Who Can Help Me with this Change Request?

Huzefa Kagdi[1], Denys Poshyvanyk[2]

[1]*Department of Computer Science*
Missouri University of Science and Technology
Rolla, MO 65409
kagdih@mst.edu

[2]*Computer Science Department*
The College of William and Mary
Williamsburg, VA 23185
denys@cs.wm.edu

Abstract

An approach to recommend a ranked list of developers to assist in performing software changes given a textual change request is presented. The approach employs a two-fold strategy. First, a technique based on information retrieval is put at work to locate the relevant units of source code, e.g., files, classes, and methods, to a given change request. These units of source code are then fed to a technique that recommends developers based on their source code change expertise, experience, and contributions, as derived from the analysis of the previous commits. The commits are obtained from a software system's version control repositories (e.g., Subversion). The approach is demonstrated on a bug report from KOffice, an open source application suite.

1. Introduction

It is a common, but by no means trivial, task in software maintenance for managers and technical leads alike to delegate the responsibility of implementing change requests (e.g., bug fixes and new feature request) to the developer(s) with the right expertise. Change requests are typically specified via free-form textual description in natural language (e.g., a new feature request/wish or a bug reported to the *Bugzilla*[1] repository of an open source project).

This triaging like task typically involves a project and/or organization wide knowledge and balancing many factors; all of which if handled manually can be quite tedious [2]. For example, one solution is to email the project team or developers and seek for suggestions or advice on who is the most knowledgeable about a certain part of source code or bug or feature. Therefore, managers in such a situation would be well served by an approach that automatically recommends the appropriate developers by using only the minimum, largely non-technical input, i.e., textual change requests.

Here, we present a novel approach that combines two existing techniques to recommend developers that are best suited to help with an incoming change request. The combined techniques are an Information Retrieval (IR) based technique that uses Latent Semantic Indexing (LSI) [9] for *concept location* [16] and an approach that is based on Mining Software Repositories (MSR) [14] to recommend a ranked list of candidate developers for source code change [15].

We use the umbrella term *concept* to generally refer to the textual description of the change request irrespective of its specific intent (e.g., description of a new feature that needs to be added or a bug that needs to be fixed). In a nutshell, our approach is a two-step procedure:

1. Given a concept description, we use LSI technique to locate a ranked list of relevant units of source code (e.g., files, classes, and methods) that implement that concept in a version (typically the version in which an issue is reported) of the software system.
2. The version histories of units of source code from the above step are then analyzed to recommend a ranked list of developers that are the most experienced and/or have substantial contributions in dealing with those units (e.g., classes).

This combined approach is different from previous approaches, including those using IR, for expert developer recommendations that rely solely on the historical account of past change requests and/or source code changes [2, 5, 6, 15]. Our approach does not need mining of past change requests (e.g., history of similar bug reports to resolve the bug request in question), and requires source code change history of only selective entities.

2. Locating Concepts with Information Retrieval

Using advanced IR techniques, such as those based on LSI [16, 21], allow users to capture relations between terms (words) and documents in large bodies of text. In software engineering, LSI has been used for a variety of tasks including concept location. Marcus et al. [16] introduced previously a methodology to index and search the source code using IR methods. Subsequently, Poshyvanyk et al. [21] refined the methodology and

[1] Bugzilla is a bug-tacking system available at http://www.bugzilla.org/ (verified on 12/05/08)

combined it with dynamic information to improve its effectiveness.

In our approach, the comments and identifiers from the source code are extracted and a corpus is created. In this corpus, each document corresponds to a user chosen unit of source code (e.g., class) in the system. LSI indexes this corpus and creates a signature for each document. These indices are used to define similarity measures between documents. Users can originate queries in natural language (as opposed to regular expressions or some other structured format) and the system returns a list of all the documents in the system, ranked by their semantic similarity to the query. This use is similar to many existing web search engines.

The first two steps are usually performed once, while the other ones are performed repeatedly until the user finds the desired parts of the source code.

1. **Creating a corpus of a software system.** The source code is parsed using a developer-defined granularity level (i.e., methods or classes) and documents are extracted from the source code. A corpus is created, so that each method (and/or class) will have a corresponding document in the resulting corpus. Only identifiers and comments are extracted from the source code. In addition, we also created corpus builder for large C++ projects, using srcML [8] and Columbus [11].
2. **Indexing.** The corpus is indexed using LSI and a representation of the corpus as a real-valued vector subspace is created. Dimensionality reduction is performed in this step, capturing important semantic information about identifiers and comments in the source code, and their relationships. In the resulting subspace, each document (method or class) has a corresponding vector.
3. **Formulating a query.** A developer selects a set of terms that describe the concept of interest (for example, 'open file'). This set of words constitutes the initial query. The tool spell-checks all the terms from the query using the vocabulary of the source code (generated by LSI). If any word from the query is not present in the vocabulary, then the tool suggests similar words based on editing distance and removes the term from the search query.
4. **Ranking documents.** Similarities between the user query and documents from the source code (for example, methods or classes) are computed. The similarity between a query reflecting a concept and a set of data about the source code indexed via LSI allows generating a ranking of documents relevant to the concept. All the documents are ranked by the similarity measure in descending order (i.e., most relevant at the top, whereas, least relevant at the bottom).

LSI offers many unique benefits compared to other natural language processing techniques. Among which include the robustness of LSI with respect to outlandish identifier names and stop words (which are eliminated) and no need of a predefined vocabulary or morphological rules. The finer details of the inner workings of LSI used in this work is similar to its previous uses; we refer the interested readers to [16, 21].

Here, we demonstrate the working of the approach using an example from *KOffice*[2]. The change request or concept of interest is the bug# *173881*[3] that was reported to the bug tracking system (in this case maintained by *Bugzilla*) on 2*008-10-30*. The reporter described the bug as follows:

"splitting views duplicates the tool options docker".

Table 1. Top five classes extracted and ranked by the concept location tool that are relevant to the description of bug# 173881 reported in KOffice.

Rank	Class Names	Similarity
1	KoDockerManager	0.66
2	ViewCategoryDelegate	0.54
3	ViewListDocker	0.51
4	KisRulerAssistantToolFactory	0.49
5	KWStatisticsDocker	0.46

We consider the above textual description as a concept of interest. We collected the source code of *KOffice 2.0-Beta 2* from the development *trunk* on *2008-10-31* (the bug was not fixed as of this date). We parsed the source code of *KOffice* using class level granularity (i.e., each document is a class). After indexing with LSI we obtained a corpus consisting of 4,756 documents and containing 19,990 unique terms. We formulated the search query "*split docker view tool option*", which was used as an input to LSI-based concept location tool. The partial results of the search (i.e., a ranked list of relevant classes) are summarized in Table 1.

3. Recommending Developers using Version History

We use the approach of *xFinder* to recommend expert developers by mining version archives of a system [15]. The basic premise of this approach is that the developers who contributed substantial changes to a specific part of source code in the past are likely to best assist in its current or future change. More specifically, past contributions are analyzed to derive a mapping of the developers' expertise, knowledge, or ownership to particular entities of the source code - a *developer-code map*. Once a developer-code map is obtained, a list of

[2] KOffice is an integrated office suite for K Development Environment and is available at http://www.koffice.org/ (verified on 12/05/08)

[3] http://bugs.kde.org/show_bug.cgi?id=173881 (verified on 12/05/08)

developers who can assist in a given part of the source code can be obtained in a straightforward manner.

Our approach uses the commits in repositories that record source code changes submitted by developers to the version-control systems (i.e., *Subversion*). *Subversion* preserves the grouping of several changes in multiple files to a single change-set as performed by the committer. *Subversion*'s commit log entries include the dimensions *author*, *date*, and *paths* (e.g., files) involved in a change-set. Additionally, a text message describing the change entered by the developer is also recorded. Figure 1 shows a log entry from the *Subversion* repository of *KOffice*. A log entry corresponds to a single *commit* operation. In this case, the changes in the file *koffice/kword/part/frames/KWAnchorStrategy.cpp* are committed by the developer *zander* on the date/time *2008-11-14T17:22:26.488329Z*. The *revision* number *884334* is assigned to the entire change-set (and not to each file that is changed as is in the case with some version-control systems such as CVS).

```
<?xml version="1.0" encoding="utf-8"?>
<log>
  <log entry revision="884334">
    <author>zander</author>
    <date>2008-11-14T17:22:26.488329Z</date>
    <paths>
      <path  action="M">  koffice/libs/guiutils/
KWAnchorStrategy.cpp
      </path>
    </paths>
    <msg>
       Don't assert but try to put the anchored
shape in a parent shape.
    </msg>
  </log entry>
</log>
```

Figure 1. Part of *KOffice* subversion log message.

We presented a few ways of gauging developer contributions from commits in [15]. We used the measures to determine developers that were likely to be experts in a specific source code file, i.e., developer–code map. The developer-code map is represented via the developer-code vector *DV* for the developer *d* and file *f*, as shown below,

$DV_{(d,f)} = <C_f, A_f, R_f>$, where:

- C_f is the number of commits, i.e., commit contributions that include the file *f* and are committed by the developer *d*.
- A_f is the number of workdays in the activity of the developer *d* with commits that include the file *f*.
- R_f is the most recent workday in the activity of the developer *d* with a commit that includes the file *f*.

Similarly, the change contributions to the file *F* can be represented via the file-change vector *FV* (given below).

$FV_{(f)} = <C'_f, A'_f, R'_f>$, where

- C'_f is the total number of commits, i.e., commit contributions, that include the file *f*.
- A'_f is the total number of workdays in the activity of all developers with commits that include the file $_f$.
- R'_f is the most recent workday with a commit that includes the file *f*.

The contribution or expertise factor, termed *Xfactor*, for the developer *d* and the file *f* is computed using a similarity measure of the developer-code vector and the file-change vector. For example, we use the *Euclidean* distance to find the distance between the two vectors. Distance is an opposite of similarity, thus lesser the value of the *Euclidean* distance, greater the similarity between the vectors. *Xfactor* is the inverse of distance.

We use *Xfactor* as a basis of the recommendation method used to suggest a ranked list of developers to assist with a change in a given file. The developers are ranked based on their *Xfactor* values. The developer with the highest value is ranked first. Now, there maybe some files that have not been changed in a very long time, or this is the first change where a file is added. As a result, there will not be any recommendation. To overcome this problem, we look for developers who are experts in a package that contains the file, and recommend them instead. As a final option, if no package expert can be identified, we turn to the idea of the system experts). By doing so, we strive for guaranteed recommendation from our tool.

Now we demonstrate the working of the second step of our approach, i.e., *xFinder*, using the *KOffice* bug example from Section 2. The classes from Table 1, given by the concept location, are fed to the *xFinder* tool. The files in which these classes are implemented are first identified. In our example, it turned out that each class was located in a different file. *xFinder* is used to recommend developers for one file at a time. Here, we show the working on one file. The file *guiutils/KoDockerManager.cpp* contains the top ranked class *KoDockerManager*. *xFinder* started with the *KOffice 2.0-Beta 2* from the development *trunk* version on *2008-10-31* and worked its way backward in the version history to look for recommendations for the file *guiutils/KoDockerManager.cpp*. We setup *xFinder* to recommend a maximum of three developers at every level. The ranked list of developers (actually their user IDs) that *xFinder* recommended at file and package levels are given below.

File Experts:	Package Experts:
1. *jaham*	3. *mpfeiffer*
2. *boemann*	4. *jaham*
	5. *zander*

4. Evaluation Method

One of the main focuses of our current work is assessing the effectiveness of our approach, i.e., to investigate how accurate the developer recommendations are. The bug/issue tracking and source code change information in software repositories is used for this purpose. Our evaluation method consists of the following steps:

1. Select an issue (e.g., bug or feature/wish) from the bug tracking system that is resolved as fixed (or implemented).
2. Select a development version on the day at which the selected issue was reported (but not resolved) and apply concept location to get a ranked list of relevant source code classes given its description.
3. Use *xFinder* to collect a ranked list of developers for the classes from 2.
4. Use the changed source code files (or classes) in the patch (or commit) that fixed the issue and the developer who contributed to it as the *ground truth*.
5. Compare the results of steps 2 and 3 with the ground truth in step 4.
6. Repeat the above steps for *N* issues (i.e., change requests).

We illustrate the application of our evaluation method on the *KOffice* bug example from the previous two sections. The bug# 173881 was fixed on 2008-11-02 (two days after it was reported) by a developer with the user id *jaham* and the patch included the file *KoDockerManager.cpp*. As can be clearly seen, both these file and developer were ranked first in the respective steps of our approach. We have observed similar type of accuracy on a number of other reported bugs, which we find very encouraging and promising. Therefore, we plan to systematically evaluate the accuracy of our approach in terms of widely used metrics such as precision and recall. Our evaluation plan includes validating the approach across these metrics on a number of open source projects such as *Apache httpd*, *KOffice*, and *GNU gcc* across a wide variety of issues.

5. Related Work

Our work falls under two broad areas of concept location and recommendation systems. Here, we briefly discuss the related work in both these areas.

5.1. Concept Location

Concept location is a widely studied problem in software engineering and includes many flavors such as *feature* identification and *concern* location. Wilde et al. [26] were the first to address the problem of feature location using the Software Reconnaissance method, which utilizes dynamic information. This approach has been recently improved in [1].

Among other static-based techniques for concept location is the one proposed by Chen et al. [7], which is based on the search of abstract system dependence graph. This technique was subsequently extended by Robillard [22]. Zhao et al. [27] proposed a technique that combines information retrieval with branch-reserving call-graph information to automatically assign features to respective elements in the source code. Eisenbarth et al. [10] combined both static (i.e., dependencies) and dynamic (i.e., execution traces) information to identify features in programs. Hill et al. [13] combined static and textual information to expedite traversal of program dependence graphs for impact analysis. Poshyvanyk et al. [20] combined an IR based technique with scenario-based probabilistic ranking of the execution traces to improve the precision of feature location.

5.2. Developer Recommendation

McDonald and Ackerman [17] developed a heuristic based recommendation system called the Expertise Recommender (ER) to identify experts at the module level. Mino and Murphy [18] produced a tool called Emergent Expertise Locator (EEL). EEL helps in finding the developers who can assist in solving a particular problem. Expertise Browser (ExB) [19] is another tool to locate people with a desired expertise. Anvik and Murphy [3] did an empirical evaluation of two approaches to locate expertise. They found that both approaches have relative strengths in different ways. A machine learning technique has been used in [2] to automatically assign a bug report to the right developer who can resolve it. Another text-based approach is used in [23] to build a graph model called ExpertiseNet for expertise modeling.

Tsunoda et al. [24] analyzed the developers' working time of open source software. Bird et al. [4] mined email archives to analyze the communication and co-ordination activities of the participants. Weissgerber et al. [25] analyze and visualize the check-in information for open source projects. The visualization shows the relationship between the lifetime of the project and the number of files and the number of files updated by each author. German [12] studied the modification records (MRs) of CVS logs to visualize who are the people who tend to modify certain files. These works along with some other related approaches are further elaborated in [15].

6. Conclusions

The main contribution of our work is the synergistic use of a concept location technique with a technique based on MSR for the expert developer recommendation task. While both these techniques have been investigated and used independently with a reasonable level of success in the past, their combined use for tasks such as recommending expert developers for incoming change requests has not been systematically and fully investigated. We envision that our approach would provide a valuable aid to team leads or developers in maintenance tasks such as impact analysis right on the onset of a change request, which is typically specified in free-form text. Our primary, litmus test results suggest that the proposed technique can identify relevant developers with fairly high precision, which we plan to further validate with systematic studies.

7. Acknowledgements

This research was supported in part by the United States Air Force Office of Scientific Research under grant number FA9550-07-1-0030.

8. References

[1] Antoniol, G. and Guéhéneuc, Y. G., "Feature Identification: An Epidemiological Metaphor", *IEEE Transactions on Software Engineering*, vol. 32, no. 9, 2006, pp. 627-641.

[2] Anvik, J., Hiew, L., and Murphy, G. C., "Who Should Fix This Bug?" in Proc. of 28th international conference on Software engineering, Shanghai, China, 2006, pp. 361 - 370.

[3] Anvik, J. and Murphy, G., "Determining Implementation Expertise from Bug Reports", in Proc. of 4th International Workshop on Mining Software Repositories, 2007.

[4] Bird, C., Gourley, A., Devanbu, P., Gertz, M., and Swaminathan, A., "Mining Email Social Networks", in Proc. of 2006 International Workshop on Mining Software Repositories, Shanghai, China, May 22-23 2006, pp. 137-43.

[5] Canfora, G. and Cerulo, L., "Impact Analysis by Mining Software and Change Request Repositories", in Proc. of 11th IEEE International Symposium on Software Metrics, September 19-22 2005, pp. 20-29.

[6] Canfora, G. and Cerulo, L., "Fine Grained Indexing of Software Repositories to Support Impact Analysis", in Proc. of International Workshop on Mining Software Repositories, 2006, pp. 105 - 111.

[7] Chen, K. and Rajlich, V., "Case Study of Feature Location Using Dependence Graph", in Proc. of 8th IEEE International Workshop on Program Comprehension, Limerick, Ireland, June 2000, pp. 241-249.

[8] Collard, M. L., Kagdi, H. H., and Maletic, J. I., "An XML-Based Lightweight C++ Fact Extractor", in Proc. of 11th IEEE International Workshop on Program Comprehension, Portland, OR, May 10-11 2003, pp. 134-143.

[9] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R., "Indexing by Latent Semantic Analysis", *Journal of the American Society for Information Science*, vol. 41, 1990, pp. 391-407.

[10] Eisenbarth, T., Koschke, R., and Simon, D., "Locating Features in Source Code", *IEEE Transactions on Software Engineering*, vol. 29, no. 3, March 2003, pp. 210 - 224.

[11] Ferenc, R., Beszedes, A., and Gyimóthy, T., "Extracting Facts with Columbus from C++ Code", in Proc. of 8th European Conference on Software Maintenance and Reengineering, March 24-26 2004, pp. 4-8.

[12] German, D. M., "An Empirical Study of Fine-grained Software Modifications", *Empirical Software Engineering*, vol. 11, no. 3, September 2006, pp. 369-393.

[13] Hill, E., Pollock, L., and Vijay-Shanker, K., "Exploring the Neighborhood with Dora to Expedite Software Maintenance", in Proc. of 22nd International Conference on Automated Software Engineering, Nov. 2007, pp. 14-23.

[14] Kagdi, H., Collard, M. L., and Maletic, J. I., "A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution", *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, no. 2, March/April 2007, pp. 77-131.

[15] Kagdi, H., Hammad, M., and Maletic, J. I., "Who Can Help Me with this Source Code Change?" in Proc. of IEEE International Conference on Software Maintenance, Beijing, China, September 28-October 3 2008.

[16] Marcus, A., Sergeyev, A., Rajlich, V., and Maletic, J., "An Information Retrieval Approach to Concept Location in Source Code", in Proc. of 11th IEEE Working Conference on Reverse Engineering, Nov. 9-12 2004, pp. 214-223.

[17] McDonald, D. and Ackerman, M., "Expertise Recommender: A Flexible Recommendation System and Architecture", in Proc. of ACM Conference on Computer Supported Cooperative Work, Dec. 2-6 2000, pp. 231-240.

[18] Minto, S. and Murphy, G., "Recommending Emergent Teams", in Proc. of 4th International Workshop on Mining Software Repositories, May 20-26 2007.

[19] Mockus, A. and Herbsleb, J., "Expertise Browser: a Quantitative Approach to Identifying Expertise", in Proc. of 24th International Conference on Software Engineering, Orlando, FL, May 19-25 2002, pp. 503-512.

[20] Poshyvanyk, D., Guéhéneuc, Y. G., Marcus, A., Antoniol, G., and Rajlich, V., "Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval", *IEEE Transactions on Software Engineering*, vol. 33, no. 6, June 2007, pp. 420-432.

[21] Poshyvanyk, D. and Marcus, D., "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code", in Proc. of 15th IEEE International Conf. on Program Comprehension, June 2007, pp. 37-48.

[22] Robillard, M. P., "Topology Analysis of Software Dependencies", *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 4, August 2008.

[23] Song, X., Tseng, B., Lin, C., and Sun, M., "ExpertiseNet: Relational and Evolutionary Expert Modeling", in Proc. of 10th International Conference on User Modeling, Jul. 24-29 2005.

[24] Tsunoda, M., Monden, A., Kakimoto, T., Kamei, Y., and Matsumoto, K.-i., "Analyzing OSS Developers' Working Time Using Mailing Lists Archives", in Proc. of Intern. Workshop on Mining Software Repositories, 2006, pp.181- 182.

[25] Weissgerber, P., Pohl, M., and Burch, M., "Visual Data Mining in Software Archives to Detect How Developers Work Together", in Proc. of Fourth International Workshop on Mining Software Repositories, May 20-26 2007.

[26] Wilde, N., Gomez, J. A., Gust, T., and Strasburg, D., "Locating User Functionality in Old Code", in Proc. of Intern. Conf. on Software Maintenance, 1992, pp. 200-205.

[27] Zhao, W., Zhang, L., Liu, Y., Sun, J., and Yang, F., "SNIAFL: Towards a Static Non-interactive Approach to Feature Location", *ACM Transactions on Software Engineering and Methodologies*, vol. 15, no. 2, 2006, pp. 195-226.