NL-based Query Refinement and Contextualized Code Search Results: A User Study

Emily Hill, Manuel Roldan-Vega, Jerry Alan Fails, Greg Mallet
Department of Computer Science
Montclair State University
Montclair, NJ, USA
{hillem, roldanvegam1, failsj, malletg1}@mail.montclair.edu

***Abstract*—As software systems continue to grow and evolve, locating code for software maintenance tasks becomes increasingly difficult. Source code search tools match a developer's keyword-style or natural language query with comments and identifiers in the source code to identify relevant methods that may need to be changed or understood to complete the maintenance task. In this search process, the developer faces a number of challenges: (1) formulating a query, (2) determining if the results are relevant, and (3) if the results are not relevant, reformulating the query. In this paper, we present a NL-based results view for searching source code for maintenance that helps address these challenges by integrating multiple feedback mechanisms into the search results view: prevalence of the query words in the result set, results grouped by NL-based information, as a result list, and suggested alternative query words. Our search technique is implemented as an Eclipse plug-in, CONQUER, and has been empirically validated by 18 Java developers. Our results show that users prefer CONQUER over a state of the art search technique, requesting customization of the interface in future query reformulation techniques.**

***Index Terms*—feature location, source code search, software maintenance**

I. Introduction

Developers spend more time finding and understanding code than making modifications during maintenance [1]. Thus, we can reduce maintenance costs by helping developers to more effectively find the code relevant to their software maintenance tasks. Similar to how we use Google to search the web, some source code search tools match a developer's query with comments and identifiers in the source code to identify relevant program elements. Source code search techniques may focus on searching for code to reuse, such as APIs or open source implementations [2], [3], [4], [5], [6], [7], or may focus on searching a single source project for maintenance by finding features or concerns to be modified or copied [8], [9], [10], [11], [12], [13], referred to as *local code search* [14]. In this paper, we are concerned with the latter problem, that of searching a single program prior to a software change, rather than searching a large repository of code for reuse.

Techniques that simply list search results often consider the words occurring in comments and identifiers individually as a bag of words [10], [11], [12]. In contrast, recent techniques have taken advantage of *phrasal concepts*, which are concepts expressed as groups of words such as noun phrases (e.g., 'cell attributes for DB') and verb phrases (e.g., 'get current task from list'). We use the term *lexical concept* to describe a concept evoked by a single word, and *phrasal concept* to describe a concept expressed as a sequence of words [15]. Phrasal concepts have been used to improve search accuracy [9] and suggest alternate query words [13]. Inspired by prior work [8], which categorizes search results in a hierarchy based on partial phrase matching, we take advantage of phrasal concepts to reorganize and filter search results and dramatically change how the results are viewed to facilitate query refinement.

In this paper, we present CONQUER, a source code search technique that organizes and presents search results to the user based on state of the art phrasal concept-based searching [9], [16]. The technique combines insights gained from the verb-DO [13] and contextual search [8] approaches, and introduces novel approaches to organizing the results, quantifying the query's relationship to the search results, and suggesting alternative query words. Our results from 18 Java developers show that users prefer CONQUER over a state of the art search technique when searching unfamiliar code. More information about the CONQUER Eclipse plug-in implementation can be found in our tool demo [16], which provides a brief (1 paragraph) summary of user feedback. In this paper, our contribution focuses on detailing this full user study evaluating the CONQUER approach.

In prior work, we presented search results in hierarchical categories of phrases [8], implemented as a proof of concept. The search mechanism used by this prototype, and upon which the hierarchical display depends, has limitations that are not trivially overcome in creating a source code search tool that is usable in practice. First, the prior search technique required all query words to be present, in order, in automatically generated phrases derived from method signature information. Thus, the hierarchical structure of the prior approach is incompatible with more flexible keyword searching, leading us to develop CONQUER's novel results view. Because the phrase grouping in prior work [8] required query words to be in a specific order, overly specific queries would return no results. In contrast, CONQUER is built on the more flexible SWUM-search framework [9], and is not subject to such limitations.

II. Background and Motivation

In order to provide the developers useful context on how the query words are used in the source code, our approach

leverages phrasal concepts, which are automatically derived from linguistic factors such as the action and theme of a method signature. The action represents the verb or main activity of the method, whereas the theme is the direct object of that action. In this section, we motivate the need for query reformulation support, discuss why phrasal concepts like action and theme are so important to local code search and how they can be used to improve source code search accuracy.

A. The Need for Query Reformulation Support

Existing source code search techniques typically list the search results as a list of files, method signatures, or lines where the query matched (i.e., Eclipse file search or grep), and display the results by decreasing relevance [10], [11], [12], [9]. As has been shown in prior work [8], simply listing the search results can increase the burden on the developer by requiring unnecessary comprehension and relevance judgements to (1) determine if the results returned are relevant at all, and (2) determine where the relevant results are in the list. These types of result displays are very difficult to skim quickly, meaning that the developer must comprehend snippets of code or method names before determining whether or not the query even matches their information need. This high cognitive load is in addition to the developer needing to comprehend the code necessary for the actual maintenance task itself, and should be minimized as much as possible.

When a user searches source code with a natural language or keyword-style query, there are three possible outcomes:

1) The query is ideal, and the user can quickly hone in on the relevant search results.
2) The query is close to ideal, but is either overly general and returns too many results, needing additional words to specialize the query further, or the query contains a mixture of ideal and non-ideal query words, requiring substituting non-ideal query words with ideal words.
3) The query is completely inaccurate, and needs alternative words related to the information need.

Our research focuses on how to design a single search results view interface to meet these three possible outcomes, and present the search results in such a way that the user can (1) quickly determine if the results are relevant, (2) quickly find relevant results, or (3) see alternate query words to help them further refine their query if it is overly general or inaccurate. Our key insight is to consider the search problem from the perspective of the *query*, rather than the *information need*, because the same query may be used to search for multiple distinct information needs.

For example, consider two related, but distinct, features in the music management software, Jajuk, which is similar to iTunes. The "smart shuffle" feature shuffles the entire music collection and adds all the songs to the current playlist queue. In contrast, the "shuffle playlist" feature takes the songs on the current playlist queue and randomly shuffles their order. Each feature contains approximately 8-12 methods in its implementation, and although the features seem similar, they only share one utility method (getRandom), and a generic GUI update method that handles many user events such as play, pause, repeat mode, increasing or decreasing volume, etc. Although these two features represent different information needs, a developer might reasonably begin with the query "shuffle" or "random" for either feature. However, the ideal query for smart shuffle is "global random shuffle" and "shuffle queue" for shuffle playlist. Our goal is to design a *single* search results view interface that can be used for the same query with different information needs, and still effectively help the developer quickly and easily refine their query or locate relevant results.

B. The Importance of Actions and Themes in Code Search

Most existing search techniques use lexical concepts, that is, they treat a program as a "bag of words". In cases where bag of words would return many results, phrasal concepts can differentiate between the relevant and irrelevant results.

For example, consider searching for code related to the concept of *adding items to a cart in a shopping system*. Figure 1 shows three methods returned by a bag of words technique when searching for the "add item" query. The addEntry method, m_1, is highly relevant to the target concept of adding an item, containing occurrences of the "add item" phrasal concept in both the method name and body statements. In contrast, sum and loadAllItemsFromURLString, m_2 and m_3, respectively, are not relevant to the target concept, despite occurrences of both the words "add" and "item" in the source code. Specifically, sum uses a different sense of the word "add", and contains the phrasal concept "add price" rather than "add item". Although loadAllItemsFromURLString contains the "add item" phrasal concept, we can conclude form m_3's method name that the main intent of this method is to load items, during which "adding items" is just one substep.

In summary, this example demonstrates that knowing how words occur together in the action and theme and where they occur can distinguish between relevant and irrelevant search results, which we leverage in the CONQUER results view. We capture the action and theme phrasal concepts for a given method using the Software Word Usage Model (SWUM) [17].

C. Finding Relevant Methods with Phrasal Concepts

Phrasal concepts derived by SWUM have been used to improve source code search [9][1]. In the current work, we take advantage of this SWUM-based scoring function, which integrates the following sources of information (i.e., search signals) to determine a method's relevance to the query:

- *Location.* When a method is well-named, its signature summarizes its intent, while the body implements it using a variety of words that may be unrelated. A query word in the signature is a stronger indicator of relevance than the body.
- *Semantic role.* Prior research has shown that using semantic roles such as action and theme can improve search effectiveness [13]. The SWUM-based score makes use of this insight

[1]Identifiers are split using conservative camel case.

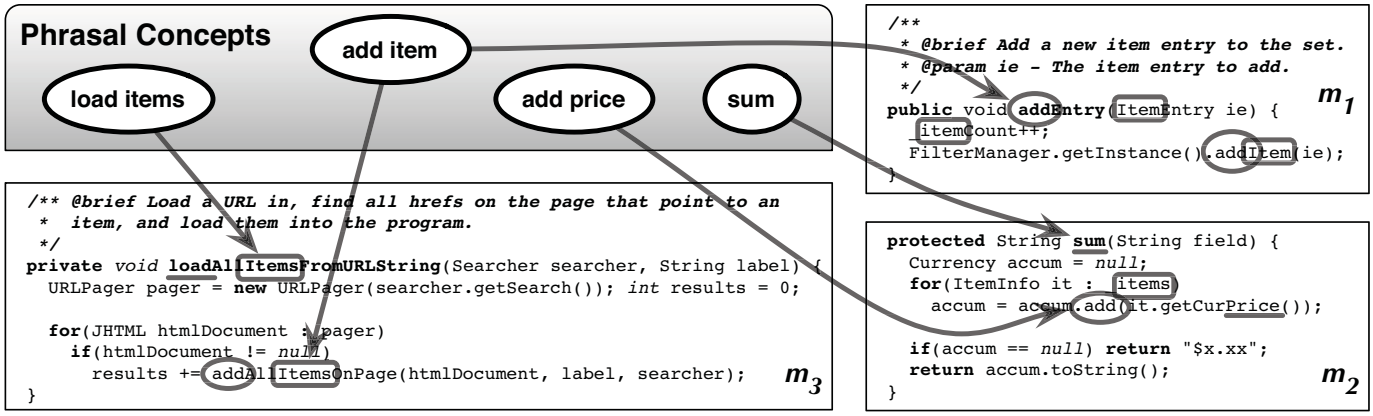


Figure 1. Search results and phrasal concepts for "add item" query. All the methods returned by the search contain the words "add" and "item", but "add item" is not necessarily the main action taken by each method.

and leverages SWUM's advanced extraction rules to increase the accuracy of action-theme extraction.

- *Head distance.* The closer a query word occurs to the head, or right-most, position of a phrase, the more strongly the phrase relates to the query word. For example, the phrase "image file" is more relevant to "saving a file" than "file server manager".
- *Usage.* If a query word frequently occurs throughout the rest of the program, it is not as good at discriminating between relevant and irrelevant results. This idea is commonly used in information retrieval techniques [18].

In an evaluation of state of the art search techniques, the SWUM-based search mechanism was consistently ranked more highly overall [9].

III. The CONQUER Approach

We implemented our approach as an Eclipse plug-in [16], CONQUER, for searching Java projects[2]. The search is integrated into Eclipse's general search dialog box with a CONQUER tab. After the developer enters a natural language query (i.e., a series of keywords), the plug-in initiates the SWUM-based search mechanism [9]. The results are then preprocessed before being displayed to the user. We designed the CONQUER results display to address 3 key challenges:

1) Quickly determine if results are relevant (i.e., did the query work?).
2) Find alternative query words to refine overly general queries or substitute words in a partially correct query.
3) Find relevant results quickly.

Figure 2 shows an example CONQUER results view for the query "shuffle global playlist". At the top of the results view (A), we include numbers indicating the prevalence of the query words in dominant positions in the method signatures. Next, the view is split into portions for action and theme organization. In each portion, alternative action or theme query words are displayed above the search results (B), which are presented in nested trees organized by action and theme (C). Finally, the results are listed in order of relevance score (D), preceded by a short, natural language phrase that describes the search result's action and theme to facilitates quick skimming of the results. The method signature of each result is displayed as a clickable Eclipse Java element, allowing the developer to directly navigate to the source code of a selected method.

The driving insight for our approach is that the same query might be used to search for multiple information needs. Our challenge is in designing a results view that meets the needs of developers searching with queries that are ideal, overly general, partially correct, or completely incorrect. In developing our approach, we began with a set of 10 search tasks with ideal queries across 5 different Java programs. Next, we listed a number of possible queries a developer might reasonably begin with in creating a query to search for that feature, making sure to take advantage of overlapping query words between distinct features. This set of search tasks and queries formed the basis of our training set in developing the various components of the results view described in the following section.

A. Prevalence of Query Words

The first key challenge we are trying to address is determining whether the query returns relevant results. We begin communicating relevance to the developer right at the top of the display by indicating the frequency of the query words in the action and theme position in the method signatures, as shown in Figure 2(A). The relative frequency of query words roughly indicates the success of the query in meeting the developer's information need.

For example, consider the query "mute music" searching for the "mute" feature in music management software, where the concept of 'mute' is more important to the feature than 'music.' If 'music' is much more prevalent in the source code than 'mute', it will fill the search results with irrelevant entries, causing more work for the developer. The frequencies next to each query word approximate each word's power to discriminate between relevant and irrelevant results. For instance, if 'music' occurs 100 times in the result set and

[2]Available at lee.cs.montclair.edu/~hillem/CONQUER/

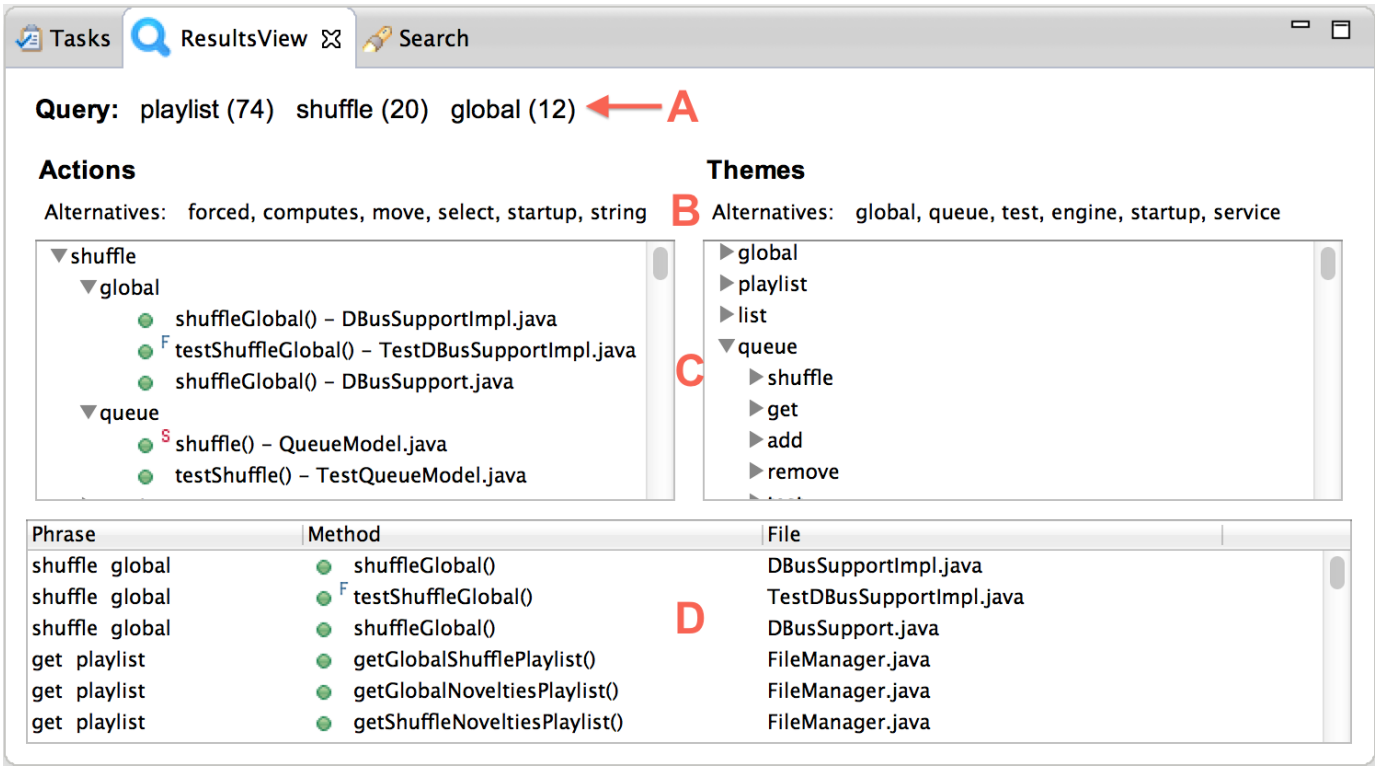


Figure 2. CONQUER results view for the "shuffle global playlist" query. The ideal query for this search task is "shuffle queue". The components of the results view include: (A) prevalence of query words, (B) suggested alternative query words, (C) categorizing by action or theme, and (D) result phrase list.

'mute' just 4 times, the developer will know to revise the query to focus on 'mute', since 'music' is likely to occur with irrelevant results. This will also help the developer focus on the concept of 'mute' in the other parts of the results view.

We use the occurrence of words in prominent positions in the method signatures over raw frequency values because it is possible for a word to occur many times in positions indicating weaker relevance in the code. For example, the word 'return' may be included many times throughout the code in keywords, but rarely in a method signature. In addition, the frequency in actions and themes affects the organization of the remaining results view components. If the developer sees from the frequencies that the query words are out of balance in terms of her information need, she will better understand how to interpret the results in the remaining views.

B. Suggested Alternative Query Words

Considering developers may select accurate query words just 10-15% of the time [19], it is likely that a developer will need to refine his query. Our goal is to suggest alternative query words that appear in the source code base. We use different approaches for recommending alternate action and theme query words because verbs and nouns are used differently in source code. Except for overriding method names, verbs tend to be used with more variety and synonyms in source code, whereas nouns tend to be better discriminators to identify relevant documents [20]. The suggestions leverage the relative frequency of the query words within the result set and the program. In our development set, we observed that queries needing more specific or substituting alternate query words relied on verbs that frequently occur with the query words in the action-theme pairs in the result set, but occur less frequently in the rest of the program (i.e., can be good discriminators). In contrast, the alternate themes in our result set frequently occur in the action-theme pairs in the result set with the query words.

For example, consider searching for the shuffling global playlist feature in a music player. A developer might first enter the query "shuffle global playlist", where the ideal query for this particular search task is "shuffle queue". In this example, 'queue' would be suggested as an alternative word, as shown in Figure 2(B). The dominance of 'playlist' in the search results (as indicated in the query section of the view), will help the developer refine the query to "shuffle global queue", getting much closer to the ideal query.

C. Categorizing by Action or Theme

When a query is not ideal, there still may be relevant results buried in the result list. We introduce two hierarchies of results, organized by action and theme, to allow more opportunities for a variety of search results to be displayed in the initial result view. For example in Figure 2(C), the 'shuffle' query word causes the shuffle category to be top-ranked in the action view. This verb only occurs with two themes: global and queue. Although 'global' is in the query, a more ideal query word for this search task is 'queue'. Even though the query is less than

ideal, the relevant method QueueModel.shuffle() is only the fourth method in the action view. In addition, the action and theme views help organize the search results to jump around to different areas of the results, based on the appropriateness of the query words while exploring the result set.

To construct the action and theme hierarchies, we break the search relevance score down into its action and theme components, which are combined together into the final relevance score which determines the result phrase list order in the bottom component of the results view. In the action view, we group all the results by action, and sort the groups by decreasing action score so that the most relevant actions are at the top of the list. Below the action level we group the remaining results by theme, sorted by decreasing theme score. If a group of actions or themes contains multiple items with different action/theme scores, we sort by the maximum action/theme score of the group. The theme view is constructed in a similar fashion, with the first level of the hierarchy based on the theme, and the second level based on the actions.

D. Result Phrase List

The result phrase list is closer to a more traditional search results view, simply listing the methods and files where matches occur, shown in Figure 2(D). However, we go beyond a simple list view by providing a short natural language phrase that describes each method, to make the results easier to quickly skim without requiring the cognitive overhead of mentally parsing the syntax of method signatures and file names. This facilitates rapidly scrolling through the results, to see what types of words and phrases describe the methods appearing in the result set. For an ideal query, this section will contain the relevant results at the top of the list.

In this portion of the results view, we report query matches at the method and file level. This is in contrast to a grep-like search, such as Eclipse's built-in File Search, that displays the files and lines matching the query. We believe that in larger software systems (over 20 KLOC), where source code search for maintenance is more likely to be needed, methods will have meaningful names [21]. Thus, we focus on summarizing the methods' main actions and themes in our phrase list view, rather than summarizing the individual line matches.

IV. Evaluation

Since the underlying SWUM-based search mechanism has been evaluated in prior work [9], we focus our evaluation on the main contribution of the current work, namely, the search interface. To evaluate this, we compared CONQUER search (CONQUER) with two baseline methods: Eclipse's built-in File Search (ECLIPSE) and the same SWUM-based search mechanism as CONQUER with only the Result Phrase List view (called SCORE). In this study, we compare feedback from 18 developers performing 28 source code search tasks.

A. Search Mechanisms

The independent variable in our study is the search technique: CONQUER, SCORE, and ECLIPSE. The CONQUER search technique is the full CONQUER approach described in Section III. As a baseline, we compare CONQUER to the built-in Eclipse File Search (ECLIPSE) that uses wild card queries similar to UNIX grep. To separate out the effect of CONQUER's results view interface versus the SWUM-based search mechanism, we also created a stripped-down version of CONQUER's results view interface, called SCORE. The SCORE results view only includes the phrase list (e.g., bottom of Fig. 2), and not the action or theme hierarchies, the suggested alternative query words, nor prevalence of query words.

B. Participants

Participants were invited to participate via e-mail and online message boards, focusing on attracting Java developers with industry experience. In total, we obtained results from 18 volunteer software developers with varying levels of programming and industry experience. Table I 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. We initially recruited 28 subjects, however only 18 completed the study (some of the subjects had trouble installing the plug-in). The developers had little to no prior knowledge of the subject applications.

C. Search Tasks

We used 28 search tasks in the study that have been used in previous experiments [8], broken into 7 groups of 4 search tasks. Because the description of the search tasks can add significant variability to the study, we used search tasks from two different sets, which have different types of descriptions. The first set of 19 tasks is from the 45 KLOC JavaScript/ECMAScript interpreter and compiler, Rhino. Each search task maps to a feature described by a subsection of the documentation, which is used as the task description that is given to subjects before formulating a query.

The second set of search tasks consists of 9 user-observable, action-oriented features from 4 programs ranging in size from 23 to 75 KLOC [13]. 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 search task descriptions consist of screen shots of each feature being executed. The participants are asked to look at a screen shot and then formulate a query to search for the code that implements the feature.

Both sets of search tasks were derived by two groups of independent researchers, and have been used as search tasks in previous evaluations [8], [12], [13], [22], [23]. It should be noted that as a compiler, Rhino is out of most of our developers' familiar domain. In addition, it is known from previous experience that the search tasks from javaHMO and Jajuk are implemented using different words than those that are visible in the user interface, which is used for the task description. Thus, the queries from these two programs

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

Table I
SUBJECT DEVELOPER CHARACTERISTICS: NUMBER OF YEARS OF EXPERIENCE (LEFT) AND MAINTENANCE FREQUENCY (RIGHT)

may require more query reformulation, since it will be more challenging for participants to 'guess' the ideal query. Each block of 4 search tasks includes 1–2 tasks with screen shots and 2–3 tasks with documentation as the task description.

D. Measures

We compare the 3 search tools by measuring user preferences in terms of 4 factors: enjoyment, effectiveness, ease of use, and likelihood of future use. We measure these factors using the questions administered as a follow-up survey:

1) How much did you enjoy using the search tool?
2) Rate how effective the search tool was in helping you complete the search tasks
3) How easy or complicated is the interface to use?
4) How likely are you to use this search technique in your own work?

Each response is measured using a 7-point Likert scale, where 1 is worst (negative) and 7 best (positive). Since subjects are given multiple tools to work with, we can compare the relative difference between each technique per user. We also measure user preferences in terms of the specific components of the CONQUER results view using a 7-point Likert scale:

1) How helpful was the action view in helping you complete the search tasks?
2) How helpful was the theme view in helping you complete the search tasks?
3) How helpful was the phrase list view in helping you complete the search tasks?
4) How helpful were the alternative query word suggestions in helping you complete the search tasks?

In addition to the above quantifiable questions, we asked a number of open-ended questions about user preferences in using the various search techniques.

E. Design

In order to maximize the number of participants in the study with industry development experience, we sought to minimize the overall study time to within 30-60 minutes. To meet this goal, we asked each subject to use two of the three search tools in the study, on a total of 8 search tasks (4 per search technique). We used 4 tasks to minimize the potential variations due to search task difficulty, and keep the time required of each participant within the desired range.

Thus, in the design there were three main blocking factors: the search tools used, the set of search tasks used, and the order the tools were used (to control for learning effects). This gives us 6 possible combinations of two out of three search tools, and 42 possible combinations of search task blocks. Given that a full factorial design was impossible, we designed the tool and search task combinations so that every possible combination of tool orderings was repeated in blocks of 6, and randomly assigned search task groups such that each technique was used with each search task as evenly as possible, with approximately equal numbers using the first and second search sets. (That is, we tried to avoid the first search task set only occurring with the first tool of the study.)

Participants were assigned to a random group as they volunteered. Because not all the participants completed the study, we do not have an equal number of replications for each combination. All the experimental materials, including the instructions, are available online: lee.cs.montclair.edu/∼hillem/CONQEvalSite.

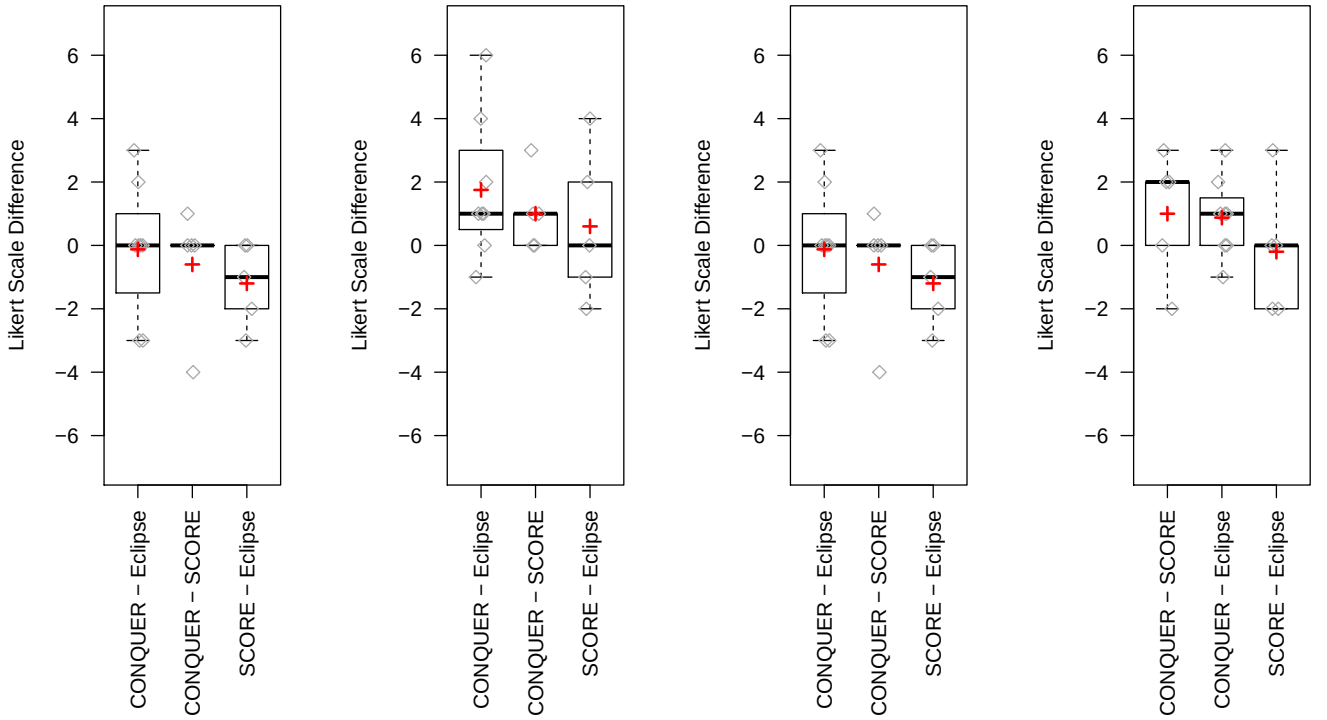
F. Methodology

After agreeing to participate via e-mail or online message, participants were e-mailed detailed instructions for completing the study. First, participants were asked to install the Eclipse plug-in, containing all 3 tools in the study, and to import the Java projects to be searched. When installing the plug-in, all developers were asked to complete an example search task to familiarize themselves with the experimental procedure. Next, participants were asked to formally agree to participate in the study by beginning the online survey that served to collect most of the information in the study.

The study itself consists of viewing instructions and screen shots of the first search tool, and attempting to formulate queries for the four search tasks listed. This process was repeated for the second search tool. Participants were then asked to complete the survey. As part of the survey, they were asked to select a menu option in the Eclipse plug-in interface that would anonymously e-mail the results. Anonymity was preserved in this process by asking the user to select an anonymous identifier to enter into the survey, and this same identifier was requested when sending the e-mail results.

G. Threats to Validity

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



(a) Easiness of interface (b) How effective was search tool (c) Level of enjoyment (d) How likely to use in own work

Figure 3. Differences in user preferences according to a 7-point Likert scale. The values are differences between user preferences of each technique combination. The box plots, which show the median and quartiles, are overlaid with the actual data (diamonds) and mean (plus).

The search tasks are an unavoidable threat. To minimize the effect that some tasks are more difficult to locate and formulate queries for, we used search tasks from 5 different programs with two different types of descriptions. In addition, each participant applied each search tool to 4 tasks to avoid any one task dominating the results and ensure comparability across sets of search tasks. However, it is possible that the task groups that we randomly selected were not of equivalent difficulty. We avoided this as much as possible by ensuring that each group contained search tasks from at least 2 different programs, under the assumption that tasks from the same program will be of approximately the same difficulty.

The experiment was administered as a volunteer online survey and electronically distributed plug-in to gain access to as many developers with industry experience as possible. This means that participants are not in a controlled environment, and other distractions may have influenced the attention that subjects devoted to the experiment.

Because the study was administered as an Eclipse plug-in and online survey, rather than in a controlled lab setting, there were some challenges in reconciling the data after collection. The Eclipse plug-in logged every query entered into one of the 3 search tools. However, especially for ECLIPSE, participants occasionally searched through the primary Eclipse search, rather than the one we distributed with logging enabled. In the online survey, participants were asked to enter in the best queries they found for each search task. Due to the more naturalistic setting, there were a number of inconsistencies between the queries logged by the plug-in and those entered into the survey. For example, some of the queries entered in the survey were never logged by Eclipse, and so were never executed or executed using an unknown search mechanism. Not all participants entered queries for all 8 search tasks assigned. Because of these inconsistencies in the data, we were unable to analyze search effectiveness through traditional IR measures or via the number of queries entered. Instead, we focus on the most reliable data we have, which is the direct user feedback on the relative advantages of each tool.

V. RESULTS AND DISCUSSION

Because not all the participants who volunteered were able to successfully complete the study, we do not have an even number of replications. The CONQUER and ECLIPSE techniques were each used 13 times, whereas SCORE was only used 10 times. Thus, we have 8 observations for analyzing differences between CONQUER and ECLIPSE, with 5 observations each for comparing differences with SCORE−ECLIPSE and CONQUER−SCORE.

Figure 3 shows the overall difference in results between pairs of tools for the 4 factors measured: ease of use, effectiveness, enjoyment, and likeliness to use in work in future. On the x-axis are the pairs of tools, and on the y-axis the difference in ratings based on a Likert scale. For instance, a value of 2 for CONQUER−ECLIPSE implies that the user

preferred CONQUER over ECLIPSE by 2 out of a 7-point scale. Similarly, a value of -4 for CONQUER$-$SCORE indicates that the user preferred SCORE over CONQUER by 4 out of a 7-point scale. In taking the differences, we ordered the search techniques based on our hypothesis that the users would prefer CONQUER, followed by SCORE, and ECLIPSE.

The results in Figure 3 are displayed as box plots where the thick horizontal line represents the median, the plus represents the mean, and the box indicates the interquartile range (i.e., the difference between the first and third quartiles). The whiskers extend to the maximum and minimum values of the range, except for outliers. The box plots are overlaid with the actual data points from the data set, indicated by $\diamond$.

We did not observe the mean ratings for each search tool to be statistically significantly different. In addition, we used a paired t-test to verify that there was no significant difference in any of the factors between the search tools used first or second. Thus, we did not observe any learning effects.

A. Quantitative Results

From Figure 3(a) we see that the CONQUER ease of use could be improved. Given the mean difference is close to 0, it appears that just as many subjects thought CONQUER was easier to use as compared to ECLIPSE. Subjects overwhelmingly preferred ECLIPSE search over the simplified SCORE interface. Figure 3(c) shows a similar trend for enjoyment.

Figures 3(b) and 3(d) more closely follow our expected trend. In terms of both effectiveness and likelihood of using in future work, CONQUER is clearly preferred over ECLIPSE, and more modestly outranking SCORE. However, the relationship between SCORE and ECLIPSE is less clear. Based on effectiveness, subjects seem to prefer SCORE, but in terms of using any of the techniques in future, users predominantly prefer the more familiar ECLIPSE search.

Figure 4 shows user preferences for the individual components of the CONQUER results view, using the same box plot representation as in Figure 3. The difference is that the y-axis represents a raw Likert scale, rather than the difference of two scales. Thus, a value of 4 indicates neutrality. The individual box plots are ordered by increasing mean, and thus the x-axis gives us a view of the relative preference of users for each component of the CONQUER results view. The alternative query words have the highest mean, with the phrase list view as a close second. This is interesting in light of the fact that the SCORE search, which only consists of the phrase list, was not well regarded overall. The query numbers view has the lowest score. Although it appears that users preferred to see the query words in the results view (a common complaint about SCORE), the query numbers do not appear as useful as the other additions to the interface. This may be because the subjects had little intuition as to what the numbers meant.

B. Qualitative Feedback

Many of the users appreciated the alternative suggested query words as well as the action and theme tree view:

> "I was actually really surprised at how well [CONQUER] worked. At first, I didn't think it would be useful, even compared to [SCORE]. However, the tool seemed to actively detect synonyms that I wouldn't have used at first (construct wasn't my first guess, but it was picked up right away by the tool)."

Users responded positively with "the summary trees help faster navigation" and "Query recommendations are $*very*$ helpful and important when searching unfamiliar code." However, not every user found all aspects of the interface to be intuitive:

> "I didn't really understand the action vs theme view. For a while I thought I should be able to double-click to replace with the synonyms and didn't notice that they were in a treeview. After a while I realized you could drill down to a fully refined query."

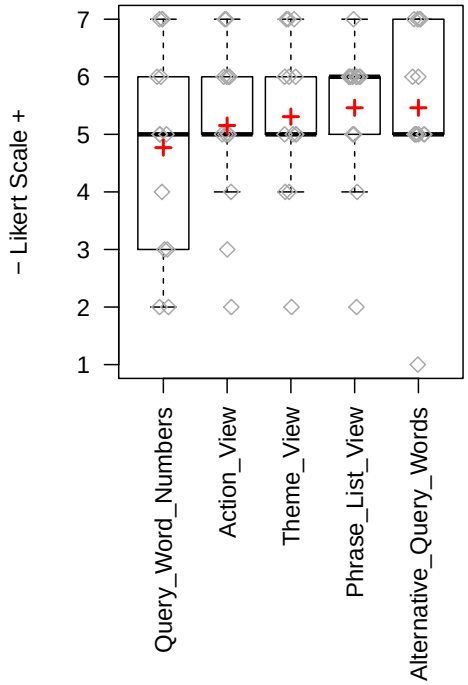


Figure 4. User preferences for individual CONQUER results view components (7 is best, 1 is worst).

The CONQUER responses for improvement primarily focused on two things: search behavior and customizability of the query interface. In terms of search behavior, multiple subjects wanted the ability to match all keywords or match entire identifiers. CONQUER by default splits identifiers, which made the search problem more challenging for some search tasks, especially when the documentation (i.e., feature description) included specific identifiers to search for. In these instances, the users overwhelmingly requested the ability to match specific strings:

> "I did not like that I could not make specific string searches (match case). What I mean is that if i want

to search for (toString) and not (to)(String) I could possibly do it by searching like: 'toString' (like you could on google) with quotes around the keyword."

In terms of the interface, users wanted more variety, more flexibility, and more control:

> "Query recommendations should be surfaced via a different UI technique (autocomplete or suggestions link below search box after searching, like google)."

> "Provide sort options for Actions and Themes (alphabetic order, relevance order, etc...)."

In addition, some users disliked how much space the results view uses: "Takes up way too much screen real estate." By allowing more flexibility and customization in the interface, users could better tailor the view to fit their workflow.

One user, who is also an avid googler, felt they would recommend the CONQUER search tool to colleagues:

> "Most of my queries narrowed the results to < 7. Compared to the actual number of generic search results, this tool shows vast potential. If released, I would totally recommend this to some of my work colleagues who face similar issues concerning code."

Like CONQUER, SCORE and ECLIPSE saw mixed results. ECLIPSE is familiar, and thus many participants found it easy to use and uncomplicated. Some especially liked the straightforward results view: "I liked seeing the list of class names and the line numbers and the context of where my keywords were found." Similarly, some participants preferred the simplicity of the SCORE interface: "It seemed like a very simple, bare-bones tool that would be easy to use quickly," "The simplicity of the tool was nice and made for quick searching"; whereas some found the phrases misleading:

> "I found the phrase distracting. Maybe it was the searches that I was doing, but the phrase wasn't as helpful to me as seeing where my words were showing up in the code. The phrases seemed useful for some methods but not all. I'm not sure that I used the phrases that often when I was trying to verify my results."

Likewise, some participants found the ECLIPSE queries difficult to use: "Touchy results. Make one wrong move and your search results get it."

C. Discussion

In analyzing the results, we observe that there are certain situations that lend themselves to each search mechanism. When a developer has an idea of the appropriate identifier names to search for, they want to perform strict matching of identifiers or keywords with an ECLIPSE-like search. In contrast, when a developer is not familiar with a codebase or its naming conventions, and has no insight into what identifiers would be relevant, they need the support of a natural language search provided by SCORE or CONQUER. Providing a flexible interface for either scenario will further enable the developer to use a single search interface for all their search needs. Future work will investigate how to integrate that customizability in an intuitive way, without using too much screen real estate.

In general, participants seemed to appreciate the alternative query words suggested. Some participants requested suggesting synonyms as well co-occurring words. In future, we would like to explore using synonyms [24], [23] to enhance the suggested alternative query words.

VI. Related Work

Traditional search techniques simply list the search results in order of decreasing relevance [9], [10], [11], [12]. In contrast, we focus on intelligently displaying search results to the user to quickly determine result relevance, find relevant results, and refine inaccurate queries.

In prior work we developed an approach to extract natural language phrases from method signatures to help developers reformulate better queries and group search results [8]. Although these hierarchical categories are similar in spirit to CONQUER's action and theme views, they rely on completely different approaches. First, the prior search technique required all query words to be present, in order, in automatically generated phrases derived from method signature information. Counter-intuitively, better results were obtained using *fewer* keywords, and no information from method bodies or comments was used in the search mechanism. Thus, the hierarchical structure of the prior approach is incompatible with more flexible keyword searching, leading us to develop CONQUER's novel results view. The phrase grouping in prior work [8] required query words to be in a specific order, and overly specific queries would return no results. In contrast, CONQUER is built on the more flexible SWUM-search framework [9], and is not subject to such limitations. Lastly, the prior approach did not suggest any alternative words.

Another closely related work is FindConcept, which automatically extracts verb-DO pairs (similar to our action-theme phrasal concepts) from source code comments and identifiers for source code search and query recommendations [13]. Our approach makes use of insights gained during the course of FindConcept's research, and is based on next generation action and theme extraction using SWUM [17], which has been used to improve source code search accuracy [9]. While both techniques take advantage of the occurrences of actions and themes in source code to recommend alternative query words, the methodologies for determining alternative query words are different. In addition, CONQUER introduces an integrated results view that simplifies FindConcept's multi-step query refinement process that includes iteratively adding recommended verbs, direct objects and synonyms, before finally executing the query. In contrast, CONQUER is designed to allow for rapidly refining the query, with alternatives suggested within the interface as needed.

Structural searching using natural language queries [25] targets a different problem. With this type of search, the user enters in a natural language (NL) query such as, "where is this method called?" or "what fields are declared of this type?", the NL query is parsed, and the appropriate structural model of the

program is searched to answer the low-level structural question. These queries represent traditional structural queries [26], [27], but expressed in natural language. In contrast, our goal is to find code that implements a high-level concept or feature of the software system. Once the developer has located the code related to a software maintenance task, a structural information query can be used to find other related code or determine the impact of a proposed change.

There is also work on automatically extracting topic words and phrases from source code [28], [29], [30], displaying search results in a concept lattice of keywords [12], and clustering program elements that share similar phrases [31]. These techniques could be used in conjunction with phrasal concepts to help refine phrases and organize search results.

VII. Conclusion

In this paper, we present CONQUER, a novel source code search interface that organizes and presents search results using a novel NL-based results view that integrates multiple feedback mechanisms into the search results view: prevalence of the query words in the result set, results grouped by NL-based information, traditional result list, and suggested alternative query words. We empirically validated our CONQUER approach, which is implemented as an Eclipse plug-in, based on user feedback from 18 Java developers. In general, participants tend to prefer CONQUER over a grep-like search, but would prefer the results view interface to be simplified and made more customizable by the user. Our results show that CONQUER is a promising platform from which to continue refining a results view interface for query refinement and contextualizing search results.

In the future, we plan to investigate automatically suggesting synonyms [24], [23] and making the interface more intuitive. Although our evaluation is based on Java, the underlying technology is also implemented for C++. In future work, we anticipate integrating our approach into a Visual Studio search framework such as Sando [14].

References

[1] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during Soft. maintenance tasks," *IEEE Transactions on Soft. Eng.*, vol. 32, no. 12, 2006.

[2] S. K. Bajracharya, J. Ossher, and C. V. Lopes, "Leveraging usage similarity for effective retrieval of examples in code repositories," in *Proc. 18th ACM SIGSOFT Int'l Symposium on Foundations of Soft. Eng.*, 2010.

[3] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: integrating web search into the development environment," in *Proc. SIGCHI Conf. on Human Factors in Computing Systems*, 2010.

[4] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: a search engine for finding functions and their usages," in *Proc. 33rd Int'l Conf. on Soft. Eng.*, 2011.

[5] J. Stylos and B. A. Myers, "Mica: A web-search tool for finding api components and examples," in *Proc. Visual Languages and Human-Centric Computing*, 2006.

[6] L. Wang, L. Fang, L. Wang, G. Li, B. Xie, and F. Yang, "Apiexample: An effective web search based usage example recommendation system for java apis," in *Proc. 26th IEEE/ACM Int'l Conf. on Automated Soft. Eng.*, 2011.

[7] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," in *Proc. 23rd European Conf. on Object-Oriented Programming*, 2009.

[8] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of NL-queries for Soft. maintenance and reuse," in *Proc. 31st Int'l Conf. on Soft. Eng.*, 2009.

[9] ——, "Improving source code search with natural language phrasal representations of method signatures," in *Proc. 26th IEEE Int'l Conf. on Automated Soft. Eng.*, 2011.

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

[11] D. Poshyvanyk, M. Petrenko, A. Marcus, X. Xie, and D. Liu, "Source code exploration with Google," in *Proc. 22nd IEEE Int'l Conf. on Soft. Maintenance*, 2006.

[12] D. Poshyvanyk, M. Gethers, and A. Marcus, "Concept location using formal concept analysis and information retrieval," *ACM Transactions on Soft. Eng. and Methodology*, vol. 21, no. 4, 2012.

[13] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, "Using natural language program analysis to locate and understand action-oriented concerns," in *Proc. 6th Int'l Conf. on Aspect-Oriented Soft. Development*, 2007.

[14] D. Shepherd, K. Damevski, B. Ropski, and T. Fritz, "Sando: an extensible local code search framework," in *Proc. ACM SIGSOFT 20th Int'l Symposium on the Foundations of Soft. Eng.*, 2012.

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

[16] M. Roldan-Vega, G. Mallet, E. Hill, and J. Fails, "CONQUER: A tool for NL-based query refinement and contextualizing source code search results," in *Proc. 29th IEEE Int'l Conf. on Soft. Maintenance*, 2013.

[17] E. Hill, "Integrating natural language and program structure information to improve Soft. search and exploration," Ph.D. dissertation, University of Delaware, Aug. 2010.

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

[19] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais, "The vocabulary problem in human-system communication," *Communications of the ACM*, vol. 30, no. 11, 1987.

[20] G. Capobianco, A. De Lucia, R. Oliveto, A. Panichella, and S. Panichella, "On the role of the nouns in IR-based traceability recovery," in *IEEE 17th Int'l Conf. on Program Comprehension*, 2009.

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

[22] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho, "Do crosscutting concerns cause defects?" *IEEE Transactions on Soft. Eng.*, vol. 34, no. 4, 2008.

[23] J. Yang and L. Tan, "Inferring semantically related words from Soft. context," in *Proc. 9th IEEE Working Conf. on Mining Soft. Repositories*, 2012.

[24] M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker, "Automatically mining Soft.-based, semantically-similar words from comment-code mappings," in *Proc. 10th Working Conf. on Mining Soft. Repositories*, 2013.

[25] M. Würsch, G. Ghezzi, G. Reif, and H. C. Gall, "Supporting developers with natural language queries," in *Proc. 32nd Int'l Conf. on Soft. Eng.*, 2010.

[26] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *Proc. 27th Int'l Conf. on Soft. Eng.*, 2005.

[27] D. Janzen and K. D. Volder, "Navigating and querying code without getting lost," in *Proc. 2nd Int'l Conf. on Aspect-Oriented Soft. Development*, 2003.

[28] A. Hindle, N. A. Ernst, M. W. Godfrey, and J. Mylopoulos, "Automated topic naming to support cross-project analysis of Soft. maintenance activities," in *Proc. 8th Working Conf. on Mining Soft. Repositories*, 2011.

[29] G. Maskeri, S. Sarkar, and K. Heafield, "Mining business topics in source code using latent dirichlet allocation," in *Proc. 1st India Soft. Eng. Conf.*, 2008.

[30] M. Ohba and K. Gondow, "Toward mining "concept keywords" from identifiers in large Soft. projects," in *Proc. Int'l Workshop on Mining Soft. Repositories*, 2005.

[31] A. Kuhn, S. Ducasse, and T. Gírba, "Semantic clustering: Identifying topics in source code," *Information Systems and Technologies*, vol. 49, no. 3, 2007.