Detecting Similar Software Applications

Collin McMillan
College of William and Mary
Williamsburg, VA 23185
cmc@cs.wm.edu

Mark Grechanik
Accenture Technology Labs and U. of Illinois
Chicago, IL 60601
drmark@uic.edu

Denys Poshyvanyk
College of William and Mary
Williamsburg, VA 23185
denys@cs.wm.edu

***Abstract*—Although popular *text* search engines allow users to retrieve similar web pages, *source code* search engines do not have this feature. Detecting similar applications is a notoriously difficult problem, since it implies that similar high-level requirements and their low-level implementations can be detected and matched automatically for different applications.**

We created a novel approach for automatically detecting *Closely reLated ApplicatioNs (CLAN)* that helps users detect similar applications for a given Java application. Our main contributions are an extension to a framework of relevance and a novel algorithm that computes a similarity index between Java applications using the notion of semantic layers that correspond to packages and class hierarchies. We have built CLAN and we conducted an experiment with 33 participants to evaluate CLAN and compare it with the closest competitive approach, MUDABlue. The results show with strong statistical significance that CLAN automatically detects similar applications from a large repository of 8,310 Java applications with a higher precision than MUDABlue.

I. 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 [11]. 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 [19, pages 74,80] [26]. This situation is aggravated by the fact that many application repositories are polluted with poorly functioning projects [17]; 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 implementations of some functions even if they do not perform the same high-level functionality [10]. 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 [23], [25], [31], [39], [42]. 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* [4]. 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 [24]) 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* [18] that are modeled as existing

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

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 [5], [12], [13]; 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] [22] and a system that combines CLAN and MUDABlue that we implemented[4].

II. 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.

A. 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 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 [21, 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.

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

B. Similarity Between Applications

We define the meaning of similarity between applications by using Mizzaro's well-established conceptual framework for relevance [32], [33]. 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 [46].

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 [1].

C. 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 [36]. 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 [36]. For example, the similarity engine MUDABlue uses syntagmatic associations for computing similarities among applications [22]. The problem with this approach is that computed relevance is relatively imprecise when compared with CLAN as we show in Section V.

D. 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 [5], [8], [12], [13], [43]. 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.

E. 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 [41]. 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" [9]. 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 [14]. 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 [35].

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.

III. Our Approach

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

A. 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 [18]. Moreover, recent work has shown that API classes and packages can be used to categorize software applications using those classes and packages [30].

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.

B. 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 [7]. 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.

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 $r = 300$ [7], [34]. 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.

[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.

C. CLAN Architecture and Workflow

The architecture for CLAN is shown in Figure 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 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.

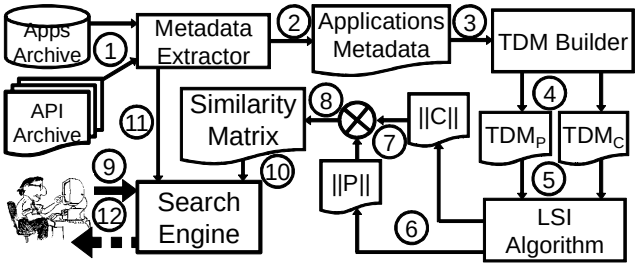


Figure 1. CLAN architecture and workflow.

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)`.

IV. Experimental Design

Typically, search and retrieval engines are evaluated using manual relevance judgments by experts [28, 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.

A. Background on MUDABlue and Combined

MUDAblue is the closest relevant work to CLAN since it provides automatic categorization for applications [22]. The *cluster hypothesis* specifies that documents that cluster together tend to be relevant to the same concept [46]. 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 [22].

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.

B. 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.

C. Precision

Two main measures for evaluating the effectiveness of retrieval are precision and recall [49, 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.

D. 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, 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.

E. 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].

F. 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 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.

G. Threats to Validity

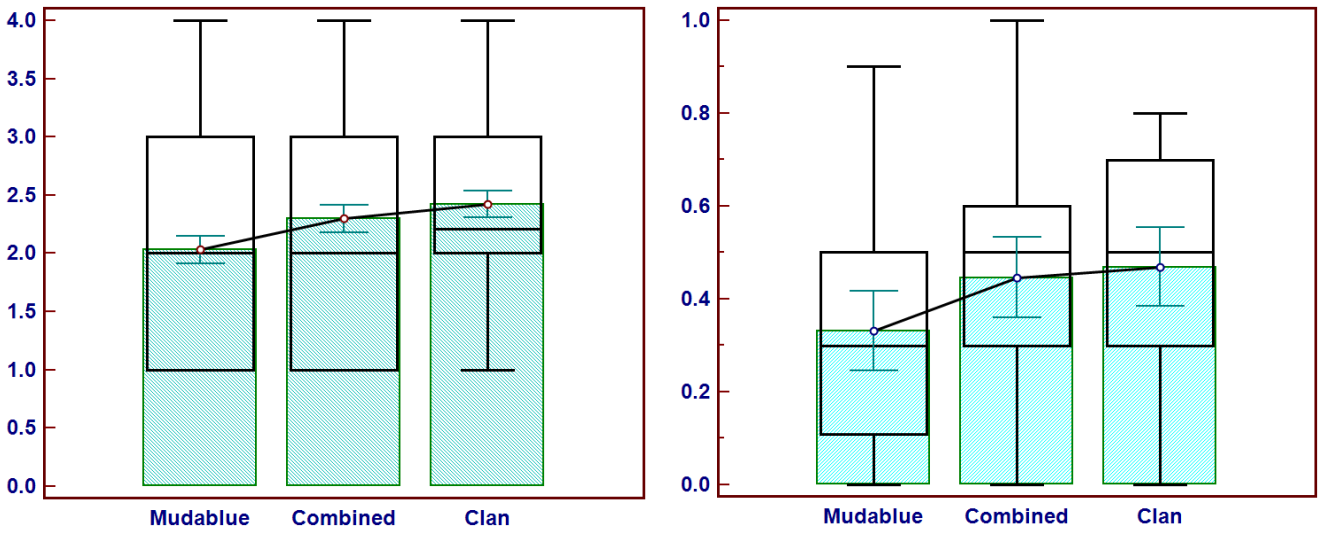
In this section, we discuss threats to the validity of this experimental design and how we address and minimize these threats.

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

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



(a) Confidence level, *C*.

(b) Precision, *P*.

Figure 2. Statistical summary of the results of the experiment 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.

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.

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.

V. Results

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

A. 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 IV-D.

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*.

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 2(a) and Figure 2(b) correspondingly with 95% confidence interval for the mean.

Table I
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) 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 I. 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**.

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 I. 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**.

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 I. 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 [34] and detecting duplicate bug reports [48].

VI. DISCUSSION

During the experiment, programmers identified more relevant applications using CLAN than when using MUDABlue (see Section V). This result points to a key advantage of CLAN: we help programmers effectively compare two applications by elevating highly-relevant details of these applications. Without CLAN, programmers must examine the whole source code of different applications in order to compare them. Consider the example in Figure 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.

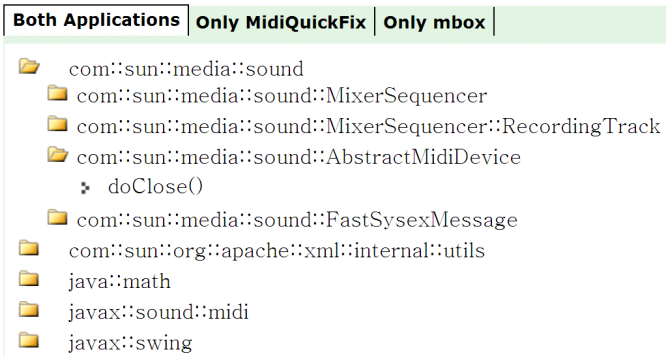


Figure 3. Part of the CLAN interface, showing the API calls common to two applications. CLAN shows these calls in order to help programmers concentrate on highly-relevant details when comparing applications.

VII. Related Work

The five most related tools to our work are those based on CodeWeb by Michail and Notkin [31], MUDABlue by Kawaguchi et al. [22], Hipikat by Cubranic and Murphy [47] and CodeBroker by Ye and Fischer [50] and SSI by Bajracharya, Ossher, and Lopez [2]. 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) [31]. 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 [2]. 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 [15], Mica [43], Exemplar [13], SNIFF [5], Prospector [27], Suade [38], Starthcona [16], XSnippet [40], ParseWeb [44], SPARS-J [20], Portfolio [29], Sourcerer [3], S6 [37] and SpotWeb [45]. 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 similarity 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 [42]. 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 [6]. Bertillonage is designed to locate duplicate code, however, and does not compute the similarity of software which may be related, but is not duplicated.

VIII. 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.

Acknowledgments

This work is supported by NSF CCF-0916139, NSF CCF-0916260, and NSF CCF-1016868. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

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] S. K. Bajracharya, J. Ossher, and C. V. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *FSE*, pages 157–166, 2010.

[3] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya. A theory of aspects as latent topics. In *OOPSLA '08*, pages 543–562, New York, NY, USA, 2008. ACM.

[4] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster. Program understanding and the concept assigment problem. *Commun. ACM*, 37(5):72–82, 1994.

[5] S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine using free-form queries. In *FASE*, pages 385–400, 2009.

[6] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle. Software bertillonage: finding the provenance of an entity. In *MSR'11*, pages 183–192, New York, NY, USA, 2011. ACM.

[7] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *JASIS*, 41(6):391–407, 1990.

[8] U. Dekel and J. D. Herbsleb. Improving api documentation usability with knowledge pushing. In *ICSE*, 2009.

[9] 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.

[10] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 147–156, New York, NY, USA, 2010. ACM.

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

[12] M. Grechanik, K. M. Conroy, and K. Probst. Finding relevant applications for prototyping. In *MSR*, page 12, 2007.

[13] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. M. Cumby. A search engine for finding highly relevant applications. In *ICSE (1)*, pages 475–484, 2010.

[14] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi-Reghizzi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi. An empirical investigation into a large-scale java open source code repository. In *ESEM*, 2010.

[15] S. Henninger. Supporting the construction and evolution of component repositories. In *ICSE '96*, pages 279–288, Washington, DC, USA, 1996. IEEE Computer Society.

[16] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE*, 2005.

[17] J. Howison and K. Crowston. The perils and pitfalls of mining Sourceforge. In *MSR*, 2004.

[18] X. Hu, Z. Cai, A. C. Graesser, and M. Ventura. Similarity between semantic spaces. In *CogSci'05*, 2005.

[19] E. Hull, K. Jackson, and J. Dick. *Requirements Engineering*. SpringerVerlag, 2004.

[20] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component rank: relative significance rank for software component search. In *ICSE '03*, pages 14–24. IEEE Computer Society, 2003.

[21] C. Jones. *Applied software measurement: assuring productivity and quality*. McGraw-Hill, Inc., 3rd edition, 2008.

[22] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. Mudablue: an automatic categorization system for open source repositories. *J. Syst. Softw.*, 79(7):939–953, 2006.

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

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

[25] C. Liu, C. Chen, J. Han, and P. S. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *KDD'06*, pages 872–881. ACM Press, 2006.

[26] W. Liu, K.-Q. He, J. Wang, and R. Peng. Heavyweight semantic inducement for requirement elicitation and analysis. *Semantics, Knowledge and Grid*, 0:206–211, 2007.

[27] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI '05*, 2005.

[28] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[29] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: Finding relevant functions and their usages. In *ICSE'11*, 2011.

[30] C. McMillan, M. Linares-Vasquez, D. Poshyvanyk, and M. Grechanik. Categorizing software applications for maintenance. In *ICSM'11*, 2011.

[31] A. Michail and D. Notkin. Assessing software libraries by browsing similar classes, functions and relationships. In *ICSE '99*, pages 463–472, New York, NY, USA, 1999. ACM.

[32] S. Mizzaro. Relevance: The whole history. *JASIS*, 48(9):810–832, 1997.

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

[34] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Software Eng.*, 33(6):420–432, 2007.

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

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

[37] S. P. Reiss. Semantics-based code search. In *ICSE '09*, pages 243–253, Washington, DC, USA, 2009.

[38] M. P. Robillard. Automatic generation of suggestions for program investigation. In *ESEC/FSE-13*, pages 11–20, New York, NY, USA, 2005. ACM.

[39] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer. Detecting similar java classes using tree algorithms. In *MSR '06*, pages 65–71, New York, NY, USA, 2006. ACM.

[40] N. Sahavechaphan and K. Claypool. Xsnippet: mining for sample code. In *OOPSLA '06*, pages 413–430, New York, NY, USA, 2006. ACM.

[41] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1986.

[42] D. Schuler, V. Dallmeier, and C. Lindig. A dynamic birthmark for java. In *ASE '07*, pages 274–283.

[43] 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.

[44] 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.

[45] 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, 2008.

[46] C. J. van Rijsbergen. *Information Retrieval*. Butterworth, 1979.

[47] D. Čubranić and G. C. Murphy. Hipikat: recommending pertinent software development artifacts. In *ICSE '03*, pages 408–418, 2003.

[48] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *ICSE*, pages 461–470, 2008.

[49] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*. Morgan Kaufmann, 1999.

[50] Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. In *ICSE '02*, pages 513–523, New York, NY, USA, 2002. ACM.