CLAN: Closely reLated ApplicatioNs

Collin McMillan
College of William & Mary
Williamsburg, VA 23185
cmc@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. Knowing similarity between applications plays an important role in improving understanding and assessing reusability of these applications, rapid prototyping, and discovering code theft and plagiarism. However, 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 present our approach for automatically detecting Closely reLated ApplicationNs (CLAN) for a given Java application. CLAN compares applications based on the calls they make to the standard Java Application Programming Interface (API). Our idea is that similar applications will make similar API calls. CLAN is publicly available via a standard web browser; users may view similar applications as well as the API calls they share.

1. INTRODUCTION

Retrieving similar or related web pages is a new 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 them and their positions and sizes [4]. For example, for the main VSGC page, Google returns three top similar web sites: Washington Space Grant Consortium, Wyoming Space Grant Consortium, and the NASA Langley Research Center.

Detecting similar applications is a notoriously difficult problem, since it means automatically detecting that high-level requirements for these applications match semantically [10, pages 74,80][15]. This situation is aggravated by the fact that many application repositories are polluted with poorly functioning projects [8]; 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. 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 [13, 14, 18, 22, 24]. Enabling programmers to compare automatically how different applications implement the same requirements makes it easier to acquire knowledge about these requirements and subsequently to decisions that these developers make about code reuse. Programmers can concentrate on the relevant aspects of functionality if they can see a list of relevant applications.

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 Application Programming Interface (API) whose semantics are well-defined. The intuition behind our approach is that if two applications make API calls that implement the same requirement (e.g., 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 [2, 5]; however, this idea has never been used to compute similarities among software applications.

We implemented CLAN and applied it to 8,310 Java applications that we downloaded from SourceForge. We created a visual interface that enables users to quickly evaluate how applications are similar by viewing the overlap of API calls each makes. CLAN is publicly available via a standard web browser1.

2. CLAN APPROACH

Our key idea is threefold. First, if two applications share some API calls, then their similarity index should be higher than for applications that do not share any API calls. Sharing API calls 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 a 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 JDK to improve the precision of computing similarity. This idea is related to semantic spaces where

1http://www.javaclan.net/
3. IMPLEMENTATION

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 [3]. In LSI, terms are elevated to an abstract space, and terms that are used in documents are implemented in the underlying corpus of documents [3]. In LSI, similarity space while simultaneously revealing latent concepts that are organized in structured layers and similarity scores between documents are computed using relations between layers [9]. 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.

4. CLAN USAGE

Consider the following example task for a programmer: “Create an application that records musical data from an electronic instrument into a MIDI file.”2 After the user submits a search query that contains key words record, MIDI, and file, the search engine retrieves a list of applications that are relevant to these key words. This initial list is created using the Exemplar search engine [5]. In this example, the application MidiQuickFix may pique the user’s interest. After clicking on the link that represents this application, CLAN presents the user with a list of similar applications ranked in the descending order. Once the user selects a relevant application and clicks on it, CLAN presents this user with the visual interface a part of which is shown in Figure 2. This interface shows three tabs: the leftmost tab (it is shown active in Figure 2) presents common packages, classes, and API calls in both applications, and two other tabs that present

\[ S_{ij} = \begin{cases} 0 \leq s \leq 1, & \text{if } i \neq j, \\ 1, & \text{if } i = j \end{cases} \]

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).

We found questions about this task in different programming forums and newsgroups, for example http://forums.sun.com/thread.jspa?threadID=5432750.
Comparing applications directly based on functionally related API calls helps programmers to concentrate on highly related details rather than examine the entire source code.

Specifically, the top similar application that CLAN retrieved is `mbox`, a command-line utility to convert MIDI files to mappings of music box drums. Packages `com.sun.media.sound` and `javax.sound.midi` are shown in Figure 2 as common for both applications `MidiQuickFix` and `mbox`. When expanded, common classes and API calls are shown to the user, for example, the class `AbstractMidiDevice` whose API call `doClose` is invoked in both applications.

5. EVALUATION

Typically, search and retrieval engines are evaluated using manual relevance judgments by experts [17, 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.

5.1 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.

5.2 Hypotheses

We introduce the following null and alternative hypotheses to evaluate how close the means are for the Cs for control and treatment groups, where C is the confidence level assigned by programmers in the user study. 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 per task between participants who use MUDABlue, Combined, and CLAN.
An alternative hypothesis to \( H_0 \) is that there is statistically significant difference in the values of confidence level between participants who use MUDABlue, Combined, and CLAN.

Once we test the null hypothesis \( H_0 \), we are interested in the directionality of means, \( \mu \), 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, \( C \).

\[ H_1: C \text{ of CLAN versus MUDABlue.} \]

\[ H_2: C \text{ of CLAN versus Combined.} \]

\[ H_3: C \text{ of MUDABlue versus Combined.} \]

The rationale behind the alternative hypothesis to \( H_1 \) 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 hypothesis to \( H_2 \) is 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 confidence with which users evaluate retrieved similar applications. Finally, having the alternative hypothesis to \( H_3 \) 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.

### 5.2.1 Comparing MUDABlue with CLAN

To test the null hypothesis \( H_1 \), we applied two t-tests for paired sample for means, for \( C \) for participants who used MUDABlue and CLAN. Based on these results we reject the null hypotheses \( H_1 \), and find that participants who use CLAN report higher relevance on finding similar applications than those who use MUDABlue.

### 5.2.2 Comparing CLAN with Combined

To test the null hypothesis \( H_3 \), we applied two t-tests for paired sample for means, for \( C \) for participants who used the baseline CLAN and Combined. Based on these results we accept the null hypothesis \( H_3 \) that say that participants who use CLAN do not report higher relevance 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 [19] and detecting duplicate bug reports [29].

### 6. RELATED WORK

Source code search engines have become an active research area in the recent years, including CodeFinder [6], Mica [25], Exemplar [5], SNIFF [2], Prospector [16], Suade [21], Starthcona [7], XSnippet [23], ParseWeb [26], SPARS-J [11], Sourcerer [1], S6 [20] and SpotWeb [27]. These source code mining techniques and tools have been proposed to search for relevant components from software repositories in response to user queries. The feature of retrieving “similar pages”, however, has been recently made available only in the realm of Web text search engines [4].

The four most-related tools to our work are those based on CodeWeb by Michail and Notkin [18], MUDABlue by Kawaguchi et al. [12], Hipikat by Cubrnican and Murphy [28] and CodeBroker by Ye and Fischer [30]. CodeWeb is an automated approach for comparing and contrasting software libraries based on matching similar classes and functions across libraries (via name and similarity matching) [18]. CodeBroker [30] and Hipikat [28] are related approaches that use source code and comments written by programmers to query code repository to find relevant artifacts. MUDABlue [12] is a solution for automatic categorization of software systems in large repositories. CLAN differs from all four approaches in that it uses API calls to compute the similarities among applications in a repository, rather than returning portions of applications for reuse or categorizing those applications.

### 7. 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 applied it to a set of 8,310 Java applications downloaded from Sourceforge. Users of CLAN can quickly evaluate the similarity of applications by viewing which API calls that applications have in common. CLAN is available for public use.

### 8. REFERENCES


