

Searching and Skimming: An Exploratory Study

Jamie Starke, Chris Luce, Jonathan Sillito
University of Calgary
Calgary, Canada
{jrstarke,cluce,sillito}@ucalgary.ca

Abstract

Source code search is an important activity for programmers working on a change task to a software system. As part of a larger project to improve tool support for finding information in source code, we conducted a formative study in which programmers were asked to perform corrective tasks to a system they were initially unfamiliar with. Our analysis focused specifically on how programmers decide what to search for, and how they decide which results are relevant to their task. Based on our analysis, we present five observations about our participant's approach to finding information and some of the challenges they faced. We also discuss the implications these observations have for the design of source code search tools.

1. Introduction

Source code search is an important activity for programmers working on a change task to a software system. A range of search tools are bundled with today's Integrated Development Environments. Research studies have been conducted to explore aspects of search as it relates to change task activities, including studies focused on how programmers perform change tasks using today's tools (e.g., [9]) and developing models of comprehension (e.g., [21]). Other studies have identified the questions programmers ask [15, 16] and explored their information needs as they perform change tasks [7].

To build on this previous work, we have embarked on a research program that aims to improve tool support for finding information in source code. As a first step in this research we have conducted a study with the aim of answering two specific research questions: (1) how do programmers turn their information needs into concrete searches? and (2) how do programmers work with the search results to find the information they are seeking? We are particularly interested in understanding the challenges programmers face and how tool support can be improved.

Our study involved ten programmers working on assigned change tasks to a large sized code base (70KLOC). Our study was conducted in a laboratory setting; however, care had been taken to ensure that the tasks were realistic. As our participants performed the task we gathered qualitative and quantitative data about their search activities. In this way we have gathered information about 96 search episodes. This paper presents the results of our analysis of this data organized around five key observations. For example, we observed that our participants performed quite broad searches which returned many irrelevant results and that they rarely looked closely at more than one of the results preferring to try different searches rather than explore a large result set more systematically. The key difficulty our participants faced was finding the information they needed despite conducting on average about ten searches each.

Our observations are discussed further in Section 4. The implications these observations have for search tools are described in Section 5.

2. Related Work

Various models of program comprehension have been proposed including the top-down model [18], the bottom-up model [11], and the integrated metamodel [21]. These models focus on describing a programmer's mental representation of a program and the cognitive processes and information structures used to form that mental representation. Other work has attempted to use these models or theories to inform tool design. For example, Storey et al. develop a hierarchy of cognitive issues to be considered during the design of a software exploration tool [19]. O'Brien et al. [10] performed a study to investigate how the system being modified affects the use of these comprehension strategies. Our work takes a different approach to influencing the design of tools, and we believe our results are complementary to work in the area of program comprehension. In particular, we aim to conduct studies to fill in details around the search activities of programmers (how they form searches, how they work with the search results returned, etc.) including the

challenges they face. We believe these details provide an important connection between program comprehension theories and programming tool research and design.

A number of studies have been conducted to explore the impact of tools on program comprehension. For example, Storey et al. carried out a user study focused on how program understanding tools enhance or change the way that programmers understand programs [20]. As another example, Singer et al. carried out studies of programmers' work practices, investigating how programmers approach comprehension tasks, including their use of tools [17]. Our research is along these same lines; however, we have focused specifically on searching activity in the context of today's development environments.

Five more recent studies have focused on the use of current development environments (as do our studies). Robillard et al. characterize how programmers who are successful at maintenance tasks typically navigate a code base [14]. DeLine et al. report on a formative observational study also focusing on navigation [3]. Our work differs from these in considering more broadly the process of searching and investigating search results, rather than focusing specifically on navigation. Ko et al. report on a study in which Java programmers used the Eclipse development environment to work on five maintenance tasks on a small program [8, 6]. Their intent was to gather design requirements for a maintenance-oriented development environment. Our studies differ in focusing on more realistic situations involving larger code bases and more involved change tasks. Our analysis differs in that we aim specifically to understand the search activity. LaToza et al. explored how experience effects work on change tasks and found, for example, that more experienced programmers tend not to explore as many irrelevant elements [9]. Our work builds on this by investigating how programmers determine which elements to explore.

Several researchers have proposed enhanced search tools. For example, Poshyvanyk et al. looked at a combination of latent semantic indexing and scenario based probabilistic ranking techniques as a new method for ranking search results [13, 12]. As another example, de Alwis et al. [2] looked at combining information from multiple sources (static, dynamic and evolution) to answer conceptual queries. In contrast, we are not currently proposing new search techniques; instead, we are reporting on a formative study that we hope may inspire future tool development.

3. Methodology

Our study was conducted in a laboratory setting and involved ten programmers (referred to as P1...P10) participating in ten 30 minute sessions. Eight of our ten participants were programmers with industrial programming ex-

perience. The remaining two (P1 and P3) were computer science graduate students. All participants had prior experience programming with the Java programming language and the Eclipse Java development environment¹, which we refer to as Eclipse throughout this paper.

During each study session, the participant was asked to perform a programming change task to a large software system called Subclipse², which is a popular open source plugin. Subclipse is comprised of approximately 70KLOC and all of our participants were initially newcomers to the system. For the purposes of our study, it was not important for the participants to complete the task during the session (and none of them did complete it); we were simply interested in putting them in a position where they would naturally perform various searches in Eclipse.

For each session we randomly assigned one of two different change tasks to get a broader range of situations. To ensure that the tasks were realistic, we selected them from the Subclipse project's issue tracking system (task 1 was based on issue 798 and task 2 was based on issue 801). These issues were addressed in revisions 3993 and 4012 of the Subclipse version control system, so we had our participants work with revision 3992 for task 1, and 4011 for task 2. Both of the tasks had a similar scope and required the participant to fix a fault in the code base. At the start of the session we explained to the participant the incorrect behaviour exhibited by the system as follows:

- *Task 1:* When multiple items are selected in the commit or revert dialog and the spacebar is pressed, the current behaviour is to toggle only the first of the selected items. The correct behaviour is to toggle the checked state for all selected items.
- *Task 2:* When a new file is added to version control, the current behaviour is to add a star overlay to the file icon. The correct behaviour is to add a plus overlay to the file icon.

Participants were restricted to only features provided by Eclipse (ex. use of the debugger, running of the program, navigation, search tools, etc.). They were also given paper and a pen. System documentation was available to our participants, but its presence was not highlighted by the researchers. To help judge the effectiveness of their searching activity, we investigated the experts solution to the tasks (that is, the solution found in the Subclipse's version management system) and identified several source code elements as *highly-relevant* to the tasks.

To facilitate data collection during the study sessions, we had each participant work as a pair [22] with one of the researchers (the second author of this paper). The researcher

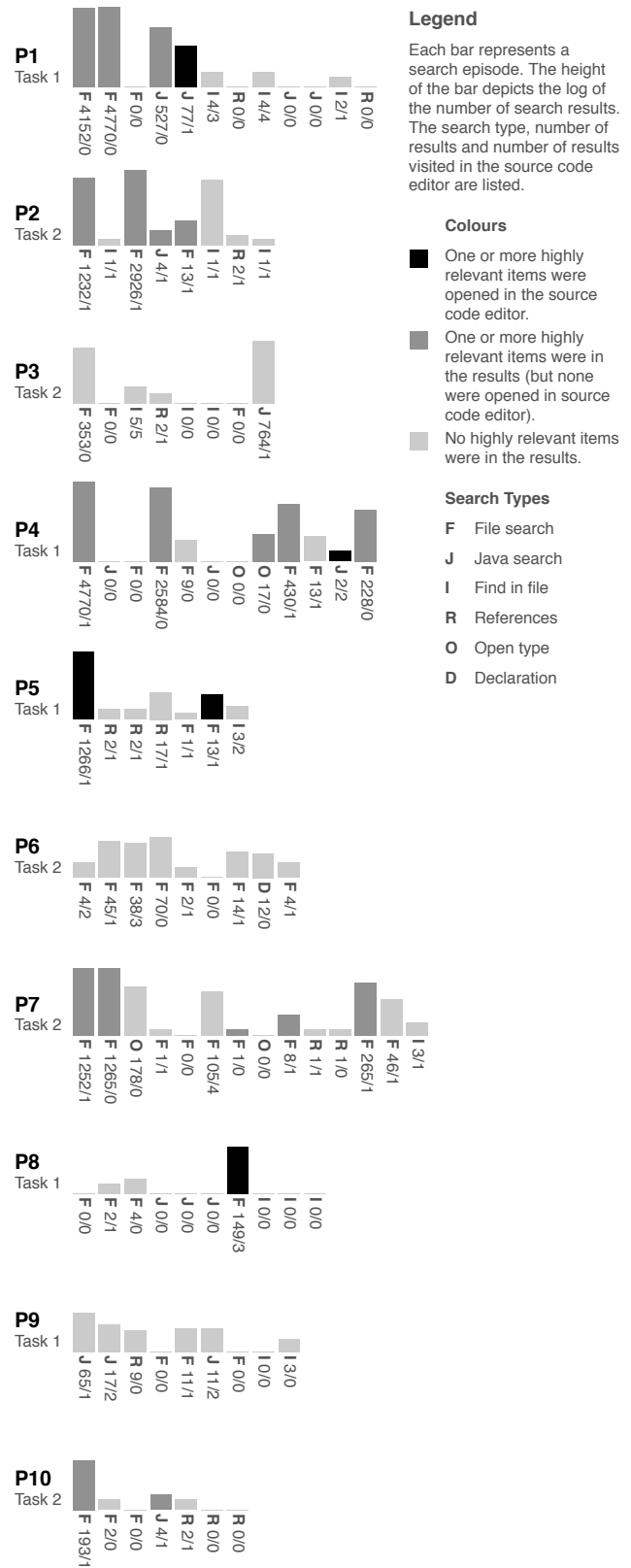
¹<http://www.eclipse.org>

²<http://subclipse.tigris.org>

was at the keyboard and the participant directed the work on the task by giving instructions to the researcher. We took this Dialog-Based Protocol [23] approach in the hope that it would provide insights into why various search activities were being carried out. Instructions given to the researcher were at a high level and included directions such as “Perform a File search with the query *icon*”. The discussion between the participant and the researcher was recorded and a screen capture video was made for each session. Following the session, another researcher (the first author of the paper), who was also present during each session, conducted a short ten minute interview with the participant. This was done in addition to their task time, to further explore issues around their searching activities. Two pilot studies were conducted to ensure that 30 minutes was sufficient time for participants to make progress in the task and to ensure that there would be enough searching activity for our analysis.

To carry out the change task, our participants used Eclipse and they were given a one page document describing the eight major kinds of searches supported by Eclipse to ensure that they were aware of the options they had. This document shows what each search type tries to accomplish but does not show how those search results are displayed. All eight of these search types are described below. Note however, that only six of these search types were used by our participants during the study (see Figure 2 for the search types actually used).

1. *Open Type*: Supports searching for classes or interfaces based on a partial name or pattern.
2. *File*: Supports searching for text within all of the files in the Workspace.
3. *Find in File*: Supports searching for a piece of text within a specified file.
4. *Java*: For finding declarations, references and occurrences of Java elements (packages, types, methods and fields).
5. *References*: Performs a search for all references to a specified code element or elements matching a keyword.
6. *Implementors*: Performs a search for all classes that implement a specified interface or interfaces matching a keyword.
7. *Declaration*: Performs a search for all declarations of a specified code element or elements matching a keyword.
8. *Occurrences in File*: Performs a search for all occurrences of a specified code element in the current file.



Each time a participant used one of these searches we call this a search episode. Our data set consists of 96 completed episodes, and in our analysis we have used the search episode as the basic unit of analysis. The first phase of our analysis involved segmenting our data into search episodes. We consider a search episode to be started as soon as a participant starts to describe a search that they would like to perform. We considered this episode ended when they last explore one of the results in the result set. If no results are explored, we considered the episode to be complete at the latter of when the search returned the last result into the result set, or the participant last comments on the results. Both quantitative data (such as the amount of time spent on each search episode) and qualitative data (such as comments made by our participants about what information they were seeking) have informed the findings discussed in the next section. In our analysis we do not consider the use of the debugger or other non-search tools available in Eclipse.

4. Findings

Our data, summarized in Figure 1, can indicate the effectiveness of our participants' searches as well as the effectiveness of their approach to working with the search results. In Figure 1, a medium grey or dark grey bar depicts a search that returned at least one highly-relevant element. For example, half of participant P2's searches returned highly-relevant elements, while none of participant P3's searches returned highly-relevant elements. The ideal search contains highly-relevant results and few other results, such as the fourth search conducted by participant P2. From this perspective the searches conducted by participants P3, P6, and P9 were particularly ineffective. A dark bar in Figure 1 indicates an episode in which the participant opened at least one highly-relevant element in the source code editor, suggesting a successful exploration of the results. Participants P1, P4, P5, and P8 were our only participants that managed this.

To describe the challenges our participants faced, in this section we discuss how our participants searched and worked with the search results. The discussion is organized around five key observations (see Table 1) that have several implications for the design of search tools as discussed further in Section 5. Each observation is supported by data collected from task 1 and task 2.

Observation 1 *Our participants approached a corrective maintenance task by first forming hypotheses about what the problem is based on their past experience. These initial hypotheses were often correct and guided much of their work on the task.*

Being newcomers to the code base they were working on, our participants began their task by making some

guesses about how the code was structured. They also needed to make some guesses about what the cause of the defect could be. At times a hypothesis was formed after some initial exploration of the code. This observation is consistent with previous work in the area of program comprehension which suggests that hypotheses are the result of comprehension and code cognition, but may take seconds or minutes to occur [21].

For example, participant P4, working on task 1, began with a broad search for 'commit' followed by several searches for words that he thought "might have been used in the naming of variables or [in] comments". After a few searches, he guessed that the problem was located within the UI packages and so focused his exploration on those packages. He described his approach to starting the task as creating a hypothesis about what is wrong in the code:

"The problem is you have some sort of undesirable behaviour right, and you know a few things about it. I tried to create a hypothesis that made sense. I tried to come up with some idea that could conceivably explain this odd behaviour."

This participant based his hypothesis on his past programming experience. Specifically he relied on his understanding of how code tends to be structured including the patterns he tends to use in his own coding. In the process he asked himself questions such as:

"If I were writing this system, where would I put the important code, how would I structure my data, how would I organize my classes?"

The highly-relevant elements for both tasks were located in the UI package, and most of our participants correctly guessed this location. Specifically, six participants working on the tasks mentioned the UI as being involved. Of the remaining four participants, two performed at least one explicit search within the UI packages, and one exclusively expanded the UI packages within the results. For example, since task 2 requires finding the place where icons are changed, participant P7 felt that it "makes sense to be looking in there for where those images would be chosen". Similarly, participant P5, working on task 1, noticed that the system was layered and guessed it was using the listener pattern [4]. He further guessed the problem would be located in "a UI class of some sort". This guess affected the searches they conducted and also their exploration of the search results.

Experience in the domain of the application was helpful to participants in forming their hypotheses. As all of them had some experience with version management systems, the concepts in the system and those specifically mentioned in

Table 1. Key Observations

- 1 Our participants approached a corrective maintenance task by first forming hypotheses about what the problem is based on their past experience. These initial hypotheses were often correct and guided much of their work on the task.
- 2 From their hypothesis, our participants formed search queries based on experience and expectations around naming conventions. Sometimes trying multiple searches based on synonyms or various ways to express similar ideas.
- 3 Our participants often had only a fuzzy notion of what to search for (or what to look for in a result set) especially early in a task. Often their searches were very general and returned many results.
- 4 Rather than systematically investigate search results, our participants generally skimmed through results looking for evidence of relevance (based on their hypotheses), at times focusing on a package that was consistent with their guiding hypothesis.
- 5 Our participants opened a small number of results in the source code editor and rarely return to their search results (i.e., not iterating between results and elements). Most often the opened element was only skimmed.

the task descriptions were familiar to our participants. However, participants P2 and P3's version management experience was limited to CVS³ (rather than Subversion). This limitation misled them in their work on their respective tasks, because of the differences between the Commit and Add to Version Control concepts. Participant P3 was particularly ineffective (see Figure 1) and after the session said:

“I was thinking CVS [...] everything is a commit even adding something is a commit. So that's probably why I was probably on the wrong track.”

Observation 2 *From their hypothesis, our participants formed search queries based on experience and expectations around naming conventions. Sometimes trying multiple searches based on synonyms or various ways to express similar ideas.*

In this section we discuss further how our participants used their hypotheses in formulating searches. Naturally, our participants based their searches on their expectations around the naming conventions used in the code base. For example, participant P4 described his approach as placing himself into the original programmers' situation and trying to figure out “what would I put in the comments, how would I call my variables, how would I call my class names and function names?”.

Of course there are different ways to express the same concept, and so our participants would try different searches—that is, different ways to express similar ideas—to learn about how terms are used in the system and of course to find task relevant elements. Participant P4 searched for “any number of terms [...] that is related to my hypothesis” and tried using synonyms. Half of our participants took a similar approach searching for ‘icon’ then

‘image’, ‘spacebar’ and then ‘space bar’, or ‘pressed’ followed by ‘keypressed’ to make the search more specific. File searches were used for a majority of these kinds of searches.

This process was not always straightforward even if a participant's hypothesis about what the problem might be was correct. Eclipse's search tools require the user to specify a specific term, and if the supplied term is not precisely correct no relevant results will be returned. For example, participant P10 (who never managed to open a highly-relevant element) felt that he had trouble making progress because the system's “naming conventions [...] didn't seem to be the same as in my mind”. When looking for the icons related to the task participant P10 performed a File search in the UI package for ‘modified’ proceeded by another file search with the query ‘update.gif’ (which was his guess for the name of the file). He never did find “the icons that are actually doing these things [related to the task]” because the naming of the icons were not what the participant had expected.

Our participants generally spent only a short amount of time per search episode before starting a new search. The longest amount of time spent in a single search episode was about five minutes, and the shortest amount of time for a particular search episode was five seconds. The mean amount of time spent in a single search episode was 69 seconds and the median was 39 seconds. Figure 5 shows the distribution of time spent on each episode. In some cases, a large result set seemed to deter the participants from spending much time going through the results—or in even looking at any of the results. When one of participant P4's searches returned 4770 matches, he said, “that doesn't seem to be especially helpful” and moved on to another search. Later he performed a search that returned 2584 matches, and said, “ok, so that didn't really work out” before abandoning the search.

³<http://www.nongnu.org/cvs>

The log of the number of search results returned grouped by search type. For an explanation of the colours, see the legend in Figure 1.



46 of the 96 episodes used the **file search** tool. Only the 36 that returned non-empty results are shown. Mean result size: 570.



16 of the 96 episodes used the **Java search** tool. Only the 9 that returned non-empty results are shown. Mean result size: 92.



16 of the 96 episodes used the **find in file** tool. Only the 10 that returned none empty results are shown. Mean result size: 2.



13 of the 96 episodes used the **references search** tool. Only the 9 that returned non-empty results are shown. Mean result size: 3.

Not shown are the 4 **open type** episodes and 1 **declaration search** episode.

Figure 2. Search episodes by search type

Observation 3 *Our participants often had only a fuzzy notion of what to search for (or what to look for in a result set) especially early in a task. Often their searches were very general and returned many results.*

Figure 2 shows the distribution of the number of results per episode for the top four types of searches performed. This figure also shows the average number of results for each search. Eclipse’s File search was by far the most popular amongst our participants, accounting for approximately half of all search episodes. File search is the most inclusive search (as compared to a Java search) and has fewer options for restricting the scope of the search. Therefore, it is unsurprising that on average it returns the most results (570) and also had the largest maximum result size in our study (4770).

Many of the searches performed by our participants were broad in scope and involved terms that were common to the code base. For example, our participants carried out a File search for ‘commit’ eight times. This search returns more than a thousand results in the Subclipse code base, as it is based on a common term in the application domain. Thirty-five of the 96 searches returned ten or more results and 22 returned more than 50 results. On the other hand, there were also many cases (29 out of 96 searches) in which a search returned no results.

Eight of our ten participants began their work on the assigned task using a File search as they had little other infor-

mation to start with; “when I don’t know what I’m looking for, then usually I start with a text search” (P2). However participant P2 also felt that generally file searches “results in a lot of garbage”. Similarly, P1 began his investigation by conducting a File search for ‘*commit*’ which related to the fact that Task 1 was an issue within the commit dialog. The search returned 4152 results. He explained that he had nothing to base his search on other than a “keyword” from the task description.

Even later in the study sessions, our participants had a difficult time articulating searches that would retrieve highly-relevant search results. Our participants generally formed hypotheses about what was wrong, but turning that into a precise search was not straightforward because the concepts that they were looking for were commonly vague. For example, participant P4 was looking for “anything to do with pressing spacebar, selecting and deselecting checkboxes” but wasn’t able to turn this into a meaningful search using the available tools. He “mostly [searched] the entire workspace” to make sure that he didn’t miss “something important”, preferring to get too many results rather than too few.

Our participants also expressed a lack of confidence in the task relevance of those search results and as a result they did not always carefully explore the results. For example, P2 performed several searches just to “see what it returns” and only looked briefly at the results because she was not “sure the term [...] is useful for the task”. She further felt that once she had more “concrete information” she would be “more patient with the search results”. The way that our participants worked with search results is discussed further below.

Observation 4 *Rather than systematically investigate search results, our participants generally skimmed through results looking for evidence of relevance (based on their hypotheses), at times focusing on a package that was consistent with their guiding hypothesis.*

In Eclipse, results from File searches, Java searches, References searches and Declarations searches are presented in a tree view, organized by package, class and then matching element (or by the directory structure), all sorted alphabetically. Initially the package and class with the first matching element are expanded and the element is highlighted, as illustrated in Figure 3. The vast majority of the searches performed by our participants (76 out of 96) were searches that returned tree based results. Results from File searches are presented along with one line of source code that shows the matching search term in context (see of Figure 4). We refer to this as the context snippet.

As described previously, many of the searches performed by our participants were quite broad and returned a large number of results. Despite this, the time spent per search

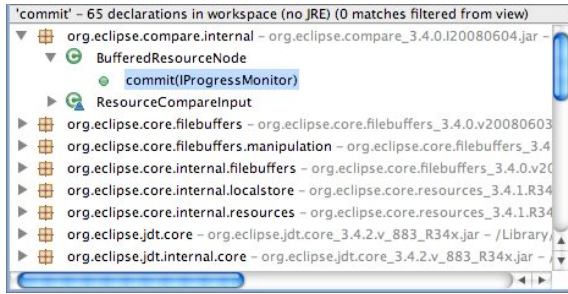


Figure 3. The default presentation of search results by the tree based search results.

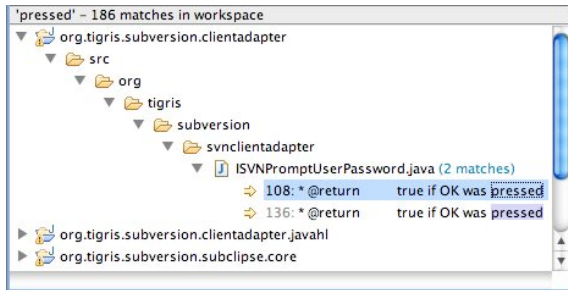


Figure 4. Presentation of results with inline context included.

episode was small, at only a mean of 68 seconds. The reason for this is that, rather than systematically exploring each item returned from a search result, our participants tended to quickly skim the results, often starting by briefly considering the names of packages and classes. As participant P6 put it in one instance, “there were a lot of results” so he “just skimmed through them”.

This skimming of the package names (etc.) was guided by a participant’s hypothesis about the fault they had been asked to fix and aimed to see if any of the package names were “interesting”. For example, participant P3 performed a Java search for ‘getImage’, which returned 764 results in total. He quickly scrolled through the results until he found a package that he thought might have something relevant—one of the user interface (UI) packages. He then restricted his investigation to the results in that package, feeling that the other packages were “completely useless to me so I ignored those”.

Focusing on the various UI packages in search results was common for our participants because they had hypothesized that the fault would be found in the UI layer of the application. As participant P4 said, the problem “seems like a user interface issue” and he “would just ignore all of the other packages entirely and jump directly to the UI”. During 40 search episodes, participants opened files from tree

based results. Of these, 31 times our participants focused their exploration on the UI packages. In other cases, our participants simply explored the first result, which is highlighted simply by virtue of the alphabetical sorting of the results. Participants explored results outside the UI packages on only nine occasions. In seven of those, the participants first exploration was the result that was highlighted. For example, after performing a search, participant P9 opted to “just open the first one there”, which was unrelated to the task. He then returned to the search results and expanded one of the UI packages.

Once a package of interest was selected by the participants, they tended to skim the files and classes again looking for something “interesting”, or in other words, names that seemed relevant to the task at hand. Some participants expanded the contents of a selected package so that they could see the context snippet, because “the little context it provides when you expand everything [...] gives you a pretty good idea of what you’re actually looking at” (P4). This context was used effectively by participants P4 and P6; however, most participants did not expand the search results sufficiently to show the context snippet. Participant P3, for example, seemed unaware that this information was available and complained that “you don’t get any other information other than the name itself”.

In many cases our participants decided very quickly that nothing was relevant and tried a different search. For example, participant P4, who worked on task 1, performed a search for “pressed” and skimmed the results looking for anything that seemed relevant. In the process he saw that some of the results were about an “ok button” or “cancel button” being pressed which he judged as “fairly obviously not relevant”. Interestingly, our participants generally needed strong indications of relevance before they would spend time exploring a result further. For example, participant P2 said that because many searches return a large number of results, the items in the result sets “have to look quite right at first glance for me to look into it”. She also said that for something to be judged as “quite right” it needs to match (or be synonymous) to the “terms going through my mind”.

At times this reliance on pre-conceived terms was problematic, such as when participant P2 performed a search that returned only four matches, one of which was arguably the element most directly related to the assigned change task. She briefly looked at the first result in the source editor, returned to the results, looked over the names of the remaining classes returned by the search before stating that she was “not sure this is useful, maybe you should do a text search for the same string”. When performing a search, specific terms need to be supplied; however, when skimming through search results it is possible to look for anything “that would make sense” (P8).

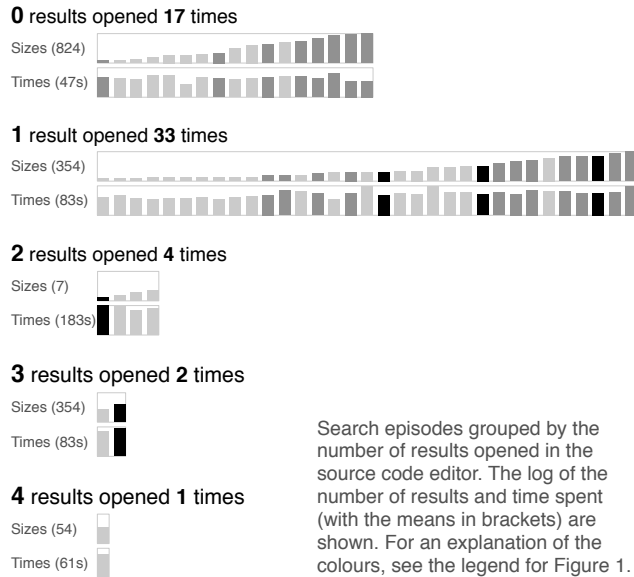


Figure 5. Number of results and time spent per episode by number of files opened

Observation 5 *Our participants opened a small number of results in the source code editor and rarely returned to their search results (i.e., not iterating between results and elements). Most often the opened element was only skimmed.*

Over the course of our study, participants performed 57 searches that returned non-empty result sets. Given a non-empty result set, our participants would decide which elements (if any) to open in the source code editor. None of our participants ever opened more than four results, and the most common behaviour (occurring in 33 of the 57 cases) was for the participant to explore just one of the results. Interestingly, the second most common behaviour was for our participants to explore none of the results. The exploration behaviour we observed is summarized in Figure 5.

In some of the cases when none of the search results were explored, it meant that our participants briefly skimmed the results (as discussed above) and did not notice any that seemed particularly relevant. In other situations the sheer size of the result set discouraged extensive investigation. For example, participant P2’s first search was a File search for the term ‘decorator’ which returned 1,232 matches. She briefly explored one of the results and then decided that it was not “useful” and abandoned the search. Her reasoning was that the term she searched for was “pretty overloaded” and that she “did not want to spend a lot of time going through each search result.”

Similarly participant P4, when faced with a large result set, decided not to investigate it in detail saying that “it’s not

necessarily feasible to kind of go through them one by one.” He also felt that the large result set suggested that he was “on the wrong track” and possibly “searching too broadly.” More generally, our participants tended to open fewer of the results when the result sets were large (see Figure 5), deciding that their time would be better spent performing a different search. As a result, iterating between the search results and the source code editor to find relevant elements was rare.

When a source element was selected for further investigation, our participants tended to skim through the source code quickly to decide how relevant the element was. This skimming was guided by their hypothesis about the problem that needed fixing. For example, participant P2 skimmed through a file and stated, “this doesn’t seem quite right” because it was an abstract class and she expected that the problem would be in the “concrete implementation”.

Similarly, participant P4 opened a particular file in the source code editor and spent twenty seconds scrolling through the file, performing what he termed a “visual analysis of the code.” He found that none of the method names “seemed to really have anything to do with” the task, so he abandoned the search all together. During the interview portion of the session, participant P4 explained his approach to skimming files:

“When you open up a file, again if you’re thinking a variable is being changed at a wrong time, you can just look over really quickly for assignment statements or anything like that. If you expect its a bad loop, you can just look for the presence of loops or something like that. I mean its not always easy but it really does go back to kind of, the whole notion of creating hypothesis, and you think, well, if my hypothesis is valid, how could I recognize it, what would the code look like under such a situation.”

5. Implications for Design

As discussed in Section 4, our participants’ behaviour can be described as searching and skimming. We believe that if tool support for this behaviour was improved, then several of the challenges our participants faced in performing the assigned change task could be mitigated. Several examples follow.

Support for Skimming The searches performed by our participants were generally broad searches producing many irrelevant results, so they continued their finding activity by skimming the result set looking for relevant entities. The advantage is that skimming (unlike searching) does not require articulating a particular term of interest; the challenge

is that the results, as presented by Eclipse, contain little information upon which programmers can base their judgement of relevance. However, our participants made use of two kinds of information that Eclipse does make available: structural information (primarily package structure) and names. Building on this observation, it is possible that more contextual information would be valuable. Programmers can obviously get significantly more information about search results by opening the elements in the source code editor; however, we found that when they do this, they tend not to return to their result set to consider other results. Providing additional contextual information, possibly including block level information, right in the search result view may allow programmers to more quickly determine what is relevant.

Ranking and Grouping of Results Our participants explored only a small number of matches (often zero or one) and did not carefully consider all of the matches before selecting which element(s) to explore. Based on this we argue that the sorting of results should be given more attention, possibly based on confidence values such as those used by the Hipikat system [1]. We have found that programmers are put off by large result sets; but, if the results were ranked in a meaningful way, it is possible that programmers would be able to make use of such results. How best to rank results is an open question; however, we believe that a successful ranking would likely need to be context aware, possibly based on contextual information maintained by tools such as Mylyn [5]. During our study we found that many participants used the packages as a way of segmenting their results, so they could focus only on a smaller subset they believed to be relevant. Other groupings of results (e.g., spheres as defined in [2]) are possible and may be similarly useful.

Support for Exploring Result Sets Beyond a quick skim and looking at a small number of results, our participants made only limited use of Eclipse’s search results view. Each search episode took on average 68 seconds and involved exploring a very small number of results. Rather than extensively explore the results, the participants would perform a different search altogether, though at times they would later repeat their search. It is possible that enhancements to the interaction supported by the search results view would allow programmers to make better use of their results and possibly view them as a basis for ongoing exploration. Possible enhancements include tracking what has been explored and what has not, support for removing or highlighting elements based on their judged relevance, and support for searching within results.

6. Limitations and Future Work

Our participants were newcomers to the system we asked them to change. In our experience, this is a common scenario for programmers in industry and is convenient for conducting controlled studies. However, it may also mean that our results provide limited insights into the search behaviour of programmers performing change tasks to systems with which they are familiar.

Our study set-up followed the Dialog-Based Protocol with one of the researchers being the “driver” during the session [23]. This was effective as it encouraged the participant to vocalize their intentions. While this approach was helpful, it is an open question what effect this technique had on the search behaviour. The researcher was careful not to make suggestions to the participant, but it is possible that he may have provided incidental assistance such as help spanning the gulf of execution.

At the start of each study session, our participants were given a list of searches supported by Eclipse. This ensured that our participants knew the options available. On the other hand, this likely encouraged our participants to consider searches in that list. This limits our ability to determine what other types of searches would be valuable to programmers. It is also possible that the search behaviour may have been different given different tools.

To date the search episode has been our unit of analysis. Going forward with this research we plan to analyze our data at the session level and to categorize sessions as more or less successful. In this way, we hope to identify search patterns and correlate those to how successful programmers search. We hope that this analysis will improve programmer practices.

7. Summary

As part of a research program to improve the tool support for finding information in source code, we have conducted a study to explore how programmers decide what to search for, and how they decide which results are relevant to their task. We observed ten programmers as they performed a corrective maintenance task to a software system and collected qualitative and quantitative data on 96 search episodes. This paper presents the results of our analysis of this data organized around five key observations. For example, we observed that our participants performed broad searches that returned many irrelevant results. Thus, they rarely looked closely at more than one of the results and preferred to try different searches rather than explore the results more carefully. We have also discussed some of the implications our results have for the design of search tools.

Acknowledgements

We thank our participants for their participation. This research is also funded in part by NSERC.

References

- [1] D. Cubranic and G. Murphy. Hipikat: recommending pertinent software development artifacts. In *Proceedings of the International Conference on Software Engineering*, pages 408–418, 2003.
- [2] B. de Alwis and G. C. Murphy. Answering conceptual queries with ferret. In *Proceedings of the international conference on Software engineering*, pages 21–30, 2008.
- [3] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson. Towards understanding programs through wear-based filtering. In *Proceedings of ACM 2005 Symposium on Software Visualization*, pages 183–192, 2005.
- [4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.
- [5] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for IDEs. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 159–168, 2005.
- [6] A. J. Ko, H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *Proceedings of the International Conference on Software Engineering*, pages 126–135, 2005.
- [7] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proceedings of the international conference on Software Engineering*, pages 344–353, 2007.
- [8] 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 software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(10):971–987, 2006.
- [9] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers. Program comprehension as fact finding. In *Proceedings of Foundations of Software Engineering*, pages 361–370, 2007.
- [10] M. P. O’Brien and J. Buckley. Inference-based and expectation-based processing in program comprehension. In *Proceedings of the International Workshop on Program Comprehension*, pages 71–78, 2001.
- [11] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.
- [12] D. Poshyvanyk, A. Marcus, V. Rajlich, Y.-G. Gueheneuc, and G. Antoniol. Combining probabilistic ranking and latent semantic indexing for feature identification. In *Proceedings of the International Conference on Program Comprehension*, pages 137–148, 2006.
- [13] D. Poshyvanyk, M. Petrenko, A. Marcus, X. Xie, and D. Liu. Source code exploration with google. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 334–338, 2006.
- [14] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, 2004.
- [15] J. Sillito, G. C. Murphy, and K. D. Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the SIGSOFT Foundations of Software Engineering Conference*, pages 23–34, 2006.
- [16] J. Sillito, G. C. Murphy, and K. D. Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, 2008.
- [17] J. Singer and T. Lethbridge. Studying work practices to assist tool design in software engineering. In *Proceedings of the International Workshop on Program Comprehension*, pages 173–179, 1998.
- [18] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE10(5):595–609, 1984.
- [19] M.-A. D. Storey, F. D. Fracchia, and H. A. Muller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Software Systems, special issue on Program Comprehension*, 44(3):171–185, 1999.
- [20] M.-A. D. Storey, K. Wong, and H. A. Muller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2–3):183–207, 2000.
- [21] A. Von Mayrhauser and A. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, Aug 1995.
- [22] L. Williams, R. R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the case for pair-programming. *IEEE Software*, 17(4):19–25, 2000.
- [23] S. Xu and V. Rajlich. Dialog-based protocol: an empirical research method for cognitive activities in software engineering. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 10–19, 2005.