Cross-Library API Recommendation using Web Search Engines

Wujie Zheng, Qirun Zhang, Michael Lyu
Computer Science and Engineering
The Chinese University of Hong Kong, China
{wjzheng,qrzhang,lyu}@cse.cuhk.edu.hk

ABSTRACT

Software systems are often built upon third party libraries. Developers may replace an old library with a new library, for the consideration of functionality, performance, security, and so on. It is tedious to learn the often complex APIs in the new library from the scratch. Instead, developers may identify the suitable APIs in the old library, and then find counterparts of these APIs in the new library. However, there is typically no such cross-references for APIs in different libraries. Previous work on automatic API recommendation often recommends related APIs in the same library. In this paper, we propose to mine search results of Web search engines to recommend related APIs of different libraries. In particular, we use Web search engines to collect relevant Web search results of a given API in the old library, and then recommend API candidates in the new library that are frequently appeared in the Web search results. Preliminary results of generating related C# APIs for the APIs in JDK show the feasibility of our approach.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Documentation, Reliability

1. INTRODUCTION

Code reuse is one of the primary techniques to improve the productivity of building software systems. Due to the requirement changes of a software system, developers may need to replace an old library with a new library. Working with complex APIs in the new library presents many barriers, such as selecting the appropriate APIs and figuring out how to use the selected APIs. Since the developers often can identify the suitable APIs in the old library easily, a possible solution is to find counterparts of the old APIs in the new library. However, there is typically no such cross-references for APIs in different libraries, and thus it can still take much time of developers to look for the suitable API counterparts.

There has been much work on automatically generating cross-references for APIs. Doxygen [4] and Javadoc [3] are well-known for generating cross-references and documentations from developers' comments in source code. However, there are few, if any, comments that talk about the relationship of APIs in different libraries. Many approaches recommend related APIs in the same library by analyzing the source code of the library [8, 9, 11] or the client code [7, 10, 13]. These approaches rely on the assumption that two related APIs are structurally or statistically connected in some source code, which are not valid for APIs in different libraries. Zhong et al. [12] mine API mappings from Java to C# from the same projects' different implementations in these two languages (based on these languages' basic libraries). However, for two arbitrary libraries, there may not be implementations of the same project in these two libraries, and the APIs used in such projects may be limited.

In this paper, we propose to mine search results of Web search engines to recommend related APIs of different libraries. The idea is that developers often share their knowledge of related APIs of different libraries in the Web. For example, the following question illustrates a case that developers ask about equivalent APIs in C# (more exactly, the .NET Framework) for the "HashMap" class in Java (more exactly, the Jave Development Kit) (http://stackoverflow.com/questions/1273139/c-java-hashmap-equivalent).

- "Coming from a Java world into a C# one is there a HashMap equivalent? If not what would you recommend?"

There can be many similar questions and discussions in the Web. However, such knowledge is often described in unstructured web pages that are spread in the Web. Moreover, the results can be different and many unrelated APIs may also be discussed. To mine the knowledge of developers for cross-library API recommendation, we use Web search engines to collect relevant Web search results of a given API query, and then extract API candidates that are frequently appeared in the Web search results. For a library, after getting the results of each API, we also re-rank the related APIs based on the whole distributions to reduce false positives. We have built a prototype for the proposed approach and applied it to generate related APIs in .NET Framework for the APIs in JDK.

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

2. RELATED WORK

In this section we discuss the related work, including the most related work by us and by others. We also point out the novelty of our approach.

There have been many approaches to recommend related APIs in the same library by analyzing the source code of the library [8, 9] or the client code [7, 10, 13]. Robillard [9] proposed an approach named Suade to automatically rank program elements for an investigation task based on the structural dependencies in a program. Long et al. [8] developed a tool Altair that ranks related API methods for a given method according to the share variables between methods. Li and Zhou [7] developed a tool named PR-Miner that uses frequent itemset mining to extract implicit API usage patterns from source code. Zhong et al. [13] proposed a tool named MAPO that combines the frequent subsequence mining technique with the clustering technique to mine code snippets with respect to programming contexts. Thummalapenta and Xie [10] developed a tool named PARSEWeb that uses Google code search engine [2] to collect relevant code snippets and then mines code snippets for a given target object type.

We [11] have augmented the call graph with control flow analysis to capture the significance of the caller-callee linkages for API recommendation. In the previous work, we focus on recommending related APIs in the same library using static analysis techniques. While in this work, we would like to recommend related APIs in different libraries using information retrieval and statistical analysis techniques. Both the usage scenarios and the required techniques are very different.

The single most related work by others is the MAM approach, proposed by Zhong et al. [12]. MAM mines API mappings from Java to C# from the same projects' implementations in these two languages. However, for any two languages, there may not be implementations of the same project in these two languages, the APIs used in such projects may be limited, and the mapping strategies could be difficult to design. German and Davies [6] describe several challenges of comparing the source code of different libraries. Differently, our approach recommends related APIs in different libraries by mining the knowledge of the library users shared in the Web.

Compared with the existing approaches, the new idea of our approach is to exploit the huge amount of knowledge of the library users shared in the World Wide Web. The knowledge can reveal API correlations cross libraries (in different programming languages). However, the knowledge is described in unstructured Web pages. New techniques are needed to collect relevant Web search results and to extract knowledge of the APIs from the noisy data.

3. APPROACH

3.1 Overview

We first describe the framework of mining Web search results for API recommendation. Given a library for reference, which the developers are familiar with, and a targeted library for studying, we would like to recommend similar APIs in the latter for each API in the former.

Figure 1 shows the procedure of our approach. First, for each API query, our approach constructs a Web query in the purpose of searching for related APIs in the targeted library.

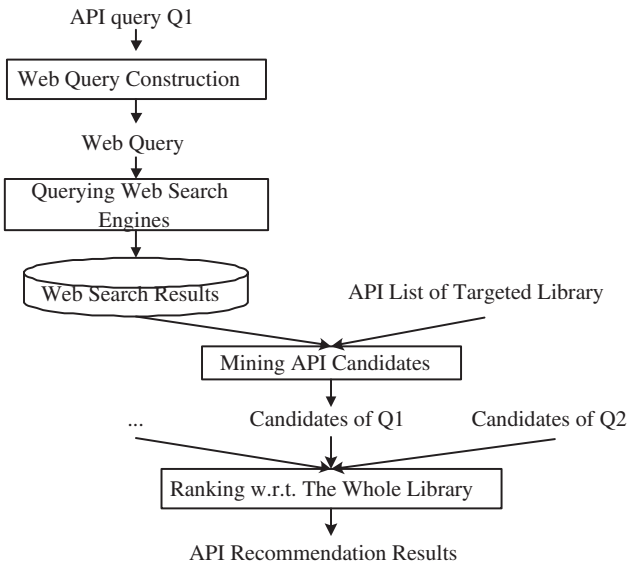

Figure 1: Overview of our approach

Our approach then queries a major Web search engine such as Google to get the top-ranked Web search results. Our approach then mines API candidates in the targeted library that often appear in the Web search results. Finally, out approach ranks the mined API candidates for each API query based on the whole distributions of API candidates, using the tf-idf definition in Information Retrieval community [5]. Currently the API lists of the reference library and the targeted library are both extracted manually. In Section 5 we shall discuss how to extract them automatically in future.

3.2 Web Query Construction

Developers often share their knowledge of related APIs of different libraries in the Web. However, this knowledge is often described in unstructured web pages that are spread in the Web. Web search engines are the best tools to collect such information automatically. Given a API query, our approach constructs a Web query by concatenating the given API and the targeted library (or program language). Take the HashMap class of Java Development Kits (JDK) as an example. To find similar APIs of it in the C# language (i.e., in the .NET Framework), our approach constructs a Web query "HashMap C#" for the Web search engines.

It is also possible to mine the common words used for querying equivalent APIs and add them to the Web query. Moreover, we may use multiple Web queries for a given API, and merge the Web search results for mining related APIs.

3.3 Querying Web Search Engines

We applied the Web services of Google to collect the top search results of a Web query. Other major search engines such as Bing and Yahoo also provide Web services for Web search. We use Web search engines (text retrieval) instead of code search engines, because an API in a library and its similar API in another library are seldom used together in some code. The search results of "HashMap C#" returned by the Google search engine are shown in Figure 2. From the example search results, we can see that developers discuss the

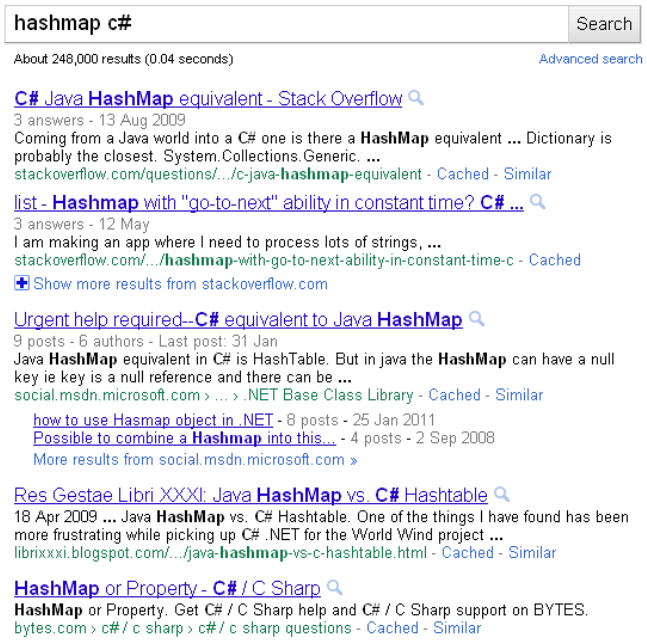

Figure 2: Search results of HashMap

equivalent of HashMap in C# in different forums. Moreover, the results can be different. For example, the first result says that the Dictionary class is preferable, while the third result says that "Java HashMap equivalent in C# is HashTable".

3.4 Mining API Candidates

Different APIs have different naming conventions. It is difficult to distinguish the APIs from the general English words. For mining API candidates from the Web search results, we employ a dictionary-based approach.

In the context of API recommendation, developers often know what the targeted library is. That is, the API candidates for a API query should be in the list of all APIs in the targeted library. Therefore, we extract all APIs in the targeted library, e.g., .NET Framework, as the dictionary. We then calculate the frequency of each API of the targeted library in the title and the summary of Web search results. Basically, the higher frequency a API candidate is, the more likely it is relevant to the API query.

3.5 Ranking w.r.t. the Whole Library

Some common APIs, such as the Object Class, may appear frequently in the Web search results. However, these common APIs are not interesting for a API query. Our approach uses the definitions of tf-idf from the Information Retrieval community [5] to distinguish these common APIs from the true relevant APIs.

Recall that we have mined API candidates for each API query. That is, for an API query Q_i, we have a set of API candidates, $T1$, $T2$, etc. We can thus build a set of documents, where each document contains the API candidates (with duplicates) for each API query. Figure 3 shows an example, where $T1$ appears in one search results of three API queries and $T5$ appears in two search results of the API query $Q3$.

Q1: T1, T2, T3
Q2: T1, T4
Q3: T1, T5, T2, T5
Q4: T6

Figure 3: Example documents of API candidates

We then use the definitions of tf and idf to identify the relevance of a API candidate to the API query. The term frequency (tf) is the number of a API candidate appeared in the document of a API query. The larger it is, the more relevant the API candidate is to the API query. The inverse document frequency (idf) is obtained by dividing the total number of documents by the number of documents containing the API candidate, and then taking the logarithm of that quotient. The larger it is, the less common the API candidate is and thus the more relevant the API candidate is to the API query. In particular, the relevance of a API candidate T to a API query Q is defined as follows.

$$relevance(T, Q) = tf(T, Q) * idf(T)$$

where $tf(T, Q)$ is the frequency of the API candidate T in the document of Q, and $idf(T)$ is the inverse document frequency of T in the whole corpus, i.e., the set of documents for all API queries.

Finally, for each API query, the candidate APIs are ranked in the descending order of the relevance to the API query.

3.6 Explanation Generation

It is often not enough to tell developers only *which API is relevant to a given query*. Developers need more information to understand *why and how the recommended API is relevant to a given query*. To suit this need, our approach also generates explanations for each recommended API, so as to help developers understand the causality. Currently, for each recommended API, our approach lists the Web pages (within the Web search results) that discuss both the recommended API and the API query. The Web pages often contain the users' discussions or explanations of how an API is relevant to a given query. It is also possible to generate other kinds of explanations, such as the formal specification of the recommended APIs.

4. PRELIMINARY RESULTS

We have built a prototype for the proposed approach and applied it to generate related APIs in C# (mostly in the .NET Framework) for the APIs in JDK. The Java Development Kit (JDK) is a Sun Microsystems product aimed at Java developers. Since the introduction of Java, it has been by far the most widely used Java SDK. C# is a multi-paradigm programming language encompassing imperative, declarative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines. It was developed by Microsoft within the .NET initiative. We focus on recommending the classes in the experiment. We crawl the API list of JDK and C# from their official Websites. The characteristics of these two subject libraries are summarized in Table 1.

For each API query in JDK, our approach generates an individual result page that lists the recommended APIs in C#.

Table 1: Subject programs

Library	#Packages	#Classes
JDK	135	2723
.NET	408	13791

- Hashtable Res Gestae Libri XXXI: Java HashMap vs. C# Hashtable
- Dictionary C# Java HashMap equivalent - Stack Overflow
- Stack C# Java HashMap equivalent - Stack Overflow
- ComVisibleAttribute Hashtable Class (System.Collections)
- Expression Expression evaluator - ANTLR 3 - ANTLR Project

(a) Recommended APIs for HashMap

- WebRequest C# I/O and Networking - O'Reilly Media
- Stack C# connect to URL which gives a xml document - Stack Overflow
- Network URL Connection Test : Server Â« Network Protocol Â« Java
- FtpWebRequest Win32 Networks C++ class/library that implements TCP FTP ...
- WebException [ASP .NET C#] Load a local webpage from code behind - Dev Shed

(b) Recommended APIs for URLConnection

Figure 4: API recommendation results

Figure 4 shows the recommendation results for *java.util.HashMap*. The first recommended APIs in C# include *Hashtable* and *Dictionary*. These two APIs are similar to *HashMap* as they are all collections that maps keys to values. Along with each recommended API, our approach provides a link (and the title) to the explanation of their correlations with the API query. The link for *Hashtable* points to a blog that compares *HashMap* and *Hashtable*, while the link for *Dictionary* points to a discussion in a question and answer site. These explanations not only increase the confidence of users for the recommendation results, but also help users learn the key similarities and differences between the recommended APIs and the API query. Figure 4 also shows the results for *java.net.URLConnection*. The first recommended API is *WebRequest*. It is relevant to *URLConnection* because they are both used to communicate to a URL. The link for *WebRequest* points to an article that describes C# I/O and networking APIs, compared with the Java ones. Finally, our approach generates an index html for the JDK library, where each entry links to the recommendation results of a API query.

5. DISCUSSIONS

5.1 Keyword-based Web Query Construction

While users may talk about the names of a API query and a target library, a more common scenario is that users discuss the functionality of a API query and a target library. Therefore, we may extract the keywords of a API based on its documentation or the search results of the API, and then construct a Web query that consists of the keywords and the name of the target library.

5.2 API Candidates Extraction

Currently, our approach uses a dictionary-based approach to extract API candidates for recommendation from Web search results. The dictionary is the list of APIs in the targeted library. Having such a API list can help to reduce false positives effectively. However, it could be tedious for the users to create such a API list manually. A possible solution is to analyze the source code of a given library to extract API lists, using tools such as Ctags [1]. Another possible solution is to use a template-based approach. That is, instead of giving a API list, the users provide a template for the potentially interested APIs, using regular expressions.

6. CONCLUSIONS

Sometimes developers need to replace an old library with a new library for more effective code reuse. Learning the often complex APIs in the new library could be tedious. We propose an approach to help developers find APIs in the new library that work similarly to a given API in the old library automatically. In particular, we use Web search engines to collect relevant Web search results of a given API in the old library, and then recommend API candidates in the new library that are frequently appeared in the Web search results. We have built a prototype for the proposed approach and applied it to generate related C# APIs for the APIs in JDK. We have also discussed several issues for further improvements.

7. ACKNOWLEDGMENTS

The work described in this paper was fully supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK 415410), and a grant under Google Focused Grant Project "Mobile 2014".

8. REFERENCES

[1] Ctags. `http://ctags.sourceforge.net/`.
[2] Google code search. `http://www.google.com/codesearch`.
[3] Javadoc. http://java.sun.com/j2se/javadoc/.
[4] Source code documentation generator tool. http://www.doxygen.org/.
[5] R. Baeza-Yates, B. Ribeiro-Neto, et al. *Modern information retrieval*. ACM press New York., 1999.
[6] J. Davies, D. M. Germán, M. W. Godfrey, and A. Hindle. Software bertillonage: finding the provenance of an entity. In *MSR*, pages 183–192, 2011.
[7] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/SIGSOFT FSE*, pages 306–315, 2005.
[8] F. Long, X. Wang, and Y. Cai. Api hyperlinking via structural overlap. In *ESEC/SIGSOFT FSE*, pages 203–212, 2009.
[9] M. P. Robillard. Automatic generation of suggestions for program investigation. In *ESEC/SIGSOFT FSE*, pages 11–20, 2005.
[10] S. Thummalapenta and T. Xie. PARSEWeb: a programmer assistant for reusing open source code on the web. In *ASE*, pages 204–213, 2007.
[11] Q. Zhang, W. Zheng, and M. R. Lyu. Flow-augmented call graph: A new foundation for taming api complexity. In *FASE 2011, Fundamental Approaches to Software Engineering*, pages 386–400, 2011.
[12] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining api mapping for language migration. In *ICSE (1)*, pages 195–204, 2010.
[13] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending api usage patterns. In *ECOOP*, pages 318–343, 2009.