

Portfolio: A Search Engine for Finding Functions and Their Usages

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

Mark Grechanik
Accenture and UIC
Chicago, IL 60601
drmark@uic.edu

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

Qing Xie, Chen Fu
Accenture Technology Lab
Chicago, IL 60601
qing.xie@accenture.com

ABSTRACT

In this demonstration, we present a code search system called *Portfolio* that retrieves and visualizes relevant functions and their usages. We will show how chains of relevant functions and their usages can be visualized to users in response to their queries.

1. INTRODUCTION

Source code search engines locate and display sections of code relevant to the concepts given in a user query. Different studies show that when searching source code, programmers are more interested in finding definitions of functions and chains of function invocations than variables, statements, or arbitrary fragments of source code [20, 21]. Moreover, the process of understanding the code returned by search engines and determining how to use it is a manual and laborious process that takes anywhere from 50% to 80% of programmers' time [3, 6]. Short code fragments returned by existing source code search engines do not give enough background or context to help programmers determine how to reuse these code fragments, and programmers typically invest a significant intellectual effort (i.e., they need to overcome a high cognitive distance [11]) to understand how to reuse these code fragments. On the other hand, if code fragments are retrieved as functions, developers can more easily understand how to reuse these functions.

A majority of code search engines treat code as plain text where all words have unknown semantics. The words in source code do have meaning, however. For example, applications contain functional abstractions that already provide a basic level of code reuse, since programmers define functions once and call them from different places in source code. The idea of using functional abstractions to improve code search was proposed and implemented elsewhere [1, 7, 15, 22]; however, these code search engines do not automatically analyze how functions are used in the context of other functions, despite the fact that understanding the chains of function invocations is a key component of programmer understanding of source code. Unfortunately, existing code search engines do little to ensure that they retrieve code fragments in a broader context of relevant functions that invoke one another to accomplish certain tasks.

Consider the situation where a programmer wants to accomplish the complete task of editing and saving a PDF file. He or she may enter the query "edit save pdf file" into a search engine. Existing source code search engines would return some functions

that create PDF files, some functions that write data in the PDF file format, and some functions that manipulate PDF files. Typically, programmers investigate these functions to determine which of them are relevant and to determine how to compose the concepts in these functions into complete tasks. Unfortunately, the results from existing engines do not show to programmers how isolated functions interact in the context of other functions, despite the fact that the programmer wants to see the code for the whole task of how to edit PDF data and save it. A search engine can support programmers efficiently if it incorporates in its user ranking how these functions call one another, and displays that information to the user.

We demonstrate a code search system called *Portfolio* [14] that supports programmers in finding relevant functions that implement high-level requirements reflected in query terms (i.e., finding initial focus points), determining how these functions are used in a way that is highly relevant to the query (i.e., building on found focus points), and visualizing dependencies of the retrieved functions to show their usages. Portfolio finds highly relevant functions in close to 270 Millions LOC in projects from FreeBSD Ports¹ by combining various *natural language processing (NLP)* and indexing techniques with *PageRank* and *spreading activation network (SAN)* algorithms. With NLP and indexing techniques, initial focus points are found that match key words from queries; with PageRank, we model the surfing behavior of programmers, and with SAN we elevate highly relevant chains of function calls to the top of search results. Portfolio is free and available for public use². A video accompanying this demonstration is online³. We evaluated Portfolio with an experiment involving 49 professional programmers from Accenture. We tested Portfolio against two other source code search engines: Google Code Search and Koders. The participants used each engine to perform specified tasks. We found statistically-significant improvement by Portfolio over Google Code Search and Koders⁴.

2. PORTFOLIO APPROACH

The search model of Portfolio uses a key abstraction in which the search space is represented as a directed graph with nodes as functions and directed edges between nodes that specify usages of these functions (i.e., a *call graph*). For example, if the function g is invoked in the function f , then a directed edge exists from the node

¹<http://www.freebsd.org/ports>

²<http://www.searchportfolio.net>

³http://www.youtube.com/watch?v=FNZYXZNo_g0

⁴For full details about Portfolio and our evaluation, we direct readers to our technical paper. This work is supported by NSF CCF-0916139, CCF-0916260, and Accenture.

Function Name	Project
PDFExportDialog	scribus
convert	scribus
HaveNewDoc	scribus
EPSPPlug	scribus
doSaveAsPDF	scribus
startUpDialog	scribus
initMenuBar	scribus
initScribus	scribus
enableSave	scribus
updateDocOptions	scribus
cmsDescriptor	scribus
getPagesString	scribus
DoExport	scribus
colorSpaceComponents	scribus
fileNameChanged	scribus
ChangeFile	scribus

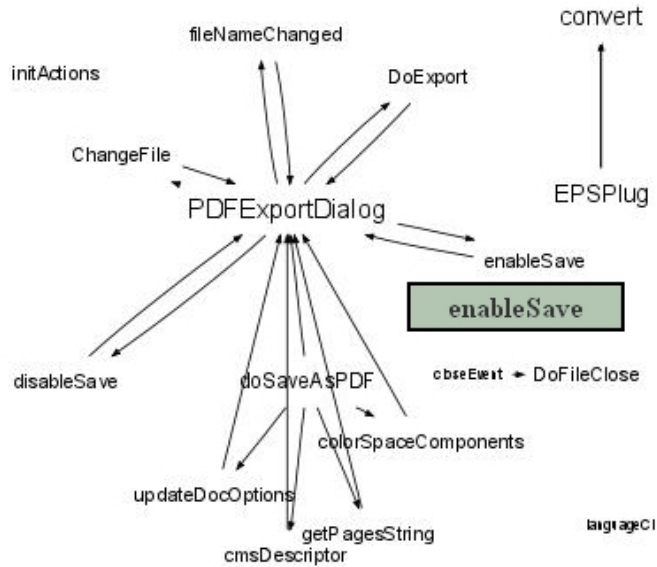


Figure 1: The visual interface of Portfolio. The left side contains a list of ranked retrieved functions and the right side contains a static call graph that contains these and other functions; edges of this graph indicate the directions of function invocations. Hovering a cursor over a function on the list shows a label over the corresponding function on the call graph. Font sizes reflect the score; the higher the score of the function, the bigger the font size used to show it on the graph. Clicking on the label of a function loads its source code in a separate browser window.

that represents the function f to the node that represents the function g . Since the main goal of Portfolio is to enable programmers to find relevant functions and their usages, we need models that effectively represent the behavior of programmers when navigating a large graph of functional dependencies. These are navigation and association models that address surfing behavior of programmers and associations of terms in functions in the search graph.

When using text search engines, users navigate among pages by following links contained in those pages. Similarly, in Portfolio, programmers can navigate between functions by following edges in the directed graph of functional dependencies using Portfolio's visual interface. To model the navigation behavior of programmers, we adopt the model of the *random surfer* that is the basis for the popular ranking algorithm PageRank [12]. Specifically, we compute the PageRank of every function in the call graph, and rank functions higher if those functions receive high PageRank values.

Portfolio also establishes relevance among functions whose content does not contain terms that match user queries directly. Consider the query "edit save pdf file." This situation is shown in Figure 2, where the function F contains the term `edit`, the function G contains the term `postscript`, the function P contains the terms `PDF` and `file`, and the function Q contains the term `save`. Function F calls the function G , which in turn calls the function H , which is also called from the function Q , which is in turn called

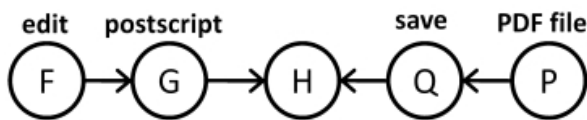


Figure 2: Example of associations between different functions.

from the function P . The functions F , P , and Q will be returned by a search engine that is based on matching query terms to those that are contained in documents. Meanwhile, the functions H and G may be highly relevant to the query but are not retrieved since they have no words that match the search terms. In addition, the function Q can be called from many other functions since its saving functionality is generic; however, its usage is most valuable for programmers in the context of the function related to query terms.

To ensure that functions such as H and G end up on the list of relevant functions, Portfolio uses an association model based on a *Spreading Activation Network* [2, 4]. In SANs, nodes represent documents, while edges specify properties that connect these documents. The edges' direction and weight reflect the meaning and strength of associations among documents. Given an entry point node, spreading activation traverses a graph to locate a chain of similar nodes. For example, if we use the function Q as the entry point, spreading activation will find the functions H and P because the two are connected with an edge in the call graph. In our case, the associations among functions are the calls that those functions make to one another. Portfolio locates the entry point nodes in the graph using the textual similarity of user queries to functions in source code as in many current source code search engines. Once applied to SAN, Portfolio uses spreading activation to compute new weights for nodes (e.g., ranks of functions) that reflect implicit associations in the network of these nodes (e.g., the call graph).

3. DEMONSTRATING PORTFOLIO

In this section, we demonstrate Portfolio in two ways. In the first, the user submits a search query to the Portfolio search engine web interface. Portfolio presents functions relevant to the query in a browser window as it is shown in Figure 1. The left side contains the ranked list of retrieved functions and project names, while the right side contains a static call graph that contains these and other

functions. Edges of this graph indicate the directions of function invocations. Hovering a cursor over a function on the list shows a label over the corresponding function on the call graph. Font sizes reflect the combined ranking; the higher the ranking of the function, the bigger the font size used to show it on the graph. Clicking on the label of a function loads its source code in a separate browser window. Also next to the function name is the name of the project to which that function belongs. Hovering the mouse over the project name will display a short description of that project.

The static call graph in Figure 1 helps programmers by visualizing the chain of function invocations among relevant functions. Displaying this chain makes the connections among functions immediately obvious to the user. For example, the function `PDFExportDialog` is shown in large font with many incoming edges, indicating its importance. Programmers may be guided to functions such as `doSaveAsPDF` since it calls a variety of functions including `PDFExportDialog`. Portfolio also shows that certain functions, such as `EPSPPlug` and `convert`, do not connect to `PDFExportDialog`, which suggests that those functions are relevant to different tasks. This knowledge about how functions interact can help during reuse because it provides an at-a-glance view of how functions are organized, and which functions may perform the lowest-level functionality. Also, functions which call several of these low-level functions are obvious without having to first read the source code.

The second way in which developers can access Portfolio is programmatically by using our SOAP web service⁵. The purpose of this service is to enable programmers to build source code search directly into their software or development environments. At the time of writing, we have three SOAP functions available. The first, `search`, returns a user-specified number of results to a query. The second, `code`, can return the source code of any function in the repository. Finally, `edges`, returns functions which call or are called by a given function. In addition, all of our supporting tools are online, including *Findex*, which we built to extract the call graph from the 270 Million lines of code in FreeBSD Ports. Interested parties may also download our repository, the full, extracted call graph, and other information.

4. RELATED WORK

Different code mining techniques and tools have been proposed to find relevant software components. Some search based primarily on textual artifacts extracted from source code or produced by programmers [23, 8], in contrast to other approaches which use documentation external to retrieved functions (such as documentation for API calls) [7, 22, 1]. Portfolio, on the other hand, uses PageRank and SANs to help programmers navigate and understand usages of retrieved functions.

Web-mining techniques have been applied to graphs derived from program artifacts before [15, 18, 16]. Notably, Inoue et al. proposed Component Rank[10] as a method to highlight the most-frequently used classes by applying a variant of PageRank to a graph composed of Java classes. Portfolio differs from these approaches in that it retrieves relevant functions using an association model based on SANs in addition to PageRank.

There are task-oriented tools to assist programmers in writing complicated code through reuse [13, 5, 9, 19, 17], however, their utilities require additional environment information such as existing project source code or data types of test cases. Portfolio retrieves functions when given only a natural-language query.

⁵<http://www.searchportfolio.net/>, follow the “Programmer Access” link

5. CONCLUSION

We created an approach called Portfolio for finding highly relevant functions and projects from a large archive of C/C++ source code. In Portfolio, we combined various *natural language processing* and indexing techniques with a variation of *PageRank* and *spreading activation network* algorithms to address the need of programmers to reuse retrieved code as functional abstractions. Portfolio differs from previous approaches in that it both retrieves and visualizes functions and chains of function invocations using navigation and association models. Moreover, we have made Portfolio available to programmers as a free and extensible project.

6. REFERENCES

- [1] S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine for java using free-form queries. In *FASE*, pages 385–400, 2009.
- [2] A. M. Collins and E. F. Loftus. A spreading-activation theory of semantic processing. *Psychological Review*, 82(6):407–428, 1975.
- [3] T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.
- [4] F. Crestani. Application of spreading activation techniques in information retrieval. *Artificial Intelligence Review*, 11(6):453–482, 1997.
- [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] J. W. Davison, D. Mancl, and W. F. Opdyke. Understanding and addressing the essential costs of evolving systems. *Bell Labs Technical Journal*, 5(2):44–54, 2000.
- [7] 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.
- [8] S. Henninger. Supporting the construction and evolution of component repositories. In *ICSE*, pages 279–288, 1996.
- [9] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE*, pages 117–125, 2005.
- [10] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto. Ranking significance of software components based on use relations. *IEEE Trans. Softw. Eng.*, 31(3):213–225, 2005.
- [11] C. W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.
- [12] A. N. Langville and C. D. Meyer. *Google’s PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, Princeton, NJ, USA, 2006.
- [13] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *PLDI*, pages 48–61, 2005.
- [14] C. McMillan, M. Grechanik, D. Poshyvanyk, X. Qing, and C. Fu. Portfolio: Finding relevant functions and their usages. In *ICSE ’11*.
- [15] J. Ossher, S. Bajracharya, E. Linstead, P. Baldi, and C. Lopes. Sourcererd: An aggregated repository of statically analyzed and cross-linked open source java projects. *MSR*, 0:183–186, 2009.
- [16] D. Puppini and F. Silvestri. The social network of java classes. In *SAC ’06*, pages 1409–1413, New York, NY, USA, 2006. ACM.
- [17] S. P. Reiss. Semantics-based code search. In *ICSE*, pages 243–253, 2009.
- [18] M. Revelle, B. Dit, and D. Poshyvanyk. Using data fusion and web mining to support feature location in software. In *ICPC ’10*, pages 14–23, 2010.
- [19] N. Sahavechaphan and K. T. Claypool. XSnippet: mining for sample code. In *OOPSLA*, pages 413–430, 2006.
- [20] J. Sillito, G. C. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *IEEE Trans. Softw. Eng.*, 34(4):434–451, 2008.
- [21] S. Sim, C. Clarke, and R. Holt. Archetypal source code searches: A survey of software developers and maintainers. *ICPC*, 0:180, 1998.
- [22] 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.
- [23] Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. In *ICSE*, pages 513–523, 2002.