Automatically Capturing Source Code Context of NL-Queries for Software Maintenance and Reuse [*]

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

Abstract

As software systems continue to grow and evolve, locating code for maintenance and reuse tasks becomes increasingly difficult. Existing static code search techniques using natural language queries provide little support to help developers determine whether search results are relevant, and few recommend alternative words to help developers reformulate poor queries. In this paper, we present a novel approach that automatically extracts natural language phrases from source code identifiers and categorizes the phrases and search results in a hierarchy. Our contextual search approach allows developers to explore the word usage in a piece of software, helping them to quickly identify relevant program elements for investigation or to quickly recognize alternative words for query reformulation. An empirical evaluation of 22 developers reveals that our contextual search approach significantly outperforms the most closely related technique in terms of effort and effectiveness.

1. Introduction

When performing software maintenance or reuse tasks, developers must first identify the relevant code fragments to be modified or reused. As software systems continue to grow and evolve, identifying code relevant to a particular task within millions of lines of code becomes increasingly difficult. Thus, there is a critical need for automated support to help developers work effectively and minimize maintenance and reuse costs.

To identify code relevant to the task, developers typically use an *iterative refinement* process [8, 10] as shown in Figure 1. 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.

Challenges. Studies show that formulating effective natural language queries can be as important as the search algorithm itself [10]. 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% [9]. Specifically, query formulation is complicated by the vocabulary mismatch problem [10] (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 [20].

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 [32]. Thus, we believe the role of automation is not to automatically expand the query, but to provide information about the underlying word usage in the code that will enable the human user to quickly formulate an effective query. Currently, few systems recommend alternative words to help developers reformulate poor queries [27, 30].

Another challenge in the iterative refinement process is discriminating between relevant and irrelevant search results. Presentation of the search results is not always adequate to determine relevance, forcing the user to further examine the code. If users cannot quickly determine that

[*]This material is based upon work supported by the National Science Foundation under a Graduate Research Fellowship and Grant No. CCF-0702401.

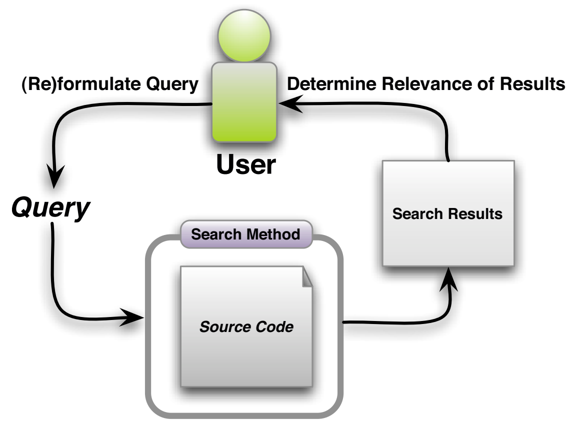
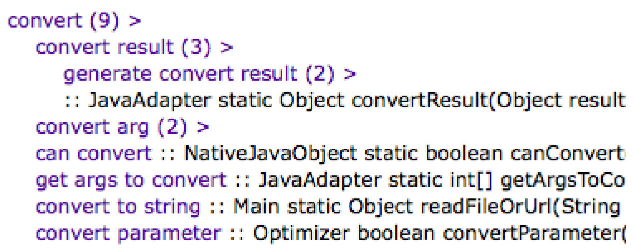


Figure 1. Iterative Query Refinement and Search Process



```
convert (9) >
   convert result (3) >
      generate convert result (2) >
      :: JavaAdapter static Object convertResult(Object result
   convert arg (2) >
   can convert :: NativeJavaObject static boolean canConvert
   get args to convert :: JavaAdapter static int[] getArgsToCo
   convert to string :: Main static Object readFileOrUrl(String
   convert parameter :: Optimizer boolean convertParameter(
```

Figure 2. Example results for "convert" query. Phrases are to the left, followed by the number of matching signatures, and signatures follow '::'.

results are irrelevant, they could waste significant time investigating irrelevant code. Existing static code search techniques using natural language queries [21, 28] provide little support to help developers determine whether search results are relevant beyond ranking the results [26, 30].

Providing Automated Support. In this paper, we present a novel approach that provides automated support to the developer both in formulating queries and discriminating between relevant and irrelevant search results. Our 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. We automatically capture the context of the 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. Because we provide word context for the occurrences of query words in the source code during the iterative refinement process, we call our approach *contextual search*.

For example, consider the search results for the query "convert" in Figure 2. Extracted phrases are to the left, followed by the matching signatures. By skimming the list of words occurring with "convert", 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.

The phrases, which are extracted from source code, naturally form a hierarchy of related phrases. At the top of the hierarchy are more general phrases, and at the bottom are the most specific phrases, which contain the most words. Continuing with the "convert" example, the most general phrase is the query, "convert", and more specific phrases include "convert result", "can convert", and "get args to convert". Further, the phrase "convert result" is more general than "generate convert result", which occurs below it in the hierarchy. The leaf nodes of the hierarchy are the specific program elements that match the phrases. This phrase hierarchy allows the developer to quickly identify relevant program elements by reducing the number of relevance judgments, while the natural language phrases help the developer to formulate effective queries.

Why Phrases? Our contextual approach is motivated by insights gained from Shepherd et al.'s approach to query expansion and code search [30]. The approach by Shepherd et al. uses <verb, direct object> (V-DO) pairs from method signatures and comments to find actions that crosscut object-oriented systems. The previous experimental study showed that by capturing specific word relations in identifiers, such as V-DO pairs, users were able to produce more effective queries more consistently than with two competing search tools. However, strict V-DO queries cannot be used to search for every feature. For example, V-DO cannot search for features expressed as noun phrases without a verb, such as "reserved keyword" or "mp3 player".

One potential approach to go beyond V-DO pairs is to capture all word relation pairs in software by using co-occurrences [19]. Although co-occurrences can find meaningful word relationships, in our investigation, we found that co-occurrences did not consistently find relationships that would aid in query reformulation. We tried co-occurring terms within identifiers, within methods, in comment to method mappings, from the method name to methods called within the method's implementation, etc. The key problem that we observed with co-occurring word pairs is that *word order matters*. For example, we observed that knowing that "item" and "add" co-occurred more often than due to chance was less useful than simply knowing that the phrase "add item" frequently occurred.

Contributions. The main contributions of this paper are:

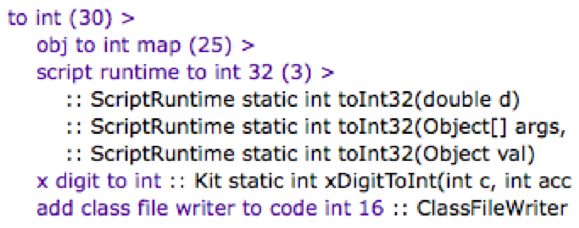
- An algorithm to automatically extract and generate noun, verb, and prepositional phrases from method and field signatures, capturing word context of natural language queries for software maintenance and reuse
- An approach to automatically categorize extracted phrases into a hierarchy based on partial phrase matching, to help software maintainers quickly discriminate between relevant and irrelevant search results and reformulate queries
- An empirical evaluation of 22 developers comparing our contextual search approach to verb-direct object, the most closely related search technique. Our results show that contextual search significantly outperforms verb-direct object in terms of effort and effectiveness.

Our contextual approach involves only static analysis of source code, requiring no execution information, and thus can be applied to incorrect, incomplete, and unexecutable legacy programs. The phrase extraction process, currently implemented for Java, requires less than 10 seconds for 100 KLOC, and can analyze 1.5 million LOC in 3 minutes.

2. Contextual Search Example

Consider hypothetical developer Amanda, who needs to add a 32-bit signed integer conversion to a new web language. She has access to the documentation and implementation of a JavaScript interpreter with a similar feature. After reading the related documentation, Amanda notices that the words "convert," "to," "int," "integer," and "32" look important. She begins her search using our contextual approach by first entering the query "convert," shown in Figure 2. However, none of the results appear to have anything to do with integers, numbers, or math. Amanda moves on to the next few words she considers relevant, and tries the query "to int." With two clicks, Amanda sees the results shown in Figure 3, and finds 3 method signatures that look like promising candidates for further investigation. Without the natural language phrases and hierarchy, Amanda would have to make relevance judgments for the entire list of 30 results, rather than just 7.

In this example, Amanda was able to backtrack from the ineffective query "convert" to try the next best query on her list, "to int." Contextual search with backtracking can also be used to help users overcome vocabulary mismatch. For example, if a user searches for "delete item" and sees few or no results, the user can backtrack to the shorter query "item." Upon seeing results for "item," the user can view co-occurring words and determine alternate verbs, such as "remove" in the phrase "remove item."



```
to int (30) >
   obj to int map (25) >
   script runtime to int 32 (3) >
      :: ScriptRuntime static int toInt32(double d)
      :: ScriptRuntime static int toInt32(Object[] args,
      :: ScriptRuntime static int toInt32(Object val)
   x digit to int :: Kit static int xDigitToInt(int c, int acc
   add class file writer to code int 16 :: ClassFileWriter
```

Figure 3. Example results for "to int" query. Indentation indicates level of hierarchy. Phrases are to the left, followed by the number of matching signatures, and signatures follow '::'.

3. Approach: Capturing Context with Phrases

Figure 4 illustrates our contextual search process. There are two main components to our approach: (1) extracting and generating natural language phrases from source code, followed by (2) search and hierarchical categorization of the phrases. Each subsection describes the challenges to be addressed, followed by our approach and detailed algorithm.

3.1. Information Extraction

Challenges. The first challenge we faced in developing a phrase extraction process was deciding what phrases should be extracted that best capture word context. We used exploratory data analysis techniques [19] to cluster groups of related signatures and develop phrase extraction rules for each subgroup. For example, we developed extraction rules for static methods and fields, methods with and without parameters, and methods and fields with verbs and objects in the name. We analyzed methods with different return types [16], as well as prepositions in the beginning, end and middle of names. When possible, we generalized common rules. For instance, the same rule to extract direct objects from formal parameters was developed independently for static and non-static methods. We also dropped any rule that extracted meaningful phrases for only some signatures, if we could not determine under what conditions the rule should be applied. In general, we expect our rules to work well for about 75-95% of method and field signatures.

The quality of our extraction rules depends on the variety of signatures under study and their naming conventions. We endeavored to study a diverse set of Java signatures and

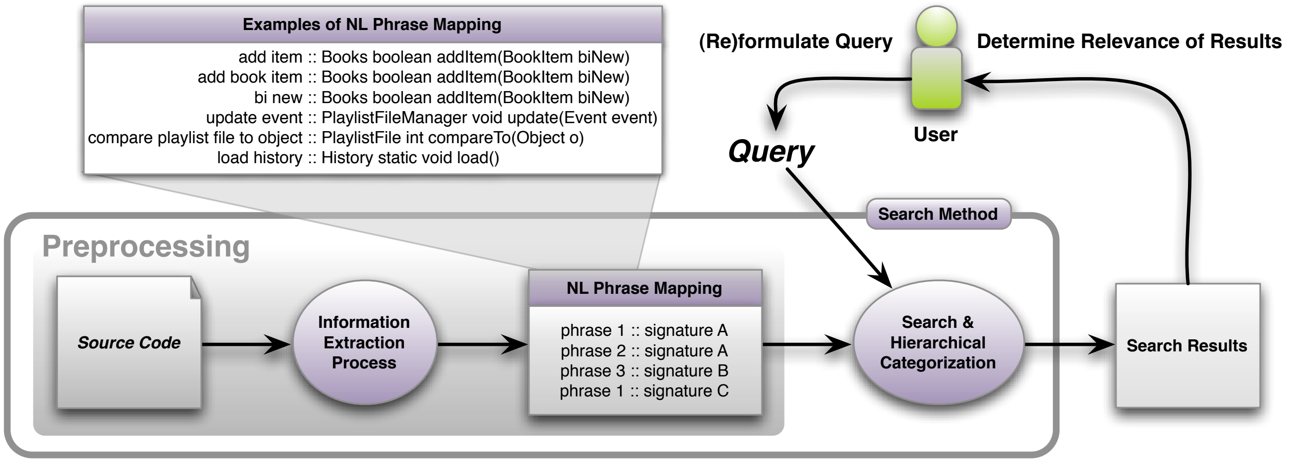

Examples of NL Phrase Mapping
add item :: Books boolean addItem(BookItem biNew)
add book item :: Books boolean addItem(BookItem biNew)
bi new :: Books boolean addItem(BookItem biNew)
update event :: PlaylistFileManager void update(Event event)
compare playlist file to object :: PlaylistFile int compareTo(Object o)
load history :: History static void load()
(Re)formulate Query
Determine Relevance of Results
User
Query
Search Method
Preprocessing
Source Code
Information Extraction Process
NL Phrase Mapping
phrase 1 :: signature A
phrase 2 :: signature A
phrase 3 :: signature B
phrase 1 :: signature C
Search & Hierarchical Categorization
Search Results

Figure 4. The Contextual Search Process

naming conventions by analyzing the most frequently occurring identifiers in a set of 9,000 open source Java programs downloaded from `sourceforge.net`. This set of programs contains over 18 million signatures, with 3.5 million unique names consisting of over 200 thousand unique words. Algorithm 1 is the culmination of our detailed analysis of thousands of unique method and field signatures.

Overview. Our guiding principle to phrase extraction is that providing the user with incorrect or misleading information is worse than providing no information at all. Therefore, we strived to achieve balance between conservatively extracting information in which we have a high confidence of accurately portraying word context, and yet still providing enough information that the system is usable.

Our technique can extract verb, noun, and prepositional phrases from method and field signatures. A *noun phrase* (NP) is a sequence of noun modifiers, such as nouns and adjectives, followed by a noun, and optionally followed by other modifiers or prepositional phrases [19]. A *verb phrase* (VP) is a verb followed by an NP, and does not usually include the subject of the verb. A *prepositional phrase* (PP) is a preposition plus an NP, and can be part of a VP or NP.

Extracting Phrases from Signatures. We extract phrases from both method signatures (including class name, method name, type, and formal parameters) and field signatures (including class name, type, and field name). Our extraction process has four major steps: (1) splitting identifiers into space-delimited phrases; (2) determining if the (method or field) name should be treated as an NP, VP, or PP; (3) identifying the verb, direct object, preposition, and indirect object of the VP, and (4) inferring arguments for VPs to generate additional phrases. Our phrase extraction technique is presented in Algorithm 1, with example extracted phrases shown in Figure 4.

During NP construction and VP generation, we use formal parameter names and types as objects. In line 6 of Algorithm 1, all formal parameter names and non-primitive formal types are added to $args$. Primitive types include the basic types int, void, boolean, etc. as well as String since it is a very common type. We add non-primitive parameter types as well as parameter names to account for instances where a parameter name is an abbreviation or conveys little meaning.

Splitting Identifiers. Before determining if an identifier is an NP, VP, or PP, the first step is to process the identifiers and split them into space-delimited phrases. The $split$ function takes as input an identifier and outputs a sequence of words. In addition to splitting on punctuation and numbers, we also split based on camel case. For instance, MP3FileFilter would become "mp 3 file filter" and XYLine3DRenderer "xy line 3 d renderer." Like Caprile and Tonella [4], in certain cases we translate "2" into "to."

To improve readability, some identifiers do not use camel casing. For instance, XYZtoRGB and getRunMPwithout-MASC do not follow strict camel casing ("to" and "without" should be capitalized, respectively). We observed that these cases were mostly prepositions, and added special splitting rules for the most common ones: to, from, without, by, for, with. We did not add rules for prepositions that often end legitimate words, such as "in" in HOMEBinDIR.

Identifying NPs and PPs. To check if a method name should be treated as an NP, we first identify whether the signature is a constructor in line 8. If so, we consider the name and each formal argument in $args$ as individual NPs that each map to the signature. Next, in line 10, we check

Algorithm 1 $extractPhrasesFromSignature(sig)$

```
 1: Input: field or method signature, $sig$
 2: Output: set of phrases for the signature, $pset$
 3: $name \leftarrow split(name(sig))$
 4: $type \leftarrow split(typeOrReturnType(sig))$
 5: $class \leftarrow split(declaringClass(sig))$
 6: $args \leftarrow$ set of $split(formals(sig))$
 7: $pset \leftarrow \emptyset$
 8: if $isConstructor(sig)$ then
 9:     $pset \leftarrow \{name\} \cup args$ // NPs
10: else if $hasTrailingPastParticiple(name)$ then
11:     $pset \leftarrow \{name\}$ // NP
12: else if $hasLeadingPreposition(name)$ then
13:     $name \leftarrow \{class + name\}$ // NP
14: else if $hasLeadingVerb(name)$ then {// Construct VP}
15:     $v \leftarrow getVerb(name)$
16:     if $hasObjectInName(name)$ then
17:         $DO \leftarrow getObject(name)$
18:     else if $hasParameters(sig)$ then
19:         $DO \leftarrow \{getFirstFormalName(sig) \cup$
            $getFirstFormalType(sig)\}$
20:     else
21:         $DO \leftarrow class$
22:     end if
23:     if $containsPreposition(DO)$ then {// Generate VPs}
24:         for all prepositions $p \in DO$ do
25:             $DO_i \leftarrow getWordsBeforePrep(DO, p)$
26:             $IO \leftarrow getWordsAfterPrep(DO, p)$
27:             $pset \leftarrow pset \cup inferArguments(v, DO_i, p, IO, args)$
28:         end for
29:     else
30:         $pset \leftarrow inferArguments(v, DO, \emptyset, \emptyset, args)$
31:     end if
32: else
33:     $pset \leftarrow pset \cup name$ // NP
34:     if $isField(sig)$ then
35:         $pset \leftarrow pset \cup type$ // NP
36:     end if
37: end if
38: return $pset$
```

if the signature contains a trailing past participle, such as in "action performed" or "key pressed." If so, we consider the name to be an NP and add it to $pset$, the set of phrases for the signature. These cases comprise the NPs for which we have high confidence of correctness.

We next determine if the name is a PP. In line 12, we check if the first word is a preposition, and if so, we concatenate the class name before the method name, and add the constructed phrase to $pset$. For example, toByteArray in class FileWriter would become "file writer to byte array." In contrast to previous work [30], we do not automatically infer a verb such as "convert" in these situations. Following our conservative policy, we treat such cases as PPs rather than potentially erroneous VPs. Next, we attempt to identify the name as a VP. If we do not successfully identify the name as VP, we treat the name as an NP in line 33.

Identifying and Constructing VPs. In line 14, we consider the name to be a VP if the first word is a verb. Note that the behavior of $hasLeadingVerb$ depends on whether a method or field signature is being analyzed. For fields, the name must begin with a verb *and* consist of more than one word; for methods, the name need only start with a verb. Since fields are less likely to begin with verbs and have no parameters, we only process a field name as a VP if there is also an object in the name, e.g., printWhenExpression.

After extracting the verb, we locate the verb's arguments, starting in line 16. First, we determine if there is an object in the name following the verb, as in getConnectionType. If not, we use the first formal parameter name as the object as well as the first parameter's type if it is not primitive, as in line 19. Otherwise, we use the class name as the object.

Next, in line 23, we look for any prepositions in the verb's object. For every preposition we find, we gather information about direct and indirect objects and call *inferArguments* to generate additional phrases. We identify the direct object from the words before the preposition, and the indirect object from the words after the preposition. We continue examining all prepositions in the verb's object because not all words that can be prepositions act as such in identifiers. For example, in "show about dialogue," "about" is actually acting as an adjective that modifies "dialogue."

Generating Additional Phrases. Based on the formal parameter names and types, we construct additional VPs that represent the signature without repeating the name. For example, a method may have a general name, such as addItem. However, a parameter (name or type) may indicate that only a specific type of item is being added, e.g., a BookItem. By inferring the phrase "add book item" in addition to "add item," this signature will also be returned for queries such as "add book."

To generate new phrases, we look for partial matches to the direct and indirect objects in the argument list of method signatures, $args$. Recall that all formal parameter names and non-primitive formal types were added to $args$ earlier. For every phrase in $args$ that overlaps one word in the direct or indirect object of the VP, we emit all possible combinations of original and inferred direct and indirect objects as phrases. For example, for the signature Base64 static Object decodeToObject(String sourceObject), we would output the phrase "decode base 64 to source object." If the name ends in a preposition, and thus contains no indirect object, we treat every argument as an indirect object. In general, no more than 1-2 phrases are added for signatures with parameters, but we have seen as many as 10 for complicated method names that contain prepositions.

3.2. Search and Hierarchical Categorization

Once natural language phrases have been extracted from the source code, the second component searches the phrases associated with each program element and groups related signatures into a hierarchy based on partial phrase matching. As illustrated in Figure 2, 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.

In general, it is computationally infeasible to enumerate all possible sub-phrases, which is an instance of the power set problem. Thus, our technique is based on the top-down approach used in Max-Miner [2] to efficiently mine long data patterns from databases. In contrast to Max-Miner, our implementation is recursive, and we use regular expressions to approximate the set operations as well as to extend the algorithm to handle sequences of words instead of sets.

In displaying search results, it is important to give an example of what is retrieved so the user can decide whether the query needs to be reformulated and in what way [8]. For this reason, in addition to phrases, we also display signatures that match each phrase in the source code. Our existing implementation only places a signature in the hierarchy once, in the topmost category possible. We achieve this by sorting the candidate phrases by the number of phrase matches. We also use the total number of matches of a phrase to sort the branches when displaying the hierarchy to the user.

During the contextual search process and when building the hierarchy, query words do not have to match a phrase exactly but only preserve word order. For example, if the query is "text field," and the phrase "text field xml file" maps to a signature, both "text field file" and "text field xml" will be considered as candidates for sub-phrase matching, even if neither phrase exactly describes a signature. In addition, our approach uses the longest possible phrase that describes a subset of signatures in the hierarchy. For example, if all the signatures matching "text field" also match "j text field", our approach will skip the shorter phrase "text field" and only add "j text field" to the hierarchy.

3.3. Implementation

Our current implementation extracts and generates phrases from Java code as an Eclipse plug-in. We use a morphological parser, PC-Kimmo [1], to determine the possible parts of speech for individual words. The hierarchical categorization of phrases is implemented in php (`http://www.cis.udel.edu/~hill/context`).

Ideally, this technique should be integrated into a development environment. For this to be feasible, it must be possible to incrementally update the phrase representation as the code evolves. Because phrases are stored per method or per field, a new set of phrases can be extracted for modified program elements in the background or before the next search is executed. The initial extraction time from source code is very reasonable: 2-5 seconds for 20 KLOC, 8 seconds for 75 KLOC, and under 3 minutes for 1.5 million LOC. The extraction time depends on the length of signature names as well as the number of parameters overlapping NPs in method names. The hierarchy of phrases is constructed online based on the query with imperceptible delay for small (less than a thousand) result sets. For larger result sets, delay has been minor (less than ten seconds), but noticeable. For more substantial code sizes, a map/reduce architecture could be used to reduce overall search costs.

Future Extensions. Our current approach extracts information from method and field signatures. In the future, we plan to extract phrases from body statements as well as comments to get a more accurate set of phrases that describe a method's actions. However, building these extraction rules requires further analysis. Comments are free form, containing less structure than signatures, and need a different set of rules to properly extract phrases in VP, NP, or PP form. Body statements present additional challenges in determining appropriate direct and indirect objects.

Although other approaches to identifier splitting have used a dictionary to further split identifier words with no boundaries [15, 24], such as "scrollbar" or "textfield," in this work we have conservatively chosen to leave such words intact. In addition, we have chosen not to use stemming in the current version, since removing a stem can affect the part of speech of a word. The effect of changing the part of speech of a word on the readability of these phrases has not yet been studied. Finally, other approaches have used synonyms [30]. Rather than use potentially inaccurate, domain-independent synonyms [32], we have chosen to leave exploration of synonyms for future work.

4. Evaluation

The research question under investigation is:

What effect does a contextual search of natural language phrases have on the effort and effectiveness of developers searching source code?

To evaluate this research question, we compared our contextual search with an approximation of the verb-DO search method [30], as well as to a version of our tool that does not display the phrases or hierarchy, only the matching signatures. In this study, we compare search results of 22 developers performing 28 concern location tasks.

4.1. Independent Variable

The independent variable in our study is the search technique: $context_H$, $V\text{-}DO$, and $context_L$. The $context_H$ search technique is the contextual search approach described in Section 3. We compare $context_H$ to a $V\text{-}DO$ approach [30] which we implemented. For the purposes

Number of Software Developers in Study					
No. Years	**Programming Experience**	**Industry Experience**	**Perform Maintenance**	**Perform Maintenance on Code Not Authored**	**Frequency**
10+ years	13	–	5	2	**Daily**
5-9 years	4	6	7	3	**Weekly**
1-4 years	5	11	7	7	**Monthly**
$<$ 1 year	–	5	3	10	**Yearly**

Table 1. Subject Developer Characteristics

of evaluation, we removed as much variability as possible between $V\text{-}DO$ and $context_H$ by implementing $V\text{-}DO$ within the $context_H$ framework, while still maintaining the essence of the approach. We did this to explore whether natural language phrases beyond $V\text{-}DO$ improve searching capabilities, without studying effects caused by synonym recommendations or other minor algorithmic differences.

The $V\text{-}DO$ approach requires the user to enter verb and direct object queries as input and outputs a list of signatures. The query consists of a single verb, followed by a direct object which may be multiple words. Examples include "remove item" and "lookup performance event." The approach will always treat the first word of the query as a verb, and matches exact V-DO phrases only [31]. For example, the query "lookup performance" will only match "lookup performance" and not "lookup performance event."

To address the vocabulary mismatch problem, $V\text{-}DO$ displays a set of verb and object recommendations for query reformulation below the list of exact matches. One column shows a list of the verbs that co-occur in the code with the direct object in the query, sorted by frequency. Similarly, another column displays all the direct objects that co-occur in the code with the query's verb.

The extraction process for $V\text{-}DO$ differs slightly from $context_H$. First, in the case of constructors, the verb "constructs" is added in front of the class name. Second, non-void methods beginning with the prepositions "from," "to," and "as," are replaced with the verb "convert." Method names beginning with "is" are replaced with "check." Finally, no prepositional phrases are explored or new phrases generated as with $context_H$; as soon as the direct object is identified the phrase is complete.

We also compare $context_H$ to a baseline version, $context_L$. The $context_L$ technique uses the same query and search technique as $context_H$ to identify matches, but skips the hierarchical categorization step and simply displays the results in a list. We use the $context_L$ technique to explore whether the phrase matching or the hierarchical categorization has more of an effect on the search.

4.2. Dependent Variables and Measures

The dependent variables in the study are user effort and search effectiveness. We measure *effort* in terms of the number of queries submitted, ignoring any identical consecutive queries. We measure *effectiveness* by calculating the common measures of precision and recall on each search result set [20]. *Precision* is the percent of search results that are relevant, and captures how many irrelevant results were present with the relevant results. *Recall* is the percent of all relevant results that were correctly returned as search results, and captures how many of the actually relevant results were predicted as relevant. We combine precision and recall using the *F measure*, which is high only when both precision and recall are similarly high.

Although results for effective queries will ideally have both high precision as well as high recall, and thus a high F measure, it is unlikely that a single query will be capable of capturing both high precision and high recall. For the search techniques in this experiment, individual queries will typically be able to capture *either* high recall (by returning many results) *or* high precision (by returning few, but very relevant results).

4.3. Subjects

The subjects of our study are the human developers and concerns. The concerns formed the search tasks for which subject developers were required to construct queries. The concerns were also used as a gold standard, i.e., a set of relevant program elements, to evaluate effectiveness.

Developers. We obtained results from 22 volunteer software developers with varying levels of programming and industry experience. Table 1 shows characteristics of our subject population. The distribution of years of programming and industry experience for each subject is displayed on the left of the table, and the frequency that they perform maintenance tasks is on the right. Although we confirmed that 27 subjects would participate, 22 completed the study.

Concerns. The description and contents of concerns add significant variability to the study. To control for this variability as much as possible, we used concerns from two completely different sets, which have completely different types of concern descriptions, and different methodologies for deriving the gold sets.

The first set of 19 concerns is from the 45 KLOC JavaScript/ECMAScript interpreter and compiler, Rhino. The gold sets of the concerns were derived by the *removal dependency rule* [7]: under this rule, a method or field was only considered to be associated with a concern if it should be removed or modified when the concern is removed from the program. Two human analysts used this rule to determine 415 concerns for the Rhino program [7]. Each concern maps to a subsection of the documentation, which is used as the concern description.

We selected a random subset of these 415 with some restrictions. First, the concerns varied in size from over 300 program elements to just a single element. Therefore, we restricted our sample to just the middle 50% of the concerns, with sizes ranging from 4 to 25. Second, since each concern mapped to a specific section or subsection of the documentation, we wanted to have a representative sample to control for whether different sections tend to have certain types of concerns. Thus, we selected a random sample of 19 concerns such that every major section of the documentation was represented before repeats were selected. We selected 19 concerns so that the total number could be divided evenly into four groups for the design.

The second set of concerns consists of 9 user-observable, action-oriented concerns from 4 programs ranging in size from 23 to 75 KLOC [30]. The four programs are: iReport, a visual report builder and designer; jBidWatcher, an auction bidding, sniping, and tracking tool for online auction sites such as eBay or Yahoo; javaHMO, a media server for the Home Media Option from TiVo; and Jajuk, a music organizer for large music collections. The concern descriptions consist of screen shots of each concern being executed. The concern implementations were derived by a set of two human analysts who agreed on the concern implementations after executing and exploring the concerns [30].

Both sets of concerns were derived by two groups of independent researchers, and have been used as subjects in previous evaluations [7, 30]. It should be noted that as a compiler, Rhino is out of most of our developer's familiar domain. In addition, it is known from previous experience that the concerns from javaHMO and Jajuk are implemented using very different words than appear in the user interface, which is used for the concern description.

4.4. Design and Methodology

We designed the experiment to compare $context_H$ with $V\text{-}DO$ and $context_L$. In the design, there were two blocking factors: the order that the search techniques, or treatments, were applied, and the concerns. The order the treatments were applied is important to control for learning effects. To create concern blocks, the concerns were randomly assigned to 4 groups of 7 concerns ($A - D$) such that each group contained 4-5 Rhino concerns and 2-3 concerns from 3 other programs. This ensured that each treatment was applied to a consistent variety of concerns.

Unit	Order of $context_H$	Concern group for $context_H$	Concern group for $V\text{-}DO$
1	1	A	D
2	1	C	B
3	2	B	A
4	2	D	C

Table 2. Experimental units for comparing $context_H$ with $V\text{-}DO$

We used a randomized crossed block design [5] to create 8 experimental units, 4 for each comparison ($context_H$ with $V\text{-}DO$ and $context_H$ with $context_L$). The four experimental units used to compare $context_H$ with $V\text{-}DO$ are presented in Table 2. Similar units were used for comparing $context_H$ with $context_L$. Every treatment was applied to every concern and every order, although not every order was applied to every possible combination of concern blocks.

Subjects were initially randomly assigned to experimental units. Because not all subjects completed the study, we do not have an equal number of replications for each experimental unit. Units with less replications were assigned to new subjects as they volunteered; thus, every unit was completed by at least 2 subjects, but no more than 4. The subjects were asked to fill out an exit survey after completing the experiment. All the experimental materials, including the instructions for each experimental unit, are available online: `http://www.cis.udel.edu/~hill/context`.

4.5. Threats to Validity

Studying the effects of human subjects on such an open-ended task as concern location poses many challenges. Although we endeavored to control for variability as much as possible, there are still threats to the validity of the results.

The subject concerns are an unavoidable threat. To minimize the effect that some concerns are more difficult to locate and formulate queries for, we used concerns from 5 different programs with two different types of descriptions. In addition, each participant applied each treatment to 7 concerns to avoid any one concern dominating the results. However, it is possible that the concern groups that we randomly selected were not of equivalent difficulty. We avoided this as much as possible by ensuring that each

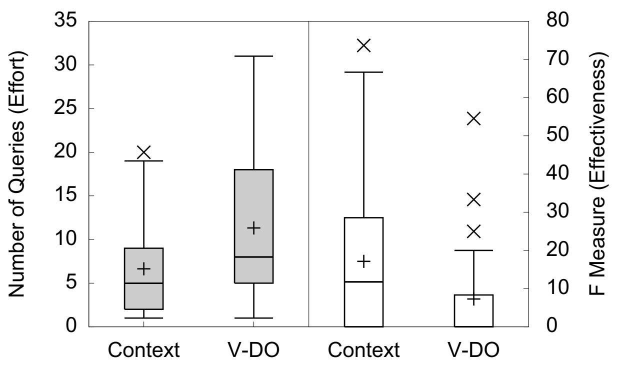


Figure 5. Effort and Effectiveness Results for $context_H$ and V-DO. Effort is measured in terms of the number of queries entered, shown on the left. Effectiveness is measured in terms of the F Measure, shown on the right.

group contained concerns from at least 3 different programs, under the assumption that concerns from the same program will be of approximately the same difficulty.

The experiment was administered as a volunteer online survey to gain access to as many developers with industry experience as possible. However, this meant that subjects were not in a controlled environment, and other distractions may have influenced the attention that subjects devoted to the experiment. For example, one subject was eating during the first part of the experiment, but not the second part. Another subject took an hour break in the middle of reformulating a query for one concern. For this reason, we cannot analyze effort in terms of time, only in terms of the number of submitted queries. Again, we attempted to minimize this threat as much as possible by observing the subjects over 7 concerns per treatment.

5. Results and Analysis

Contextual Phrases versus V-DO Pairs. We found that $context_H$ significantly outperforms V-DO in terms of effort and effectiveness. Figure 5 presents the results of our comparison in a box and whisker plot. The shaded 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 '$\times$'.

In terms of effort, shown on the left, developers entered 5 more queries on average for V-DO than for $context_H$. In most cases, this was due to the fact that users found it difficult to formulate strict verb-direct object 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 $context_H$.

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 $context_H$ allowed for higher F measure values.

To verify our observation that $context_H$ outperforms V-DO, we performed a two-sample t-test [5]. Our dependent variables, number of queries (nq) and F measure (f), had unequal variances, leading us to use the Satterthwaite approximation. We found that $context_H$ outperforms V-DO with statistical significance at the $\alpha = 0.05$ level (nq: $p = 0.0004$, f: $p = 0.0021$).

Because our experiment includes repeated measures of the same subjects, the assumption that the two samples are independent does not hold. In such situations it is more appropriate to use a mixed model. When we analyzed the data as a mixed model, we achieved the same level of significance, with similar p values.

At first inspection, the F measure values appear to be low. Although we used concerns as benchmarks, the goal of the study was not to locate the entire concern, but to locate seed starting points to begin concern location. We do not expect any single query to be capable of locating all relevant items in a concern. However, an automatic program exploration technique [11, 33] can explore structural edges to locate program elements not returned by the natural language search. Because most concerns in this study contain just one or two structurally connected components, non-zero F measure values translate into fairly decent concern coverage.

Contextual Hierarchy. We also compared our contextual search technique both with and without the hierarchical topic display, $context_L$. In contrast to V-DO, we did not see a significant difference between $context_H$ and $context_L$. In fact, the distributions of the nq and f variables for both techniques are quite similar. We found that the results in this part of the experiment are complicated by user interface issues. First, when faced with a long list of signatures from $context_L$, many subjects trusted the results and simply moved on. Such behavior led to very few queries entered, and very high recall from the huge size of the result set. Second, a number of developers were frustrated by the interactive nature of $context_H$'s hierarchy. They liked the hierarchy, but disliked having to click to expand every branch. Currently, it is impossible to determine how much of an effect the phrase hierarchy and topic display has on the user without further investigation of the user interface.

Qualitative Results. At the end of the experiment, subjects were asked to comment on the techniques they used. Of the 17 that responded, 5 commented on which tool they preferred. Four of the subjects preferred using the $context_H$ technique:

> The nice thing about the reformulation technique [$context_H$] is that most of the time you put in a 1-word query and find the concern(s) quickly, because they are nicely organized in groups. This way, you definitely see other functions you might miss if your queries were longer than one word.

However, one subject disagreed:

> I felt like Part II [$context_H$] was too exact and that I should remove words from my query–to only one word, which for some reason wasn't intuitive. I've been trained to at least use two words (unless they're, like, proper names).

The subjects also suggested improvements, such as acceptable instances for stemming. Two of the subjects would have liked at least trailing 's' characters stemmed from plural nouns and third person singular verbs. One subject suggested the use of synonyms, which we plan to add in the future. Some subjects also felt that the word order restriction on the query made searching difficult; in future, the word order could be used for creating the phrase hierarchy, and not for the search mechanism.

6. Related Work

The most closely related work is Shepherd et al.'s approach to automatically extracting V-DO pairs from source code comments and identifiers for search and query recommendations [30, 31]. Our technique generalizes this approach by extracting NPs, VPs, and PPs from signatures. Another approach to query recommendation automatically suggests close matches for misspelled query terms [27]. Other static search techniques supporting natural language queries do not provide query recommendations [21, 28].

There is work on automatically extracting topic words and phrases from source code [22, 25], displaying search results in a concept lattice of keywords [26], and clustering program elements that share similar phrases [14]. 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 our approach, these approaches either filter the topics based on perceived importance to the system [14, 25, 26], or do not produce human understandable topic labels [22]. Since it is impossible to predict a priori what will be of interest to the developer, we let the developer filter the results with a natural language query, and have endeavored to keep our extracted phrases as human readable as possible.

Existing research into design recovery and reuse has also used information from identifiers [3, 6, 10, 23, 29]. However, all of these approaches require an expert-defined domain model or knowledge base, which is not available for all software systems or domains. One approach for automatic generation of domain representations has been suggested for software artifacts, but has not yet been evaluated on source code [17]. Another approach automatically constructs and categorizes reuse libraries based on comments and documentation, but does not process identifiers [18]. Michail et al. use similarity of identifier names to compare and contrast software libraries for reuse [24].

Another approach to querying source code is to use structural queries [12, 13, 34]. These techniques use structural information such as call and use relationships [12, 13], type information [13, 34], or can match exact identifier names [12, 13]. Although these techniques do not support natural language queries, they could be used in conjunction with our approach for reuse.

7. Conclusion

As software systems continue to grow and evolve, locating code for maintenance and reuse tasks becomes increasingly difficult. In this paper, we present a novel approach that provides automated support to the developer both in formulating queries and discriminating between relevant and irrelevant search results. Our contextual search approach automatically captures the context of query words in source code by extracting and generating natural language phrases from method and field signatures. These phrases naturally form a hierarchy that allows the developer to quickly identify relevant program elements by reducing the number of relevance judgments, while the phrases help the developer to formulate effective queries.

We conducted an empirical evaluation of 22 developers comparing our contextual search approach to verb-direct object, the most closely related search technique. Our results show that contextual search significantly outperforms verb-direct object in terms of effort and effectiveness. Feedback from the subject developers indicate further areas of research.

8. Acknowledgments

The authors would like to thank Giriprasad Sridhara, Amy Siu, Haley Boyd, Oana Tudor, and Keith Trnka for their useful help and comments on this paper.

References

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

[2] Roberto J. Bayardo, Jr.. Efficiently mining long patterns from databases. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 85–93, 1998.

[3] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proc. of the 15th Intl. Conf. on Software Engineering*, pages 482–498, 1993.

[4] B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Proc. of the 6th Working Conf. on Reverse Engineering*, pages 112–122, 1999.

[5] A. Dean and D. Voss. *Design and Analysis of Experiments*. Springer, New York, NY, USA, 1999.

[6] P. T. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard. Lassie—a knowledge-based software information system. In *Proc. of the 12th Intl. Conf. on Software Engineering*, pages 249–261, 1990.

[7] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho. Do crosscutting concerns cause defects? *IEEE Trans. on Software Engineering*, 34(4):497–515, 2008.

[8] G. Fischer and H. Nieper-Lemke. Helgon: extending the retrieval by reformulation paradigm. In *Proc. of the SIGCHI Conf. on Human Factors in Computing Systems*, pages 357–362, 1989.

[9] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971, 1987.

[10] S. Henninger. Using iterative refinement to find reusable software. *IEEE Software*, 11(5):48–59, 1994.

[11] E. Hill, L. Pollock, and K. Vijay-Shanker. Exploring the neighborhood with Dora to expedite software maintenance. In *Proc. of the 22nd IEEE Intl. Conf. on Automated Software Engineering*, 2007.

[12] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proc. of the 27th Intl. Conf. on Software Engineering*, 2005.

[13] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *Proc. of the 2nd Intl. Conf. on Aspect-oriented Software Development*, 2003.

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

[15] D. Lawrie, H. Feild, and D. Binkley. Extracting meaning from abbreviated identifiers. In *Proc. of the 7th IEEE Intl. Working Conf. on Source Code Analysis and Manipulation*, 2007.

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

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

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

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

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

[21] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Proc. of the 11th Working Conf. on Reverse Engineering*, 2004.

[22] G. Maskeri, S. Sarkar, and K. Heafield. Mining business topics in source code using latent dirichlet allocation. In *Proc. of the 1st India Software Engineering Conf.*, 2008.

[23] E. Merlo, I. McAdam, and R. D. Mori. Feed-forward and recurrent neural networks for source code informal information analysis. *Journal of Software Maintenance*, 15(4):205–244, 2003.

[24] A. Michail and D. Notkin. Assessing software libraries by browsing similar classes, functions and relationships. In *Proc. of the 21st Intl. Conf. on Software Engineering*, pages 463–472, 1999.

[25] M. Ohba and K. Gondow. Toward mining "concept keywords" from identifiers in large software projects. In *Proc. of the 2005 Intl. Workshop on Mining Software Repositories*, 2005.

[26] D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *Proc. of the 15th IEEE Intl. Conf. on Program Comprehension*, 2007.

[27] D. Poshyvanyk, A. Marcus, and Y. Dong. JIRiSS – an Eclipse plug-in for source code exploration. In *Proc. of the 14th Intl. Conf. on Program Comprehension*, 2006.

[28] D. Poshyvanyk, M. Petrenko, A. Marcus, X. Xie, and D. Liu. Source code exploration with Google. In *Proc. of the 22nd IEEE Intl. Conf. on Software Maintenance*, 2006.

[29] R. Prieto-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4(1):6–16, 1987.

[30] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proc. of the 6th Intl. Conf. on Aspect-oriented Software Development*, 2007.

[31] D. Shepherd, L. Pollock, and K. Vijay-Shanker. Towards supporting on-demand virtual remodularization using program graphs. In *Proc. of the 5th Intl. Conf. on Aspect-Oriented Software Development*, 2006.

[32] G. Sridhara, E. Hill, L. Pollock, and K. Vijay-Shanker. Identifying word relations in software: A comparative study of semantic similarity tools. In *Proc. of the 16th IEEE Intl. Conf. on Program Comprehension*, 2008.

[33] M. Sridharan, S. Fink, and R. Bodik. Thin slicing. In *Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2007.

[34] A. M. Zaremski and J. M. Wing. Signature matching: a tool for using software libraries. *ACM Trans. on Software Engineering and Methodology*, 4(2):146–170, 1995.