer oversplits. Lawrie, et al. [51] results from comparing Greedy, Samurai, and GenTest on same-case identifier splitting for Java showed that both GenTest and Samurai achieve at least 84% accuracy in identifier splitting, with GenTest achieving slight higher accuracy than Samurai. The Dynamic Time Warping approach has not been evaluated against the other techniques yet.

2.2 Abbreviation Expansion

When writing software, developers often use abbreviations in identifier names, especially for identifiers that must be typed often and for domain-specific words used in comments. Most existing software tools that use the natural language information in comments and identifiers do nothing to address abbreviations, and therefore may miss meaningful pieces of code or relationships between software artifacts. For example, if a developer is searching for context handling code, she might enter the query 'context'. If the abbreviation 'ctx' is used in the code instead of 'context', the search tool will miss relevant code.

Abbreviations used in program identifiers generally fall into two categories: single-word and multi-word. Single-word abbreviations are short forms whose long form (full word expansion) consists of a single word, such as 'attr' (attribute)

and 'src' (source). Single letter abbreviations are also commonly used, predominantly for local variables with very little scope outside a class or method [53], such as 'i' (integer). Multi-word abbreviations are short forms that when expanded into long form consist of more than one word, such as acronyms. In fact, acronyms can be so widely used that the long form is rarely seen, such as 'ftp' or 'gif'. Some uses of acronyms are very localized, such as type acronyms. When creating local variables or naming method parameters, a common naming scheme is to use the type's abbreviation. For example, a variable of the type ArrayIndexOutOfBoundsException may be abbreviated 'aiobe'. Some multi-word abbreviations combine single-word abbreviations, acronyms, or dictionary words. Examples include 'oid' (object identifier) and 'doctype' (document type).

Expansion Techniques Automatically expanding abbreviations requires the following steps: (1) identifying whether a token is a non-dictionary word, and therefore a short form candidate; (2) searching for potential long forms for the given short form; and (3) selecting the most appropriate long form from among the set of potential long form candidates.

One simple way to expand short forms in code is to manually create a dictionary of common short forms [85]. Although most developers understand that 'str' is a short form for 'string', not all abbreviations are as easy to resolve. Consider the abbreviation 'comp'. Depending on the context in which the word appears, 'comp' could mean either 'compare' or 'component'. Thus, a simple dictionary of common short forms will not suffice. In addition, manually created dictionaries are limited to abbreviations known to the dictionary builders.

More intelligent abbreviation expansion mechanisms have been developed for software. Lawrie, Feild, and Binkley (LFB) [49] extract lists of potential expansions as words and phrases, and perform a two-stage expansion for each abbreviation occurrence in the code. For each function *f* in the program, they create a list of words contained in the comments before or within the function *f* or in identifiers with word boundaries (e.g., camel casing) occurring in *f*, and a phrase dictionary created by running the comments and multi-word-identifiers through a phrase finder [25]. They create a stop word list containing programming language keywords. After stemming and filtering by the stop word list, expansion of a given non-dictionary word occurrence in a function *f* involves first looking in *f*'s word list and phrase dictionary, and then in a natural language dictionary. A word is a potential expansion of an abbreviation when the abbreviation starts with the same letter and every letter of the abbreviation occurs in the word in order. This technique returns a potential expansion only if there is a single possible expansion.

When they manually checked a random sample of 64 identifiers requiring expansion (from a set of C, C++, and Java codes), only approximately 20% of the identifiers were expanded correctly. In another quantitative study of all identifiers in their 158-program suite of over 8 million unique terms, only 7% of the total number of identifier terms were expanded by their technique; these expansions were not checked for correctness.

The AMAP abbreviation expansion approach [35] was developed with the goal of improving on this technique and including heuristics to choose between multiple possible expansions. AMAP utilizes a scoping approach similar to variable scoping and automatically mines potential long forms for a given short form from the source code. AMAP creates a regular expression from the short form to search for potential long forms. When looking for long forms, AMAP starts at the closest scope to the short form, such as type names and statements, and gradually broadens its scope to include the method, its comments, and the class comments. If the technique is still unsuccessful in finding a long form, it attempts to find the most likely long form found within the program and then in Java SE 1.5. With each successive scope, AMAP includes more general, i.e., less domain-specific, information in its long form search.

In an evaluation of 227 non-dictionary words randomly selected from 5 open source programs, AMAP automatically expanded 63% of the words to the correct long form. On the same set, LFB expanded 40% correctly.

2.3 Part-of-Speech Tagging and Identifier Parsing

To facilitate extraction of relations between words appearing in identifiers, the identifier splitting and abbreviation expansion are followed by identifying (i.e., tagging) the parts of speech of each word in the identifier and then the lexical components of the identifier. Each word in an identifier is tagged with a part-of-speech such as noun, noun modifier, verb, verb modifier, or preposition, and identifiers are then chunked into phrases such as noun phrases, verb groups, and prepositional phrases. In the example below, words in identifier `findFileInPaths` are tagged as verb - noun - preposition - noun, and chunked as follows:

```
File findFileInPaths(): [find]: VG [file]: NP [in [paths]: NP]: PP
```

Part-of-speech tagging and identifier parsing in software is complicated by several programmer behaviors. Programmers invent new non-dictionary words and their own grammatical structures when naming program elements. For instance, they form new adjectives by adding “able” to a verb (e.g., “give” becomes “givable”). Thus, traditional parsers for natural language fail to accurately capture the lexical structure of program identifiers.

Liblit, et al. identified common morphological patterns and naming conventions [53], which can serve as a starting point for parsing rule development. Høst and Østvold created a phrase book of commonly occurring method name patterns [41]. Although the contents of the phrase book can be used to generate more accurate semantic representations, the rules cannot be applied to parse arbitrary method signatures.

To represent each method name as a verb and direct object, Shepherd, et al. [88] used WordNet [70] to approximate possible parts of speech for words in method names, favoring the verb tag for words in the first position because methods typically encapsulate actions. Through manual analysis of function

identifiers, Caprile and Tonella developed a grammar for function identifiers [12], and applied it to an identifier restructuring tool [13]. Hill [34] developed a set of identifier grammar rules for Java, as part of the construction of the Software Word Usage Model (SWUM) defined in Section 3. Caprile and Tonella's grammar shares similarities with the SWUM grammar rules; however, SWUM's 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 provides further evidence that the construction rules built for Java can translate to other languages.

Parsing rules can be developed by identifying common word and grammar patterns from a corpus of programs, developing a generalized set of rules based on the word and grammar patterns, evaluating their effectiveness, and then iterating to expand the set of parsing rules to capture more identifier categories accurately. Using this approach, Hill, et al. [34] developed an algorithm to automatically parse identifiers according to the grammar. Malik [60] improves the accuracy of the SWUM grammar by extensive POS tagging based on suffixes, discovering more kinds of method signatures due to programmer conventions, and using context frequencies in determining POS tags.

2.4 Synonyms in Programs

A human developer skimming for code related to "removing an item from a shopping cart" understands that the method deleteCartItem(Item) is relevant, even though it uses the synonym *delete* rather than *remove*. Similarly, automated tools such as code search and query reformulation need to automatically recognize these synonym relations between words to be able to successfully help humans find related code in large-scale software systems. In fact, knowledge of word relations such as synonyms, antonyms, hypernyms, and hyponyms can all aid in improving the effectiveness of software tools supporting software maintenance activities.

Broadly defined, search tools use queries and similarity measures on software artifacts (source code, documentation, maintenance requests, version control logs, etc.) to facilitate a particular software engineering or program comprehension task. Tools which use natural language or keyword queries and matching can benefit by expanding queries and adding related words to textual artifact representations. For example, synonyms are especially useful in overcoming vocabulary mismatches between the query and software artifacts, especially with regard to the concept assignment problem [8].

Several software maintenance tools have been developed that use some notion of synonyms. FindConcept [87] expands search queries with synonyms to locate concerns more accurately in code. FindConcept obtains synonyms from WordNet [70], a lexical database of word relations that was manually constructed for English text. iComment [98] automatically expands queries with similar topic words to resolve inconsistencies between comments and code and therefore helps

to automatically locate bugs. Their lexical database of word relations was automatically mined from the comments of two large programs.

One potential technique for finding synonyms and other word relations in software is to use Latent Semantic Indexing, an information retrieval technique that uses the co-occurrences of words in documents to discover hidden semantic relations between words [64, 66]. However, since the technique is based on co-occurrences of words, the resulting word relations are not guaranteed to be semantically similar. Another approach is to use synonyms found in English text, such as the synonyms found in WordNet, for finding the synonyms used in software.

Sridhara, et al. [92] performed a comparative study of six state of the art, English-based semantic similarity techniques (used for finding various word relations) to evaluate their effectiveness on words from the comments and identifiers in software. Their results suggest that applying English-based semantic similarity techniques to software without any customization could be detrimental to the performance of the client software tools. The analysis indicated that none of the techniques appear to perform well at recall levels above 25%, where recall is the percentage of true positives in related word pairs returned. In general, the number of returned possible synonyms can be 10 times greater than the number of desired results, and much more for high levels of recall. Sridhara, et al. propose two promising strategies to customize the existing semantic similarity techniques to software: (1) augment WordNet with relations specific to software, possibly by mining word relations in software, or (2) improve the estimation of word probabilities used by the information content-based techniques, which currently use a probability distribution of words based on English text.

Falleri et al. [23] use English-based POS tagging to automatically extract and organize concepts from program identifiers in a WordNet-like structure, focusing on hyperonym and hyponym relations between the extracted concepts. They share similar insights into the challenges of applying these techniques to software with [92]. Host and Ostvold [38] propose identifying synonymous verbs by associating a verb with each method, characterizing each method by a set of attributes, and measuring different forms of entropy over the corpus of methods with their attributes and associated verbs. Specifically, they investigate the effect on entropy in the corpus when they eliminate one of the verbs as a possible synonymous verb pair. If the effects are beneficial, they say that a possible synonym has been identified. Their results showed that they could identify reasonable synonym candidates for many verbs, but choosing genuine synonyms among the candidates will require a more sophisticated model of the abstract semantics of methods.

3 Analysis and Modeling

For software maintenance tasks, action words are central because most maintenance tasks involve modifying, perfecting, or adapting existing actions [88]. Actions tend to be scattered in object-oriented programs, because the organi-

zation of actions is secondary to the organization of objects [100]. Like English, typically actions (or operations) in source are represented by verbs, and nouns correspond to objects [10]. In a programming language, verbs usually appear in the identifiers of method names, possibly in part because the Java Language Specification recommends that "method names should be verbs or verb phrases and class types should be descriptive nouns or noun phrases" [30]. Therefore, initial extraction efforts focused on method names and the surrounding information (i.e., method signatures and comments).

In English or software, identifying the verb in a phrase does not always fully describe the phrase's action. To fully describe a specific action, it is important to consider the *theme*. A *theme* is the object that the action (implied by the verb) acts upon, and usually appears as a direct object (DO). There is an especially strong relationship between verbs and their themes in English [14]. An example is (parked, car) in the sentence "The person parked the *car*." Similarly, in the context of a single program code, often verbs, such as "remove," act on many different objects, such as "remove attribute", "remove screen", "remove entry", and "remove template". Therefore, to identify specific actions in a program, it is important to examine the direct objects of each verb (e.g., the direct object of the phrase "remove the attribute" is "attribute").

Initial NLPA [28, 88, 87] analyzed the source code to extract action words, in the form of verb-DO pairs. A verb-DO pair is defined to be two terms in which the first term is an action or verb, and the second term is a direct object for the first term's action. The program is modeled by an action-oriented identifier graph (AOIG) that explicitly represents the occurrences of verbs and direct objects of a program, mapping each verb-DO pair with their occurrences in the code.

The analysis focuses on occurrences of verbs and DOs in method declarations and comments, string literals, and local variable names within or referring to method declarations. After identifier splitting, POS tagging and chunking, a set of verbs is extracted from the method signature. POS tagging typically finds verbs in either the leftmost position or rightmost position or not in the signature. Special verbs such as "run" and "convert", which do not occur explicitly in names, are implicitly identified through common word patterns in identifiers. The direct object is identified based on the position of the verb. If a set of extraction rules fits the same identifier, a set of verb-DO pairs is returned with the client application ultimately determining the most appropriate pair. Table 1 shows examples of the locations where verbs are identified and how the corresponding DO is identified.

Class	**Verb**	**DO**	**Example**
Left-Verb	Leftmost word	Rest of method name	public URL *parse***Url**()
Right-Verb	Rightmost word	Rest of method name	public void **mouse***Dragged*()
Special-Verb	Leftmost word	Specific to Verb	public void **HostList**.on*Save*()
Unidentifiable-Verb	"no verb"	Method name	public void **message**()

Table 1. Locating Verb-DO Pairs in Method Signatures

The AOIG is a first step in representing the actions and themes occurring in source code as verb-DO pairs. The next step in NLPA, the Software Word Usage Model (SWUM), was developed to address two of AOIG's shortcomings: (1) any verb in a methods name is assumed to be its action, and (2) verb-DO pairs were the only NLPA information extracted from source code.

First, the AOIG's rules to extract verb-DO pairs seek to capture the method's intended action and theme as verb and DO only. This works well for methods like parseURL(String url), where the verb-DO "parse URL" accurately represents the method's action and theme. However, not all identifiable verbs and objects in a method's name capture intended actions. Consider mouseDragged(), where the method's implementation is reacting to the mouse dragged event. AOIG's rules greedily identify the verb-DO as "dragged mouse." Instead, when the action is difficult to determine from the signature alone, SWUM marks such methods as general or event handlers to indicate that the model may not accurately capture the method's action. In this example, SWUM extracts the generic "handle mouse dragged" as an event handler.

Aside from these cases, the AOIG generally identifies the correct verb and DO as the action and theme. There are some situations where the action and theme are insufficient to accurately represent the source code's intent. For example, consider searching an online store application for code related to adding an item to the shopping cart. The action-theme "add item" will occur in many different contexts, such as adding an item to a shopping cart, a wish list, or an inventory list. Although the AOIG can model "add item," it cannot model the more specific "add item to cart," "add item to wish list," or "add item to inventory" concepts that would differentiate between the relevant and irrelevant pieces of source code.

SWUM moves beyond verb-DO pairs by capturing arbitrary *phrasal concepts*, such as full verb phrases (VPs) with verbs, direct objects, indirect objects and prepositions, and noun phrases (NPs) with no identifiable verb. For example, consider the method addEntry(ItemEntry ie) in the ShoppingCart class. SWUM's phrasal concepts can capture the verb, DO, and the indirect object: "add entry to shopping cart". SWUM also associates the DO in the method's name, entry, with the equivalent parameter, ItemEntry. Thus, SWUM's phrasal concepts can capture deeper semantic relationships found anywhere in the source code. Table 2 includes additional examples of how SWUM extracts the action (e.g., verb), theme (e.g., direct object), and secondary argument (e.g., indirect object) semantic roles for different categories of method signatures, overcoming many AOIG limitations.

SWUM uses a number of heuristics to determine whether the method name should be parsed as a NP, VP, or some special class of method name. For example, boolean checkers like isVisible or containsKey are a special class of method name. Another special class is the set of general names, which include event-driven methods like actionPerformed(), keyPressed(), or Thread.run(). If a method name has not been classified as a checker, general, or beginning with a preposition, SWUM assumes the name starts with a verb in base form and moves on to identifying the verb's arguments. The direct and indirect objects are inferred by

Class	**Example**	**SWUM action-theme [secondary argument]**
Base verb	URLHandler.parseURL(String url)	parse-URL
Modal verb	FIFO.canUnmount(Device device)	can unmount-device
Contains Preposition	DropDownButton.addToToolBar(JToolBar toolbar)	add-drop down button [*to* tool bar]
Event Handler	MotionListener.mouseDragged(MouseEvent e)	handle-mouse dragged
Begins with Preposition	HostList.onSave()	handle-on save
Noun Phrase	BaseList.newIterator()	get-new iterator
Void Noun Phrase	JoinAction.message()	handle-message

Table 2. Example SWUM Representation for Method Signatures

looking in the name, parameters, declaring class, and return type of the method signature.

4 Applications of Natural Language Program Analysis

This section surveys several client tools that have been built to demonstrate how leveraging NLPA can improve a broad range of tools useful during software maintenance.

4.1 Targeted Software Maintenance Tools

To modify an application, developers must identify the high-level idea, or concept, to be changed and then locate the concept's concern, or implementation, in the code. This is called the concern location problem. In object-oriented languages where the code is organized around objects or classes, action-oriented concerns, such as "play track", are scattered across different classes and files, i.e., cross-cutting.

Source Code Search for Concern Location To identify code relevant to a concern, developers typically use an *iterative refinement* process [26, 33] as shown in Figure 2. In this process, the developer enters a query into a source code search tool. Depending on the relevance of the results, the user will reformulate the query and search again. This process continues until the user is satisfied with the results (or gives up). In this process, the user has two important tasks: (1) query formulation and (2) determining whether the search results are relevant.

Studies show that formulating effective natural language queries can be as important as the search algorithm itself [33]. During query formulation, the developer must guess what words were used by the original developer to implement the targeted feature. Unfortunately, the likelihood of two people choosing the same keyword for a familiar concept is only between 10-15% [29]. Specifically, query formulation is complicated by the vocabulary mismatch problem [33]

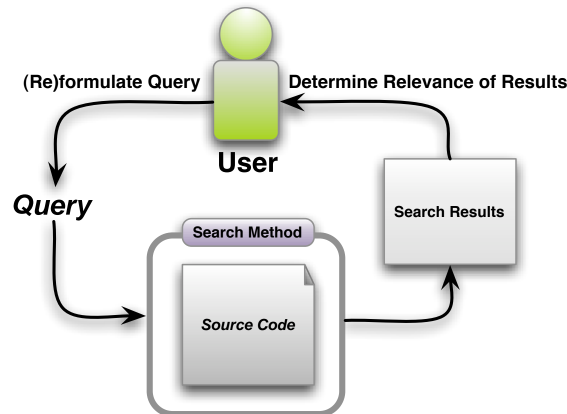

Fig. 2. Iterative Query Refinement and Search Process

(multiple words for the same topic), polysemy (one word with multiple meanings), and the fact that queries with words that frequently occur in the software system will return many irrelevant results [63].

It is very difficult to overcome these challenges by automatically expanding a query on the user's behalf. For polysemy and word frequency, the user needs to add additional query words about the feature to restrict the search results. Such detailed knowledge about the feature exists only in the developer's mind. Further, automatically expanding a query with inappropriate synonyms can return worse results than using no expansion [92]. Thus, the role of automation is not to automatically expand the query, but to provide support that will enable the human user to quickly formulate an effective query.

Few systems recommend alternative words to help developers reformulate poor queries. One approach automatically suggests close matches for misspelled query terms [73], but does not address the larger vocabulary mismatch problem. Sections 4.2 and 4.3 present two complementary approaches that help the developer to formulate effective queries. These sections, along with Section 4.4, also present innovative ways of using NLPA to improve source code search for concern location.

Exploring and Understanding Concerns Navigation and exploration tools help developers explore and understand the program structure from a starting point in the code. In general, these fall into two main categories: semi-automated approaches, which provide automatically gathered information to the user but require the developer to initiate every navigation step (stepwise), and approaches that automatically traverse the program structure and return many related elements without user intervention (recursive).

Stepwise navigation tools begin from a relevant starting element and allow developers to explore structurally related program elements such as methods,

fields, or classes. Some navigation tools allow developers to query structurally connected components one edge away [15, 82, 89] or recommend structurally related elements 1-2 edges away [80, 86]. Stepwise navigation tools suggest manageable numbers of elements to be investigated, but provide limited contextual information since the developer is only presented a small neighborhood of program elements at each step. Each successive structural element to be explored must be manually selected. For example, if a developer were to use a stepwise navigation tool for an "add auction" concern consisting of 24 methods and 6 fields, the developer would have to initiate as many as 19 exploration steps.

In contrast, recursive exploration tools provide more structural context by automatically exploring many structural edges away from the starting element [95, 102, 106, 107] (e.g., by including callers 5 edges up the call chain). For instance, program slicing identifies which elements of a program may affect the data values computed at some point of interest, usually by following edges in a program dependence graph [102]. Because the number of structurally connected components can grow very quickly as new program elements are added to the result set, some recursive navigation tools (e.g., thin slicing) employ filtering techniques to eliminate unnecessary results [95]. In addition, a textual similarity metric has been used as a stopping criteria in slicing [43], which is another way of filtering.

Rather than explore data dependences, some recursive navigation techniques reduce expense by exploring the call graph [36, 107]. One approach is to filter based purely on call graph information such as the number of edges away from the starting element or the number of callees [107]. These filters can be further refined by using textual information [36]. Section 4.5 presents a recursive program exploration technique that takes advantage of NLPA.

Once the developer has located the elements of the concern, the next step is to understand the related code. Several studies have demonstrated the utility of comments for understanding software [97, 101, 105]. However, few software projects adequately document the code to reduce future maintenance costs [44, 90]. To alleviate this problem, Section 4.6 presents an NLPA-based technique to automatically generate comments directly from the source code.

4.2 FindConcept: A Concern Location Tool based on the Action-oriented Identifier Graph

FindConcept [87] is a concern location tool that leverages both traditional structural program analysis and natural language processing of source code. FindConcept improved upon the state-of-the-art by expanding the user's initial query and by searching over a natural language representation of source code (i.e., the AOIG).

When using FindConcept, developers formulate their query as a verb-DO pair (e.g., "draw circle"). FindConcept then helps the user expand their query by suggesting additional terms. FindConcept generates these suggestions by comparing the initial query to existing terms in the AOIG and by analyzing the usage patterns of these relevant words. When the user is satisfied with their expanded query, they trigger a search over the AOIG program representation,

which returns search results displayed as a program graph. Displaying results as a graph of methods connected by structural edges (e.g., call graph edges) allows developers to quickly understand the concern.

Shepherd, et al. [87] evaluated FindConcept against two other concern location tools. During this evaluation, eighteen programmers completed a set of nine search tasks, where each search task consisted of an application and a concept to be found. Programmers were asked to use one of the three tools to complete each task (see [87] for detailed experimental setup).

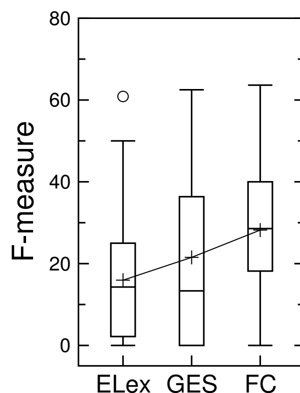

Fig. 3. Overall effectiveness results by search tool; FC = FindConcept.

FindConcept was compared with Eclipse's built-in lexical search (ELex [42]) and a modified Google Eclipse search (GES [74]). Similar to grep's functionality, ELex allows users to search using a regular expression query over source code, returning an unranked list of files that match the query. GES integrates Google Desktop Search into the Eclipse workbench, allowing users to search Java files with information-retrieval-style queries and return a set of ranked files. GES was modified to return individual methods instead of files, for comparison.

Shepherd, et al. [87] used F measure, which combines precision and recall, to measure the effectiveness of FindConcept, GES, and Elex. They measured user effort by tracking the time that users spent formulating a final query for each task. Figure 3 shows the F measure results. The box represents the inner 50% of the data, the middle line represents the median, the plus represents the mean, and outliers are represented by an '∘'. According to these measures, their study showed that FindConcept was more consistently effective than either Elex or GES, without requiring additional user effort. Analysis of the cases in which FindConcept's performance was worse or similar to GES or Elex indicated that straightforward improvements to the AOIG creation process would improve FindConcept's effectiveness. These observations have informed subsequent work in extracting natural language information from source code [34].

Based on this initial success, Hill, et al.'s [34] work has generalized FindConcept's approach, including significant contributions to the query expansion process and the creation of a more general natural language representation of source code. Their work not only extracts verb-DO pairs but entire verb phrases from source code, which avoids the issues FindConcept encountered during its evaluation.

4.3 Contextual Query Reformulation

In addition to providing automated support to the developer in formulating queries in a different way than FindConcept, the contextual query reformulation technique [37], called *contextual search*, also helps the user discriminate between relevant and irrelevant search results. The key insight is that the *context* of words surrounding the query terms in the code is important to quickly determine result relevance and reformulate queries. For example, online search engines such as Google display the context of words when searching natural language text. The contextual search approach automatically captures the context of query words by extracting and generating natural language *phrases*, or word sequences, from the underlying source code. By associating and displaying these phrases with the program elements they describe, the user can see the context of the matches to the query words, and determine the relevance of each program element to the search.

Consider the search results for the query "convert" in Figure 4. The method signatures matching the query are to the right, with the corresponding phrases to the left. By skimming the list of words occurring with "convert" in these phrases, we notice that convert can behave as a verb which acts on objects such as "result,", "arg", or "parameter"; or convert can itself be acted upon or modified by words such as "can" and "get args to." If the user were searching for code related to "converting arguments," they could quickly scan the list of phrases and identify "convert arg" as relevant. Thus, understanding this context allows the user to quickly discard irrelevant results without having to investigate the code, and focus on groups of related signatures that are more likely to be relevant.

```
convert (9) >
   convert result (3) >
   convert arg (2) >
   can convert :: NativeJavaObject static boolean canConvert(Object fromObj, Class to)
   get args to convert :: JavaAdapter static int[] getArgsToConvert(Class[] argTypes)
   convert to string :: Main static Object readFileOrUrl(String path, boolean convertToString)
   convert parameter :: Optimizer boolean convertParameter(Node n)
```

Fig. 4. Example results for "convert" query. Phrases are to the left, followed by the number of matching signatures, and signatures follow '::'.

Going Beyond Verb-DO Queries The experimental study described in Section 4.2 showed that capturing specific word relations in identifiers, such as verb-DO pairs, enabled users to produce more effective queries more consistently than with two competing search tools. However, strict verb-DO queries cannot be used to search for every feature. For example, verb-DO pairs cannot be used to search for features expressed as noun phrases without a verb, such as "reserved keyword" or "mp3 player."

One potential approach to go beyond verb-DO pairs is to capture all word relation pairs in software by using co-occurrences [62]. The key problem with co-occurring word pairs is that *word order matters*. For example, knowing that "item" and "add" co-occur more often than due to chance is less useful than simply knowing that the phrase "add item" frequently occurs. This observation prompted the use of phrases, based on SWUM's phrasal concepts, to develop the contextual query reformulation technique.

Contextual query reformulation relies on SWUM's phrasal concepts to extract phrases from source code because existing techniques for extracting phrases did not meet the needs of the concern location problem. There is work on automatically extracting topic words and phrases from source code [67, 71], displaying search results in a concept lattice of keywords [72], and clustering program elements that share similar phrases [46]. Although useful for exploring the overall word usage of an unfamiliar software system, these techniques are not sufficient for exploring all usage. In contrast to the contextual approach, these approaches either filter the topics based on perceived importance to the system [46, 71, 72], or do not produce human understandable topic labels [67]. Since it is impossible to predict a priori what will be of interest to the developer, the contextual approach lets the developer filter the results with a natural language query, and uses human-readable extracted phrases.

The Contextual Query Reformulation Approach After using SWUM to automatically extract phrases for method signatures, the contextual query reformulation technique searches the resulting phrases for instances of the query words. Related phrases, along with the methods they were generated from, are grouped into a hierarchy based on partial phrase matching. As illustrated in Figure 4, phrases at the top of the hierarchy are more general and contain fewer words, whereas phrases more deeply nested in the hierarchy are more specific and contain more words.

An empirical evaluation with 22 developers was conducted to compare contextual search (*context*) with verb-DO (*V-DO*) recommendations without synonym suggestions from WordNet. Synonyms were not used in order to explore whether natural language phrases beyond *V-DO* improve searching capabilities, without studying effects caused by synonym recommendations or other minor algorithmic differences.

The results show that contextual search significantly outperforms V-DO recommendations in terms of effort and effectiveness. Figure 5 presents the results of the comparison in a box and whisker plot. The box represents the inner 50%

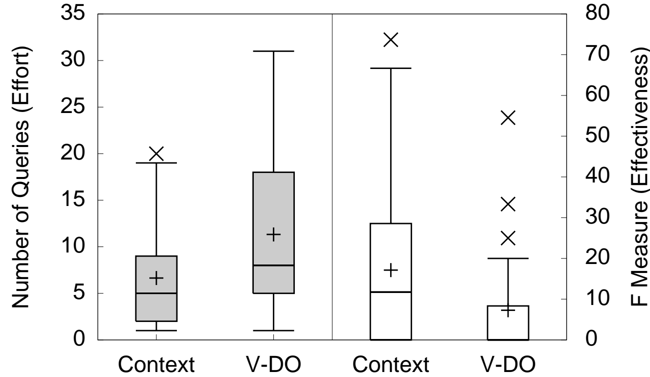

Fig. 5. Effort and Effectiveness Results for *context* and *V-DO*. Effort is measured in terms of the number of queries entered, shown on the left. Effectiveness is measured in terms of the F Measure, shown on the right.

of the data, the middle line represents the median, the plus represents the mean, and outliers are represented by an '×'.

In terms of effort, shown on the left, developers entered 5 more queries on average for *V-DO* than for *context*. In most cases, this was due to the fact that users found it difficult to formulate strict V-DO queries for all the concerns. One subject said,"I really liked the verb-direct object search add-on, but had trouble formulating some of the mandatory verbs, for example with the sqrt2 query." In situations where *V-DO* could not extract a verb, users had trouble formulating successful queries and therefore expended more effort than with *context*.

V-DO's inability to extract verbs in all situations also led to poor effectiveness, shown on the right in Figure 5. Although the developers found *V-DO*'s query recommendations to be helpful, the recommendations did not provide significantly improved results. For example, another subject said, "In the V-DO part especially, it was difficult to find an accurate list [of signatures] for each concern by specifying complete V-DO combinations." Thus, the more flexible phrase extraction process of *context* allowed for higher F measure values.

4.4 SWUM-based Search

As described in Section 4.2, experimental results showed that AOIG-based Find-Concept is more consistently effective than two existing search techniques. Source code search effectiveness can be even further increased by taking advantage of SWUM's richer representation of natural language in a program [34]. The core of SWUM-based search is the SWUM-based scoring function, *swum*, which scores the relevance of program elements based on where the query words occur in the code by integrating location, semantic role, head distance, and usage information:

- *Location.* When a method is well-named, its signature summarizes its intent, while the body implements it using a variety of words that may be unrelated. A query word in the signature is a stronger indicator of relevance than the body.
- *Semantic role.* Prior research has shown that using semantic roles such as action and theme can improve search effectiveness [87]. That intuition is taken further by distinguishing where query words occur in terms of additional semantic roles.
- *Head distance.* The closer a query word occurs to the head, or right-most, position of a phrase, the more strongly the phrase relates to the query word. For example, the phrase "image file" is more relevant to the concept of "saving a file" than "file server manager".
- *Usage.* If a query word frequently occurs throughout the rest of the program, it is not as good at discriminating between relevant and irrelevant results. This idea is commonly used in information retrieval techniques [63].

Individual query words are first scored based on their usage pattern in the code as well as their head distance within a phrase. This score for a phrase is then scaled based on its semantic role, where actions and themes are assigned the highest coefficient multiplier. If a method is difficult for SWUM to split or parse, purely lexical regular expressions are used to calculate the score, scaled by a low coefficient. The score from a method's signature is combined with lexical body information in the final *swum* score.

This *swum* score was compared with the existing FindConcept search results [87], except for 1 concern which was used in *swum*'s training set. The queries for *swum* were formulated by the subjects using the contextual query reformulation technique, as presented in Section 4.3. Two variants of SWUM were compared: *SWUM10*, which uses the top 10 ranked results, and *SWUMT*, which uses a more sophisticated threshold that takes the average of the top 20 results to determine relevance. Depending on how the distribution of scores is skewed, the threshold for SWUMT can be more or less than 10.

Based on the F measure shown in Figure 6, ELex appears inferior to the other search techniques. The SWUM-based techniques, SWUM10 and SWUMT, appear to be more consistently effective than FindConcept or GES. These results are confirmed by the precision and recall results. In terms of precision, SWUMT is a clear front-runner closely followed by FindConcept. For recall, SWUM10, SWUMT, and GES appear to have similar results.

It is not surprising that ELex, the technique with the worst precision, also has the best recall. For most queries in this study, ELex typically returns too many results. This ensures ELex finds many relevant results, but too many irrelevant ones. These observations independently confirm earlier results that used tools similar to ELex for feature location [5]. SWUM10 and SWUMT begin to approach ELex's high recall, without sacrificing precision.

Overall, SWUMT is a very competitive search technique when the query words match relevant signatures. However, when body information is important to locating the concern, GES is the best state of the art technique in this study.

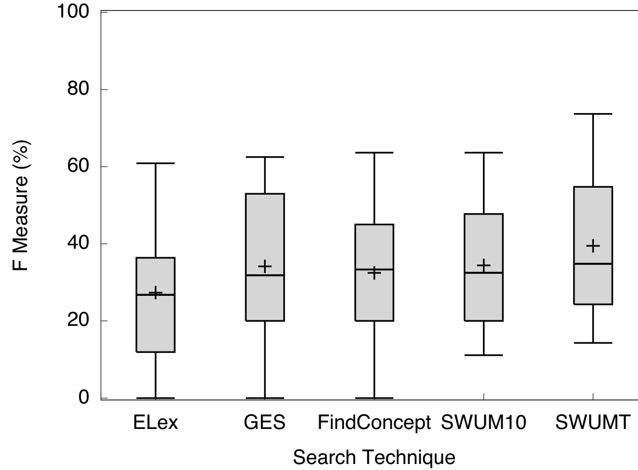

Fig. 6. f-measure results for state of the art search techniques

Although GES outperformed SWUMT, SWUM10, and FindConcept for some of the concerns, its performance in general seems to be unpredictable. When GES did not have the best performance, it tended to be a little better, and sometimes even worse, than ELex. In contrast, even though SWUMT did not always have the best results, it was usually competitive.

To investigate this observation, the approaches were ranked from 1–5 based on their maximum F measure score for each concern, giving ties the same rank. Using this measure, SWUMT is the most highly ranked technique with an average rank of 2.38 and a standard deviation (std) of 1.18. GES has an average of 2.75 (std 1.19), SWUM10 an average of 2.88 (std 1.64), FindConcept an average of 3.00 (std 0.93), and ELex and average of 3.50 (std 1.41). From these results, we can see that SWUMT and GES are the best overall techniques in this study, but that SWUMT is consistently ranked more highly overall.

4.5 Program Exploration

Despite evidence that successful programmers use program structure *as well as* identifier names to explore software [81], most existing program exploration techniques use either structural *or* textual information. Using only one type of information, current automated tools ignore valuable clues about a developer's intentions [7]—clues critical to the human program comprehension process.

By utilizing textual *as well as* structural program information, automatic program exploration tools can potentially mirror how humans attempt to understand code [47]. Combining information enables exploration tools to automatically prune irrelevant structural edges. By eliminating irrelevant edges, ex-

ploration tools can recursively search a structural program representation to provide the maintainer with a broad, high level view of the code relevant to a maintenance task—without including the entire program.

Dora the Program Explorer, or Dora, is an automatic exploration technique that takes as input a natural language query related to the maintenance task and a program structure representation to be explored [36]. Dora then outputs a subset of the program structure relevant to the query, called a *relevant neighborhood*. Dora currently uses the call graph for program structure, and takes a seed method as a starting point. By recursively traversing call edges, Dora identifies the relevant neighborhood for this seed.

Dora uses structural information by traversing structural call edges to find the set of callers and callees for the seed method. These methods become candidates for the relevant neighborhood. Dora uses textual information to score each candidates' relevance to the query. Candidates scored higher than a given threshold, $t_1 = 0.5$, are added to the relevant neighborhood. Candidates scored less than t_1 but more than a threshold $t_2 = 0.3$ are further explored to ensure they are not connected to more relevant methods. The exploration process is recursively repeated for each method added to the relevant neighborhood.

To determine a method's relevance to the query, Dora uses a unique similarity measure that takes into account how frequently the query words occur in the method versus the remainder of the program, as well as where the query words appear. Dora captures word frequency based on the tf-idf score commonly used in information retrieval (IR) [63]. In addition, Dora more highly weights the tf-idf of query words occurring in the method name versus the body. The weights were automatically trained using a logistic regression model on a set of 9 concerns [36].

Dora's sophisticated relevance score ($Dora$) was evaluated against two simpler relevance scores: boolean-AND (AND) and boolean-OR (OR). These techniques output either 0 or 1: AND outputs 1 if *all* query terms appear in the method; OR outputs 1 if *any* query term appears in the method. In addition, Dora was compared to a purely structural technique, $Suade$ [80, 104]. These 4 techniques were compared using 8 concerns from 4 open source Java programs [83]. The 8 concerns contain a total of 160 seed methods and 1885 call edges (with overlap). For each method m in the set of evaluation concerns, each scoring technique was applied to all the callers and callees of m, and the precision and recall for m were calculated.

The results of this study are summarized in Figure 7. Each bar shows the distribution of F measures calculated for each seed method across all the concerns. The shaded box represents 50% of the data, from the 25th to 75th percentiles. The horizontal bar represents the median, and the plus represents the mean.

Since each shaded box extends from 0, at least 25% of the 160 methods considered by each technique have 0% recall and precision. However, $Dora$ achieves 100% precision and recall for 25% of the data—more than any other technique. $Suade$ and OR appear to perform similarly to one another, although OR has a slightly higher mean F measure. These trends were verified with a Bonferroni mean separation test. $Dora$ performs significantly better than structural-based

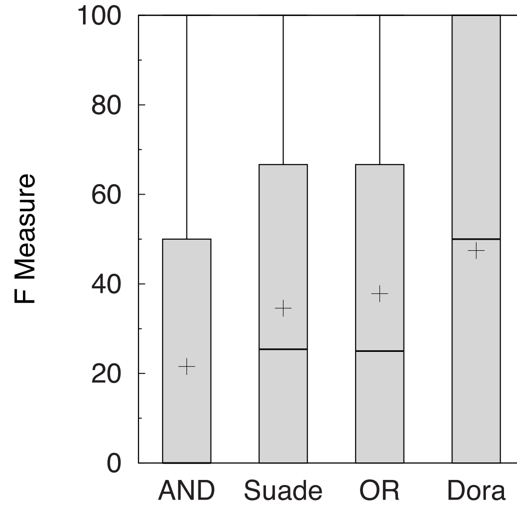

Fig. 7. f-measure across exploration techniques

Suade, although neither *Dora* nor *Suade* are significantly different from *OR*. All the approaches outperform *AND* with statistical significance.

Overall, *Dora* appears to be the most successful technique, and structural-based *Suade* to be competitive with the naive textual- and structural-based *OR*. Of all the techniques, naive *AND* had the worst performance. *AND*'s poor performance indicates that simply combining textual and structural information alone does not guarantee success. The success of a textual- and structural-based (NLPA) technique is highly dependent on the performance of the textual scoring technique.

4.6 Comment Generation

In spite of numerous studies demonstrating the utility of comments for understanding and analyzing software [97–99, 101, 105], few software projects adequately document the code to reduce future maintenance costs [44, 90]. Lack of comments may be fine when programmers use descriptive identifier names [27]; however, precise identifiers that accurately describe an entity lead to very long identifier names [9, 53], which can actually *reduce* code readability, Another way is to encourage the developer to write comments (1) by automatically prompting the developer to enter them [21, 79], or, (2) by using a top-down design paradigm and generating comments directly from the specification [84], or, (3) by using a documentation-first approach to development [45]. Although these solutions can be used to comment newly created systems, they are not suitable for existing legacy systems.

An alternative to developer-written comments is to automatically generate comments directly from the source code [11, 56]. These approaches are limited to inferring documentation for exceptions [11] and generating API function cross-references [56], and are not intended for generating descriptive summary comments.

This section describes how Sridhara, et al. [91, 93, 94] leveraged NLPA, specifically SWUM, to automatically generate method summary comments [91], detect high level actions in method bodies for improved summaries [93], and generate parameter comments and summaries with integrated parameter usage information [94]. The method summaries are leading comments that describe a method's intent, called *descriptive comments*. Descriptive comments summarize the major algorithmic actions of the method, similar to how an abstract provides a summary for a natural language document [61]. Descriptive parameter comments describe the high-level role of a parameter in achieving the computational intent of a method. Figure 8 shows example output from the automatic summary and parameter comment generator.

```
1 public static void main(String[] args) {
2  int port = -1;
3  try {
4      port = Integer.parseInt(args[0]);
5    } catch (ArrayIndexOutOfBoundsException e) {
6     println("Usage: java MetaServer PORT_NUMBER");
7     System.exit(-1);
8    } catch (NumberFormatException e) {
9     println("Usage: java MetaServer PORT_NUMBER");
10     System.exit(-1);
11    }
12    MetaServer metaServer = null;
13    try {
14       metaServer = new MetaServer(port);
15    } catch (IOException e) {
16       logger.warning("Could not create MetaServer!");
17       System.exit(-1);
18    }
19    metaServer.start();
20 }
```

Fig. 8. Example of a generated summary and parameter comment for a Java Method @summary : /** Start meta server */. @param args: create meta server, using args

The key insight in automatic summary comment generation is to model the process after natural language generation, dividing the problem into subproblems of content selection and text generation [77]. *Content selection* involves choosing the important or central code statements within a method that must be included in the summary comment. For a selected code statement, *text generation* determines how to express the content in natural language phrases in a concise yet precise manner. Figure 9 depicts the summary comment generation process with the NLPA preprocessing to build the linguistic and traditional program representations.

Generating Method Summary Comments The first phase selects s_units as content for the summary, where an *s_unit* is a Java statement, except when

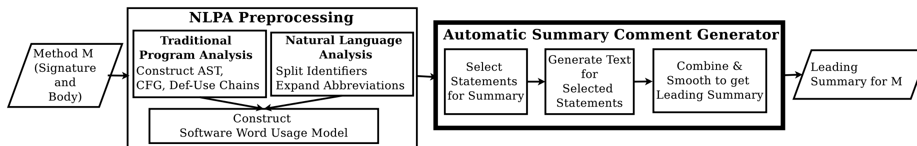

Fig. 9. The Summary Comment Generation Process

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. Heuristics for s_unit selection are based on characteristics similar to beacons [16] for summary comments, where a beacon is a surface feature which facilitates comprehension. Ending s_units are statements that lie at the control exit of a method, as methods often perform a set of actions to accomplish a final action, which is often the main purpose of the method. Same-action s_units are method calls where the callee's name indicates the same action as the method being analyzed for comment generation. A void-return s_unit is a method call that does not return a value or whose return value is not assigned to a variable; these methods often supply useful content for a summary because they are invoked purely for its side effects. These three kinds of s_units are first identified, and then their data-facilitating s_units are identified, which are those s_units that assign data to variables used in these s_units. Any s_units controlling the execution of any of these s_units are included. Finally, ubiquitous operations such as logging or exception handling are filtered out.

The text generation phase determines how to express the selected s_units as English phrases and how to integrate the phrases to mitigate redundancy. For example, for the s_unit:

```
f.getContentPane().add(view.getComponent(), CENTER)
```

The output phrase is:

```
/* Add component of drawing view to content pane of frame*/
```

A naive approach to text generation is to generate a phrase based only on the statement. For example, given `print(current)`; one can generate the phrase "print current". The problem with this approach is that the name of the variable `current` alone is insufficient; the reader is left with no concept of what is being printed. The missing contextual information is `current`'s type, which is `Document`. Instead, a process called *lexicalization* is used, in which the type information of a variable is incorporated such that more descriptive noun phrases are generated for a variable.

Text generation is achieved through a set of templates. Consider a method call `M(...)`. In Java, a method implements an operation and typically begins with a verb phrase [96]. Thus, a verb phrase for `M` is generated. The template for the verb phrase is:

action theme secondary-args
and get return-type [if M returns a value]

where *action, theme* and *secondary arguments* of M are identified by SWUM and correspond to the verb, noun phrase and prepositional phrases of the verb phrase.

Selected s_unit: os.print(msg)
Generated Phrase: /* Print message to output stream */

There are additional templates for different constructs such as *nested method calls, composed method calls, assignments, returns, conditional* and *loop expressions.* The goal in template creation is to be precise while not being too verbose.

```
9  for (int x = 0;  x < vAttacks.size(); x++) {
10  WeaponAttackAction waa=vAttacks.elementAt(x);
11  float fDanger = getExpectedDamage(g, waa);
12  if (fDanger > fHighest) {
13    fHighest = fDanger;
14    waaHighest = waa;
15  }
16 }
17 return waaHighest;
```

Listing 1.1. Lines 9-16 implement a high level action. Synthesized description: “Get weapon attack action object (in vectorAttacks) with highest expected damage.”

Improving Generated Comments More concise and higher level summary comments are achievable if groupings of related statements can be recognized as implementing a higher level action. These same identified high level actions could also be used in *ExtractMethod* refactoring and other applications like traceability recovery and concern location. For example, the code fragment from lines 9 to 16 in Listing 1.1 implements a high level action. Sridhara, et al. [93] leverage SWUM and traditional program analysis to both identify statement groupings that form high level actions, and generate the English phrases to express them.

Similarly, leading comments are improved by parameter comments and/or integrating parameter usage information into the summary comment itself. The challenges are distinguishing the main role the parameter plays among its potentially many uses within the body, expressing that role in English, and then integrating that information into the existing summary comment. Sridhara et al. [94] developed heuristics for identification of the main parameter role using both SWUM and other information such as static estimation of execution frequency [6]. Phrases are generated such that the parameter comment is linked with the summary (i.e., there are overlapping words between the parameter comment and summary). In Figure 8, in addition to using line 4 to generate the parameter comment, line 14 is used to ensure that the parameter comment is *connected* to the summary (via “meta server”).

Evaluating Automatically Generated Comments Sridhara, et al. evaluated their work [91, 93, 94] by obtaining judgements of the generated comments

from expert programmers. For summary comments, a majority of the developers believed that the generated summary was accurate in 87% of the evaluated methods. A majority stated that the summary comments did not miss important information in 75% of the evaluated methods. Finally, the majority noted that the synthesized summary was not too verbose in 87.5% of the evaluated methods. Similarly, the evaluation of the high level action identification and description also yielded positive results [93]. In an evaluation of 75 code fragments with identified high level actions, 192 of the 225 developer responses agreed that the synthesized description represented the high level action in the code fragments. In the evaluation of the generated comments for 33 parameters [94], 89 of the 99 developer responses agreed that the comments were accurate. 89 of the 99 responses also agreed that the comments were useful in understanding the parameter's role.

5 Summary

Results from various empirical evaluations described in this chapter demonstrate that Natural Language Program Analysis can significantly improve the effectiveness of tools to aid in software maintenance. By extracting phrases to represent method signatures, and using the phrases to search for query word instances in the code, developers can gain help in both reformulating search queries and discriminating between relevant and irrelevant search results. Using the Software Word Usage Model (SWUM) can improve scoring functions at the core of software search tools. SWUM has enabled automatic generation of method summary comments with parameter role information.

Experiments have also shown that the usefulness of NLPA in client applications is affected by its accuracy in extracting information from the source code. The software maintenance tools should be improved even further as identifier splitting, abbreviation expansion, part-of-speech tagging, and word relation determiners are improved for the software domain. While much progress in automation has been achieved, the empirical results thus far indicate that there is considerable room for improving all of these building blocks as well as analysis and modeling.

Acknowledgments. The authors greatly benefitted from work together with Zachary Fry, Eric Enslen, Sana Malik, Michelle Allen, and Divya Muppaneni.

References

1. Abadi, A., Nisenson, M., Simionovici, Y.: A Traceability Technique for Specifications. ICPC '08: Proceedings of the 16th IEEE International Conference on Program Comprehension pp. 103–112 (2008)
2. Abebe, S., Haiduc, S., Marcus, A., Tonella, P., Antoniol, G.: Analyzing the Evolution of the Source Code Vocabulary. pp. 189–198 (2009)

3. Antoniol, G., Gueheneuc, Y.G., Merlo, E., Tonella, P.: Mining the Lexicon Used by Programmers during Sofware Evolution. In: IEEE International Conference on Software Maintenance, 2007. pp. 14–23 (2007)
4. Antoniol, G., Canfora, G., Casazza, G., Lucia, A.D., Merlo, E.: Recovering Traceability Links between Code and Documentation. IEEE Transactions on Software Engineering 28(10), 970–983 (2002)
5. Antoniol, G., Gueheneuc, Y.G.: Feature Identification: An Epidemiological Metaphor. IEEE Transactions on Software Engineering 32(9), 627–641 (2006)
6. Ball, T., Larus, J.R.: Branch Prediction for Free. In: PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming Language Design and Implementation. pp. 300–313. ACM Press, New York, NY, USA (1993)
7. Biggerstaff, T.J.: Design Recovery for Maintenance and Reuse. Computer 22(7), 36–49 (1989)
8. Biggerstaff, T.J., Mitbander, B.G., Webster, D.: The Concept Assignment Problem in Program Understanding. In: ICSE '93: Proceedings of the 15th International Conference on Software Engineering. pp. 482–498 (1993)
9. Binkley, D., Lawrie, D., Maex, S., Morrell, C.: Impact of Limited Memory Resources. In: Proceedings of the 16th IEEE International Conference on Program Comprehension. (2008)
10. Booch, G.: Object-oriented Design. Ada Lett. I(3), 64–76 (1982)
11. Buse, R.P., Weimer, W.R.: Automatic Documentation Inference for Exceptions. In: International Symp on Software Testing and Analysis, 2008. pp. 273–282. ACM (2008)
12. Caprile, B., Tonella, P.: Nomen Est Omen: Analyzing the Language of Function Identifiers. In: WCRE '99: Proceedings of the 6th Working Conference on Reverse Engineering. pp. 112–122 (1999)
13. Caprile, B., Tonella, P.: Restructuring Program Identifier Names. In: ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00). p. 97. IEEE Computer Society, Washington, DC, USA (2000)
14. Carroll, J., Briscoe, T.: High Precision Extraction of Grammatical Relations. In: 7th International Workshop on Parsing Technologies (2001), `citeseer.ist.psu.edu/article/carroll01high.html`
15. Chen, K., Rajlich, V.: Case Study of Feature Location Using Dependence Graph. In: IWPC '00: Proceedings of the 8th International Workshop on Program Comprehension. pp. 241–249 (2000)
16. Crosby, M.E., Scholtz, J., Wiedenbeck, S.: The Roles Beacons Play in Comprehension for Novice and Expert Programmers. In: 14th Workshop of the Psychology of Programming Interest Group, Brunel University. pp. 18–21 (2002)
17. De Lucia, A., Oliveto, R., Tortora, G.: Assessing IR-based Traceability Recovery Tools through Controlled Experiments. Empirical Softw. Engg. 14(1), 57–92 (2009)
18. Deissenboeck, F., Pizka, M.: Concise and Consistent Naming. Software Quality Control 14(3), 261–282 (2006)
19. Eaddy, M., Aho, A.V., Antoniol, G., Gueheneuc, Y.G.: Cerberus: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis. In: ICPC '08: Proceedings of the 16th IEEE International Conference on Program Comprehension. IEEE Computer Society, Washington, DC, USA (2008)
20. Enslen, E., Hill, E., Pollock, L., Vijay-Shanker, K.: Mining Source Code to Automatically Split Identifiers for Software Analysis. Proceedings of the 6th Interna-

tional Working Conference on Mining Software Repositories, MSR 2009 0, 71–80 (2009)
21. Erickson, T.E.: An Automated FORTRAN Documenter. In: Proceedings of the 1st annual International Conference on Systems documentation, 1982. pp. 40–45. ACM, New York, NY, USA (1982)
22. Erlikh, L.: Leveraging Legacy System Dollars for E-Business. IT Professional 2(3), 17–23 (2000)
23. Falleri, J.R., Huchard, M., Lafourcade, M., Nebut, C., Prince, V., Dao, M.: Automatic Extraction of a WordNet-Like Identifier Network from Software. In: 18th Int'l Conf. on Program Comprehension. pp. 4 –13. IEEE (2010)
24. Feild, H., Binkley, D., Lawrie, D.: An Empirical Comparison of Techniques for Extracting Concept Abbreviations from Identifiers. In: Proceedings of IASTED International Conference on Software Engineering and Applications (SEA'06) (2006)
25. Feng, F., Croft, W.B.: Probabilistic Techniques for Phrase Extraction. Information Processing and Management 37(2), 199–220 (2001)
26. Fischer, G., Nieper-Lemke, H.: Helgon: Extending the Retrieval by Reformulation Paradigm. In: CHI '89: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. pp. 357–362 (1989)
27. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
28. Fry, Z., Shepherd, D., Hill, E., Pollock, L., Vijay-Shanker, K.: Analysing Source Code: Looking for Useful Verb-Direct Object Pairs in all the Right Places. Software, IET 2(1), 27–36 (2008)
29. Furnas, G.W., Landauer, T.K., Gomez, L.M., Dumais, S.T.: The Vocabulary Problem in Human-System Communication. Communications of the ACM 30(11), 964–971 (1987)
30. Gosling, J., Joy, B., Steele, G.: Java Language Specification. online (September 2006), http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html
31. Haiduc, S., Marcus, A.: On the Use of Domain Terms in Source Code. ICPC '08: Proceedings of the 16th IEEE International Conference on Program Comprehension pp. 113–122 (2008)
32. Hayase, Y., Kashima, Y., Manabe, Y., Inoue, K.: Building Domain Specific Dictionaries of Verb-Object Relation from Source Code. In: 15th European Conference on Software Maintenance and Reengineering (CSMR 2011). pp. 93–100. IEEE Computer Society (2011)
33. Henninger, S.: Using Iterative Refinement to Find Reusable Software. IEEE Software 11(5), 48–59 (1994)
34. Hill, E.: Integrating Natural Language and Program Structure Information to Improve Software Search and Exploration. Ph.D. thesis, University of Delaware (2010)
35. Hill, E., Fry, Z.P., Boyd, H., Sridhara, G., Novikova, Y., Pollock, L., Vijay-Shanker, K.: 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. IEEE Computer Society, Washington, DC, USA (2008)
36. Hill, E., Pollock, L., Vijay-Shanker, K.: 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). pp. 14–23. IEEE Computer Society, Washington, DC, USA (2007)

37. Hill, E., Pollock, L., Vijay-Shanker, K.: 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)
38. Høst, E., Østvold, B.: Canonical Method Names for Java. In: Software Language Engineering, Lecture Notes in Computer Science, vol. 6563, pp. 226–245. Springer Berlin / Heidelberg (2011)
39. Høst, E.W., Østvold, B.M.: 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. pp. 193–202. IEEE Computer Society, Washington, DC, USA (2007)
40. Høst, E.W., Østvold, B.M.: Debugging Method Names. In: ECOOP '09: Proceedings of the 23rd European Conference on Object-Oriented Programming (2009)
41. Høst, E.W., Østvold, B.M.: The Java Programmer's Phrase Book. In: Proceedings of the 1st International Conference on Software Language Engineering. pp. 322–341. Springer-Verlag, Berlin, Heidelberg (2009)
42. IBM: Eclipse IDE. Online (2010), `http://www.eclipse.org`
43. Ishio, T., Niitani, R., Murphy, G.C., Inoue, K.: A Program Slicing Approach for Locating Functional Concerns. Tech. rep., Graduate School of Information Science and Technology, Osaka University (2007), `http://sel.ist.osaka-u.ac.jp/~ishio/TR-slicing2007.pdf`
44. Kajko-Mattsson, M.: A Survey of Documentation Practice within Corrective Maintenance. Empirical Software Engineering 10(1), 31–55 (2005)
45. Knuth, D.E.: Literate Programming. The Computer Journal 27(2), 97–111 (1984)
46. Kuhn, A., Ducasse, S., Gírba, T.: Semantic Clustering: Identifying Topics in Source Code. Information Systems and Technologies 49(3), 230–243 (2007)
47. Lawrance, J., Bellamy, R., Burnett, M., Rector, K.: Using Information Scent to Model the Dynamic Foraging Behavior of Programmers in Maintenance Tasks. In: CHI '08: Proceeding of the twenty-sixth annual SIGCHI Conference on Human Factors in Computing Systems. pp. 1323–1332. ACM, New York, NY, USA (2008)
48. Lawrie, D., Feild, H., Binkley, D.: An Empirical Study of Rules for well-formed Identifiers. Journal of Software Maintenance and Evolution 19(4), 205–229 (2007)
49. Lawrie, D., Feild, H., Binkley, D.: Extracting Meaning from Abbreviated Identifiers. In: SCAM '07: Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007). pp. 213–222 (2007)
50. Lawrie, D., Morrell, C., Feild, H., Binkley, D.: What's in a Name? A Study of Identifiers. In: ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06). pp. 3–12. IEEE Computer Society, Washington, DC, USA (2006)
51. Lawrie, D.J., Binkley, D., Morrell, C.: Normalizing Source Code Vocabulary. In: Working Conference on Reverse Engineering (WCRE). pp. 3–12 (2010)
52. Lawrie, D.J., Feild, H., Binkley, D.: Leveraged Quality Assessment using Information Retrieval Techniques. In: ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06). pp. 149–158. IEEE Computer Society, Washington, DC, USA (2006)
53. Liblit, B., Begel, A., Sweetser, E.: Cognitive Perspectives on the Role of Naming in Computer Programs. In: Proceedings of the 18th Annual Psychology of Programming Workshop (2006)
54. Linstead, E., Bajracharya, S., Ngo, T., Rigor, P., Lopes, C., Baldi, P.: Sourcerer: Mining and searching internet-scale software repositories. Data Mining and Knowledge Discovery 18(2), 300–336 (2009)

55. Linstead, E., Rigor, P., Bajracharya, S., Lopes, C., Baldi, P.: Mining Concepts from Code with Probabilistic Topic Models. In: ASE '07: Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering. pp. 461–464. ACM, New York, NY, USA (2007)
56. Long, F., Wang, X., Cai, Y.: API Hyperlinking via Structural Overlap. In: ACM SIGSOFT Symp on The Foundations of Software Engineering, 2009. ACM (2009)
57. Lukins, S., Kraft, N., Etzkorn, L.: Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation. In: WCRE '08: Proceedings of the 15th Working Conference on Reverse Engineering. pp. 155 –164 (2008)
58. Maarek, Y.S., Berry, D.M., Kaiser, G.E.: An Information Retrieval Approach for Automatically Constructing Software Libraries. IEEE Transactions on Software Engineering 17(8), 800–813 (1991)
59. Madani, N., Guerrouj, L., Penta, M.D., Gueheneuc, Y.G., Antoniol, G.: Recognizing Words from Source Code Identifiers using Speech Recognition Techniques. In: European Conference on Software maintenance and Reengineering (CSMR) (2010)
60. Malik, S.: Parsing Java Method Names for Improved Software Analysis. Tech. rep., University of Delaware (Senior Thesis) (2011)
61. Mani, I.: Automatic Summarization. John Benjamins (2001)
62. Manning, C., Schütze, H.: Foundations of Statistical Natural Language Processing. MIT Press, Cambridge, MA, USA (1999)
63. Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge University Press, New York, NY, USA (2008)
64. Marcus, A., Maletic, J.I.: Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing. In: ICSE '03: Proceedings of the 25th International Conference on Software Engineering. pp. 125–135 (2003)
65. Marcus, A., Poshyvanyk, D.: The Conceptual Cohesion of Classes. In: ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05). pp. 133–142. IEEE Computer Society, Washington, DC, USA (2005)
66. Marcus, A., Sergeyev, A., Rajlich, V., Maletic, J.I.: An Information Retrieval Approach to Concept Location in Source Code. In: WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04). pp. 214–223 (2004)
67. Maskeri, G., Sarkar, S., Heafield, K.: Mining Business Topics in Source Code using Latent Dirichlet Allocation. In: ISEC '08: Proceedings of the 1st India Software Engineering Conference. pp. 113–120 (2008)
68. McMillan, C., Poshyvanyk, D., Revelle, M.: Combining Textual and Structural Analysis of Software Artifacts for Traceability Link Recovery. In: TEFSE '09: Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering. pp. 41–48. IEEE Computer Society, Washington, DC, USA (2009)
69. Michail, A., Notkin, D.: Assessing Software Libraries by Browsing Similar Classes, Functions and Relationships. In: ICSE '99: Proceedings of the 21st International Conference on Software Engineering. pp. 463–472. IEEE Computer Society Press, Los Alamitos, CA, USA (1999)
70. Miller, G.: WordNet: a lexical database for English. In: Communications of the ACM. pp. 39–41 (1995)
71. Ohba, M., Gondow, K.: Toward Mining "Concept Keywords" from Identifiers in Large Software Projects. In: MSR '05: Proceedings of the 2005 International Workshop on Mining Software Repositories. pp. 1–5 (2005)

72. Poshyvanyk, D., Marcus, A.: 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. pp. 37–48. IEEE Computer Society, Washington, DC, USA (2007)
73. Poshyvanyk, D., Marcus, A., Dong, Y.: JIRiSS – an Eclipse Plug-in for Source Code Exploration. In: Proceedings of the 14th International Conference on Program Comprehension (ICPC '06). pp. 252–255 (2006)
74. Poshyvanyk, D., Petrenko, M., Marcus, A., Xie, X., Liu, D.: Source Code Exploration with Google. In: ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06). pp. 334–338 (2006)
75. Rao, S., Kak, A.: Retrieval from Software Libraries for Bug Localization: a Comparative Study of Generic and Composite Text Models. In: Proceeding of the 8th working conference on Mining software repositories. pp. 43–52. MSR '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1985441.1985451
76. Ratanotayanon, S., Sim, S.E., Raycraft, D.J.: Cross-artifact Traceability using Lightweight Links. In: TEFSE '09: Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering. pp. 57–64. IEEE Computer Society, Washington, DC, USA (2009)
77. Reiter, E., Dale, R.: Building Natural Language Generation Systems. Cambridge University Press (2000)
78. Revelle, M., Dit, B., Poshyvanyk, D.: Using Data Fusion and Web Mining to Support Feature Location in Software. In: IEEE 18th International Conference on Program Comprehension (ICPC '10) (2010)
79. Roach, D., Berghel, H., Talburt, J.R.: An Interactive Source Commenter for Prolog Programs. SIGDOC Asterisk J. Comput. Doc. 14(4), 141–145 (1990)
80. Robillard, M.P.: Automatic Generation of Suggestions for Program Investigation. In: ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 11–20 (2005)
81. Robillard, M.P., Coelho, W.: How Effective Developers Investigate Source Code: An Exploratory Study. IEEE Transactions on Software Engineering 30(12), 889–903 (2004)
82. Robillard, M.P., Murphy, G.C.: Concern Graphs: Finding and Describing Concerns using Structural Program Dependencies. In: ICSE '02: Proceedings of the 24th International Conference on Software Engineering. pp. 406–416 (2002)
83. Robillard, M.P., Shepherd, D., Hill, E., Vijay-Shanker, K., Pollock, L.: An Empirical Study of the Concept Assignment Problem. Tech. Rep. SOCS-TR-2007.3, School of Computer Science, McGill University (2007), http://www.cs.mcgill.ca/~martin/concerns/
84. Robillard, P.N.: Schematic pseudocode for program constructs and its computer automation by SCHEMACODE. Commun. ACM 29(11), 1072–1089 (1986)
85. Runeson, P., Alexandersson, M., Nyholm, O.: Detection of Duplicate Defect Reports Using Natural Language Processing. In: ICSE '07: Proceedings of the 29th International Conference on Software Engineering. pp. 499–510. IEEE Computer Society, Washington, DC, USA (2007)
86. Saul, Z.M., Filkov, V., Devanbu, P., Bird, C.: Recommending Random Walks. In: ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. pp. 15–24. ACM Press, New York, NY, USA (2007)

87. Shepherd, D., Fry, Z.P., Hill, E., Pollock, L., Vijay-Shanker, K.: 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)
88. Shepherd, D., Pollock, L., Vijay-Shanker, K.: Towards Supporting on-demand Virtual Remodularization using Program Graphs. In: AOSD '06: Proceedings of the 5th International Conference on Aspect-Oriented Software Development. pp. 3–14 (2006)
89. Sinha, V., Karger, D., Miller, R.: Relo: Helping Users Manage Context during Interactive Exploratory Visualization of Large Codebases. In: Visual Languages and Human-Centric Computing (VL/HCC 2006) (2006)
90. de Souza, S.C.B., Anquetil, N., de Oliveira, K.M.: A Study of the Documentation Essential to Software Maintenance. In: 23rd annual International Conference on Design of communication. pp. 68–75. ACM (2005)
91. Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., Vijay-Shanker, K.: Towards Automatically Generating Summary Comments for Java Methods. In: ASE '10: Proceedings of the 25th IEEE International Conference on Automated Software Engineering (ASE'10). (2010)
92. Sridhara, G., Hill, E., Pollock, L., Vijay-Shanker, K.: Identifying Word Relations in Software: A Comparative Study of Semantic Similarity Tools. In: Proceedings of the 16th IEEE International Conference on Program Comprehension. IEEE (2008)
93. Sridhara, G., Pollock, L., Vijay-Shanker, K.: Automatically Detecting and Describing High Level Actions within Methods. In: ICSE '11 : Proceedings of the 33rd International Conference on Software engineering. pp. 101–110. ICSE '11, ACM, New York, NY, USA (2011)
94. Sridhara, G., Pollock, L., Vijay-Shanker, K.: Generating Parameter Comments and Integrating with Method Summaries. In: International Conference on Program Comprehension (ICPC'11). (2011)
95. Sridharan, M., Fink, S., Bodik, R.: Thin Slicing. In: PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (2007)
96. SUN: How to Write Doc Comments for the Javadoc Tool. online, `http://java.sun.com/j2se/javadoc/writingdoccomments/`
97. Takang, A.A., Grubb, P.A., Macredie, R.D.: The effects of comments and identifier names on program comprehensibility: an experimental investigation. J. Prog. Lang. 4(3), 143–167 (1996)
98. Tan, L., Yuan, D., Krishna, G., Zhou, Y.: /*iComment: Bugs or Bad Comments?*/. In: SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles. pp. 145–158. ACM, New York, NY, USA (2007)
99. Tan, L., Zhou, Y., Padioleau, Y.: aComment: Mining Annotations from Comments and Code to Detect Interrupt Related Concurrency Bugs. In: Proceeding of the 33rd international conference on Software engineering. pp. 11–20. ICSE '11, ACM, New York, NY, USA (2011)
100. Tarr, P., Ossher, H., Harrison, W., Stanley M. Sutton, J.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: ICSE '99: Proceedings of the 21st International Conference on Software Engineering. pp. 107–119. IEEE Computer Society Press, Los Alamitos, CA, USA (1999)
101. Tenny, T.: Program Readability: Procedures Versus Comments. IEEE Trans. Softw. Eng. 14(9), 1271–1279 (1988)

102. Tip, F.: A Survey of Program Slicing Techniques. Journal of Programming Languages 3(3), 121–189 (1995)
103. Wang, X., Lai, G., Liu, C.: Recovering Relationships between Documentation and Source Code based on the Characteristics of Software Engineering. Electron. Notes Theor. Comput. Sci. 243, 121–137 (2009)
104. Warr, F.W., Robillard, M.P.: Suade: Topology-Based Searches for Software Investigation. In: ICSE '07: Proceedings of the 29th International Conference on Software Engineering. pp. 780–783 (2007)
105. Woodfield, S.N., Dunsmore, H.E., Shen, V.Y.: The Effect of Modularization and Comments on Program Comprehension. In: Proceedings of the 5th International Conference on Software Engineering. IEEE Press (1981)
106. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A Brief Survey of Program Slicing. SIGSOFT Software Engineering Notes 30(2), 1–36 (2005)
107. Ying, A.T.T., Tarr, P.L.: Filtering out methods you wish you hadn't navigated. In: Eclipse '07: Proceedings of the 2007 OOPSLA workshop on Eclipse Technology eXchange. pp. 11–15. ACM, New York, NY, USA (2007)