

A Search Engine For Finding Highly Relevant Applications

Mark Grechanik, Chen
Fu, Qing Xie
Accenture Technology Labs
Chicago, IL 60601
{mark.grechanik,chen.fu,
qing.xie}@accenture.com

Collin McMillan, Denys
Poshyvanyk
The College of William and
Mary
Williamsburg, VA 23185
{cmc,denys}@cs.wm.edu

Chad Cumby
Accenture Technology Labs
Chicago, IL 60601
chad.c.cumby@accenture.com

ABSTRACT

A fundamental problem of finding applications that are highly relevant to development tasks is the mismatch between the high-level intent reflected in the descriptions of these tasks and low-level implementation details of applications. To reduce this mismatch we created an approach called *Exemplar (EXEcutable exAMPLes ARchive)* for finding highly relevant software projects from large archives of applications. After a programmer enters a natural-language query that contains high-level concepts (e.g., MIME, data sets), Exemplar uses information retrieval and program analysis techniques to retrieve applications that implement these concepts. Our case study with 39 professional Java programmers shows that Exemplar is more effective than Sourceforge in helping programmers to quickly find highly relevant applications.

1. INTRODUCTION

Creating software from existing components is a fundamental challenge of software reuse. Naturally, when programmers develop software, they instinctively sense that there are fragments of code that other programmers wrote and these fragments can be reused. Reusing fragments of existing applications is beneficial because complete applications provide programmers with the contexts in which these fragments exist. Unfortunately, few major challenges make it difficult to locate existing applications that contain relevant code fragments.

A fundamental problem of finding relevant applications is 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* [3]. Many search engines match keywords in queries to words in the descriptions of the applications, comments in their source code, and the names of program variables and types. If no match is found, then potentially relevant applications are never retrieved from repositories. This situation is aggravated by the fact that many application repositories are polluted with poorly functioning projects [13]; 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.

Currently, a prevalent way for programmers to determine if an application is relevant to their task(s) is to download the application, locate and examine fragments of the code that implement features of interest, and observe and analyze the runtime behavior to ensure that the features behave as desired. This process is manual and laborious; programmers study the source code and executions profiles of the retrieved applications in order to determine whether they match task descriptions.

Typically, search engines do little to ensure that retrieved applications contain code fragments that are relevant to requirements that developers need to implement. Short code snippets that are returned as results to user queries do not give enough background or context to help programmers determine how to reuse these snippets, and programmers typically invest a significant intellectual effort (i.e., they need to overcome a high cognitive distance [17]) to understand how to use these code snippets in larger scopes. On the other hand, if code fragments are retrieved in the contexts of applications, it makes it easier for programmers to understand how to reuse these code fragments.

A majority of code search engines 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 defined precisely. The idea of using API calls to improve code search was proposed and implemented elsewhere [8][4]; however, it was not evaluated with statistical significance over a large code-base using a standard information retrieval methodology [22, pages 151-153].

We created an application search system called *Exemplar (EXEcutable exAMPLes ARchive)* that helps users find highly relevant executable applications for reuse. Exemplar combines information retrieval and program analysis techniques to reliably link high-level concepts to the source code of the applications via standard third-party *Application Programming Interface (API)* calls that these applications use. We have built Exemplar as part of our S^3 architecture [27] and conducted a case study with 39 professional Java programmers to evaluate this search engine. The results show with strong statistical significance that users find more relevant applications with higher precision with Exemplar than those with Sourceforge. Exemplar is available for public use¹.

2. OUR APPROACH

In this section we describe the key ideas and give intuition about why and how our approach works.

2.1 The Problem

A straightforward approach for finding highly relevant applications is to search through the source code of applications to match

¹<http://www.xemplar.org>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE '10, May 2-8 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-719-6/10/05 ...\$10.00.

keywords from queries to the names of program variables and types. The precision of this approach depends highly on programmers choosing meaningful names, but their compliance is generally difficult to enforce [1].

This problem is partially addressed by programmers who create meaningful descriptions of the applications that they deposit into 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, making it difficult for users to guess exact keywords to find relevant applications. This is known as the vocabulary problem, which states that “no single word can be chosen to describe a programming concept in the best way” [7]. This problem is general to all search engines but somewhat mitigated by the fact that different programmers who participate in the projects use their vocabularies to write code, comments, and descriptions of these projects.

Modern search engines do little to ensure that retrieved applications are highly relevant to tasks or requirements that are described using high-level queries. To ensure relevancy, a code mining system should take high-level queries and return executable applications whose functionality is described by high-level requirements thereby solving an instance of the concept assignment problem. We believe that the rich context provided by whole applications make it easier for programmers to reuse code fragments from these applications.

2.2 Key Ideas

Our goal is to automate parts of the human-driven procedure of searching for relevant applications. Suppose that requirements specify that a program should encrypt and compress data. When retrieving sample applications from Sourceforge² using the keywords `encrypt` and `compress`, programmers look at the source code to check to see if some API calls from third-party packages are used to encrypt and compress data. Even though the presence of these API calls does not guarantee that the applications are relevant, it is a good starting point for deciding whether to check these applications further.

What we seek is to augment standard code search to include API documentations of widely used libraries, such as standard *Java Development Kit (JDK)*. Of course, 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 in addition to finding keyword matches in the descriptions and the source code of applications. When programmers read these help documents about API calls that implement certain high-level concepts, they trust these documents because they 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. Help documents are more verbose and accurate, and consequently trusted more than the descriptions of applications from repositories [6].

In addition, we observe that relations between concepts entered in queries are often preserved 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, where relations between elements of high-level models (e.g., processing elements of software architectures) are preserved in their implementations in source code [24]. 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 ones where these calls are not connected.

Consider, for example, 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 we say that these functions are connected using a dataflow link which reflects the implicit logical connection between keywords in queries, specifically, the data should be encrypted and then sent to some destination.

To improve the precision of our approach, our idea is to determine relations between API calls in retrieved applications. All things equal, if a dataflow link is present between two API calls in the program code of one application and there is no link between the same API calls in some other application, then the former application should have a higher ranking than the latter. In addition, knowing how API calls are connected using dataflows enables programmers to better understand the contexts of the code fragments that contain these API calls. Finally, it is possible to utilize dataflow connections to extract code fragments, which is a subject of our future work on our S^3 architecture [27].

2.3 Our Goal

Our goal is to develop a search engine that is most effective in the solution domain (i.e., the domain in which engineers use their ingenuity to solve problems [14, pages 87,109]). In the problem domain, requirements are expressed using vague objectives or wish lists. Conversely, in the solution domain engineers go into implementation details. To realize requirements in the solution domain, engineers look for reusable abstractions that are often implemented using third-party API calls. Thus Exemplar should be most effective when keywords reflect the reality of the solution domain.

Consider a situation in which engineers develop a matchmaking application. Using keywords such as `sweet` and `love` to describe requirements from the problem domain, it is possible to find a variety of applications in Sourceforge with according descriptions. However, it is unlikely that these keywords are used to describe API calls in Java documentation, so it is up to engineers to investigate retrieved applications to determine if any code can be reused. Since Exemplar uses basic word matches in addition to locating API calls, it performs equally well in this situation when compared with existing search engines.

Exemplar is more effective than existing code search engines when keywords come from the solution domain. 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`, `share`, `view`, `data sets`, `key value pairs`, an unlikely candidate application called BIOLAP is retrieved using Exemplar with a high ranking score. The description of this application matches only the keywords `data sets`, and yet this application made it to the top five of the list.

The reason is that BIOLAP uses the class `MimeType`, specifically its method `getParameterMap` that 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 solution domain for the given task. It is needless to say that BIOLAP, which is contained in the Sourceforge repository does not show on the top 300 list of retrieved applications when the search is performed with the Sourceforge search engine. The same situation happens when users enter other queries that contain matching keywords in the descriptions of API calls.

²<http://sourceforge.net/> as of September 6, 2009.

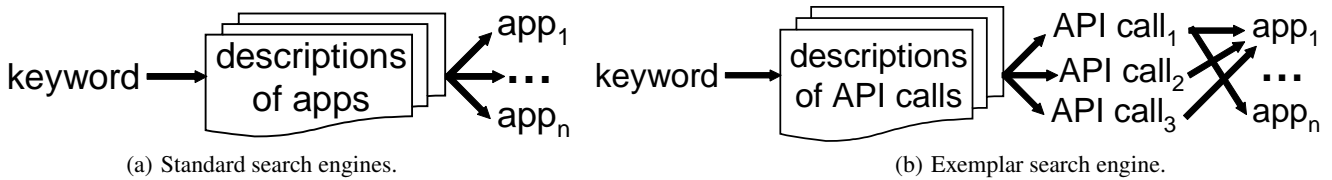


Figure 1: Illustrations of the processes for standard and Exemplar search engines.

The idea behind Exemplar is to enable engineers to retrieve hidden and highly relevant applications for their solution domains.

2.4 Our Approach

We describe our approach using an illustration of differences between the process for standard search engines shown in Figure 1(a) and the Exemplar process shown in Figure 1(b).

Consider the process for standard search engines (e.g., Sourceforge, Google code search) shown in Figure 1(a). A keyword from the query is matched against words in the descriptions of the applications in some repository (Sourceforge, Krugle) or words in the entire corpus of source code (Google Code Search). When a match is found, applications app_1 to app_n are returned.

Consider the process for Exemplar shown in Figure 1(b). A keyword from the query is 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 $call_1$ to $call_k$ are returned. These names are matched against the names of the functions invoked in these applications. When a match is found, applications app_1 to app_n are returned.

A fundamental difference between these search schemes is that Exemplar uses help documents to obtain the names of the API calls in response to user queries. Doing so can be viewed as instances of the *query expansion* concept in information retrieval systems [2] and *concept location* [20]. The aim of query expansion is to reduce this query/document mismatch by expanding the query with concepts that have similar meanings to the set of relevant documents. Using help documents, the initial query is expanded to include the names of the API calls whose semantics unequivocally reflects specific behavior of the matched applications.

In addition 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 API calls from several different libraries, 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 $API\ call_1$ to $API\ call_k$. 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 [7].

As it is shown in Figure 1(b), API calls $API\ call_1$, $API\ call_2$, and $API\ call_3$ are invoked in the 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 app_1 will be retrieved from the repository.

Searching help documents produces additional benefits. API calls from help documents 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 assignment problem [3].

3. EXEMPLAR ARCHITECTURE

The architecture for Exemplar is shown in Figure 2. The main elements of the Exemplar architecture are the database holding applications (i.e., the Apps Archive), the Search and Ranking engines, and the API call lookup. Applications metadata describes dataflow links between different API calls invoked in the applications. Exemplar is being built on an internal, extensible database of help documents that come from the JDK API documentation. It is easy to extend Exemplar by plugging in different help documents for other widely used third-party libraries.

The inputs to Exemplar are shown in Figure 2 with thick solid arrows labeled (1) and (4). The output is shown with the thick dashed arrow labeled (14).

Exemplar works as follows. The input to the system are help documents describing various API calls (1). The Help Page Processor indexes the description of the API calls in these help documents and outputs the API Calls Dictionary, which is the set of tuples $\langle \langle word_1, \dots, word_n \rangle, API\ call \rangle$ linking selected words from the descriptions of the API calls to the names of these API calls (2). Our approach for mapping words in queries to API calls is different from the *keyword programming* technique [19], since we derive mappings between words and APIs from external documentation rather than source code.

When the user enters a query (4), it is passed to the API call lookup component along with the API Calls Dictionary (3). The lookup engine searches the dictionary using the words in the query as keys and outputs the set of the names of the API calls whose descriptions contain words that match the words from the query (5). These API calls serve as an input (6) to the Search Engine along with the Apps Archive (7). The engine searches the Archive and retrieves applications that contain input API calls (8).

The Analyzer pre-computes the Applications Metadata (10) that contains dataflow links between different API calls from the applications source code (9). Since this is done offline, precise program analysis can be accommodated in this framework to achieve better results in dataflow ranking. This metadata is supplied to the Ranking Engine (12) along with the Retrieved Applications (11), and the Ranking Engine combines keyword matching score with API call scores to produce a unified rank for each retrieved

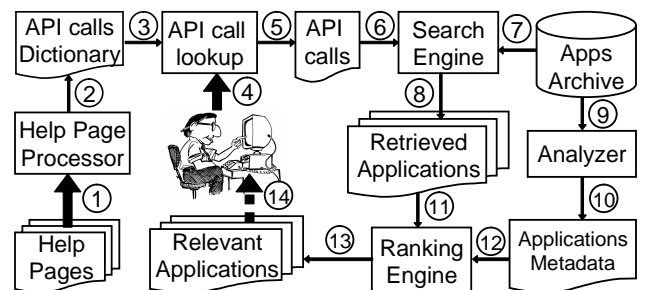


Figure 2: Exemplar architecture.

application. Finally, the engine sorts applications using their ranks and it outputs Relevant Applications (13), which are returned to the user (14).

4. RANKING

In this section we discuss our ranking algorithm and its components such as dataflow computations.

4.1 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 (WOS), a component that computes a score based on the number of 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. Each component produces results of different perspectives (i.e., word matches, API calls, dataflow connections). Our goal is to produce a unified ranking by putting these different rankings together in a single score.

The purpose of WOS is to enable Exemplar to retrieve applications based on matches between words in queries and words in the descriptions of applications in repositories. This is a baseline Exemplar search that should be as effective as the one of Sourceforge. This approach may be useful for a small subset of applications that do not use any third-party API calls to implement different functions. In this case, Exemplar’s ranking engine should rely on word matches to return relevant applications.

However, our investigation of available projects in Sourceforge shows that a majority of applications use third-party API calls, and some of these calls implement functionality that is typically referred in keywords from user queries. Simply put, the more relevant API calls are found in an application, the higher its rank should be. The component RAS computes the API call-based ranking score.

Finally, the component DCS computes a score based on weighted dataflow links. We detect different ways of passing data between API calls and assign weights differently to these dataflow links. We discuss these ranking components in depth below.

4.2 WOS Ranking

WOS component uses *Okapi BM25* [29], which is a ranking function that is typically used by search engines to rank matching documents according to their relevance to a given search query. This function is implemented in the Lucene Java Framework which is used in Exemplar, and it is distinguished by TREC for its performance and considered as state-of-the-art in the IR community [26]. 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. BM25 score is computed as $S_{wos} = \sum_{i=1}^n IDF(q_i) \frac{f(q_i, D) \cdot (k+1)}{f(q_i, D) + k \cdot (1-b + b \cdot \frac{|D|}{\mu(|D|)})}$,

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, and finally the $IDF(q_i)$ is the inverse document frequency weight of the query term q_i .

4.3 RAS Ranking

We consider each section in the library documentation that describes different API calls as a separate document. The collection of API documents is defined as $D_{API} = (D_{API}^1, D_{API}^2, \dots, D_{API}^k)$. 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 [32].

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 com-

puted as $S_{ras}^j = \frac{\sum_{i=1}^p n_i^j \cdot C_i^j}{|A|^j}$, where $|A|^j$ is the total number of API calls in the application, j .

4.4 DCS Ranking

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 invoking an API call is `t var=ocallname(p1, ..., pn)`. 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 [23, 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(...); ...; g(var, ...)`). Anti-dependence occurs when the API call f reads a memory location that the API call g later writes (e.g., `f(var, ...); ...; var=g(...)`). 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 between these calls. The absence of an edge means that there is no dependency between two API calls. Let the total number of connections between n relevant 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 applica-

tion, j , is computed as $S_{dcs}^j = \frac{\sum_{i=1}^{n(n-1)} w_i^j \cdot Link_i^j}{n(n-1)}$, where w_i is the weight to each type of flow dependency for the given link $Link_i$, 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.

4.5 Combined Ranking

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.

5. IMPLEMENTATION

In this section we describe how we implemented Exemplar.

5.1 Crawlers

Exemplar consists of two crawlers: *Archiver* and *Walker*. *Archiver* populated Exemplar's repository by retrieving from Sourceforge more than 30,000 Java projects that contain close to 50,000 submitted archive files, which comprise the total of 414,357 files. *Walker* traverses Exemplar's repository, opens each project by extracting its source code from zipped archive, and applies a dataflow computation utility to the extracted source code. In addition, the Archiver regularly checks Sourceforge to see if there are new updates and it downloads these updates into the Exemplar repository.

To extract all occurrences of invocations of JDK API calls in all available Java projects, we ran 65 threads for over 50 hours on five servers and 25 workstations: three of these servers have two dual core 3.8Ghz EM64T Xeon processors with 8Gb RAM each, and two have four 3.0Ghz EM64T Xeon CPUs with 32Gb RAM each. The workstations uniformly had one 2.83Ghz quad-core CPU and 2Gb RAM. This job resulted in finding close to twelve million invocations of these API calls from JDK 1.5 across all projects. The next item was to compute dataflow connections between these calls in all Java applications in the Exemplar's repository.

5.2 Dataflow Computation

Our approach relies on the tool PMD³ for computing approximate dataflow links, which are based on patterns of dataflow dependencies. Using these patterns it is possible to recover a large number of possible dataflow links between API calls; however, some of these recovered links can be false positives. In addition, we currently recover links among API calls within files (intraprocedurally), hence it is likely that some intraprocedural links are missed and no interprocedural analyses are performed.

5.3 Computing Rankings

We use the Lucene search engine⁴ to implement the core retrieval based on keyword matches. We indexed descriptions and titles of Java applications, and independently we indexed Java API call documentation by duplicating descriptions about the classes and packages in each methods. Thus when users enter keywords, they are matched separately using the index for titles and descriptions and the index for API call documents. As a result, two lists are retrieved: the list of applications and the list of API calls. Each entry in these lists are accompanied by a rank (i.e., conceptual similarity, C , a number between 0 and 1).

The next step is to locate retrieved API calls in the retrieved applications. To improve the performance we configure Exemplar to use the positions of the top two hundred API calls in the retrieved list. These API calls are crosschecked against API calls invoked in the retrieved applications, and the combined ranking score is computed for each application. The list of applications is sorted using the computed ranks, and returned to the user.

6. CASE STUDY DESIGN

Typically, search engines are evaluated using manual relevance judgments by experts [22, pages 151-153]. To determine how effective Exemplar is, we conducted a case study with 39 participants who are Java 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.

6.1 Methodology

We used a cross validation study design in a cohort of 39 participants who were randomly divided into three groups. The study was sectioned in three experiments in which each group was given a different search engine (i.e., SF, EWD, or END) to find applications for given tasks. Each group used a different task 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 individually, 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.

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.

6.2 Precision

Two main measures for evaluating the effectiveness of retrieval are precision and recall [38, page 188-191]. The precision is calculated as $P_r = \frac{\# \text{ of retrieved applications that are relevant}}{\text{total } \# \text{ of retrieved applications}}$, 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

³<http://pmd.sourceforge.net/> as of September 6, 2009.

⁴<http://lucene.apache.org/> as of September 6, 2009.

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.

6.3 Hypotheses

We introduce the following null and alternative hypotheses to evaluate how close the means are for the C s and 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 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_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 SF, EWD, and END.

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 EWD versus the END and SF with respect to the values of confidence level, C , and precision, P .

H1 (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 $\mu_C^{EWD} > \mu_C^{SF}$.

H2(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 $\mu_P^{EWD} > \mu_P^{SF}$.

H3 (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 $\mu_C^{EWD} > \mu_C^{END}$.

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

H5 (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 $\mu_C^{END} > \mu_C^{SF}$.

H6(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 $\mu_P^{END} > \mu_P^{SF}$.

The rationale behind the alternative hypotheses to H1 and H2 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 H3 and H4 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 H5 and H6 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.

6.4 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. A sample task, for instance, asks a user to design a Java applications for sharing, viewing, and exploring large data sets that are encoded using MIME.

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.

6.5 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.

6.6 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 [33, pages 244-245]. Since we have 39 participants, the central limit theorem applies, and the above-mentioned tests have statistical significance.

6.7 Threats to Validity

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

6.7.1 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

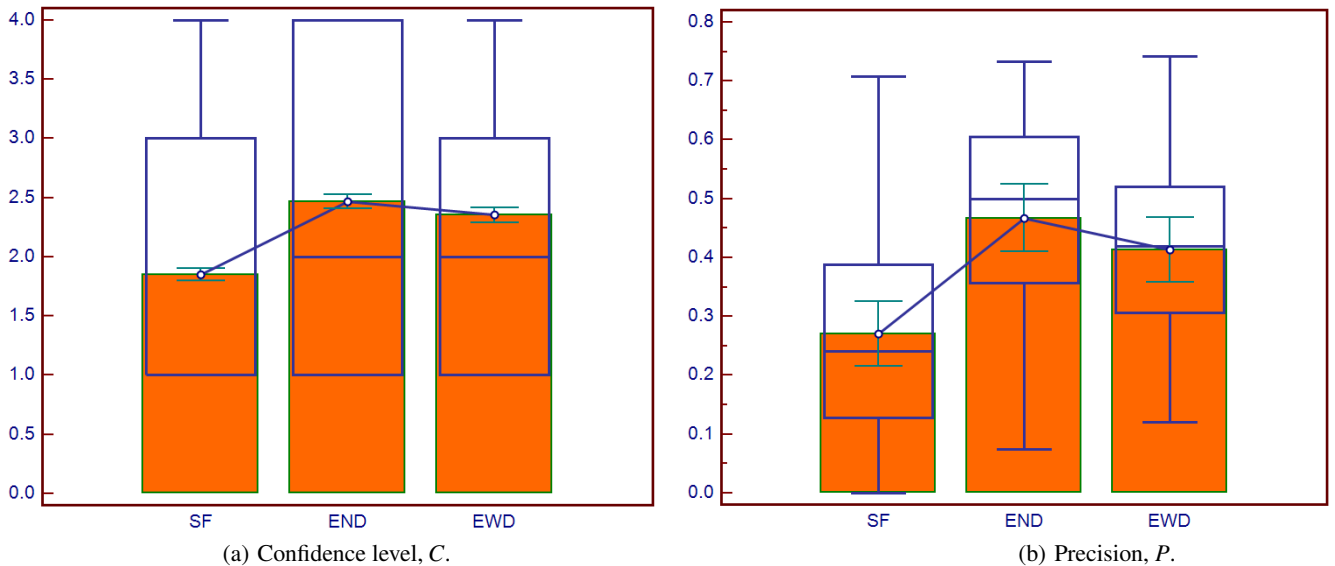


Figure 3: Statistical summary of the results of the case study for C and P . The central box represents the values from the lower to upper quartile (25 to 75 percentile). The middle line represents the median. The thicker vertical line extends from the minimum to the maximum value. The filled-out box represents the values from the minimum to the mean, and the thinner vertical line extends from the quarter below the mean to the quarter above the mean.

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 of specifications of different software systems that were written by different people for different companies.

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.

6.7.2 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.

The other 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.

7. RESULTS

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

7.1 Case Study Results

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 6.3.

7.1.1 Variables

A main independent variable is the search engine (SF, EWD, END) that participants use to find relevant Java applications. The other independent variable is participants’ Java experience. Dependent variables are the values of confidence level, C , and precision, P . 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.

7.1.2 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 differ-

H	Var	Approach	Samples	Min	Max	Median	μ	σ^2	DF	C	p	T	T_{crit}
H1	C	EWD	1273	1	4	2	2.35	1.19	1272	-0.02	$3.2 \cdot 10^{-34}$	12.5	1.96
		SF	1273	1	4	1	1.82	1.02					
H2	P	EWD	35	0.12	0.74	0.42	0.41	0.026	34	0.34	$5.6 \cdot 10^{-5}$	4.6	2.03
		SF	35	0.075	0.73	0.48	0.46	0.17					
H3	C	EWD	1273	1	4	2	2.35	1.19	1272	0.01	0.004	2.68	1.96
		END	1273	1	4	3	2.47	1.27					
H4	P	EWD	35	0.12	0.74	0.42	0.41	0.26	34	0.41	0.15	1.5	2.03
		END	35	0.075	0.73	0.48	0.46	0.17					
H5	C	END	1307	1	4	3	2.47	1.13	1306	-0.02	$6.2 \cdot 10^{-46}$	14.8	1.96
		SF	1307	1	4	1	1.84	1.02					
H6	P	END	37	0.075	0.73	0.5	0.47	0.03	36	0.4	$1.1 \cdot 10^{-7}$	6.6	2
		SF	37	0	0.71	0.24	0.27	0.16					

Table 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, C, are reported along with the results of the evaluation of the hypotheses, i.e., statistical significance, p , and the T statistics.

Java Expert	C per par for relev scores			P, average		
	SF	END	EWD	SF	END	EWD
Yes	8.3	14	15	0.27	0.42	0.41
No	10.4	17.8	14.8	0.28	0.53	0.41
Summary	18.7	31.7	29.8	0.275	0.475	0.41

Table 2: Contingency table shows relationship between Cs per participant for relevant scores and Ps for participants with and without expert Java experience.

ences 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_1 .

A statistical summary of the results of the case study for C and T (median, quartiles, range and extreme values) are shown as box-and-whisker plots in Figure 3(a) and Figure 3(b) correspondingly with 95% confidence interval for the mean.

7.1.3 Comparing Sourceforge with Exemplar

To test the null hypothesis H1, H2, H5, and H6 we applied four t-tests for paired two sample for means, for C and P for participants who used SF and both variants of Exemplar. The results of this test for C and for P are shown in Table 1. The column Samples shows that 37 out of a total of 39 participants participated in all experiments (two participants missed one experiment). Based on these results we reject the null hypotheses H1, H2, H5 and H6, 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**.

7.1.4 Comparing EWD with END

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 END and EWD. The results of this test for C and for P are shown in Table 1. Based on these results we accept the null hypotheses H3 and H4 that say that **participants who use EWD do not report higher relevance and precision on finding relevant applications than those who use END**.

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. As it turns out, a few participants had vague ideas as to what dataflow connections meant and how to incorporate them into the evaluation process. At this point it is a subject of our future work to investigate this phenomenon in more detail.

7.1.5 Experience Relationships

We construct a contingency table to establish a relationship between C and P for participants with (3+ years) and without (less than 3 years) expert Java experience as shown in Table 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 $\chi^2_C = 2.9$ for $p < 0.24$ and $\chi^2_P = 0.67$ for $p < 0.71$. 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 Java programming experiences of participants and the values of reported Cs and Ps**. That is, participants performed better with Exemplar than with Sourceforge independently of their Java experience.

8. 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 3. CodeFinder iteratively refines code repositories in order to improve the precision of returned software components [9]. Like Exemplar, Codefinder reformulates queries

Approach	Granularity		Corpora	Query Expansion
	Search	Input		
CodeFinder [9]	M	C	D	Yes
CodeBroker [39]	M	C	D	Yes
Mica [34]	F	C	C	Yes
Prospector [21]	F	A	C	Yes
Hipikat [5]	A	C	D,C	Yes
xSnippet [31]	F	A	D	Yes
Strathcona [12][11]	F	C	C	Yes
AMC [10]	F	C	C	No
Google Code	F,M,A	C,A	D,C	No
Sourceforge	A	C	D	No
SPARS-J [15][16]	M	C	C	No
Sourcerer [25]	A	C	C	No
CodeGenie [18]	A	C	C	No
SpotWeb [37]	M	C	C	Yes
ParseWeb [36]	F	A	C	Yes
S ⁶ [28]	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 [4]	F,M	C,A	D,C	Yes
Exemplar	F,M,A	C,A	D,C	Yes

Table 3: 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.

in order to expand the search scope. Unlike Exemplar, CodeFinder heavily depends on the descriptions (often incomplete) of software components while Exemplar links API help pages whose information is of higher quality than ad-hoc descriptions of components.

Codebroker system uses source code and comments written by programmers to query code repositories to find relevant artifacts [39]. 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 [34]. 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. In addition, Exemplar returns executable projects while Mica returns code snippets as well as non-code artifacts.

SNIFF extends the idea of using documentation for API calls for query expansion [8][34] in several ways [4]. 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 [4], MICA [34], and our original MSR idea [8], 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 [21]. Prospector is an effective tool to assist programmers in writing complicated code, however, it falls short of providing support for a full-fledged code search engine.

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 [5]. 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 [12][11]. 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.

Robillard proposed an algorithm for calculating program elements of likely interest to a developer [30]. Exemplar is similar to this algorithm in that it uses relations between API calls in the retrieved projects to compute the level of interest (ranking) of the project, however, it does not find relevant applications.

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 [31]. It remains to be seen how XSnippet can be applicable to finding relevant projects, since its goal is to return code fragments based on existing code context.

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 [35], 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 [15][16]. Unlike CRM, Exemplar ranking mechanism is based on a combination of query expansion and relations between API calls that implement high-level concepts in queries.

S⁶ 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 [28], not complete applications. Like Exemplar, S⁶ uses query expansion, however, it requires additional low-level details from the user, such as data types of test cases.

9. CONCLUSION

We created an approach called Exemplar for finding highly relevant software projects from a large archive of executable examples. In Exemplar, we combined program analysis techniques with information retrieval to convert high-level user queries to basic functional abstractions that are used automatically in code search engines to retrieve highly relevant applications. We evaluated Exemplar with 39 professional Java programmers and found with strong statistical significance that it performed better than Sourceforge in terms of reporting higher confidence levels and precisions for retrieved Java applications. In addition, participants expressed strong satisfaction with using Exemplar since it enabled them to assess retrieved applications using well-documented API calls from third-party trusted Java libraries.

Acknowledgments

We warmly thank anonymous reviewers for their comments and suggestions that helped us to improve the quality of this paper. We are especially grateful to Dr. Kishore Swaminathan, the Chief Scientist and Director of Research for his invaluable support. This work is supported by NSF CCF-0916139, CCF-0916260, Accenture, and United States AFOSR grant number FA9550-07-1-0030. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

10. REFERENCES

- [1] N. Anquetil and T. C. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *CASCON*, page 4, 1998.
- [2] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.
- [3] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster. Program understanding and the concept assignment problem. *Commun. ACM*, 37(5):72–82, 1994.
- [4] S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine for java using free-form queries. In *FASE*, pages 385–400, 2009.
- [5] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Trans. Software Eng.*, 31(6):446–465, 2005.
- [6] U. Dekel and J. D. Herbsleb. Improving api documentation usability with knowledge pushing. In *ICSE*, pages 320–330, 2009.
- [7] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Commun. ACM*, 30(11):964–971, 1987.
- [8] M. Grechanik, K. M. Conroy, and K. Probst. Finding relevant applications for prototyping. In *MSR*, page 12, 2007.
- [9] S. Henninger. Supporting the construction and evolution of component repositories. In *ICSE*, pages 279–288, 1996.
- [10] R. Hill and J. Rideout. Automatic method completion. In *ASE*, pages 228–235, 2004.
- [11] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE*, pages 117–125, 2005.
- [12] R. Holmes, R. J. Walker, and G. C. Murphy. Strathcona example recommendation tool. In *ESEC/SIGSOFT FSE*, pages 237–240, 2005.
- [13] J. Howison and K. Crowston. The perils and pitfalls of mining Sourceforge. In *MSR*, 2004.
- [14] E. Hull, K. Jackson, and J. Dick. *Requirements Engineering*. SpringerVerlag, 2004.
- [15] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component rank: Relative significance rank for software component search. In *ICSE*, pages 14–24, 2003.
- [16] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto. Ranking significance of software components based on use relations. *IEEE Trans. Softw. Eng.*, 31(3):213–225, 2005.
- [17] C. W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.
- [18] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, and C. V. Lopes. Codegenie: using test-cases to search and reuse source code. In *ASE '07*, pages 525–526, New York, NY, USA, 2007. ACM.
- [19] G. Little and R. C. Miller. Keyword programming in java. *Automated Software Engg.*, 16(1):37–71, 2009.
- [20] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *ASE*, pages 234–243, 2007.
- [21] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *PLDI*, pages 48–61, 2005.
- [22] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.
- [23] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.
- [24] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *SIGSOFT FSE*, pages 18–28, 1995.
- [25] J. Ossher, S. Bajracharya, E. Linstead, P. Baldi, and C. Lopes. SourcererdB: An aggregated repository of statically analyzed and cross-linked open source java projects. *MSR*, 0:183–186, 2009.
- [26] J. Pérez-Iglesias, J. R. Pérez-Agüera, V. Fresno, and Y. Z. Feinstein. Integrating the Probabilistic Models BM25/BM25F into Lucene. *CoRR*, abs/0911.5046, 2009.
- [27] D. Poshyvanyk and M. Grechanik. Creating and evolving software by searching, selecting and synthesizing relevant source code. In *ICSE Companion*, pages 283–286, 2009.
- [28] S. P. Reiss. Semantics-based code search. In *ICSE*, pages 243–253, 2009.
- [29] S. E. Robertson, S. Walker, and M. Hancock-Beaulieu. Okapi at trec-7: Automatic ad hoc, filtering, vlc and interactive. In *TREC*, pages 199–210, 1998.
- [30] M. P. Robillard. Automatic generation of suggestions for program investigation. In *ESEC/SIGSOFT FSE*, pages 11–20, 2005.
- [31] N. Sahavechaphan and K. T. Claypool. XSnippet: mining for sample code. In *OOPSLA*, pages 413–430, 2006.
- [32] G. Salton. *Automatic text processing: the transformation, analysis, and retrieval of information by computer*. Addison-Wesley, Boston, USA, 1989.
- [33] R. M. Sirkin. *Statistics for the Social Sciences*. Sage Publications, third edition, August 2005.
- [34] J. Stylos and B. A. Myers. A web-search tool for finding API components and examples. In *IEEE Symposium on VL and HCC*, pages 195–202, 2006.
- [35] N. Tansalarak and K. T. Claypool. Finding a needle in the haystack: A technique for ranking matches between components. In *CBSE*, pages 171–186, 2005.
- [36] S. Thummalapenta and T. 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.
- [37] S. Thummalapenta and T. 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.
- [38] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*. Morgan Kaufmann, 1999.
- [39] Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. In *ICSE*, pages 513–523, 2002.