Searching, Selecting, and Synthesizing Source Code Components

Collin McMillan

Prairie Village, Kansas

Master of Science, College of William and Mary, 2009
Bachelor of Science, University of Tulsa, 2007

A Dissertation presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

The College of William and Mary
August, 2012

ABSTRACT PAGE

As programmers develop software, they instinctively sense that source code exists that could be reused if found – many programming tasks are common to many software projects across different domains. Oftentimes, a programmer will attempt to create new software from this existing source code, such as third-party libraries or code from online repositories. Unfortunately, several major challenges make it difficult to locate the relevant source code and to reuse it. First, there is a fundamental mismatch between the high-level intent reflected in the descriptions of source code, and the low-level implementation details. This mismatch is known as the *concept assignment problem*, and refers to the frequent case when the keywords from comments or identifiers in code do not match the features implemented in the code. Second, even if relevant source code is found, programmers must invest significant intellectual effort into understanding how to reuse the different functions, classes, or other components present in the source code. These components may be specific to a particular application, and difficult to reuse.

One key source of information that programmers use to understand source code is the set of relationships among the source code components. These relationships are typically structural data, such as function calls or class instantiations. This structural data has been repeatedly suggested as an alternative to textual analysis for search and reuse, however as yet no comprehensive strategy exists for locating relevant and reusable source code. In my research program, I harness this structural data in a unified approach to creating and evolving software from existing components. For locating relevant source code, I present a search engine for finding applications based on the underlying Application Programming Interface (API) calls, and a technique for finding chains of relevant function invocations from repositories of millions of lines of code. Next, for reusing source code, I introduce a system to facilitate building software prototypes from existing packages, and an approach to detecting similar software applications.

Table of Contents

ACKNOWLEDGMENTS

This disseration represents over five years of research during my time in Virginia. I have had been blessed with the honor to work with many fantastic people, and it is my pleasure to express my deep and sincere gratitude. Without them, this dissertation could never have been completed.

- Denys Poshyvanyk, who as my advisor taught me throughout my studies. His vision and patience made this dissertation possible. It was an honor to be one of his first students.
- Mark Grechanik, a research collaborator and mentor, who has spent countless hours of his own time giving me invaluable advice on many subjects.
- Jane Cleland-Huang, Peter Kemper, Xipeng Shen, Andrian Marcus, and John Hale, for their committee service and recommendations. I deeply respect their help and collaboration.
- My friends, brothers, uncle, and parents who encouraged me during both the highs and lows of my work. I will never be able to repay their endless support.
- Above all, my wife Margaret, whose dedication and inspiration cannot be expressed in words.

The work described in this dissertation was partially funded by the grant CCF-0916260 from the U.S. National Science Foundation. Any opinions, findings, and conclusions expressed herein are those of the author and do not necessarily reflect those of the sponsor.

List of Tables

List of Figures

Searching, Selecting, and Synthesizing Source Code Components

Chapter 1

Introduction

As programmers develop software, they instinctively sense that source code exists that could be reused if found – many programming tasks are common to many software projects across different domains. Oftentimes, a programmer will attempt to create new software from this existing source code, such as third-party libraries or code from online repositories. Unfortunately, several major challenges make it difficult to locate the relevant source code and to reuse it. First, there is a fundamental mismatch between the high-level intent reflected in the descriptions of source code, and the low-level implementation details. This mismatch is known as the *concept assignment problem*, and refers to the frequent case when the keywords from comments or identifiers in code do not match the features implemented in the code [9]. Second, even if relevant source code is found, programmers must invest significant intellectual effort into understanding how to reuse the different functions, classes, or other components present in the source code. These components may be specific to a particular application, and difficult to reuse.

One key source of information that programmers use to understand source code is the set of relationships among the source code components. These relationships are typically structural data, such as function calls or class instantiations. This structural data has been repeatedly suggested as an alternative to textual analysis for search and reuse, however as yet no comprehensive strat-

egy exists for locating relevant and reusable source code. In my research program, I harness this structural data in a unified approach to creating and evolving software from existing components. For locating relevant source code, I present a search engine for finding applications based on the underlying Application Programming Interface (API) calls, and a technique for finding chains of relevant function invocations from repositories of millions of lines of code. Next, for reusing source code, I introduce a system to facilitate building software prototypes from existing packages, and an approach to detecting similar software applications.

1.1 A Search Engine for Finding Highly-Relevant Applications

Software contains functional abstractions, in the form of API calls, that support the implementation of the *features* of that software, and programmers commonly build these API calls into their applications. Software with the feature of playing music, for example, is likely to contain API calls from third-party sound libraries. However, API calls are an untapped resource for source code search; a majority of source code search engines treat code as plain text, where all words have unknown semantics. Efforts to introduce structural information such as API calls into source code search engines have remained largely theoretical, being implemented only on small codebases and not evaluated by developers in statistically-significant case studies.

In contrast, we designed and implemented Exemplar, a search engine for software applications [34, 68]. Exemplar addresses an instance of the concept assignment problem because it matches keywords in queries to keywords in the documentation of the API calls used in the applications. For example, a query containing keywords related to music will match API calls that implement various multimedia tasks, and Exemplar will return applications which use those calls, regardless of

whether those applications actually contain keywords relevant to the query. We built Exemplar with a repository of 8,310 Java applications, and compared it to a state-of-the-art search engine provided by Sourceforge in a cross-validation design case study with 39 professional developers. Our results demonstrate how API calls can be used to improve source code search in large repositories.

1.2 Detecting Similar Software Applications

Retrieving similar or related web pages is a popular feature of search engines. After users submit search queries, the engine displays links to relevant pages labeled "Similar." These pages are ranked as similar based on different factors, including text content, popularity scores, and the links' position and size [31]. Existing techniques for detecting similar software applications, for use in source code search engines, are based solely on the textual content of the code. In contrast, we created an approach to automatically detect *Closely reLated ApplicatioNs* (CLAN) [67]. Our approach works by comparing the API calls used in the applications. By comparing the applications based on their API usage, we are able to significantly outperform a state-of-the-art approach that uses only text content.

To navigate the large repositories, as well as the results from search engines, it is useful to group software systems into categories which define the broad functionality provided by the software. This categorization helps programmers reuse source code by showing similar software, which may be used as a reference or alternative. Therefore, we developed a technique using API calls to categorize software [71]. The advantage to our approach is that it does not rely on textual information from source code, which may not be available due to privacy concerns or language barriers. This work is a key step towards helping developers to reuse source code located by search engines.

1.3 Locating Relevant Functions and Their Usages in Millions of Lines of Code

The functional abstractions in software are not limited to API calls. Generally speaking, functional abstractions are the basic units of functionality in source code (e.g., known as functions, methods, subprocedures, in different languages). More advanced features in software are accomplished by combining these functions into chains of function invocations. For example, consider the feature of recording microphone audio and saving it to a file. This feature is unlikely to be implemented by a single function. Instead, some functions may access the microphone, some functions may process the audio, and others write the data to a file; these functions will then be connected via different function calls. When searching for source code, programmers need to see this chain of function invocations to understand how a feature is implemented. However, current source code search engines focus on locating individual functions, statements, or arbitrary fragments of code.

My work addresses this shortcoming with a code search system called Portfolio that retrieves and visualizes relevant chains of function invocations from two open-source repositories of over 710 million total lines of code [69]. Portfolio works by computing a textual similarity value for functions to a query, and then propagating this value to other functions which are connected via the function call graph using a technique called spreading activation. In this way, we address the concept assignment problem, in that we locate functions which are relevant to a task, even if those functions do not contain any keywords. Also, we reduce the manual effort required by programmers to understand the code, because we show programmers a chain of function invocations that implements the task, rather than only individual functions. We evaluated Portfolio in a case study with 49 professional programmers, and found statistically-significant improvement over two commercial-grade engines.

1.4 Recommending Source Code for Rapid Software Prototyping

Current source code search engines return code relevant to a single feature description that the programmer needs to implement. However, when programmers reuse code, they often implement software with multiple, interacting features. Even if a developer locates relevant code for each of the feature descriptions he or she needs to build, that code may be incompatible and require substantial modification before code for multiple features can be integrated. In this situation, the effort required for a developer to understand and integrate the returned source code can drastically reduce the benefits of reuse.

We introduce a recommender system for source code in the context of rapid software prototyping [70]. During prototyping, programmers iteratively propose, review, and demonstrate the features of a software product. Our system helps programmers in two ways. First, we expand the list of feature descriptions to be implemented by mining repositories for similar sets of feature descriptions. We use a k-Nearest-Neighbor algorithm to cluster the feature descriptions which our mining tool detects are frequently implemented in the same projects. Second, we locate source code that implements multiple features that the programmer specifies. We use a combination of PageRank, set coverage, and Coupling Between Objects to maximize the coverage of desired features in the recommended source code, while minimizing the external coupling of that source code. Programmers using the recommendations from our approach must perform less manual work than with code from other approaches because our recommendations include multiple features selected by the programmer.

Chapter 2

A Search Engine For Finding Highly Relevant Applications

2.1 Introduction

Programmers face many challenges when attempting to locate source code to reuse [102]. One key problem of finding relevant code is the mismatch between the high-level intent reflected in the descriptions of software and low-level implementation details. This problem is known as the *concept assignment problem* [9]. Search engines have been developed to address this problem by matching keywords in queries to words in the descriptions of applications, comments in their source code, and the names of program variables and types. These applications come from repositories which may contain thousands of software projects. Unfortunately, many repositories are polluted with poorly functioning projects [42]; a match between a keyword from the query with a word in the description or in the source code of an application does not guarantee that this application is relevant to the query.

Many source code search engines return snippets of code that are relevant to user queries. Programmers typically need to overcome a high cognitive distance [52] to understand how to use these

code snippets. Moreover, many of these code fragments are likely to appear very similar [29]. If code fragments are retrieved in the contexts of executable applications, it makes it easier for programmers to understand how to reuse these code fragments.

Existing code search engines (e.g., Google Code Search, SourceForge) often treat code as plain text where all words have unknown semantics. However, applications contain functional abstractions in a form of API calls whose semantics are well-defined. The idea of using API calls to improve code search was proposed and implemented elsewhere [33, 15]; however, it was not evaluated over a large codebase using a standard information retrieval methodology [66, pages 151-153].

We created an application search system called *Exemplar (EXEcutable exaMPLes ARchive)* as part of our *Searching, Selecting, and Synthesizing* (S^3) architecture [82]. Exemplar helps users find highly relevant executable applications for reuse. Exemplar combines three different sources of information about applications in order to locate relevant software: the textual descriptions of applications, the API calls used inside each application, and the dataflow among those API calls. We evaluated the contributions by these different types of information in two separate case studies. First, in Section 2.6, we compared Exemplar (in two configurations) to SourceForge. We analyzed the results of that study in Section 2.7 and created a new version of Exemplar. We evaluated our updates to Exemplar in Section 2.8. Our key finding is that our search engine's results improved when considering the API calls in applications instead of only the applications' descriptions. We have made Exemplar and the results of our case studies available to the public[1].

[1] `http://www.xemplar.org` (verified 03/28/2011)

2.2 Exemplar Approach

2.2.1 The Problem

A direct approach for finding highly relevant applications is to search through the descriptions and source code of applications to match keywords from queries to the names of program variables and types. This approach assumes that programmers choose meaningful names when creating source code, which is often not the case [3].

This problem is partially addressed by programmers who create meaningful descriptions of the applications in software repositories. However, state-of-the-art search engines use exact matches between the keywords from queries, the words in the descriptions, and the source code of applications. Unfortunately, it is difficult for users to guess exact keywords because "no single word can be chosen to describe a programming concept in the best way" [28]. The vocabulary chosen by a programmer is also related to the concept assignment problem because the terms in the high-level descriptions of applications may not match terms from the low-level implementation (e.g., identifier names and comments).

2.2.2 Key Ideas

Suppose that a programmer needs to encrypt and compress data. A programmer will naturally turn to a search engine such as SourceForge[2] and enter keywords such as `encrypt` and `compress`. The programmer then looks at the source code of the programs returned by these search engines to check to see if some API calls are used to encrypt and compress data. The presence of these API calls is a good starting point for deciding whether to check these applications further.

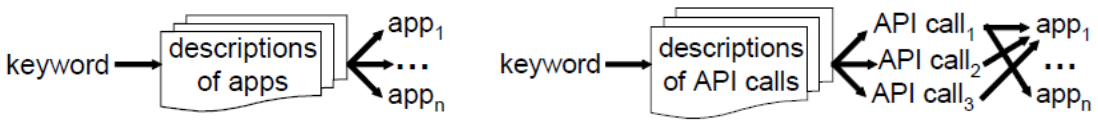
[2] `http://sourceforge.net/` (verified 03/28/2011)

What we seek is to augment standard code search to include help documentations of widely used libraries, such as the standard *Java Development Kit (JDK)*[3]. Existing engines allow users to search for specific API calls, but knowing in advance what calls to search for is hard. Our idea is to match keywords from queries to words in help documentation for API calls. These help documents are descriptions of the functionality of API calls as well as the usage of those calls. In Exemplar, we extract the help documents that come in the form of *JavaDocs*. Programmers trust these documents because the documents come from known and respected vendors, were written by different people, reviewed multiple times, and have been used by other programmers who report their experience at different forums [25].

We also observe that relations between concepts entered in queries are often reflected as dataflow links between API calls that implement these concepts in the program code. This observation is closely related to the concept of the *software reflexion models* formulated by Murphy, Notkin, and Sullivan. In these models, relations between elements of high-level models (e.g., processing elements of software architectures) are preserved in their implementations in source code [77][76]. For example, if the user enters keywords `secure` and `send`, and the corresponding API calls `encrypt` and `email` are connected via some dataflow, then an application with these connected API calls are more relevant to the query than applications where these calls are not connected.

Consider two API calls `string encrypt()` and `void email(string)`. After the call `encrypt` is invoked, it returns a string that is stored in some variable. At some later point a call to the function `email` is made and the variable is passed as the input parameter. In this case these functions are connected using a dataflow link which reflects the implicit logical connection between

[3] `http://www.oracle.com/technetwork/java/javase/downloads/index.html` (verified 03/28/2011)



(a) Standard search engines. (b) Exemplar search engine.

Figure 2.1: Illustrations of the processes for standard and Exemplar search engines.

keywords in queries. Specifically, the data should be encrypted and then sent to some destination.

2.2.3 Motivating Example

Exemplar returns applications that implement the tasks described in by the keywords in user queries. Consider the following task: find an application for sharing, viewing, and exploring large data sets that are encoded using MIME, and the data can be stored using key value pairs. Using the following keywords `MIME`, `type`, `data`, an unlikely candidate application called BIOLAP is retrieved using Exemplar with a high ranking score. The description of this application matches only the keyword `data`, and yet this application made it to the top ten of the list.

BIOLAP uses the class `MimeType`, specifically its method `getParameterMap`, because it deals with MIME-encoded data. The descriptions of this class and this method contain the desired keywords, and these implementation details are highly-relevant to the given task. BIOLAP does not show on the top 300 list of retrieved applications when the search is performed with the SourceForge search engine.

2.2.4 Fundamentals of Exemplar

Consider the process for standard search engines (e.g., Sourceforge, Google code search[4], Krugle[5]) shown in Figure 2.1(a). A keyword from the query is matched against words in the descriptions

[4] `http://www.google.com/codesearch` (verified 03/28/2011)
[5] `http://opensearch.krugle.org` (verified 03/28/2011)

of the applications in some repository (Sourceforge) or words in the entire corpus of source code (Google Code Search, Krugle). When a match is found, applications $\texttt{app}_1$ to $\texttt{app}_n$ are returned.

Consider the process for Exemplar shown in Figure 2.1(b). Keywords from the query are matched against the descriptions of different documents that describe API calls of widely used software packages. When a match is found, the names of the API calls $\texttt{call}_1$ to $\texttt{call}_k$ are returned. These names are matched against the names of the functions invoked in these applications. When a match is found, applications $\texttt{app}_1$ to $\texttt{app}_n$ are returned.

In contrast to the keyword matching functionality of standard search engines, Exemplar matches keywords with the descriptions of the various API calls in help documents. Since a typical application invokes many API calls, the help documents associated with these API calls are usually written by different people who use different vocabularies. The richness of these vocabularies makes it more likely to find matches, and produce API calls $\texttt{API call}_1$ to $\texttt{API call}_k$. If some help document does not contain a desired match, some other document may yield a match. This is how we address the vocabulary problem [28].

As it is shown in Figure 2.1(b), API calls $\texttt{API call}_1$, $\texttt{API call}_2$, and $\texttt{API call}_3$ are invoked in the $\texttt{app}_1$. It is less probable that the search engine fails to find matches in help documents for all three API calls, and therefore the application $\texttt{app}_1$ will be retrieved from the repository.

Searching help documents produces additional benefits. API calls from help documents (that match query keywords) are linked to locations in the project source code where these API calls are used thereby allowing programmers to navigate directly to these locations and see how high-level concepts from queries are implemented in the source code. Doing so solves an instance of the concept location problem [61].

2.3 Ranking Schemes

2.3.1 Components of Ranking

There are three components that compute different scores in the Exemplar ranking mechanism: a component that computes a score based on word occurrences in project descriptions (WOS), a component that computes a score based on the relevant API calls (RAS), and a component that computes a score based on dataflow connections between these calls (DCS). The total ranking score is the weighted sum of these three ranking scores.

We designed each ranking component to produce results from different perspectives (e.g., application descriptions, API calls, and dataflows among the API calls). The following three sections describe the components. Section 2.4 discusses the implentation of the components and includes important technical limitations that we considered when building Exemplar. We examine how WOS, RAS, and DCS each contribute to the results given by Exemplar in Section 2.7. Section 2.7 also covers the implications of our technical considerations.

2.3.2 WOS Ranking Scheme

The WOS component uses the *Vector Space Model* (VSM), which is a ranking function used by search engines to rank matching documents according to their relevance to a given search query. VSM is a bag-of-words retrieval technique that ranks a set of documents based on the terms appearing in each document as well as the query. Each document is modeled as a vector of the terms it contains. The weights of those terms in each document are calculated in accordance to the *Term Frequency/Inverse Document Frequency* (TF/IDF). Using TF/IDF, the weight for a term is calculated as $tf = \frac{n}{\sum_k n_k}$ where n is the number of occurrences of the term in the document, and $\sum_k n_k$ is the sum of

the number of occurences of the term in all documents. Then the similarities among the documents are calculated using the cosine distance between each pair of documents $cos(\theta) = \frac{d_1 \cdot d_2}{\|d_1\|\|d_2\|}$ where d_1 and d_2 are document vectors.

2.3.3 RAS Ranking Scheme

The documents in our approach are the different documents that describe each API call (e.g., each JavaDoc). The collection of API documents is defined as $D_{API} = (D^1_{API}, D^2_{API}, \ldots, D^k_{API})$. A corpus is created from D_{API} and represented as the term-by-document $m \times k$ matrix M, where m is the number of terms and k is the number of API documents in the collection. A generic entry $a[i, j]$ in this matrix denotes a measure of the weight of the i^{th} term in the j^{th} API document [94].

API calls that are relevant to the user query are obtained by ranking documents, D_{API} that describe these calls as relevant to the query Q. This relevance is computed as a conceptual similarity, C, (i.e., the length-normalized inner product) between the user query, Q, and each API document, D_{API}. As a result the set of triples $\langle A, C, n \rangle$ is returned, where A is the API call, n is the number of occurrences of this API call in the application with the conceptual similarity, C, of the API call documentation to query terms.

The API call-based ranking score for the application, j, is computed as $S^j_{ras} = \frac{\sum_{i=1}^{p} n^j_i \cdot C^j_i}{|A|^j}$, where $|A|^j$ is the total number of API calls in the application j, and p is the number of API calls retrieved for the query.

2.3.4 DCS Ranking Scheme

To improve the precision of ranking we derive the structure of connections between API calls and use this structure as an important component in computing rankings. The standard syntax for invok-

ing an API call is `t var=o.callname`$(p_1,\ldots,p_n)$. The structural relations between API calls reflect compositional properties between these calls. Specifically, it means that API calls access and manipulate data at the same memory locations.

There are four types of dependencies between API calls: input, output, true, and anti-dependence [75, page 268]. True dependence occurs when the API call `f` write a memory location that the API call `g` later reads (e.g., `var=f`$(\ldots)$; $\ldots$; `g`$(\texttt{var},\ldots)$;). Anti-dependence occurs when the API call `f` reads a memory location that the API call `g` later writes (e.g., `f`$(\texttt{var},\ldots)$, $\ldots$; `var=g`$(\ldots)$;). Output dependence occurs when the API calls `f` and `g` write the same memory location. Finally, input dependence occurs when the API calls `f` and `g` read the same memory location.

Consider an all-connected graph (i.e., a clique) where nodes are API calls and the edges represent dependencies among these calls for one application. The absence of an edge means that there is no dependency between two API calls. Let the total number of connections among n retrieved API calls be less or equal to $n(n-1)$. Let a connection between two distinct API calls in the application be defined as *Link*; we assign some weight w to this Link based on the strength of the dataflow or control flow dependency type. The ranking is normalized to be between 0 and 1.

The API call connectivity-based ranking score for the application, j, is computed as $S^j_{dcs} = \frac{\sum_{i=1}^{n(n-1)} w_i^j}{n(n-1)}$, where w_i is the weight to each type of flow dependency for the given link *Link*, such that $1 > w_i^{true} > w_i^{anti} > w_i^{output} > w_i^{input} > 0$. The intuition behind using this order is that these dependencies contribute differently to ranking heuristics. Specifically, using the values of the same variable in two API calls introduces a weaker link as compared to the true dependency where one API call produces a value that is used in some other API call.

2.3.5 Integrated Scheme

The final ranking score is computed as $S = \lambda_{wos}S_{wos} + \lambda_{ras}S_{ras} + \lambda_{dcs}S_{dcs}$, where λ is the interpolation weight for each type of the score. These weights are determined independently of queries unlike the scores, which are query-dependent. Adjusting these weights enables experimentation with how underlying structural and textual information in application affects resulting ranking scores. The formula for S remains the same throughout this paper, and all three weights were equal during the case study in Section2.5. We explore alterations to Exemplar, including λ, based on the case study results in Section 2.7.

2.4 Implementation Details

Figure 2.2 shows the architecture of Exemplar. In this section we step through Figure 2.2 and describe some technical details behind Exemplar.

Two crawlers, *Application Extractor* and *API Call Extractor* populate Exemplar with data from SourceForge. We currently have run the crawlers on SourceForge and obtained more than 8,000 Java projects containing 414,357 files[6]. The Application Extractor downloads the applications and extracts the descriptions and source code of those applications (the Application Metadata `(1)`). The API Call Extractor crawls the source code from the applications for the API calls that theyuse, the descriptions of the API calls, and the dataflow amoung those calls (the API Call Metadata `(2)`). The API Call Extractor ran with 65 threads for over 50 hours on 30 computers: three machines have two dual-core 3.8Ghz EM64T Xeon processors with 8Gb RAM, two have four 3.0Ghz EM64T Xeon CPUs with 32Gb RAM, and the rest have one 2.83Ghz quad-core CPU and 2Gb RAM. The API

[6]We ran the crawlers in August 2009.

Call Extractor found nearly twelve million API invocations from the JDK 1.5 in the applications. It also processes the API calls for their descriptions, which in our case are the JavaDocs for those API calls.

Our approach relies on the tool PMD [7] for computing approximate dataflow links, which are based on the patterns described in Section 2.3.4. PMD extracts data from individual Java source files, so we are only able to locate dataflow links among the API calls as they are used in any one file. We follow the variables visible in each scope (e.g., global variables plus those declared in methods). We then look at each API call in the scope of those variables. We collect the input parameters and output of those API calls. We then analyze this input and output for dataflow. For example, if the output of one API call is stored in a variable which is then used as input to another API call, then there is dataflow between those API calls. Note that our technique is an approximation and can produce both false positive and false negatives. Determining the effects of this approximation on the quality of Exemplar's results is an area of future work.

The *Retrieval Engine* locates applications in two ways `(3)`. First, the input to the Retrieval Engine is the user query, and the engine matches keywords in this query *(5)* to keywords in the descriptions of applications. Second, the Retrieval Engine finds descriptions of API calls which match keywords [8]. The Retrieval Engine then locates applications which use those API calls. The engine outputs a list of Retrieved Applications `(6)`.

The *Ranking Engine* uses the three ranking schemes from Section 2.3 (WOS, RAS, and DCS) to sort the list of retrieved applications `(7)`. The Ranking Engine depends on three sources of information: descriptions of applications, the API calls used by each application, and the dataflow

[7] `http://pmd.sourceforge.net/` (verified 03/28/2011)

[8] Exemplar limits the number of relevant API calls it retrieves for each query to 200. This limit was necessary due to performance constraints. See Section 2.7.4.

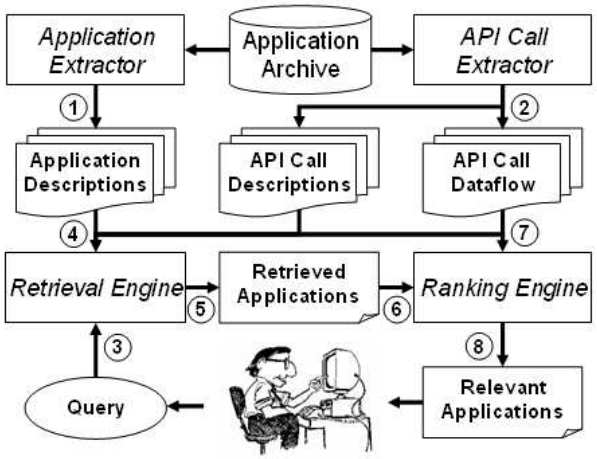


Figure 2.2: Exemplar architecture.

among those API calls `(4)`. The Ranking Engine uses Lucene[9], which is based on VSM, to implement WOS. The combination of the ranking schemes (see Section 2.3.5) determines the relevancy of the applications. The Relevant Applications are then presented to the user `(8)`.

2.5 Case Study Design

Typically, search engines are evaluated using manual relevance judgments by experts [66, pages 151-153]. To determine how effective Exemplar is, we conducted a case study with 39 participants who are professional programmers. We gave a list of tasks described in English. Our goal is to evaluate how well these participants can find applications that match given tasks using three different search engines: Sourceforge (SF) and Exemplar with (EWD) and without (END) dataflow links as part of the ranking mechanism. We chose to compare Exemplar with Sourceforge because the latter has a popular search engine with the largest open source Java project repository, and Exemplar is populated with Java projects from this repository.

[9] `http://lucene.apache.org` (verified 03/28/2011)

Experiment	Group	Search Engine	Task Set
1	G1	EWD	T1
	G2	SF	T2
	G3	END	T3
2	G1	END	T2
	G2	EWD	T3
	G3	SF	T1
3	G1	SF	T3
	G2	END	T1
	G3	EWD	T2

Table 2.1: Plan for the case study of Exemplar and Sourceforge.

2.5.1 Methodology

We used a cross validation study design in a cohort of 39 participants who were randomly divided into three groups. We performed three separate experiments during the study. In each experiment, each group was given a different search engine (i.e., SF, EWD, or END) as shown in Table 2.1. Then, in the experiments, each group would be asked to use a different search engine than that group had used before. The participants would use the assigned engine to find applications for given tasks. Each group used a different set of tasks in each experiment. Thus each participant used each search engine on different tasks in this case study. Before the study we gave a one-hour tutorial on using these search engines to find applications for tasks.

Each experiment consisted of three steps. First, participants translated tasks into a sequence of keywords that described key concepts of applications that they needed to find. Then, participants entered these keywords as queries into the search engines (the order of these keywords does not matter) and obtained lists of applications that were ranked in descending order.

The next step was to examine the returned applications and to determine if they matched the tasks. Each participant accomplished this step by him or herself, assigning a confidence level, C, to the examined applications using a four-level Likert scale. We asked participants to examine only

top ten applications that resulted from their searches. We evaluated only the top ten results because users of search engines rarely look beyond the tenth result [32] and because other source code search engines have been evaluated using the same number of results [40].

The guidelines for assigning confidence levels are the following.

1. Completely irrelevant - there is absolutely nothing that the participant can use from this retrieved project, nothing in it is related to your keywords.
2. Mostly irrelevant - only few remotely relevant code snippets or API calls are located in the project.
3. Mostly relevant - a somewhat large number of relevant code snippets or API calls in the project.
4. Highly relevant - the participant is confident that code snippets or API calls in the project can be reused.

Twenty-six participants are Accenture employees who work on consulting engagements as professional Java programmers for different client companies. Remaining 13 participants are graduate students from the University of Illinois at Chicago who have at least six months of Java experience. Accenture participants have different backgrounds, experience, and belong to different groups of the total Accenture workforce of approximately 180,000 employees. Out of 39 participants, 17 had programming experience with Java ranging from one to three years, and 22 participants reported more than three years of experience writing programs in Java. Eleven participants reported prior experience with Sourceforge (which is used in this case study), 18 participants reported prior experience with other search engines, and 11 said that they never used code search engines. Twenty six participants have bachelor degrees and thirteen have master degrees in different technical disciplines.

2.5.2 Precision

Two main measures for evaluating the effectiveness of retrieval are precision and recall [114, page 188-191]. The precision is calculated as $P_r = \frac{\texttt{relevant}}{\texttt{retrieved}}$, where `relevant` is the number of retrieved applications that are relevant and `retrieved` is the total number of applications retrieved. The precision of a ranking method is the fraction of the top r ranked documents that are relevant to the query, where $r = 10$ in this case study. Relevant applications are counted only if they are ranked with the confidence levels `4` or `3`. The precision metrics reflects the accuracy of the search. Since we limit the investigation of the retrieved applications to top ten, the recall is not measured in this study.

2.5.3 Discounted Cumulative Gain

Discounted Cumulative Gain (DCG) is a metric for analyzing the effectiveness of search engine results [1]. The intuition behind DCG is that search engines should not only return relevant results, but should rank those results by relevancy. Therefore, DCG rewards search engines for ranking relevant results above irrelevant ones. We calculate the DCG for the top 10 results from each engine because we collect confidence values for these results. We compute DCG according to this formula: $G = C_1 + \sum_{i=2}^{10} \frac{C_i}{log_2\, i}$, where C_1 is the confidence value of the result in the first position and C_i is the confidence value of the result in the ith position. We normalize the DCG using the following formula: $NG = \frac{G}{iG}$, where iG is the ideal DCG in the case when the confidence value for the first ten results is always 4 (indicating that all ten results are highly-relevant). We refer to normalized DCG as NG in the remainder of this paper.

2.5.4 Hypotheses

We introduce the following null and alternative hypotheses to evaluate how close the means are for the confidence levels (*C*s) and precisions (*P*s) for control and treatment groups. Unless we specify otherwise, participants of the treatment group use either END or EWD, and participants of the control group use SF. We seek to evaluate the following hypotheses at a 0.05 level of significance.

H_{0-null} The primary null hypothesis is that there is no difference in the values of confidence level and precision per task between participants who use SF, EWD, and END.

H_{0-alt} An alternative hypothesis to H_{0-null} is that there is statistically significant difference in the values of confidence level and precision between participants who use SF, EWD, and END.

Once we test the null hypothesis H_{0-null}, we are interested in the directionality of means, μ, of the results of control and treatment groups. We are interested to compare the effectiveness of EWD versus the END and SF with respect to the values of *C*, *P*, and *NG*.

H_1 **(C of EWD versus SF)** The effective null hypothesis is that $\mu_C^{EWD} = \mu_C^{SF}$, while the true null hypothesis is that $\mu_C^{EWD} \leq \mu_C^{SF}$. Conversely, the alternative hypothesis is $\mu_C^{EWD} > \mu_C^{SF}$.

H_2 **(P of EWD versus SF)** The effective null hypothesis is that $\mu_P^{EWD} = \mu_P^{SF}$, while the true null hypothesis is that $\mu_P^{EWD} \leq \mu_P^{SF}$. Conversely, the alternative hypothesis is $\mu_P^{EWD} > \mu_P^{SF}$.

H_3 **(NG of EWD versus SF)** The effective null hypothesis is that $\mu_{NG}^{EWD} = \mu_{NG}^{SF}$, while the true null hypothesis is that $\mu_{NG}^{EWD} \leq \mu_{NG}^{SF}$. Conversely, the alternative hypothesis is $\mu_{NG}^{EWD} > \mu_{NG}^{SF}$.

H_4 **(C of EWD versus END)** The effective null hypothesis is that $\mu_C^{EWD} = \mu_C^{END}$, while the true null hypothesis is that $\mu_C^{EWD} \leq \mu_C^{END}$. Conversely, the alternative is $\mu_C^{EWD} > \mu_C^{END}$.

H_5 **(P of EWD versus END)** The effective null hypothesis is that $\mu_P^{EWD} = \mu_P^{END}$, while the true null hypothesis is that $\mu_P^{EWD} \leq \mu_P^{END}$. Conversely, the alternative is $\mu_P^{EWD} > \mu_P^{END}$.

H_6 **(NG of EWD versus END)** The effective null hypothesis is that $\mu_{NG}^{EWD} = \mu_{NG}^{END}$, while the true null hypothesis is that $\mu_{NG}^{EWD} \leq \mu_{NG}^{END}$. Conversely, the alternative is $\mu_{NG}^{EWD} > \mu_{NG}^{END}$.

H_7 **(C of END versus SF)** The effective null hypothesis is that $\mu_{C}^{END} = \mu_{C}^{SF}$, while the true null hypothesis is that $\mu_{C}^{END} \leq \mu_{C}^{SF}$. Conversely, the alternative hypothesis is $\mu_{C}^{END} > \mu_{C}^{SF}$.

H_8 **(P of END versus SF)** The effective null hypothesis is that $\mu_{P}^{END} = \mu_{P}^{SF}$, while the true null hypothesis is that $\mu_{P}^{END} \leq \mu_{P}^{SF}$. Conversely, the alternative hypothesis is $\mu_{P}^{END} > \mu_{P}^{SF}$.

H_9 **(NG of END versus SF)** The effective null hypothesis is that $\mu_{NG}^{END} = \mu_{NG}^{SF}$, while the true null hypothesis is that $\mu_{NG}^{END} \leq \mu_{NG}^{SF}$. Conversely, the alternative hypothesis is $\mu_{NG}^{END} > \mu_{NG}^{SF}$.

The rationale behind the alternative hypotheses to H_1, H_2, and H_3 is that Exemplar allows users to quickly understand how keywords in queries are related to implementations using API calls in retrieved applications. The alternative hypotheses to H_4, H_5, H_6 are motivated by the fact that if users see dataflow connections between API calls, they can make better decisions about how closely retrieved applications match given tasks. Finally, having the alternative hypotheses to H_7, H_8, and H_9 ensures that Exemplar without dataflow links still allows users to quickly understand how keywords in queries are related to implementations using API calls in retrieved applications.

2.5.5 Task Design

We designed 26 tasks that participants work on during experiments in a way that these tasks belong to domains that are easy to understand, and they have similar complexity. The following are two example tasks; all others may be downloaded from the Exemplar about page[10].

1. "Develop a universal sound and voice system that allows users to talk, record audio, and play MIDI records. Users should be able to use open source connections with each

[10] `http://www.cs.wm.edu/semeru/exemplar/#casestudy` (verified 03/28/2011)

other and communicate. A GUI should enable users to save conversations and replay sounds."

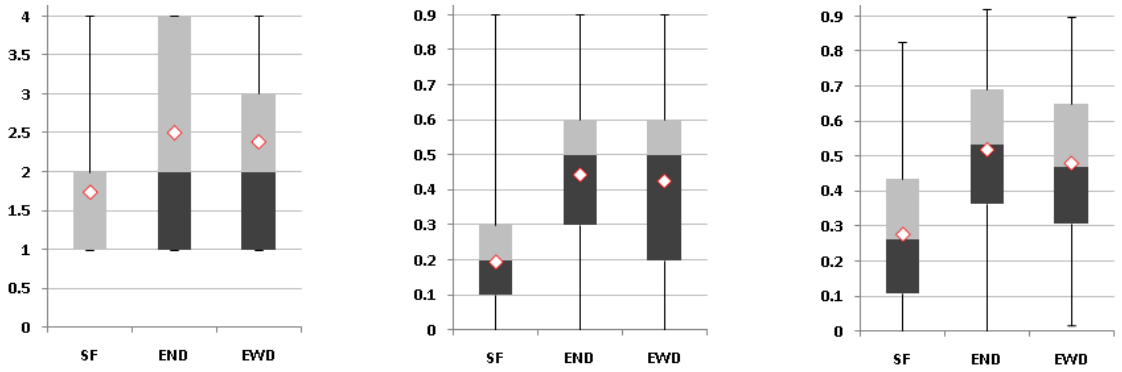
2. "Implement an application that performs pattern matching operations on a character sequences in the input text files. The application should support iterating through the found sequences that match the pattern. In addition, the application should support replacing every subsequence of the input sequence that matches the pattern with the given replacement string."

Additional criteria for these tasks is that they should represent real-world programming tasks and should not be biased towards any of the search engines that are used in this experiment. Descriptions of these tasks should be flexible enough to allow participants to suggest different keywords for searching. This criteria significantly reduces any bias towards evaluated search engines.

2.5.6 Normalizing Sources of Variations

Sources of variation are all issues that could cause an observation to have a different value from another observation. We identify sources of variation as the prior experience of the participants with specific applications retrieved by the search engines in this study, the amount of time they spend on learning how to use search engines, and different computing environments which they use to evaluate retrieved applications. The first point is sensitive since some participants who already know how some retrieved applications behave are likely to be much more effective than other participants who know nothing of these applications.

We design this experiment to drastically reduce the effects of covariates (i.e., nuisance factors) in order to normalize sources of variations. Using the cross-validation design we normalize variations to a certain degree since each participant uses all three search engines on different tasks.



(a) Confidence level, *C*.

(b) Precision, *P*.

(c) Normalized Discounted Gain, *NG*.

Figure 2.3: Statistical summary of the results of the case study for *C* and *P*.The center point represents the mean. The dark and light gray boxes are the lower and upper quartiles, respectively. The thin line extends from the minimum to the maximum value.

2.5.7 Tests and The Normality Assumption

We use one-way ANOVA, and randomization tests [104] to evaluate the hypotheses. ANOVA is based on an assumption that the population is normally distributed. The law of large numbers states that if the population sample is sufficiently large (between 30 to 50 participants), then the central limit theorem applies even if the population is not normally distributed [103, pages 244-245]. Since we have 39 participants, the central limit theorem applies, and the above-mentioned tests have statistical significance.

2.5.8 Threats to Validity

In this section, we discuss threats to the validity of this case study and how we address these threats.

2.5.8.1 Internal Validity

Internal validity refers to the degree of validity of statements about cause-effect inferences. In the context of our experiment, threats to internal validity come from confounding the effects of differences among participants, tasks, and time pressure.

Participants. Since evaluating hypotheses is based on the data collected from participants, we identify two threats to internal validity: Java proficiency and motivation of participants.

Even though we selected participants who have working knowledge of Java as it was documented by human resources, we did not conduct an independent assessment of how proficient these participants are in Java. The danger of having poor Java programmers as participants of our case study is that they can make poor choices of which retrieved applications better match their queries. This threat is mitigated by the fact that all participants from Accenture worked on successful commercial projects as Java programmers.

The other threat to validity is that not all participants could be motivated sufficiently to evaluate retrieved applications. We addressed this threat by asking participants to explain in a couple of sentences why they chose to assign certain confidence level to applications, and based on their results we financially awarded top five performers.

Tasks. Improper tasks pose a big threat to validity. If tasks are too general or trivial (e.g., open a file and read its data into memory), then every application that has file-related API calls will be retrieved, thus creating bias towards Exemplar. On the other hand, if application and domain-specific keywords describe task (e.g., `genealogy` and `GENTECH`), only a few applications will be retrieved whose descriptions contain these keywords, thus creating a bias towards Sourceforge. To avoid this threat, we based the task descriptions on a dozen specifications of different software

H	Var	Approach	Samples	Min	Max	Median	μ	C	p
H_1	C	EWD	1273	1	4	2	2.35	-0.02	< 0.0001
		SF	1273	1	4	1	1.82		
H_2	P	EWD	76	0.12	0.74	0.42	0.41	0.34	< 0.0001
		SF	76	0.075	0.73	0.48	0.46		
H_3	NG	EWD	76	0.02	0.89	0.47	0.48	-0.05	< 0.0001
		SF	76	0	0.83	0.26	0.28		
H_4	C	EWD	1273	1	4	2	2.35	0.01	< 0.0001
		END	1273	1	4	3	2.47		
H_5	P	EWD	76	0.12	0.74	0.42	0.41	0.41	0.78927
		END	76	0.075	0.73	0.48	0.46		
H_6	NG	EWD	76	0.02	0.89	0.47	0.48	-0.02	0.71256
		END	76	0	0.92	0.53	0.52		
H_7	C	END	1307	1	4	3	2.47	-0.02	< 0.0001
		SF	1307	1	4	1	1.84		
H_8	P	END	76	0.075	0.73	0.5	0.47	0.4	< 0.0001
		SF	76	0	0.71	0.24	0.27		
H_9	NG	END	76	0	0.92	0.53	0.52	0.08	< 0.0001
		SF	76	0	0.83	0.26	0.28		

Table 2.2: Results of randomization tests of hypotheses, H, for dependent variable specified in the column Var (C, P, or NG) whose measurements are reported in the following columns. Extremal values, Median, Means, μ, and the pearson correlation coefficient, C, are reported along with the results of the evaluation of the hypotheses, i.e., statistical significance, p.

systems that were written by different people for different companies. The tasks we used in the case study are available for download at the Exemplar website [11].

Time pressure. Each experiment lasted for two hours, and for some participants it was not enough time to explore all retrieved applications for each of eight tasks. It is a threat to validity that some participants could try to accomplish more tasks by shallowly evaluating retrieved applications. To counter this threat we notified participants that their results would be discarded if we did not see sufficient reported evidence of why they evaluated retrieved applications with certain confidence levels.

[11] `http://www.xemplar.org`, follow the "About Exemplar" link to the "Case Study" section.

2.5.8.2 External Validity

To make results of this case study generalizable, we must address threats to external validity, which refer to the generalizability of a casual relationship beyond the circumstances of our case study. The fact that supports the validity of the case study design is that the participants are highly representative of professional Java programmers. However, a threat to external validity concerns the usage of search tools in the industrial settings, where requirements are updated on a regular basis. Programmers use these updated requirements to refine their queries and locate relevant applications using multiple iterations of working with search engines. We addressed this threat only partially, by allowing programmers to refine their queries multiple times.

In addition, it is sometimes the case when engineers perform multiple searches using different combinations of keywords, and they select certain retrieved applications from each of these search results. We believe that the results produced by asking participants to decide on keywords and then perform a single search and rank applications do not deviate significantly from the situation where searches using multiple (refined) queries are performed.

Another threat to external validity comes from different sizes of software repositories. We populated Exemplar's repository with all Java projects from the Sourceforge repository to address this threat to external validity.

Finally, the help documentation that we index in Exemplar is an external threat to validity because this documentation is provided by a third-party, and its content and format may vary. We addressed this thread to validity by using the Java documentation extracted as JavaDocs from the official Java Development Kit, which has a uniform format.

2.6 Empirical Results

In this section, we report the results of the case study and evaluate the null hypotheses.

2.6.1 Variables

A main independent variable is the search engine (SF, EWD, END) that participants use to find relevant Java applications. Dependent variables are the values of confidence level, C, precision, P, and normalized discounted cumulative gain, NG. We report these variables in this section. The effect of other variables (task description length, prior knowledge) is minimized by the design of this case study.

2.6.2 Testing the Null Hypothesis

We used ANOVA[103] to evaluate the null hypothesis H_{0-null} that the variation in an experiment is no greater than that due to normal variation of individuals' characteristics and error in their measurement. The results of ANOVA confirm that there are large differences between the groups for C with $F = 129 > F_{crit} = 3$ with $p \approx 6.4 \cdot 10^{-55}$ which is strongly statistically significant. The mean C for the SF approach is 1.83 with the variance 1.02, which is smaller than the mean C for END, 2.47 with the variance 1.27, and it is smaller than the mean C for EWD, 2.35 with the variance 1.19. Also, the results of ANOVA confirm that there are large differences between the groups for P with $F = 14 > F_{crit} = 3.1$ with $p \approx 4 \cdot 10^{-6}$ which is strongly statistically significant. The mean P for the SF approach is 0.27 with the variance 0.03, which is smaller than the mean P for END, 0.47 with the variance 0.03, and it is smaller than the mean P for EWD, 0.41 with the variance 0.026. Based on these results we reject the null hypothesis and we accept the alternative hypothesis H_{0-alt}.

A statistical summary of the results of the case study for *C*, *P*, and *NG* (median, quartiles, range and extreme values) are shown as box-and-whisker plots in Figure 2.3(a), Figure 2.3(b), and Figure 2.3(c) correspondingly with 95% confidence interval for the mean.

2.6.3 Comparing Sourceforge with Exemplar

To test the null hypothesis H_1, H_2, H_3, H_7, H_8, and H_9 we applied six randomization tests, for *C*, *P*, and *NG* for participants who used SF and both variants of Exemplar. The results of this test are shown in Table 2.2. The column `Samples` shows that 37 out of a total of 39 participants participated in all experiments and created rankings for *P* (two participants missed one experiment). `Samples` indicates the number of results which were ranked in the case of variable *C*. For *NG*, `Samples` shows the number of sets of results. Based on these results we reject the null hypotheses H_1, H_2, H_3, H_7, H_8, and H_9, and we accept the alternative hypotheses that states that **participants who use Exemplar report higher relevance and precision on finding relevant applications than those who use Sourceforge**.

2.6.4 Comparing EWD with END

To test the null hypotheses H_4, H_5, and H_6, we applied two t-tests for paired two sample for means, for *C*, *P*, and *NG* for participants who used END and EWD. The results of this test are shown in Table 2.2. Based on these results we reject the null hypothesis H_4, and that say that **participants who use END report higher relevance when finding relevant applications than those who use EWD**. On the other hand, we fail to accept the null hypotheses H_5 and H_6, and say that **participants who use END do not report higher precision or normalized discounted cumulative gain than those who use EWD**.

There are several explanations for this result. First, given that our dataflow analysis is imperfect, some links are missed and subsequently, the remaining links cannot affect the ranking score significantly. Second, it is possible that our dataflow connectivity-based ranking mechanism needs fine-tuning, and it is a subject of our future work. Finally, after the case study, a few participants questioned the idea of dataflow connections between API calls. A few participants had vague ideas as to what dataflow connections meant and how to incorporate them into the evaluation process. This phenomenon points to a need for better descriptions of Exemplar's internals in any future case studies.

2.6.5 Qualitative Analysis and User Comments

Thirty-five of the participants in the case study completed exit surveys (see Table 2.3) describing their experiences and opinions. Of these, 22 reported that seeing standalone fragments of the code alongside relevant applications would be more useful than seeing only software applications. Only four preferred simply applications listed in the results, while nine felt that either would be useful. Several users stated that seeing entire relevant applications provides useful context for code fragments, while others read code in order to understand certain algorithms or processes, but ultimately re-implement the functionality themselves. After performing the case study, we responded to these comments by providing the source code directly on Exemplars results page, with links to the lines of files where relevant API calls are used. This constitutes a new feature of Exemplar, which was not available to the participants during the user study.

Nineteen of the participants reported using source code search engines rarely, six said they sometimes use source code search engines, and nine regularly. Of those that only rarely use source code search engines, eight adapted Googles web search to look for code. Meanwhile, when asked

	Question
1	How many years of programming experience do you have?
2	What programming languages have you used and for how many years each?
3	How often do you use code search engines?
4	What code search engines have you used and for how long?
5	How often can you reuse found applications or code fragments in your work?
6	What is the biggest impediment to using code search engines, in your opinion?
7	Would you rather be able to retrieve a standalone fragment of code or an entire application with a relevant fragment of code in it?

Table 2.3: The seven questions answered by the case study participants during the exit survey. All questions were open-ended.

to state the biggest impediment in using source code search engines, 14 participants answered that existing engines return irrelevant results, four were mostly concerned with the quality of the returned source code, six did not answer, and 11 reported some other impediment. These results support the recent studies [102] and point to a strong need for improved code engines that return focused, relevant results. New engines should show the specific processes and useful fragments of code. We believe that searching by API calls can fill this role because calls have specific and well-defined semantics along with high-quality documentation.

The following is a selection of comments written by participants in the user study. Scanned copies of all questionnaires are publicly available on the Exemplar about page.

- "The Exemplar search is handy for finding the APIs quickly."
- "Many SourceForge projects [have] no files or archives."
- "A standalone fragment would be easy to see and determine relevance to my needs, but an entire application would allow for viewing context which would be useful."
- "[I] typically reuse the pattern/algorithm, not [the] full code."

- "Often [retrieved code or applications] give me a clue as to how to approach a development task, but usually the code is too specific to reuse without many changes."
- "Often, [with source code search engines] I find results that do not have code."
- "[I reuse code] not in its entirety, but [I] always find inspiration."
- "There seems to be a lot of time needed to understand the code found before it can be usefully applied."
- "Could the line number reference [in Exemplar] invoke a collapsible look at the code snippet?"
- "With proper keywords used, [Exemplar] is very impressive. However, it does not filter well the executables and non-code files. Overall, great for retrieving simple code snippets."
- "Most, if not all, results returned [by Exemplar] provided valuable direction/foundation for completing the required tasks."
- "During this experiment it became clear that searching for API can be much more effective than by keywords in many instances. This is because it is the APIs that determine functionality and scope potential."
- "SourceForge was not as easy to find relevant software as hoped for."
- "[Using SourceForge] I definitely missed the report within Exemplar that displays the matching API methods/calls."
- "SourceForge appears to be fairly unreliable for projects to actually contain any files."
- "Exemplar seems much more intuitive and easier to use than SourceForge."
- "Great tool to find APIs through projects."
- "It was really helpful to know what API calls have been implemented in the project while using Exemplar."

The users were overall satisfied with Exemplar, preferring it to SourceForges search. In Section 2.6, we found that they rated results from Exemplar with statistically-significantly higher confidence levels than SourceForge. From our examination of these surveys, we confirm the findings from our analysis in Section 2.6 and conclude that the participants in the case study did prefer to search for applications using Exemplar rather than SourceForge. Moreover, we conclude that the reason they preferred Exemplar is because of Exemplar's search of API documentation.

2.7 Analysis of user study results

During our case study of Exemplar (see Section 2.5), we found that the original version of Exemplar outperformed SourceForge in terms of both confidence and precision. In this section, we will explore why Exemplar outperformed SourceForge. Our goal is to identify which components of Exemplar lead to the improvements and to determine how users interpreted tasks and interacted with the source code search engine. Specifically, we intend to answer the following research questions (RQ):

RQ_1 Do high Exemplar scores actually match high confidence level ranks from the participants?

RQ_2 Do the components of the Exemplar score (WOS, RAS, and DCS scores) indicate relevance of applications when the others do not (e.g., do the components capture the same or orthogonal information about retrieved software applications)?

RQ_3 Is Exemplar sensitive to differences in the user queries when those queries were generated for the same task by different users?

We want to know how we can optimize Exemplar given answers to these research questions. Additionally, we want to study how design decisions (such as whether RAS considers the frequency of API calls, see Section 2.4) affected Exemplar.

2.7.1 Comparing Scores in Confidence Levels

Exemplar computes a score for every application to represent that application's relevance to the user query (see Section 2.4). Ideally, higher scores will be attached to applications with greater relevance. We know from Section 2.6 that Exemplar returns many relevant results, but this information alone is insufficient to claim that a high score from Exemplar for an application is actually an indicator of

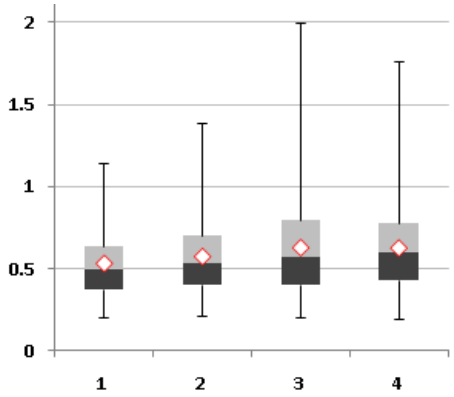


Figure 2.4: Statistical summary of the scores from the case study of Exemplar. The y-axis is the score given by Exemplar during the case study. The x-axis is the confidence level given by users to results from Exemplar.

the relevance of that application, because irrelevant applications could still obtain high scores (see Section 2.9).

To better understand the relationship of Exemplar ranking scores to relevance of retrieved software applications, and to answer RQ_1, we examined the scores given to all results given by Exemplar during the user study. We also consider the Java programmers' confidence level rankings of those results. The programmers ranked results using a four-level Likert scale (see Section 2.5.1). We grouped Exemplars scores for applications by the confidence level provided by the case study participants for those applications. Figure 2.4 is a statistical summary of the scores for the results, grouped by the confidence level. These scores were obtained from Exemplar using all 209 queries that the users produced for 22 tasks during the case study[12]. We have made all these results available for download from the Exemplar website so that other researchers can reproduce our analysis and the results.

[12]Note that the participants only completed 22 out of 26 total tasks available.

2.7.1.1 Hypotheses for RQ_1

We want to determine to what degree the mean of the scores from Exemplar increase as the user confidence level rankings increase. We introduce the following null and alternative hypotheses to evaluate the significance of any difference at a 0.05 level of confidence.

$H_{10-null}$ The null hypothesis is that there is no difference in the values of Exemplar scores of applications among the groupings by the confidence level.

H_{10-alt} An alternative hypothesis to $H_{10-null}$ is that there is a statistically significant difference in the values of Exemplar scores of applications among the groupings by the confidence level.

2.7.1.2 Testing the Null Hypothesis

The results of ANOVA for $H_{10-null}$ confirm that there are statistically-significant differences among the groupings by confidence level. Intuitively, these results mean that higher scores imply higher confidence levels from programmers. Higher confidence levels, in turn, point to higher relevance (see Section 2.5). Table 2.6 shows the F-value, P-value, and critical F-value for the variance among the groups. We reject the null hypothesis $H_{10-null}$ because the $F > F_{critical}$. Additionally, $P < 0.05$. Therefore, we find evidence supporting the alternative hypothesis H_{10-alt}.

Finding supporting evidence for H_{10-alt} suggests that we can answer RQ_1. To confirm these results, however, we grouped the results in terms of relevant (e.g., confidence 3 or 4) and non-relevant (e.g., confidence 1 or 2), and tested the difference of these groups. A randomization test of these groups showed a P-value of < 0.0001, which provides further evidence for answering RQ_1. Therefore, we find that higher Exemplar scores do in fact match to higher confidence level rankings from participants in the user study.

	PC1	PC2	PC3
Proportion	43.8%	31.5%	24.8%
Cumulative	43.8%	75.3%	100%
WOS	-0.730	0.675	0.106
RAS	0.995	0.091	-0.039
DCS	-0.010	-0.303	0.953
ALL	0.477	0.839	0.263

Table 2.4: Factor loading through Principal Component Analysis of each of the scores (WOS, RAS, and DCS) that contribute to the final score in Exemplar (ALL).

	WOS	RAS	DCS	ALL
WOS	1	-0.741	-0.104	0.142
RAS	-0.741	1	-0.046	0.482
DCS	-0.104	-0.046	1	-0.005
ALL	0.142	0.482	-0.005	1

Table 2.5: Spearman correlations of the score components to each other and to the final ranking.

2.7.2 Principal Components of the Score

The relevance score that Exemplar computes for every retrieved application is actually a combination of the three metrics (WOS, RAS, and DCS) presented in Section 2.3. Technically, these three metrics were added together with equal weights using an affine transformation during the case study. Ideally, each of these metrics should contribute orthogonal information to the final relevance score, meaning that each metric will indicate the relevance of applications when the others might not. To analyze the degree to which WOS, RAS, and DCS contribute orthogonal information to the final score, and to address RQ_2, we used Principal Component Analysis (PCA)[47]. PCA locates uncorrelated dimensions in a dataset and connects input parameters to these dimensions. By looking at how the inputs connect to the principal components, we can deduce how each component relates to the others.

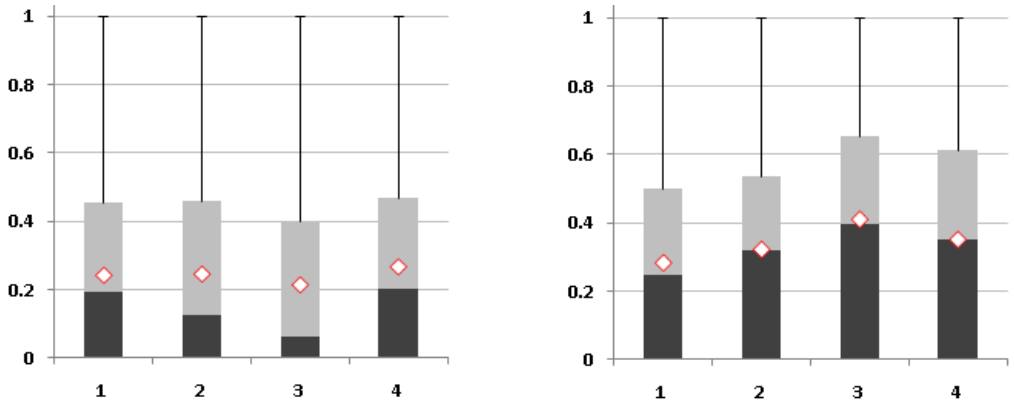
To apply PCA, we ran Exemplar using the queries from the case study and obtained WOS,

RAS, DCS, and combined scores for the top ten applications for each of the queries. We then used these scores as the input parameters to be analyzed. PCA identified three principal components; Table 2.4 shows the results of this analysis. We find that the first principal component is primarily RAS (99.5% association), the second component is somewhat linked to WOS (67.5% association), and the third component is primarily DCS (95.3% association). The final Exemplar score (denoted ALL) is linked to each of the primary components, which we expect because the input parameters combine to form the Exemplar score. Because WOS, RAS, and DCS are all positively associated with their own principal components, we conclude that each metric provides orthogonal information to Exemplar.

We also computed the Spearman correlations[103] for each input parameter to each other. These correlations are presented in Table 2.5. WOS and RAS are negatively correlated to one another, a fact suggesting that the two metrics contribute differently to the final ranking score. Moreover, RAS exhibits moderate correlation to the final Exemplar score, while WOS is at least positively correlated. DCS, however, is entirely uncorrelated to either RAS or WOS. We draw two conclusions given these results. First, we answer RQ_2 by observing that RAS and WOS do capture orthogonal information (see PCA results in Table 2.4). Second, because DCS does not correlate to the final score and because DCS did not appear to benefit Exemplar during the case study (see Section 2.6.4), we removed DCS from Exemplar. We do not consider DCS in any other analysis in this section.

2.7.2.1 Analysis of WOS and RAS

Given that WOS and RAS contribute orthogonally to the Exemplar score, we now examine whether combining them in Exemplar returns more relevant applications versus each metric individually. We judged the benefit of WOS and RAS by computing each metric for every application using the



(a) WOS (b) RAS

Figure 2.5: Statistical summary of the WOS and RAS scores from the case study of Exemplar.

queries from the case study. We then grouped both sets of scores by the confidence level assigned to the application by the case study participants in a setup similar to that in Section 2.7.1. Figure 2.5a and 2.5b are statistical summaries for the WOS and RAS scores, respectively. We introduce the following null and alternative hypotheses to evaluate the significance of any difference at a 0.05 level of confidence.

$H_{11-null}$ The null hypothesis is that there is no difference in the values of WOS scores of applications among the groupings by confidence level.

H_{11-alt} An alternative hypothesis to $H_{11-null}$ is that there is a statistically significant difference in the values of WOS scores of applications among the groupings by confidence level.

$H_{12-null}$ The null hypothesis is that there is no difference in the combined values of RAS scores of applications among the groupings by confidence level.

H_{12-alt} An alternative hypothesis to $H_{12-null}$ is that there is a statistically significant difference in the values of RAS scores of applications among the groupings by confidence level.

	F	P	$F_{critical}$
$H_{10-null}$	12.31	6E-08	2.61
$H_{11-null}$	1.97	0.12	2.61
$H_{12-null}$	8.18	2E-05	2.61

Table 2.6: Results of testing $H_{10-null}$, $H_{11-null}$, and $H_{12-null}$

2.7.2.2 Testing the Null Hypotheses

We used one-way ANOVA to evaluate $H_{11-null}$ and $H_{12-null}$ that the variation in the experiment is no greater than that due to normal variation of the case study participants choices of confidence level as well as chance matching by WOS and RAS, respectively. The results of ANOVA confirm that there are statistically-significant differences among the groupings by confidence level for RAS, but not for WOS. Table 2.6 shows the F-value, P-value, and critical F-value for the variance among the groups for WOS. Table 2.6 shows the same values for RAS. We do not reject the null hypothesis $H_{11-null}$ because $F < F_{critical}$. Additionally, $P > 0.05$. Therefore, we can not support the alternative hypothesis H_{12-alt}. On the other hand, we reject the null hypothesis $H_{12-null}$ because the $F > F_{critical}$. $P < 0.05$. Therefore, we find evidence supporting the alternative hypothesis H_{12-alt}.

We finish our study of the contributions of RAS, WOS, and DCS by concluding that RAS improves the results by a statistically-significant amount. Meanwhile, we cannot infer any findings about WOS because we could not reject $H_{11-null}$. We did observe specific instances in the case study where WOS contributed to the retrieval of relevant results when RAS did not (see Section 2.9). Therefore, we include WOS in the final version of Exemplar, albeit with a weight reduced by 50% from 0.5 to 0.25. We also increased the weight of RAS by 50% from 0.5 to 0.75 because we found that RAS contibutes to more relevant results than WOS.

2.7.3 Keyword Sensitivity of Exemplar

Recent research shows that users tend to generate different kinds of queries [4]. It may be the case that different users of Exemplar create different queries which represent the same task that those users need to implement. If this occurs, some users may see relevant results, whereas others see irrelevant ones. During the case study, we provided the participants with 22 varied tasks. The participants were then free to read the tasks and generate queries on their own. Exemplar may retrieve different results for the same task given different queries, even if the participants generating those queries all interpreted the meaning of the task in the same way. This presents a threat to validity for the case study because different participants may see different results (and produce different rankings) for the same task. For example, consider Task 1 from Section 2.5.5. Table 2.7 shows two separate queries generated independently by users during the case study for this task[13]. By including more keywords, the author of the second query found three different applications than the author of the first query. In this section, we will answer RQ_3 by studing how sensitive Exemplar is to variations in the query as formulated by different users for the same task.

First, we need to know how different the queries and the results are for individual tasks. We computed the *query overlap* to measure how similar queries are for each task. We defined query overlap as the pairwise comparison of the number of words, which overlap for each query. The formula is $queryoverlap = \frac{|query_1 \bigcap query_2|}{|query_1 \bigcup query_2|}$ where `query1` is the set of words is the first query and `query2` is the set of words in the second query. For example, consider the queries "sound voice midi" and "sound voice audio midi connection gui". The queries share the words "sound", "voice", and "midi". The total set of words is "sound voice midi audio connection gui". Therefore, the query

[13]We generated the results in Table 2.7 using Exemplar in the same configuration as in the case study, which can be accessed here: `http://www.xemplar.org/original.html` (verified 03/28/2011)

	"sound voice midi"	"sound voice audio midi connection gui"
1	Tritonus	Tritonus
2	Java Sound Res	RasmusDSP
3	RasmusDSP	Audio Develop
4	TuxGuitar	TuxGuitar
5	MidiQuickFix	MidiQuickFix
6	Audio Develop	Java Sound Res
7	FluidGUI	RPitch
8	DGuitar	DGuitar
9	Cesar	Music and Audio
10	Saiph	JVAPTools

Table 2.7: The top ten applications returned by Exemplar for two separate queries. Both queries were generated by users during the case study while reading the same task. Shaded cells indicate applications in both sets of results. Application names in bold were rated with confidence level 3 or 4 (relevant or highly-relevant) by the author of the associated query. Note: Ties of relevance scores are broken randomly; applications with identical scores may appear in a different order.

overlap is 0.5, or 50%. To obtain the query overlap for a task, we simply computed the overlap numbers for every query to every other query in the task. The queries were processed in the same way as they are in Exemplar; we did not perform stemming or removal of stop words.

Because we see different queries for each task, we expect to see different sets of results from Exemplar over a task. We surmise that if two users give two different queries for the same task, then Exemplar will return different results as well. We want to study the degree to which Exemplar is sensitive to changes in the query for a task. Therefore, we calculate the *results overlap* for each task using the formula $resultsoverlap = \frac{|unique-total|}{|expected-total|}$ where `total` is the total number of results found for a given task, `unique` is the number of those results which are unique, and `expected` is the number of results we expect if all the results overlapped (e.g., the minimum number of unique results possible). For example, consider the situation in Table 2.7 where, for a single task, two users created two different queries. In the case study, participants examined the top ten results, meaning that Exemplar returned 20 total results. At least ten of the results must be unique, which is the

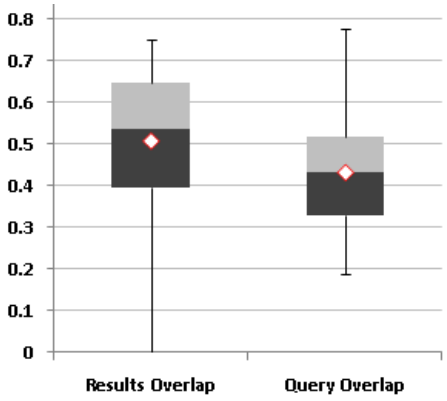


Figure 2.6: Statistical summary of the overlaps for tasks. The x-axis is the type of overlap. The y-axis is the value of the overlap.

expected number if Exemplar returned the same set for all three queries. In Table 2.7, however, 13 of the results were unique, results overlap would be 0.7, or 70% overlapped.

Statistical summaries of the results overlap and query overlap are in Figure 2.6. The Spearman correlations for the overlaps was 0.356. We observe a weak correlation between results and query overlap, which we expect because more similar queries will most likely cause Exemplar to produce more similar results. Therefore, to answer RQ_3, we do find evidence that Exemplar is sensitive to differences in the queries, even if those queries were created to address the same task.

2.7.4 Sensitivity to the Number of API Calls

The RAS component of Exemplar is responsible for ranking applications based on the API calls made in those applications. This component first locates a number of descriptions of API calls which match the keywords provided in the user's query. It then matches those API calls to applications which use those calls. During the case study, we limited the number of API calls that RAS considers to 200 due to performance overhead. In this section, we analyze the effect this design decision had on the search results.

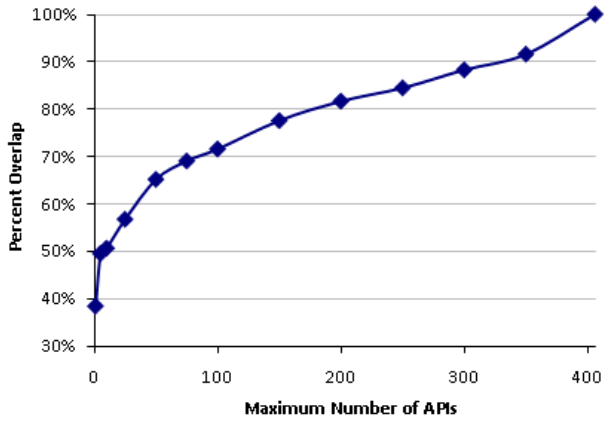


Figure 2.7: A chart of the results overlap from various levels of *maxapi*. The x-axis is the value of the overlap. The y-axis is the value of *maxapi*.

The maximum number of APIs to consider is an internal parameter to Exemplar called *maxapi*. To study its effects, we first obtained all 209 queries written by participants in the case study from Section 2.5. We then set *maxapi* to infinity (so that potentially every API could be returned) and ran every query through Exemplar. From this run, we determined that the maximum number of API calls extracted for any query was 406. We also stored the list of results from this run.

We then ran Exemplar with various entries as input for *maxapi* ranging between 1 and 406[14]. We then calculated the *results overlap* for the results of each of these runs against the results from the run in which *maxapi* was set to infinity. In this way, we computed the percent of overlap of the various levels of *maxapi* with case in which all API calls are considered. The results of this analysis are summarized in Figure 2.7. We observe that when *maxapi* is set to a value greater than or equal to 200, the percent overlap is always above 80%, meaning that 80% of the results are identical to those in the case when all API calls are considered. We set *maxapi* to 200 in the remainder of this paper.

[14]Note that Exemplar produces the same results when *maxapi* is set to 406 and infinity since 406 was the maximum amount of API calls returned.

2.7.5 Sensitivity to Frequency of API Calls

The RAS component ranking considers the frequency of each API call that occurs in each application. For example, if an application A makes an API call c twice, and an application B makes an API call c only once, and c is detemined to be relevant to the user query, then application A will be ranked higher than B. In Exemplar, we use static analysis to determine the API calls used by an application. Therefore, we do not know the precise number of times an API call is actually made in each application because we do not have execution information for these applications. For example, consider the situation where application A calls c twice and B calls c once. If the call to c in B occurs inside a loop, B may call c many more times than A, but we will not capture this information.

We developed a binary version of RAS to study the effects this API frequency information may cause in our case study. The binary version of RAS does not consider the frequency of each API call in the applications. More formally, the binary RAS calculates the scores according to the formula $S^j_{ras} = \frac{\sum_{i=1}^{p} C^j_i}{|A|^j}$, where $|A|^j$ is the total number of API calls in the application j, and p is the number of API calls retrieved for the query.

We then executed Exemplar using the 209 queries from the case study in Section 2.5 for both the binary version of RAS and the RAS that considers frequencies of API calls as described in Section 2.3.3. We computed the *results overlap* between the results for both. The mean overlap for the results of every query was 93.2%. The standard deviation was 13.4%. Therefore, we conclude that the results from Exemplar with the binary version of RAS are not dramatically different from the frequency-based version of RAS. We use the frequency-based version of RAS in the remainder of this paper.

2.8 Evaluation of changes to Exemplar

We made several alterations to Exemplar based on our analysis in Section 2.7. Specifically, we removed DCS, rebalanced the weights of WOS and RAS (to 0.25 and 0.75), and updated the interface so that project source code is visible without downloading whole projects. We compare the quality of the results from the updated version of Exemplar against the previous version. In this study, we refer to the previous Exemplar as Exemplar$_{OLD}$ and the new Exemplar as Exemplar$_{NEW}$.

2.8.1 Methodology

We performed a case study identical in design to that presented in Section 2.5, except that we evaluate two engines (Exemplar$_{NEW}$, Exemplar$_{OLD}$) instead of three (EWN, END, SF). Table 2.8 outlines the study. We chose END to represent the old Exemplar because END was the best-performing configuration. In this case, we randomly divided 26 case study participants[15] into two groups. There were two experiments, and both groups participated in each. In each experiment, each group was given a different search engine (e.g., Exemplar$_{NEW}$ or Exemplar$_{OLD}$) and a set of tasks. The participants then generated queries for each task and entered those queries into the specifed search engine. The participants rated each result on a four-point Likert scale as in Section 2.5. From these ratings, we computed the three measures confidence (C), precision (P), and normalized discounted cumulative gain (NG).

[15]Nine of the participants in this study were graduate students from the University of Illinois at Chicago. Five were graduate students at the College of William & Mary. Ten were undergraduate students at William & Mary. We reimbursed the participants $35 after the case study.

Experiment	Group	Search Engine	Task Set
1	G1	NEW	T1
	G2	OLD	T2
2	G1	OLD	T2
	G2	NEW	T1

Table 2.8: Plan for the case study of Exemplar$_{NEW}$ and Exemplar$_{OLD}$.

2.8.2 Hypotheses

We introduce the following null and alternative hypotheses to evaulate the differences in the metrics at a 0.05 confidence level.

H_{13} The null hypothesis is that there is no difference in the values of C for Exemplar$_{NEW}$ versus Exemplar$_{OLD}$. Conversely, the alternative is that there is statistically significant difference in the values of C for Exemplar$_{NEW}$ versus Exemplar$_{OLD}$.

H_{14} The null hypothesis is that there is no difference in the values of P for Exemplar$_{NEW}$ versus Exemplar$_{OLD}$. Conversely, the alternative is that there is statistically significant difference in the values of P for Exemplar$_{NEW}$ versus Exemplar$_{OLD}$.

H_{15} The null hypothesis is that there is no difference in the values of NG for Exemplar$_{NEW}$ versus Exemplar$_{OLD}$. Conversely, the alternative is that there is statistically significant difference in the values of NG for Exemplar$_{NEW}$ versus Exemplar$_{OLD}$.

2.8.3 Results

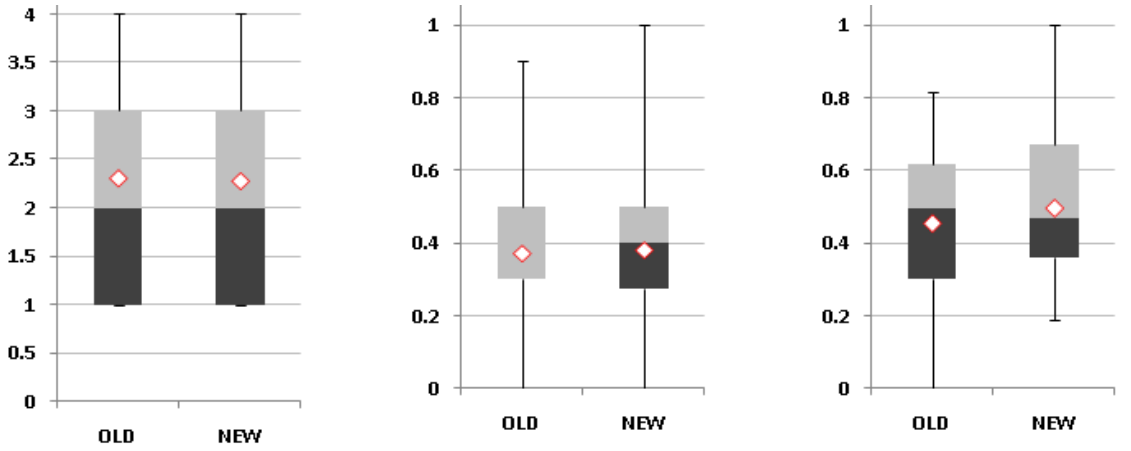
We applied randomization tests to evaluate the hypotheses H_{13}, H_{14}, and H_{15}. The results of this test are in Table 2.9. We do not reject the null hypothesis H_{14} because the P-value is greater than 0.05. Therefore, participants do not report a statistically-significant difference in terms of precision of the results. On the other hand, we reject the null hypotheses H_{13} and H_{15}, meaning that participants report higher confidence level in the results. Also, the participants report higher normalized discounted cumulative gain when using Exemplar$_{NEW}$ versus Exemplar$_{OLD}$.

H	Var	Approach	Samples	Min	Max	Median	μ	C	p
H_{13}	C	Exemplar$_{NEW}$	556	1	4	2	2.27	0.05	0.00156
		Exemplar$_{OLD}$	556	1	4	2	2.30		
H_{14}	P	Exemplar$_{NEW}$	40	0	1.00	0.40	0.38	-0.15	0.23738
		Exemplar$_{OLD}$	40	0	0.90	0.30	0.37		
H_{15}	NG	Exemplar$_{NEW}$	40	0.19	1.00	0.47	0.50	-0.15	0.04507
		Exemplar$_{OLD}$	40	0	0.82	0.49	0.46		

Table 2.9: Results of randomization tests of hypotheses, H, for dependent variable specified in the column Var (C, P, or NG) whose measurements are reported in the following columns. Extremal values, Median, Means, μ, and the pearson correlation coefficient, C, are reported along with the results of the evaluation of the hypotheses, i.e., statistical significance, p.

The difference in average confidence level between the updated and original versions of Exemplar is statistically significant, as seen in Figure 2.8(a), though the difference is very small. The difference in precision is not statistically significant (see Figure 2.8(b)). One explanation for the small size of this difference is that both versions of Exemplar return the same sets of applications to the user. Returning the same set of applications is expected because both Exemplar$_{NEW}$ and Exemplar$_{OLD}$ use the same underlying information to locate these applications (e.g., API calls and project descriptions). The order of the results is also important, and the new version of Exemplar does return the more-relevant results in higher positions, as reported by the normalized discounted cumulative gain (NG, see Figure 2.8(c)).

Table 2.10 illustrates an example of the improvement made by Exemplar$_{NEW}$. This table includes the results for the same query on both engines as well as the confidence level for the applications as reported by a participant in the case study. The normalized discounted cumulative gain is higher in this example for Exemplar$_{NEW}$ than Exemplar$_{OLD}$. Even though a majority of the applications are shared by both sets of results, Exemplar$_{NEW}$ organizes the results such that the most-relevant applications appear sooner.



(a) Confidence level, C.

(b) Precision, P.

(c) Normalized Discounted Gain, NG.

Figure 2.8: Statistical summary of C, P, and NG from the case study evaluating the new version of Exemplar. The y-axis is the value for C, P, or NG from the case study. The x-axis is the version of Exemplar.

2.8.4 Participant Comments on Exemplar$_{NEW}$

Seventeen of the case study participants answered the same exit survey from Table 2.3. The responses generally support those which we discuss in Section 2.6.5: roughly half of the participants reported rarely or never using source code search engines, and of those a majority prefer to use Google. The top reason cited for not using source code search engines was the preceived poor quality results given by those engines. These results, along with those in Section 2.6.5, are a strong motivation for improvements in source code search engines.

In addition to rebalacing the weights of the ranking components in Exemplar$_{NEW}$, we made the source code of the applications immediately available through the engine. The following are comments provided by participants regarding these changes. We conclude from these comments that (1) users prefer to see source code along with relevant applications, and (2) API calls helped participants determine the relevance of results.

"glyph painting"			
Exemplar$_{OLD}$		Exemplar$_{NEW}$	
Jazilla	1	Jazilla	1
DrawSWF	4	DrawSWF	4
Image inpainting	1	McBilliards	3
SandboxPix	1	Waba for Dos	3
McBilliards	3	BioGeoTools	1
Waba for Dos	3	TekMath	2
BioGeoTools	1	SWTSwing	0
TekMath	2	Java2C	0
SWTSwing	0	JSpamAssassin	0
DESMO-J	0	netx	0
NG Top 6	0.5143		0.5826
NG Top 10	0.4247		0.4609

Table 2.10: The search results from a single query from the second case study; applications are listed with the assigned confidence levels. A case study participant generated the query and provided the relevancy rankings when evaluating Exemplar$_{OLD}$. Applications with a confidence level zero were not able to be accessed by the participant, and are discarded during our analysis. We ran the same query on Exemplar$_{NEW}$. The confidence levels for the results of Exemplar$_{NEW}$ are copied from the confidence levels given by the participant who ran Exemplar$_{OLD}$. *NG* represents the normalized discounted cumulative gain for the top 6 (all evaluated, zeros discarded) and top 10 (all retrieved, zeros included).

- "Very convenient to be able to open to view source files immediately. Much much more convenient to user."
- "[WOS in Exemplar$_{OLD}$] got in the way quite a bit"
- "I definitely like viewing code in the browser better"
- "[Exemplar$_{NEW}$] is really useful since we can know which API we should choose."
- "[API calls] are very useful if the call is relevant, a lot of API calls had nothing to do with the task."
- "[API calls] are very useful for determining initial area of source code which should be examined."

2.8.5 Suggestions for Future Work

The participants in the case study had several suggestions for Exemplar, and we have incorporated these into our future work. One participant asked that we filter "trivial" results such as API calls named `equal()` or `toString()`. Another suggested that we provide descriptions of API calls

directly on the results page. A participant also requested a way to sort and filter the API calls; he was frustrated that some source code files contain "the same type-check method many times."

2.9 Supporting Examples

Table 2.11 shows the results from Exemplar for three separate queries, including the top ten applications and the WOS and RAS scores for each[16]. For instance, consider the query *connect to an http server*. Only one of the top ten results from Exemplar is returned (see Table 2.11) due to a high WOS score (e.g., because the query matches the high-level description of the project). The remaining nine projects pertain to different problem domains, including internet security testing, programming utilities, and bioinformatics. These nine applications, however, all use API calls from the Java class `java.net.HttpURLConnection`[17]. Exemplar was able to retrieve these applications only because of the contribution from the RAS score.

Other queries may reflect the high-level concepts in a software application, rather than low-level details. For example, for the query *text editor*, Exemplar returns six of ten top results without any matching from RAS (see Table 2.11). While the query does match certain API calls, such as those in the class `javax.swing.text.JTextComponent`[18], Exemplar finds several text editing programs, which do not use API calls from matching documentation. Locating these applications was possible because of relatively high WOS scores.

[16]We generated the results in Table 2.11 using Exemplar in the same configuration as in the case study, which can be accessed here: `http://www.xemplar.org/original.html`

[17]The documentation for this API class can be found at: `http://download.oracle.com/javase/6/docs/api/java/net/HttpURLConnection.html` (verified 03/28/2011

[18]The documentation for this API class can be found at: `http://cupi2.uniandes.edu.co/site/images/recursos/javadoc/j2se/1.5.0/docs/api/javax/swing/text/JTextComponent.html` (verified 03/28/2011)

	"connect to http server"			"text editor"			"find replace string text files"		
	Application	WOS	RAS	Application	WOS	RAS	Application	WOS	RAS
1	DataShare	100%	0%	jeHep	52%	89%	RText	91%	0%
2	X4technology	0%	100%	XNap Commons	0%	100%	Nodepublisher	0%	66%
3	jpTools	0%	96%	SWediT	92%	0%	XERP	44%	18%
4	JMS for j2ms	0%	96%	Plugins jext	87%	0%	J	54%	0%
5	MicroEmulator	0%	96%	PalmEd	87%	0%	j-sand	53%	0%
6	ReadSeq bioinfo	0%	95%	PowerSwing	0%	85%	DocSearch	48%	0%
7	httpunit	0%	95%	Graveyard	83%	0%	MMOpenGraph	43%	0%
8	WebCQ	0%	95%	JavaTextEditor	82%	0%	AppletServer	0%	41%
9	WebXSSDetector	0%	95%	Eclipse Edit	81%	0%	MultiJADS	0%	39%
10	Organism System	0%	90%	Comic book edit	65%	15%	GalleryGrabber	0%	39%

Table 2.11: The top ten applications returned by Exemplar for three separate queries, along with the WOS and RAS scores for each. The DCS score was zero in every case. Note: Ties of relevance scores are broken randomly; applications with identical scores may appear in a different order.

We observed instances during the case study where the negative correlation between WOS and RAS improved the final search results. Consider Task 2 from Section 2.5.5. For this task, one programmer entered the query *find replace string text files* into Exemplar (see Table 2.11). The first result was a program called RText, which is a programmer's text editor with find/replace functionality. The second result was Nodepublisher, a content management system for websites. Nodepublisher's high-level description did not match the query and has a WOS score of 0%. The query did match several API call descriptions, including calls inside the class `java.text.DictionaryBasedBreakIterator`[19] which Nodepublisher uses. Conversely, RText contained no API calls with documentation matching the query, but had a relevant high-level description. Since both applications were rated as highly-relevant by the programmer in the case study, both WOS and RAS aided in finding a relevant result for this query. Specific situations such as this one support our decision to keep WOS in the final version of Exemplar, even with a reduced weight (see Section 2.7.2.2). Not all applications with high WOS or RAS scores were relevant, however. Despite occurring in the top ten list of applications, both MMOpenGraph and AppletServer were rated with a confidence level of 2 ("mostly irrelevant") by the author of the query.

2.10 Related Work

Different code mining techniques and tools have been proposed to retrieve relevant software components from different repositories as it is shown in Table 2.12. CodeFinder iteratively refines code repositories in order to improve the precision of returned software components [37]. Codefinder finds similar code using spreading activation based on the terms that appear in that code. Exemplar

[19]The documentation for this API class can be found at: `http://www.docjar.com/docs/api/java/text/DictionaryBasedBreakIterator.html` (verified 03/28/2011)

Approach	Granularity		Corpora	Query Expansion
	Search	Input		
CodeFinder [37]	M	C	D	Yes
CodeBroker [116]	M	C	D	Yes
Mica [106]	F	C	C	Yes
Prospector [65]	F	A	C	Yes
Hipikat [21]	A	C	D,C	Yes
xSnippet [92]	F	A	D	Yes
Strathcona [40][41]	F	C	C	Yes
AMC [38]	F	C	C	No
Google Code	F,M,A	C,A	D,C	No
Sourceforge	A	C	D	No
SPARS-J [45][46]	M	C	C	No
Sourcerer [58]	F,M,A	C	C	No
Sourcerer API Search [5]	F	C,A	C	No
CodeGenie [56]	F,M	T	C	No
SpotWeb [109]	M	C	C	Yes
ParseWeb [108]	F	A	C	Yes
S^6 [87]	F	C,A,T	C	Manual
Krugle	F,M,A	C,A	D,C	No
Koders	F,M,A	C,A	D,C	No
SNIFF [15]	F,M	C,A	D,C	Yes
Blueprint [10]	F	C,A	C	No
Exemplar [68]	F,M,A	C,A	D,C	No

Table 2.12: Comparison of Exemplar with other related approaches. Column `Granularity` specifies how search results are returned by each approach (**F**ragment of code, **M**odule, or **A**pplication), and how users specify queries (**C**oncept, **A**PI call, or **T**est case). The column `Corpora` specifies the scope of search, i.e., **C**ode or **D**ocuments, followed by the column `Query Expansion` that specifies if an approach uses this technique to improve the precision of search queries.

is different in that we locate source code based on keywords from API documentation. It is not necessary for Exemplar to find any matching keywords in the source code itself.

Codebroker system uses source code and comments written by programmers to query code repositories to find relevant artifacts [115]. Unlike Exemplar, Codebroker is dependent upon the descriptions of documents and meaningful names of program variables and types, and this dependency often leads to lower precision of returned projects.

Even though it returns code snippets rather than applications, Mica is similar to Exemplar since

it uses help pages to find relevant API calls to guide code search [106]. However, Mica uses help documentation to refine the results of the search while Exemplar uses help pages as an integral instrument in order to expand the range of the query.

SSI examines the API calls made in source code in order to determine the similarity of that code [58]. SSI indexes each source code element based on the identifier names and comments in that code. Then SSI adds terms to the index of a source element. The new terms come from other source code elements which use the same set of API calls. Additionally, SSI seeds the index with keywords from API call documentation. On the other hand, Exemplar matches query keywords directly to API documentation, and then calculates RAS, which is a ranking based on which projects uuse the API calls that the matching documentation describes. The fundamental difference between Exemplar and SSI is that Exemplar bases its ranking on how many relevant API calls appear in the source code (RAS, Section 3.3), unlike SSI, which ranks source code based on the keyword occurrences in the source code. Also, Exemplar has been evaluated with a user-study of professional programmers.

SNIFF extends the idea of using documentation for API calls for source code search [33][106] in several ways [15]. After retrieving code fragments, SNIFF then performs intersection of types in these code chunks to retain the most relevant and common part of the code chunks. SNIFF also ranks these pruned chunks using the frequency of their occurrence in the indexed code base. In contrast to SNIFF [15], MICA [106], and our original MSR idea [33], we evaluated Exemplar using a large-scale case study with 39 programmers to obtain statistically significant results, we followed a standard IR methodology for comparing search engines, and we return fully executable applications. Exemplar's internals differ substantially from previous attempts to use API calls for searching, including SNIFF: our search results contain multiple levels of granularity, we conduct a thorough comparison with the state of art search engine using a large body of Java application code,

and we are not tied to a specific IDE.

Prospector is a tool that synthesizes fragments of code in response to user queries that contain input types and desired output types [65]. Prospector is an effective tool to assist programmers in writing complicated code, however, it does not provide support for a full-fledged code search engine.

Keyword programming is a technique which translates a few user-provided keywords into a valid source code statement [59]. Keyword programming matches the keywords to API calls and the parameters of those calls. Then, it links those parameters to variables or other functions also mentioned in the keywords. Exemplar is similar to keyword programming in that Exemplar matches user queries to API calls, and can recommend usage of those calls. Unlike keyword programming, Exemplar show examples of previous usage of those APIs, and does not attempt to integrate those calls into the user's own source code.

The Hipikat tool recommends relevant development artifacts (i.e., source revisions associated with a past change task) from a project's history to a developer [21]. Unlike Exemplar, Hipikat is a programming task-oriented tool that does not recommend applications whose functionalities match high-level requirements.

Strathcona is a tool that heuristically matches the structure of the code under development to the example code [40][39]. Strathcona is beneficial when assisting programmers while working with existing code, however, its utility is not applicable when searching for relevant projects given a query containing high-level concepts with no source code.

There are techniques that navigate the dependency structure of software. Robillard proposed an algorithm for calculating program elements of likely interest to a developer [89][90]. FRAN is a technique which helps programmers to locate functions similar to given functions [96]. Finally,

XSnippet is a context-sensitive tool that allows developers to query a sample repository for code snippets that are relevant to the programming task at hand [92]. Exemplar is similar to these algorithms in that it uses relations between API calls in the retrieved projects to compute the level of interest (ranking) of the project. Unlike these approaches, Exemplar requires only a natural language query describing a programming task. We found in this paper that considering the dataflow among API calls does not improve the relevancy of results in our case.

Existing work on ranking mechanisms for retrieving source code are centered on locating components of source code that match other components. Quality of match (QOM) ranking measures the overall goodness of match between two given components [107], which is different from Exemplar which retrieves applications based on high-level concepts that users specify in queries. *Component rank model (CRM)* is based on analyzing actual usage relations of the components and propagating the significance through the usage relations [45][46]. Yokomori et al. used CRM to measure the impact of changes to frameworks and APIs [117]. Unlike CRM, Exemplar's ranking mechanism is based on a combination of the usage of API calls and relations between those API calls that implement high-level concepts in queries.

S^6 is a code search engine that uses a set of user-guided program transformations to map high-level queries into a subset of relevant code fragments [87], not complete applications. Like Exemplar, S^6 returns source code, however, it requires additional low-level details from the user, such as data types of test cases.

2.11 Conclusions

We created Exemplar, a search engine for highly relevant software projects. Exemplar searches among over 8,000 Java applications by looking at the API calls used in those applications. In evaluating our work, we showed that Exemplar outperformed SourceForge in a case study with 39 professional programmers. These results suggest that the performance of software search engines can be improved if those engines consider the API calls that the software uses. Also, we modified Exemplar to increase the weight of RAS, and performed a second case study evaluating the effects of this increase. We found that not only does including API call information increase the relevance of the results, but it also improves the ordering of the results. In other words, Exemplar places the relevant applications at the top of list of results.

Chapter 3

Detecting Similar Software Applications

3.1 Introduction

Retrieving similar or related web pages is a feature of popular search engines (e.g., Google, Ask.com, HotBot). After users submit search queries, Google displays links to relevant web pages along with a link labeled `Similar` next to each result. These `Similar` links point to web pages that the Google similarity algorithm computes by aggregating many factors that include, but are not limited to, the popularity scores of the retrieved pages, links among the pages, and the links' positions and sizes [31]. For example, for the main ACM SigSoft page, Google returns three top similar web sites: IEEE Computer Society, Software Engineering Institute, and ESEC/FSE 2009[1].

Detecting similar applications is a notoriously difficult problem, since it means automatically detecting that high-level requirements for these applications match semantically [44, pages 74,80][62]. This situation is aggravated by the fact that many application repositories are polluted with poorly functioning projects [42]; a match between words in requirement documents with words in the descriptions or in the source code of applications does not guarantee that these applications are relevant to the requirements. Applications may be highly-similar to one another at a low-level of the

[1] Last time checked: September 20, 2011.

implementations of some functions even if they do not perform the same high-level functionality [29]. Rarely do programmers record any traceability links between software artifacts, which belong to different applications, to establish their functional similarity.

Knowing similarity between applications plays an important role in assessing reusability of these applications, improving understanding of source code, rapid prototyping, and discovering code theft and plagiarism [51, 60, 72, 91, 98]. Enabling programmers to compare automatically how different applications implement the same requirements greatly contributes to knowledge acquisition about these requirements and subsequently to decisions that these developers make about code reuse. Retrieving a list of similar applications provides a faster way for programmers to concentrate on relevant aspects of functionality, thus saving time and resources for programmers. Programmers can spend this time understanding specific aspects of functionality in similar applications, and see the complete context in which the functionality is used.

A fundamental problem of detecting closely related applications is in the mismatch between the high-level intent reflected in the descriptions of these applications and low-level implementation details. This problem is known as the *concept assignment problem* [9]. For any two applications it is too imprecise to establish their similarity by simply matching words in the descriptions of these applications, comments in their source code, and the names of program variables and types. Since programmers typically invest a significant intellectual effort (i.e., they need to overcome a high cognitive distance [52]) to understand whether retrieved applications are similar, existing code search engines do not alleviate the task of detecting similar applications because they return only a large number of different code snippets.

We created a novel approach for detecting *Closely reLated ApplicatioNs (CLAN)*. This paper makes the following contributions:

- A major contribution of our approach is that CLAN uses complete software applications as input, not only natural language queries. This feature is useful when a developer needs to find similar applications to a known software application.

- We introduce a new abstraction that is relevant to *semantic spaces* [43] that are modeled as existing inheritance hierarchies of *Application Programming Interface (API)* classes and packages.

- We extended a well-established conceptual framework of relevance with our new abstraction. The intuition behind our approach is that if two applications contain functional abstractions in a form of inheritance hierarchies and packages that contain API calls whose semantics are defined precisely, and these calls implement the same requirement (e.g., different API calls from a data compression library), then these applications have a higher degree of similarity than those that do not have API calls that are related to some requirement. The idea of using API calls to improve code search was proposed and implemented elsewhere [15, 33, 68]; however, this idea has never been used to compute similarities between software applications.

- Based on this extension, we designed a novel algorithm that computes a similarity index between Java applications, and we implemented this algorithm in CLAN and applied to 8,310 Java applications that we downloaded from Sourceforge. CLAN is available for public use[2].

- We conducted an experiment with 33 Java programmers to evaluate CLAN. The results show with strong statistical significance that users find more relevant applications with higher precision with CLAN than those based on the closest competitive approach MUDABlue[3] [50]

[2]http://www.javaclan.net
[3]http://www.mudablue.net

and a system that combines CLAN and MUDABlue that we implemented[4].

3.2 Our Hypothesis And The Problem

In this section we use a conceptual framework for relevance to define the concept of similarity between applications, formulate a hypothesis, and describe problems that we should solve to test this hypothesis.

3.2.1 A Motivating Scenario

A motivating scenario for detecting similar application is based on a typical project lifecycle in Accenture, a global software consulting company with over 250,000 employees as of February, 2012. At any given time, company consultants are engaged in over 3,000 software projects. Since its first project in 1953, Accenture's consultants delivered tens of thousand of projects, and many of these projects are similar in requirements and their implementations. Knowing the similarity of these applications is important for preserving knowledge, experience, winning bids on future projects, and successfully building new applications.

A typical lifecycle of a large-scale project involves many stages that start with writing a proposal in response to a bid from a company that needs an application. A major part of writing a proposal and developing a prototype is to elicit requirements from different stakeholders. There are quite a few competing companies for each bid: IBM Corp, HP Corp, Tata Consultancy Services to name a few. A winning bid proposal has many components: well-elicited requirements, preliminary models and design documents, proof of experience of building and delivering similar applications in the

[4]http://www.clancombined.net

past. Clearly, a company that submits a bid proposal that contains these components as closely matching a desired application as possible, will win the bid.

It is important to reuse these components from successfully delivered applications in the past - doing so will save time and resources and increase chances of winning the bid. It is shown that over a dozen different artifacts can be successfully reused from software applications [48, pages 3–5]. The process of finding similar applications starts with code search engines that return code fragments and documents in response to queries that contain key words from elicited requirements. However, returned code fragments are of little help when many other non-code artifacts are required (e.g., different (non)functional requirements documents, UML models, design documents).

Matching words in queries against words in documents and source code is a good starting point, however, it does not help stakeholders to establish how applications are similar at a bigger scale. In this paper, we refer *application* as a collection of all source code modules, libraries, and programs that, when compiled, result in the final deliverable that customers install and use to accomplish certain business functions. Applications are usually accompanied by non-code artifacts, which are important for the bidding process. Establishing their similarity at large from different similar components of the source code is a goal of this paper.

The concept of similarity between applications is integrated in the software lifecycle process as follows. After obtaining the initial set of requirements, the user enters keywords that represent these requirements into a search engine that returns relevant applications that contain these keywords. In practice, it is unlikely that the user finds an application that perfectly matches all the requirements - if it happens, then the rapid prototyping process is finished. Otherwise, the user takes the returned applications and studies them to determine how relevant they are to the requirements.

After examining some returned application, the user determines what artifacts are relevant to

requirements, and which ones are missing. At this point the user wants to find similar applications that contain the missing artifacts while retaining similarity to the application that the user has found. That is, using the previously found application, the initial query is further expanded to include artifacts from this application that matched some of requirements as the user determined, and similar applications would contain artifacts that are similar to the ones in the found application.

3.2.2 Similarity Between Applications

We define the meaning of similarity between applications by using Mizzaro's well-established conceptual framework for relevance [73, 74]. In Mizzaro's framework, similar documents are relevant to one another if they share some common concepts. Once these concepts are known, a corpus of documents can be clustered by how documents are relevant to these concepts. Subsequently all documents in each cluster will be more relevant to one another when compared to documents that belong to different clusters. This is the essence of the cluster hypothesis that specifies that documents that cluster together tend to be relevant to the same concept [110].

Two applications are similar to each other if they implement some features that are described by the same abstraction. For example, if some applications use cryptographic services to protect information then these applications are similar to a certain degree, even though they may have other different functionalities for different domains. Another example is text editors that are implemented by different programmers, but share many features: copy and paste, undo and redo, saving data in files using standard formats. A straightforward approach for measuring similarity between applications is to match the names of their program variables and types. The precision of this approach depends highly on programmers choosing meaningful names that reflect correctly the concepts or abstractions that they implement, but this compliance is generally difficult to enforce [3].

3.2.3 Our Hypothesis

In Mizzaro's framework, a key characteristic of relevance is how information is represented in documents. We concentrate on *semantic anchors*, which are elements of documents that precisely define the documents' semantic characteristics. Semantic anchors may take many forms. For example, they can be expressed as links to web sites that have high integrity and well-known semantics (e.g., cnn.com or whitehouse.gov) or they can refer to elements of semantic ontologies that are precisely defined and agreed upon by different stakeholders.

This is the essence of *paradigmatic associations* where documents are considered similar if they contain terms with high semantic similarities [86]. Our hypothesis is that by using semantic anchors and dependencies among them it is possible to compute similarities between documents with a higher degree of accuracy when compared to documents that have no commonly defined semantic anchors in them.

Without semantic anchors, documents are considered as bags of words with no semantics, then the relevance of these documents to user queries and to one another can be determined by matches between these words. This is the essence of *syntagmatic associations* where documents are considered similar when terms (i.e., words) in these documents occur together [86]. For example, the similarity engine MUDABlue uses syntagmatic associations for computing similarities among applications [50]. The problem with this approach is that computed relevance is relatively imprecise when compared with CLAN as we show in Section 3.5.

3.2.4 Semantic Anchors in Software

Since programs contain API calls with precisely defined semantics, these API calls can serve as semantic anchors to compute the degree of similarity between applications by matching the semantics

of these applications that is expressed with these API calls. Programmers routinely use API calls from third-party packages (e.g., the *Java Development Kit (JDK)*) to implement various requirements [15, 25, 33, 68, 106]. API calls from well-known and widely used libraries have precisely defined semantics unlike names of program variables and types and words that programmers use in comments. In this paper, we use API calls as semantic anchors to compute similarities among applications.

3.2.5 Challenges

Our hypothesis is based on our idea that it is better to compute similarity between programs by utilizing API calls as semantic anchors that come from JDK and that programmers use to implement various requirements. This idea has advantages over using *Vector Space Model (VSM)* where documents are represented as vectors of words and a similarity measure is computed as the cosine between these vectors [95]. One main problem with VSM is that different programmers can use the same words to describe different requirements (i.e., the synonymy problem) and they can use different words to describe the same requirements (i.e., the polysemy problem). This problem is a variation of the vocabulary problem, which states that "no single word can be chosen to describe a programming concept in the best way" [28]. This problem is general to *Information Retrieval (IR)*, but somewhat mitigated by the fact that different programmers who participate in the projects use coherent vocabularies to write code and documentation, thus increasing the chance that two words in different applications may describe the same requirement.

The sheer number of API calls suggests that many of these calls are likely to be shared by different programs that implement completely different requirements leading to significant imprecision in calculating similarities. Our study shows that out of 2,080 randomly chosen Java programs in

Sourceforge, over 60% of these programs use `String` objects and over 80% contain collection objects; these programs invoke API calls that these string and collection classes exports [35]. If similarity scores are computed based on these common API calls, most Java programs would be similar to one another. On top of that, it is not computationally feasible to compute similarity scores with high precision for hundreds of thousands of API calls. It is an instance of a problem known as *the curse of dimensionality*, which is a problem caused by the exponential increase in processing by adding extra dimensions to a representational space [84].

Graphically, programs are represented as dots in a multidimensional space where dimensions are API calls and coordinates in this space reflect the numbers of API calls in programs. The JDK contains close to 115,000 API calls that are exported by a little more than 13,000 classes and interfaces that are contained in 721 packages. Computing similarity scores between programs using VSM in a space with hundreds of thousands of dimensions is not always computationally feasible, it is imprecise, and difficult to interpret. We need to reduce the dimensionality of this space while simultaneously revealing similarities between implemented latent high-level requirements.

3.3 Our Approach

In this section we describe our key idea, provide background material on LSI that we use in CLAN, and explain its architecture.

3.3.1 Key Idea

Our key idea is threefold. First, if two applications share some semantic anchors (e.g., API calls), then their similarity index should be higher than for applications that do not share any semantic

anchors. Sharing semantic anchors means more than the exact syntactic match between the same two API calls; it also means that two different API calls will match semantically if they come from the same class or package. This idea is rooted in the fact that classes and packages in JDK contain semantically related API calls; for example, the package `java.security` contains classes and API calls that enable programmers to implement security-related requirements, and the package `java.util.zip` exports classes that contain API calls for reading and writing the standard `ZIP` and `GZIP` file formats. Thus we exploit relationships between inheritance hierarchies in the JDK to improve the precision of computing similarity. This idea is related to semantic spaces where concepts are organized in structured layers and similarity scores between documents are computed using relations between layers [43]. Moreover, recent work has shown that API classes and packages can be used to categorize software applications using those classes and packages [71].

Second, different API calls have different weights. Recall that many applications have many API calls that deal with collections and string manipulations. Our idea is to automatically assign higher weights to API calls that are encountered in fewer applications and, conversely to assign lower weights to API calls that are encountered in a majority of applications. There is no need to know what API calls are used in applications – this task should be done automatically. Doing it will improve the precision of our approach since API calls that come from common packages like `java.lang` will have less impact to skew the similarity index.

Finally, we observed that a requirement is often implemented using combinations of different API calls rather than a single API call. It means that co-occurrences of API calls in different applications form patterns of implementing different requirements. For example, a requirement of efficiently and securely exchanging XML data is often implemented using API calls that read XML data from a file, compress and encrypt it, and then send this data over the network. Even though

different ways of implementing this requirement are possible, detecting patterns in co-occurrences of API calls and using these patterns to compute the similarity index may lead to higher precision when compared with competitive approaches.

3.3.2 Latent Semantic Indexing (LSI)

To implement our key idea we rely an IR technique called *Latent Semantic Indexing (LSI)* that reduces the dimensionality of the similarity space while simultaneously revealing latent concepts that are implemented in the underlying corpus of documents [24]. In LSI, terms are elevated to an abstract space, and terms that are used in similar contexts are considered similar even if they are spelled differently. LSI automatically makes embedded concepts explicit using *Singular Value Decomposition (SVD)*, which is a form of factor analysis used to reduce dimensionality of the space to capture most essential semantic information.

The input to SVD is an $m \times n$ *term document matrix (TDM)*. Each of m rows corresponds to a unique term, which in our case is either a class or a package name that contains a corresponding API call that is invoked in a corresponding application (i.e., document). Columns correspond to unique documents, which in our case are Java applications. Each element of the TDM contains the weight that shows how frequently this API call is used in this application when compared to its usage in other applications[5]. We cannot use a simple metric such as the API call count since it is biased – it shows the number of times a given API call appears in applications, thus skewing the distribution of these calls toward large applications, which may have a higher API call count regardless of the actual importance of that API call.

[5]Note that we do not consider the number of times each API call is executed, e.g., in a loop. Instead, we count occurrences of API calls in source code.

SVD decomposes TDM into three matrices using a reduced number of dimensions, r, whose value is chosen experimentally. The number of dimensions for LSI is commonly chosen $r = 300$ [24, 83]. One of these matrices contains document vectors that describe weights that documents (i.e., applications) have for different dimensions. Each column in this matrix is a vector whose elements specify coordinates for a given application in the r–dimensional space. Computing similarities between applications means computing the cosines between vectors (i.e., rows) of this matrix.

3.3.3 CLAN Architecture and Workflow

The architecture for CLAN is shown in Figure 3.1. The main elements of the CLAN architecture are the Java Applications (Apps Archive) and the API call Archive, the Metadata Extractor, the Search Engine, the LSI Algorithm, and the Term Document Matrix (TDM) Builder. In TDM, rows represent packages or classes that contain JDK API calls that are invoked in Java applications and columns represent Java applications. Applications metadata describes different API calls that are invoked in the applications and their classes and packages. The input to CLAN (i.e., a user query) is shown in Figure 3.1 with a thick solid arrow labeled `(9)`. The output is shown with the thick dashed arrow labeled `(12)`.

CLAN works as follows. The Metadata Processor takes as its inputs `(1)` the Apps Archive with Java applications and API archive that contains descriptions of JDK API calls. The Metadata Processor outputs `(2)` the Application Metadata, which is the set of tuples $<<<$`package`, `class` $>$, `API call`$>$,$A >$ linking API calls and their packages and classes to Java applications A that use these API calls.

Term-Document Matrix (TDM) Builder takes `(3)` Application Metadata as its input, and it uses this metadata `(4)` to produce two TDMs: Package-Application Matrix ($\texttt{TDM}_\texttt{P}$) and Class-

Application Matrix ($\texttt{TDM}_\texttt{C}$) that contain TFIDFs for JDK packages and classes whose API calls are invoked in respective applications. The LSI Algorithm is applied (5) separately to $\texttt{TDM}_\texttt{P}$ and $\texttt{TDM}_\texttt{C}$ to compute (6) class and package matrices $\|C\|$ and $\|P\|$. That is, each row in these matrices contain coordinates that represent its corresponding application in a multidimensional space with respect to either classes or packages of API calls that are invoked in this application.

Class-level and package-level similarities are different since applications are often more similar on the package level than on the class level because there are fewer packages than classes in the JDK. Therefore, there is the higher probability that two applications may have API calls that are located in the same package but not in the same class.

Matrices $\|C\|$ and $\|P\|$ are combined (7) into the Similarity Matrix using the following formula $\|S\| = \lambda_C \cdot \|S\|_C + \lambda_P \cdot \|S\|_P$, where λ is the interpolation weight for each similarity matrix, and matrices $\|S\|_C$ and $\|S\|_P$ are similarity matrices for $\|C\|$ and $\|P\|$ respectively. These similarity matrices are obtained by computing the cosine between the vector for each application (i.e., a corresponding row in the matrix) and vectors for all other applications. Weights λ_P and λ_C are determined independently of applications. Adjusting these weights enables experimentation with how underlying structural and textual information in application affects resulting similarity scores. In this paper we selected $\lambda_C = \lambda_P = 0.5$, thus stating that class and package-level scores contribute equally (8) to the Similarity Matrix.

The Similarity Matrix, $\|S\|$ is a square matrix whose rows and columns designate applications. For any two applications A_i and A_j, each element of $\|S\|$, S_{ij} is the similarity score between these applications that is defined as follows: $S_{ij} = \begin{cases} 0 \le s \le 1, & \texttt{if } i \neq j, \\ 1, & \texttt{if } i = j \end{cases}$.

It took us close to three hours to construct the TDM for MUDABlue using Intel Xeon CPU

W3540, 2.93GHz with 2GB RAM, less than one hour for TDM for the package- and class-level TDMs for CLAN. Running SVD on these TDMs took less than three hours for MUDABlue, and less than 30 minutes for the package- and class-level TDMs for CLAN. For all three TDMs, we used the same corpus of 8,310 Java projects from SourceForge with 114,146 API calls.

When the user enters a query `(9)`, it is passed to the Search Engine that retrieves relevant applications `(10)` with ranks in the descending order using the Similarity Matrix. In addition, the Search Engine uses the Application Metadata `(11)` to extract a map of API calls for each pair of similar applications. This map shows API calls along with their classes and packages that are shared by similar applications, and this map is given to the user `(12)`.

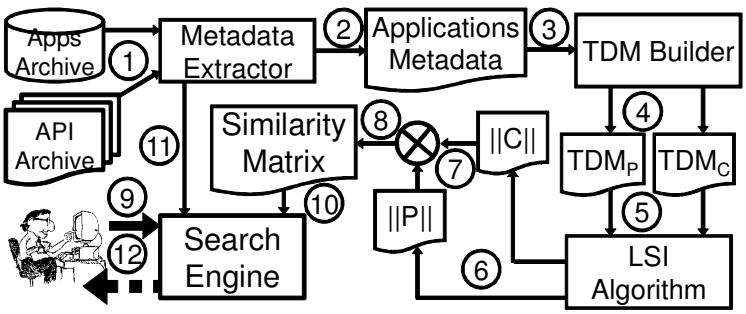


Figure 3.1: CLAN architecture and workflow.

3.4 Experimental Design

Typically, search and retrieval engines are evaluated using manual relevance judgments by experts [66, pages 151-153]. To determine how effective CLAN is, we conducted an experiment with 33 participants who are Java programmers. Our goal is to evaluate how well these participants can find similar applications to the ones that are considered highly relevant to given tasks using three different similarity engines: MUDABlue, CLAN, and an integrated similarity engine that combines

MUDABlue and CLAN.

3.4.1 Background on MUDABlue and Combined

MUDAblue is the closest relevant work to CLAN since it provides automatic categorization for applications [50]. The *cluster hypothesis* specifies that documents that cluster together tend to be relevant to the same concept [110]. To the best of our knowledge, there is no other system that is competitive to CLAN in that it finds similar applications. We faithfully reimplemented MUDABlue for our experiment as it is described in the original paper [50].

The original MUDABlue was implemented and evaluated on a small repository of 41 C applications that were selected from five different categories from Sourceforge. Comparing two similarity search engines that do not work with the same code base or different granularity levels (i.e., applications vs. code fragments) might introduce considerable threats to validity. Sourceforge has a popular search engine and contains a large Java repository online; Apps Archive is populated with all Java projects from this repository, and we applied MUDABlue as baseline approach to this archive thus making its set of applications comparable with those of CLAN.

Since Similarity Matrices of MUDABlue and CLAN have the same dimensions, it is possible to construct a combined matrix whose values are the average of the values of the MUDABlue and CLAN matrix elements at the corresponding position. The intuition behind this combined approach lies in integrating two approaches: MUDABlue where every word in the source code of applications is taken into consideration versus the CLAN approach where only API calls with precisely defined semantics are considered. A research question is whether this integration produces a superior result when compared to each of the constituent approaches. Experimenting with this combined Similarity Matrix allows us to seek an answer to this question about the benefit of the combined approach.

3.4.2 Methodology

We used a cross validation study design in a cohort of 33 participants who were randomly divided into three groups. The study was sectioned in three experiments in which each group was given a different engine to find similar applications to the ones that we provided for given tasks. Each participant used a different task in each experiment. Participants translated tasks into key words, searched for relevant applications using a code search engine, and selected an application that matched their key words the best. We call this application *the source application*. Then a similarity engine returned a list of top ten *target applications* that were most similar to the source application. Thus each participant used each subject engine on different tasks and different applications in this experiment. Before the experiment we gave a one-hour tutorial on using these engines to find similar applications.

The next step was to examine the retrieved applications and to determine if they are relevant to the tasks and the source application. Each participant accomplished this step individually, assigning a confidence level, C, to the examined applications using a four-level Likert scale. Since this examination is time consuming, manual and laborious we asked participants to examine only top ten applications that resulted from searches.

The guidelines for assigning confidence levels are the following.

1. Completely dissimilar - there is absolutely nothing in the target application that the participant finds similar to the source application, nothing in it is related to the task and the functionality of the subject application.

2. Mostly dissimilar - only few remotely related requirements are located in source and target application.

3. Mostly similar - a somewhat large number of implemented requirements are located in the target application that are similar to ones in the source application.

4. Highly similar - the participant is confident that the source and the target applications share the same semantic concepts expressed in the task.

All participants were computer science students from the University of Illinois at Chicago who had at least six months of Java experience. Twelve participants were upper-level undergraduate students, and the other 21 participants were graduate students. Out of 33 participants, 15 had programming experience with Java ranging from one to three years, and 11 participants reported more than three years of experience writing programs in Java. Sixteen participants reported prior experience with search engines, and eight said that they never used code search engines before.

3.4.3 Precision

Two main measures for evaluating the effectiveness of retrieval are precision and recall [114, page 188-191]. The precision, $P_r = \frac{\texttt{\# of retrieved applications that are similar}}{\texttt{total \# of retrieved applications}}$, i.e., the precision of a ranking method is the fraction of the top r ranked target applications that are relevant to the source application, where $r = 10$ in this experiment, which means that each similarity engine returned top ten similarity matches. Relevant or similar applications are counted only if they are ranked with the confidence levels `4` or `3`. The precision metrics reflects the accuracy of the similarity search. Since we limit the investigation of the retrieved applications to top ten, the recall is not measured in this study.

We created the variable precision, P as a categorization of the response variable confidence, C. We did it for two reasons: improve discrimination of subjects in the resulting data and additionally

validate statistical evaluation of results. Precision, P imposes a stricter boundary on what is considered reusable code. For example, consider a situation where one participant assigns the level two to all returned applications, and another participant assigns level three to half of these applications and level one to the other half. Even though the average of $C = 2$ in both cases, the second participant reports much higher precision, $P = 0.5$ while the precision that is reported by the first participant is zero. Achieving statistical significance with a stricter discriminative response variable will give assurance that the result is not accidental.

3.4.4 Hypotheses

We introduce the following null and alternative hypotheses to evaluate how close the means are for the Cs and Ps for control and treatment groups, where C and P are the confidence level and the precision respectively. Unless we specify otherwise, participants of the treatment group use either MUDABlue or Combined approaches, and participants of the control group use CLAN. We evaluate the following hypotheses at a 0.05 level of significance.

H_0 The primary null hypothesis is that there is no difference in the values of confidence level and precision per task between participants who use MUDABlue, Combined, and CLAN.

H_1 An alternative hypothesis to H_0 is that there is statistically significant difference in the values of confidence level and precision between participants who use MUDABlue, Combined, and CLAN.

Once we test the null hypothesis H_0, we are interested in the directionality of means, μ, of the results of control and treatment groups. We are interested to compare the effectiveness of CLAN

(CN) versus the MUDABlue (MB) and Combined (MC) with respect to the values of confidence level, *C*, and precision, *P*.

H1: C of CLAN versus MUDABlue.

H2: P of CLAN versus MUDABlue.

H3: C of CLAN versus Combined.

H4: P of CLAN versus Combined.

H5: C of MUDABlue versus Combined.

H6: P of MUDABlue versus Combined.

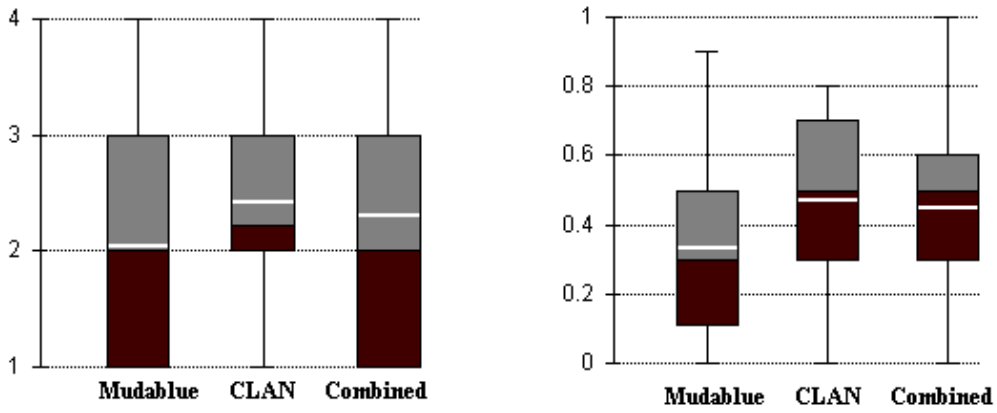
The rationale behind the alternative hypotheses to `H1` and `H2` is that CLAN allows users to quickly understand why applications are similar by reviewing visual maps of their common API calls, classes, and packages. The alternative hypotheses to H3 and H4 are motivated by the fact that if all words from source code are used in the analysis in addition to API calls, it will worsen the precision with which users evaluate retrieved similar applications. Finally, having the alternative hypotheses to `H5` and `H6` ensures that the Combined approach still allows users to quickly understand how similar applications share the same semantic concepts using their common API calls, classes, and packages.

3.4.5 Task Design

We designed 36 tasks that participants work on during experiments in a way that these tasks belong to domains that are easy to understand, and they have similar complexity. The authors of this paper visited various programming forums and internet groups to extract descriptions of tasks from the

questions that programmers asked. In addition, we interviewed a dozen programmers at Accenture who explained what tasks they worked on in the past year.

Additional criterion for these tasks is that they should represent real-world programming tasks and should not be biased towards any of the similarity search engines that are used in this experiment. Descriptions of these tasks should be flexible enough to allow participants to find different matching applications for similarity search. This criterion significantly reduces any bias towards evaluated similarity search engines. These tasks and the results of the experiment are available for download[6].



(a) Confidence level, C. (b) Precision, P.

Figure 3.2: Statistical summary of the results of the experiment for C and P.

3.4.6 Tasks

The following two tasks are examples from the set of 36 tasks we used in our experiment.

- Create an application for sharing, viewing, and exploring large data sets that are encoded using MIME. The data sets may represents blogs or genom sequences. The data can be stored

[6]http://www.javaclan.net, follow the Experiment link.

using key value pairs. The application should support retrieving data items by mapping keys to values.

- Implement a library for checking XPath expressions. The checker should support compiling XPath expressions, evaluating XPath expressions in the context of the specified XML document and returning the results as the specified type.

3.4.7 Threats to Validity

In this section, we discuss threats to the validity of this experimental design and how we address and minimize these threats.

3.4.7.1 Internal Validity

Participants. Since evaluating hypotheses is based on the data collected from participants, we identify three threats to internal validity: Java proficiency, motivation, and the uniformity among participants.

Even though we selected participants who had working knowledge of Java, we did not conduct an independent assessment of how proficient these participants were in Java. The danger of having poor Java programmers as participants of our experiment is that they can make poor choices of which retrieved applications have higher similarity to the source application. This threat is mitigated by the fact that all participants from UIC have documented experience working on course projects that required writing Java code, taking classes on programming with Java, and having experience working as Java programmers for commercial companies.

Tasks. Improper tasks pose a big threat to validity. If tasks are too general or trivial (e.g., open a file and read its data into memory), then every application that has file-related API calls

will be retrieved, thus inundating participants with results that are hard to evaluate. On the other hand, if application and domain-specific keywords describe a task (e.g., `astronomy` and `cosmic vacuum`), only a few applications will be retrieved that contain these keywords, thus creating a bias towards MUDABlue. To avoid this threat, we based the task descriptions on 12 specifications of different software systems that were written by different people including professional programmers at Accenture. While this diversification of tasks does not completely eliminate this threat to validity, it reduces it significantly.

3.4.7.2 External Validity

To make results of this experiment generalizable, we must address threats to external validity, which refer to the generalizability of a casual relationship beyond the circumstances of our experiment. The fact that supports the validity of this experimental design is that the participants are representative of professional Java programmers since some of them have already joined workforce and others will do soon. A threat to external validity concerns the usage of search tools in the industrial settings, where applications may not use third-party API call libraries. However, it is highly unlikely that modern large-scale software projects can be effectively developed, maintained, and evolved without this reuse.

3.5 Results

In this section, we report the results of the experiment and evaluate the null hypotheses.

Table 3.1: Results of t-tests of hypotheses, H, for paired two sample for means for two-tail distribution, for dependent variable specified in the column Var (either *C* or *P*) whose measurements are reported in the following columns. Extremal values, Median, Means (μ), variance (σ^2), degrees of freedom (DF), and the pearson correlation coefficient (PC), are reported along with the results of the evaluation of the hypotheses, i.e., statistical significance, p, and the T statistics. A decision to accept or reject the null hypothesis is shown in the last column Decision.

H	Var	Approach	Samples	Min	Max	Median	μ	σ^2	DF	PC	p	T	T_{crit}	Decision
H1	*C*	CLAN	304	1	4	2	2.42	1.14	321	0.1	$4.4 \cdot 10^{-7}$	5.02	1.97	Reject
		MUDABlue	322	1	4	1	2.03	1.13						
H2	*P*	CLAN	33	0	0.8	0.5	0.47	0.24	32	0.1	0.02	2.43	2.04	Reject
		MUDABlue	33	0	0.9	0.3	0.33	0.24						
H3	*C*	CLAN	304	1	4	2	2.42	1.14	321	0.1	0.11	1.6	1.96	Accept
		Combined	322	1	4	2	2.3	1.06						
H4	*P*	CLAN	33	0	0.8	0.5	0.47	0.24	32	0.16	0.68	0.41	2.04	Accept
		Combined	33	0	1	0.5	0.45	0.24						
H5	*C*	MUDABlue	322	1	4	1	2.03	1.13	321	-0.02	0.002	-3.16	1.97	Reject
		Combined	322	1	4	2	2.3	1.06						
H6	*P*	MUDABlue	33	0	0.9	0.3	0.33	0.24	32	0.21	0.04	-2.15	2.04	Reject
		Combined	33	0	1	0.5	0.45	0.24						

3.5.1 Results of Hypotheses Testing

We use one-way ANOVA and t-tests for paired two sample for means to evaluate the hypotheses that we stated in Section 3.4.4.

3.5.1.1 Variables

A main independent variable is the similarity engine (MUDABlue, CLAN, Combined) that participants use to find similar Java applications. Dependent variables are the values of confidence level, C, and precision, P.

3.5.1.2 Testing the Null Hypothesis

We used ANOVA to evaluate the null hypothesis H_0 that the variation in an experiment is no greater than that due to normal variation of individuals' characteristics and error in their measurement. The results of ANOVA confirm that there are large differences between the groups for C with $F = 11.7 > F_{crit} = 3$ with $p \approx 9.7 \cdot 10^{-6}$ which is strongly statistically significant. The mean C for the MUDABlue approach is 2.03 with the variance 1.12, which is smaller than the mean C for Combined, 2.3 with the variance 1.13, and it is smaller than the mean C for CLAN, 2.42 with the variance 1.08. Based on these results we can reject the null hypothesis and we accept the alternative hypothesis H_1.

However, the results of ANOVA confirm that there are insignificant differences between the groups for P with $F = 3.04 < F_{crit} = 3.09$ with $p = 0.052$. The mean P for the MUDABlue approach is 0.33 with the variance 0.06, which is smaller than the mean P for Combined, 0.45 with the variance 0.06, and it is smaller than the mean P for CLAN, 0.47 with the variance 0.057. Aggregating the values of C into P changes the results of the statistical test making it difficult to reach

a conclusion, and it requires more precise statistical tests, specifically, t-tests for paired two sample for means, which we describe below.

A statistical summary of the results of the experiment for *C* and *T* (median, quartiles, range and extreme values) are shown as box-and-whisker plots in Figure 3.2(a) and Figure 3.2(b) correspondingly with 95% confidence interval for the mean.

3.5.1.3 Comparing MUDABlue with CLAN

To test the null hypothesis H1 and H2 we applied two t-tests for paired two sample for means, for *C* and *P* for participants who used MUDABlue and CLAN. The results of this test for *C* and for *P* are shown in Table 3.1. The column `Samples` shows that the number of samples for CLAN is smaller than the obtained number of samples for MUDABlue because three participants missed one experiment. We replaced missing values with the average value for *C* for CLAN for this experiment. Based on these results we reject the null hypotheses H1 and H2, and we accept the alternative hypotheses that states that **participants who use CLAN report higher relevance and precision on finding similar applications than those who use MUDABlue**.

3.5.1.4 Comparing MUDABlue with Combined

To test the null hypotheses H5 and H6, we applied two t-tests for paired two sample for means, for *C* and *P* for participants who used the baseline MUDABlue and Combined. The results of this test for *C* and for *P* are shown in Table 3.1. Based on these results we accept the alternative hypotheses H5 and H6 that say that **participants who use Combined report higher relevance and precision on finding similar applications than those who use MUDABlue**.

3.5.1.5 Comparing CLAN with Combined

To test the null hypotheses H3 and H4, we applied two t-tests for paired two sample for means, for *C* and *P* for participants who used the baseline CLAN and Combined. The results of this test for *C* and for *P* are shown in Table 3.1. Based on these results we accept the null hypotheses H3 and H4 that say that **participants who use CLAN do not report higher relevance and precision on finding similar applications than those who use Combined**.

The result of comparing CLAN with Combined is somewhat surprising. We expected that combining two different methods of computing similarities would yield a better result than each of these methods alone. We have a possible explanation based on debriefing of the participants. After the experiment a few participants expressed confusion about using the Combined engine, which reported similar applications even though these applications had no common API calls, classes, or packages. Naturally, this phenomenon is a result of the MUDABlue's component of Combined that computes a high similarity score based on word occurrences while the CLAN's component provides a low score because of the absence of semantic anchors. At this point it is a subject of our future work to investigate this phenomenon in more detail. While combining CLAN and MUDABlue did not produce noticeable improvements, combining textual and structural information was successful for tasks of feature location [83] and detecting duplicate bug reports [113].

3.6 Discussion

During the experiment, programmers identified more relevant applications using CLAN than when using MUDABlue (see Section 3.5). This result points to a key advantage of CLAN: we help programmers effectively compare two applications by elevating highly-relevant details of these applica-

tions. Without CLAN, programmers must examine the whole source code of different applications in order to compare them. Consider the example in Figure 3.3. CLAN returned the application `mbox` as the most-similar application to `MidiQuickFix` for the task of recording music data into a MIDI file. CLAN marked these applications as similar because they share important elements of the API (e.g., `com.sun.media.sound`). For the same task, MUDABlue did not place `mbox` even in the top ten similar applications to `MidiQuickFix`. This example illustrates how CLAN improves over the state-of-the-art.

Figure 3.3: Part of the CLAN interface, showing the API calls common to two applications. CLAN shows these calls in order to help programmers concentrate on highly-relevant details when comparing applications.

3.7 Related Work

The five most related tools to our work are those based on CodeWeb by Michail and Notkin [72], MUDABlue by Kawaguchi et al. [50], Hipikat by Cubranic and Murphy [112] and CodeBroker by Ye and Fischer [115] and SSI by Bajracharya, Ossher, and Lopez [6]. CodeWeb is an automated approach for comparing and contrasting software libraries based on matching similar classes and

functions cross libraries (via name and similarity matching) [72]. This work was inspirational for us in extending the relevance framework with semantic anchors. In contrast to CodeWeb, CLAN also uses advanced dimensionality reduction techniques based on LSI and SVD and computes similarities among applications in the context of the complete software repository. SSI creates an index of code based on the keywords extracted from that code, and then expands that index with keywords from other code that uses the same API calls [6]. CLAN is different from SSI for three reasons: 1) CLAN locates the applications similar to a given application, and does not require a natural-language query, 2) CLAN is independent of the keywords chosen in the code, and 3) CLAN has been evaluated using a standard methodology with programmers against a state-of-the-art approach (MUDABlue).

Source code search engines have become an active research area in the recent years. While these approaches are different from CLAN we believe that majority of these approaches may benefit from the ideas implemented in CLAN. Among these source code engines are CodeFinder [37], Mica [106], Exemplar [68], SNIFF [15], Prospector [65], Suade [89], Starthcona [39], XSnippet [92], ParseWeb [108], SPARS-J [45], Portfolio [69], Sourcerer [7], S6 [87] and SpotWeb [109]. While none of these approaches retrieve similar applications to a given candidate software application, these approaches are effective in retrieving relevant software components from open source repositories.

Our previous work successfully uses the idea of functional abstraction in a search engine called Exemplar to find highly relevant applications. However, this idea has never been used to compute similarities between software applications. Unlike Exemplar, CLAN uses a novel combination of semantic layers that correspond to packages and class hierarchies, and based on our extension to Mizzaro's relevance framework we designed a novel algorithm based on LSI that computes a simi-

larity index between Java applications.

Other related approaches identify programs that are likely to share the same origin rely on dynamic analysis and known as API Birthmarks [98]. However, our approach uses static information and assumes that similar applications may have been implemented by different software developer teams. Likewise, software bertillonage is a technique for comparing software components based on the dependencies of those components [22]. Bertillonage is designed to locate duplicate code, however, and does not compute the similarity of software which may be related, but is not duplicated.

3.8 Conclusion

We created a novel search system for finding *Closely reLated ApplicatioNs (CLAN)* that helps users find similar or related applications. Our main contribution is in using a framework for relevance to design a novel approach that computes similarity scores between Java applications. We have built CLAN and we conducted an experiment with 33 participants to evaluate CLAN and compare it with the closest competitive approach, MUDABlue, and a system that combines CLAN and MUDABlue. The results show with strong statistical significance that CLAN finds similar applications with a higher precision than MUDABlue.

Chapter 4

Finding Relevant Functions and Their Usages In Millions of Lines of Code

4.1 Introduction

Different studies show that programmers are more interested in finding definitions of functions and their uses than variables, statements, or arbitrary fragments of source code [101]. More specifically, programmers use different tools including code search engines to answer three types of questions [100, 99]. First, programmers want to find initial focus points such as relevant functions that implement high-level requirements. Second, programmers must understand how a function is used in order to use it themselves. Third, programmers must see the chain of function invocations in order to understand how concepts are implemented in these functions. It is important that source code search engines support programmers in finding answers to these questions.

In general, understanding code and determining how to use it, is a manual and laborious process that takes anywhere from 50% to 80% of programmers' time [19, 23]. Short code fragments that are returned as results to user queries do not give enough background or context to help programmers determine how to reuse these code fragments, and programmers typically invest a significant intel-

lectual effort (i.e., they need to overcome a high cognitive distance [52]) to understand how to reuse these code fragments. On the other hand, if code fragments are retrieved as functions, it makes it easier for developers to understand how to reuse these functions.

A majority of code search engines treat code as plain text where all words have unknown semantics. However, applications contain functional abstractions that provide a basic level of code reuse, since programmer define functions once and call them from different places in source code. The idea of using functional abstractions to improve code search was proposed and implemented elsewhere [15, 68, 80, 106]; however, these code search engines do not automatically analyze how functions are used in the context of other functions, despite the fact that understanding the chains of function invocations is a key question that programmers ask. Unfortunately, existing code search engines do little to ensure that they retrieve code fragments in a broader context of relevant functions that invoke one another to accomplish certain tasks.

Our idea is that since programmers frequently ask various questions about functions, a code search engine should incorporate information about these functions that is used to answer the programmers' questions. Browsing retrieved functions that are relevant to queries means that programmers follow function calls and review declarations, definitions, and uses of these functions to combine them in a solution to a given task. That is, programmers want to accomplish the whole task quickly, rather than obtain multiple examples for different components of the task.

For example, consider the query "`record compress MIDI file`." Programmers don't want to just see examples that compress, and others that record, and others that manipulate MIDI files. A programmer wants to accomplish the complete task of recording and compressing a MIDI file. However, among relevant results there are functions that create and save MIDI files, functions that write data into a file in the MIDI format, and there are multiple functions that compress data.

Typically, programmers investigate these functions to determine which of them are relevant and determine how to compose these functions to achieve the goal that is expressed with the query. That is, a programmer wants to see code for the whole task of how to record MIDI data into a file and compress it. A search engine can support programmers efficiently if it incorporates in its ranking how these functions call one another, and displays that information to the user.

We created a code search system called *Portfolio* that supports programmers in finding relevant functions that implement high-level requirements reflected in query terms (i.e., finding initial focus points), determining how these functions are used in a way that is highly relevant to the query (i.e., building on found focus points), and visualizing dependencies of the retrieved functions to show their usages. Portfolio finds highly relevant functions in close to 270 Millions LOC in projects from FreeBSD Ports[1] by combining various *natural language processing (NLP)* and indexing techniques with *PageRank* and *spreading activation network (SAN)* algorithms. With NLP and indexing techniques, initial focus points are found that match key words from queries; with PageRank, we model the surfing behavior of programmers, and with SAN we elevate highly relevant chains of function calls to the top of search results. We have built Portfolio and conducted an experiment with 49 professional C++ programmers to evaluate Portfolio and compare it with the well-known and successful engines Google Code Search and Koders. The results show with strong statistical significance that users find more relevant code with higher precision with Portfolio than those with Google Code Search and Koders. To the best of our knowledge, we are not aware of any existing code search engines that have been evaluated against and shown to be more accurate than widely used commercial code search engines, with strong statistical significance and over a large codebase and using a standard information retrieval methodology [66, pages 151-153]. Portfolio is free and

[1] http://www.freebsd.org/ports

available for public use[2].

4.2 The Model

The search model of Portfolio uses a key abstraction in which the search space is represented as a directed graph with nodes as functions and directed edges between nodes that specify usages of these functions (i.e., a *call graph*). For example, if the function `g` is invoked in the function `f`, then a directed edge exists from the node that represents the function `f` to the node that represents the function `g`. Since the main goal of Portfolio is to enable programmers to find relevant functions and their usages, we need models that effectively represent the behavior of programmers when navigating a large graph of functional dependencies. These are navigation and association models that address surfing behavior of programmers and associations of terms in functions in the search graph.

4.2.1 Navigation Model

When using text search engines, users navigate among pages by following links contained in those pages. Similarly, in Portfolio, programmers can navigate between functions by following edges in the directed graph of functional dependencies using Portfolio's visual interface. To model the navigation behavior of programmers, we adopt the model of the *random surfer* that is used in popular search engines such as Google. Following functional dependencies helps programmers to understand how to use found functions. The surfer model is called random because the surfer can "jump" to a new URL, or in case of source code, to a new function. These random jumps are called *telepor-*

[2]http://www.searchportfolio.net

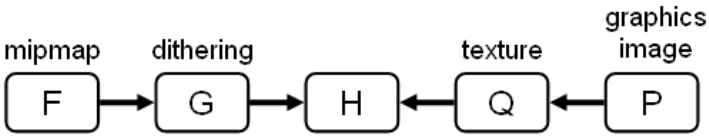


Figure 4.1: Example of associations between different functions.

tations, and this navigation model is the basis for the popular ranking algorithm PageRank [13, 54].

In the random surfer model, the content of functions and queries do not matter, navigations are guided only by edges in the graph that specifies functional dependencies. Accordingly, PageRank reflects only the surfing behavior of users, and this rank is based on the popularity of a function that is determined by how many functions call it. However, the surfing model is query independent since it ignores terms that are used in search queries. Taking into consideration query terms may improve the precision of code searching. That is, *if different functions share concepts that are related to query terms and these functions are connected using functional dependencies, then these functions should be ranked higher.* We need a search model that should automatically make embedded concepts explicit by using associations between functions that share related concepts, and then we combine this model with the surfing model in Portfolio.

4.2.2 Association Model

The main idea of an association model is to establish relevance among facts whose content does not contain terms that match user queries directly. Consider the query "`record compress music file`." Among relevant results there are functions that create and save MIDI files, functions that write data into a file in the MIDI format, and there are multiple functions that compress data. This situation is schematically shown in Figure 4.1, where the function `F` contains the term `record`, the function `G` contains the term `MIDI`, the function `P` contains the terms `music` and `file`, and the

function `Q` contains the term `compress`. Function `F` calls the function `G`, which in turn calls the function `H`, which is also called from the function `Q`, which is in turn called from the function `P`. The functions `F`, `P`, and `Q` will be returned by a search engine that is based on matching query terms to those that are contained in documents. Meanwhile, the functions `H` and `G` may be highly relevant to the query but are not retrieved since they have no words that match the search terms. In addition, the function `Q` can be called from many other functions since its compression functionality is generic; however, its usage is most valuable for programmers in the context of the function that is related to query terms. A problem is how to ensure that the functions `H` and `G` end up on the list of relevant functions.

To remedy this situation we use an association model that is based on a *Spreading Activation Network (SAN)* [18, 20]. In SANs, nodes represent documents, while edges specify properties that connect these documents. The edges' direction and weight reflect the meaning and strength of associations among documents. For example, an article about clean energy and a different article about the melting polar ice cap are connected with an edge that is labeled with the common property "climate change." Once applied to SAN, spreading activation computes new weights for nodes (i.e., ranks) that reflect implicit associations in the networks of these nodes.

In Portfolio, we view finction call graphs as SANs where nodes represent functions, edges represent functional dependencies, and weights represent a *strength of associations*, which includes the number of shared terms. After the user enters a query, a list of functions is retrieved and sorted based on the score that reflects the match between query terms and terms in functions. Once Portfolio identifies top matching functions, it computes SAN to propagate concepts from these functions to others. The result is that every function will have a new score that reflects the associations between concepts in these functions and user queries.

4.2.3 The Combined Model

The ranking vectors for PageRank $\|\pi\|_{PR}$ and spreading activation $\|\pi\|_{SAN}$ are computed separately and later are linearly combined in a single ranking vector $\|\pi\|_C = f(\|\pi\|_{PR}, \|\pi\|_{SAN})$. PageRank is query independent and is precomputed automatically for a function call graph, while $\|\pi\|_{SAN}$ is computed automatically in response to user queries. Assigning different weights in the linear combination of these rankings enables fine-tuning of Portfolio by specifying how each model contributes to the resulting score.

4.3 Our Approach

In this section we describe the architecture of Portfolio and show how to use Portfolio.

4.3.1 Portfolio Architecture

The architecture for Portfolio is shown in Figure 4.2. The main elements of the Portfolio architecture are the database holding software applications (i.e., the Projects Archive), the Metadata Builder, the Function Graph Builder, the SAN and PageRank algorithms, the Visualizer and the key word search engine. Applications metadata describes functions that are declared, defined and invoked in the applications and words that are contained in the source code of these functions and comments. Portfolio is built on an internal, extensible database of 18,203 C/C++ projects that contain close to 2.3Mil files with close to 8.6Mil functions that contain 2,496,172 indexed words. Portfolio indexes and searches close to 270Mil LOC in these C/C++ projects that are extracted from FreeBSD's source code repository called *ports*[3]. It is easy to extend Portfolio by adding new projects to the Projects

[3]http://www.freebsd.org/ports - last checked August 17,2010.

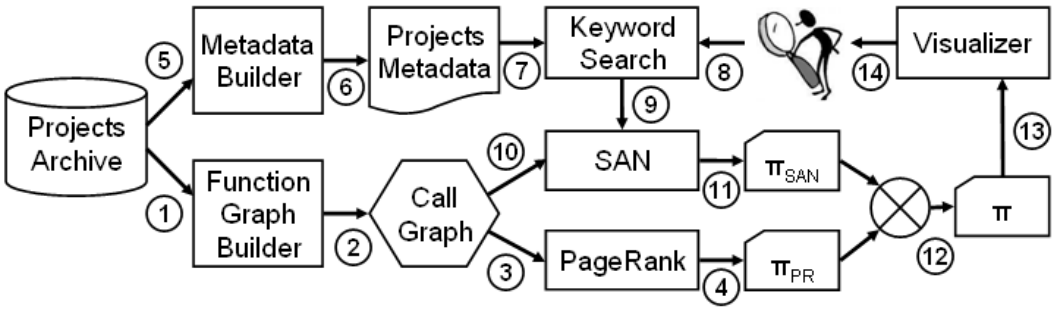


Figure 4.2: Portfolio architecture.

Archive. The user input to Portfolio is shown in Figure 4.2 with the arrow labeled (7). The output is shown with the arrow labeled (18).

Portfolio works as follows. The input to the system is the set of applications from the Projects Archive that contain various functions (1). The Function Graph Builder analyzes the source code of these applications statically and it outputs (2) the *function call graph (FCG)* that contains functional dependencies. This operation is imprecise since resolving dynamic dispatch calls and function pointers statically is an undecidable problem [53]. Since this is done offline, precise program analysis can be accommodated in this framework to achieve better results in obtaining correct functional dependencies. We conduct the sensitivity analysis of Portfolio and its constituent algorithms in Section 4.5.8.1. Next, the algorithm PageRank is run (3) on the FCG, and it computes (4) the rank vector, $||\pi||_{PR}$, in which every element is a ranking score for each function in the FCG.

The Metadata Builder reads in (5) the source code of applications, applies NLP techniques such as stemming and identifier splitting, and indexes the source code as text resulting (6) in Projects Metadata. When the user enters a query (7), it is passed to the key word search component along with the Projects Metadata (8). The key word search engine searches the metadata using the words in the query as keys and outputs (9) the set of Relevant Functions whose source code and comments contain words that match the words from the query. These relevant functions (10)

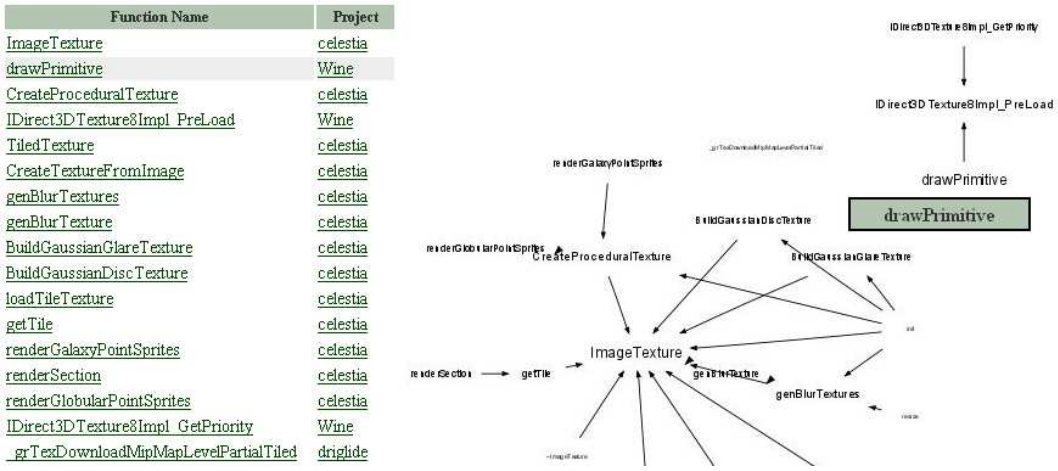


Figure 4.3: A visual interface of Portfolio. The left side contains a list of ranked retrieved functions and the right side contains a static call graph that contains these and other functions; edges of this graph indicate the directions of function invocations. Hovering a cursor over a function on the list shows a label over the corresponding function on the call graph. Font sizes reflect the score; the higher the score of the function, the bigger the font size used to show it on the graph. Clicking on the label of a function loads its source code in a separate browser window.

along with the FCG (11) serve as an input to the algorithm SAN. The algorithm SAN computes (12) spreading activation vector of scores $\|\pi\|_{SAN}$ for functions that are associated with the relevant functions (10). Ranking vectors $\|\pi\|_{PR}$ (14) and $\|\pi\|_{SAN}$ (13) are combined into the resulting vector $\|\pi\|$ (15) that contains ranking scores for all relevant functions. The Visualizer takes (16) the list of relevant functions that are sorted in descending order using their ranking scores and (17) the metadata, in order to present (18) the resulting visual map to the user as it is shown in Figure 4.3.

4.3.2 Portfolio Visual Interface

After the user submits a search query, the Portfolio search engine presents functions relevant to the query in a browser window as it is shown in Figure 4.3. The left side contains the ranked list of retrieved functions and project names, while the right side contains a static call graph that contains these and other functions. Edges of this graph indicate the directions of function invocations. Hov-

ering a cursor over a function on the list shows a label over the corresponding function on the call graph. Font sizes reflect the combined ranking; the higher the ranking of the function, the bigger the font size used to show it on the graph. Clicking on the label of a function loads its source code in a separate browser window.

4.4 Ranking

In this section we discuss our ranking algorithm.

4.4.1 Components of Ranking

There are three components that compute different scores in the Portfolio ranking mechanism: a component that computes a score based on word occurrences (WOS), a component that computes a score based on the random surfer navigation model (PageRank) described in Section 4.2.1, and a component that computes a score based on SAN connections between these calls based on the association model described in Section 4.2.2. WOS ranking is used to bootstrap SAN by providing rankings to functions based on query terms. The total ranking score is the weighted sum of the PageRank and SAN ranking scores. Each component produces results from different perspectives (i.e., word matches, navigation, associations). Our goal is to produce a unified ranking by putting these orthogonal, yet complementary rankings together in a single score.

4.4.2 WOS Ranking

The purpose of WOS is to enable Portfolio to retrieve functions based on matches between words in queries and words in the source code of applications. This is a bootstraping ranking procedure that serves as the input to the SAN algorithm.

The WOS component uses *Okapi BM25*, which is a ranking function typically used by search engines to rank matching documents according to their relevance to a given search query [88]. This function is implemented in the Lucene Java Framework which is used in Portfolio, and is distinguished by TREC for its performance and considered as state-of-the-art in the IR community [81]. BM25 is a standard bag-of-words retrieval function that ranks a set of documents based on the relative proximity of query terms (e.g., without dependencies) appearing in each document. The BM25 score is $S_{wos} = \sum_{i=1}^{n} IDF(q_i) \frac{f(q_i,D) \cdot (k+1)}{f(q_i,D) + k \cdot (1 - b + b \cdot \frac{|D|}{\mu(|D|)})}$, where $f(q_i,D)$ is the q_i's term frequency in the document D with the length (i.e., the number of words) $|D|$, $\mu(|D|)$ is the average document length in the text collection from which documents are drawn, k and b are parameters whose values are usually chosen 1.2 and 0.75 respectively. Finally, the $IDF(q_i)$ is the inverse document frequency weight of the query term q_i.

4.4.3 Spreading Activation

Spreading activation computes weights for nodes in two steps: pulses and termination checks. Initially, a set of starting nodes is selected using a number of top ranked functions using the WOS ranking. During pulses, new weights for different nodes are transitively computed from the starting nodes using the formula $N_j = \Sigma_i f(N_i w_{ij})$, where the weight of the node N_j is equal to the sum of all nodes N_i that are incident to the node N_j with edges whose weights are w_{ij}. The function f is typically called the threshold function that returns nonzero value only if the value of the argument is greater than some chosen threshold, which acts as a termination check preventing "flooding" of the SAN.

4.4.4 PageRank

PageRank is widely described in literature, so here we give its concise mathematical explanation as it is related to Portfolio [13, 54]. The original formula for PageRank of a function F_i, denoted $r(F_i)$, is the sum of the PageRanks of all functions that invoke F_i: $r(F_i) = \sum_{F_j \in B_{F_i}} \frac{r(F_j)}{|F_j|}$, where B_{F_i} is the set of functions that invoke F_i and $|F_j|$ is the number of functions that the function F_j invokes. This formula is applied iteratively starting with $r_0(F_i) = 1/n$, where n is the number of functions. The process is repeated until PageRank converges to some stable values or after some number of steps. Functions that are called from many other functions have a significantly higher score than those that are used infrequently or not at all.

4.4.5 Combined Ranking

The combined rank is $S = \lambda_{PR}\|\pi\|_{PR} + \lambda_{SAN}\|\pi\|_{SAN}$, where λ is the interpolation weight for each type of the score. These weights are determined independently of queries unlike the scores WOS and SAN, which are query-dependent. Adjusting these weights enables experimentation with how underlying structural and textual information in application affects resulting ranking scores. Experimentation with PageRank involves changing the teleportation parameter that we briefly discussed in Section 4.2.1.

4.5 Experimental Design

Typically, search engines are evaluated using manual relevance judgments by experts [66, pages 151-153]. To determine how effective Portfolio is, we conducted an experiment with 49 participants who are C/C++ programmers. Our goal was to evaluate how well these participants could find code

fragments or functions that matched given tasks using three different search engines: Google Code Search (or simply, Google)[4], Koders[5] and Portfolio[6]. We chose to compare Portfolio with Google and Koders because they are popular search engines with the large open source code repositories, and these engines are used by tens of thousands of programmers every day.

4.5.1 Methodology

We used a cross validation experimental design in a cohort of 49 participants who were randomly divided into three groups. The experiment was sectioned in three experiments in which each group was given a different search engine (i.e., Google, Koders, or Portfolio) to find code fragments or functions for given tasks. Each group used a different task in each experiment. Thus each participant used each search engine on different tasks in this experiment. Before the experiment we gave a one-hour tutorial on using these search engines to find code fragments or functions for tasks.

Each experiment consisted of three steps. First, participants translated tasks into a sequence of keywords that described key concepts of code fragments or functions that they needed to find. Then, participants entered these keywords as queries into the search engines (the order of these keywords did not matter) and obtained lists of code fragments or functions that were ranked in descending order.

The next step for participants was to examine the returned code fragments and functions and to determine if they matched the tasks. Each participant accomplished this step individually, assigning a confidence level, C, to the examined code fragments or functions using a four-level Likert scale. We asked participants to examine only the top ten code fragments that resulted from their searches

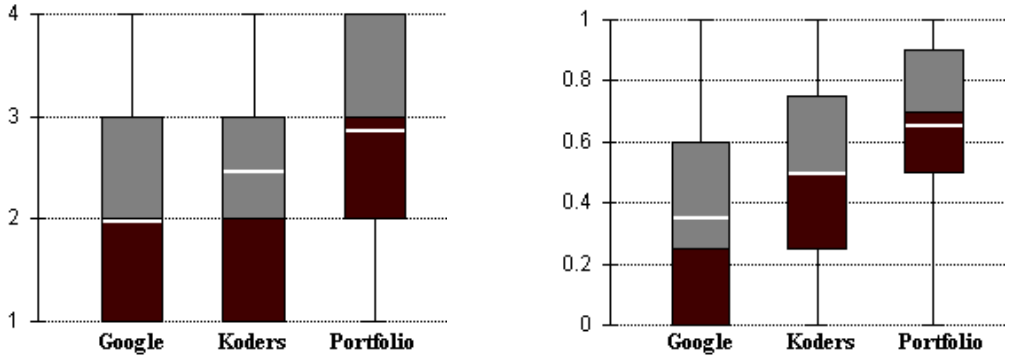
[4]http://www.google.com/codesearch
[5]http://www.koders.com
[6]http://www.searchportfolio.net

since the time for each experiment was limited to two hours.

The guidelines for assigning confidence levels are the following.

1. Completely irrelevant - there is absolutely nothing that the participant can use from this retrieved code fragments, nothing in it is related to keywords that the participant chose based on the descriptions of the tasks.
2. Mostly irrelevant - a retrieved code fragment is only remotely relevant to a given task; it is unclear how to reuse it.
3. Mostly relevant - a retrieved code fragment is relevant to a given task and participant can understand with some modest effort how to reuse it to solve a given task.
4. Highly relevant - the participant is highly confident that code fragment can be reused and s/he clearly see how to use it.

Forty four participants are Accenture employees who work on consulting engagements as professional programmers for different client companies. Five participants are graduate students from the University of Illinois at Chicago who have at least six months of C/C++ experience. Accenture participants have different backgrounds, experience, and belong to different groups of the total Accenture workforce of approximately 203,000 employees. Out of 49 participants, 16 had programming experience with C/C++ ranging from six months to two years, and 18 participants reported more than three years of experience writing programs in C++. Ten participants reported prior experience with Google Code Search and three participants with Koders (which are used in this experiment thus introducing a bias toward these code search engines), nine participants reported frequent use of code search engines, and 16 said that they never used code search engines. All participants have bachelor degrees and 28 have master degrees in different technical disciplines.



(a) Confidence level, C. (b) Precision, P.

Figure 4.4: Statistical summary of the results of the experiment for C and P.

4.5.2 Precision

Two main measures for evaluating the effectiveness of retrieval are precision and recall [114, page 188-191]. The precision is calculated as $P_r = \frac{\texttt{\# of retrieved functions that are relevant}}{\texttt{total \# of retrieved functions}}$, i.e., the precision of a ranking method is the fraction of the top r ranked documents that are relevant to the query, where $r = 10$ in this experiment. Relevant code fragments or functions are counted only if they are ranked with the confidence levels `4` or `3`. The precision metrics reflects the accuracy of the search. Since we limit the investigation of the retrieved code fragments or functions to top ten, the recall is not measured in this experiment.

4.5.3 Hypotheses

We introduce the following null and alternative hypotheses to evaluate how close the means are for the Cs and Ps for control and treatment groups. Unless we specify otherwise, participants of the treatment group use Portfolio, and participants of the control group use either Google or Koders. We seek to evaluate the following hypotheses at a 0.05 level of significance.

H_0 The primary null hypothesis is that there is no difference in the values of confidence level and precision per task between participants who use Portfolio, Google, and Koders.

H_1 An alternative hypothesis to H_0 is that there is statistically significant difference in the values of confidence level and precision between participants who use Portfolio, Google, and Koders.

Once we test the null hypothesis H_0, we are interested in the directionality of means, μ, of the results of control and treatment groups. We are interested to compare the effectiveness of Portfolio versus Google Code Search and Koders with respect to the values of confidence level, C, and precision, P.

H1 (C of Portfolio versus Google) The effective null hypothesis is that $\mu_C^{Port} = \mu_C^G$, while the true null hypothesis is that $\mu_C^{Port} \leq \mu_C^P$. Conversely, the alternative hypothesis is $\mu_C^{Port} > \mu_C^G$.

H2(P of Portfolio versus Google) The effective null hypothesis is that $\mu_P^{Port} = \mu_P^G$, while the true null hypothesis is that $\mu_P^{Port} \leq \mu_P^G$. Conversely, the alternative hypothesis is $\mu_P^{Port} > \mu_P^G$.

H3 (C of Portfolio versus Koders) The effective null hypothesis is that $\mu_C^{Port} = \mu_C^K$, while the true null hypothesis is that $\mu_C^{Port} \leq \mu_C^K$. Conversely, the alternative is $\mu_C^{Port} > \mu_C^K$.

H4(P of Portfolio versus Koders) The effective null hypothesis is that $\mu_P^{Port} = \mu_P^K$, while the true null hypothesis is that $\mu_P^{Port} \geq \mu_P^K$. Conversely, the alternative is $\mu_P^{Port} < \mu_P^K$.

The rationale behind the alternative hypotheses to `H1–H4` is that Portfolio allows users to quickly understand how queries are related to retrieved functions. These alternative hypotheses are motivated by our belief that if users see visualization of functional dependencies in addition to functions whose ranks are computed higher using our ranking algorithm, they can make better decisions about how closely retrieved functions match given tasks.

4.5.4 Task Design

We designed 15 tasks for participants to work on during experiments in a way that these tasks belong to domains that are easy to understand, and they have similar complexity. The authors of this paper

visited various programming forums and internet groups to extract descriptions of tasks from the questions that programmers asked. In addition, we interviewed several programmers at Accenture who explained what tasks they worked on in the past year. Additional criteria for these tasks is that they should represent real-world programming tasks and should not be biased towards any of the search engines that are used in this experiment. Descriptions of these tasks should be flexible enough to allow participants to suggest different keywords for searching. This criteria significantly reduces any bias towards evaluated search engines. These tasks and the results of the experiment are available for download[7].

4.5.5 Tasks

The following three tasks are examples from the set of 15 tasks we used in our experiment.

- Implement a module for reading and playing midi files[8].
- Implement a module that adjusts different parameters of a picture, including brightness, contrast and white balance[9].
- Build a program for managing USB devices. The program should implement routines such as opening, closing, writing and reading from an USB device[10].

4.5.6 Normalizing Sources of Variations

Sources of variation are all issues that could cause an observation to have a different value from another observation. We identify sources of variation as the prior experience of the participants with specific code fragments or functions retrieved by the search engines in this experiment, past

[7] http://www.searchportfolio.net, follow the Experiment link.
[8] http://www.codeproject.com/Messages/1427393/How-Can-I-Read-Midi-File.aspx
[9] http://www.codeguru.com/forum/showthread.php?t=432339
[10] http://www.cplusplus.com/forum/general/25172/

experience in implementing requirements that is similar to one or several of the tasks used in this experiment, the amount of time they spend on learning how to use search engines, and different computing environments which they use to evaluate retrieved code fragments or functions. The first point is sensitive since some participants who already know how some retrieved functions behave are likely to be much more effective than other participants who know nothing of these functions.

We designed this experiment to drastically reduce the effects of covariates (i.e., nuisance factors) in order to normalize sources of variations. Using the cross-validation design we normalize variations to a certain degree since each participant uses all three search engines on different tasks.

H	Var	Approach	Samples	Min	Max	Median	μ	StdDev	σ^2	DF	PCC	p	T	T_{crit}
H1	C	Portfolio	1276	1	4	3	2.86	1.07	1.15	1372	0.04	$4.2 \cdot 10^{-108}$	24	1.96
		Google	1373	1	4	2	1.97	1.11	1.23					
H2	P	Portfolio	184	0	1	0.7	0.65	0.28	0.08	197	0.12	$3 \cdot 10^{-22}$	10.9	1.97
		Google	198	0	1	0.25	0.35	0.33	0.11					
H3	C	Portfolio	1276	1	4	3	2.86	1.07	1.15	1485	0.06	$1.1 \cdot 10^{-26}$	10.9	1.96
		Koders	1486	1	4	2	2.45	1.12	1.25					
H4	P	Portfolio	184	0	1	0.7	0.65	0.28	0.8	207	0.041	$3 \cdot 10^{-8}$	5.76	1.97
		Koders	208	0	1	0.5	0.49	0.3	0.09					

Table 4.1: Results of t-tests of hypotheses, H, for paired two sample for means for two-tail distribution, for dependent variable specified in the column Var (either C or P) whose measurements are reported in the following columns. Extremal values, Median, Means, μ, standard deviation, StdDev, variance, σ^2, degrees of freedom, DF, and the pearson correlation coefficient, PCC, are reported along with the results of the evaluation of the hypotheses, i.e., statistical significance, p, and the T statistics.

4.5.7 Tests and The Normality Assumption

We use one-way ANOVA, t-tests for paired two sample for means, and χ^2 to evaluate the hypotheses. These tests are based on an assumption that the population is normally distributed. The law of large numbers states that if the population sample is sufficiently large (between 30 to 50 participants), then the central limit theorem applies even if the population is not normally distributed [103, pages 244-245]. Since we have 49 participants, the central limit theorem applies, and the above-mentioned tests have statistical significance.

4.5.8 Threats to Validity

In this section, we discuss threats to the validity of this experiment and how we address these threats.

4.5.8.1 Internal Validity

Internal validity refers to the degree of validity of statements about cause-effect inferences. In the context of our experiment, threats to internal validity come from confounding the effects of differences among participants, tasks, and time pressure.

Participants. Since evaluating hypotheses is based on the data collected from participants, we identify two threats to internal validity: C++ proficiency and motivation of participants.

Even though we selected participants who have working knowledge of C++ as it was documented by human resources, we did not conduct an independent assessment of how proficient these participants are in C++. The danger of having poor C++ programmers as participants of our experiment is that they can make poor choices of which retrieved code fragments or functions better match their queries. This threat is mitigated by the fact that out of 44 participants from Accenture, 31 have worked on successful commercial projects as C++ programmers for more than two years.

The other threat to validity is that not all participants could be motivated sufficiently to evaluate retrieved code fragments or functions. We addressed this threat by asking participants to explain in a couple of sentences why they chose to assign certain confidence level to retrieved, and we discarded 27 results for all search engines that were not properly explained.

Time pressure. Each experiment lasted for two hours. For some participants, this was not enough time to explore all 50 retrieved code fragments for five tasks (ten results for each of five tasks). Therefore, one threat to validity is that some participants could try to accomplish more tasks by shallowly evaluating retrieved code fragments and functions. To counter this threat we notified participants that their results would be discarded if we did not see sufficient reported evidence of why they evaluated retrieved code fragments and functions with certain confidence levels.

Sensitivity of Portfolio. Recovering functional dependencies automatically introduces imprecision, since it is an undecidable problem to recover precise functional dependencies in the presence of dynamic dispatch and functional pointers [53]. Since the precision of Portfolio depends on the quality of recovered functional dependencies, we conducted an evaluation of these recovered dependencies with twelve graduate computer science students at DePaul university. We randomly selected a representative sample of 25 different projects in Portfolio and we asked these students to manually inspect source code of these projects to determine the precision of FCG computed in Portfolio.

The results of this evaluation show that the precision of recovered functional dependencies is approximately 76%. While the precision appears to be somewhat lower than desired, it is known that Pagerank is resilient to incorrect links. Link farms, for example, are web spam where people create fake web sites that link to one another in an attempt to skew the PageRank vector. It is estimated that close to 20% of all links on the Internet are spam [36, 93, 8]. However, it is shown that the PageRank vector is not affected significantly by these spam links since its sensitivity is controlled

by different factors, one of which is teleportation parameter [30]. To evaluate the effect of incorrect links on Pagerank vector we conducted experiments where we randomly modified 25% and 50% of links between functions. Our results show that the metric length of the Pagerank vector (computed as the square root of the sum of squares of its components) changes only by approximately 7% for 50% of perturbed functional dependencies. A brief explanation is that by adding or removing a couple of links to functions that are either well-connected or not connected at all, their Pagerank score is not strongly affected. Investigating the sensitivity of Portfolio as well as improving recovery of functional dependencies is the subject of future work.

4.5.8.2 External Validity

To make the results of this experiment generalizable, we must address threats to external validity, which refer to the generalizability of a casual relationship beyond the circumstances of our experiment. The fact that supports the validity of this experimental design is that the participants are highly representative of professional C/C++ programmers. However, a threat to external validity concerns the usage of search tools in the industrial settings, where requirements are updated on a regular basis. Programmers use these updated requirements to refine their queries and locate relevant code fragments or functions using multiple iterations of working with search engines. We addressed this threat only partially, by allowing programmers to refine their queries multiple times.

In addition, participants performed multiple searches using different combinations of keywords, and they select certain retrieved code fragments or functions from each of the search results. We believe that the results produced by asking participants to decide on keywords and then perform a single search and rank code fragments and functions do not deviate significantly from the situation where searches using multiple (refined) queries are performed.

Another threat to external validity comes from different sizes of software repositories. Koders.com claims to search more than 3 Billion LOC, which is also close to the number of LOC reported by Google Code Search. Even though we populated Portfolio's repository with close to 270 Mil LOC, it still remains a threat to external validity.

4.6 Results

In this section, we report the results of the experiment and evaluate the hypotheses.

4.6.1 Results Of The Experiment

We use one-way ANOVA, t-tests for paired two sample for means, and χ^2 to evaluate the hypotheses that we stated in Section 4.5.3.

4.6.1.1 Variables

The main independent variable is the search engine (Portfolio, Google Code Search, and Koders) that participants use to find relevant C/C++ code fragments and functions. The other independent variable is participants' C++ experience. Dependent variables are the values of confidence level, C, and precision, P. We report these variables in this section. The effects of other variables (task description length, prior knowledge) are minimized by the design of this experiment.

4.6.1.2 Testing the Null Hypothesis

We used ANOVA to evaluate the null hypothesis H_0 that the variation in an experiment is no greater than that due to normal variation of individuals' characteristics and error in their measurement. The results of ANOVA confirm that there are large differences between the groups for C with $F =$

$261.3 > F_{crit} = 3$ with $p \approx 5 \cdot 10^{-108}$ which is strongly statistically significant. The mean C for the Google Code Search is 1.97 with the variance 1.14, which is smaller than the mean C for Koders, 2.45 with the variance 1.26, and it is smaller than the mean C for Portfolio, 2.86 with the variance 0.99. Also, the results of ANOVA confirm that there are large differences between the groups for P with $F = 52.5 > F_{crit} = 3.01$ with $p \approx 8.6 \cdot 10^{-22}$ which is strongly statistically significant. The mean P for the Google Code Search is 0.35 with the variance 0.1, which is smaller than the mean P for Koders, 0.49 with the variance 0.09, and it is smaller than the mean P for Portfolio, 0.65 with the variance 0.07. Based on these results we reject the null hypothesis and we accept the alternative hypothesis H_1.

A statistical summary of the results of the experiment for C and T (median, quartiles, range and extreme values) is shown as box-and-whisker plots in Figure 4.4(a) and Figure 4.4(b) correspondingly with 95% confidence interval for the mean.

4.6.1.3 Comparing Portfolio with Google Code Search

To test the null hypothesis `H1` and `H2` we applied two t-tests for two paired sample means, in this case C and P for participants who used Google Code Search and Portfolio. The results of this test for C and for P are shown in Table 4.1. The column `Samples` shows different values that indicate that not all 49 participants participated in all experiments (three different participants missed two different experiments). Based on these results we reject the null hypotheses `H1` and `H2` and we accept the alternative hypotheses that states that **participants who use Portfolio report higher relevance and precision on finding relevant functions than those who use Google Code Search and Koders**.

C/C++ Expert	*C* per par for relev scores			*P*, average		
	Google	Koders	Port	Google	Koders	Port
Yes	10	15.8	17.3	0.38	0.5	0.66
No	7.13	13	15.1	0.3	0.48	0.63
Summ	17.3	28.8	32.4	0.34	0.49	0.645

Table 4.2: Contingency table shows relationship between *C*s per participant for relevant scores and *P*s for participants with and without expert C/C++ experience.

4.6.1.4 Comparing Portfolio with Koders

To test the null hypotheses `H3` and `H4`, we applied two t-tests for two paired sample means, in this case *C* and *P* for participants who used Portfolio and Koders. The results of this test for *C* and for *P* are shown in Table 4.1. Based on these results we reject the null hypotheses `H3` and `H4` that say that **participants who use Portfolio report higher relevance and precision on finding relevant functions than those who use Koders**.

4.6.1.5 Experience Relationships

We construct a contingency table to establish a relationship between *C* and *P* for participants with (2+ years) and without (less than 2 years) expert C++ experience as shown in Table 4.2. To test the null hypotheses that the categorical variables *C* and *P* are independent from the categorical variable Java experience, we apply two χ^2-tests, χ^2_C and χ^2_P for *C* and *P* respectively. We obtain $\chi^2_C = 1.176$ for $p < 0.556$ and $\chi^2_P = 0.48$ for $p < 0.787$. The small values of χ^2 allow us to reject these null hypotheses in favor of the alternative hypotheses suggesting that **there is no statistically strong relationship between expert C++ programming experiences of participants and the values of reported *C*s and *P*s**. That is, participants performed better with Portfolio than with Google Code Search and Koders independently of their C++ experience.

4.6.1.6 Qualitative Evaluation And Reports

Thirty three participants reported that the visualization of functional dependencies in Portfolio is useful and helped them to evaluate potential reuse of retrieved functions, while 12 respondents did not find this visualization useful. Out these 33 participants who found it useful, 27 had more than one year of C++ programming experience, while out of these 12 participants who did not find this visualization useful, only two had more than one year of C++ experience.

Many participants expressed strong dissatisfaction with Google Code Search. A surprising comment came from several participants who said that they preferred to use standard the Google search engine rather than Google code search to look for relevant code fragments in text documents that describe these code fragments. Below are a few comments that participants left on their answer sheets.

- I was very impressed with the accuracy of Portfolio. It even returned code that I realized I would need only after seeing it! I will be using this engine in the future.
- I found Google code search to be less efficient than Google! Very few of the retrieved documents had ANY relevance to the query submitted. *Surprisingly* (the emphasis of the participant), both Koders and Portfolio are better compared to Google code.
- The graph idea (in Portfolio) was really good, but would be better if when the mouse goes over the file or library give some descritpion to summarize info (# of references, # of functions, etc.).
- (Google code search returns) a lot more results, but quantity isn't quality. Not many of these results are useful.
- The "code web" of search results was very helpful for finding out which things to analyze.
- Google code search engine was very hit or miss. It was almost luck when you found what you were looking for.
- Portfolio provided relevant bits based on the search criteria I entered. The most relevant were not always displayed on the top of the list. If multiple search criteria were used (i.e., more than one search query), additional results could have been found.

- Portfolio is user-friendly, fast, and easy to use.
- The search engine Portfolio is a good search tool. It either matches with search criteria or does not match. So developers won't waste time exploring different projects or functions. Also Portfolio gives the description of project which gives idea about this project.
- Search results (in Portfolio) were better as compared to Google search. Clicking on the link takes user to a specific function selected. One thing I liked in Google code search was that right side panel has marking for keywords and I was able to look at different sections of the code that have keywords.

4.7 Related Work

Different code mining techniques and tools have been proposed to find relevant software components as it is shown in Table 4.3. CodeFinder iteratively refines code repositories in order to improve the precision of returned software components [37]. Unlike Portfolio, CodeFinder heavily depends on the descriptions (often incomplete) of software components to use word matching, while Portfolio uses Pagerank and SANs to help programmers navigate and understand usages of retrieved functions.

Codebroker system uses source code and comments written by programmers to query code repositories to find relevant artifacts [115]. Unlike Portfolio, Codebroker is dependent upon the descriptions of documents and meaningful names of program variables and types, and this dependency often leads to lower precision of returned projects.

Even though it returns code snippets rather than functions, Mica is similar to Portfolio since it uses API calls from Java Development Kit to guide code search [106]. However, Mica uses help documentation from third parties to refine the results of the search, while Portfolio automatically retrieves functions from arbitrary code repositories and it uses more sophisticated models to help programmers evaluate the potential of code reuse faster and a with higher precision.

Approach	Granularity		Search Method	Result
	Unit	Usage		
AMC [38]	U	N	W	T
CodeBroker [115]	P,U	Y	W,Q	T
CodeFinder [37]	F,U	Y	W,Q	T
CodeGenie [57]	P	N	W	T
Exemplar [68]	A	Y	W,Q	T
Google Code Search	U	N	W	T
Gridle [85]	U	N	W	T
Hipikat [21]	P	Y	W,Q	T
Koders	U	N	W	T
Krugle	U	N	W	T
Mica [106]	U,F	Y	W,Q	T
ParseWeb [108]	U,F	N	W,Q	T
Portfolio	F,P	Y	P,S,W	G
Prospector [65]	F	N	T	T
S^6 [87]	F,P,U	Y	W,Q	T
SNIFF [15]	F,U	Y	T,W	T
Sourceforge	A	N	W	T
Sourcerer [80]	F,P,U	Y	P,W	T
SPARS-J [45][46]	F	Y	P	T
SpotWeb [109]	U	N	W	T
Strathcona [39]	F	Y	W	T
xSnippet [92]	F	Y	T,W	T

Table 4.3: Comparison of Portfolio with other related approaches. Column `Granularity` specifies how search results are returned by each approach (**P**rojects, **F**unctions, or **U**nstructured text), and if the usage of these resulting code units is shown (**Y**es or **N**o). The column `Search Method` specifies the search algorithms or techniques that are used in the code search engine, i.e., **P**agerank, **S**preading activation, simple **W**ord matching, parameter **T**ype matching, or **Q**uery expansion techniques. Finally, the last column tells if the search engine shows a list of code fragments as **T**ext or it uses a **G**raphical representation of search results to illustrate code usage for programmers.

Exemplar, SNIFF, and Mica use documentation for API calls for query expansion [68, 106, 15]. SNIFF then performs the intersection of types in these code chunks to retain the most relevant and common part of the code chunks. SNIFF also ranks these pruned chunks using the frequency of their occurrence in the indexed code base. In contrast to SNIFF, Portfolio uses navigation and association models that reflect behavior of programmers and improve the precision of the search engine. In addition, Portfolio offers a visualization of usages of functions that it retrieves automatically from existing source code, thus avoiding the need for third-party documentation for API calls.

Web-mining techniques have been applied to graphs derived from program artifacts before. Notably, Inoue et al. proposed Component Rank[46] as a method to highlight the most-frequently used classes by applying a variant of PageRank to a graph composed of Java classes and an assortment of relations among them. Quality of match (QOM) ranking measures the overall goodness of match between two given components [107], which is different from Portfolio in many respects, one of which is to retrieve functions based on surfing behavior of programmers and associations between concepts in these functions.

Gridle[85] also applies PageRank to a graph of Java classes. In Portfolio, we apply PageRank to a graph with nodes at function-level granularity and edges as call relationships among the functions. In addition, we use spreading activation on the call graph to retrieve chains of relevant function invocations, rather than single fragments of code.

Programming task-oriented tools like Prospector, Hipikat, Strathcona, and xSnippet assist programmers in writing complicated code [65, 21, 39, 92]. However, their utilities are not applicable when searching for relevant functions given a query containing high-level concepts with no source code.

Robillard proposed an algorithm for calculating program elements of likely interest to a developer [89]. Portfolio is similar to this algorithm in that it uses relations between functions in the retrieved projects to compute the level of interest (ranking) of the project, however, Robillard does not use models that reflect the surfing behavior of programmers and association models that improve the precision of search. We think there is a potential in exploring connections between Robillard's approach and Portfolio.

S^6 is a code search engine that uses a set of user-guided program transformations to map high-level queries into a subset of relevant code fragments [87], not necessarily functions. Like Portfolio,

S^6 uses query expansion, however, it requires additional low-level details from the user, such as data types of test cases.

4.8 Conclusion

We created an approach called Portfolio for finding highly relevant functions and projects from a large archive of C/C++ source code. In Portfolio, we combined various *natural language processing (NLP)* and indexing techniques with a variation of *PageRank* and *spreading activation network (SAN)* algorithms to address the need of programmers to reuse retrieved code as functional abstractions. We evaluated Portfolio with 49 professional C/C++ programmers and found with strong statistical significance that it performed better than Google Code Search and Koders in terms of reporting higher confidence levels and precisions for retrieved C/C++ code fragments and functions. In addition, participants expressed strong satisfaction with using Portfolio's visualization technique since it enabled them to assess how retrieved functions are used in contexts of other functions.

Chapter 5

Recommending Source Code for Rapid Software Prototypes

5.1 Introduction

Rapid prototyping is a software development activity in which programmers build a prototype of a software product by iteratively proposing, reviewing, and demonstrating the features of that product [64]. It is designed to help project stakeholders explore the features they would like to include in a product, and to interact with the prototype in order to discover and specify requirements. As prototypes are generally thrown-away, they must be built quickly and inexpensively, and must provide the flexibility to easily add or remove features. Other factors, such as efficiency or portability, are less important as the prototype may not even share the same programming language or hardware platform as the final product [64]. Therefore, it is essential to minimize the manual effort involved in building prototypes, and to maximize automation and source code reuse. As such, tool support for automatically locating and reusing features from open-source repositories offers a tremendous opportunity for reducing this manual effort [64].

Rapid prototyping is often divided into a horizontal and a vertical phase [78]. In the horizon-

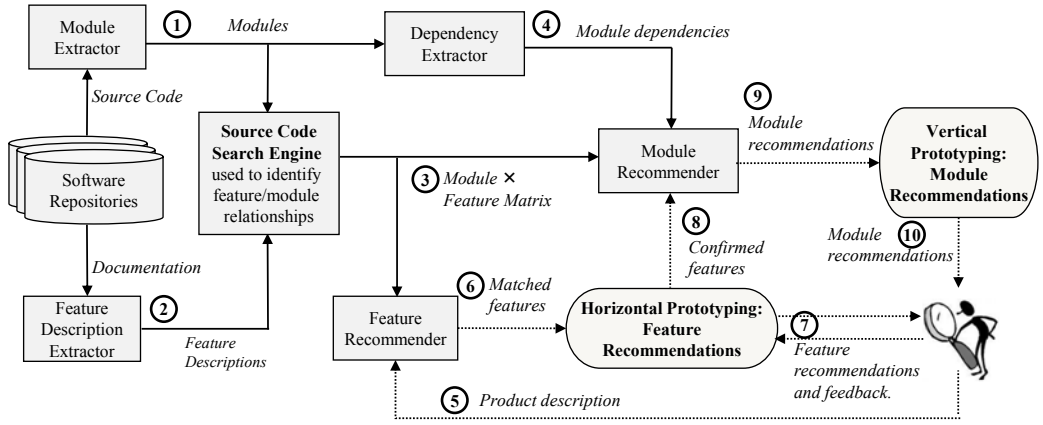


Figure 5.1: Overview of the architecture of our approach.

tal phase, domain analysts identify an initial set of candidate features for implementation in the product. These features, which are often cursorily defined, are presented to the stakeholders for discussion, feedback, and refinement. This activity is often supported by domain analysis tools and techniques which identify features that are common across similar or competitive software systems [27, 49, 26]. However, such approaches provide only limited information about the implementation of those features. In contrast, during the vertical phase of rapid prototyping, developers build full functionality for a selection of features identified during the horizontal phase. This provides a much richer user experience, in which project stakeholders can run the software and interact with the features in order to decide on specific use cases and to identify potential problems.

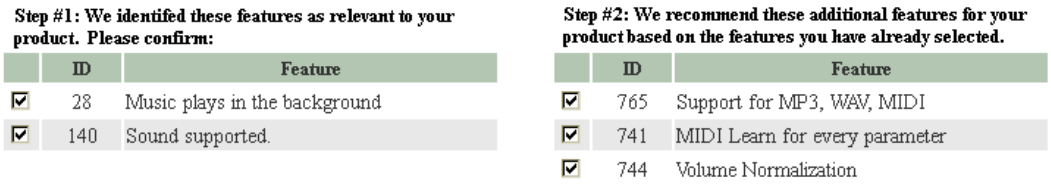
To reduce programming effort and shorten time-to-market, programmers can find and reuse existing solutions for their prototypes. Source code search engines have been developed to locate implementations that are highly-relevant to a feature specified by a programmer (e.g., via a natural-language query) [58, 69]. However, although these engines are effective for locating single features, they are not designed for the more complex, yet common case, in which a prototype will incorporate

a set of interacting features. As a result, existing search engines often return packages that match only a small subset of the desired features, and developers have to invest considerable effort to integrate features from several different packages and projects. Under these circumstances, the cost and effort required for a programmer to comprehend and integrate the returned source code can significantly reduce the benefits of reuse [52].

In this paper we present a novel recommender system for supporting rapid prototyping. Our system directly addresses several shortcomings of existing techniques and tools, by integrating the horizontal and vertical phases of rapid prototyping. Our approach first recommends features, and then locates and recommends relevant source code. We utilize a hybrid set of algorithms based on PageRank [54], set coverage, and Coupling Between Objects (CBO) [17] in order to maximize the coverage of features while proposing a set of packages that minimize the integration effort involved in building a prototype.

We implemented the recommender system and have conducted a cross-validation user study with 31 participants to compare the effectiveness of our approach against that of a state-of-the-art search engine, *Portfolio* [69]. During the study, users entered product descriptions and selected features from those recommended by our system. The users then evaluated the packages recommended by each of the approaches. Results from the study showed that our approach returned more of the desired features per recommendation than Portfolio, that a greater proportion of the source code was relevant to the product description, and that users spent less time evaluating the results from our approach. Our recommender and user study data online for public use [1].

[1] `http://www.cs.wm.edu/semeru/prefab`



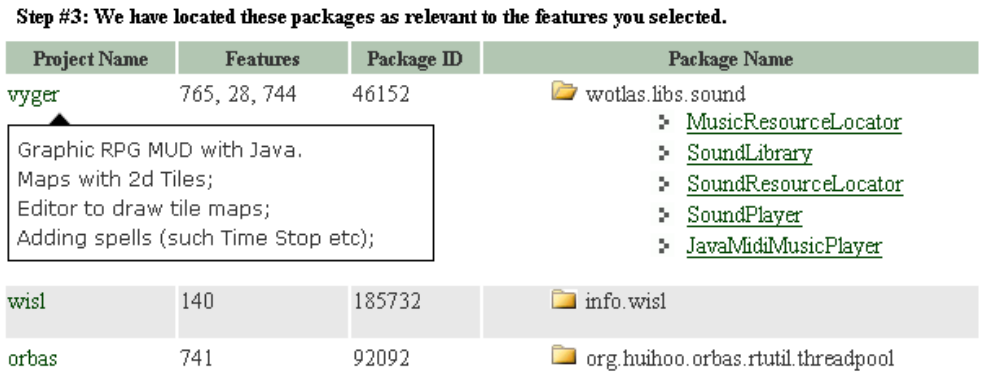
Step #1: We identifed these features as relevant to your product. Please confirm:

	ID	Feature
☑	28	Music plays in the background
☑	140	Sound supported.

Step #2: We recommend these additional features for your product based on the features you have already selected.

	ID	Feature
☑	765	Support for MP3, WAV, MIDI
☑	741	MIDI Learn for every parameter
☑	744	Volume Normalization

(a) Feature Recommendation and Selection



Step #3: We have located these packages as relevant to the features you selected.

Project Name	Features	Package ID	Package Name
vyger	765, 28, 744	46152	wotlas.libs.sound
			MusicResourceLocator
			SoundLibrary
			SoundResourceLocator
			SoundPlayer
			JavaMidiMusicPlayer
wisl	140	185732	info.wisl
orbas	741	92092	org.huihoo.orbas.rtutil.threadpool

Graphic RPG MUD with Java.
Maps with 2d Tiles;
Editor to draw tile maps;
Adding spells (such Time Stop etc);

(b) Module Recommendation

Figure 5.2: Example of Rapid Prototyping in which the user entered the product description "MIDI music player."

5.2 Overview

Before describing the specific details of the underlying algorithms, we provide an architectural overview of our approach. As depicted in in Figure 5.1, there are ten primary steps. Steps (1) and (2) focus on extracting features and modules from one or more software repositories. First, the *Module Extractor* retrieves software modules from one or more repositories (1). These modules are collections of source code related to a particular application or functionality, such as C# namespaces or Java packages; in this paper, we focus on Java packages. Next, the *Feature Extractor* discovers the set of features implemented in the repositories (2). Each feature describes a common function of the software such as "email spam detection". These features are discovered by analyzing the written specifications of applications in the repositories. Further details are provided in Section 5.3.

In order to formulate package recommendations, it is necessary to understand the relationships between features and modules, and also the dependencies between modules. Steps (3) and (4) therefore focus on discovering these relationships. In step (3) a *source code search engine* is used to identify modules that contain specific features and to produce a Module $\times$ Feature Matrix that is used as input to the *Feature and Module Recommenders*. In step (4) *Module Dependencies* are extracted through examining the source code. Further details are provided in Section 5.3.

A user then initiates a request for a recommendation by describing the required functionality of the product they intend to prototype (5). This description is parsed and then elements of the description are matched to features known by the recommender system (6). If matching features are found, they are presented to the user who is asked to confirm or reject their relevance (7). The feature recommender then generates additional feature recommendations and these are also presented to the user for feedback. These recommendations support the horizontal phase of rapid prototyping. A more complete description is provided in Section 5.4.

Our approach also supports the vertical phase of rapid prototyping. In this phase, the selected features are sent to the *Module Recommender* (8), and a series of computations are performed in order to generate a set of module recommendations designed to provide high feature coverage and low external coupling (9). A detailed explanation of this process is provided in Section 5.5. The recommended modules are then presented to the user (10).

We illustrate this process from the users' perspective with a simple scenario showing both feature and module recommendations for the rapid prototyping of a "MIDI music player." As depicted in Figure 5.2(a), the product description was initially matched to features labeled "Music plays in the background" and "Sound supported", and once these features were accepted by the user, the feature recommender suggested three additional features. All recommendations were accepted by

the user. The module recommender then proposed the three packages shown on the right hand side of Figure 5.2(b). The projects from which the packages originate are displayed on the left. The GUI allows the user to see a description of the project as well as browse the Java classes and source code inside the package.

5.3 Mining Product and Feature Data

In order to construct the recommender environment, two different types of data are extracted from the software repositories. First, the feature recommender requires rich textual descriptions of features to provide meaningful and descriptive information to software developers, and second, the module recommender requires high quality source code for effective rapid prototyping. Although, both of these artifact types could be extracted from a single repository, we decided to use separate repositories in order to optimize the effectiveness of both recommenders. This created the additional requirement that there would be significant overlap between the features contained in each repository.

5.3.1 Feature Descriptions

Feature descriptions were extracted from applications in *Softpedia*[2]. Although SoftPedia is not a source code repository; it does provide a repository of product descriptions that include marketing-like summaries and bullet-point lists of features. In the remainder of the paper we, therefore, refer to it as a repository. In general, feature descriptions are mined from product documentation. In the case of Softpedia, we extracted individual sentences from the product summary information and bulleted

[2] `http://www.softpedia.com/`

lists describing features from 117,265 products, categorized under 21 of Softpedia's predefined categories and 159 sub categories. Together these formed a set of 493,347 feature descriptors [26].

Many feature descriptors describe similar functionality. For example a product that "monitors CPU usage in real-time" likely provides similar functionality to one that claims to "show information about CPU usage." Our approach therefore clusters feature descriptors in order to discover a set of meaningful features. We utilized the incremental diffusive clustering algorithm (IDC) and feature naming approach described in our prior work [26]. IDC takes an iterative approach. In each iteration the SPK-Means clustering algorithm is used to cluster the feature descriptors, and then to identify and retain the "best" cluster based on the cohesiveness and size of the cluster. This cluster's dominant terms are then identified and removed from all feature descriptors in order to allow latent topics to emerge in subsequent clustering iterations. The clustering is repeated until no further meaningful terms remain. All identified clusters represent a single feature, and the feature is named by identifying the most representative descriptor for the cluster. Using this approach, the Softpedia data produced a set of 1,135 features.

5.3.2 Source Code Modules

Source code modules were extracted from 13,701 Java applications downloaded from *Sourceforge*[3]. The modules contained 241,655 Java packages and 400 million lines of code. The large size and public accessibility of both Sourceforge and Softpedia repositories suggests a large overlap in their domains, meaning that many of the features discovered through analyzing the Softpedia documentation, are implemented in Sourceforge applications.

[3] `http://www.sourceforge.net/`

5.3.3 Relating Features to Modules

A module is considered related to a feature if that module implements the feature. In order to discover these relations, we used the *Portfolio* search engine [69]. Portfolio takes a natural-language query as input and locates chains of function invocations relevant to that query. For this paper, we modified Portfolio to locate Java packages, and instantiated it over the source code modules we mined from Sourceforge. Then, we used the 1,135 features identified by our IDC algorithm as queries for Portfolio. The Module $\times$ Feature matrix is a matrix where the rows are the modules, the columns are the features, and the cells indicate whether Portfolio detected that feature as implemented by the package.

5.4 Feature Recommendation

When the user provides a description of the product to be prototyped, the feature recommendation algorithm constructs an initial profile of the product by using the cosine similarity metric to match parts of the description to relevant features in our model. We established a threshold score of 0.6 in order for the product to be matched to a feature in keeping with previous practice [26]. As previously explained, these features are presented to the user in order to confirm that the matching has been performed correctly.

Given the feature set of the new product, our feature recommender module identifies similar products and uses their feature profiles to make predictions about the existence of other relevant features in the new product. In our prior work we used a Product $\times$ Feature matrix, based on features found in the Softpedia products, in order to generate recommendations [26]. The objective of the recommender system was to suggest features to include in a product. In contrast, the recommender

system described in this paper is designed to recommend actual source code packages. Therefore, although we utilize the algorithm defined in our previous work to recommend features [26], we use a Product $\times$ Feature matrix mined from the open-source repositories. One benefit of this approach is that recommendations are based on the actual co-occurrence of features in implemented source code, as opposed to the more abstract and incomplete descriptions of features provided by the Softpedia product descriptions. Given the Module $\times$ Feature matrix generated by the source code search engine, the feature recommender module merges the rows representing modules originating from a single product to form a binary Product $\times$ Feature matrix, $M := (m_{i,j})_{P \times F}$, can be generated, where P represents the number of products mined from Sourceforge (13,701), F is the number of feature descriptions from Softpedia (1,135), and $m_{i,j}$ is 1 if and only if the feature j is implemented in product i.

5.4.1 Recommending additional Features

Next, our feature recommender module generates an additional set of feature recommendations, which are presented to the user. This is accomplished using the k-Nearest Neighbor (kNN) algorithm. This method has been shown to be efficient for recommending features and requirements [14]. For the purpose of feature recommendation, the similarity of the new product and each of the existing products in the Product $\times$ Feature matrix, M, is computed and the top k (20) most similar products are selected as *neighbors* of the new product. The binary equivalent of cosine similarity is used to compute the similarity of the new product p with each existing product n as follows:

$$\text{similarity}(p,n) = \frac{|F_p \cap F_n|}{\sqrt{|F_p| \cdot |F_n|}} \tag{5.1}$$

where F_p denotes the set of features of product p [105]. After forming the neighborhoods, features are recommended to the new product using an approach based on Schafer's technique [97] to predict the likelihood of feature f being relevant to product p as follows:

$$\text{pred}(p,f) = \frac{\sum_{n \in nbr(p)} \text{similarity}(p,n) \cdot m_{n,f}}{\sum_{n \in nbr(p)} \text{similarity}(p,n)} \tag{5.2}$$

where $n \in nbr(p)$ represents a neighbor of p, and $m_{n,f}$ is an entry in the binary matrix M indicating whether product n contains feature f. In general, prediction scores will be computed for each candidate feature, and the features with highest predictions will be recommended.

5.4.2 Evaluating Feature Recommender

To statistically evaluate the performance of the feature recommender based on the integration of Softpedia and Sourceforge data, we performed a standard leave-one-out cross validation experiment. Given the Product $\times$ Feature matrix, M, at each run of the experiment, a random feature is removed from one of the products and the recommendation algorithm is executed. The results are then analyzed to see if the recommender was able to recommend back the removed feature. The *Hit Ratio* measures the likelihood that the removed feature is recommended as part of the top N recommendations. In order to calculate the hit ratio, for each test product p, a feature f is randomly removed from the product profile and N recommendations are generated using the remaining features. If feature f is contained in the recommendation list, then the hit ratio for p is 1.0, otherwise, it is 0.0. The hit ratio of the recommendation algorithm is calculated by averaging over the hit ratio values of all the test products. Figure 5.3 compares the hit ratio values of our feature

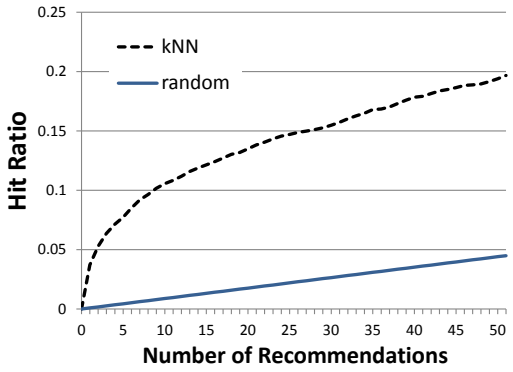


Figure 5.3: Hit ratio comparison for *k*NN and Random Recommender

recommender and a random recommender for different values of N. The results show that there was a sharp improvement over the random case for the early recommendations, meaning that targeted features were recommended towards the top of the list of recommendations, and that the feature recommender was effective.

5.5 Module Recommendation

The module recommender takes as input the list of features agreed upon by the user as a result of the feature recommendation process and produces a list of recommended packages for use in creating the desired rapid prototype.

5.5.1 Recommender Goals

Our recommendation algorithm is designed to optimize the following goals in order to minimize the cost and effort of reusing existing packages in a rapid prototype.

5.5.1.1 Coverage

The recommended packages should provide coverage of as many targeted features as possible.

5.5.1.2 Minimize number of recommended projects

The overhead involved in downloading, installing, and integrating packages from many different projects makes it preferable to construct a rapid prototype using packages drawn from as few projects as possible. Our algorithm, therefore, attempts to minimize the number of projects from which the recommended packages are drawn.

5.5.1.3 Minimize the external coupling of recommended packages

High external coupling decreases developer comprehension of the package, increases the effort needed to execute code in the package, and makes it difficult and costly to reuse the packages.

5.5.2 Package Coupling Costs

Before describing our module recommender algorithm we present our technique for computing package coupling costs. These costs are measured using the Coupling Between Objects (CBO) metric [17], in which a coupling cost is defined for each package. The total coupling cost for the package depends upon both direct and indirect couplings of that package to other packages; however, the dependency chain of coupling costs between packages makes the cost calculation problem nontrivial. In the following section, we explain how to calculate individual coupling costs, and then to extend this metric to account for the common case in which multiple packages are selected from a single project.

The dependency information between packages can be modeled as a weighted directed graph $G = (V, E)$ with each vertex $v \in V$ representing a package and each directed edge $e \in E$ representing the dependency of one package to another. An edge $e_{i,j}$ from node v_i to node v_j exists if and only if one or more classes in package v_i use one or more classes in package v_j. The weight, $w_{i,j}$, on the edge $e_{i,j}$, represents the CBO between the two endpoints and is defined as the fraction of classes in v_i that use at least one class in v_j.

Calculating the coupling cost for packages can be seen as assigning real weights to vertices in the graph, such that the weight of each vertex is a function of the weights on the outgoing edges as well as the weights assigned to all of its outgoing neighbors. In this paper, a variation of the PageRank algorithm [54] is used to compute the vertices weights. The PageRank algorithm was first developed to support the hyperlink analysis of web pages, such that each page in the web graph is assigned a numerical weight, between 0 and 1, known as its PageRank, which represents the relative importance of the page. The PageRank is then used by the search engine to sort and rank the results for a given query. The PageRank algorithm is commonly referred to as the "random surfer model". When a random surfer reaches a page with n outgoing links, he or she will take any of the outgoing links or will jump to a random page in the graph. The PageRank score for each page depends on the number of times it has been visited. More formally, in a directed weighted graph the PageRank score of an arbitrary vertex v_i is iteratively computed as in Equation 5.3 until the algorithm converges:

$$\text{PageRank}(v_i) = \frac{1-d}{N} + d * \sum_{v_j \in In(v_i)} \frac{w_{ji} * \text{PageRank}(v_j)}{\sum_{v_k \in Out(v_j)} w_{jk}} \tag{5.3}$$

where d is a damping factor that ranges between 0.0 and 1.0. If the damping factor is set to one, then pages that have no outgoing external links will act as *rank sinks* and absorb all of the rank in the system. For this reason, the formula is adjusted so that with some probability, the surfer jumps to a random node in the graph.

In the original PageRank algorithm, the score of each node in the graph depends on all its incoming edges. Our problem is different in the sense that the coupling cost of a package depends on the cost of all the packages that it is using and hence depends on all the outgoing edges. Therefore, in order to apply the PageRank method, all the edges in the package graph G were first reversed and then the PageRank scores were computed for the reversed graph. The calculated PageRank scores are an indication of the relative connectivity level of each package to other packages and so are used as the coupling cost values.

5.5.3 Project Coupling Costs

Given a set of features, a project can contain *useful packages* that implement desired features, plus some additional *utility packages* that provide essential services to the useful packages, but which do not directly implement any of the desired features. The coupling cost associated with each project depends on the combined external coupling of the set of useful packages to their utility packages.

In order to accurately compute this cost, all of the useful packages are merged together in the graph through removing internal edges that connect the useful packages, and then replacing external edges, i.e. edges between utility packages and useful packages, with edges to or from the merged package. In the case that a utility package is connected to more than one useful package through outgoing edges, all these edges are merged into a single edge and the weight of this newly formed edge is computed as the sum of all outgoing edges to the useful packages. Similarly, all

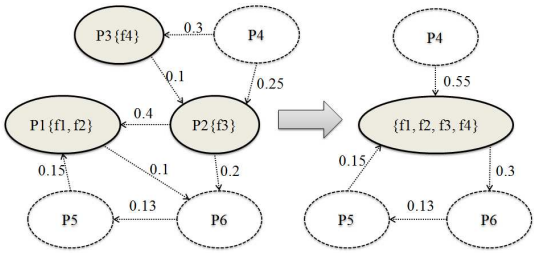


Figure 5.4: Partial reversed package graph for an example project

the incoming edges from the set of useful packages to the utility package are replaced with a single edge connecting the merged package to the utility package. After merging all useful packages, the PageRank scores are recalculated and the project coupling cost for the set of given features is computed as the PageRank score of the merged package.

Figure 5.4 provides an illustrative example. On the left hand side of the diagram, the reversed package graph depicts a set of features, $F = \{f_1, f_2, f_3, f_4\}$, for which the package set $UP = \{p_1, p_2, p_3\}$ are useful and they are connected to other packages that do not implement any of the desired features. On the right-hand side of Figure 5.4, the graph is shown after the useful packages are merged.

Unfortunately this approach can be computationally expensive, as PageRank scores need to be recalculated each time a user issues a new recommendation request. Therefore we considered two computationally inexpensive cost estimation techniques. The first approach sums the individual costs of all useful packages in the project and has a tendency for overestimation; while the second approach underestimates costs by using the cost from the package that exhibits the highest coupling values. An initial analysis showed that the second approach produced better results and so it was adopted for all the remaining experiments described in this paper.

5.5.4 Package Recommendations

Package recommendations are, therefore, made as follows. Given a set of features $F = \{f_1, ..., f_n\}$, our code search engine finds the set of all relevant packages, $PK = \{pk_1, ..., pk_n\}$ and relevant projects $PR = \{pr_1, ..., pr_m\}$ where each pk_i is part of a project in PR.

As a single feature can be implemented in different packages across various projects, the challenge is to find the optimal set of packages with respect to the objectives and constraints mentioned in section 5.5.1. By simplifying the problem to find the minimum number of projects that cover all the features, our problem can be seen as equivalent to the set cover optimization problem which has been shown to be NP-complete [111]. Furthermore, if the problem were to find the minimum coupling cost combination of projects that cover all the features, then it would be another variation of set cover optimization and NP-complete.

The greedy algorithm has previously been used to provide a good approximation of a near-optimal solution [111]. We, therefore, adopted this approach. Our method, as described in Algorithm 1, iteratively selects the *best* project at each step and then selects all of the packages in this project which implement a targeted feature. This process continues until all the targeted features are covered or there are no more candidate projects to choose from. Our criterion for selecting the *best* project is based on the *average cost per feature*, computed by determining the *project coupling cost* as described in section 5.5.3, divided by the number of targeted features implemented by the project.

Algorithm 1 Greedy set-cover algorithm

$selectedPackages \leftarrow \emptyset$
$selectedProjects \leftarrow \emptyset$
while $F \neq \emptyset$ **do**
  $best \leftarrow getBestProject(PR)$
  $selectedProjects \leftarrow selectedProjects \cup best$
  $selectedPackages \leftarrow selectedPackages \cup$
  $usefulPackages(best, F)$
  $F \leftarrow F - coveredFeatures(best, F)$
end while

5.6 Evaluation

In addition to the quantitative study reported in Section 5.4.2 of this paper, we also conducted a qualitative assessment designed to compare the efficacy of our approach against the current state-of-the-art approach. This kind of assessment relies on expert human judgement and is an accepted practice for evaluating recommendations [66].

5.6.1 State-of-the-Art Comparison

The current state-of-the-art technique for locating source code that is relevant to a given feature utilizes a source code search engine. For purposes of this study we, therefore, compared our approach against the *Portfolio* search engine, which has been shown to outperform Google Code Search and Koders in studies where developers search for source code relevant to features they need to implement [69]. We replaced the Package Recommender from our approach with Portfolio by concatenating the text descriptions of the features selected by the user into a single query. This concatenation simulates the case where programmers search for code relevant to multiple features by entering those features into a search engine as a single query. The Java packages recommended by Portfolio were then presented to the user using the same interface we designed for our approach. In this way the user interface was identical across the user study regardless of whether the underlying

recommendations were made by our approach or by the search engine.

5.6.2 Research Questions

The ultimate goal of our rapid prototyping system is to support vertical prototyping through recommending relevant source code packages. Our approach is designed to maximize the number of features covered by the returned source code, while minimizing the amount of source code returned that does not directly implement features. Therefore, our study was designed to address the following research questions (RQs):

RQ_1 Are the recommendations from our approach more relevant to the original product description than the recommendations from the state-of-the-art approach?

RQ_2 Does our approach recommend fewer false positives than the state-of-the-art approach?

RQ_3 Does our approach provide better feature coverage than the state-of-the-art approach?

RQ_4 Do users require less time to understand the recommendations from our approach than from the state-of-the-art approach?

RQ_1 is designed to evaluate the recommendations from our approach in terms of overall relevance to the original product description given by the user. This addresses the possibility that the recommended source code is relevant to the features selected, but not relevant to the query entered by the user. RQ_2 is designed to evaluate whether the recommended source code implements the selected features. Each source code package that is returned should implement one or more of the previously specified features, and our approach attempts to maximize the number of selected features implemented per package. RQ_3 is designed to evaluate feature coverage. Finally, a stated goal of our approach is to reduce manual prototyping effort by minimizing the external coupling of the recommended source code, as well as the amount of that source code. We designed RQ_4 to evaluate the effort in terms of time required to understand the recommendations.

H	Var	Approach	Samples	Min	Max	Median	μ	F	$F_{critical}$	p_1	t	$t_{critical}$	p_2	Decision
H_1	R	Our Approach	331	1	4	2	2.1	25.6	3.85	5e-7	5.06	1.96	<1e-4	Reject
		State-of-the-Art	673	1	4	1	1.7							
H_2	P	Our Approach	128	0	1	0.50	0.59	11.0	3.88	1e-3	3.32	1.97	1e-3	Reject
		State-of-the-Art	96	0	1	0.33	0.43							
H_3	C	Our Approach	331	0	1	0.20	0.29	13.4	3.85	2e-4	3.66	1.96	<1e-4	Reject
		State-of-the-Art	673	0	1	0	0.21							
H_4	T	Our Approach	91	1	38	10	11.5	46.5	3.90	2e-10	6.82	1.98	<1e-4	Reject
		State-of-the-Art	62	6	46	20	20.2							

Table 5.1: Summary of results from the user study showing relevance (R), precision (P), coverage (C), and time required in minutes (T). The column Samples is the number of recommended packages for R and C, the number of queries for P, and the number of queries that users recorded their times for T. ANOVA results are F, $F_{critical}$, and p_1. Student's t-test results are t, $t_{critical}$, and p_2.

Experiment	Group	Approach	Task Set
1	A	Our Approach	T1
	B	State-of-the-Art	T2
2	A	State-of-the-Art	T3
	B	Our Approach	T4

Table 5.2: The cross-validation design of our user study. Different participants used different tasks with different approachs.

5.6.3 Cross-Validation Design of the User Study

A cross-validation design was used in which experts compared the results from our approach to the results from a state-of-the-art approach. A cross-validation design is important because it limits potential threats to validity such as fatigue, bias towards tasks, and bias due to unrelated factors (e.g., user interfaces). Table 5.2 shows an outline of the experimental design. The study was split into two experiments, each lasting one hour. The participants were randomly placed into two equally sized groups, *A* and *B*. The approaches and tasks were rotated among the groups such that different participants used different tasks on different approaches. Also, the participants were prevented from knowing whether they were evaluating our approach or the state-of-the-art approach to avoid introducting bias. During the study our approach was denoted as the *Green* approach and the state-of-the-art approach as *Orange*. The approaches shared the same interface and participants saw only the color denotations.

5.6.3.1 Participants

31 computer science students were recruited from the College of William & Mary to participate in our user study. Twenty-eight were graduate students, while three were undergraduates. The

participants had an average of 4.8 years programming experience and 3.4 years experience with Java. Fourteen reported professional programming experience in various industries.

5.6.3.2 Tasks

The experiments were designed around a set of 12 different tasks. These tasks were roughly equal in complexity and represented a range of potential prototyping tasks. The following is an example task from the user study. A complete listing of the tasks and other case study materials may be downloaded from our online appendix.

> *Build a video player with adjustable bitrate and other video and audio paramters. Your program should support multiple video formats and display the video inside a resizable GUI window.*

In each experiment, a participant was assigned one of the two approaches and a set of tasks. The participant had to formulate a query by defining a set of keywords that represented at least some of the features needed for the task at hand. The participant then entered the query into the GUI and selected features relevant to the query. The system then returned a set of recommended packages.

The participants were asked to evaluate the results according to the relevance of the recommended packages, and through specifying which packages implemented each of the targeted features.

5.6.4 Metrics and Statistical Tests

The following metrics were collected during the study.

5.6.4.1 Relevance

The relevance of a recommended package was evaluated by the participants on a four-point Likert score, rated as an integer from one to four, where four is highly-relevant, three is relevant, two is largely irrelevant, and one means completely irrelevant. The relevance metric was used to answer RQ_1.

5.6.4.2 Precision

Precision is the percent recommendations which implement at least one of the targeted features. Precision will be high when the number of false positives (packages that implement no features) is low; precision is intended to help us answer RQ_2.

5.6.4.3 Coverage

Coverage measures the number of features implemented by a recommended package, and is used to answer RQ_3. Coverage is defined as $\frac{|F_I|}{|F_S|}$, where F_I is the set of features implemented by a given package, and F_S is the set of features selected by the user. Coverage is high when the recommended packages implement a large portion of the features selected by the user.

5.6.4.4 ANOVA

One-way ANOVA and the Student's t-test [104]were used to evaluate the statistical significance of differences in relevance, precision, and coverage. ANOVA is a parametric test that assumes a normally-distributed sample. According to the law of large numbers, the central limit theorem applies when the sample size is greater than 30 [103]. The study included 31 participants, indicating that the results are statically-significant.

5.6.5 Threats to Validity

There are two main threats to internal validity in our study. First, the partipants manually judged the recommendations and their ratings could be influenced by external factors such as fatigue, prior knowledge of the approaches being evaluated, programming proficiency, or lack of motivation. We addressed threats due to fatigue and prior knowledge in the design of our user study by rotating the tools among different groups of participants and denoting the different tools with only a color, rather than a name. The programming proficiency participants could also affect results because users with different proficiency levels could take different factors into consideration. This threat was minimized by randomly distributing participants to the various groups. Finally, the potential motivation problem was at least partially addressed by providing a small stipend to participants who completed the study.

The second main source of threats to internal validity are the tasks. We selected tasks which were easily understood by the authors, and which are in the scope of the projects in the repositories we used. Still, tasks that are out of scope or which are too complex to be understood could cause our recommendation engine to produce low quality results. Therefore, we rotated the sets of tasks that participants used so that in each experiment, each group used different tasks on different tools. Also, we ensured that our approach and the state-of-the-art approach both recommended packages from the same repository.

Sources of threats to external validity include the repositories we used and a potential mismatch of the features from one repository and the source code in another. Our approach relies on a search engine to determine which features are implemented in which packages (see Section 5.3). The search engine we used has shown to perform well in controlled experiments [69], however an

external threat to validity remains in that the performance may vary on different repositories.

Finally, a threat to external validity exists in that we asked programmers to evaluate the relevance of the features in certain packages returned by our tools. The evaluation was based on the programmers' intuition and experience. The programmers may give different responses once they attempt to build software from the components, because they may discover new information about the packages once they attempt to reuse the source code. Future studies could improve our understanding of these results by monitoring how programmers actually reuse specific components in their software.

5.7 Empirical Results

Confidence, precision, and coverage were measured for both our approach and a state-of-the-art approach in a cross-validated user study. The statistical differences were then tested for these metrics. In this section, we present the results of these tests in order to answer our research questions.

5.7.1 Hypotheses

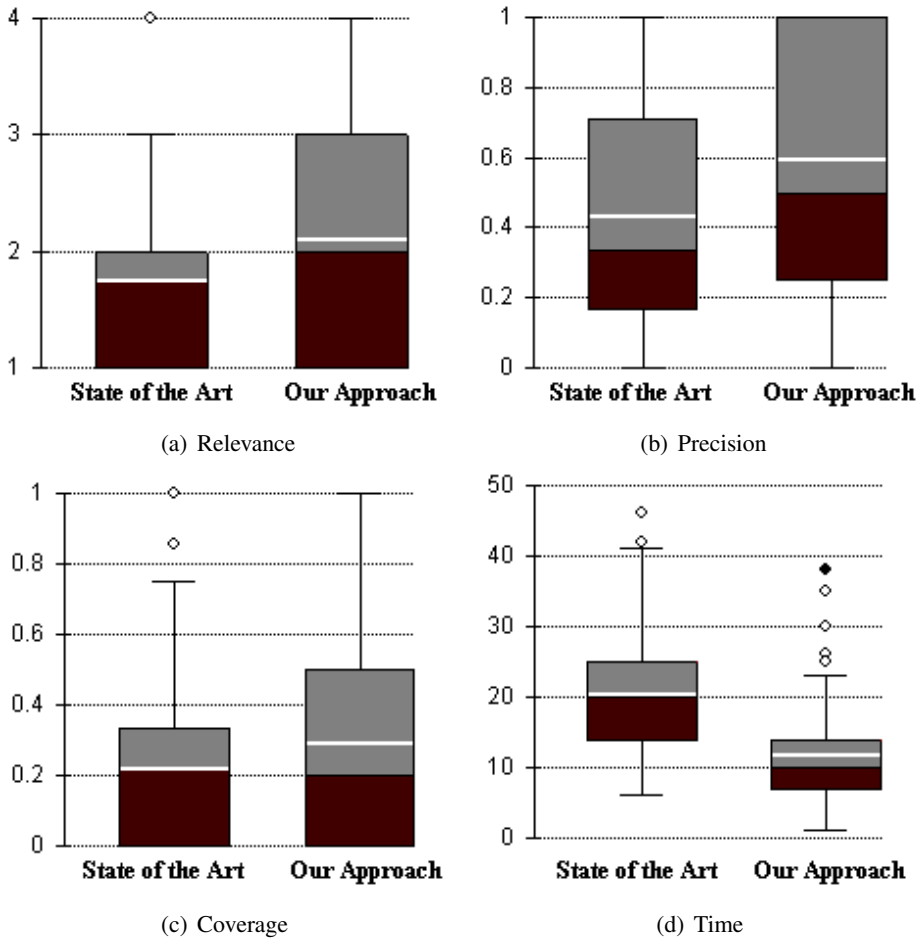
The following null hypotheses are meant to evaluate the directionality of the difference of means for relevance, precision, and coverage. These hypotheses are used in the case when ANOVA indicates a statistically-significant difference in the values of the metrics.

H_1 The mean values of *relevance* are greater for the state-of-the-art approach than for our approach.

H_2 The mean values of *precision* are greater for the state-of-the-art approach than for our approach.

H_3 The mean values of *coverage* are greater for the state-of-the-art approach than for our approach.

H_4 The mean time per query (in minutes) is lower for the state-of-the-art approach than for our approach.



(a) Relevance (b) Precision

(c) Coverage (d) Time

Figure 5.5: Boxplots showing the relevance, precision, coverage, and time per query (in minutes) reported during the user study for the two different approaches. The thick white line is the median. The lower dark box is the lower quartile, while the light box is the upper quartile.

Table 5.1 is a summary of the results from the user study. We reject the three null hypotheses. For ANOVA, the value of F is greater than $F_{critical}$, and $p < 0.05$ in all cases. Moreover, for the Student's t-test of directionality, t exceeds $t_{critical}$. Therefore, the mean values of relevance, precision, and coverage are all greater for our approach than the state-of-the-art approach.

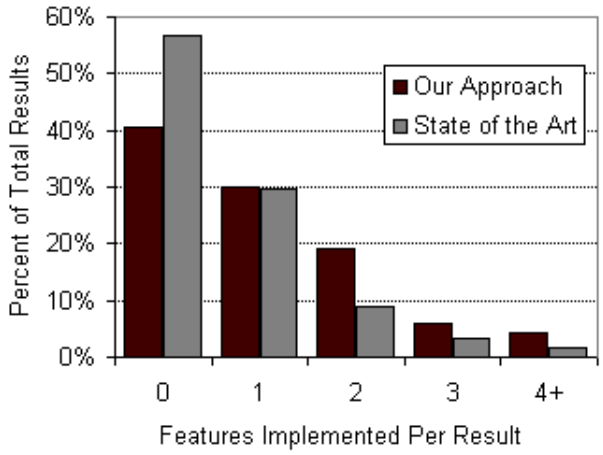


Figure 5.6: A histogram showing the number of features implemented per package, as a percentage of the total number of packages recommended in the user study. Our approach recommends more packages that implement multiple features, compared to the state-of-the-art, and fewer that implement no features.

5.7.2 RQ_1 - Overall Relevance

In Section 5.7.1 we found that the mean values of relevance were greater for our approach than for the state-of-the-art approach. This result indicates that our approach recommends packages which are more-relevant to queries than the state-of-the-art approach. A key difference in the relevance values is that our approach returns a larger number of packages rated as 4 (that is, highly-relevant), as shown in Figure 5.5(a). Seventeen percent of the packages from our approach were rated highly-relevant, while only 7% from the state-of-the-art approach were, and these results were considered outliers. The tasks required multiple features to be implemented, and it is likely that the users only rated packages as highly-relevant if those packages implemented many of the necessary features. However, the state-of-the-art approach, a source code search engine, focuses on locating packages that are relevant to single features. Thus our approach outperforms the state-of-the-art approach in terms of relevance to the queries.

5.7.3 RQ_2 - Recommendations Implementing Features

Precision is a measure of the number of recommended packages which implemented at least one feature that the user selected (see Section 5.6.4.2). We found that the levels of precision for our approach were greater than for the state-of-the-art, which suggests that our approach outperforms the state-of-the-art in terms of the number of recommendations containing useful features. Note that both approaches recommended a large number of packages which did not include any of the selected features, as shown in Figure 5.5(b). This result can be expected when recommending source code because of the difficulty in matching features to source code, and has been widely documented [9, 1, 102]. On the other hand, for many queries, our approach recommended a large number of packages which included relevant features. For half of the queries, at least 60% of the packages included desired features. The state-of-the-art approach performed as well for only 35% of the queries.

5.7.4 RQ_3 - Features Covered by Recommendations

The packages recommended by our approach should implement as many features as possible. We measured the amount of selected features in each package with the coverage metric, and we found that our approach has greater levels of coverage that the state-of-the-art, showing that our approach outperforms the state-of-the-art techniques in terms of features covered by each package recommendation.

Figure 5.5(c) shows the levels of coverage from the user study. While both approaches returned packages that did not implement the selected features, our approach made recommendations that covered a larger percentage of the features. For example, 20% of the packages from our approach implemented at least half of the features selected by the user, compared to 11% of the state-of-

the-art's recommendations. A histogram of our results (Figure 5.6) illustrates that our approach returns packages that implement multiple features. Roughly 30% of recommendations from both approaches implemented one feature. For packages with more than one feature, our approach outperforms the state-of-the-art.

5.7.5 RQ_4 - Time per Query

We found that the participants in the user study were able to complete their evaluations of the recommendations in less time when using our approach than when using the state-of-the-art approach. A stated goal of our approach is to reduce the effort programmers must expend in reusing code for prototypes, and this result indicates that users of our approach are able to understand the source code more quickly than with a state-of-the-art approach.

5.8 Related Work

Our technique for rapid prototyping combines domain analysis for horizontal prototyping with source code recommendation for vertical prototyping. This section gives a brief summary of these areas.

Domain analysis is the process of analyzing a set of relevant software systems to identify, organize, and represent features common to systems within a domain [49]. Most approaches involve either the manual or automated extraction of domain vocabulary from requirements specifications and then use clustering to identify associations and common domain entities [27], [2]. Some authors have taken more structural approaches, for example Chen et. al. constructed requirements relationship graphs (RRG) from several different requirements specifications which they then merged into

a single domain tree [16]. Other researchers, such as Niu et. al. have applied similar techniques to analyze functional requirements in a product line [79]. In contrast to our approach, these techniques are generally applied to a set of requirements specifications with associated design documents, code, and test cases stored in a project repository, making it relatively simple to retrieve code alongside a list of desired features. However, such approaches are constrained by the scope of an organization's project repository, while our approach incorporates hundreds of thousands of project descriptions and source code packages to identify and recommend a far broader set of features.

Building prototypes from existing source code has long been a goal of rapid prototyping tool support [63]. Studies of rapid prototyping have shown that programmers often build prototypes through an iterative process of adding features by using source code examples [11, 55]. This iterative process is known as *opportunistic programming* [12]. Our approach builds on opportunistic programming by allowing programmers to locate source code relevant to several features. In addition, we recommend features that frequently occur in software alongside the features that the programmer needs to implement. Other techniques have been proposed for locating relevant source code, including source code search engines. These engines commonly match keywords in user queries to keywords from source code [37] or documentation [106, 34]. Recent efforts have focused on improving search results using contextual information either from the programmer's development environment [10, 21], the dependencies of the source code being searched [69, 58], or test cases and use cases [87, 57].

5.9 Conclusion

The continuing growth of open source software creates ongoing opportunities for mining useful domain knowledge and for reusing code across projects. In this paper we have explored the idea of using these repositories to support rapid prototyping. Our work has demonstrated that different types of repositories can be used synergistically to create an effective recommender system which can be used to help developers identify relevant source code packages. It has advanced the current state of practice in which source code search engines consider only individual features. In contrast, our approach recommends sets of packages which are designed to facilitate the prototyping and development tasks, and has demonstrated that source code recommendation can be substantially improved with algorithms that consider multiple features as selected by the developer.

Bibliography

[1] AZZAH AL-MASKARI, MARK SANDERSON, AND PAUL CLOUGH. The relationship between ir effectiveness measures and user satisfaction. In *SIGIR*, pages 773–774, New York, NY, USA, 2007. ACM.

[2] VANDER ALVES, CHRISTA SCHWANNINGER, LUCIANO BARBOSA, AWAIS RASHID, PETER SAWYER, PAUL RAYSON, CHRISTOPH POHL, AND ANDREAS RUMMLER. An exploratory study of information retrieval techniques in domain analysis. In *Proceedings of the 2008 12th International Software Product Line Conference*, SPLC '08, pages 67–76, Washington, DC, USA, 2008. IEEE Computer Society.

[3] NICOLAS ANQUETIL AND TIMOTHY C. LETHBRIDGE. Assessing the relevance of identifier names in a legacy software system. In *CASCON*, page 4, 1998.

[4] SUSHIL BAJRACHARYA AND CRISTINA LOPES. Analyzing and mining an internet-scale code search engine usage log. *Journal of Empirical Software Engineering (Special Issue MSR-2009)*, 2009.

[5] SUSHIL BAJRACHARYA, JOEL OSSHER, AND CRISTINA LOPES. Searching api usage examples in code repositories with sourcerer api search. In *Proceedings of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*, SUITE '10, pages 5–8, New York, NY, USA, 2010. ACM.

[6] SUSHIL K. BAJRACHARYA, JOEL OSSHER, AND CRISTINA V. LOPES. Leveraging usage similarity for effective retrieval of examples in code repositories. In *FSE*, pages 157–166, 2010.

[7] PIERRE F. BALDI, CRISTINA V. LOPES, ERIK J. LINSTEAD, AND SUSHIL K. BAJRACHARYA. A theory of aspects as latent topics. In *OOPSLA '08*, pages 543–562, New York, NY, USA, 2008. ACM.

[8] LUCA BECCHETTI, CARLOS CASTILLO, DEBORA DONATO, RICARDO BAEZA-YATES, AND STEFANO LEONARDI. Link analysis for web spam detection. *ACM Trans. Web*, 2(1):1–42, 2008.

[9] TED J. BIGGERSTAFF, BHARAT G. MITBANDER, AND DALLAS E. WEBSTER. Program understanding and the concept assigment problem. *Commun. ACM*, 37(5):72–82, 1994.

[10] JOEL BRANDT, MIRA DONTCHEVA, MARCOS WESKAMP, AND SCOTT R. KLEMMER. Example-centric programming: integrating web search into the development environment. In *CHI*, pages 513–522, 2010.

[11] JOEL BRANDT, PHILIP J. GUO, JOEL LEWENSTEIN, MIRA DONTCHEVA, AND SCOTT R. KLEMMER. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *CHI*, pages 1589–1598, 2009.

[12] JOEL BRANDT, PHILIP J. GUO, JOEL LEWENSTEIN, AND SCOTT R. KLEMMER. Opportunistic programming: how rapid ideation and prototyping occur in practice. In *WEUSE*, pages 1–5, 2008.

[13] SERGEY BRIN AND LAWRENCE PAGE. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[14] CARLOS CASTRO-HERRERA, JANE CLELAND-HUANG, AND BAMSHAD MOBASHER. Enhancing stakeholder profiles to improve recommendations in online requirements elicitation. In *RE*, pages 37–46, 2009.

[15] SHAUNAK CHATTERJEE, SUDEEP JUVEKAR, AND KOUSHIK SEN. Sniff: A search engine for java using free-form queries. In *FASE*, pages 385–400, 2009.

[16] KUN CHEN, WEI ZHANG, HAIYAN ZHAO, AND HONG MEI. An approach to constructing feature models based on requirements clustering. *RE*, 0:31–40, 2005.

[17] SHYAM R. CHIDAMBER AND CHRIS F. KEMERER. A metrics suite for object oriented design. *TSE*, 20:476–493, June 1994.

[18] ALLAN M. COLLINS AND ELIZABETH F. LOFTUS. A spreading-activation theory of semantic processing. *Psychological Review*, 82(6):407 – 428, 1975.

[19] THOMAS A. CORBI. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[20] FABIO CRESTANI. Application of spreading activation techniques in information retrieval. *Artificial Intelligence Review*, 11(6):453–482, 1997.

[21] DAVOR CUBRANIC, GAIL C. MURPHY, JANICE SINGER, AND KELLOGG S. BOOTH. Hipikat: A project memory for software development. *TSE*, 31(6):446–465, 2005.

[22] JULIUS DAVIES, DANIEL M. GERMAN, MICHAEL W. GODFREY, AND ABRAM HINDLE. Software bertillonage: finding the provenance of an entity. In *MSR'11*, pages 183–192, New York, NY, USA, 2011. ACM.

[23] JOSEPH W. DAVISON, DENNIS MANCL, AND WILLIAM F. OPDYKE. Understanding and addressing the essential costs of evolving systems. *Bell Labs Technical Journal*, 5(2):44–54, 2000.

[24] SCOTT C. DEERWESTER, SUSAN T. DUMAIS, THOMAS K. LANDAUER, GEORGE W. FURNAS, AND RICHARD A. HARSHMAN. Indexing by latent semantic analysis. *JASIS*, 41(6):391–407, 1990.

[25] URI DEKEL AND JAMES D. HERBSLEB. Improving api documentation usability with knowledge pushing. In *ICSE*, pages 320–330, 2009.

[26] HORATIU DUMITRU, MAREK GIBIEC, NEGAR HARIRI, JANE CLELAND-HUANG, BAMSHAD MOBASHER, CARLOS CASTRO-HERRERA, AND MEHDI MIRAKHORLI. On-demand feature recommendations derived from mining public product descriptions. In *ICSE*, pages 181–190, 2011.

[27] WILLIAM FRAKES, RUBEN PRIETO-DIAZ, AND CHRISTOPHER FOX. Dare: Domain analysis and reuse environment. *Annals of Software Engineering*, 5:125–141, January 1998.

[28] GEORGE W. FURNAS, THOMAS K. LANDAUER, LOUIS M. GOMEZ, AND SUSAN T. DUMAIS. The vocabulary problem in human-system communication. *Commun. ACM*, 30(11):964–971, 1987.

[29] MARK GABEL AND ZHENDONG SU. A study of the uniqueness of source code. In *Foundations of software engineering*, FSE '10, pages 147–156, New York, NY, USA, 2010. ACM.

[30] DAVID F. GLEICH, PAUL G. CONSTANTINE, ABRAHAM D. FLAXMAN, AND ASELA GUNAWARDANA. Tracking the random surfer: empirically measured teleportation parameters in pagerank. In *WWW*, pages 381–390, 2010.

[31] GOOGLEGUIDE. Google similar pages: Finding similar pages. *http://www.googleguide.com/similar_pages.html*, 2010.

[32] LAURA A. GRANKA, THORSTEN JOACHIMS, AND GERI GAY. Eye-tracking analysis of user behavior in www search. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '04, pages 478–479, New York, NY, USA, 2004. ACM.

[33] MARK GRECHANIK, KEVIN M. CONROY, AND KATHARINA PROBST. Finding relevant applications for prototyping. In *MSR*, page 12, 2007.

[34] MARK GRECHANIK, CHEN FU, QING XIE, COLLIN MCMILLAN, DENYS POSHYVANYK, AND CHAD M. CUMBY. A search engine for finding highly relevant applications. In *ICSE*, pages 475–484, 2010.

[35] MARK GRECHANIK, COLLIN MCMILLAN, LUCA DEFERRARI, MARCO COMI, STEFANO CRESPI-REGHIZZI, DENYS POSHYVANYK, CHEN FU, QING XIE, AND CARLO GHEZZI. An empirical investigation into a large-scale java open source code repository. In *ESEM*, 2010.

[36] ZOLTÁN GYÖNGYI AND HECTOR GARCIA-MOLINA. Link spam alliances. In *VLDB '05*, pages 517–528. VLDB Endowment, 2005.

[37] SCOTT HENNINGER. Supporting the construction and evolution of component repositories. In *ICSE*, pages 279–288, 1996.

[38] ROSCO HILL AND JOE RIDEOUT. Automatic method completion. In *ASE*, pages 228–235, 2004.

[39] REID HOLMES AND GAIL C. MURPHY. Using structural context to recommend source code examples. In *ICSE*, pages 117–125, 2005.

[40] REID HOLMES, ROBERT J. WALKER, AND GAIL C. MURPHY. Strathcona example recommendation tool. In *ESEC/FSE*, pages 237–240, 2005.

[41] REID HOLMES, ROBERT J. WALKER, AND GAIL C. MURPHY. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans. Softw. Eng.*, 32:952–970, December 2006.

[42] JAMES HOWISON AND KEVIN CROWSTON. The perils and pitfalls of mining Sourceforge. In *MSR*, 2004.

[43] XIANGEN HU, ZHIQIANG CAI, ARTHUR C. GRAESSER, AND MATTHEW VENTURA. Similarity between semantic spaces. In *CogSci'05*, 2005.

[44] ELIZABETH HULL, KEN JACKSON, AND JEREMY DICK. *Requirements Engineering*. SpringerVerlag, 2004.

[45] KATSURO INOUE, REISHI YOKOMORI, HIKARU FUJIWARA, TETSUO YAMAMOTO, MAKOTO MATSUSHITA, AND SHINJI KUSUMOTO. Component rank: Relative significance rank for software component search. In *ICSE*, pages 14–24, 2003.

[46] KATSURO INOUE, REISHI YOKOMORI, TETSUO YAMAMOTO, MAKOTO MATSUSHITA, AND SHINJI KUSUMOTO. Ranking significance of software components based on use relations. *IEEE Trans. Softw. Eng.*, 31(3):213–225, 2005.

[47] I.T. JOLLIFFE. *Principal Component Analysis*. Springer Verlag, 1986.

[48] CAPERS JONES. *Applied software measurement: assuring productivity and quality*. McGraw-Hill, Inc., 3rd edition, 2008.

[49] KYO C. KANG, SHOLOM G. COHEN, JAMES A. HESS, WILLIAM E. NOVAK, AND A. SPENCER PETERSON. Feature-oriented domain analysis (foda) feasibility study. Technical report, CMU, November 1990.

[50] SHINJI KAWAGUCHI, PANKAJ K. GARG, MAKOTO MATSUSHITA, AND KATSURO INOUE. Mudablue: an automatic categorization system for open source repositories. *J. Syst. Softw.*, 79(7):939–953, 2006.

[51] KOSTAS KONTOGIANNIS. Program representation and behavioural matching for localizing similar code fragments. In *CASCON '93*, pages 194–205. IBM Press, 1993.

[52] CHARLES W. KRUEGER. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[53] WILLIAM LANDI. Undecidability of static analysis. *LOPLAS*, 1(4):323–337, 1992.

[54] AMY N. LANGVILLE AND CARL D. MEYER. *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, Princeton, NJ, USA, 2006.

[55] THOMAS D. LATOZA, GINA VENOLIA, AND ROBERT DELINE. Maintaining mental models: a study of developer work habits. In *ICSE*, pages 492–501, 2006.

[56] OTÁVIO AUGUSTO LAZZARINI LEMOS, SUSHIL BAJRACHARYA, JOEL OSSHER, PAULO CESAR MASIERO, AND CRISTINA LOPES. Applying test-driven code search to the reuse of auxiliary functionality. In *symposium on Applied Computing*, SAC'09, pages 476–482, New York, NY, USA, 2009. ACM.

[57] OTÁVIO AUGUSTO LAZZARINI LEMOS, SUSHIL KRISHNA BAJRACHARYA, JOEL OSSHER, RICARDO SANTOS MORLA, PAULO CESAR MASIERO, PIERRE BALDI, AND CRISTINA VIDEIRA LOPES. Codegenie: using test-cases to search and reuse source code. In *ASE*, pages 525–526, New York, NY, USA, 2007. ACM.

[58] ERIK LINSTEAD, SUSHIL BAJRACHARYA, TRUNG NGO, PAUL RIGOR, CRISTINA LOPES, AND PIERRE BALDI. Sourcerer: mining and searching internet-scale software repositories. *KDD*, 18:300–336, 2009.

[59] GREG LITTLE AND ROBERT C. MILLER. Keyword programming in java. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 84–93, New York, NY, USA, 2007. ACM.

[60] CHAO LIU, CHEN CHEN, JIAWEI HAN, AND PHILIP S. YU. Gplag: Detection of software plagiarism by program dependence graph analysis. In *KDD'06*, pages 872–881. ACM Press, 2006.

[61] DAPENG LIU, ANDRIAN MARCUS, DENYS POSHYVANYK, AND VACLAV RAJLICH. Feature location via information retrieval based filtering of a single scenario execution trace. In *ASE*, pages 234–243, 2007.

[62] WEI LIU, KE-QING HE, JIANG WANG, AND RONG PENG. Heavyweight semantic inducement for requirement elicitation and analysis. *Semantics, Knowledge and Grid*, 0:206–211, 2007.

[63] L. LUQI AND MOHAMMAD KETABCHI. A computer-aided prototyping system. *IEEE Software*, pages 66–72, 1988.

[64] L. LUQI AND R. STEIGERWALD. Rapid software prototyping. In *HICSS*, pages 470–479, 1992.

[65] DAVID MANDELIN, LIN XU, RASTISLAV BODÍK, AND DOUG KIMELMAN. Jungloid mining: helping to navigate the API jungle. In *PLDI*, pages 48–61, 2005.

[66] CHRISTOPHER D. MANNING, PRABHAKAR RAGHAVAN, AND HINRICH SCHTZE. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[67] COLLIN MCMILLAN, MARK GRECHANIK, AND DENYS POSHYVANYK. Detecting similar software applications. In *ICSE*, 2012.

[68] COLLIN MCMILLAN, MARK GRECHANIK, DENYS POSHYVANYK, CHEN FU, AND QING XIE. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering*, 99(PrePrints), 2011.

[69] COLLIN MCMILLAN, MARK GRECHANIK, DENYS POSHYVANYK, QING XIE, AND CHEN FU. Portfolio: finding relevant functions and their usage. In *ICSE*, pages 111–120, 2011.

[70] COLLIN MCMILLAN, NIKKI HARIRI, DENYS POSHYVANYK, JANE CLELAND-HUANG, AND BAMSHAD MOBASHER. Recommending source code for use in rapid software prototypes. In *ICSE*, 2012.

[71] COLLIN MCMILLAN, MARIO LINARES-VASQUEZ, DENYS POSHYVANYK, AND MARK GRECHANIK. Categorizing software applications for maintenance. In *ICSM'11*, 2011.

[72] AMIR MICHAIL AND DAVID NOTKIN. Assessing software libraries by browsing similar classes, functions and relationships. In *ICSE '99*, pages 463–472, New York, NY, USA, 1999. ACM.

[73] STEFANO MIZZARO. Relevance: The whole history. *JASIS*, 48(9):810–832, 1997.

[74] STEFANO MIZZARO. How many relevances in information retrieval? *Interacting with Computers*, 10(3):303–320, 1998.

[75] STEVEN S. MUCHNICK. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[76] GAIL C. MURPHY, DAVID NOTKIN, AND KEVIN J. SULLIVAN. Software reflexion models: Bridging the gap between source and high-level models. In *FSE*, pages 18–28, 1995.

[77] GAIL C. MURPHY, DAVID NOTKIN, AND KEVIN J. SULLIVAN. Software reflexion models: Bridging the gap between design and implementation. *IEEE Trans. Softw. Eng.*, 27:364–380, April 2001.

[78] JAKOB NIELSEN. *Usability Engineering*. Academic Press, San Diego, CA, USA, 1993.

[79] NAN NIU AND STEVE EASTERBROOK. On-demand cluster analysis for product line functional requirements. *SPLC*, 2008.

[80] JOEL OSSHER, SUSHIL BAJRACHARYA, ERIK LINSTEAD, PIERRE BALDI, AND CRISTINA LOPES. Sourcererdb: An aggregated repository of statically analyzed and cross-linked open source java projects. *MSR*, 0:183–186, 2009.

[81] JOAQUÍN PÉREZ-IGLESIAS, JOSÉ R. PÉREZ-AGÜERA, VÍCTOR FRESNO, AND YUVAL Z. FEINSTEIN. Integrating the Probabilistic Models BM25/BM25F into Lucene. *CoRR*, abs/0911.5046, 2009.

[82] DENYS POSHYVANYK AND MARK GRECHANIK. Creating and evolving software by searching, selecting and synthesizing relevant source code. In *ICSE Companion*, pages 283–286, 2009.

[83] DENYS POSHYVANYK, YANN-GAËL GUÉHÉNEUC, ANDRIAN MARCUS, GIULIANO ANTONIOL, AND VÁCLAV RAJLICH. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Software Eng.*, 33(6):420–432, 2007.

[84] WARREN B. POWELL. *Approximate Dynamic Programming: Solving the Curses of Dimensionality (Wiley Series in Probability and Statistics)*. Wiley-Interscience, 2007.

[85] DIEGO PUPPIN AND FABRIZIO SILVESTRI. The social network of java classes. In *SAC '06*, pages 1409–1413, New York, NY, USA, 2006. ACM.

[86] REINHARD RAPP. The computation of word associations: comparing syntagmatic and paradigmatic approaches. In *19th ICCL*, pages 1–7, Morristown, NJ, USA, 2002.

[87] STEVEN P. REISS. Semantics-based code search. In *ICSE*, pages 243–253, 2009.

[88] STEPHEN E. ROBERTSON, STEVE WALKER, AND MICHELINE HANCOCK-BEAULIEU. Okapi at trec-7: Automatic ad hoc, filtering, vlc and interactive. In *TREC*, pages 199–210, 1998.

[89] MARTIN P. ROBILLARD. Automatic generation of suggestions for program investigation. In *ESEC/FSE*, pages 11–20, 2005.

[90] MARTIN P. ROBILLARD. Topology analysis of software dependencies. *ACM Trans. Softw. Eng. Methodol.*, 17(4):1–36, 2008.

[91] TOBIAS SAGER, ABRAHAM BERNSTEIN, MARTIN PINZGER, AND CHRISTOPH KIEFER. Detecting similar java classes using tree algorithms. In *MSR '06*, pages 65–71, New York, NY, USA, 2006. ACM.

[92] NAIYANA SAHAVECHAPHAN AND KAJAL T. CLAYPOOL. XSnippet: mining for sample code. In *OOPSLA*, pages 413–430, 2006.

[93] HIROO SAITO, MASASHI TOYODA, MASARU KITSUREGAWA, AND KAZUYUKI AIHARA. A large-scale study of link spam detection by graph algorithms. In *AIRWeb '07*, pages 45–48, New York, NY, USA, 2007. ACM.

[94] GERARD SALTON. *Automatic text processing: the transformation, analysis, and retrieval of information by computer*. Addison-Wesley, Boston, USA, 1989.

[95] GERARD SALTON AND MICHAEL J. MCGILL. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1986.

[96] ZACHARY M. SAUL, VLADIMIR FILKOV, PREMKUMAR DEVANBU, AND CHRISTIAN BIRD. Recommending random walks. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 15–24, New York, NY, USA, 2007. ACM.

[97] BEN SCHAFER, DAN FRANKOWSKI, JON HERLOCKER, AND SHILAD SEN. Collaborative filtering recommender systems. *The Adaptive Web*, page 291, 2007.

[98] DAVID SCHULER, VALENTIN DALLMEIER, AND CHRISTIAN LINDIG. A dynamic birthmark for java. In *ASE '07*, pages 274–283.

[99] JONATHAN SILLITO, GAIL C. MURPHY, AND KRIS DE VOLDER. Asking and answering questions during a programming change task. *IEEE Trans. Softw. Eng.*, 34(4):434–451, 2008.

[100] JONATHAN SILLITO, GAIL C. MURPHY, AND KRIS DE VOLDER. Questions programmers ask during software evolution tasks. In *SIGSOFT FSE*, pages 23–34, 2006.

[101] SUSAN ELLIOTT SIM, CHARLES L.A. CLARKE, AND R.C. HOLT. Archetypal source code searches: A survey of software developers and maintainers. *ICPC*, 0:180, 1998.

[102] SUSAN ELLIOTT SIM, MEDHA UMARJI, SUKANYA RATANOTAYANON, AND CRISTINA V LOPES. How well do internet code search engines support open source reuse strategies? *TOSEM*, 2009.

[103] R. MARK SIRKIN. *Statistics for the Social Sciences*. Sage Publications, third edition, August 2005.

[104] MARK D. SMUCKER, JAMES ALLAN, AND BEN CARTERETTE. A comparison of statistical significance tests for information retrieval evaluation. In *CIKM*, pages 623–632, 2007.

[105] ELLEN SPERTUS, MEHRAN SAHAMI, AND ORKUT BUYUKKOKTEN. Evaluating similarity measures: a large-scale study in the orkut social network. pages 678–684, Chicago, Illinois, USA, 2005. ACM.

[106] JEFFREY STYLOS AND BRAD A. MYERS. A web-search tool for finding API components and examples. In *IEEE Symposium on VL and HCC*, pages 195–202, 2006.

[107] NAIYANA TANSALARAK AND KAJAL T. CLAYPOOL. Finding a needle in the haystack: A technique for ranking matches between components. In *CBSE*, pages 171–186, 2005.

[108] SURESH THUMMALAPENTA AND TAO XIE. Parseweb: a programmer assistant for reusing open source code on the web. In *ASE '07*, pages 204–213, New York, NY, USA, 2007. ACM.

[109] SURESH THUMMALAPENTA AND TAO XIE. Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web. In *ASE '08*, pages 327–336, Washington, DC, USA, 2008. IEEE Computer Society.

[110] CORNELIS JOOST VAN RIJSBERGEN. *Information Retrieval*. Butterworth, 1979.

[111] VIJAY V. VAZIRANI. *Approximation Algorithms*. Springer, 2004.

[112] DAVOR ČUBRANIĆ AND GAIL C. MURPHY. Hipikat: recommending pertinent software development artifacts. In *ICSE '03*, pages 408–418, 2003.

[113] XIAOYIN WANG, LU ZHANG, TAO XIE, JOHN ANVIK, AND JIASU SUN. An approach to detecting duplicate bug reports using natural language and execution information. In *ICSE*, pages 461–470, 2008.

[114] IAN H. WITTEN, ALISTAIR MOFFAT, AND TIMOTHY C. BELL. *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*. Morgan Kaufmann, 1999.

[115] YUNWEN YE AND GERHARD FISCHER. Supporting reuse by delivering task-relevant and personalized information. In *ICSE*, pages 513–523, 2002.

[116] YUNWEN YE AND GERHARD FISCHER. Reuse-conducive development environments. *Automated Software Engg.*, 12:199–235, April 2005.

[117] REISHI YOKOMORI, HARVEY SIY, MASAMI NORO, AND KATSURO INOUE. Assessing the impact of framework changes using component ranking. *Software Maintenance, IEEE International Conference on*, 0:189–198, 2009.