Finding Relevant Applications For Prototyping

Mark Grechanik, Kevin M. Conroy, and Katharina A. Probst

Systems Integration Group, Accenture Technology Labs
Chicago, IL 60601
{mark.grechanik, kevin.m.conroy, katharina.a.probst}@accenture.com

Abstract

When gathering requirements for new software projects, it is often cost-effective to find similar applications that can be used as the basis for prototypes rather than building them from scratch. However, finding such sample applications can be difficult, often making prototyping time-consuming and expensive.

We offer a novel approach called Exemplar (EXEcutable exaMPLes ARchive) for finding highly relevant software projects from a large archive of executable applications. After a programmer enters a query that contains high-level concepts (e.g., toolbar, download, smart card), Exemplar uses information retrieval and program analysis to retrieve applications that implement these concepts. We hypothesize that Exemplar will be effective and efficient in helping programmers to quickly find highly relevant applications to support prototyping.

1 Introduction

In the spiral model of software development [3], first, requirements are described, and then a prototype is built and shown to different stakeholders in order to receive early feedback [4]. This feedback often leads to changes in the prototype and the original requirements as stakeholders refine their vision. When a substantial number of requirements change, the existing prototype is often discarded, a new one is built, and the cycle repeats.

Building prototypes repeatedly from scratch is expensive, considering that these prototypes are often discarded after receiving feedback from stakeholders. Finding existing applications that match requirements, and subsequently can be shown as prototypes, would reduce the cost of many software projects. Since prototypes are approximations of desired resulting applications, similar applications from software repositories can often be used as the basis for prototypes.

Hundreds of open source repositories and internal source control management systems contain hundreds of thousands of different applications. For example, Sourceforge.net reports that it hosts 137,750 applications as of January 1, 2007. Given that many different applications have already been written, some of these applications can serve as prototypes because they are relevant to requirements. However, finding relevant applications is often very difficult because many search engines match keywords in queries to 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. As a result, programmers often choose to build prototypes from scratch.

Exemplar (EXEcutable exaMPLes ARchive) is a mining system that helps users find executable applications for rapid prototyping and development. Exemplar combines information retrieval and program analysis techniques to reliably link high-level concepts specified in the project requirements to the source code of the applications. We utilize these links to find highly relevant applications as well as fragments of the source code that implement these concepts, thereby solving an instance of the *concept assignment problem* [2], namely, to identify how high-level concepts are associated with their implementations in source code. We are currently building the system, and we plan to evaluate it and to report empirical results.

2 Our Approach

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

2.1 The Problem

When depositing applications into software repositories, programmers create meaningful descriptions of these applications. However, search engines use exact matches between the keywords from queries and the words in the descriptions of the applications, making it difficult for users to guess exact keywords to find relevant applications. This is

29th International Conference on Software Engineering Workshops(ICSEW'07)
0-7695-2830-9/07 $20.00 © 2007 IEEE

known as the vocabulary problem which states that no single word can be chosen to describe a programming concept in the best way [6].

A fundamental problem of finding relevant applications is in the mismatch between the high-level intent reflected in the descriptions of these applications and low-level implementation details. Many application repositories are polluted with poorly functioning projects [9], and matches between keywords from the queries with words in the descriptions of the applications do not guarantee that these applications are relevant.

Currently, the only way for programmers to determine if an application is relevant is to download it, locate and examine fragments of the code that implement the desired features and observe the runtime behavior of this application to ensure that this behavior matches requirements. This process is manual, with programmers studying the source code of the retrieved applications, locating various *Application Programming Interface (API)* calls, and reading information about these calls in help documents. Still, it is difficult for programmers to link high-level concepts from requirements to their implementations in source code [2].

Modern search engines do little to ensure that retrieved applications can serve as prototypes. To assist rapid prototyping, a code mining system should take high-level requirements and return executable applications whose functionality is described by these requirements. Short code snippets that are returned as a result of the query do not give enough background or context to help programmers to create rapid prototypes, as programmers typically invest a significant intellectual effort to understand how to use these code snippets in larger scopes [10].

2.2 Key Ideas

Our goal is to automate parts of the human-driven procedure of searching for relevant applications. Suppose that requirements specify that a program should encrypt and compress data. When retrieving sample applications from Sourceforge using the keywords `encrypt` and `compress`, programmers look into 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 can be used as prototypes, it is a good starting point for deciding whether to check these applications further.

Our idea is to use help documents that describe API calls to match keywords from queries to the words from the descriptions of these API calls. Help documentation is usually supplied by the same vendors whose packages are used in applications. When programmers read these help documents about API calls, they trust these documents since they come from known and respected vendors, were written by multiple people and reviewed several times, and used by other programmers who report their experience at different forums. Because of these and some other factors, help documents are more verbose and accurate, and are consequently trusted more than the descriptions of applications from software repositories.

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. Our idea of improving the precision of Exemplar is to determine relations (i.e., dataflow links) between API calls in retrieved applications. 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. We claim that it is possible to achieve a higher precision in finding relevant applications by using this heuristic to rank applications, and we plan to substantiate this claim with the results of our future experiments with Exemplar.

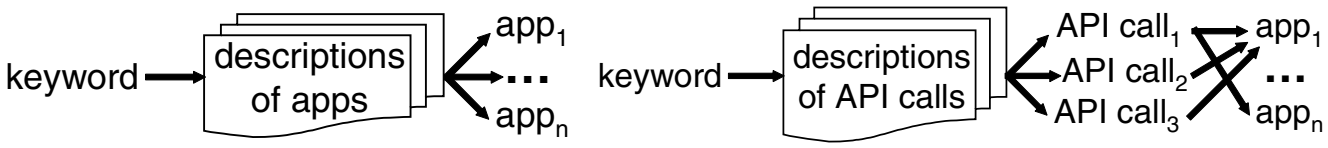
2.3 Our Approach

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 shown in Figure 1(a). A keyword from the query is matched against the descriptions of the applications in some repository. When a match is found, applications $\texttt{app}_1$ to $\texttt{app}_n$ are returned. If the keyword does not match any words in the description of, say $\texttt{app}_1$, then this application will not be returned.

Consider the process for Exemplar shown in Figure 1(b). A keyword from the query is matched against the descriptions of different help documents for software packages whose API calls are used in the applications. When a match is found, the names of the API calls $\texttt{API call}_1$ to $\texttt{API call}_3$ 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 $\texttt{app}_3$ are returned.

A fundamental difference between these search schemes is that Exemplar uses help documents to produce the names of the API calls in return to user queries. Doing so can be viewed as an instance of the *query expansion* concept in information retrieval systems [1]. The aim of query expansion is to reduce this query/document mismatch by expanding the query with keywords that have a similar meaning to the set of relevant documents. Using help documents in Exemplar, the initial query is expanded to include the names of the API calls whose semantics unequivocally reflects spe-



(a) Standard search engines.

(b) Exemplar search engine.

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

cific behavior of the matched applications.

Let us compare Exemplar with standard search engines. Suppose that the user enters a query whose keywords do not match any word in the description of the application $\texttt{app}_1$. While this application may be highly relevant, it is not returned by standard search engines, since users and programmers may not use the same words to describe the same concept, i.e., it is an instance of the vocabulary problem [6].

In the case of Exemplar, it matches the entered keyword 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 several different people who have and use different vocabularies. The richness of these vocabularies makes it more likely to find matches, and produce different API calls. If some help document does not contain a desired match, some other document may yield a match.

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 $\texttt{app}_1$, and consequently the application $\texttt{app}_1$ is returned with a higher rank than other applications.

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

3 Exemplar Architecture and Process

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 Java API and Microsoft documentation as well as help documents for other third-party software.

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)`.

When the user enters a query `(4)`, it is passed to the API call lookup 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 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).

Before proceeding to rank the retrieved projects using the dataflow links heuristic, the Analyzer computes the Applications Metadata `(10)` that contains dataflow links between different API calls from the applications source code (9). This metadata is supplied to the Ranking Engine `(12)` along with the Retrieved Applications `(11)`, and the engine outputs Relevant Applications `(13)`, which are returned to the user `(14)`.

4 Related Work

Different code mining techniques and tools have been proposed to retrieve relevant software components from repositories. CodeFinder is a tool for refining code repos-

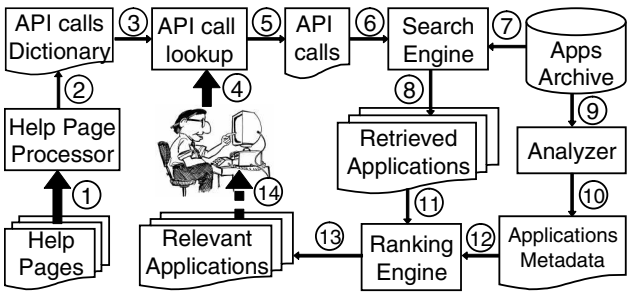


Figure 2. Exemplar architecture.

itory iteratively in order to improve the precision and relevance of returned software components [7]. Like Exemplar, CodeFinder reformulates queries in order to expand the search scope. A main difference between CodeFinder and Exemplar is that CodeFinder is heavily dependent on the descriptions (often erroneous) of software components.

The Codebroker system uses source code and comments written by programmers to query code repositories to find relevant artifacts [15]. Unlike Exemplar, Codebroker is dependent upon the descriptions of documents and meaningful names of program variables and types, and this dependency often leads to a lower precision of returned projects.

Even though it returns code snippets rather than applications, Mica is the most relevant work to Exemplar [14]. Mica uses an external search engine, such as Google in order to search the Internet for relevant examples. 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 ranges of the queries.

In the past several years, several approaches and tools (e.g., Prospector, Hippikat, XSnippet, Strathcona) have been developed to retrieve snippets of code based on the context of the source code that programmers work on [11][5][8][12][13]. While these approaches and tools improve programmer's productivity, they do not target returning relevant applications for high-level queries.

5 Conclusion

We offer a novel approach called Exemplar for finding highly relevant software projects from a large archive of executable examples. After a programmer enters a query that contains high-level concepts (e.g., toolbar, download, smart card), Exemplar combines these concepts with information retrieval and program analysis to retrieve projects that implement high-level concepts entered as part of this initial query. We plan to evaluate Exemplar on open-source software projects and obtain results that will show its effectiveness and compare it with existing approaches.

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 assignment problem. *Commun. ACM*, 37(5):72–82, 1994.

[3] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[4] B. W. Boehm, T. E. Gray, and T. Seewaldt. Prototyping vs. specifying: A multi-project experiment. In *ICSE '84*, pages 473–484, Piscataway, NJ, USA, 1984. IEEE Press.

[5] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Trans. Software Eng.*, 31(6):446–465, 2005.

[6] 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] S. Henninger. Supporting the construction and evolution of component repositories. In *ICSE*, pages 279–288, 1996.

[8] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE*, pages 117–125, 2005.

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

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

[11] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *PLDI*, pages 48–61, 2005.

[12] M. P. Robillard. Automatic generation of suggestions for program investigation. In *ESEC/SIGSOFT FSE*, pages 11–20, 2005.

[13] N. Sahavechaphan and K. T. Claypool. XSnippet: mining for sample code. In *OOPSLA*, pages 413–430, 2006.

[14] 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.

[15] Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. In *ICSE*, pages 513–523, 2002.