Archetypal Source Code Searches: A Survey of Software Developers and Maintainers

Susan Elliott Sim
Dept. of Computer Science
University of Toronto
Toronto, Ontario,
Canada M5S 3G4
simsuz@cs.utoronto.ca

Charles L.A. Clarke
Dept. of Elec. and Comp. Engineering
University of Toronto
Toronto, Ontario
Canada M5S 3G4
clclarke@eecg.utoronto.ca

Richard C. Holt
Dept. of Computer Science
University of Waterloo
Waterloo, Ontario,
Canada N2L 3G1
holt@plg.uwaterloo.ca

Abstract

In this study, we conducted a survey to generate archetypes of source code searching by programmers across maintenance tasks. Using a questionnaire on a web page, we obtained 69 responses from readers of 7 newsgroups. Respondents were asked about their source code searching habits: what tools they used, why they searched, and what they searched for. The four most common search targets were function definitions, all uses of a function, variable definitions, and all uses of a variable. The most common search motivations were defect repair, code reuse, program understanding, feature addition, and impact analysis. Eleven archetypes were generated from the anecdotes and results. The implications and practical applications of these findings and method are discussed.

1. Introduction

Searching source code is an essential part of many software maintenance tasks. The structure of source code is not conducive to being read in a linear fashion. Furthermore, it is often infeasible to read the entire source code of many large pieces of software. Consequently, programmers must read selectively, which means identifying the parts of the source relevant to the task at hand. When repairing a defect, the location of the error must be found. When reusing a function, its declaration must be found, so that it can be invoked with the right parameters. When learning about a program the flow of control needs to be followed, which means jumping from one function definition to another.

In the past, research in program comprehension has focussed on a specific development or maintenance task. These studies looked at the construction of mental models in program understanding [3, 10, 14], program maintenance [9], and defect repair strategies and behaviour [5, 8]. Searching is an integral part of models of these activities. Over several studies and different measures, Singer et al. found that searching was the most common activity for software engineers [12]. Aside from this work, there has been little research solely on searching. In contrast, one field of computing that has looked at searching behaviour extensively is information retrieval. Researchers in this area have developed detailed theories and models of search strategies [1, 2].

With the accumulation of this body of research, it has become apparent that a study specifically on searching would yield results applicable in many areas of program comprehension. Singer et al. have used their results to inform their design and implementation of a source code searching tool [12]. This study followed a similar line of inquiry, but used a different approach which is reflected in the method and level of analysis. In this study, we conducted a questionnaire-based survey of software developers and maintainers contacted using availability sampling over the Internet.

The form of this study was based on one by Eisenstadt in which the author posted to various online fora requesting anecdotes of “particularly thorny bugs in LARGE pieces of software,” so he could characterize why some bugs were more difficult to repair than others [5]. Respondents from a number of electronic conferences and USENET newsgroups provided anecdotes of defects and the author was able to find systematic trends in their responses. In our study, we asked readers of various newsgroups to tell us about their source code searching activities.

An implication of our approach was a shift in emphasis from a particular maintenance task to a constituent activity. Previous work tended to study a small number of subjects intensively while doing a task. This study surveyed a large number of respondents about their searching behaviour, without delving into the cognitive aspects of the activity. We investigated source code searching specifically, as it occurred across maintenance activities.

1.1. Objectives

There were two objectives in this study. The primary objective was to understand how and why programmers searched source code. We asked about the tools they used and situations in which they searched source code. Qualitative and quantitative data from participants were used to construct a model of searching behaviours. Anecdotes of the situations were used to develop a series of archetypes of source code searching.

An archetype is a concept from literary theory. It serves to unify recurring images across literary works with a similar structure [7]. In the context of source code searching, an archetype is a theory to unify and integrate typical or recurring searches. As with literature, a set of them will be necessary to characterize the range of searching anecdotes.

The secondary objective was to determine the efficacy of using a web-based questionnaire to survey programmers. Surveying is a method often used in the social sciences to collect data in a structured or systematic manner [4]. The methods in this study were similar to those used in Eisenstadt [7].

The method is further described in Section 2, and the results are presented in Section 3. In Section 4, archetypal and uncommon search situations are presented. The paper concludes with summary of the results and an exploration of future work in Section 5.

2. Method

Fields such as sociology, political science, and market research use surveys to study a particular phenomenon. Normally, a survey is used to collect primarily quantitative data from a large number of respondents. Data gathering techniques include face-to-face interviews, telephone interviews, self-administered questionnaires, or archival research [6]. Regardless of how data is gathered, there are five steps in performing a survey, each corresponding to subsections 2.1 to 2.5.

2.1. Formulate Research Questions

In this study, we wanted to understand how and why programmers searched source code. The three questions that we wanted to answer in this study were:

- What tools do programmers use to search code?
- What do they look for when searching source?
- Which tasks require them to perform a search?

The tools currently used for searching provide role models for future tool development, and their shortcomings suggest areas for improvement. The targets and motivations for searches indicate some of the functionality required in such a tool Answers to the last two questions were given in anecdotes, so analysis of this data resulted in a set of archetypes to further inform tool design.

2.2. Create a Data Gathering Instrument

A questionnaire was selected to be the data gathering instrument because we wanted to collect information from a large number of respondents, many of whom we would not be able to contact personally.

The questionnaire consisted of two World Wide Web pages. The first was an introductory page with an explanation of the purpose of the survey and the rights of the participants. A link at the bottom of the page led to the actual survey. This two page format was used to encourage respondents to read this preamble before beginning the survey. The introduction had two parts, each fulfilling a distinct aim: a purpose statement motivated participants to give thoughtful responses to all the questions; and the statement of participant rights informed respondents of their rights according to standard ethics procedures [4, 6].

The questions and their wordings were tested in a pilot study of six respondents. These respondents were contacted by personal email and they were later debriefed, again by email. Our experiences from the pilot study are reflected in the final text of the survey. Data from the pilot study was not included in the analysis of the main survey.

The text of the survey is found in Appendix A. In the final version, there were six multiple choice and two open-ended questions. The remaining materials are available at: http://www.turing.utoronto.ca/~simsuz/survey .

2.3. Define the Population and Sampling Method

The population of interest for the survey was loosely defined, so a random sampling method could not be used. The population was any programmer who had worked with relatively large pieces of existing source code. Due to a lack of demographic information it was difficult to operationalize this definition. It was not possible to enumerate the population and randomly select participants. Consequently, availability sampling, also known as convenience sampling, was chosen. Normally, this method is used only in exploratory studies.

Availability sampling operates by publicly soliciting volunteers to participate in the study. The main drawback of this technique is that the sample does not represent the population of interest, in this case developers of large systems. However, had another sampling method been used, it still would be difficult to generalize from the results. Since the study was exploratory in nature and its goal was to build a model of source code searching, availability sampling was adequate for the task.

2.4. Administer the Survey

The pages were published on a web site and participants were solicited from eight USENET newsgroups. Messages were posted to:

comp.lang.c.moderated, comp.lang.c++.moderated, comp.lang.java.programmer, comp.lang.smalltalk, comp.lang.lisp, comp.lang.fortran, comp.software-eng and comp.lang.cobol.

The same message was reposted one week later with an additional paragraph at the beginning. There were no participants from comp.lang.c++.moderated because requests for participation were filtered out by the moderator. All of the data were collected within a four week period.

2.5. Analyze the Data

Coding is the process of assigning values to variables to represent each respondent. In the analysis of the six multiple choice questions, the variables were scalar, such as counts and ratings. For the two free-form responses, the variables were qualitative, meaning their "values" were text descriptions or lists. These variables were analyzed by grouping similar responses together. The anecdotes were coded using qualitative data analysis techniques in several iterations [11]. Coding of situations is described in greater detail in Section 3.3.1. During this process, we used grounded analysis; categorization of search situations was driven by the data, rather than a theory of how a task is performed [13].

3. Results

Sixty-nine respondents provided description of 111 search scenarios and 207 suggestions for features. Overall, the quality of the results was good; only a small number of respondents did not answer every question on the questionnaire. Most of the responses were in point form, but their thoroughness often compensated for the lack of formality, while others were long, spanning more than a page. Some responses were humorous, for instance, "'Show me the location of the next error I should fix' :-)."

The origin of the participants are discussed in Section 3.1 and the tools they used are in Section 3.2. The anecdotes that they provided about their source code searches are described in Section 3.3. Where results are presented the data given in brackets are counts, except where noted.

3.1. Participants

The sixty-nine participants who submitted questionnaires came from a variety of newsgroups and email domains. Their origins by newsgroup are in Table 1.

Table 1: Origin of Participants By Newsgroup

Newsgroup	Number of Respondents
comp.lang.c.moderated	28
comp.lang.lisp	12
comp.software-eng	7
comp.lang.fortran	7
comp.lang.cobol	5
comp.lang.java.programmer	4
comp.lang.smalltalk	3
unknown	3
Total	**69**

Forty-five respondents were willing to participate in future studies and consequently gave their email addresses. Their address domains indicated more than two-thirds of them were from commercial and government domains. The distribution of participants by domain name is in Table 2.

Table 2: Origin of Participants by Email Domain

Domain	Number
com, gov, co.uk	26
net, org	5
edu, ac.uk	6
other	8
Total	**45**

3.2. Tools Used

There was a multiple choice question on the tools that respondents used to search source code. The available choices are shown in Table 3. In addition, a fill-in box was provided for tools that fell into the "other" category. Participants generally relied on standard tools. The utility `grep` performs regular expression matching over files and the category included its variants. The `find` tool, in basic form, searches for file names. All but one respondent (68) used either an editor or IDE (integrated development environment) to search source code, yet a large number of them used other tools as well.

Table 3: Tools Used

Tools Used	Number
editor	57
`grep`	47
`find` or "File Find"	38
other	38
IDE	26

In the fill-in box for the "other" category, a total of nineteen different tools were mentioned. The tools mentioned are in Table 4. Some participants entered more than one tool. If a tool was mentioned in the anecdotes that was not on the list of "other" tools, then it was also added.

Table 4: "Other" tools used

Tool	Number
tagging utilities	11
scripts	7
proprietary source browsers	6
language environments	5
`xref`	4
miscellaneous	10

Included in the category of "tagging utilities", were `etags`, `ctags`, and `ftags`. "Scripts" denotes any shell scripts, Perl or awk programs, and batch files. "Proprietary source browsers" included tools that were sold for the purpose of source browsing, such as Cygnus Source Navigator, SoftBench, and tools that were bundled with third party libraries. Smalltalk and Lisp programming environments were included in "language environments." These tools were included in this category rather than IDE because they incorporate a number of elements that are tightly integrated with the language and run-time environment. The UNIX tool, `xref`, builds a cross-referencing index of functions and variables. The last category included Norton Text Search, `javadoc`, "the compiler", and "my brain".

3.3. Situations

We received 111 anecdotes of search situations. All but four(65) respondents provided an anecdote. They ranged in length from a single line to more than a page. Figure 1 is a typical anecdote of a situation that required source code to be searched. In this section, the results of analyzing the anecdotes are presented. First, the search targets and the motivations for searching are discussed. In Section 4, the relationships between these two dimensions are examined to formulate searching archetypes.

```
I needed to understand old spaghetti
code which used global variables for
everything. Say there was a variable
'foo' which stored a critical value.
I'd grep for reads and writes to this
variable, to see which functions were
involved in creating and using this
value. I'd also search for it(in
emacs) in a cross-reference listing
to make sure I didn't miss some
place.
```

Figure 1: Example of Scenario Anecdote

3.3.1. Coding. Anecdotes were categorized along two orthogonal dimensions: the specific search target and the motivation for the search. Search targets tended to be quite easy to categorize, whereas motivations required stricter rules. Some responses had multiple search targets and motivations. In Figure 1, the search targets were coded as "function definition" and "all uses of a variable," and the motivation was coded as "program understanding".

The program understanding and maintenance categories were used as little as possible because it could be argued that all searches were performed for that purpose. In the example, a program understanding motivation was selected because the respondent gave no other explanation for why she was performing the search. The maintenance category was also used in a similar manner. The most specific motivation that was appropriate for the search was selected, based on the statements by the participant.

3.3.2. Search Targets. Within the 111 scenarios, 154 search targets and 94 motivations were identified. The four most common search targets were function definitions (26), all uses of a function (23), all uses of a variable (23), and variable definitions (19). Definitions are the portion of the source code that implements a function body or determines the type of a variable. Searches on functions, variables, and classes are summarized in Table 5. Further analysis of the searches on variables indicated that respondents were more interested in locations where a variable was written or assigned to (6) as opposed to simply read or referenced (1). Clearly, a piece of code that changes a variable has more potential problems than one that only reads it .

Table 5: Summary of Common Searches: Numbers shown on table are counts of occurrences. Top four values are in bold. There were 154 total search targets in the data.

	Function	Variable	Class	row total
declaration	10	-	-	10
definition	**26**	**19**	5	50
use	11	9	1	21
all uses	**23**	**23**	5	51
column total	70	51	11	132

Other common targets of searches were strings, either those output by the program or those in comments (10); and files where code was located (5). All searches for output strings coincided with a defect repair.

3.3.3. Motivations for Searching. The motivations for source code searching could be grouped into eleven categories. The categories and names are straightforward, with the exception of "clean-up", and "naming conflicts". These two categories will be discussed in greater detail and examples for each are provided.

Clean-up occurs before a program is frozen for release. A programmer may hard-code some strings during development, or leave notes to herself in the code. These items are removed before the code is shipped. A naming conflict occurs if a new function, variable, or class uses an existing identifier. A developer searches the code to ensure a proposed identifier is safe.

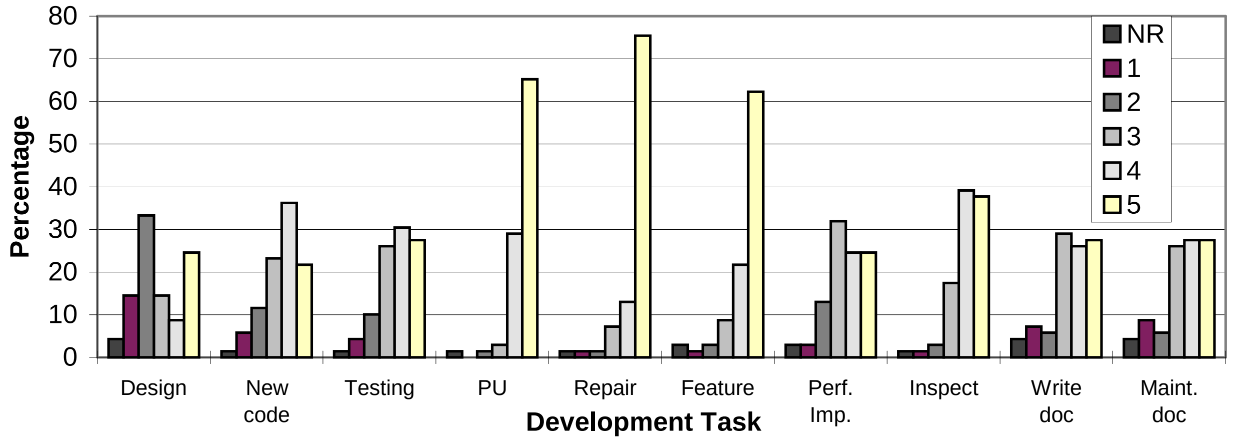

Figure 2: Usefulness of Searching Source Code by Task: This histogram shows the distribution of ratings for each development task. A bar shows the percentage of respondents giving that rating for a task. The tasks are low-level design, writing new code, testing, program understanding, repairing a defect, adding a new feature, improving performance, code inspection, writing documentation and maintaining documentation.

Table 6: Summary of Motivations for Searching

Motivation	Number
defect repair	19
code reuse	14
program understanding	13
impact analysis	12
maintenance	7
feature addition	7
clean-up	5
naming conflicts	4
porting	3
dead code elimination	3
other	7
Total	94

The four most common motives for searching source code were defect repair (19), code reuse (14), program understanding (13), and impact analysis (12). The results of this analysis should be compared with those from question three of the survey. It asked, “How useful is it to search source code when…” along with a list of ten activities from the software development cycle, and asked respondents to give a rating on a scale of one (low) to five (high). It was found that the tasks in which searching was most useful (median rating 5) were repairing bugs or defects, understanding old code, and adding a new feature to old software. The distribution of the ratings is presented in Figure 2. Further analysis yields some relationships between the motives for searching and the search targets.

4.Searching Archetypes

Archetypes were generated by examining the search targets and motivations presented in the previous section for patterns. Common or frequently-occurring relationships between targets and motivations were identified as a pattern. Eleven archetypes are presented in this section, beginning with the strongest ones. Also presented in this section are uncommon searches because they complement the archetypes by capturing the additional variability.

4.1.Common Searches

The pattern that emerged in the impact analysis category is the most definite.

1. During impact analysis, developers often looked for all uses of a variable or function.

Of the twelve searches with this motivation, nine were for all uses of a function or variable. Impact analysis is usually done to evaluate a change to the software. The developer wants to make sure that she has not broken anything inadvertently, therefore checks all uses of the modified component. This relationship is credible not only because the underlying explanation is plausible, but also because the numbers in this category are consequential.

In the program understanding category there were two main patterns of searching.

2. Searches motivated by program understanding sometimes sought function and variable definitions.

3. At other times, the search targets were a use of a function, variable or object.

Of the thirteen searches performed for this purpose, five were looking for definitions of functions or variables, and five were looking for function or variable or object use. In the case of definitions, the maintainer was trying to determine the effect of a particular function call or the data type of a variable. In the case of the latter, she understood the object, variable, or function, but wanted to know how it fit with the rest of the program.

The code reuse category revealed two patterns of searches.

4. To reuse code, a programmer searched for function signatures to call it correctly.

5. Alternatively, a programmer searched for functionality that was known to exist, but the name may not have been known.

Of the fourteen searches undertaken for the purpose of reusing code, seven were for function definitions and three for function declarations. When reusing code, one of two scenarios may occur: the developer knew the name of the function but needed to check the parameters in the declaration or definition; or the developer knew that code to perform a certain procedure existed, but was unsure of its name, so she performed a search.

In the bug repair category, there were a large number of examples (19) with a variety of search targets.

6. Maintainers tackled bugs by identifying the function that was misbehaving.

7. Another approach was to track usage of a variable.

8. An output string served as the starting point for a bug-hunt.

The three most common targets were function definitions (4), all uses of a variable (3), and output strings (3). The first pattern corresponds to a situation where a programmer knew that something was going wrong and was looking for the function responsible. Consequently, she looked at a lot of function implementations or definitions. The second archetype corresponds to a scenario where a maintainer knew a variable was set incorrectly during execution. In such a case, she looked at all uses of that variable to find the error. In the case of the third pattern, the programmer has received a bug report containing an error message. The search for the faulty code began by tracing how the message came to be printed. This pattern was particularly strong because all instances of searches for output strings were motivated by bug repairs.

In the porting, feature addition, and dead code elimination categories, relationships were found, but due to the small number of anecdotes it is difficult to know how significant they are.

9. To eliminate dead code, a maintainer needed to find all uses of the entity being removed.

In all of the dead code elimination searches(3), the targets were all uses of either a function (1) or a variable (2). In order to eliminate a variable or function, the maintainer has to make sure that it is either not used at all or used only in functions that will never be called. Therefore, she needs to be able to account for every use of that function or variable. This relationship is more credible than the others that have a small number of examples because its underlying explanation was present in the anecdotes and is highly plausible.

10. When porting code, developers often examined variables.

In all of the porting examples (3), the respondent was looking for information about variables. In two cases, it was all uses of a variable, and in the third it was the variable definition.

11. When adding features, developers sometimes examine functions.

In four of seven feature addition searches, the respondents were looking for information about functions. There were no clear patterns found among the searches in the clean-up, naming conflicts, and maintenance categories.

4.2. Uncommon Searches

In this section, we present some of the unique situations described in the survey. These anecdotes are noteworthy because they illustrate issues that software maintainers deal with, but are not captured by the archetypes. We look at searches performed for preventative maintenance, code reuse, and testing.

Although preventative maintenance is generally agreed to be a good idea, many software shops don't have time to do it. In the study, we received two anecdotes that described searches that were performed for that purpose on a small scale. Respondent 17 recalls an occasion when she discovered a variable had been used unsafely, she went through the source to verify that other uses of a variable were correct.

```
Upon noting an unchecked strcpy()
into a global char *, needing to
locate the declaration for the
variable to discover it's size and
locate references to that variable to
see if bounds checking was performed
explicitly.
```

Another application of preventative maintenance searching was described by respondent 66. She would look through the code for:

```
mundane spell correction: how many
ways did i spell one variable name by
accident
```

If an identifier is used only once, then it is likely an error. An unused variable can be caught by a compiler or interpreter if warning levels are set appropriately, but these discrepancies can be a problem in languages that do not require variables to be declared before they are used.

Respondent 23 adds the following example of searching in order to reuse code:

```
It has also helped in the design
phase to be able to find another
program that was used for the same
purpose and this helps others to
develop their applications quicker.
```

Rather than reusing code only during the implementation phase, her team builds this into the design. Here, the code is searched to make further development easier. It's not clear how the respondent performs these searches, but the possibilities are intriguing.

The usefulness of searching during testing had a low ranking (fifth out of ten maintenance tasks), but a high rating (median of 4). An anecdote also from respondent 66 illustrates this finding:

```
how many ifdef? where are they? used
to figure out relevant test cases for
ported code
```

Hence, searching is probably not used during the actual testing of code itself, but in generating test cases.

5. Conclusion

The primary purpose of this study was to characterize the source code searching behaviour for the purpose of constructing a tool to assist programmers. An understanding of the situations in which searching was performed can be used to determine the functionality of a search tool. The archetypes can be used to improve usability during design and tool validation. During the design phase, the tool can be applied to the situations described by the archetypes. During validation, the archetypes can be used to guide selection of tasks for usability testing. A set of uncommon searches was also identified to complement the archetypes in the model.

Searching is an important part of many software maintenance tasks. We found that the tasks in which searching was most important were defect repair, code reuse, program understanding, feature addition, and impact analysis. The most common search targets were function definitions, all uses of a function, all uses of a variable, and variable definitions.

The archetypes presented in this paper are preliminary in nature. They are the fruits of an exploratory study and still need to be validated. More needs to be known about the population of software developers and maintainers and the products they support in order to develop robust models. Much work remains to be done in the collection of demographic information about software engineering.

The secondary purpose of this study was to determine the efficacy of our research method. We were able to obtain responses from respondents in different organizations from around the world, without the drawbacks of travelling. In many ways, web-based questionnaires are superior to their paper-based counterparts: the logistics of dealing with paper are eliminated; the researcher has greater control over the format and administration of the questionnaires; and the respondent submits the data in electronic form, which removes the need for transcription. We used availability sampling to obtain participants, a method that proved to be adequate for our goals.

We plan to use the results of this study in two ways. In the development of a lightweight source code searching tool, we will use them to guide our design decisions. In future empirical studies, this study will serve as an example of how survey methods can be applied to software engineering. By expanding this knowledge, we can construct a body of knowledge about the population of software developers and maintainers, and thus increase the validity of our work.

6. Acknowledgments

This research was generously supported by CSER (Consortium for Software Engineering Research) and CITO (Communication and Information Technology Ontario).

7. References

[1] M.J. Bates. The Berry-picking Search: User Interface Design. in *Interfaces for Information Retrieval and Online Search: The State of the Art* edited by M. Dillon, Chapter 4, pages 55-61, Greenwood Press, (1991).

[2] N.J. Belkin, C. Cool, A. Stein, and U. Thiel. Cases, Scripts, and Information-Seeking Strategies: On the Design of Interactive Information Retrieval Systems. *Expert Systems With Applications*, 9(3):379-395, (1995).

[3] R. Brooks. Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies*, 18: 543-554, (1993).

[4] D.A. deVaus. *Surveys in Social Research, Fourth Edition.* UCL Press, London, 1996.

[5] M. Eisenstadt. My Hairiest Bug War Stories. *Communications of the ACM*, 40(4): 30-37, (1997).

[6] W. Foddy *Constructing Questions for Interviews and Questionnaires: Theory and Practice in Social*

Research. Cambridge University Press, Cambridge, 1993.

[7] N. Frye. *Anatomy of Criticism: Four Essays.* Princeton University Press, Princeton, 1957.

[8] I.R. Katz and J.R. Anderson. Debugging: An Analysis of Bug-Location Strategies. *Human-Computer Interaction*, 3: 351-399, (1987-1988).

[9] A. Lakhotia. Understanding Someone Else's Code: Analysis of Experiences. *Journal of Systems Software*, 23:269-275, (1993).

[10] D.C Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental Models and Software Maintenance. *Empirical Studies of Programmers, First Workshop*, pages 80-98, Washington DC, USA,1986.

[11] M.B. Miles and A.M. Huberman. *Qualitative Data Analysis: An Expanded Sourcebook, Second Edition.* Sage Publications, Thousand Oaks, 1994.

[12] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An Examination of Software Engineering Work Practices. In *Proceedings of CASCON '97*, pages 209-223, Toronto, Canada, 1997.

[13] A. Strauss and J. Corbin. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*, Sage Publications, Thousand Oaks, 1990.

[14] A. von Mayrhauser and A.M. Vans. From Code Understanding Needs to Reverse Engineering Tool Capabilities. *Proceedings of the 6th International Workshop on Computer Aided Software Engineering*, pages 230-239, Piscataway, USA, 1993.

APPENDIX A: Text of the Survey

Question 1: Tools Used
What tools do you use to search source code? Check all that apply.

grep, fgrep, etc.	[ ]
find or "File Find"	[ ]
editor e.g. vi, emacs, edit	[ ]
integrated development environment e.g. MSDS	[ ]
other	[ ]

Please specify: _______________

Question 2: Program Analysis Tools
Do you use an integrated software analysis and exploration tool? Two examples are SNiFF+ and CIA.

Yes	[ ]
No	[ ]

Question 3: Development Activities Requiring Searching
How useful is it to search source code when:

	Not at all useful				Very useful
doing low-level design?	1	2	3	4	5
writing new code?	1	2	3	4	5
testing?	1	2	3	4	5
understanding old code?	1	2	3	4	5
repairing bugs/defects?	1	2	3	4	5
adding a new feature to old software?	1	2	3	4	5
improving performance?	1	2	3	4	5
inspecting and reviewing code?	1	2	3	4	5
writing documentation?	1	2	3	4	5
maintaining documentation?	1	2	3	4	5

Question 4: Typical Usage Situations
Describe one or more situations when you needed to search source code. What did you use to find it? What were you trying to find? Why did you need to find it?

Question 5: Wish List
What types of searches would you like to be able to perform?

Question 6: Primary Responsibilities
What are your primary job responsibilities? Check all that apply.

Research	[ ]
Consulting	[ ]
Developing software for a customer	[ ]
Maintaining software for a customer	[ ]
Developing a software product	[ ]
Maintaining a software product	[ ]
Developing in-house software	[ ]
Maintaining in-house software	[ ]

Question 7: Time With Source Code Written By Others

Of your total time spent working with source code, what percentage of that time is spent working on source code written by other people?

0-20%	[ ]
21-40%	[ ]
41-60%	[ ]
61-80%	[ ]
81-100%	[ ]

Question 8: Participation
Where did you hear about this survey? (Please give the name of the newsgroup or email sender.)

____

Question 9: Future Studies
Would you be willing to participate in future user studies of source code searching?

No	[ ]
Yes	[ ]

If yes, please provide your email address.
Email:_________________