Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns

David Shepherd, Zachary P. Fry, Emily Hill, Lori Pollock, and K. Vijay-Shanker

Computer and Information Sciences
University of Delaware
Newark, Delaware 19716
{shepherd, fry, hill, pollock, vijay}@cis.udel.edu

Abstract

Most current software systems contain undocumented high-level ideas implemented across multiple files and modules. When developers perform program maintenance tasks, they often waste time and effort locating and understanding these scattered concerns. We have developed a semi-automated concern location and comprehension tool, Find-Concept, designed to reduce the time developers spend on maintenance tasks and to increase their confidence in the results of these tasks. Find-Concept is effective because it searches a unique natural language-based representation of source code, uses novel techniques to expand initial queries into more effective queries, and displays search results in an easy-to-comprehend format. We describe the Find-Concept tool, the underlying program analysis, and an experimental study comparing Find-Concept's search effectiveness with two state-of-the-art lexical and information retrieval-based search tools. Across nine action-oriented concern location tasks derived from open source bug reports, our Eclipse-based tool produced more effective queries more consistently than either competing search tool with similar user effort.

Categories and Subject Descriptors: D.2.3 [Software Engineering]: Coding Tools and Techniques

General Terms: Design, Experimentation, Languages

Keywords: Reverse engineering, Program analysis, Feature Location, Remodularization

1. Introduction

Throughout the life cycle of an application, between 60-90% of resources are devoted to modifying the application to meet new requirements and to fix discovered faults [12]. To modify an application, developers must identify the high-level idea, or ***concept***, to be changed and then locate (or find), comprehend, and modify the concept's ***concern***, or implementation, in the code [19]. Because no complete methodologies address the concern location and comprehension problem, more development time is spent reading, locating, and comprehending source code than actually writing code [25]. Therefore, a software development organization could reduce maintenance costs by reducing the difficulty of locating and comprehending concerns.

One of the most common approaches to easing concern location is to group related concerns into modules by *decomposing* a system with respect to a particular type of concept, namely objects. An object-oriented decomposition facilitates locating and comprehending object-oriented concerns because the relevant code is grouped into an object class file instead of disjointed code segments scattered throughout many program files. For example, in the open source juke box application Jajuk [13], program code related to a music player resides in the abstract `Player` object (this includes the core player functionality of playing, pausing, stopping, muting, and seeking). This object-oriented decomposition makes comprehending and maintaining `Player` easier but can cause *action-oriented* concepts, such as "play track", to become scattered across the code base. The research community agrees that object-oriented programming causes certain concerns to become scattered [17, 36], and we argue that many of these scattered concerns are action-oriented because of the natural tension between objects and actions [32]. In addition to the `Player` object, in the Jajuk example, the "play track" concern is scattered in classes representing a track, a music repository view, the concrete player, and a FIFO playlist queue. In this case, the action-oriented concepts are scattered to satisfy the object-oriented decomposition [32].

Aspect-oriented programming (AOP) can help modularize high-level, action-oriented concerns by implementing actions in aspects. However, implementing too many actions as aspects could cause *objects* to become scattered. Imagine refactoring the above "play track" concern into an aspect by removing the `play()` method from the `Player` class—creating a `Player` that has no independent ability to play. Thus, refactoring the scattered concern "play track" into an aspect may not be appropriate. Yet there are many concerns where AOP offers a pleasing alternative to the scattering of action-oriented concerns. To perform (some) refactoring from objects to actions (in aspects) as well as locate and comprehend actions that are not refactored, developers need techniques to locate and comprehend action-oriented concerns.

Locating a concern is commonly called the *concept assignment problem*. In addition to the *concept assignment problem*, we also target concern comprehension. Our strategy for addressing the challenges of the concept assignment problem and concern comprehension is based on a novel approach to processing source code—namely, a hybrid of structural program analysis and natural language processing (NLP) applied to source code. Our approach is search-based with queries performed over a program model that captures the *action-oriented relations between identifiers* in a program. We perform NLP analysis on source code by leveraging the information about occurrences of verbs and their direct objects

AOSD 07 March 12–16, 2007, Vancouver, Canada.

in the program, which are explicitly represented in the program's action-oriented identifier graph model (AOIG) [32] and by performing additional NLP analysis.

The AOIG represents the actions in a program, supplemented with the direct object of each action. Often verbs, such as "remove," act on many different objects in a single program, such as "remove attribute", "remove screen", "remove entry", and "remove template". Therefore, to identify specific actions, the AOIG represents the direct objects of each verb (e.g., the direct object of the phrase "remove the attribute" is "attribute"). We discuss the rationale for extracting this information from programs as well as the general structure of the AOIG in Section 4.1.

In our previous work, we motivated the use of NLP on program source code and presented the action-oriented identifier graph (AOIG), focusing on its definition, an example use, and a construction algorithm. We also suggested applications of the AOIG to demonstrate its usefulness, including a simple filter viewing tool over the AOIG to be used for feature location [32]. This paper focuses on NLP-based source code analysis and structural program analysis using the AOIG to address the challenges of the concept assignment problem. Specifically, this paper provides the following set of ***novel contributions*** to the state of the art beyond our previous work [32]:

- The application of the AOIG program representation to concern location, including
  - *A query expansion algorithm*, with a novel word recommendation algorithm which combines NLP analysis and structural program analysis to assist in expanding user search queries into more effective queries
  - *A result graph construction algorithm* that connects search results via structural links to create an easily comprehendible, graphical concern representation
  - *An implementation of our approach* as an Eclipse plug-in enabling interactive concern location and comprehension with tedious NLP and structural analysis performed automatically
- An evaluation of our approach to locating action-oriented concerns versus a state-of-the-art lexical search tool and a commercial information retrieval tool, including
  - A comparison of each tool's search effectiveness
  - An analysis of the tasks on which our approach performed well, average, and poor
  - A comparison of user effort required to operate each tool

  Although we believe the result graph is more understandable than other tools' result sets (i.e., ranked or un-ranked lists), we leave such an evaluation to future work.

For the software engineer, these contributions translate into (1) decreased time spent locating and comprehending code related to a concept and (2) increased confidence in a code modification due to increased confidence in locating the relevant code with a search.

Our ***experimental evaluation*** investigated which of the state-of-the-art code search tools (Find-Concept and two others) is more effective at locating concerns by forming and executing a query. We asked 18 human subjects to each complete nine concern location tasks using the tools, and we measured the search effectiveness and the required effort for each task. Find-Concept found concerns more effectively and more consistently than either of its competitors across all tasks while requiring similar effort. However, the performance of each tool varied for each concern location task. While Find-Concept performed more effectively or equivalently to its strongest competitor on seven out of nine tasks, it performed less effectively on two out of nine tasks. After a thorough analysis of these two tasks, we concluded that Find-Concept's effectiveness could be improved by enhancing its underlying technology (i.e., the AOIG-Building software). For instance, the AOIG-Building software failed on a few key cases, decreasing Find-Concept's effectiveness. We are currently extending and revising the AOIG-Builder to handle these cases.

In Section 2, we present the state of the art for concern location. We give an overview of our natural language-based search process with an example in Section 3 and provide details of how we combine program structure and natural language analysis in Section 4. We present our evaluation procedure in Section 5 and present the results and analysis in Section 6. Finally, we conclude with a summary in Section 7 and future research directions in Section 8.

2. State of the Art

Locating and understanding a concern, or the *concept assignment problem*, is a fundamental activity that developers, especially software maintainers, must perform often [21, 29]. When software engineers try to locate source code related to a concept, they typically use a variety of ad hoc techniques such as scrolling through files, following call graph links, analyzing dynamic information, or searching files using mechanisms similar to UNIX `grep`. These approaches can be categorized as one of three fundamental approaches: search-based (scrolling and `grep`), program structure navigation (call graph and type hierarchy links), and dynamic approaches.

2.1 Search-based Approaches

In the current literature, there are two types of searches that are used to locate concerns: lexical-based and information retrieval-based. Most searching techniques are evaluated in terms of *precision*, the number of desirable items found divided by the number of actual items found; and *recall*, the number of desirable items found divided by the number of possible desirable items. Here, we discuss both types of search-based approaches, with the goal of high precision and high recall in mind.

2.1.1 Lexical Searches

Programmers commonly use lexical searches to locate concepts in code using regular expression queries. The problem with lexical search tools like `grep` [28] is that regular expression queries are extremely fragile, causing low recall. Expert developers often compensate by searching for overly general terms, leading to large result sets with low precision and no ranking of relevance within the large result sets.

Many natural language features cause regular expression queries to exhibit low recall. Features such as morphology changes, synonyms, line breaks, and reordered terms will cause regular expression queries to fail. For example, a user's search for the concept "find" will fail if he uses a regular expression search for "find" and the concern is implemented using a different morphological form of the word, such as "found". Similarly, a user's search for the concept "remove" will fail if the concept is implemented using the synonym "delete". Also, searches for two interacting words that occur near each other (such as the phrase "find node") will often fail, because line breaks and word ordering changes break most simple regular expression searches. Searches for "find*node" [1] might return false positives, such as "find the term in the node".

The low recall of regular expression queries causes expert developers to broaden their query if an initial query fails to return results.

[1] "*" stands for 0-many characters of any type

However, commonly occurring sub-strings in the code cause even mildly broad search terms to return large result sets, leading to low precision searches. If a user initially searches for the term "prints" and no results are returned, he is likely to broaden his query and search for "print". However, in Java code, the result will include a large number of false positives caused by the commonly used method `System.out.println()`. It is too cumbersome for a developer to read through a low precision result set caused by overly broad lexical searches.

2.1.2 Information Retrieval

Information retrieval (IR) technology uses the frequency of words in documents to determine similarity between documents and queries. Because IR calculates a similarity score, it can rank the results of a query according to relevance. IR also gracefully handles multiple word queries [27]. IR does not, therefore, suffer from all the difficulties of regular expression queries. IR's search queries are not *as fragile* and it does not return un-ranked result sets.

IR technology by itself does not avoid all of the problems that trouble lexical searches. For instance, IR tools rarely handle natural language issues such as morphology, resulting in reduced recall searches. IR technology also returns false positives because it does not account for sentence structure. A search for the concept "play music" using the terms "play" and "music" could return the irrelevant comment "The video should play while the music is silent". Furthermore, IR generally does not account for synonyms, resulting in reduced recall searches (although some approaches attempt to learn related words from word frequency and context information [22]).

In spite of its apparent shortcomings, researchers have successfully applied IR to locate concepts [22, 39] and reconstruct documentation traceability links in source code [1, 20]. We argue that this success is a testament to the wealth of information stored in the identifiers and comments of a code base and that NLP-based techniques can be effective. Using light weight NLP, we can account for morphology, synonyms, and sentence structure, providing opportunities for improved search.

2.2 Program Structure Navigation

Program navigation is one of the more promising approaches to identifying concerns in code. Researchers have proposed following program structure links (e.g., call graph edges, class inheritance links, and class membership links) to discover code related to a concern [8, 30], and also for recommending which link to follow next to identify related code [29].

While program navigation is an excellent technique for refining a set of modules in a mostly discovered concern, it is difficult to discover an entirely new concern. This is due to the large number of links in a program, and the potential for structural disconnect between parts of the same concern [33]. For this reason, we believe that program structure navigation is an excellent complement to our approach and should be used to refine our result sets.

2.3 Dynamic Approaches

Software reconnaissance is a technique for deducing modules that implement a certain feature by analyzing dynamic information [18]. Other approaches to the concept assignment problem have been inspired by this work [7, 11]. Dynamic analysis-based approaches like software reconnaissance usually require test cases that exercise the target concern and those that do not. These test cases are often difficult to construct, especially when a concern is not user-triggerable.

2.4 Other Related Work

Researchers have used natural language processing to identify aspects in requirements, whereas we identify concerns in source code. Baniassad et. al. created one of the first techniques that used NLP to mine for aspects in requirements, the Theme approach, which is semi-automated [2]. Sampaio et. al. later created a technique for *automatically* identifying aspects in requirements [31]. Both works have served as inspiration for our approach, but the nature of analyzing requirements as opposed to source code has led to notably different approaches.

Thorsten [10] and De Volder [38] have both proposed ways of visualizing the history of an exploration through a program as users search for a feature. We believe that visualizing the interactions between modules is important for program understanding, therefore, the ideas presented in Thorsten and De Volder's works have served as inspiration for parts of our tool's design.

2.5 Summary

We hypothesize that search-based approaches to the concept assignment problem are the most appropriate fundamental methodology because they have the potential to locate many points related to a concept with little effort from the software engineer [22]. It should be noted that currently both program structure navigation and dynamic approaches require significantly more effort from the user.

3. Our NLP-Based Search Approach

When using a search-based methodology to solve the concept assignment problem, there are three major challenges: *(1)* the difficulty of mapping from a high-level concept to an appropriate query on the source code, *(2)* an inability to search with high precision and recall, and *(3)* the tedious task of understanding large result sets. Our approach leverages natural language information provided by the AOIG along with program structure information to address these challenges by *(1)* aiding the user in mapping their concept to a concrete query, *(2)* searching over an NLP-based representation of the concerns, and *(3)* presenting the search result set of methods as a graph that shows the relationships between the methods. After describing our search-based process, we discuss how we met the above challenges in Section 3.2.4.

3.1 Overview of Approach

To locate a concern in source code using our approach, as embodied in the Find-Concept tool, the user must take three major steps: initial query formulation, query expansion, and search and result graph inspection.

First the user ***(1) formulates a query*** from their initial concept. OOP style requires programmers to organize their code according to objects (or nouns, using natural language as a metaphor), causing a program's actions (verbs) to become scattered during implementation. Therefore, we primarily search for verbs. Often verbs, such as "remove", act on many different objects in a single program. If the user is interested in a subset of this scattered verb's implementation, he can also specify a direct object, such as "circle" (creating the query "remove circle"). Thus, an initial query in our system consists of a verb and a direct object.

Next, the user ***(2) expands the query*** by examining the recommendations given by the system. The system recommends words related to those already in the query, using its knowledge of natural language as well as how the words are used within the program.

Finally, the user ***(3) searches the AOIG and inspects the result graph*** (a visualization of the query results). Find-Concept constructs the result graph from the search result set by performing

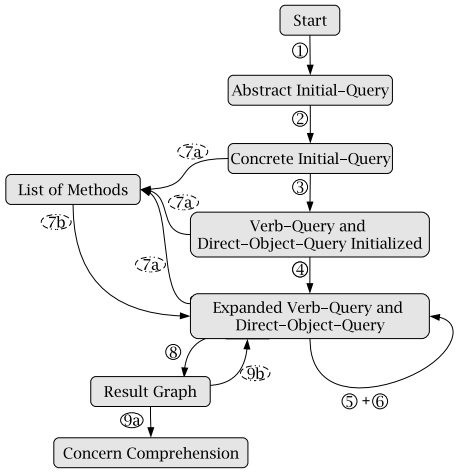

Figure 1. The Find-Concept Process

Step	Set Updates
1	Abstract Initial-Query = "automatically finish the word"
2	Concrete Initial-Query = finish word
3	Verb-Query = finish Direct-Object-Query = word
4	Verb-Query = finish, finished Direct-Object-Query = the word, all words, complete word, first word
5	Verb-Recommendations = complete, end, stop, close, get, ...
6	...user chooses "complete" Verb-Query = finish, finished, complete, completed
5	Direct-Object-Recommendations = completions, text, line, paragraph, ...
6	...user chooses "completions" Direct-Object-Query = the word, all words, ... first word, completions
5	Verb-Recommendations = end, stop, close, get, ...
6	...user decides not to add any words

Figure 2. Evolution of query and lists in example

program analysis to discover the structural relationships between methods. The user can browse the result graph and directly access related source code by clicking on a node, which helps the user understand the concern.

Larger Context of Approach: Because any code search tool is unlikely to achieve 100% accuracy, developers should use search tools in conjunction with program navigation tools. Code search tools provide the "seeds", or starting methods, for program navigation tools, which are more accurate, but much more time consuming. When developers use code search tools (a seed-finder) in conjunction with program navigation tools (a seed-expander), even a small improvement in the code search tool's accuracy speeds up the overall process significantly, because the developer has to complete less of the task using the time consuming program navigation tool.

3.2 The Find-Concept Process

Find-Concept takes as input a target concept, and after interaction with the user, outputs a search result graph for the concept. Figure 1 provides an overview of the Find-Concept process. In Figure 1, each node is a state in the Find-Concept process and each edge is labeled with the corresponding step in our running example, presented below. In this example, we consider the problem of locating the concern 'automatically finish the word' in a text editing program. This feature allows users to press a combination of keys to automatically finish a partially completed word.

3.2.1 Forming the Initial Query

Step 1 - Formulate Abstract Query: The user formulates an abstract Initial-Query for the target concept. In our example, the Initial-Query is based on the idea 'automatically finish the word'.

Step 2 - Formulate Concrete Query: The user decomposes this idea into a concrete Initial-Query consisting of a verb V and direct object DO, finish and word. We chose the term finish instead of complete in this example to highlight how Find-Concept handles even naive user input well.

Step 3 - Input Query: Find-Concept maintains both a Verb-Query and Direct-Object-Query (see Figure 2), which are initialized by the user inputting the Initial-Query consisting of V and DO. In this case, the Verb-Query is initialized to {finish} and the Direct-Object-Query is initialized to {word}.

Step 4 - Initial Query Expansion: When the user enters the Initial-Query, Find-Concept recommends adding *different forms* of V and DO to Verb-Query and Direct-Object-Query, respectively. The user chooses to accept or omit these added words for Verb-Query and Direct-Object-Query. In this example, Find-Concept suggests that the user add the word finished to Verb-Query and direct objects, such as the word and all words, to Direct-Object-Query. After the user enters the Initial-Query and accepts Find-Concept's recommendations, the initial Verb-Query consists of {finish, finished} and the initial Direct-Object-Query is {the word, all words, complete word, first word}, as in Figure 2.

3.2.2 Query Expansion

The Initial-Query consisting of Verb-Query and Direct-Object-Query is expanded to a complete query by incrementally creating a complete Verb-Query and a complete Direct-Object-Query through the user's selections of *automatic word suggestions* (Steps 5-7).

Step 5 - Generate Recommendation Lists: The user begins to expand the Verb-Query by having Find-Concept make a ranked, related Verb-Recommendations list for the Verb-Query. Next, the user examines the Verb-Recommendations. Find-Concept only presents the top 10 ranked recommendations. Words that Find-Concept estimates are most relevant to the current query appear near the beginning of Verb-Recommendations, and less relevant words appear towards the end of Verb-Recommendations. The user begins to expand the Direct-Object-Query in the same manner. The recommendations for this example are presented in Figure 2.

Step 6 - Examine Recommendation Lists and Choose Recommendations: The user examines the Verb-Recommendations {complete, end, stop, close, get, ...} as in Figure 2. Notice that since the Verb-Query is {finish, finished}, Find-Concept recommends *synonyms* of finish that appear in the program. Since the Direct-Object-Query is {word, all words,...} and complete appears with word in the AOIG, complete is recommended first. The user examines the recommendations and quickly determines that the word complete is related to the concept because complete is ranked first, and adds complete to Verb-Query. Then, the user examines the ranked Direct-Object-Recommendations {completions, text, line, paragraph, ...}. From the Direct-Object-Recommendations, the user decides to add {completion, completions} to Direct-Object-Query. As the user adds words to the Verb-Query or the Direct-Object-Query, Find-Concept suggests adding all forms of an added word (e.g., if "complete" is added then Find-Concept will recommend adding "completing" and "completes").

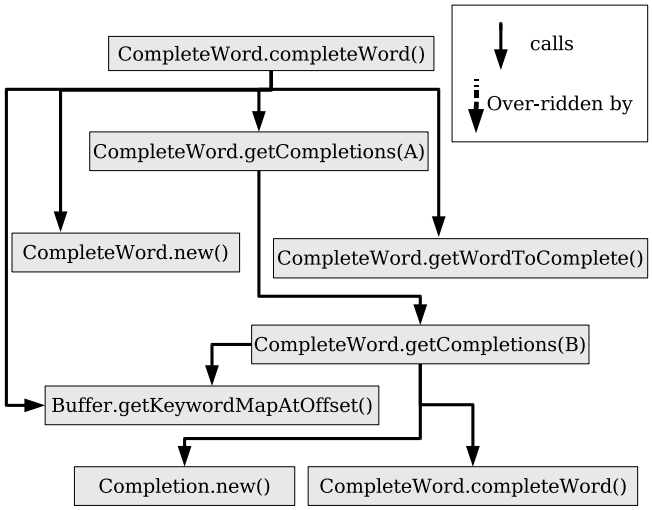

Figure 3. "complete word"'s result graph. This figure is virtually identical to the result graph Find-Concept presents to the user—the only difference is that we reduced the horizontal spacing to conform to this document's two-column format.

Repeat Steps 5-6 - Continue Expansion Process Until Satisfied: The user repeats steps 5-6 until satisfied with the expanded query consisting of Verb-Query and Direct-Object-Query. After adding a word to Verb-Query or Direct-Object-Query, the Verb-Recommendations and the Direct-Object-Recommendations are likely to change since they are calculated using information from both Verb-Query and Direct-Object-Query. Therefore in this example, after updating Direct-Object-Query, the user again checks the Verb-Recommendations { end, stop, close, get,...}, but decides not to add any more words because none of the recommendations appear to be relevant. The user then checks the Direct-Object-Recommendations and also decides not to add more words, leaving the Verb-Query and Direct-Object-Query as they stand in Figure 2.

Step 7a - Optional Feedback: At any point during the query formulation or expansion process, the user can inspect the list of methods that match the current query. The size of this list helps the user decide when to stop expanding his query. For instance, if the list of methods after Step 4 in Figure 2 contains no members, the user will likely continue expanding his query.

Step 7b - Optional Manual Additions: Occasionally, for difficult instances of the concern location problem, the user might have few relevant search results even after expanding the query several times. To look for possible additions to the query, the user should inspect the source code of the list of methods generated in Step 7a. There, he will often discover a term to add to the original query that was not included on a recommendation list. This usually happens when a term is extremely misspelled (e.g., spelling "complete" as "cmpleet") or when a developer uses a word as a synonym that is not actually a synonym (e.g., using "complete" and "append" as synonyms) because these cases cause Find-Concept's recommendation algorithm to perform sub-optimally. However, even in these extreme cases, the Find-Concept recommendation algorithm can perform well because it uses complementary clues.

3.2.3 Search and Display Result Graph

Step 8 - Generate Result Graph: The end of a user's possibly iterative, interactive session of query expansion triggers a search over the AOIG using the developed query. Find-Concept analyzes the results and generates the results set in an easily understood form, called the *result graph*. The result graph consists of nodes that represent methods and edges that represent structural relationships. For this example, the result graph for the query finish word is shown in Figure 3.

Step 9a - Examine Result Graph: The user would most likely inspect the result graph by viewing the largest connected subgraph, top-down. The user can then determine whether a node represents part of the concern by browsing the linked source code or by simply examining the node's name and neighbors in the program's structural graph. In this example, the graph's root node is method `CompleteWord.completeWord()`, as shown by the result graph in Figure 3. By viewing the source code for `CompleteWord.completeWord()` (triggered by clicking on the associated node in the displayed result graph) as well as its child nodes, the user realizes this method constitutes the core of finish word. Given the result graph for the initial query, whose nodes are tightly linked with source code, it is easy to understand this concern.

Step 9b - Optionally Explore Result Graph for Additional Terms: For very challenging instances of the concept assignment problem, the user may opt to follow an extra step. If a connected subgraph appears incomplete, users can explore the source code associated with the subgraph, using structural navigation, to identify additional terms to expand their query. The user then repeats steps 4-9.

3.2.4 Discussion

In our example, the user was faced with many of the typical challenges that occur during search-based concern location tasks. Find-Concept is well-suited to deal with these difficulties, as seen when we look more closely at the following details.

Find-Concept addresses the challenge of creating effective queries by expanding users' initial queries. The user's initial query of "finish word" is ineffective, returning no results. However, the user interacted with Find-Concept to expand the query via synonyms to add "complete", and then via morphological relationships to add "completing" and "words", which leads to a more effective query.

Find-Concept addresses the goal of searching with high precision and recall by assisting the user in expanding queries as well as searching over the AOIG instead of the program text. The query expansion (i.e., "finish word" → "finish, finished, complete, completing, word, words") increases the recall of the search. A lexical or IR-based search for finish word is difficult because the string complet (the beginning of complete, completing, etc.) appears 182 times in this code, mainly in other contexts unrelated to the concern finish word.

Find-Concept searches with high precision by returning only procedures where both the verb "complete" and the direct object "word" appear. When the word complete is used with another direct object, such as complete choices, then complete appears in the code but is unrelated to finish word. Similarly, when the string complete appears, but it is not used as a verb (e.g., search complete graph), then complete appears in code but is unrelated to finish word. Find-Concept assures that this search has high recall by including all morphological forms of "complete" and "word" in its search.

Lastly, Find-Concept presents our search results in a meaningful and understandable way to the developer by displaying the query results as a graph. Once found, the finish word concern is difficult to understand because the concern is implemented in three different classes: CompleteWord, Buffer, and Completion (see Figure 3). It is difficult to view all of the concern's code segments and inter-relationships at once in a textual view. Find-Concept visualizes the identified concern by showing all code segments as well as their inter-relationships in graph form, thus making the concern easier to understand.

4. Underlying Program Analysis

To enable the natural language-based search process, Find-Concept first builds a natural language representation of source code (described in Section 4.1). Throughout the process of building an effective query, Find-Concept uses the algorithm in Section 4.2 to recommend possible query expansions. Finally, once Find-Concept searches the AOIG using the developed query, Find-Concept analyzes the results and computes the result graph using the algorithm described in Section 4.3.

4.1 Extracting Verb-DO Information

Both the word recommendation for query expansion and the search process utilize NLP in conjunction with an underlying Action-Oriented Identifier Graph (AOIG) [32] to reconnect the scattered actions of an OOP system. The AOIG is well-suited for use in Find-Concept because it focuses on the actions of the program, and Find--Concept's intended use is to locate action-oriented concerns. In a programming language, verbs correspond to actions (or operations) and nouns correspond to objects [4]. Similarly, the Java Language Specification recommends that "method names should be verbs or verb phrases..." and "names of class types should be descriptive nouns or noun phrases" [14].

It is important to consider the *theme* to precisely identify a specific action. 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 [6]. An example is (parked, car) in the sentence "The person parked the *car*."

An algorithm to construct the AOIG for a program is the focus of a previous paper [32]. We highlight the intuitive definition and example here. The AOIG model explicitly represents the occurrences of verbs and direct objects (DOs) in a program, as implied by the usage of user-defined identifiers. We currently only analyze occurrences of verbs and DOs in method declarations and comments, string literals, and local variable names within or referring to method declarations, because method declarations are the core of concerns. We map each verb-DO pair to all its occurrences in the source code.

An AOIG representation of a program contains four kinds of nodes: a *verb node* for each distinct verb in the program, a *direct object (DO) node* for each unique direct object in the program, a *verb-DO node* for each verb-DO pair identified in the program, and a *use node* for each occurrence of a verb-DO pair in a program's comments or code. A verb-DO pair is defined to be two co-located identifiers in which the first identifier is an action or verb, and the second identifier is being used as a direct object for the first identifier's action.

An AOIG has two kinds of edges: pairing edges and use edges. There exists a *pairing edge* from a verb *v* or DO *do* node to a verb-DO node when *v* and *do* are used together, i.e., appear in the same sentence or phrase and interact, as (jump, hurdle) do in the phrase "jump the hurdle". For each use of a verb-DO pair in the program, there exists a *use edge* in the AOIG mapping the verb-DO node to the corresponding use node. Although a verb (or DO) node may have edges to multiple verb-DO nodes, a verb-DO node has only two incoming edges: one from the verb and one from the DO node involved in the relation.

Figure 4 shows the form of an AOIG. In this figure, we can see that `verb1` has two *pairing edges*, one to <`verb1, DO1`> and one to <`verb1, DO2`>, which are both *verb-DO nodes*. <`verb1, DO1`> has two *use edges*, which represent locations in the source code where this pair occurs. In previous work, we described how to build the AOIG with reasonable time and space costs using open-source NLP components and open-source Java analysis components to extract the verb-DO information from both comments and method signatures [32].

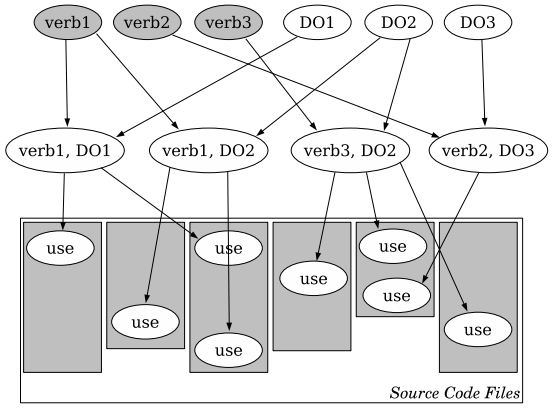

Figure 4. Example of an AOIG

4.2 Word Recommendation

An important contribution of our Find-Concept search process is word recommendation. The following relationships are used to construct a ranking of recommended words `w` for addition to Verb-Query, which are stored and presented to the user as Verb-Recommendations. Note that words related to the current Verb-Query and Direct-Object-Query are both considered when constructing Verb-Recommendations. To construct Direct-Object-Recommendations, a similar set of relationships is considered, but with all Verb-Query and Verb-Recommendations references replaced with Direct-Object-Query and Direct-Object-Recommendations, respectively, and all Direct-Object-Query references replaced with Verb-Query.

• **Similar Semantics** - If the word `w` is semantically related to a word in Verb-Query or Direct-Object-Query, `w` is added to Verb-Recommendations:

- **Stemmed/Rooted Matching** - If the word `w` has the same stem/root as a word in Verb-Query or Direct-Object-Query (e.g., finished and finishing have the same stem), `w` is added to Verb-Recommendations.
- **Synonym Matching** - If the word `w` is a synonym of a word in Verb-Query or Direct-Object-Query (e.g., finish and complete are synonyms), `w` is added to Verb-Recommendations.

• **Similar Use** - If the word `w` is co-located in the AOIG with a word which appears in Direct-Object-Query (i.e., appear in a verb-DO node together), `w` is added to Verb-Recommendations. For example, if Direct-Object-Query contains word, and word is co-located with complete in the AOIG, then complete is potentially relevant and added to Verb-Recommendations.

If the word `w` is co-located with several words in Direct-Object-Query, then `w` will be added to Verb-Recommendations several times, which effectively increases `w`'s ranking in Verb-Recommendations. The rationale is that a word that interacts with several confirmed words of interest is often related.

4.2.1 Word Recommendation Algorithm

Figure 5 presents the algorithm for creating Verb-Recommendations—the ranked recommendations of highly-related words to Verb-Query based on these relationships. Given a word set, our new word recommendation algorithm builds a ranked list of highly-related words, using the AOIG model of the program and addi-

Input: Verb-Query , Direct-Object-Query, AOIG
Output: a ranked list of recommendations, Verb-Recommendations
1: For all words w in (Verb-Query ∪ Direct-Object-Query)
2: $Synonyms$ = getSynonyms(w)
3: $AllForms$ = getAllFormsOfWord(w)
4: $AllSimilar = Synonyms \cup AllForms$
5: For all words w_s in $AllSimilar$
6: if(AOIG contains w_s)
7: weight(w_s) = syn
8: add w_s to Verb-Recommendations
9: For all words w in Verb-Query
10: $Uses$ = getCoLocatedWords(w, AOIG)
11: For all words w_u in $Uses$
12: if(Verb-Recommendations contains w_u)
13: weight(w_u) = weight(w_u) + use
14: else weight(w_u) = use
15: add w_u to Verb-Recommendations
16:Sort Verb-Recommendations by decreasing weight

Figure 5. Word Recommendation Algorithm

Input: Verb-Query, Direct-Object-Query, AOIG, $database(P)$
Note: $database(P)$ is a database of program structure facts.
Output: Result Graph RG
1: Verb-Query = Verb-Query ∪ {get, set, execute, construct}
2: $methods[]$ = identify_verbDOpairs(Verb-Query, Direct-Object-Query, AOIG)
3: $HashSetOfMethods$ = new HashSet($methods$)
4: RG = empty graph
5: For each method i ∈ $methods[]$
6: $edges[]$ = getEdges($methods[i], database(P)$)
7: For each edge k ∈ $edges[]$
8: if($HashSetOfMethods$ contains target($edges[k]$)
9: addEdge($RG, edges[k]$)

Figure 6. Result Graph Construction Algorithm

tional NLP analysis on the source program. The algorithm takes as input a Verb-Query, a Direct-Object-Query, and the AOIG representation of the source program. The terms syn and use designate configurable weights that change the relative importance of the synonym (syn) and similar use (use) relationships just described. From our initial studies and hand tuning of parameters, we consider syn more important than use by a factor of two.

Consider a situation where Verb-Query consists of the term complete and Direct-Object-Query consists of the term word, as in the word editing program in Section 3.2. Looking for recommendations for Direct-Object-Query, we would expect Direct-Object-Recommendations to be similar to {completions, finisher, lexeme, choice}. The term completions is ranked highest because it is a synonym of a word in Verb-Query (complete) and appears in the AOIG paired with a word in the Verb-Query (complete again). This causes completions to be added to Direct-Object-Recommendations on line 8 in the algorithm in Figure 5 and its weight to be increased further on line 13. The term finisher is ranked highly because it is a synonym of a word in Verb-Query, namely complete, causing it to be added to Direct-Object-Recommendations on line 8. lexeme is also ranked highly because it is a synonym of a word in the Direct-Object-Query, namely word. Finally, choice is added to Direct-Object-Recommendations on line 15, even though it is ranked lowest. choice is ranked lowest because, although it appears often in code with the verb complete, it does not have a synonym relationship with any word in either Verb-Query or Direct-Object-Query.

4.2.2 Runtime Analysis

The word recommendation algorithm's runtime is O($w * \max(s, c)$), where w is the number of words in Verb-Query, c is the size of the largest set of co-located words, and s is the size of the largest set of synonyms for a word contained in Verb-Query or Direct-Object-Query. However, in practice, both c and s are usually small (less than 10). In the context of Find-Concept, we have found the word recommendation algorithm to perform quickly, presenting little to no noticeable delay for the user.

4.3 Computing the Result Graph

Figure 6 presents the algorithm for the actual search and the construction of the search result graph. When the user is satisfied with the query expansion, i.e., the current Verb-Query and Direct-Object-Query, Find-Concept identifies all verb-DO nodes in the AOIG model of the program that involve both a verb from Verb-Query and a direct object from Direct-Object-Query. From experience, we have found that the search algorithm works better if Verb-Query is first augmented with connecting verbs, such as get, set, execute, and construct (line 1). If the Verb-Query starts as {complete, completes, completed}, it becomes {complete, completes, completed, get, set, execute, construct}.

If the Direct-Object-Query is {word, completions, keyword }, the algorithm finds all verb-DO pairs (and corresponding methods) that are composed of a word from each set, such as complete keyword or get completions (line 2). Find-Concept follows the use edges of the AOIG to map these verb-DO pairs to the methods where they exist in the program. By performing the search over the AOIG's use edges, Find-Concept returns search results that are critical to the understanding of the original verb-DO pair's implementation and, furthermore, enable return of a small set of methods very relevant to the concern.

Given the set of methods M with the verb-DO pairs involving both Verb-Query and Direct-Object-Query, Find-Concept uses structural program analysis to determine all edges in the call graph, class hierarchy graph, or precondition graph that connect any of these methods in M to each other. In Figure 6, we refer to these structural facts as the database of program structure facts. Find--Concept presents a visualization of the result graph consisting of all the methods in M and any structural links connecting the methods in M, as the search result. The user can then examine the result graph and click a node to view the associated code.

In Figure 6, the function identify_verbDOpairs(), which is a search function over the AOIG, runs in O($largestSet$), where $largestSet$ is the size of the largest of Verb-Query and Direct-Object-Query sets. The result graph construction time is O(me), where m is the size of the set of methods in M and e is the largest number of program structure edges attached to one method. In practice, e is usually small (less than 5). The size of M is dependent on the verb-DO pair uses in the source program, which is dependent on the target concept and its implementation in the source program. Typically, the size of M is considerably smaller than the number of methods in the application. In the context of Find-Concept, computing the result graph takes the most time, but we attribute this to our prototype implementation and not to fundamentally high runtime costs.

Application Name	NCLOC	No. Methods	% Code Commented	No. Bugs (Open/Total)
jBidWatcher	23,179	1,571	37.34	213/1,024
javaHMO	23,797	1,532	28.24	75/185
Jajuk	30,847	1,586	54.24	1/193
iReport	74,392	4,364	27.76	59/65

Table 1. Subject Applications' Characteristics

5. Evaluation Methodology

To validate our ideas, we implemented our technique as an Eclipse plugin, named Find-Concept, and compared Find-Concept to Eclipse's built-in lexical search (ELex) [16] and a modified Google Eclipse search (GES) [27]. We focused our evaluation on the following research questions:

RQ1 Which search tool (Find-Concept, GES, or ELex) is most effective at locating concerns by forming and executing a query?

RQ2 Which search tool requires the least human subject effort to form an effective query?

While we believe that the result graph provides a result set format that is easier to understand than other tools' formats, we leave an evaluation of the result graph's understandability for future work.

5.1 Independent Variables

Because Find-Concept's purpose is to help developers complete maintenance and evolution tasks, evaluating Find-Concept's query expansion mechanism and search over the AOIG requires studying actual human subjects. Therefore, we designed our experiment to manipulate three independent variables: search tools, search tasks, and human subjects.

5.1.1 Search Tools

Because we believe that search-based approaches to concern location are faster than navigation-based approaches (see Section 1), we focused our evaluation on search-based approaches. We chose one search tool from each of the following state-of-the-art program search technology categories: lexical-based (ELex), IR-based (GES), and NLP-based (Find-Concept).

Lexical Search. ELex allows users to search using a regular expression query over source code files of a given project, returning an unranked list of files that match the query. The results are ordered alphabetically by package name and then file name. ELex's functionality is similar to `grep`'s functionality, and ELex is a good representative of the state of the art in lexical searches.

Google Eclipse Search. GES integrates Google Desktop Search into the Eclipse workbench, allowing users to search Java files with IR-style queries (i.e., a set of words) and returning a set of ranked files. We altered GES slightly to return individual procedures instead of entire files, which is more appropriate for the given tasks. Although Google's exact IR algorithm is proprietary, the authors of GES claim that Google Desktop provides an accurate, IR-based search and does not suffer from inefficient queries or inefficient re-indexing of files during evolution, which provides an advantage over their previous prototypes which used latent semantic analysis [27].

Find-Concept. To implement Find-Concept, we used many components, related to both NLP and structural program analysis. For NLP, we used the Porter Stemmer [26] to perform stemming to recognize different forms of the same word, Maxent in OpenNLP [24] for part-of-speech tagging to determine the root of direct-object phrases, and a stand-alone, optimized version of WordNet [35] as a synonym finder. We used Eclipse's JDT [9] to perform quick parsing of the source code, and a database of program analysis information constructed by Eclipse's JDT to quickly access the call graph and the type hierarchy graph [29]. We used Grappa [3] to enable visualization of the result graph and modified Grappa to support one-click access from the result graph to the corresponding source code.

Comparing Heterogeneous Results. To evaluate precision and recall fairly for tools with different forms of output, we converted each tool's output into a list of methods. For ELex, we returned the entire result set because each result was an occurrence of the search expression, and there is no metric for ranking a single occurrence higher than another. For GES, we returned the top ten results, ranking results by the number of search terms they contained. Returning more than ten results would unfairly skew GES's precision lower than the other tools. For Find-Concept, we returned the top ten results. Although Find-Concept does not naturally rank results, we ranked methods by the prominence of their placement in the result graph using the *connectedness-factor*. For a given node n, the *connectedness-factor* is the number of nodes in n's connected sub-graph added to the sum of n's outgoing and incoming edges. This heuristic approximates the visual prominence of n in the result graph.

5.1.2 Search Tasks

To compare the search results of different tools, we first identified a set of search tasks. A **search task** consists of an *application* and a *concept*. We identified four active, sizable, open-source Java applications with available bug report repositories. In an effort to identify realistic concepts, we extracted commonly occurring concepts from bug reports. We then identified the concern for each concept in the source.

Subject Applications. We identified open-source Java applications that met the following constraints: at least 25 KLOC, at least 50 available bug reports, and at least 1000 methods. *JbidWatcher* is an online auction monitoring and sniping application. *Jajuk* is a multi-platform music jukebox designed for users with large music collections. *IReport* is a report generation and editing application. *JavaHMO* is an expansion of the TiVo Home Media Option software that allows users to access and display different types of media.

We also identified a text messenger application for use in training, *PlanetaMessenger* [5]. Because this application was only used for training the human subjects, it was not subjected to the same constraints as the subject applications. *PlanetaMessenger* meets our size and procedure count constraints, but has less than 50 bug reports.

Relevant characteristics of our four subject applications are presented in Table 1. We calculated these characteristics using the RefactorIt plug-in [34] and by consulting Sourceforge's project statistics [15]. The '% Code Commented' was calculated by dividing the number of commented lines of code by the non-commented lines of code (NCLOC).

Concepts. From these four subject applications and the training application, we identified eleven action-oriented concepts – nine as search targets and two as training tasks. Because we presented the concepts to our human subjects in a visual form (as explained in Section 5.3 under *Subjects' Tasks*), we only selected user-observable concepts [23] for the experiment. To ensure the concepts in our experiment were realistic, we searched through bug reports for each application. For each application, we started with the most recent bugs in the database, identified the concept(s) for each bug report, and stopped when we identified two to three

Task Name	Application	Description	Gold Set Size	Textual Clues
Add Textfield	iReport	Insert a textfield into a report	5	Part
Compile Report	iReport	Compile the source of a report into the final report	8	Full
Add Auction	jBidWatcher	Add an auction to the local list of auctions to monitor	10	Full
Set Snipe	jBidWatcher	Set the price for the program to automatically bid on the user's behalf at a specified time	12	Part
Save Auctions	jBidWatcher	Save the list of auctions that the user is currently monitoring	9	Full
Gather Music Files	javaHMO	Inspect the local hard-drive for music files to automatically add to the music library	4	None
Load Movie Listings	javaHMO	Download movie listings from a remote movie listings service	5	none
Search for Songs	Jajuk	Search the user's music library for a track using regular expressions	5	none
Play Track	Jajuk	Play an audio file	12	Part

Table 2. Search Tasks

concepts that were associated with several bug reports. To ensure the tools were used evenly by all human subjects, we selected an additional concern from jBidWatcher because it had the most bug reports—for a total of nine concerns. Thus, we focused the study on concerns that actual developers need to search for to fix bugs. A summary of the concepts we selected is in Table 2.

5.1.3 Human Subjects

To make our study's findings relevant to the practice of software engineering, we sought out human subjects who program on a regular basis. We used 13 full-time professional Java developers as well as five graduate students (all in systems research), for a total of 18 subjects. Although the subjects were a mix of advanced and intermediate programmers, the developers are typically required to use search tools for maintenance tasks on a daily basis.

5.2 Dependent Variables and Measures

To answer RQ1 and RQ2, we measured the effectiveness of each tool's query formulation and execution (RQ1) and human subject effort (RQ2).

5.2.1 Effectiveness

To measure the effectiveness of a search, $precision$ and $recall$ are often used. *Precision* is the ratio of the number of *relevant* procedures retrieved to the total number of procedures retrieved. *Recall* is the ratio of relevant procedures retrieved to the total number of relevant procedures existing in the source application. High precision means the result set contains few irrelevant results, whereas high recall implies most of the relevant results are included in the result set.

A good search result set requires both precision and recall to be high, and the quality of the result is bounded by the lower measure. To combine these measures, we use the ***f-measure***, which is commonly used to evaluate query effectiveness in IR [37]. The ***f-measure*** is defined to be the harmonic mean of precision and recall, calculated as $(2{*}precision{*}recall)/(precision{+}recall)$. The f-measure is better suited than more simplistic combination techniques like averaging because it weighs the lower measure more heavily. For instance, a query with 90% precision but only 10% recall returns few results, but most results are relevant. The f-measure for this case is 18%, whereas the average of precision and recall is 50%. The f-measure (18%) better reflects the query's effectiveness, since the query only locates 10% of the overall target concern.

To calculate precision and recall, we need to identify the "gold set" of methods that represent each concern. However, we are aware of no benchmark concerns in the community or a rigorous definition of concerns. Thus, we used our intuitive definition of a concern (the implementation of a high-level concept) and a human's ability to interpret and apply this definition to locate the gold set. To provide low bias, we recruited a new group member, unfamiliar with our previous and current work, to identify concerns. For each concept, we asked him to locate the code that implements it. He was given access to the application's code base (which he was unfamiliar with) and all of Eclipse's typical functionality [16]. After he had identified a set of methods, the first author, who was also unfamiliar with the applications, verified this set. The two then discussed and reconciled the few methods they disagreed upon. More than 90% of each identified gold set was located by the new project member alone. To enable replication of this experiment and to offer concern benchmarks to the community, we provide a list of the methods in each concept's gold set[2]. Table 2 displays the size of each gold set. The "Textual Clues" column documents the level of search term clues contained in the application's GUI, as explained in Section 5.3.

5.2.2 Effort

To evaluate human subject effort, we measured the amount of time each subject required to form a satisfactory query for each task. We began timing when the subject claimed to understand the concept visually presented and stopped when the subject was satisfied with his query.

5.3 Experimental Procedure

Each of the 18 human subjects completed 9 tasks. Subjects were assigned one search tool for each task such that every subject used each tool three times. We randomized the task and tool order to avoid biasing our results. With one exception, each tool was used three times consecutively to avoid human subject confusion from switching between tools. An example schedule for three human subjects is presented below.

		Execution Slots								
Subject 1	Task	1	3	6	2	8	5	9	7	4
	Tool	2	2	2	3	3	3	1	1	1
Subject 2	Task	1	3	6	2	8	5	9	7	4
	Tool	3	3	3	1	1	1	2	2	2
Subject 3	Task	1	3	6	2	8	5	9	7	4
	Tool	1	1	1	2	2	2	3	3	3

Note that each row only covers nine out of the possible 27 Task-Tool combinations (Task 1 with Tool 2 is a single Task-Tool combination). Our experimental setup requires three subjects to cover every possible Task-Tool combination, represented by the three rows in the above table.

In the schedule above, each human subject performs the tasks in the same order. While we randomize the task order for each group of three subjects, within that group each subject's tasks have the same order. Since a group of three human subjects represents every Task-Tool combination, we felt it reasonable to keep the same

[2] www.cis.udel.edu/~shepherd/Research.htm

ordering of tasks within a group to ensure that, for instance, Task 1 is completed with each tool in the first execution slot.

In the overall experiment, we had 18 subjects, meaning that 6 schedules similar to the above schedule were run. Since each schedule covers each Task-Tool combination, we observed every Task-Tool combination 6 times.

Training. To ensure subjects understood how to use each tool, we trained every human subject on each tool prior to the evaluation tasks. We presented the subject with a written script that guided him through the use of each tool on two training tasks, locating the "send message" and "add profile" concepts in PlanetaMessenger, our training application. We observed each subject as they performed the training tasks and corrected any misuse of the tools. However, during experimental evaluation, we did not correct any usage. During the training session, we included advanced search features, such as the wildcard "*" for lexical search, because expert developers use advanced features to cope with the short-comings of current search tools.

Task Setup. For each task, we asked the subjects to form a query using the assigned tool that yielded the highest precision and recall values using as little time as possible. To avoid biasing the human subject's query terms by providing a natural language description of the task, we presented each task to the subject visually. We displayed a series of screen shots that showed the corresponding concept occurring during the execution of the subject application. Some screen shots of the action occurring in the application contained natural language clues (e.g., a menu item labeled "compile"). We have noted whether a screen shot gave the subject clues in the last column of Table 2. Since this text was visible during the normal execution of the application we consider these clues reasonable. After the subjects viewed the screen shots, we asked them to verbalize the concept, and we confirmed with a "yes" if the subject correctly understood the concept. In a few cases, the subject did not understand the concept and we showed them the screen shots another time before they understood the concept. We then asked the subject to search for that concern with the tool they were assigned for that task. Once the queries were formed by the human subject, we executed the queries with the assigned tool for that task and examined the result sets to compare with the gold set, calculating precision, recall, and f-measure.

5.4 Threats to Validity

The set of concerns and corresponding gold sets of methods is a threat to validity because it is possible that the selected tasks favored a certain tool. To minimize this threat, we selected concerns through an objective process involving bug reports, and the naturally subjective selection of the gold sets was conducted by a researcher with no prior understanding of Find-Concept or the subject applications.

Because we used only four subject applications in our experiment, it is possible that our results cannot be generalized to all Java applications. To maximize the generality of our results, we used reasonably-sized, popular open-source applications. We also identified concerns from bug reports, ensuring that the concerns were of interest in real-world situations. Because we only used action-oriented, user-observable concepts, we cannot generalize our results to all types of concepts.

6. Results

In this section, we describe the overall effectiveness of each tool, compare Find-Concept's performance to ELex and GES, and summarize user effort measured across all tools.

6.1 Effectiveness

Figure 7 (a) presents a box plot of the f-measure for each tool across every subject user and search task. For completeness, we have included the precision and recall results for each task in Figure 9. However, we believe that f-measure is a more appropriate performance measure for concern location tasks. Figure 7 (b)–(j) illustrates the f-measure for each tool separated by task. In each figure, the box represents 50% of the data and spans the width of the inner quartile range (IQR), with each whisker extending $1.5*$IQR beyond the top and bottom of the box. The center horizontal line within each box denotes the median, $+$ represents the mean, and outliers are represented by $\circ$.

As illustrated by Figure 7 (a), Find-Concept tends to have better precision and recall, as combined in the f-measure. The short height of Find-Concept's box compared to GES's box indicates that Find-Concept more consistently produced higher results than GES.

6.1.1 Find-Concept vs. ELex

Find-Concept is consistently more effective than ELex at locating concerns (see Figure 7 (a)). We believe this is due to the problems described in Section 2.1.1 that plague ELex, causing low performance. For instance, it is difficult to use ELex to locate the concept "play track" because ELex does not handle concatenated search terms well. This caused subjects to broaden initial queries like "play*music" (0 matches) to"play" (1995 matches); neither of which is an effective query due to the number of results returned. ELex also does not recommend synonyms nor does it account for morphological forms. Users searching for the concept "add textfield" often searched for only "add*textfield" or "create*textfield", thus missing relevant results.

6.1.2 Find-Concept vs. GES

While Figure 7 (a) shows that Find-Concept is more effective than GES overall, we found that the tasks had a large impact on tool performance. Out of 9 tasks, we found that Find-Concept outperformed (i.e., achieved a higher mean F-measure than) GES on 4 tasks, GES outperformed Find-Concept on 2 tasks, and Find--Concept and GES performed similarly for 3 tasks. Find-Concept performed as well or better than GES on 7 out of 9 tasks. Here, we analyze the results closely.

Find-Concept $>$ GES

Find-Concept performed better than GES on tasks "add textfield", "add auction", "save auctions", and "play track" (see Figure 7(b), 7(d), 7(f), and 7(j), respectively); we believe for many of the reasons we anticipated in Section 3.

When searching for the concept "add textfield", subjects often began with the initial query "add textfield". Because of its ability to identify synonyms, Find-Concept recommended that the users add "construct" and "create" to this query. These additions increased the query's effectiveness because the relevant concern is implemented using the verbs "add", "create", and "construct".

Find-Concept also expanded queries to include morphological variations of words, expanding from "auction" to include "auctions" (for the "save auctions" task). This expansion led to an increase in query effectiveness because a search for "auction" misses an important method where "auctions" is used instead of "auction".

Find-Concept created recommendations based on the structure of the AOIG, sometimes leading to surprising but effective recommendations. For instance, in the "save auctions" task, Find-Concept recommended adding the direct object "file" when the initial query was "save auctions". Find-Concept recommended "file" because "file" appears to be similar to "auctions" in the AOIG. In

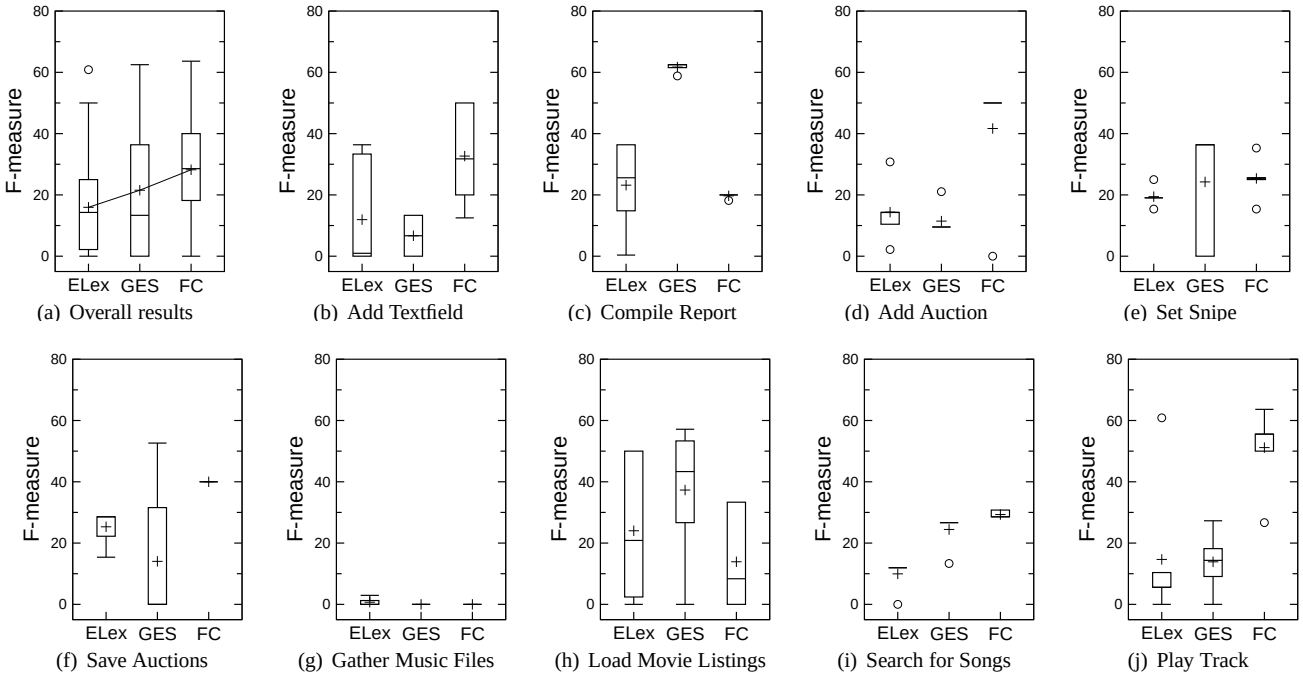

Figure 7. Overall results by tool shown by task. Find-Concept is abbreviated with FC. See Section 6.1 for a thorough explanation of the graph's notation.

our study, the user often accepted this recommendation, leading to increased query effectiveness.

Searching over the AOIG created a higher precision search for Find-Concept. During the "play track" task, a subject using Find-Concept can search for the *verb* "play" and locate the relatively small occurrences of the term "play" where "play" is used as a verb (approximately 41 occurrences in approximately 34 different methods). However, using GES or ELex, users' queries returned over 500 methods where "play" occurs, with many relevant methods ranked low.

Find-Concept < GES

GES performed better than Find-Concept on tasks "compile report" and "load movie listings" (see Figure 7(c) and 7(h), respectively); we believe this is due to limitations of the current AOIGBuilder and a lack of natural language interpretable clues in the corresponding source code (see Table 1 for the percent of code that is commented).

Users searching for the concept "compile report" often used the terms "compile report" as their initial query. Because the AOIGBuilder has difficulty interpreting natural language clues for certain program elements, the AOIG misses a few key nodes. For instance, the AOIGBuilder has trouble interpreting the class-name IReportCompiler. The AOIGBuilder is able to identify the verb "compile" but is unable to identify the direct object "report" because of its limited rule set. Thus, a search for "compile report" will fail because no "report" node exists for the class name IReportCompiler. We intend to augment the AOIGBuilder to interpret class names like IReportCompiler in the future.

GES performed better on the "load movie listings" task because terms relevant to the search were located in text our NLP analysis does not consider—meaning these clues were not in the AOIG. For instance, the term "movie" only appeared in the string literal "www.movies.go.com/cgi/movielistings/request.dll&ZIPSPECIFIC...". The AOIGBuilder does not currently consider string literals of this format. We intend to address this limitation in future work.

Find-Concept ≡ GES

On tasks "set snipe", "gather music files", and "search for song", Find-Concept's performance was very similar to GES's performance (see Figure 7(g), 7(i), and 7(e), respectively). The following does not compare Find-Concept with GES but rather shows areas where Find-Concept can be improved.

Subjects had difficulty translating the concept "set snipe" into a concrete query. Subjects often entered the initial query "snipe" as a verb. However, in the concept "set snipe" "snipe" is the direct object. The query "snipe" as a verb caused Find-Concept to search for a different concept than "set snipe".

Almost all subjects that searched for the concept "gather music files" with Find-Concept entered the query "find music". Find-Concept did not recommend the relevant term "files" during query expansion because the verb "find" is used with too many other direct objects besides "files". We envision improving our recommendations by performing analysis on the AOIG to determine that the direct objects "music" and "music file" are similar if their uses and their semantic similarity are analyzed.

Find-Concept had difficulty locating three of the five methods in the gold set of "search for song" because three methods (group A: getResu(), toStringSearch(), and SearchResult.new()) had weak natural language clues. Find-Concept found the other two methods that had strong clues (group B: search() and search()). However, the methods in group A were strongly linked with group B via program structure links. In future work, we plan to expand search result sets via program structure links, which could capture group A by expanding outwards from group B, even with only partial natural language clues.

6.2 Effort

In Figure 8, we present the mean time required by subjects to perform search tasks for each tool. Our results indicate that the time

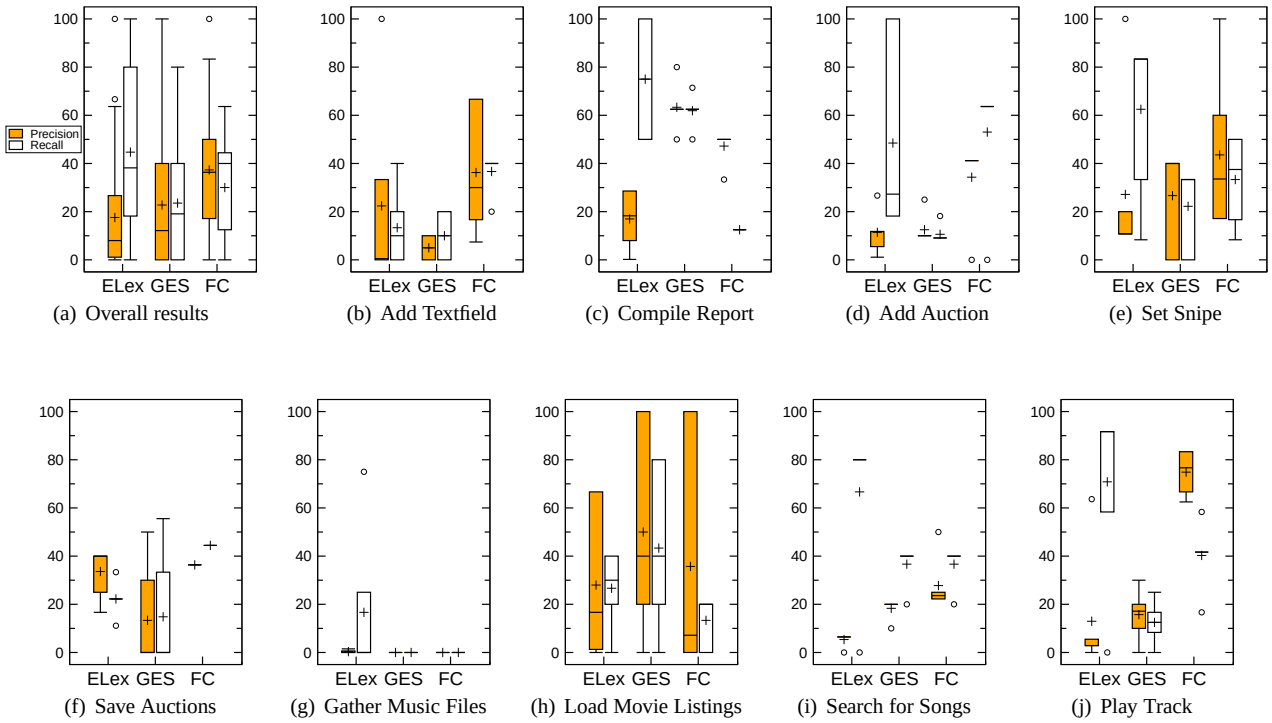

Figure 9. Precision and Recall results shown by tool and task. (Find-Concept is abbreviated with FC).

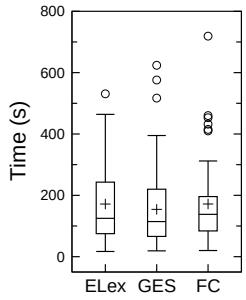

Figure 8. Overall effort results for each tool.

required by each tool is not significantly different. Therefore, we conclude that all the tools require equivalent user effort. We infer that Find-Concept saves time for maintenance tasks by increasing the $location\ rate$, or the average number of relevant methods found per time unit. Finding relevant methods faster should lead to understanding concerns sooner and thus solving maintenance tasks more quickly.

7. Conclusions

In this paper, we described the Find-Concept search process which has the goal of locating action-oriented concerns in large source code bases with the aid of NLP on source code. We presented its query expansion and result graph construction algorithms as well as our implementation. We evaluated our approach to locating action-oriented concerns and compared it to a state-of-the-art lexical search tool and a modified commercial information retrieval tool in terms of search effectiveness and user effort. Overall, we found that Find-Concept was more consistent and more effective than either ELex or GES, without requiring additional user effort. In cases where Find-Concept's performance is worse or similar to GES, Find-Concept's performance could be improved by augmenting the AOIGBuilder's theory and implementation, which we plan to examine in future work. We believe Find-Concept's demonstrated effectiveness warrants further investigation of concern location mechanisms that utilize natural language and program structure information.

8. Future Work

The most promising area of future work that would improve Find--Concept's performance is to create a more effective AOIGBuilder by improving its theory and implementation. The current AOIGBuilder accurately analyzes the majority of method names correctly [32], but, as noted in our experimental evaluation discussion, Find-Concept's performance is directly affected by any shortcomings of the AOIGBuilder. We are currently refining the AOIGBuilder by identifying cases where the AOIGBuilder fails and creating new techniques for these cases.

Although we have evaluated Find-Concept on several open source programs of varying quality and size with promising results, we have not directly evaluated the effect of naming conventions on Find-Concept's or the AOIGBuilder's performance. In general, we expect our tools to perform better on high quality source code. We plan to evaluate this quantitatively in the future.

Find-Concept assumes that many programming tasks focus on actions. We believe that most bug reports refer to actions, based on our extensive reading of bug reports for this project, our own experience submitting bug reports, and our intuition. However, we have not performed a quantitative evaluation of this phenomenon, which we leave as future work.

Within the Find-Concept framework, there are many opportunities for full automation, such as automatically expanding queries. We plan to investigate opportunities for further automating Find-Concept.

References

[1] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 2002.

[2] E. Baniassad and S. Clarke. Theme: An approach for aspect-oriented analysis and design. In *Int. Conf. on Softw. Eng.*, 2004.

[3] N. S. Barghouti, J. Mocenigo, and W. Lee. *rappa:* A GRAPh PAckage in Java. In *Graph Drawing*, pages 336–343, 1997.

[4] G. Booch. Object-oriented design. *Ada Lett.*, I(3):64–76, 1982.

[5] L. Campos. Planeta messenger. online, September 2006. http://sourceforge.net/projects/planeta/.

[6] J. Carroll and T. Briscoe. High precision extraction of grammatical relations. In *7th Int. Wkshp on Parsing Technologies*, 2001.

[7] K. Chan, Z. C. L. Liang, and A. Michail. Design recovery of interactive graphical applications. In *Int. Conf. on Softw. Eng.*, 2003.

[8] K. Chen and V. Rajlich. Ripples: Tool for change in legacy software. In *Proc. of the Int. Conf. on Softw. Maint.*, 2001.

[9] Eclipse. Java development tools, 2005. (March 13, 2003).

[10] M. Eichberg, M. Haupt, M. Mezini, and T. Schafer. Comprehensive software understanding with SEXTANT. In *Proc. of Int. Conf. on Software Maintenance*, 2005.

[11] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Trans. Softw. Eng.*, 29(3):210–224, 2003.

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

[13] B. Florat. Jajuk: Advanced jukebox. online, September 2006. http://jajuk.sourceforge.net.

[14] J. Gosling, B. Joy, and G. Steele. Java Language Specification. online, September 2006. http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html.

[15] O. S. T. Group. Sourceforge. online, September 2006. http://sourceforge.net/.

[16] IBM. Eclipse IDE. online, September 2006. http://www.eclipse.org.

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, 2001.

[18] K. Lukoit, N. Wilde, S. Stowell, and T. Hennessey. Tracegraph: Immediate visual location of software features. In *Int. Conf. on Software Maintenance*, 2000.

[19] A. Marcus, R. Koschke, A. van Deursen, V. Rajlich, P. Tonella, and H. Sneed. Identification of concepts, features, and concerns in source code. Panel Discussion at the International Conference on Software Maintenance, 2005.

[20] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proc. of Int. Conf. on Software Engineering*, 2003.

[21] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev. Static techniques for concept location in object-oriented code. In *Proc. of Int. Wkshp on Program Comprehension*, 2005.

[22] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Working Conf. on Reverse Eng.*, November 2004.

[23] A. Michail. Browsing and searching source code of applications written using a GUI framework. In *Int. Conf. on Softw. Eng.*, 2002.

[24] J. B. T. Morton and G. Bierner. OpenNLP maxent package.

[25] G. Murphy, M. Kersten, M. Robilland, and D. Cubranic. The emergent structure of development tasks. In *ECOOP*, 2005.

[26] M. F. Porter. An algorithm for suffix stripping. *Readings in information retrieval*, pages 313–316, 1997.

[27] D. Poshyvanyk, M. Petrenko, A. Marcus, X. Xie, and D. Liu. Google Eclipse Search. In *Int. Conf. on Softw. Maint.*, 2006.

[28] G. Project. grep. online, September 2006.

[29] M. Robillard. Automatic generation of suggestions for program investigation. In *Fund. of Softw. Eng.*, 2005.

[30] M. P. Robillard and G. C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Int. Conf. on Softw. Eng.*, 2002.

[31] A. Sampaio, R. Chitchyan, A. Rashid, and P. Rayson. EA-Miner: a tool for automating aspect-oriented requirements identification. In *Int. Conf. on Automated Softw. Eng.*, 2005.

[32] D. Shepherd, L. Pollock, and K. Vijay-Shanker. Towards supporting on-demand virtual remodularization using program graphs. In *Proc. of Int. Conf. on Aspect-oriented software development*, 2006.

[33] D. Shepherd, T. Tourwe, and L. Pollock. Using language clues to discover crosscutting concerns. In *Int. Wkshp on Modeling and Analysis of Concerns in Software at ICSE*, 2005.

[34] A. Software. RefactoIT Plugin 2.5. online, September 2006. http://www.refactorit.com/.

[35] D. Spencer. Lucene Synonyms. online. http://www.tropo.com/-techno/java/lucene/wordnet.html.

[36] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *ICSE*, 1999.

[37] C. J. Van Rijsbergen. *Information Retrieval, 2nd edition*. Dept. of Computer Science, University of Glasgow, 1979.

[38] K. D. Volder and D. Janzen. Navigating and querying code without getting lost. In *Aspect Oriented Software Development*, 2003.

[39] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: Towards a static non-interactive approach to feature location. In *Int. Conf. on Software Engineering*, 2004.