Exemplar: EXEcutable exaMPLes ARchive

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

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

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

ABSTRACT

Searching for applications that are highly relevant to development tasks is challenging because the high-level intent reflected in the descriptions of these tasks doesn't usually match the low-level implementation details of applications. In this demo we show a novel code search engine called *Exemplar (EXEcutable exaMPLes ARchive)* to bridge this mismatch. Exemplar takes natural-language query that contains high-level concepts (e.g., MIME, data sets) as input, then uses information retrieval and program analysis techniques to retrieve applications that implement these concepts.

Categories and Subject Descriptors

D.2.13 [**Reusable Software**]: Reusable libraries

General Terms

Algorithms, Experimentation

1. INTRODUCTION

Creating software from existing components is a fundamental challenge of software reuse. It is estimated that around one trillion lines of code have already been written with an additional 35 billion lines of source code being written every year [3]. Reusing fragments of existing applications is beneficial to programmers because complete applications provide them with the contexts in which these fragments exist. Unfortunately, a few major challenges make it difficult to locate existing applications that contain relevant code fragments.

First and foremost, bridging the mismatch between the high-level intent reflected in the descriptions of these applications and low-level implementation details is hard. This problem is known as the *concept assignment problem* [2]. Many search engines match keywords in queries to words in the descriptions of the applications, comments in their source code, and the names of program variables and types. If no match is found, then potentially relevant applications are never retrieved from repositories. This situation is aggravated by the fact that many application repositories are polluted with poorly functioning projects [11]; a match between a keyword from the query with a word in the description or in the source code of an application does not guarantee that this application is relevant to the query.

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

Finally, short code snippets that are returned as results to user queries (as many existing code search engines do) do not give enough background or context to help programmers determine how to reuse these snippets, and programmers typically invest a significant intellectual effort (i.e., they need to overcome a high cognitive distance [12]) to understand how to use these code snippets in larger scopes. On the contrary, if code fragments are retrieved in the contexts of applications, it makes it easier for programmers to understand how to reuse these code fragments.

We demonstrate a novel code search engine called *Exemplar (EXEcutable exaMPLes ARchive)* that helps users to find highly relevant executable applications for reuse. Exemplar combines information retrieval and program analysis techniques to reliably link high-level concepts to the source code of the applications via standard third-party *Application Programming Interface (API)* calls that these applications use. Exemplar is available for public use[1].

2. EXEMPLAR APPROACH

In this section we describe the key ideas and give intuition about why Exemplar works followed by the architecture of Exemplar.

2.1 Key Ideas

We want to mimic and automate parts of the human-driven procedure of searching for relevant applications. Suppose that requirements specify that a program should encrypt and compress data. When retrieving sample applications from Sourceforge[2] using the keywords `encrypt` and `compress`, programmers look at the source code to check to see if some API calls from third-party packages are used to encrypt and compress data. Even though the presence of these API calls does not guarantee that the applications are relevant, it is a good starting point for deciding whether to check these applications further.

What we seek is to augment standard code search to include API documentations of widely used libraries, such as standard *Java Development Kit (JDK)*. Of course, existing engines allow users to

[1]http://www.xemplar.org
[2]http://sourceforge.net/ as of September 6, 2009.

search for specific API calls, but knowing in advance what calls to search for is hard. Our idea is to match keywords from queries to words in help documentation for API calls in addition to finding keyword matches in the descriptions and the source code of applications. When programmers read these help documents about API calls that implement certain high-level concepts, they trust these documents because they come from known and respected vendors, were written by different people, reviewed multiple times, and have been used by other programmers who report their experience at different forums. Help documents are more verbose and accurate, and consequently trusted more than the descriptions of applications from repositories [5].

In addition, we observe that relations between concepts entered in queries are often preserved as dataflow links between API calls that implement these concepts in the program code. This observation is closely related to the concept of the *software reflexion models*, formulated by Murphy, Notkin, and Sullivan, where relations between elements of high-level models (e.g., processing elements of software architectures) are preserved in their implementations in source code [15]. For example, if the user enters keywords `secure` and `send`, and the corresponding API calls `encrypt` and `email` are connected via some dataflow, then an application with these connected API calls are more relevant to the query than ones where these calls are not connected.

Consider, for example, two API calls `string encrypt()` and `void email(string)`. After the call `encrypt` is invoked, it returns a string that is stored in some variable. At some later point a call to the function `email` is made and the variable is passed as the input parameter. In this case we say that these functions are connected using a dataflow link which reflects the implicit logical connection between keywords in queries, specifically, the data should be encrypted and then sent to some destination.

To improve the precision of our approach, our idea is to determine relations between API calls in retrieved applications. All things equal, if a dataflow link is present between two API calls in the program code of one application and there is no link between the same API calls in some other application, then the former application should have a higher ranking than the latter. In addition, knowing how API calls are connected using dataflows enables programmers to better understand the contexts of the code fragments that contain these API calls. Finally, it is possible to utilize dataflow connections to extract code fragments, which is a subject of our future work on our S^3 architecture [17].

2.2 Our Approach

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

Consider the process for standard search engines (e.g., Sourceforge, Google code search) shown in Figure 1(a). A keyword from the query is matched against words in the descriptions of the applications in some repository (Sourceforge, Krugle) or words in the entire corpus of source code (Google Code Search). When a match is found, applications $\texttt{app}_1$ to $\texttt{app}_n$ are returned.

Consider the process for Exemplar shown in Figure 1(b). A keyword from the query is matched against the descriptions of different documents that describe API calls of widely used software packages. When a match is found, the names of the API calls $\texttt{call}_1$ to $\texttt{call}_k$ are returned. These names are matched against the names of the functions invoked in these applications. When a match is found, applications $\texttt{app}_1$ to $\texttt{app}_n$ are returned.

A fundamental difference between these search schemes is that Exemplar uses help documents to obtain the names of the API calls in response to user queries. Doing so can be viewed as instances of the *query expansion* concept in information retrieval systems [1] and *concept location* [14]. The aim of query expansion is to reduce this query/document mismatch by expanding the query with concepts that have similar meanings to the set of relevant documents. Using help documents, the initial query is expanded to include the names of the API calls whose semantics unequivocally reflects specific behavior of the matched applications.

In addition to the keyword matching functionality of standard search engines, Exemplar matches keywords with the descriptions of the various API calls in help documents. Since a typical application invokes API calls from several different libraries, the help documents associated with these API calls are usually written by different people who use different vocabularies. The richness of these vocabularies makes it more likely to find matches, and produce API calls $\texttt{API call}_1$ to $\texttt{API call}_k$. If some help document does not contain a desired match, some other document may yield a match. This is how we address the vocabulary problem [6].

As it is shown in Figure 1(b), API calls $\texttt{API call}_1$, $\texttt{API call}_2$, and $\texttt{API call}_3$ are invoked in the $\texttt{app}_1$. It is less probable that the search engine fails to find matches in help documents for all three API calls, and therefore the application $\texttt{app}_1$ will be retrieved from the repository.

Searching help documents produces additional benefits. API calls from help documents are linked to locations in the project source code where these API calls are used thereby allowing programmers to navigate directly to these locations and see how high-level concepts from queries are implemented in the source code. Doing so solves an instance of the concept assignment problem [2].

2.3 Exemplar Architecture

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

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

Exemplar works as follows. The input to the system are help documents describing various API calls `(1)`. The Help Page Processor indexes the description of the API calls in these help documents and outputs the API Calls Dictionary, which is the set of tuples $<<\texttt{word}_1, \ldots, \texttt{word}_n>, \texttt{API call}>$ linking selected words from the descriptions of the API calls to the names of these API calls `(2)`. Our approach for mapping words in queries to API

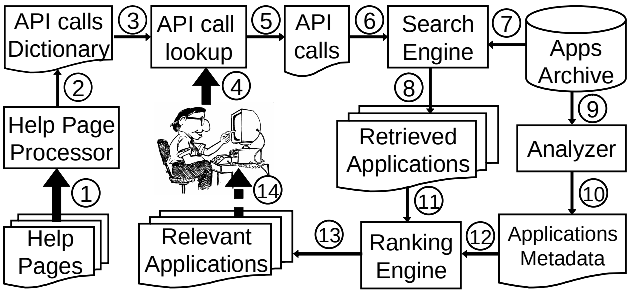
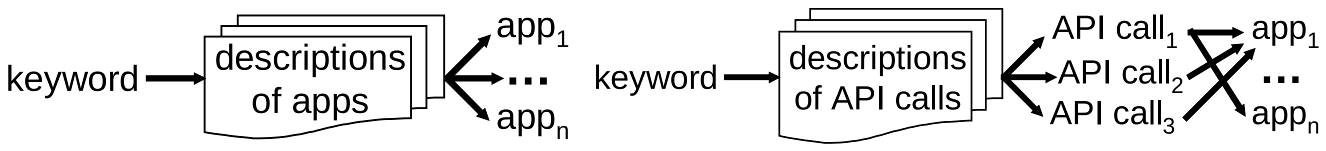


Figure 2: Exemplar architecture.



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

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

calls is different from the *keyword programming* technique [13], since we derive mappings between words and APIs from external documentation rather than source code.

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

The Analyzer pre-computes the Applications Metadata `(10)` that contains dataflow links between different API calls from the applications source code (9). Since this is done offline, precise program analysis can be accommodated in this framework to achieve better results in dataflow ranking. This metadata is supplied to the Ranking Engine `(12)` along with the Retrieved Applications `(11)`, and the Ranking Engine combines keyword matching score with API call scores to produce a unified rank for each retrieved application. Finally, the engine sorts applications using their ranks and it outputs Relevant Applications `(13)`, which are returned to the user `(14)`.

3. EXEMPLAR IMPLMENTATION

As mentioned in Section 2.3, we need to implement several major components of Exemplar: (1) crawler that populates the applications archive, (2) analyzer that pre-computes the dataflow between different API calls from the applications, and (3) ranking engine that sorts the applications based on the computed ranking scores.

3.1 Crawlers

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

Both crawlers are multithreaded, the Archiver is written in Scala and the Walker is written in Java. Currently, it takes approximately 35 hours for the four-threaded Walker to run through all applications in the Exemplar repository without any dataflow computation payload (only unpack each project) on a computer with Intel Core Quad CPU Q9550 at 2.83GHz.

3.2 Dataflow Computation

Our approach relies on the tool PMD [3] for computing approximate dataflow links, which are based on patterns of dataflow dependencies. Using these patterns it is possible to recover a large number of possible dataflow links between API calls; however, some of these recovered links can be false positives. In addition, we currently recover links among API calls within files (intraprocedurally), hence it is likely that some intraprocedural links are missed and no interprocedural analyses are performed.

[3]http://pmd.sourceforge.net/ as of September 6, 2009.

3.3 Computing Rankings

We use the Lucene search engine[4] to implement the core retrieval based on keyword matches. We indexed descriptions and titles of Java applications, and independently we indexed Java API call documentation by duplicating descriptions about the classes and packages in each methods. Thus when users enter keywords, they are matched separately using the index for titles and descriptions and the index for API call documents. As a result, two lists are retrieved: the list of applications and the list of API calls. Each entry in these lists are accompanied by a rank (i.e., conceptual similarity, C, a float number between 0 and 1).

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

4. EXEMPLAR USAGE

After users go to the Exemplar website they are presented with a Google-like interface that includes a text box for entering query keywords and two buttons. One says `Exemplar Search` and it is associated with the functionality to retrieve applications using the basic keyword search, API and the connectivity ranking. The other button is `I Am Feeling Lucky` and it is associated with functionality to retrieve applications using the basic keyword search and API ranking, no connectivity is used in ranking.

Users can type in any keywords in the search box and click on either of the above search buttons. Retrieved applications are sorted based on their relevance scores and presented on the next page using two kinds of links. If a link is presented with a button, it means that relevant API calls are located within this application. Clicking on this link leads users to a web page with a list of API calls, names of application files in which these calls are located, and the line numbers on which these calls can be found. Otherwise, the link takes the user directly to the Sourceforge page where this application is hosted. It is needless to say that if no source code is uploaded for an application, it is not presented on the list, which helps users to reduce the clutter introduced by irrelevant applications.

5. RELATED WORK

[4]http://lucene.apache.org/ as of September 6, 2009.

Approach	Granularity		Corpora	Query Expansion
	Search	Input		
CodeFinder	M	C	D	Yes
CodeBroker	M	C	D	Yes
Mica	F	C	C	Yes
Prospector	F	A	C	Yes
Hipikat	A	C	D,C	Yes
xSnippet	F	A	D	Yes
Strathcona	F	C	C	Yes
AMC	F	C	C	No
Google	F,M,A	C,A	D,C	No
Sourceforge	A	C	D	No
SPARS-J	M	C	C	No
Sourcerer	A	C	C	No
CodeGenie	A	C	C	No
SpotWeb	M	C	C	Yes
ParseWeb	F	A	C	Yes
S^6	F	C,A,T	C	Manual
Krugle	F,M,A	C,A	D,C	No
Koders	F,M,A	C,A	D,C	No
Exemplar	F,M,A	C,A	D,C	Yes

Table 1: Comparison of Exemplar with other related approaches. Column `Granularity` specifies how search results are returned by each approach (Fragment of code, Module, or Application), and how users specify queries (Concept, API call, or Test case). The column `Corpora` specifies the scope of search, i.e., Code or Documents, followed by the column `Query Expansion` that specifies if an approach uses this technique to improve the precision of search queries.

Different code mining techniques and tools have been proposed to retrieve relevant software components from different repositories as shown in Table 1.

In a nutshell, the differences between Exemplar and other search engines are listed below:

1. search engines that are heavily dependent on the descriptions of software projects/components and/or meaningful names of program variables and types, such as CodeFinder[9], and Codebroker[21];

2. search engines that do not return relevant applications, but code fragments or development artifacts, such as AMC[10], and XSnippet[19];

3. search engines that do not expand the query (automatically), such as SourcererDB[16], S^6[18], and most widely used open source projects repositories like Sourceforge, Google, etc.

Even though it returns code snippets rather than applications, Mica is the most relevant work to Exemplar [20]. In using help pages, Mica is similar to Exemplar, however, Mica uses help documentation to refine the results of the search while Exemplar uses help pages as an integral instrument in order to expand the range of the query. In addition, Exemplar returns executable projects while Mica returns code snippets as well as non-code artifacts.

SNIFF extends our original idea [7] of using API calls as basic code search abstractions [4]. Exemplar's internals differ substantially from previous attempts to use API calls for searching, including SNIFF: our search results contain multiple levels of granularity, and we are not tied to a specific IDE.

6. CONCLUSION

We offer a novel code search engine called Exemplar for finding highly relevant software projects from a large archive of executable examples. We evaluated Exemplar with 39 professional Java programmers and found with strong statistical significance that it performed better than Sourceforge in terms of reporting higher confidence levels and precisions for retrieved Java applications [8]. To our knowledge, it is the first attempt to combine program analysis techniques with information retrieval to convert high-level user queries to basic functional abstractions that are used automatically in code search engines to retrieve highly relevant applications.

Acknowledgments

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

7. REFERENCES

[1] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.
[2] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster. Program understanding and the concept assigment problem. *Commun. ACM*, 37(5):72–82, 1994.
[3] G. Booch. Keynote speech: The complexity of programming models. In *AOSD*, page 1, 2005.
[4] S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine for java using free-form queries. In *FASE*, pages 385–400, 2009.
[5] U. Dekel and J. D. Herbsleb. Improving api documentation usability with knowledge pushing. In *ICSE*, pages 320–330, 2009.
[6] 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.
[7] M. Grechanik, K. M. Conroy, and K. Probst. Finding relevant applications for prototyping. In *MSR*, page 12, 2007.
[8] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby. A search engine for finding highly relevant applications. In *ICSE*, 2010.
[9] S. Henninger. Supporting the construction and evolution of component repositories. In *ICSE*, pages 279–288, 1996.
[10] R. Hill and J. Rideout. Automatic method completion. In *ASE*, pages 228–235, 2004.
[11] J. Howison and K. Crowston. The perils and pitfalls of mining Sourceforge. In *MSR*, 2004.
[12] C. W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.
[13] G. Little and R. C. Miller. Keyword programming in java. *Automated Software Engg.*, 16(1):37–71, 2009.
[14] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *ASE*, pages 234–243, 2007.
[15] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *SIGSOFT FSE*, pages 18–28, 1995.
[16] J. Ossher, S. Bajracharya, E. Linstead, P. Baldi, and C. Lopes. Sourcererdb: An aggregated repository of statically analyzed and cross-linked open source java projects. *MSR*, 0:183–186, 2009.
[17] D. Poshyvanyk and M. Grechanik. Creating and evolving software by searching, selecting and synthesizing relevant source code. In *ICSE Companion*, pages 283–286, 2009.
[18] S. P. Reiss. Semantics-based code search. In *ICSE*, pages 243–253, 2009.
[19] N. Sahavechaphan and K. T. Claypool. XSnippet: mining for sample code. In *OOPSLA*, pages 413–430, 2006.
[20] J. Stylos and B. A. Myers. A web-search tool for finding API components and examples. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 195–202, 2006.
[21] Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. In *ICSE*, pages 513–523, 2002.