How Well Do Search Engines Support Code Retrieval on the Web?

SUSAN ELLIOTT SIM
University of California, Irvine
MEDHA UMARJI
University of Maryland, Baltimore County
SUKANYA RATANOTAYANON and CRISTINA V. LOPES
University of California, Irvine

Software developers search the web for different kinds of source code for different reasons. In a previous study, we found that searches varied along two dimensions: the size of the search target (e.g., block, subsystem, or system) and the motivation for the search (e.g., reference example or as-is reuse). Would each of these kinds of searches require different technological solutions? To answer this question, we conducted an experiment with 36 participants to evaluate three different approaches (general purpose information retrieval, source code search, and component reuse), as represented by five web sites (Google, Koders, Krugle, Google Code Search, and SourceForge). The independent variables were search engine, size of search target, and motivation for search. The dependent variable was the participants judgement of the relevance of the first ten hits. We found that it was easier to find reference examples than components for as-is reuse and that participants obtained the best results using a general-purpose information retrieval site. However, we also found an interaction effect: code-specific search engines worked better in searches for subsystems, but Google worked better on searches for blocks. These results can be used to guide the creation new tools for retrieving source code from the web.

Categories and Subject Descriptors: D.2.3 [**Software**]: Coding Tools and Techniques; D.2.13 [**Software**]: Reusable Software; H.3 [**Information Systems**]: Information Storage and Retrieval; H.5 [**Information Systems**]: Information Interfaces and Presentation

General Terms: Design, Human Factors, Languages

Additional Key Words and Phrases: Empirical study, open source, opportunistic development, search archetypes

This research was supported by a Collaborative Research Initiation Award from the Donald Bren School of Information and Computer Sciences. Authors' addresses: S.E. Sim, S. Ratanotayanon, and C.V. Lopes, Department of Informatics, Donald Bren School of Information and Computer Sciences, University of California, Irvine, 5019 Donald Bren Hall, Irvine, CA, 92697. M. Umarji, Department of Information Systems, University of Maryland, Baltimore County, 1000 Hilltop Circle, ITE Building, Suite 404, Baltimore, MD, 21250.

1. INTRODUCTION

As the quantity and quality of open source software increases, an approach to software development that takes existing components and combines them becomes a viable and competitive way to do business. "Opportunistic" or "pragmatic" reuse is the unplanned, ad hoc use of existing source code that was not specially packaged for reuse [Hartmann et al. 2006; Holmes and Walker 2007]. It often involves modification of the code being reused, or creation of wrappers or glue code. These practices represent a departure from traditional software reuse, which tends to focus on reusing components without modification. The success of an opportunistic software development approach on a project depends in part on finding what is out there [Hartmann et al. 2006].

Software developers search for source code on the web for a variety of reasons. A special issue of *IEEE Software* on how open source is affecting software development gave a number of case studies [Spinellis and Szyperski 2004]. As well, we identified a series of archetypes in a previous study for the types of searches that software developers carry out [Umarji et al. 2008]. We found that there were two major search motivations; participants tended to search for either a piece of code that could be reused as-is in a project, or for a reference example that could be consulted for information. The size of search targets varied, ranging from a block (e.g., a few lines of code) to a subsystem (e.g., an algorithm or data structure) to an entire system (e.g., a text search engine).

These results led us to the insight that "searching for source code on the web" included a wide range of behavior. Furthermore, the different kinds of searches required different kinds of tool support. In some cases, the participants were looking for information, which would suggest that tools from textual information retrieval could be used. In other cases, participants were looking for a software artifact, which would suggest that code-specific search tools, such as those found in integrated development environments (IDEs), and software reuse could be applied.

The question arises: how well do different search technologies meet the needs of software developers when retrieving code from the web? While there have been many studies of user behavior in information retrieval and the practice of open source, the behavior of users who search for source code is a relatively unexplored area.

To further understand source code retrieval, we designed a laboratory experiment to evaluate the effectiveness of different technological approaches to web-based code retrieval. The purpose of this study was to evaluate fitness for purpose, rather than to campare the underlying algorithms in the search engines. By fitness for purpose, we mean the how fit are each of the approaches for the purpose of finding code on the web. We created scenarios based on the results from our previous study [Umarji et al. 2008], and chose to vary both the motivation for the search (as-is reuse or reference example) and the size of the search target (block or subsystem). We used five search engines in our study. One was designed for information retrieval (Google), three were web sites designed to search for source code (Koders[1], Krugle[2], and Google Code Search[3]), and the last was a project hosting site (SourceForge[4]). We selected these sites because they were mentioned by participants in the previous study and because they represented different technological ap-

[1]http://www.koders.com/
[2]http://www.krugle.com/
[3]http://www.google.com/codesearch
[4]http://sourceforge.net/

proaches from textual information retrieval, source code searching, and software reuse.

Thirty-six participants were each assigned a scenario and asked to conduct the search on the five search engines. The subjects were asked to judge the relevance of the first ten matches returned (denoted as P@10). We found two statistically significant main effects and one interaction effect. Searches for reference examples produced more matches that were judged relevant ($p < 0.05$) . Google produced more relevant hits than the other search sites ($p < 0.01$) . However, Koders and Krugle were more likely to produce more relevant hits on searches for subsystems, while Google was better when searching for blocks ($p < 0.01$) .

To sum up, it is easier to find reference examples than components for as-is reuse. Google performed better overall than the other search engines, but Koders and Krugle performed better when searching for subsystems. The motivation behind the search did not have a statistically significant effect in this study.

Looking at the reasons behind these results, we noticed that searching the web for code was not a single, universal phenomenon. It is, in fact, a variety of activities that appear to be superficially similar, but are qualitatively distinct from each other. When designing tools for code retrieval, one must be clear about what kind of search is being addressed. Stating one's goals and the use cases for a tool up front permits alternatives to be compared more easily.

The remainder of this paper is organized as follows. We summarize the results of our previous study in Section 2. In Section 3, we review related work on source code searching, information retrieval, and software reuse. Our experiment and results are described in Sections 4 and 5. We discuss some implications of our results in Section 6 and present concluding remarks in Section 7.

2. ARCHETYPES OF WEB-BASED CODE SEARCHES

Previously, we conducted a web-based survey to collect data on a range of source-code searching behaviors. This study has been described in detail elsewhere [Umarji et al. 2008], so we give only a summary here.

2.1 Method

We used an online survey with 13 closed-end questions and two open-ended questions. The survey had questions about: types of information sources used by programmers while searching; popular search sites; selection criteria for code; and the search process. We solicited participants from a number of mailing lists, newsgroups, and our own social networks. We had 69 participants who provided a total of 58 anecdotes of searches that they had performed. (Some participants provided more than one anecdote, while others did not provide any, electing to answer only questions from other parts of the survey.) A majority of the developers that responded to our survey were programmers who used Java (54), C++ (58) or Perl (32). Most participants were familiar with more than one programming language. We analyzed the anecdotes for recurring patterns using open coding [Miles and Huberman 1994] and a grounded theory approach [Strauss and Corbin 1990].

2.2 Archetypes and Strategies

Our analysis revealed that there are two major archetypes: i) searching for components for use as-is in a system being constructed (34 anecdotes); and ii) searching for reference examples (17 anecdotes). There were an additional seven anecdotes that did not fall into these two archetypes. The major archetypes have sub-types that vary in the size of the search target (block, subsystem, and system). In the example anecdote given below, the motivation or goal was to find a reference example and the size of search target was a subsystem, because the participant was looking for a usage example of the SWT, which is a subsystem.

> *Sometimes I did a source code searching when I don't know how to use a class or a library. For an example I didn't know how to create a window using SWT class. I did a Google search with the description of what I want to do. I decided on the best match based on whether I understand the example code.*

Table I summarizes the frequency of searches categorized along these two dimensions.

		Major Archetypes		
		Reuse	Reference Examples	Row Total
Subtypes	Block	Code snippets, wrappers (7)	Lines, block (4)	11
	Subsystem	Data structures, algorithms, GUI widgets (21)	Implementation example, usage example (11)	32
	System	Application (6)	Approaches, ideas (2)	8
	Column Total	34	17	51

Table I. Frequency of Archetypes with Subtypes

Although we categorized the searches into archetypes and subtypes, these dimensions of variables are continuous. A programmer could be searching for a component to drop into a project to use without making any changes, or for an example to inform a re-implementation effort, or some combination of the two. For instance, a programmer may be willing to make minor, even major, modifications to a component, so it can be reused. On occasion, the search is initially seeking a re-usable component and when that fails, it becomes a search for reference information. The size of search targets ranged from whole systems to a few lines of code.

There were only 7 searches that did not fall into these categories. Four of these were searches for information about bugs or defects. Programmers were looking for confirmation, a patch, or additional information for a bug that they had found. In one search, a programming language designer was looking for examples of how Python syntax was used before modifying a feature. We were not able to categorize the final two searches, because not enough detail was provided.

The searches for systems were qualitatively different from the others. Often these started with a pointer obtained from a friend, such as the name of a system, e.g., Eclipse or Lucene. Consequently, the ways in which the searches were carried out were different. An important reason is a system often has its own web page and documentation. Once a software developer has a name of a system, it is a simple matter to type it into a search engine and find a home page. As well, system often served as a reference example for developers who could not use open source in proprietary, closed source projects. These developers used these systems as a source for ideas on how to design and implement their own systems, or new features. Also, when looking for ideas, the search parameters and selection criteria can be very broad.

The other four archetypes were similar in that they all required some work or know-how with a search engine to find them. Software developers needed to find appropriate search terms, filter through matches, expand the set of matches, or all of these iteratively. Consequently, in our experiment to evaluate search engines we will be focusing on searches for blocks and subsystems only.

2.3 Tools and Information Sources

We were interested in the kinds of tools and information sources that participants used in searching for source code. We obtained data from a closed-ended multiple choice question and from analyzing the anecdotes. The results are shown in Table 2.3 and Table III, respectively. In the multiple choice question, participants were asked to select all options that applied, so the total count exceeds the sample size. In the latter table, only specific and explicit reference to a web site or search engine were counted.

Table II. Reponses to Multiple Choice Question: Which information sources do you use to search for code?

	Count
Google, Yahoo, MSN Search etc	60
Domain knowledge	37
Sourceforge.net, freshmeat.net	34
References from peers	30
Mailing lists	16
Code-specific search engines	11

Table III. Search Engines and Site Mentioned in Open-Ended Questions

	Count
Google	28
Specific web site	9
Mailing lists and forums	6
SourceForge	3
Scientific articles	2
Yahoo!	1
Krugle	1
freshmeat.net	1

In the multiple choice question, 60 of the 69 participants said they turned to a general-purpose search engine. This data is corroborated by the answers given by the participants in the anecdotes. In the 58 scenarios, Google was mentioned 28 times and Yahoo! was cited once.

The next most common information source was domain knowledge. The name of a system plus a little context coaxes good results even from search engines not designed to search source code. Although general-purpose search engines were not designed to be used with source code, they work well enough because software developers are looking for functionality, not elements in the source code. In the anecdotes, participants also refered to specific forms of domain knowledge. Nine reported going directly to a specific web site, such as archives, repositories, and tutorial sites. Two others used scientific articles to help them locate code.

The next category of interest was project hosting sites, with 34 of the respondents using them for source code search. In the anecdotes, SourceForge was mentioned three times and freshmeat.net was mentioned once.

Participants also consulted friends or colleagues for suggestion. A recommendation and a good text search engine is a powerful combination. The recommendation usually names a system, which can then be used as a search term.

Mailing lists (16/69 responses) also provided good information. In the anecdotes, participants also mentioned mailing lists or forums six times. On these lists, newsgroups, and online forums, other programmers talked about source code. Consequently, natural language vocabulary becomes associated with a particular piece of software by proximity. Once again, it is a work-around the problem of specifying functionality using code elements. The natural language search keywords match words in posts on mailing lists, and these posts in turn lead the programmer to the source code.

Only 11 out of 69 respondents reported using a code-specific search engine. Only one participant named such a site (Krugle) in their anecdotes. In the comments box at the end of the survey, some were very skeptical of search engines for source code. One programmer wrote, "In short, I would never rely on a 'code search engine'. This idea is just plain silly. Sort of ivory tower. If you want to find something usable you have to look for 'people already using it."'

3. APPROACHES TO CODE RETRIEVAL

By some estimates, there are billions of lines of code in countless programming languages available on the web [Deshpande and Riehle 2008]. With this embarrassment of riches, comes a problem: locating the code that one wants. A natural response is to build a search engine. If code retrieval on the web is a variation on a well-understood problem, then it is a problem than can be solved using existing tools. At times, it resembles a problem from conventional source code searching, software reuse, or information retrieval. But a closer examination reveals that code retrieval is a strange hybrid of all of these, and will require new approaches and technology.

3.1 Textual Information Retrieval

Textual information retrieval is the discipline of organizing, searching, and presenting documents from large repositories [Manning et al. 2008]. General-purpose web search engines, such as Google and Yahoo!, are classic examples of information retrieval systems. "Document" is the term applied to records because they typically, but not necessarily, contain text. Searches are generally performed using keywords, a specialized dictionary, and/or Boolean operators. Current research in the area deals with increasingly large collections of documents by creating more robust infrastructure and better algorithms for summarizing results and answering questions.

This class of technology was represented in our study by Google. With this web site, searches can be specified using regular expressions and there are no special features for code search. Filtering can be achieved by using additional keywords in the search. Google uses the PageRank algorithm [Langville and Meyer 2006] and other proprietary algorithms to retrieve and order the presentation of documents. Highlighting of matched search terms is available in the cached version of the document.

3.2 Source Code Searching

Code search is a key part of program comprehension in software development. In an empirical study of software engineering work practices, Singer et al. [1997] found that searching was the most common activity for software engineers. They were typically locating a bug or a problem, finding ways to fix it and then evaluating the impact on other segments. Program comprehension, reuse, and bug fixing were cited as the chief motivations for source code searching in that study. A related study on source code searching by Sim, Clarke, and Holt [1998] found that the search goals cited frequently by developers were code reuse, defect repair, program understanding, feature addition, and impact analysis. They found that programmers were most frequently looking for function definitions, variable definitions, all uses of a function and all uses of a variable.

The recognition that search is powerful and useful has led to advances in code search tools. Software developers have needed tools to search through source code since the beginning of interactive programming environments. It started with simple keyword search and when regular expressions were added, it became possible to specify patterns and context [Thompson 1968]. An important improvement was made when search techniques started using program structure, such as identifiers of variables and functions, directly in expressing search patterns [Aiken and Murphy 1991; Paul and Prakash 1994]. Another approach to syntactic search involves processing the program and storing facts in a database file of entity-relations [Chen et al. 1990; Linton 1984]. Alternatively, the code can be parsed and transformed into other representations, such as data flow graphs or control flow graphs, and searches can be performed on those structures [Murphy and Notkin 1996]. While some of these contributions have yet to be widely adopted, searches using regular expressions and program structure are standard in today's integrated development environments (IDE).

This class of technology was represented in our study by Koders, Krugle, and Google Code Search. The characteristics of these web sites are summarized in Table IV. The first three rows have been excerpted from Hummel, Janjic, and Atkinson [Hummel et al. 2008].

3.3 Software Reuse

Software reuse usually means the reuse of code from a library as-is without modification [Mili et al. 1998; Prieto-Diaz 1991]. In this view, components should be used as black boxes, that is, to be used without change. Modification is an expensive operation; making non-trivial changes quickly increases the effort of understanding a component and any savings in effort over implementation from scratch quickly diminishes [Ravichandran and Rothenberger 2003; Holmes and Walker 2008]. The approach of taking existing components and using them on a new software project is not a new one. What is new is the way in which it is carried out; the quality and quantity of open source code that is available means that software developers shop first and ask design questions later. Others have made this same observation and applied their own labels to it. Noble and Biddle [2002] called it "postmodern programming," Boehm [2006] used the term "systems of systems," and Carnegie Mellon's Software Engineering Insitute refers to the phenomenon as "ultra-large-scale systems" [Northrop et al. 2006].

Research in software reuse is concerned with topics such as design for reuse and making reusable components easier to find. For instance, in component-based software engineering (CBSE), reuse is planned and components are created and packaged for that purpose.

		Koders	Krugle	Google Code Search	SourceForge
Repository	No. of Indexed Files	>1 million	> 10 million	> 10 million	173,065 projects
	No. of Java Files	600 000	3.5 million	>2.5 million files	None
Search Features	Retrieval Algorithm	Keyword and name matching of codes from large open source hosters	Keyword and name matching in open source code search for technical Web pages	Keyword matching of open source with regex support	Keyword matching
	Specifying Searches	Text keywords and regular expressions, plus drop-down and check boxes for filtering	Text keywords with drop-down menus for filtering	Text keywords regex, and special qualifiers in standard mode. Additional fields and boxes for filtering in advanced mode.	Text keywords in standard mode. Filtering fields and selection in advanced mode.
	Regular Expressions	Yes	No	Yes	No
	Matching	Syntax recognition of source code	Syntax recognition of source code	Syntax recognition of source code	Plain text on project names and descriptions
	Filtering	By file types, class, method, and interface. Can limit scope to a project.	By comments, source code, functions, function calls, and classes. Can limit scope to a project.	By patterns in the name of files and packages.	By project categories, project registration date, and activity rank.

Table IV. Characteristics of Search Sites

Selection of a component is driven by a set of requirements that have been specified in advance. Work on software reuse repositories has included packaging code into libraries for reuse, constructing archives of reusable components, and search on those repositories.

This class of technology was represented in our study by SourceForge (see Table IV.) The search feature on this web site matches keywords entered by the user to terms on the home pages of the various projects; the source code on the projects is not searched. Information on the web pages includes a great deal of metadata, such as the age of the project, category, license, activity level, popularity, and descriptions of the project.

4. METHOD

In Section 2, we reviewed the results of a prior study that categorized the different kinds of searches for source code undertaken by software developers. In the previous section, we examined different approaches to code search based on technologies from information retrieval, source code searching within a single project, and software reuse. In this section, we report on a study to evaluate the effectiveness of the different approaches with the goal of improving our understanding of the nature of code retrieval on the web. In this study, we gave participants a search scenario to perform on five search engines and asked them to rate the relevance of the first ten hits returned.

4.1 Independent and Dependent Variables

In the experiment, each participant was given a scenario and asked to perform the search using five different code search engines. We used multiple scenarios to represent four combinations of the two search archetypes (reuse as-is and reference example) and two sizes of search targets (block and subsystem).

Across subjects, we used three independent variables in this study: search engine, search motivation, and size of search target.

Search engine was treated as a within-subjects independent variable with five levels: Google, Koders, Krugle, Google Code Search (GCS), and Sourceforge. These five were included because they each represented a class of search engines that were mentioned in the initial survey. A within-subjects factor allows us to use a subject as his or her own control, thereby allowing us to partition variation in performance more accurately. In other words, by using search engine as a within-subjects factor, we are able to factor out some personal idiosyncrasies when making judgments about relevance of matches and the search engines.

The two remaining independent variables were between-subjects factors. There were two levels of the search motivation, corresponding to the two archetypes (reuse as-is and reference example) that we identified in the first study. There were two levels of the variable size of search target. We decided to focus on block and subsystem, because the search strategies for these relied less on suggestions from peers. A block is a few lines of code, similar to a basic block in source code, such as a call to API or a form in HTML. A subsystem could be a GUI widget, library, or data structure. This decision allowed us to make the experiment self-contained, yet consistent with what we found in our previous study. Also, by excluding other software developers as an information source, subjects were required to rely more on features in the search engines.

The dependent variable was the performance of the search engines. We operationalized this as the precision, of the first ten matches returned, or P@10. Participants were asked to look at the first ten matches and give a binary relevance judgement. The sum of these judgements were aggregated and divided by ten, giving a proportion between 0 and 1. If fewer than ten matches are returned, the denominator is the number of matches returned.

Precision and recall are two widely-used measures from information retrieval for evaluating search engines. However, they are very difficult to calculate on a large set of records, because it requires an oracle (usually a human) to generate a relevance rating for every record. Consequently, the P@10 metric was developed and it has been found to be an appropriate surrogate because it is predictive of overall search engine effectiveness and users rarely go beyond the first ten results [Craswell and Hawking 2004].

4.2 Creating Scenarios

We used the archetypes from our previous study to create scenarios for evaluating source code search tools. Whereas archetypes are abstract, scenarios are more concrete and describe a specific instance of search.

In our case, scenarios were designed to include the following information: i) goal of search (component or example); ii) size of search target (block, or subsystem); iii) programming language; iv) context of search; and v) a situation linking the first four parts.

All of this information is necessary to judge the relevance of an item returned by a search engine. The first two parts of the scenario are based on the archetypes from our previous study. Programming language is included because code in some languages is easier to find than others, e.g., PHP vs. C. As well, the context affects the usefulness of items returned, i.e., academic vs. industrial settings, Eclipse plug-ins vs. web sites. An example of a usage scenario is presented below.

> *You are working in the Python programming language, and need to have multi-threading functionality in your program. You have never used threads before, and would like to know how to create threads, to switch between threads, and so on. Look for examples by which you can learn. Any thread implementations of Python programs are relevant. Remember you will not be using the code directly, you will like to learn how to use it.*

To ensure that the treatment combinations were not biased by the specific stimuli presented to the participants, we generated multiple scenarios per treatment combination. Two scenarios were created for each of the two searches for blocks. Three scenarios were created to be used in each of the two searches for subsystems; we created more scenarios for this level of the independent variable to reflect the wider range of searches for subsystems that we found in our previous study.

We used a Latin square design to assign 36 subjects to the 20 (5x2x2) conditions. Each participant used all five search engines. Consequently, each combination of motivation and size was used with nine participants.

4.3 Procedure

The procedure in our study had three stages: training, experiment, and debriefing.

The training stage allowed the participants to become familiar with the experiment set and task. There was one warm-up task for participants to become accustomed to the think aloud procedure. In a training task, participants were given a simple scenario, asked to search for source code, and rate the relevance of the first three matches returned. We recorded audio, video, and screen activity as they worked.

In the experiment, participants were randomly assigned to a condition that was a combination of search motivation and size of search target. Participants were given a scenario and were asked to perform the search using five search engines. The participants were free to use the search engines in any order they wished, but they were required to use all five. They were allowed to change and refine their queries as many times as they liked. Once they arrived at a set of search results that they were satisfied with, they were asked to rate the relevance of the first ten matches. They were allowed to make any investigations necessary to inform their subjective judgments. We asked participants to provide P@10 judgements, not because we are interested in combining their subjective opinions to evaluate the search engines using the collective wisdom of the crowds. Instead, we are interested in their experiences in using the search engines. We expect that these lessons will be applicable to understanding the appropriateness of different technological approaches to different kinds of code search.

Finally, the debriefing stage consisted of two questionnaires. The first one asked about their preferences regarding the search engines that they had used. The second questionnaire was on their background and code search experience.

4.4 Participants

Thirty-six participants were recruited for this study based on the criteria that they should have some prior programming experience (either professional or academic). Most of the participants were graduate students and all had work experience, either in product development or industrial research. The average age of participants was 26.9 and they had on average 4.2 years of programming experience. All had previously searched for source code on the web, had used multiple programming languages, and worked with a team. Fifty percent of participants reported that they had searched for source code "Frequently" and 39% searched for it "Occasionally." All the participants had experience with HTML and C programming, 94% with Java, and 83% had worked with C++. Sixty-four percent of the participants cited their primary job responsibility as "Programming". The characteristics of the participants are summarized in Table V below.

Participant	Search the Web for Code	Years of Professional Experience	Primary Job Responsibility	Age	Participant	Search the Web for Code	Years of Professional Experience	Primary Job Responsibility	Age
1	Frequently	10	Programming	42	**19**	Occasionally	1	Programming	23
2	Frequently	6	Programming	29	**20**	Occasionally	1	Project Management	28
3	Frequently	0	Programming	22	**21**	Occasionally	7	Programming	29
4	Frequently	5	Programming	NR	**22**	Occasionally	5	Programming	36
5	Frequently	4	Programming	NR	**23**	Occasionally	5	Software Architecture	NR
6	Frequently	4	Programming	NR	**24**	Occasionally	3	Programming	22
7	Frequently	5	Programming	NR	**25**	Occasionally	8	Software Architecture	26
8	Frequently	4	Programming	29	**26**	Occasionally	10	Programming	NR
9	Frequently	1	Programming	23	**27**	Occasionally	1	Testing/QA	23
10	Frequently	4	Programming	NR	**28**	Occasionally	1	Research	23
11	Frequently	6	Project Management	28	**29**	Occasionally	6	Research	23
12	Frequently	5	Programming	30	**30**	Occasionally	5	Project Management	33
13	Frequently	3	Research	28	**31**	Occasionally	3	Programming	27
14	Frequently	4	Programming	27	**32**	Occasionally	3	Programming	24
15	Frequently	5	Programming	26	**33**	Rarely	1.5	Product Management	23
16	Frequently	7	Programming	30	**34**	Rarely	1.5	Testing/QA	23
17	Frequently	4	Programming	26	**35**	Rarely	2	Testing/QA	25
18	Frequently	7	Programming	24	**36**	Rarely	3	Technical Writing	NR

Table V. Characteristics of Participants

4.5 Hypotheses

Our previous study of search behaviors indicated that software developers performed a variety of searches. We also found that search engines differed widely in their algorithms and implementations. Given these differences, it was likely that some search engines would perform better on some search scenarios than other search engines. In other words, we are predicting an interaction effect between the search engines and the kinds of searches that are performed. Our hypotheses, and the corresponding null hypotheses, are as follows.

4.5.1 *Main Effects.* There are three main effects corresponding to the three independent variables in the study.

The first hypothesis pertains to the relationship between the motivation for the search and the success of the search. We expect searches for reference examples to be more successful, because the criteria for relevance are more flexible.

H_{M_0} : *There is no difference the precision of the first ten matches when searching for as-is reuse and for reference examples.*

H_{M_1} : *The precision of the first ten matches is higher when searching for reference examples than for components to be used as-is.*

The second hypothesis pertains to the relationship between the size of the search target and the success of the search. We expect searches for blocks to be more successful, because there are more blocks of code than subsystems in repositories and on the web.

H_{S_0} : *There is no difference in the precision of the first ten matches when searching for blocks or subsystems.*

H_{S_1} : *The precision of the first ten matches is higher when searching for blocks than for subsystems.*

The third hypothesis pertains to the relationship between the search engine, i.e., the technological approach, and the success of the search. Google was frequently used by participants in our prior study, so we are using popularity as an indicator of quality in this instance. If Google did not work well enough, or at least better than the alternative, software developers would not be mentioning it so often in their anecdotes and they would have found a better alternative.

H_{E_0} : *There is no difference in the precision of the first ten matches across the search engines.*

H_{E_1} : *The precision of the first ten matches is higher when using a general-purpose search engine than the other types of search engines.*

4.5.2 *Interaction Effects.* With the three independent variables, there are four possible interaction effects. These interaction effects are of interest, because we expect to see the success of a search to depend on more than one factor.

We draw particular attention to the interaction effect between motivation and the search engine. In other words, the success of a search with a particular motivation depends on which search engine is used. We expect that it would be easier to use SourceForge, for example, to find subsystems, than the other search engines. Our rationale for this hypothesis is that searching for subsystems was most similar to the searches performed in software reuse and that SourceForge, being a representative of software reuse repositories, would be most compatible.

H_{ME_0} : *The precision of the first ten matches when searching with a given motivation will not change when the search engine changes.*

H_{ME_1} : *The precision of the first ten matches will be higher when using a general-purpose search engine to search for reference examples, than the other types of search engines.*

5. RESULTS

5.1 Main Effect of Motivation

The ANOVA revealed that there was a main effect from the motivation of the search (F(1, 32) = 4.98, $p < 0.05$). The F statistic is used to test for significance in an Analysis of

Table VI. ANOVA Results on P@10

Between Subjects	df	Sum of Squares	Mean Sum of Squares	F value	p
Size	1	0.002	0.002	0.019	0.890
Motivation	1	0.533	0.533	4.988	0.033 *
Size:Motivation	1	0.014	0.014	0.135	0.716
Residuals	32	3.416	0.107		
Within Subjects	**df**	**Sum of Squares**	**Mean Sum of Squares**	**F value**	**p**
Engine	4	0.980	0.245	4.109	0.0036 **
Size:Engine	4	0.974	0.243	4.080	0.0038 **
Motivation:Engine	4	0.106	0.026	0.443	0.7771
Size:Motivation:Engine	4	0.260	0.065	1.090	0.3644
Residuals	128	7.635	0.060		

Significance: * 0.05 ** 0.01

Variance. The numbers in the brackets (1, 32) show the degrees of freedom in the statistic. The p-value for a statistic is an indicator of the likelihood that an effect was detected purely by chance, which is low in this case. Searches for reference examples had $P@10_{avg}$=0.43, while searches for components reuse as-is had $P@10_{avg}$=0.32, as can be seen in Figure 1. The partial η^2 statistic is a measure the size of the effect, that is, percentage of variance explained in the dependent variable by a predictor controlling for other predictors An effect size estimate is a scale-free measure of how large is difference between different conditions [Kampenes et al. 2007]. For this effect η^2=0.38, indicating a medium effect size [Cohen 1988], which means that the difference is statistically significant and large enough to be meaningful. In other words, participants found it easier to locate examples to consult than code that they could use straight away on a project.

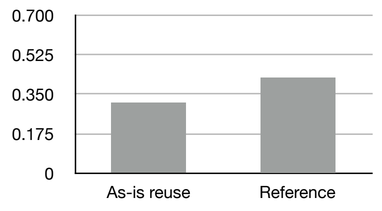
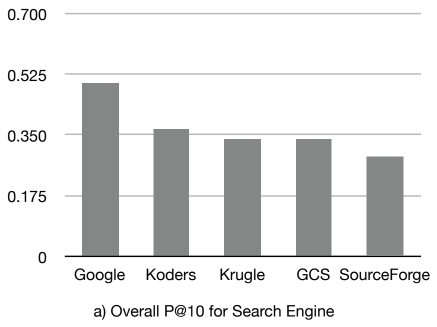


Fig. 1. P@10 for Searches with Different Motivations

5.2 Main Effect of Search Engine

In addition, there was a main effect from the search engines (F(4, 25) = 4.109, $p < 0.01$). The effect size was large with η^2=0.7. Looking at the performance of individual search engines, we can see that Google had $P@10_{avg}$=0.50, followed by Koders $P@10_{avg}$=0.37, and the remainder clustered around 0.29-0.34, as can be seen in Figure 2a. A post hoc comparison using Tukey's Honestly Significant Differences Test found no difference between the other search engines.

In the debriefing, participants also stated that they preferred Google. We asked them to rank the search engines from 1 (high) to 5 (low) in order of their overall preferences, perceived ease of use, and available features. The participants' answers are summarized in



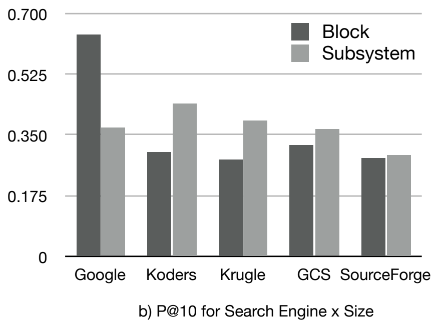


Fig. 2. Effect of Search Engines and Interaction Between Search Engine and Size

Table 5.2. We found statistically significant differences in overall preference and perceived ease of use, but not for the available features. In terms of general preference, Google was the clear winner with a median rank of 1.92, and was statistically significant using Friedman's test ($\chi^2_4 = 16.8, p < 0.01$). Google was also perceived to be easier to use ($\chi^2_4 = 20.13, p < 0.0001$). Interestingly, participants found GCS easier to use than Koders and Krugle, despite its lower performance on the precision of the searches. SourceForge also scored lowest on this measure. Participants seemed to like the features in GCS the best, but the differences between the search engines were small and not statistically significant.

Table VII. Mean Rank of Participants Preferences (1 = high, 5 = low)

	Preference	Ease of Use	Features
Google	1.92	1.92	3.00
Koders	2.83	3.25	2.54
Krugle	3.17	3.25	2.69
GCS	3.17	2.92	3.15
SourceForge	3.92	3.67	3.63

A reason often given for preferring Google is that this site helped them to learn more both about the problem they were given and the available solutions. A few other subjects preferred Koders or Krugle, because it provided the best match without extraneous information.

5.3 Interaction Effect Between Size and Engine

The final statistically significant relationship that we found in the ANOVA was an interaction effect between the search engine used and the size of the search target (F(4, 25) = 0.003, $p < 0.01$). Further examination of the mean P@10 values for each of these conditions shows that it is easier to find blocks of code on Google, but that Koders and Krugle were better for finding subsystems (see Figure 2b). Blocks are a few lines of code, such as a regular expression to check the format of an email address. A subsystem is larger, for instance, an XML parser. Here also the effect size was large with η^2=0.7.

5.4 No Other Significant Effects

We did not find any other statistically significant effects. In particular, we failed to reject the null hypothesis for size and the interaction effect between motivation and search engine. We had expected blocks to be easier to find than subsystems, but found no difference. As well, we had expected there to be an interaction effect between motivation and search engine, but found no difference.

5.5 No Order Effect

Nearly half of the participants preferred to use Google first in their searches. We had originally planned to randomize the order of presentation of the search engines to mitigate learning effects, but later had to abandon these plans. During pilot testing, we were met with an extremely high level of resistance from the subjects; multiple subjects simply refused to use the search engines in the order that we requested and insisted on using Google first. Instead, we presented participants with a list of search engines to use and allowed them to choose the order. Seventeen out of the 36 participants used Google first; of the remainder, 12 used the search sites in the order they were given on the list, and the rest started with a site that they were familiar with. In cases when the participants were not familiar with the programming language or with the problem at hand, they preferred to use Google first to gain a basic understanding of the scenario. But there is no evidence that Google raised the P@10 of the search engines used subsequently. The average P@10 was slightly higher for search engines chosen earlier, but an ANOVA revealed that this difference was not statistically significant. In other words, there did not appear to be a significant learning effect nor a bias due to order in which the search engines were used.

5.6 Testing Analysis of Variance Assumptions

When performing a statistical analysis of a multi-factorial design, it is necessary to confirm the homogeneity of variances between conditions, also known as sphericity, to ensure that the data does not violate one of the underlying assumptions of one the statistical tests. A sphericity test confirmed the homogeneity of variances between conditions (Mauchly Criterion = 0.690, $p > .25$, n.s.), which allowed us to proceed with a three-way ANOVA on the precision of the first ten matches returned, or P@10. We used two between-subjects factors (size of search target and motivation of the search) and one within-subjects factor (search engine). Results are shown in Table VI. There were three statistically significant effects. Furthermore, post hoc power analysis using G*Power [Faul et al. 2007] indicated that β=1, so the likelihood of a Type II error, or false negative, is negligible.

5.7 Threats to Validity

Our main threat to the validity is the use of relevance judgements to evaluate search engines. Participants did not always make correct judgements on whether a match was relevant to the scenario that they were given. While observing the experiments, we noticed that errors were made from time to time. However, the population of users of search engines out in the world would likely make similar errors. Also, running multiple participants in each condition allowed us to make inferences about how software developers searched as a group. The presence of judgment errors also affects the kinds of conclusions that we can draw about the search engines. Consequently, the P@10 metric is more about the applicability of the different technological approaches than some absolute measure of the algorithmic capabilities of the different search engines.

Another threat to validity is a possible bias in participants in favor of search approaches that they are familiar with. In our study, we found that a general-purpose search engine had a higher precision than the others. Participants may have found them easier to use, because they had greater past experience with them. For example, none of them used the feature in Koders that allowed them to limit searches to projects. However, the code search engines that are available on the web are not restricted to trained, expert users, which means our participants are representative of the general user population in this respect. Furthermore, participants were very willing to adapt their queries to take advantage of available features. While we can not rule out familiarity as a confounding factor, our research design does allow for a head-to-head comparison of the search approaches.

Assigning specific search scenarios was a necessary part of the experiment procedure, but introduced a serious confound. Being assigned a search scenario is very different from coming up with a need on one's own. With an assigned scenario, there is always some level of unfamiliarity and in turn a need to learn something about the problem, programming language, or any other unknowns. In such situations, participants always turned to Google to fill in background knowledge. Also, we provided some guidance on how to make relevance judgements to provide consistent direction for participants to complete their task. We made these criteria broad (including matches that aided learning), so that participants would have latitude to judge more matches relevant. Consequently, these stimuli may have biased the results in favor of a general-purpose search engine. However, learning is an essential part of opportunistic software development, so this bias should not invalidate the results.

6. DISCUSSION

In this section, we discuss and examine the results of the experiment. To sum up, we had three statistically significant results. We rejected the null hypothesis for a motivation effect H_{M_0} and found that it was easier to find reference examples. We rejected the null hypothesis for a search engine effect H_{E_0} and found that a general-purpose search engine (Google) performed best overall. We failed to reject the null hypothesis for a size of search target effect. As well, we rejected the null hypothesis for an interaction effect between search engine and size of search target H_{ES_0}, meaning that it was easier to find subsystems using code-specific search engines. Finally, we failed to reject the null hypothesis for all other interaction effects.

6.1 The Effect of Motivation

Searching for source code on the web can have different motivations such as learning to complete a programming task or to advance software development on a project. Much of the work on software development using open source has been concerned with as-is reuse of components. While there is no doubt that this occurs frequently, the focus on as-is reuse has overlooked the use of open source as reference examples and for learning. Approximately one third of the anecdotes collected in our previous study were searches for reference examples.

In this experiment, it was easier to find reference examples, likely because the criteria for judging relevance are more flexible. Code that is close to what the searcher desires is sufficient. On the other hand, the criteria for judging the relevance of a component to be used in implementation are more strict. Not only does the functionality need to match, but also the interfaces, data models, and so forth in order for the component to be compatible.

While there has been significant work on component reuse [Frakes and Kang 2005], there has been somewhat less work on finding source code examples. Code retrieval on the web could also be improved through the application of recommender systems [Holmes and Murphy 2005; Zimmermann et al. 2005]. These tools identify sections of code that are related to the software developer's current task. Tools such as Strathcona [Holmes and Murphy 2005], XSnippet [Sahavechaphan and Claypool 2006], exemplar [Grechanik et al. 2007], and Assieme [Hoffmann et al. 2007] assist developers who are using complex APIs or frameworks by providing examples of structurally similar source code and examples of API usage. Source code search for reference examples is a good application for these tools, because software developers can easily make use of matches that are "good enough" as reference examples.

We have identified two possible reasons for the relative lack of attention to reference examples in software development using open source. One, the impact of open source in the form of components is more visible and longer-lived than when the code is used for ideas for software design and implementation. A component and its corresponding files that have been incorporated into a software system are clear evidence of the open source reuse. In contrast, an idea for software design is much more ephemeral. As Kamp [2004] pointed out, when a re-implementation occurs, any link to previous work is lost. Two, reference materials have traditionally been books and other documents. While text books and reference manuals do contain source code, they rarely spring to mind when in the context of reuse. But as these resources move onto the web, the web becomes a giant desk reference that developers use to look up information.

6.2 The Effect of Search Engine

The search engines used in the experiment were representatives of different technological approaches to code retrieval on the web. Informational retrieval was represented by Google. Source code searching was represented by three search engines, Koders, Krugle, and GCS. These web sites had functionality similar to what one would find in an IDE. Finally, software reuse repositories were represented by SourceForge.

Overall, Google performed better than the other sites. The remaining search engines had lower P@10 values, but were not significantly different from each other. Considering the simplicity of the search feature, SourceForge did well to keep up with Koders, Krugle, and Google Code Search, which were purpose-built tools. There are two possible ways to interpret this result, and we feel that there is a grain of truth in both of them.

One interpretation is that there is a stronger affinity between information retrieval approaches and the problem of searching for source code on the web than the other approaches. This possible mismatch between code search on the web and code search in an IDE raises the question of whether software developers are really searching for source code, per se. In our study, the participants rarely clicked through to the source code. When looking for examples, developers use the search results to figure out how to do something, but they are trying not to get bogged down in details and design rationale. Code quality was not evaluated and searchers make very quick assessments of appropriateness. We posit that they are looking for a kind of "executable know-how." By know-how, we mean the knowledge that is needed to turn abstract principles into concrete solutions. It is the link between good ideas and usable innovations. Executable know-how gives mastery over the computer through software. Our comments on this topic are highly speculative, but point the way to further research.

Another possible interpretation is that more advanced technology from source code searching and software reuse needs to be applied to code retrieval on the web. Compared to Google, the remaining search engines have had much less research and development effort into them. As well, there are many state of the art research results that are relevant, but have not yet moved out of the laboratory.

A fundamental problem in code search is the mismatch between the language that we use to formulate and specify searches, and the language in the source code. We tend to describe functionality in natural language, while source code is in a programming language [Fischer et al. 1991]. For example, software developers are looking for solutions to a problem such as "passing data from Java to CGI" or functionality that has a property such as "encapsulated within a class." In the best cases, the source code contains comments or is embedded in a web page with other text. In such instances, algorithms that are suitable for use with text, such as PageRank [Langville and Meyer 2006] and latent semantic indexing [Marcus et al. 2004] work well. However, this is too often not the case.

Prior work in code reuse has looked at techniques such as specifying the structural properties of the desired component, using a formal specification [Zaremski and Wing 1997], an outline [Balmas 1999], a fingerprint [Gil and Maman 2005], or call graph [Thummalapenta and Xie 2007]. One drawback of these approaches is they have a steep learning curve; Without an understanding of what these innovative techniques can do for them, developers rarely put in the time and effort to learn how to use them well. While users are willing to try out features, they do not always perceive or understand the availability of functionality. For example, in our study, none of our participants used the project search feature in Krugle or looked at the project results from Koders. Consequently, search terms need to be able to span the gap between plain text in natural language and source code in a programming language.

Work on this problem has proceeded in two directions: more usable precise specifications and leveraging natural language in search specifications. In CodeGenie, search specifications are written as test cases and the tool returns slices of code that satisfy the test cases [Lemos et al. 2007]. Jungloid [Mandelin et al. 2005] accepts code searches that are specified by the source and destination object type that are needed.

The approach of including natural language in program analysis seeks to make software analysis tools smarter about the concepts embedded in abbreviations and comments to make relevant sections of code easier to find [Hill et al. 2008]. Tools such as Exemplar [Grechanik et al. 2007] and CodeBroker [Ye and Fischer 2002] attempt to improve code search by indexing the text documents and comments around the source code to aid identification of functionality and behavior. Our earlier study found that mailing lists were an important source of information and often provided a good starting point for a more focused search for source code. A code search engine could easily index these documents along with code and even add links directly into the source. In information retrieval, algorithms have been developed that do more than return matches; they provide answers to queries by analyzing and inferring information [Voorhees 2003]. This is a possible direction for code retrieval on the web.

6.3 The Interaction Between Size and Search Engine

This effect is perhaps the most surprising one to us. We had suspected that there might be an interaction effect, but not this one. We expected there to be an affinity between the motivation of the search and kind of search engine, i.e. searches for components to reuse as-is would be easier using SourceForge, due to its similarity to software reuse retrieval tools. However, this was not the case.

Instead, we found an interaction effect between the size of the search target and the search engine. It appears that Google is better for finding blocks, such as a few lines of code for a widget in JSP, because these searches are particularly hard to specify within code-specific search engines. Searches tend to be specified largely using natural language, but the documents are in a programming language, and in the best cases, accompanied by comments written in a natural language. This mismatch is most acute when searching for a snippet of a program, when comments are often missing. In these situations, the search engine must rely on surrounding text to find matches. Since Google indexes web pages without prejudice, including tutorials, discussion boards, and mailing lists, it has a better chance of returning relevant matches. For similar reasons, it is easier to find subsystems, such as a regular expression library or a parsing API, in Koders and Krugle. The comments that are likely to be found near the code are close to the keywords used in the search. In other words, the terminology of the problem domain matches with the terminology of the solution domain.

7. CONCLUSION

Ultra large-scale systems, pragmatic reuse, agile software development, and open source software are powerful trends shaping the field of software engineering today. Software development is now acquiring an opportunistic motivation, where the focus is on finding and reusing existing artifacts and information. Our study is situated at the juncture of opportunistic software development and source code searching. Consequently, code retrieval on the web becomes an important element in software development, where components and examples from web pages and open source projects are used extensively. This is the first in-depth study of the phenomenon of code retrieval and its implications for the development of code search tools. Our study has two main contributions: the methodology employed to create the scenarios that were given to participants in the study and the empirical results.

The scenarios used as stimuli in the experiment were created using empirical data. In a previous study [Umarji et al. 2008], we identified common archetypes of searches carried out by software developers. Few studies that we know of have applied this degree of rigor in selecting and designing tasks for an empirical study. These scenarios have been tested and tried. They are also self-contained, which means that other researchers can also use them in their own empirical studies.

In the study, we obtained three statistically significant results. It was easier to find reference examples than components for as-is reuse. A general-purpose search engine was the most effective overall on all tasks. However, code-specific search engines were more effective when searching for subsystems, such as libraries. More informally, if you were only allowed to use one search engine, you should choose a general-purpose one, such as Google. If you could pick and choose depending on task, you should use a code-specific search engine to find subsystems and a general-purpose search engine for all other searches.

Overall, $P@10_{avg}$ ranged from 0.50 to 0.29, indicating that there is room for improvement for code search engines. Some possible directions for this work were suggested by our data. Approximately one third of the search anecdotes reported in the survey were for reference examples. These kinds of searches have been largely overlooked by the work on reuse, which focuses on reuse of components, rather than information. Another stumbling block for code search is the mismatch between the vocabulary used to specify searches and the documents returned by the search engine. Functionality, and consequently search terms, are described using natural language, whereas source code is written in a programming language. While both have a grammar that can be leveraged by a search engine, natural language can only be found in the collateral artifacts around source code, i.e., comments, documents, and discussion forums. This impedance mismatch needs to be overcome to improve code retrieval on the web.

The contributions of this study can be used to inform the creation of tools to search for source code on the web. One, we argue that designers of tools should identify what kind of search they are aiming to support. Being clear about their goals and usage scenarios will make it easier to evaluate their claims and to compare competing tools. Two, the results of this study also provide insight into the compatibility of different kinds of technologies on the problem of code retrieval on the web, in particular, program comprehension, information retrieval, and software reuse. There are likely many more technologies that are applicable and we look forward to seeing the research results.

REFERENCES

AIKEN, A. AND MURPHY, B. R. 1991. Implementing regular tree expressions. In *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture*. Springer, New York, NY, 427–447.

BALMAS, F. 1999. QBO: a query tool specially developed to explore programs. In *Proceedings of the 6th Working Conference on Reverse Engineering.* IEEE Computer Society Press, Los Alamitos, CA, 270–279.

BOEHM, B. 2006. A view of 20th and 21st century software engineering. In *Proceedings of the 28th International Conference on Software Engineering*. ACM Press, New York, NY, 12–29.

CHEN, Y., NISHIMOTO, M., AND RAMAMOORTHY, C. 1990. The C information abstraction system. *IEEE Trans. Softw. Eng. 16,* 3, 325–334.

COHEN, J. 1988. *Statistical Power Analysis for the Behavioral Sciences, 2nd Edition*. Lawrence Erlbaum Associates, Philadelphia, PA.

CRASWELL, N. AND HAWKING, D. 2004. Overview of the trec 2004 webl track. In *Proceedings of the 13th Text REtrieval Conference*. NIST, Gaithersburg, MD, 1–9.

DESHPANDE, A. AND RIEHLE, D. 2008. The total growth of open source. In *Proceedings of the Fourth IFIP International Conference on Open Source Systems (OSS2008)*. Springer.

FAUL, F., ERDFELDER, E., LANG, A.-G., AND BUCHNER, A. 2007. G*power 3: A flexible statistical power analysis program for the social, behavioral, and biomedical sciences. *Behavioral Research Methods 39*, 175–191.

FISCHER, G., HENNINGER, S., AND REDMILES, D. 1991. Cognitive tools for locating and comprehending software objects for reuse. In *Proceedings of the 13th International Conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 318–328.

FRAKES, W. B. AND KANG, K. 2005. Software reuse research: Status and future. *IEEE Trans. Softw. Eng. 31,* 7, 529–536.

GIL, J. AND MAMAN, I. 2005. Micro patterns in Java code. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*. ACM Press, New York, NY, 97–116.

GRECHANIK, M., CONROY, K. M., AND PROBST, K. 2007. Finding relevant applications for prototyping. In *Proceedings of the 4th International Workshop on Mining Software Repositories* (2007-07-02). IEEE Computer Society, Los Alamitos, CA, 12.

HARTMANN, B., DOORLEY, S., AND KLEMMER, S. R. 2006. Hacking, mashing, gluing: A study of opportunistic design. Tech. Rep. CSTR 2006-14, Department of Computer Science, Stanford University. September.

HILL, E., FRY, Z. P., BOYD, H., SRIDHARA, G., NOVIKOVA, Y., POLLOCK, L., AND VIJAY-SHANKAR, K. 2008. Amap: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *Proceedings of the 5th Working Conference on Mining Software Repositories*. ACM Press, New York, NY.

HOFFMANN, R., FOGARTY, J., AND WELD, D. S. 2007. Assieme: Finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*. ACM Press New York, NY, USA, 13–22.

HOLMES, R. AND MURPHY, G. 2005. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering.* ACM Press, New York, NY, 117–125.

HOLMES, R. AND WALKER, R. 2007. Supporting the Investigation and Planning of Pragmatic Reuse Tasks. In *Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 447–457.

HOLMES, R. AND WALKER, R. 2008. Lightweight, semi-automated enactment of pragmatic-reuse plans. In *High Confidence Software Reuse in Large Systems*. Lecture Notes in Computer Science, vol. 5030. Springer, 330–342.

HUMMEL, O., JANJIC, W., AND ATKINSON, C. 2008. Code conjurer: Pulling reusable software out of thin air. *IEEE Software*.

KAMP, P.-H. 2004. Keep in touch! *IEEE Softw. 21,* 1 (January/February), 46–47.

KAMPENES, V. B., DYBÅ, T., HANNAY, J. E., AND SJØBERG, D. I. 2007. A systematic review of effect size in software engineering experiments. *Information and Software Technology 49,* 11-12, 1073–1086.

LANGVILLE, A. AND MEYER, C. 2006. *Google's Pagerank and Beyond: The Science of Search Engine Rankings.* Princeton University Press, Princeton, NJ.

LEMOS, O., BAJRACHARYA, S., AND OSSHER, J. 2007. Codegenie: a tool for test-driven source code search. In *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, New York, NY, 917–918.

LINTON, M. 1984. Implementing relational views of programs. *ACM SIGPLAN Notices 19,* 5, 132–140.

MANDELIN, D., XU, L., BODÍK, R., AND KIMELMAN, D. 2005. Jungloid mining: helping to navigate the api jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming language design and implementation*. ACM New York, NY, USA, 48–61.

MANNING, C. D., RAGHAVAN, P., AND SCHÜTZE, H. 2008. *Introduction to Informatio Retrieval.* Cambridge University Press.

MARCUS, A., SERGEYEV, A., RAJLICH, V., AND MALETIC, J. 2004. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 214–223.

MILES, M. B. AND HUBERMAN, A. M. 1994. *Qualitative data analysis*. Sage Publications, Thousand Oaks, CA.

MILI, A., MILI, R., AND MITTERMEIR, R. 1998. A survey of software reuse libraries. *Annals of Software Engineering 5*, 349–414.

MURPHY, G. AND NOTKIN, D. 1996. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology 5,* 3, 262–292.

NOBLE, J. AND BIDDLE, R. 2002. Notes on Postmodern Programming. In *Proceedings of the Onward Track at OOPSLA*. Vol. 2. ACM Press, New York, NY, 49–71.

NORRIS, J. S. 2004. Mission-critical development with open source software: Lessons learned. *IEEE Softw. 21,* 1, 42–49.

NORTHROP, L., FEILER, P., GABRIEL, R. P., GOODENOUGH, J., LINGER, R., LONGSTAFF, T., KAZMAN, R., KLEIN, M., , SCHMIDT, D., SULLIVAN, K., AND WALLNAU, K. 2006. Ultra-Large-Scale Systems: The Software Challenge of the Future. Tech. rep., Software Engineering Institute, Carnegie Mellon University.

PAUL, S. AND PRAKASH, A. 1994. A framework for source code search using program patterns. *IEEE Trans. Softw. Eng. 20,* 6, 463–475.

PRIETO-DIAZ, R. 1991. Implementing faceted classification for software reuse. *Communications of the ACM 34,* 5, 88–97.

RAVICHANDRAN, T. AND ROTHENBERGER, M. 2003. Software reuse strategies and component markets. *Commun. ACM 46,* 8, 109–114.

SAHAVECHAPHAN, N. AND CLAYPOOL, K. T. 2006. Xsnippet: mining for sample code. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2006-12-06). ACM Press, New York, NY, 413–430.

SIM, S. E., CLARKE, C. L. A., AND HOLT, R. C. 1998. Archetypal source code searches: A survey of software developers and maintainers. In *Proceedings of the 6th International Workshop on Program Comprehension*. IEEE Computer Society, Los Alamitos, CA, 180.

SINGER, J. AND LETHBRIDGE, T. 1997. What's so great about 'grep'? implications for program comprehension tools. Tech. rep., National Research Council, Canada.

SPINELLIS, D. AND SZYPERSKI, C. 2004. Guest editors' introduction: How is open source affecting software development? *IEEE Software 21,* 1, 28–33.

STRAUSS, A. AND CORBIN, J. 1990. *Basics of Qualitative Research: Grounded Theory Procedures and Technique*. Sage Publications, Thousand Oaks, CA.

THOMPSON, K. 1968. Programming techniques: Regular expression search algorithm. *Communications of the ACM 11,* 6, 419–422.

THUMMALAPENTA, S. AND XIE, T. 2007. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering* (2008-02-07). ACM Press, New York, NY, 204–213.

UMARJI, M., SIM, S. E., AND LOPES, C. V. 2008. Archetypal internet-scale source code searching. In *OSS*, B. Russo, Ed. Springer, New York, NY, 7.

VOORHEES, E. 2003. Overview of the TREC 2003 Question Answering Track. In *Proceedings of the 12th Text REtrieval Conference*. Vol. 142. NIST, Gaithersburg, MD.

YE, Y. AND FISCHER, G. 2002. Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of the 24th Interntional Conference on Software Engineering* (2006-02-13). ACM Press, New York, NY, 513–523.

ZAREMSKI, A. AND WING, J. 1997. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology 6,* 4, 333–369.

ZIMMERMANN, T., WEISSGERBER, P., DIEHL, S., AND ZELLER, A. 2005. Mining Version Histories to Guide Software Changes. *IEEE Trans. Softw. Eng. 31,* 6 (June), 429–445.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library by visiting the following URL: `http://www.acm.org/pubs/citations/journals/jn/2009-0-0/p1-`.