JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. (2011)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.530

Assigning change requests to software developers

Huzefa Kagdi[1,*,†], Malcom Gethers[2], Denys Poshyvanyk[2] and Maen Hammad[3]

[1]*Department of Computer Science, Winston-Salem State University, Winston-Salem, NC 27110, U.S.A.*
[2]*Computer Science Department, The College of William and Mary, Williamsburg, VA 23185, U.S.A.*
[3]*Department of Software Engineering, Hashemite University Zarqa, Jordan*

SUMMARY

The paper presents an approach to recommend a ranked list of expert developers to assist in the implementation of software change requests (e.g., bug reports and feature requests). An Information Retrieval (IR)-based concept location technique is first used to locate source code entities, e.g., files and classes, relevant to a given textual description of a change request. The previous commits from version control repositories of these entities are then mined for expert developers. The role of the IR method in selectively reducing the mining space is different from previous approaches that textually index past change requests and/or commits. The approach is evaluated on change requests from three open-source systems: *ArgoUML*, *Eclipse*, and *KOffice*, across a range of accuracy criteria. The results show that the overall accuracies of the correctly recommended developers are between 47 and 96% for bug reports, and between 43 and 60% for feature requests. Moreover, comparison results with two other recommendation alternatives show that the presented approach outperforms them with a substantial margin. Project leads or developers can use this approach in maintenance tasks immediately after the receipt of a change request in a free-form text.
Copyright © 2011 John Wiley & Sons, Ltd.

Received 28 October 2009; Revised 3 June 2010; Accepted 10 December 2010

KEY WORDS: concept and feature location; information retrieval; developer recommendation; software evolution and maintenance; mining software repositories

1. INTRODUCTION

It is a common, but by no means trivial task, in software maintenance for technical leads to delegate the responsibility of implementing change requests (e.g., bug fixes and new feature requests) to the developers with the right expertise. This task typically involves project or even organization-wide knowledge, and the balancing of many factors; all of which if handled manually can be quite tedious [1]. The issue is particularly challenging in large-scale software projects with several years of development history and developed collaboratively with hundreds of contributors, often geographically distributed (e.g., open-source development environments). It is not uncommon in such projects to receive numerous change requests daily that need to be resolved in an effective manner (e.g., within time, priority, and quality factors).

Therefore, assigning change requests to the developers with the right implementation expertise is challenging, but certainly a needed activity. For example, one straightforward practice is to e-mail the project team or developers, or discuss via issue-tracking system, and rely on them for suggestions or advice on who has helpful knowledge about a certain part of source code,

*Correspondence to: Huzefa Kagdi, Department of Computer Science, Winston-Salem State University, Winston-Salem, NC 27110, U.S.A.
†E-mail: kagdihh@wssu.edu

Copyright © 2011 John Wiley & Sons, Ltd.

a bug, or a feature. Clearly, this activity is reactive and may not necessarily yield an effective or efficient answer. An active developer of *ArgoUML*, where this activity is manual, stated that they would welcome any tool that would lead to more enjoyable and efficient job experience, and is not perceived as a hindrance. In open-source software development, where much relies on volunteers, it could serve as a catalyst if there was a tool that automatically mapped change requests to appropriate developers. That is, developers do not have to wade through the numerous change requests to seek for what they can contribute to; they are presented a 'filtered' set of change requests that suits their palates instead. Both help seekers and sustained software evolution in such a situation would greatly benefit from a proactive approach that automatically recommends the appropriate developers based solely on information available in textual change requests. Change requests are typically specified in a free-form textual description using natural language (e.g., a bug reported to the *Bugzilla* system of a software project).

The core research philosopy is that the present+past of a software system leads to its better future evolution [2]. Here, the umbrella term *concept* refers to the textual description of the change request irrespective of its specific intent (e.g., a new feature request or a bug report). An information retrieval (IR) technique, specifically Latent Semantic Indexing (LSI) [3, 4], is first used to locate the relevant units of source code (e.g., files and classes) that implement the concept of interest in a single version (e.g., *KOffice 2.0-Beta 2*) in which a bug is reported. The past commits (e.g., from *Subversion* repositories) of only the relevant units of source code are then analyzed to recommend a ranked list of candidate developers [5]. Therefore, our approach not only recommends a list of developers who should resolve a particular change request, but also the potentially relevant source code to it. The same *ArgoUML* developer mentioned above commented that pinpointing the source code associated with a problem (i.e., involved in a bug) is also interesting, which is exactly what our approach provides, as an intermediate step.

The core research philosophy is that the *present+past* of a software system leads to its better *future* evolution [6]. This combined approach integrates several latent *opportunities* in the rich set of actual changes that is left behind in the system's development history (otherwise perceived as a *challenge* and largely ignored in conventional development tools). We can now ask for developer recommendations at the textual change request level instead of at the source code level, as in our previous work [5]. For example, our approach correctly recommended a ranked list of developers, [*jaham*, *boemann*], knowledgeable in source code files related to a bug, from its description '*splitting views duplicates the tool options docker*', in *KOffice*[‡], an open-source office productivity suite.

The key contribution of our work is the first use of a concept location technique combined with a technique based on Mining Software Repositories (MSR) for the expert developer recommendation task. This combined approach is different from other 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 [1, 5, 7–9]. Our approach does not need to mine past change requests (e.g., history of similar bug reports to resolve the bug request in question), but does require source code change history. The single-version analysis with IR is employed to reduce the mining space of the source code change history to only selected entities. The paper includes a comprehensive evaluation of the approach (which was not reported in [2]) on a number of change requests spanning across bug reports, feature requests, and even refactorings, from *KOffice*, and also a number of releases and commit history periods of *ArgoUML* and *Eclipse*. The accuracy of the developer recommendations is rigorously assessed across various criteria. The results indicate that our approach fares very well with accuracies of 80% for bug reports and 42% for feature requests in *KOffice*, for example.

The remainder of the paper is organized as follows. Section 2 describes our approach to developer recommendation. Sections 3 and 4 present a systematic evaluation and analysis of three

[‡]KOffice is an integrated office suite for K Development Environment and is available at http://www.koffice.org/ (verified on 10/21/09).

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

open-source systems. Section 5 is devoted to threats to validity. Section 6 discusses the related literature and conclusions are presented in Section 7.

2. OUR APPROACH TO EXPERT DEVELOPER RECOMMENDATIONS

Our approach to recommending expert developers to assist with a given change request consists of the following two steps:

(1) Given a concept description, we use LSI 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).

2.1. Locating concepts with information retrieval

Concept refers to a human-oriented expression of the intent of implemented software or its parts [10]. Concepts are typically expressed in natural language with terms from application and/or problem domains. Examples include spell checking in a word processor or drawing a shape in a paint program. Concept location is a widely studied problem in software engineering and includes many flavors such as *feature* identification and *concern* location. In short, a typical concept location technique identifies the relevant units of a software system (e.g., classes or methods in C++) that implement a specific concept that originates from the domain of interest [11]. Existing approaches to concept location use different types of software analyses. They can be broadly classified based on their use of static, dynamic, and/or combined analysis.

In software engineering, LSI has been used for a variety of tasks such as software reuse [12, 13], identification of abstract data types [14], detection of high-level concept clones [15, 16], recovery of traceability links between software artifacts [17–20], identifying topics in source code [21], classifying and clustering software [22], measuring cohesion [23] and coupling [24], browsing relevant software artifacts [25], and tracing requirements [26, 27].

Using advanced IR techniques, such as those based on LSI [4, 28], allows users to capture relations between terms (words) and documents in large bodies of text. A significant amount of domain knowledge is embedded in the comments and identifiers present in source code. Using IR methods, users are able to index and effectively search this textual data by formulating natural language queries, which describe the concepts they are interested in. Identifiers and comments present in the source code of a software system form a language of their own without a grammar or morphological rules. LSI derives the meanings of words from their usage in passages, rather than a predefined dictionary, which is an advantage over existing techniques for text analysis that are based on natural language processing [29]. Marcus *et al.* [4] introduced a methodology to index and search source code using IR methods. Subsequently, Poshyvanyk and Marcus [28] refined the methodology and 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 analogous to many existing web search engines.

(1) *Creating a corpus of a software system*: The source code is parsed using a developer-defined granularity level (i.e., methods, classes, or files) 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

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table I. 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

source code. In addition, we also created the corpus builder for large C++ projects, using *srcML* [30] and *Columbus* [31].

(2) *Indexing*: The corpus is indexed using LSI and its real-valued vector subspace representation is created. Dimensionality reduction is performed in this step, capturing the 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. The steps 1 and 2 are usually performed once, while the others are repeated until the user finds the desired parts of the source code.

(3) *Formulating a query*: A set of terms that describe the concept of interest constitutes the initial query, e.g., the short bug description of a bug or a feature described by the developer or reporter. 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 edit distance and removes the term from the search query.

(4) *Ranking documents*: Similarities between the user query and documents from the source code (e.g., 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 for the generation of a ranked list of documents relevant to the concept. All the documents are ranked by the similarity measure in descending order (i.e., the most relevant at the top and the least relevant at the bottom).

LSI offers many unique benefits when compared to other natural language processing techniques, among which are included 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 are similar to its previous usages; we refer the interested readers to [4, 28].

Here, we demonstrate the working of the approach using an example from *KOffice*. The change request or concept of interest is the bug# 173881[§] that was reported to the bug-tracking system (maintained by *Bugzilla*) on *2008-10-30*. The reporter described the bug as follows:

splitting views duplicates the tool options docker.

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 the class-level granularity (i.e., each document is a class). After indexing with LSI, we obtained a corpus consisting of 4756 documents and containing 19 990 unique words. We formulated a search query using the bug's textual description, 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 I.

[§] http://bugs.kde.org/show_bug.cgi?id=173881 (verified on 10/21/09).

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

```
<?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 a *KOffice* subversion log message.

2.2. *Recommending developers from mining software repositories*

We use the *xFinder* approach to recommend expert developers by mining version archives of a software system [5]. 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 changes. 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 developers who can assist in a given part of the source code can be acquired in a straightforward manner.

Our approach uses the commits in repositories that record source code changes submitted by developers to the version-control systems (e.g., *Subversion*[¶] *and CVS*). Commits' log entries include the dimensions *author*, *date*, and *paths* (e.g., files) involved in a change-set. 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*.

We developed a few measures to gauge developer contributions from commits [5]. 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 by the developer–code vector *DV* for the developer d and file f.

$DV_{(d,f)} = \langle C_f, A_f, R_f \rangle$, 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, i.e., calendar days, 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 file-change vector *FV* representing the change contributions to the file f, is shown below:

$FV_{(f)} = \langle C'_f, A'_f, R'_f \rangle$, 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 that include changes to the file f.
- R'_f is the most recent workday with a commit that includes the file f.

[¶]Subversion is an open-source version control system and is available at http://subversion.tigris.org/ (verified on 10/21/09).

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

The measures C_f, A_f, and R_f are computed from the log history of commits that is readily available from source code systems, such as *CVS* and *Subversion*. More specifically, the dimensions *author*, *date*, and *paths* of log entries are used in the computation. The dimension *date* is used to derive workdays or calendar days. The dimension *author* is used to derive the developer information. The dimension *path* is used to derive the file information. Similarly, the measures C'_f, A'_f, and R'_f are computed. The log entries are readily available in the form of *XML* and straightforward *XPath* queries were formulated to compute the measures.

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 (for the details on computing this measure refer to [5]). Distance is the opposite of similarity, thus lesser the value of the *Euclidean* distance, greater the similarity between the vectors. *xFactor* is an inverse of distance and is given as follows,

$$xFactor_{(d,v)} = \frac{1}{|DV(d,f) - FV(f)|}$$

$$xFactor_{(d,v)} = \frac{1}{\sqrt{(C_f - C'_f)^2 + (A_f - A'_f)^2 + (R_f - R'_f)^2}}$$

We use *xFactor* as a basis 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 ranking is directly proportional to the *xFactor* values. That is, the developer with the highest value is ranked first, one with the second highest value is ranked second, and so forth. Multiple developers with the same values are given the same rank. Now, some files may have not changed in a very long time or added for the very first time. As a result, there will not be any recommendation at the file level. To overcome this problem, we look for developers who are experts in a package that contains the file, and recommend them instead. If no package expert can be identified, we turn to the idea of system experts as a final option. By doing so, we strive for guaranteed recommendation from our tool.

Package here means the immediate directory that contains the file, i.e., we consider the physical organization of source code. We define the *package expert* as the one who updated the largest number of unique files in a specific package. We feel the package experts are a reasonable choice and a developer with experience in several files of a specific package can most likely assist in updating a specific file in that package. As a final option, if no package expert can be identified, we turn to the idea of a system expert. The system means a collection of packages. It can be a subsystem, a module, or a big project (e.g., *kspread*, *KOffice*, and *gcc*). The *system or project expert* is the person(s) who is involved in updating the largest number of different (unique) files in the system. The person who updated the most files should have more knowledge about the system. In this way, we move from the lowest, most specific expertise level (file) to the higher, broader levels of expertise (package then system). According to this approach, we guarantee that the tool always gives a recommendation, unless this is the very first file added to the system. The procedure for the suggested approach is given in Figure 2. The integer parameter *maxExperts*, i.e., the maximum of developer recommendations desired from the tool, is user defined. The number of recommendations follow the property $maxFileExperts + maxPackageExperts + maxSystemExperts <= maxExperts$, i.e., the total number of recommendations possible from all the three levels collectively is less than or equal to the user-specified value. To help understand the process we now present a detailed example/scenario of using our approach.

Now, we demonstrate the working of the second step of our approach, i.e., *xFinder*, using the *KOffice* bug example from Section 2.1. The classes from Table I, given by 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; however, it is possible that multiple classes are implemented in a single file. *xFinder* is used to recommend developers for one file at a time. Here, we limit our discussion to a single file, i.e., the file containing the

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

```
def Recommender (f, p, maxExperts, h)
 # f: the file name
 # p: the package name that contains f
 # maxExperts: maximum number of recommended developers
 # h:  the period of history
 for each developer d in h:
    if  xFactor(f,d) > 0: fileList.add(d)
 #descending sort for developers in fileList by xFactor values
 sort(fileList,  xFactor, reverse=true)
 #print the first maxFileExperts developer
 printList(fileList,maxExperts)
 if fileList.size( ) >= maxExperts: Exit
 for each developer d in h:
     #no. of files in package p updated by d
     if fileCount (p,d) > 0: packageList.add(d)
 # descending sort for developers in fileList by fileCount values
 sort(packageList,fileCount, reverse=false)
 #print the first maxExperts- fileList.size( ) developer
 printList(packageList,maxExperts- fileList.size( ))
 If maxExperts- fileList.size( )=0: Exit
 for each developer d in h:
     #no. of files in the whole system updated by d
     if fileCount (d) > 0: sysList.add(d)
 # descending sort for developers in sysList by fileCount values
 sort(sysList,fileCount,reverse=false)
 #print the first maxSysExperts developer
 printList(fileList, maxExperts- packageList.size() - fileList.size( ))
```

Figure 2. The procedure used in *xFinder* to give a ranked list of developer candidates to assist with a given source code entity.

Table II. Candidate expert developers recommended by *xFinder* to the *KoDockerManager.cpp* file in *KOffice*.

Rank	File experts	Package experts
1	*jaham*	*mpfeiffer*
2	*boemann*	*jaham*
3		*zander*

top most class obtained from our concept location step. The file *guiutils/KoDockerManager.cpp* contains the top-ranked class *KoDockerManager*. *xFinder* started with *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 configured *xFinder* to recommend a maximum of five developers. The ranked list of developer user IDs that *xFinder* recommended at file and package levels are provided in Table II. Further details on *xFinder* are provided in our previous work on this topic [5].

The bug# 173881 was fixed on 2008-11-02 by a developer with the user id *jaham* and the patch included the file *KoDockerManager.cpp*. As can be clearly seen, both the appropriate file and developer were ranked first in the respective steps of our approach. Next, we systematically assess the accuracy of our approach.

3. CASE STUDY

We conducted a case study to empirically assess our approach according to the design and reporting guidelines presented in [32].

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

3.1. Design

The case of our study is the event of assigning change requests to developers in open-source projects. The units of analysis are the bug and feature requests considered from three open-source projects. The key objective is to study the following research questions (RQs):

RQ1: How accurate are the developer recommendations when applied to the change requests of real-world systems?

RQ2: What is the impact on accuracy with different amounts of training data, i.e., commits?

RQ3: How do the accuracy results compare when the original descriptions of bug/feature requests and their sanitized versions are formulated as LSI queries?

RQ4: How does our approach compare to alternate sources for developer recommendations (e.g., wild/educated guesses)? In particular, we posit the following two sub-questions:

RQ4.1: Are the developers recommended by our approach more accurate than randomly selected developers?

RQ4.2: Are the developers recommended by our approach more accurate than the maintainers, an obvious first choice, that are typically listed as the main points of contact in open-source projects?

The first two research questions related to the explanatory purpose, i.e., positivist prospective, of our study, as to what is the correlation of the developer recommendations from our approach with the developers who actually address change requests in open-source projects (see the rest of Section 3). The last two research questions relate to the comparative part of the study, as to how our approach compares with alternative methods (see the rest of Section 4).

We collected a fixed set of qualitative data, i.e., change and bug requests, from the software archives found in bug and source code repositories. We used a data triangulation approach to include a variety of data sources from three open-source subject systems, such as, *ArgoUML*, *Eclipse*, and *KOffice* that represent different main implementation languages (e.g., C/C++ and Java), sizes, and development environments.

KOffice is an application suite, which is about 911 KLOC. It includes several office productivity applications, such as word processing (*KWord*), spread sheet (*KSpread*), presentation (*KPresenter*), and drawing/charting (*KChart*) tools. *KOffice* closely follows the *KDE* (*K Desktop Environment*) development model and has component dependencies (e.g., core *KDE* libraries), but has an independent release cycle. We considered the release 2.0 series (different version cycles from *KDE*'s). *Eclipse*[‖] is an open-source integrated development environment, which is used in both research and industry, and provides a good representation for a large software system. We considered its releases 2.0, 3.0, and 3.3.2. Version 2.0 was released on 27 June 2002, version 3.0 was released on 25 June 2004, and 3.3.2 was released on 21 February 2008. For each version we studied the change history before the release date. *Eclipse* project contains millions of lines of code and tens of developers. For example release 3.0 contains 1 903 219 lines of code with 84 committers. *ArgoUML*[**] is an open-source tool that provides a UML modeling environment. It has gone through eleven releases, and the latest stable version is 0.28. We used version 0.26.2 in our evaluation. The commit history from version 0.20 to candidate release version 0.26.1 was used and had contributions from a total of 19 developers.

Guided by the *Goal-Question-Metric (GQM)* method, the main goal of the first part of our study is to assess the accuracy effectiveness of our approach, asking *how accurate are the developer recommendations when applied to the change requests of real systems*? That is, investigate the research questions RQ1 and RQ2. The main focus is on addressing different viewpoints, i.e., theory triangulation, of recommendation accuracy:

(1) *Request-Level Accuracy*: Does the approach include among the recommended developers the ones that actually contribute changes needed for a given change request?

[‖]http://www.eclipse.org/ (verified on 10/21/09).
[**]http://argouml.tigris.org/ (verified on 10/21/09).

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

(2) *Expertise-Granularity Accuracy*: How does the recommendation accuracy vary across different levels of granularity, i.e., file, package, and system, considered for developer expertise?
(3) *Ranking Effectiveness*: How many files (from a ranked list relevant to a concept) need to be queried for developer recommendations to get to the first accurate recommendation, i.e., effectiveness of the ranking?

The metrics corresponding to the above three questions are discussed in Sections 3.3, 3.4, and 3.5. These metrics enable a quantitative analysis of the results obtained in our study.

3.2. Evaluation procedure and protocol

The bug/issue tracking and source code changes in software repositories, i.e., commits, are used for evaluation purposes. Our general evaluation procedure consists of the following steps:

(1) Select a change request (e.g., a bug or feature/wish) from the bug-tracking system that is resolved as fixed (or implemented).
(2) Select a development version on or before the day, e.g., a major or minor release or snapshot of the source code repository, at which the selected change request was reported (but not resolved) and apply concept location to get a ranked list of relevant source code classes and files given its textual description.
(3) Use *xFinder* to collect a ranked list of developers for the classes from step 2.
(4) Compare the results of step 3 with the *baseline*. The developers who resolved the issue, e.g., contributed patches or commits that fixed a bug, are considered the *baseline*.
(5) Repeat the above steps for N change requests.

We first show the evaluation of our approach on *KOffice*, as a primary aid to highlight the details, and take a result-oriented view of the other two systems. The development of the release 2.0 series started after the second maintenance release 1.6.3 in July 2007. Thereafter, over 90 developers have contributed source code changes. In the 2.0 series timeframe:

- The *soft-freeze* (no new feature/functional requirement addition except those planned), after the development period of about a year, was announced in mid July 2008.
- The first beta version, after the *hard-freeze* (no new feature/functional requirement addition permitted), was released in mid September 2008.
- *Beta 2* and *Beta 3* were released in October 2008 and November 2008, respectively.
- The latest beta release, *Beta 4*, rolled out in early December 2008. The beta versions are primarily focused on corrective or perfective maintenance issues, such as bug fixes, feature improvements, and user interface issues, and user feedback/testing.

We sampled 18 bugs (out of 128 total) from *KOffice* that were fixed during the period between the releases *soft-freeze* and *Beta 04*. The resulting sample was checked to include representatives via consideration of factors such as priority, severity, and description of the bug (similar to stratified sampling). These bugs were reported to the *KOffice* issue-tracking system, which is managed by *Bugzilla*. A bug report typically includes a textual description of the reported problem. Besides the status of the sampled bugs marked as *resolved* or *fixed* in *Bugzilla*, there were patches/commits contributed toward these bugs in the source code repository. In this way, we ascertained that the sampled bugs were considered fixed with source code contributions.

For each bug from our sampled set, we applied our concept analysis tool to the source code in the release *Alpha 10*, which was the most recent version before the *soft-freeze*. The short textual descriptions of the bugs, i.e., change requests, were directly used as LSI queries with respect to the features or bugs that we were locating (see Tables IV and V). We configured the tool to report only the top ten relevant source code files. Our previous work on the analysis of commits that are associated with bug fixes guided the choice of this particular cut point [33]. This satisfies steps 1 and 2 of our evaluation procedure.

In the second part of our approach, each file that contains the relevant classes from concept location in the first part is fed to *xFinder* to recommend a maximum of a total of ten developers at

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

the file, package, and system levels collectively. *xFinder* was configured to use only the commits in the source control repository before the release *Alpha 10*. This satisfies step 3 of our evaluation procedure.

A common policy or heuristic that developers follow in *KOffice*, like several other open-source projects, is to include keywords, such as the *bug id* of a reported bug (in *Bugzilla*) and *FEATURE* or *new feature* for features, in the text message of the commits that are submitted with the purpose to fix/implement it. We used this information from commits to further verify that these bugs were actually (or at least attempted to be) fixed. Furthermore, we also know the specific developer(s) who contributed to these bug-fix commits. In this way, we can find the exact developer(s) who contributed to a specific bug fix. Such developers for the sampled bugs and features are used as a *baseline* for evaluating our approach.

Let the sampled evaluation set of change requests be:

$CR\_Eval = \{b_1, b_2, \ldots, b_n\}$, where each b_i is one of the n resolved, fixed, or implemented requests chosen for evaluation.

The baseline, *BL*, is then a set of pairs of an issue and a set of developers, who contributed source code changes to this issue:

$GT = \{(b_t, \{d_t\}) | bug\_id(b_t) \in ct \wedge d \in c_t\}$, where b_t is the change request and d_t is the developer who contributed at least one commit, c_t, that includes the bug id of b_t or submitted a patch to the bug-tracking system. Next, we compare the developer recommendations from our approach with the baseline and assess the three questions discussed at the beginning of Section 3.1. This suffices step 4 of our evaluation procedure.

Additionally, we also sampled five features and one refactoring request. For feature requests, we used textual descriptions of implemented features, which were recorded in the *svn* commit logs. We treated the developer-specified textual messages of the commits that implemented features (or refactorings) as LSI queries and such commits were excluded from the history used for *xFinder*. That is, the goal was to see how well our approach would have performed if these commit messages were used as change requests? For example, Table V shows the commit# 832876 from *KOffice* that was contributed after the release *Alpha 10*. The textual description '*rudimental authentication support to test out pion net authentication features*' was formulated as an LSI query to locate the relevant class (files) in the release *Alpha 10. xFinder* used these files to recommended developers from the commit history before the release *Alpha 10*, which were compared with the actual developer who contributed the commit *# 832876* to compute accuracy. It should be noted that this view of a feature is different from what is submitted as a feature request or a wish in the issue-tracking system.

3.3. Request-level accuracy

Request-level accuracy is the percentage of change requests for which there was at least one developer recommendation from our approach that matched their established baselines (similar to the widely used *recall* metric in IR, i.e., what percentage of the considered change requests had correct developers recommended?). In other words, did our approach find the correct developer for a given change request? We separately compute this level of accuracies for the 18 bugs, five features, and one refactoring sampled from *KOffice*. For example, the pair (bug# 167009, {*zander*}) was found in the baseline of the bug# 167009[††] taken from our sample of the *KOffice*'s *Bugzilla* system. For this bug, there was at least one recommendation from our approach that included the developer *zander*. Therefore, we consider that this developer was correctly recommended from the perspective of the request-level accuracy.

There was at least one correct recommendation for 11, 15, and 13 bugs at the file, package, and system levels of developer expertise considered by *xFinder*. This gives *request-level accuracy* of 61% (11/18), 83% (15/18), and 72% (13/18) at the three respective levels. There was at least one correct recommendation for each of 16 bugs. This gives the *request-level accuracy* of

[††]http://bugs.kde.org/show_bug.cgi?id=167009 (verified on 10/21/09).

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

```
# Procedure: Compute accuracy of recommendation
# for a given change request
def rec_accuracy(bs)
 # File, package, system, and overall accuracies
 bs_acc = (facc, pacc, sacc, oacc) = (0, 0, 0, 0)
 rcs = conceptLSI(bs, rcs_size = 10)
 for class in rcs:
  f = file(class)
  (fexp, pexp, sexp = xFinder( f, dmax=(10,10,10) )
 # Match: checks if the developers in the baseline
 # are covered in the recommendation lists
  if ( bs, set(fexp)) in GT: fc =T, bs_acc [0]+=1
  if ( bs, set(pexp))in GT: pc =T, bs_acc [1]+=1
  if (bs,set(sexp)) in GT: sc = T, bs_acc [2]+=1
  if fc or pc or sc: bs_acc [3]+=1
# Accuracy in terms of percentages
def bs_acc_perct (x): return x*100/size(rcs)
 return map(bas_acc_perct, bs_acc)

# Procedure: Compute accuracy of recommendation
# for the sampled change requests
def sample_accuracy()
 for bs in CR_Eval:
      acc = rec_accuracy(bs)
```

Figure 3. Procedure for computing recommendation accuracy of the approach in the evaluation process on the *KOffice* application suite (*Python*-like code).

88.89% (16/18). That is, when *request-level accuracy* is looked at irrespective of the granularity of expertise from *xFinder*, the approach was able to provide at least one correct recommendation for each of 16 bugs (and there was not a single correct recommendation for the remaining two bugs). This is because there were cases where one level of granularity correctly recommended the developer for a bug and another level did not.

3.4. Expertise-granularity accuracy

Our approach provides expert developer recommendations for every retrieved class (file) that is identified as relevant via concept location. We investigated how many of these recommendations matched with the established baseline in the above discussion. This also helps us to see how the accuracy varies with the number of relevant classes (and files) used for recommendation (similar to the widely used *precision* metric in IR, i.e., how many recommendations are false positives for a change request?). In our evaluation on *KOffice*, we considered only the top ten relevant classes for each change request. Thus, we get ten sets of recommendations from *xFinder*. A recommendation set for a given bug or a feature is considered accurate if it includes the developers who are recorded in the baseline of that bug. The accuracy is further examined at file, package, and system levels of detail. The procedure used to compute the accuracies for sampled change requests is detailed in Figure 3.

We explain the accuracy computation with the help of the bug# 167009, shown in Table III. This bug is described as '*kword crashes after deleting not existing page*'. We used the textual description of this bug as an LSI query (see Table IV) and obtained the top ten relevant source code files from the release *Alpha 10*. Examining the *Subversion* repository, we found that the developer *zander* contributed a commit for this bug, i.e., the bug# was mentioned in the commit message. (The same author id was found in the *Bugzilla* repository.) The pair (bug# 167009, {*zander*}) was formed as the baseline for this bug. Table III shows the accuracy of the recommendations for this bug on the files that contain the top ten relevant classes. In this case, our approach obtained accuracies of 40% (4/10), 100% (10/10), and 100% (10/10) at the file, package, and system levels respectively. That is, the correct developer appeared in the candidate list at the file-level expertise for files containing four relevant classes, whereas, it did not appear for files for the

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table III. Accuracies of developer recommendations for bug# 167009 *from KOffice* at the File, Package (Pack), System (Sys), and overall (Ovl) levels. Value 1 (0) indicates that the actual developer was (not) in the list of recommendations.

	Recommendation coverage			
Files with relevant classes	File	Pack	Sys	Ovl
TestBasicLayout.cpp	0	1	1	1
KWPageTextInfo.cpp	0	1	1	1
KWPageManager.cpp	1	1	1	1
KWPageManager.cpp	1	1	1	1
TestPageManager.cpp	0	1	1	1
KWPageManagerTester.cpp	0	1	1	1
KWPageInsertCommand.cpp	1	1	1	1
KWPageSettingsDialog.cpp	1	1	1	1
KPrMasterPage.cpp	0	1	1	1
KPrPageEffectSetCommand.cpp	0	1	1	1
Accuracy (%)	40	100	100	100

six relevant classes (i.e., all the recommended developers were false positives, i.e., not the ones who actually fixed the bug). At the package and system levels of expertise, i.e., the package or system, containing the relevant class, did not have any false positives in this case. Our approach is considered overall accurate, if there is at least one file, package, or system-level recommendation that is correct. This result again shows that our approach can recommend developers with a high accuracy.

Tables IV and V show the expertise-granularity accuracies for the bugs and features+refactoring change requests, respectively. The accuracies at the different levels of granularities are computed according to the bug# 167009 example described above. None of the granularity levels independently show a particularly high accuracy; however, the overall accuracies are 80% and 43% for the two types of change requests, higher than any of the file, package, and system expertise. This further suggests that different levels of granularities need to be taken into account for accurate recommendations, which was exactly done in our approach. Also, the accuracy of features is less than that of bugs (further discussion in Section 3.5).

3.5. Ranking effectiveness

Ranking effectiveness determines how many files in the ranked list that are retrieved by the concept location tool need to be examined before *xFinder* obtains the first accurate recommendation (i.e., finds the correct expert developer). This issue directly corresponds to the amount of change history that is needed and analyzed. If fewer relevant files are needed to get the correct developer then a smaller subset of change history is needed. It is desirable to have an approach that needs only a change history of a single class/file to function accurately (compared to another that requires a change history of more than one classes/files). Next, we see how manually selected values of ten classes fare with regard to the recommendation accuracy? Obviously, it is desirable to have the classes/files that give the best recommendation accuracy appear sooner than later.

We explain the computation of ranking effectiveness for different granularities with the help of the results presented in Table IV. As mentioned in Section 3.3, the correct recommendations are generated for 11, 15, and 13 bugs at the file, package, and system levels of developer expertise. Thus, the ranking effectiveness for the ranked list of ten files is 61%, 83%, and 72% at the three respective granularity levels.

We explored in how many cases different granularities (e.g., file, package, system, and overall) return relevant files that produce correct developer recommendations within the top 1, 3, 5, and 10 results, as well as cases when no correct recommendation is found in the top ten results. We analyzed ranking effectiveness of different granularities for bugs and features separately (see, Tables VI and VII). On average across the three different granularities, which also includes overall

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table IV. Summary of developer recommendation accuracies obtained for 18 *KOffice* bugs at the File, Package (Pkg), System (Sys), and Overall (Ovl) levels using their descriptions directly from the repositories, i.e., automatic queries. For each bug, the accuracy values are provided for files with the top ten relevant classes. The ranks for the first relevant recommendation on file (R_f), package (R_p), system (R_s), and overall (R_o) levels are provided. A '—' is specified if none of the files in the top ten list generates a correct recommendation.

		Accuracy (%)				Ranking effectiveness			
Bug #	Bug description excerpt/LSI Query	File	Pkg	Sys	Ovl	R_f	R_p	R_s	R_o
124527	Display of calculation results incorrect for number or scientific format cells but correct for text format cells	0	0	0	0	—	—	—	—
125306	Cannot undo insert calendar tool	0	40	100	100	—	2	1	1
130922	Import from xls treats borders wrong	0	0	100	100	—	—	1	1
137639	Karbon imports inverted radial gradient colors from open document graphics with patch	10	60	0	60	8	4	—	4
140603	!= in cell formula is buggy	0	30	100	100	—	3	1	1
141536	Crash when pasting a new layer srcobject selected from a deleted layer	20	60	100	100	6	2	1	1
156754	Cannot rotate image by less than 0 5 degrees	0	40	0	40	—	2	—	2
162872	With filter paintop transparent pixels are replaced by black or white pixels	50	70	100	100	1	1	1	1
164688	Cut copy paste is greyed when using the selection tool	0	70	100	100	—	4	1	1
166966	Karbon14 crashes while closing unsaved document	0	0	0	0	—	—	—	—
167009	Kword craches after deleting not existing page	40	100	100	100	3	1	1	1
167247	Tooltips for resource choosers broken	30	40	0	40	2	1	—	1
169696	Loaded shapes at wrong place	30	100	100	100	1	1	1	1
171969	Decoration not in sync with shape	30	90	100	100	3	1	1	1
173354	Start presentation from first page does not start from first page of custom slide show	20	90	100	100	7	1	1	1
173630	Format page layout does nothing	10	70	100	100	10	2	1	1
173882	Cannot set splitter orientation	10	90	100	100	10	1	1	1
176278	Crash on loading image	30	40	100	100	5	4	1	1
	Average	16	55	72.2	80.0	5.1	2	1	1.3

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table V. Summary of developer recommendation accuracies, obtained using automatic queries, for five (5) feature requests and one (1) refactoring of *KOffice* at the File, Package (Pkg), System (Sys), and overall (Ovl) levels. For each feature, the accuracy values are provided for files with the top ten relevant classes. The ranks for the first relevant recommendation on file (R_f), package (R_p), system (R_s), and overall (R_o) levels are provided as well. A '—' is specified if none of the files in the top ten list generates a correct recommendation.

		Accuracy (%)				Ranking effectiveness			
Commit #	Feature description excerpt/LSI Query	File	Pkg	Sys	Ovl	R_f	R_p	R_s	R_o
832876	Rudimental authentication support to test out pion net authentication features	0	0	0	0	—	—	—	—
835741	Refactoring presenter view feature enabling the presenter without opening a new view	0	0	0	0	—	—	—	—
846840	Page master styles header and footers in a document multiple page layout per document	50	10	0	50	1	3	—	1
847576	add support for input device angle rotation 4d mouse and tilt tablet pens are supported	10	10	0	10	7	7	—	7
868014	New feature kwpage setpage number which just updates the cache of page numbers	60	100	100	100	1	1	1	1
881241	Lens distortion using oldrawdata and counter to simulate old behavior of deform brush	20	20	100	100	1	1	1	1
	Average	23.3	23.3	33.3	43.3	2.5	3	1	2.5

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table VI. Automatic queries: Summary of an average ranking effectiveness for bugs of *KOffice* using different granularities (file (R_f), package (R_p), system (R_s), and overall (R_o)) for top 1, 3, 5, 10 recommendations.

	Average ranking effectiveness for bugs				
	Top 1 (%)	Top 3 (%)	Top 5 (%)	Top 10 (%)	No rec. (%)
R_f	11.11	27.78	33.33	61.11	38.89
R_p	38.89	66.67	83.33	83.33	16.67
R_s	72.22	72.22	72.22	72.22	27.78
R_o	77.78	83.33	88.89	88.89	11.11
Avg	50.00	62.50	69.44	76.39	23.61

Table VII. Automatic queries: summary of an average ranking effectiveness for features of *KOffice* using different granularities (file (R_f), package (R_p), system (R_s), and overall (R_o)) for top 1, 3, 5, 10 recommendations.

	Average ranking effectiveness for features				
	Top 1 (%)	Top 3 (%)	Top 5 (%)	Top 10 (%)	No rec. (%)
R_f	50.00	50.00	50.00	66.67	33.33
R_p	33.33	50.00	50.00	66.67	33.33
R_s	33.33	33.33	33.33	33.33	66.67
R_o	50.00	50.00	50.00	66.67	33.33
Avg	41.67	45.83	45.83	58.33	41.67

accuracies of three granularities, in 50% of cases the first relevant file is found in the first position of the ranked list. These results are quite encouraging and support the previous results in the literature [11] that IR-based concept location approach is robust in identifying the first relevant method to a feature of interest. It should be noted that the system granularity gives the highest ranking effectiveness (i.e., 72.22%), whereas the file granularity produces the lowest (i.e., 11.11%) for the topmost recommendation. We also observed that the system-level granularity has the highest ranking effectiveness across the considered ranked lists of different sizes. It should also be noted that the accuracies for file and package granularities improve drastically with the increase in the size of the ranked list (i.e., the ranking effectiveness for the top ten list is 61.11% as opposed to 11.11% for the top one list). The logical *OR* overall accuracy of three granularities (R_o) has consistently high-ranking effectiveness values ranging between 77.78% and 88.89% for the resulting lists of various sizes.

The results for the analyzed features are more encouraging in terms of identifying the first relevant file for the file-level expertise granularity. In 50% of cases the relevant file appears in the first position; however, the overall ranking effectiveness is also relatively high (66.67%) for the ranked lists containing ten files. For features, we observed somewhat different patterns of ranking effectiveness for different granularity levels. For example, we observed that the file granularity performs the same as, or better than, both package and system granularities in all the cases. It should be noted that the file granularity for features performed better than the file granularity for the bug reports. On the contrary, the ranking effectiveness results for package and system levels of granularity were consistently higher across resulting recommendation lists of various sizes for the bug reports than those for features. Overall, the results clearly indicate that our approach can recommend relevant files, and thus, correct developers, with high effectiveness for both bugs and features. We also observed the potential value in using granularities finer than the system level, such as file and package granularities, as they did contribute to the overall increase in ranking effectiveness.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

3.6. Accuracy and history periods

Another factor that could affect our approach is the amount of history period. Here, we repeated the setup of Section 3.1 for 18 bugs; however, with a different set of commit histories of *KOffice*. In other words, *what is the impact on accuracy if different amounts of previous commits are considered*? An answer to this question helps us gain insight into how much history is necessary for our approach to function well in practice. Table XI shows the results of our accuracy assessment for periods of 17 days, 1 month, 2 months, 4 months, and entire duration. That is, *xFinder* was configured to consider only the most recent commits for the duration considered from the time the issue was reported. The overall accuracy results suggest that the recommendation accuracies at all the expertise levels increase with the increase in the history duration. We noticed that the commit history in the order of weeks was necessary to get correct recommendations. For example, the file-level granularity in *KOffice* began to show accurate recommendations with a history period of 17 days. The overall accuracies in *KOffice* crossed the 50% mark with at least a month history. As can also be seen in *KOffice*, the best accuracies were obtained when its entire commit histories were considered. We did not record a single instance where there was a decline in accuracy with the increase in the commit history. In some instances, the increase in accuracy was rather small, especially when the increment was in the order of months. This behavior is seemingly obvious due to the natural evolution of the systems, i.e., the development phases into the next release than the previous, and therefore the immediate previous history may not be that relevant.

Now that we have presented the three views of accuracy assessment with an in-depth discussion on *KOffice*, we turn to the results obtained on the two other systems. Table XI shows the average accuracy and ranking effectiveness results for the three different releases of *Eclipse* and different *xFinder* configurations on *ArgoUML*. The bugs used in the evaluation were sampled based on similar criteria used for *KOffice* (see online appendix). The accuracies at the different levels of expertise granularities and ranking effectiveness were computed using the same process described for *KOffice*. Table XI shows only the average accuracy results, which correspond, for example, to the last rows in Tables IV and Table VIII. The accuracy results of *Eclipse* releases 2.0 and 3.0 (e.g., overall accuracies of 75% and 82%) are comparable to that of *KOffice* (80%); especially when its entire commit history is considered. Also, we observed an improved ranking effectiveness performance for *Eclipse* in particular. We achieved the best accuracy results of 95% in *ArgoUML*, when *xFinder* was configured to recommend the maximum of ten developers; however, we did notice a decline in accuracy when this number was reduced to five developers. Also, we have to be careful as *ArgoUML* had the least number of active developers among our considered subject systems. Overall, our assessment results suggest that our approach can yield an equivalent accuracy across different systems and releases of the same system.

The above discussion provides empirical answers to the research questions RQ1 and RQ2 of our case study. Thus, concludes the first half of our study.

4. COMPARATIVE STUDY ON KOffice

4.1. Effectiveness of automatic versus manual LSI queries

The ranked lists of files generated by LSI are somewhat sensitive to the input query [4, 34, 35]. In Section 3, the original textual descriptions from the bug reports were automatically used as LSI queries. Here, we investigate the impact of different formulations of the queries by different developers (e.g., different choices of words or simply without typographical errors—a situation not uncommon in collaborative environments such as the open-source development). Here, we consider the short descriptions of the bugs taken verbatim from the bug/issue repositories or commit messages for LSI queries as *automatic* queries. We repeated our evaluation with the same bugs and features; however, their textual descriptions were sanitized by one of the authors, which we refer to as *manual* queries.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table VIII. Summary of developer recommendation accuracies for 18 *KOffice* bugs at the File, Package (Pkg), System (Sys), and Overall (Ovl) levels using the manually formed queries. For each bug, the accuracy values are provided for files with the top ten relevant classes. The ranks for the first relevant recommendation on file (R_f), package (R_p), system (R_s), and overall (R_o) levels are provided. A '—' is specified if none of the files in the top ten list generates a correct recommendation.

		Accuracy (%)				Ranking effectiveness			
Bug #	Sanitized LSI query	File	Pkg	Sys	Ovl	R_f	R_p	R_s	R_o
124527	Number value scientific format	20	20	0	20	5	5	—	5
125306	Insert calendar undo	0	30	70	100	—	2	1	1
130922	Import xls excel	0	0	90	90	—	—	1	1
137639	Karbon inverted radial gradient color	20	60	0	60	1	1	—	1
140603	Formula equal not cell	0	10	50	60	—	4	1	1
141536	New layer image srcobject select	0	70	20	80	—	1	1	1
156754	Rotate image	0	50	0	50	—	2	—	2
162872	Filter paintop pixel transparency	10	50	90	100	2	1	1	1
164688	Selection select tool	0	60	20	70	—	1	2	1
166966	Close document save discard	10	10	0	20	8	10	—	8
167009	Delete document	90	100	20	100	1	1	5	1
167247	Tooltip resource chooser	60	70	0	90	1	1	—	1
169696	Load shape path	40	90	40	100	2	1	4	1
171969	Selection decorator shape rotate	50	90	40	100	1	1	3	1
173354	Presentation slide show slide custom	20	50	50	100	1	1	2	1
173630	Format page layout view	10	70	30	100	10	2	1	1
173882	Splitter orientation horizontal vertical	50	70	30	90	1	1	1	1
176278	Load image	20	40	30	60	6	3	5	3
	Average	22.2	52.2	32.2	77.2	3.3	2.1	2.2	1.8

Table IX. Summary of developer recommendation accuracies for five (5) feature requests and one (1) refactoring using the manually formed queries at the File, Package (Pkg), System (Sys), and overall (Ovl) levels from *KOffice*. For each feature, the accuracy values are provided for files with the top ten relevant classes. The ranks for the first relevant recommendation on file (R_f), package (R_p), system (R_s) and overall (R_o) levels are provided as well. A '—' is specified if none of the files in the top-ten-list generates a correct recommendation.

		Accuracy (%)				Ranking effectiveness			
Feature #	Sanitized LSI query	File	Pkg	Sys	Ovl	R_f	R_p	R_s	R_o
846840	Page style master header footer page layout	40	20	0	50	2	3	—	2
881241	Lens distortion oldrawdata use counter deform brush	10	10	90	100	5	5	1	1
847576	Input device angle rotation 4D mouse tilt tablet pen	0	0	0	0	—	—	—	—
832876	Authentication security login pion net	0	0	0	0	—	—	—	—
868014	Kwpage setpagenumber style page number	40	90	10	100	1	1	9	1
835741	Presenter slide show custom enable presentation view	50	20	0	50	3	7	—	3
	Accuracy	28	28	20	60	2.8	4	5	1.8

The setup in Section 3 was repeated; however, with manually revised queries by one of the authors. The queries were designed to be self-descriptive and sanitized from typographical errors (see, Tables VIII and IX).

There was at least one correct recommendation for 12, 17, and 13 bugs at file, package, and system level of developer expertise considered by *xFinder*. This gives *request-level accuracy* of 66.67% (12/18), 94% (17/18), and 72.2% (13/18) at the three respective levels. When a *request-level accuracy* is looked at irrespective of the granularity of expertise from *xFinder*, the approach was able to provide at least one correct recommendation for all of the 18 bugs. This again shows that our approach can recommend developers with a very high accuracy and the potential value in considering granularities finer than the system level. Similar accuracy was observed for the feature requests.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table X. Manual queries: summary of an average ranking effectiveness for bugs and features in *KOffice* using different granularities (file (R_f), package (R_p), system (R_s), and overall (R_o)) for top 1, 3, 5, 10 recommendations.

	Top 1 (%)	Top 3 (%)	Top 5 (%)	Top 10 (%)	No rec. (%)
Average ranking effectiveness for bugs					
R_f	27.27	36.36	40.91	54.55	45.45
R_p	45.45	68.18	77.27	81.82	18.18
R_s	31.82	45.45	59.09	59.09	40.91
R_o	63.64	72.73	77.27	81.82	18.18
Avg	42.05	55.68	63.64	69.32	30.68
Average ranking effectiveness for features					
R_f	16.67	50.00	66.67	66.67	33.33
R_p	16.67	33.33	50.00	66.67	33.33
R_s	16.67	16.67	16.67	33.33	66.67
R_o	33.33	66.67	66.67	66.67	33.33
Avg	20.83	41.67	50.00	58.33	41.67

Tables VIII and IX show the expertise-granularity accuracies for the bugs and features+refactoring change requests, respectively. None of the granularity levels independently show a particularly high accuracy; however, the overall accuracies are 77% and 60% for the two types of change requests, higher than any of the file, package, and system expertise. This further suggests that different levels of granularities need to be taken into account for accurate recommendations, which was exactly done in our approach.

We again explored in how many cases different granularities (e.g., file, package, system, and overall) return relevant files within the top 1, 3, 5, and 10 results, as well as when no correct recommendation is found in the top ten. We analyzed the ranking effectiveness of different granularities for bugs and features separately (see, Table X). The analysis of the results for 18 bugs shows that in 42% of cases, on average the first relevant file is found in the first position of the ranked list across all granularities. The results also indicate that in 69% of cases the relevant file is found in the top ten recommendations. Moreover, the ranking effectiveness is 82%, meaning that only in 18% of cases the correct recommendation was not found. In only 30.6% of cases the ranked lists of results (i.e., the top ten) do not contain any relevant methods. We also observed that the package granularity has the highest effectiveness for top one (45.45%), top three (68.18%), top five (77.27%), and top ten (81.82%) results. The package granularity also has the lowest percentage of cases when no correct recommendation is returned within top ten (18.18%). We also observed that the system granularity consistently outperformed the file granularity for all list sizes in the bug requests.

The results for the analyzed features are slightly less encouraging in terms of identifying the first relevant file immediately. On average, only in 20.82% of cases did the relevant file appear in the first position across all the three granularities; however, the overall effectiveness is also relatively high (58.33%) for the ranked lists containing ten files. For the features, we observed somewhat different patterns in terms of granularity effectiveness. For example, we observed that file granularity performs better in case of the top three and top five results; however, it performs similar to the package granularity for the top one and top ten. Similar to the package granularity, the file granularity does not return any relevant files in 33.33% cases.

We attribute some of the differences in ranking effectiveness for bugs and features to the fact that we used words directly from bug reports, whereas we used words from commit messages for features. In our case, we found that the descriptions of the bug reports were more expressive and complete than the commit messages, which only briefly summarized the implemented features. This could have impacted the choice of words for LSI queries and thus, the ranking effectiveness in the results in Table X.

In summary, the accuracies of our approach using two querying techniques, i.e., automatic and user-refined queries, are generally comparable. These are very encouraging results, as the proposed

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

approach does not impose additional overhead on users in terms of constructing queries and more than that, does not require prior knowledge of change requests nor the project, which makes it attractive not only for experienced contributors, but also to newcomers. The descriptions in the issue/bug repositories can be directly used without much impact on accuracy.

For bug reports, the manual queries improved the average accuracy by 6.6% over the automatic queries at the file-level granularity. However, accuracies at the package and system granularities are higher for automatic queries: 55% versus 52.2% and 72.2% versus 32.2%. Automatic queries also improve the overall accuracy over manual queries by 2.8% (80% versus 77.2%). On the other hand, the ranking effectiveness is better for manual queries at the file-level granularity (3.3 for manual versus 5.1 for automatic). These results indicate that manual queries on average obtain the correct recommendation quicker (fewer files need to be examined) than automatic queries, which is not surprising, given the fact that users formulate the queries (and the human time involved therein). The ranking effectiveness for the overall, package, and system granularities is generally comparable for automatic and manual queries.

The comparison of accuracies between manual and automatic queries for features across the file, package, and system granularities yielded the following results. The revised queries, as in the results for the bugs, outperform automatic queries by 4.7% (28% versus 23.3%) at the file-level granularity. We also observed that the package-level accuracy is higher for manual queries (28% versus 23.3%), whereas the system-level accuracy yields better results for automatic queries (33% versus 20%). Overall accuracy is higher for manual queries (60% versus 43.3%), whereas the overall accuracy was higher for automatic queries in the case of bugs. In terms of ranking effectiveness (the number of files that have to be explored before the correct recommendation is obtained), automatic queries outperform manual queries across all granularities, i.e., file, package, and system. While using automatic queries, fewer files on average are used from the ranked list to obtain the pertinent recommendations.

While analyzing average ranking effectiveness for manual versus automatic queries using ranked lists of different sizes (1, 3, 5, and 10), we observed similar patterns as in the case of default size of the ranked list (i.e., top ten). For bugs, automatic queries outperform manual queries in terms of the average ranking effectiveness for ranked lists of sizes 1, 3, and 5 for the overall accuracy. The results for the features are slightly different. Overall, the average ranking effectiveness for automatic queries outperforms manual queries at the ranked list sizes for one (50% versus 33%); however, degrades for three and five, while being equivalent for ten. Based on the analysis of the results of accuracies at different levels of granularity, we can conclude that automatic queries perform, at least, as well as manually revised queries, even noticeably better in some cases.

In order to obtain more insights into distinctions between manual and automatic queries, we also explored the differences among the resulting ranked lists of relevant files for both automatic and manual queries. We analyzed the resulting ranked lists of different sizes ranging from top ten to 100 relevant files. We used *Jaccard* similarity coefficient[‡‡] to compare the similarity and diversity of the files appearing in the ranked lists of results for manual and automatic queries. In our case, the Jaccard index is computed as follows. Given two sets of files in the ranked list of the results, A (obtained using automatic queries) and B (obtained using manual queries), Jaccard index is computed as $J(A, B) = |A \cap B|/|A \cup B|$.

We report the average Jaccard measure for the ranked lists of various sizes, separately for bugs and features. Our analysis of the results suggests that the resulting ranked lists for automatic and manual queries for bugs and features are quite different, which is not surprising as some of the revised queries were quite different. However, the Jaccard measure values tend to increase as the size of the ranked list increases. Figure 4 indicates that the highest average Jaccard similarity coefficients for bugs and features are 0.33 and 0.56. In other words, the results indicate that on average ranked lists for manual and automatic queries contain more similar files for features than for bugs. Again, these observations strengthen our conclusions that our approach works comparably

[‡‡]http://en.wikipedia.org/wiki/Jaccard_index (verified on 10/21/09).

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

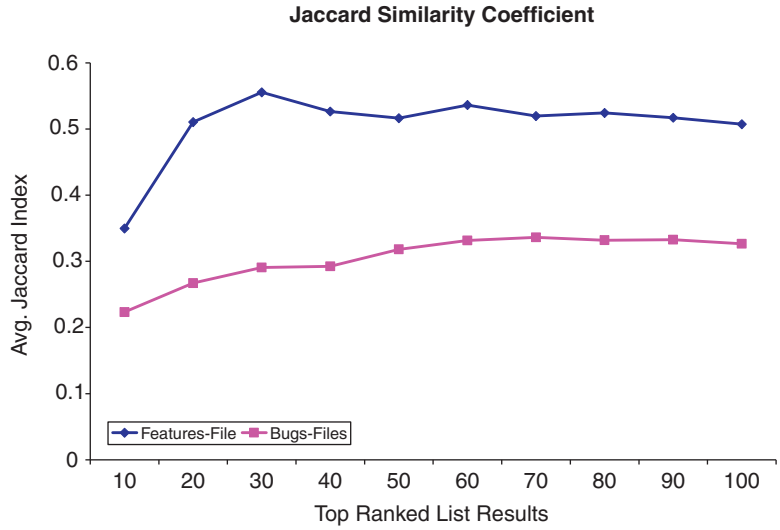

Figure 4. Comparing the similarity and diversity of files for automatic and manual queries obtained from concept location in KOffice.

well for automatic and manual queries even though they might produce different relevant files. This observation also leads to the conjecture that in order to produce pertinent recommendations (i.e., identify relevant expert developers) we do not need to locate 'exact' files that have been changed in order to locate the correct expert. We posit that locating conceptually (i.e., textually) similar files also have the potential to generate the correct recommendations, as it is reasonable to conjecture that the developers change related files throughout software evolution, as captured by contribution measures in our *xFinder* approach. We are planning an in-depth exploration of this phenomenon as our future work.

4.2. Accuracy effectiveness with alternate recommendations

We investigated how the accuracy of our approach compares to some straightforward recommendation approaches. Here, we focus our discussion on *KOffic*e (partially because it contained the largest number of developers among the data sets considered).

To answer question RQ4.1, we randomly selected ten developers for each of the same 18 bugs considered in our evaluation. That is, 18 samples of ten developers each. The value ten was chosen to match with the number of maximum developers chosen for our approach. The same history period used in *xFinder* for recommendations was used to obtain the total number of active developers (i.e., 93 developers who contributed at least one commit—the sample space). A bug was considered to have an accurate recommendation, if the correct developer, i.e., the one who ended up fixing the bug, was one of the ten randomly chosen developers. A request level accuracy of 22% (four correctly recommended/18 total bugs) was obtained from this random developer model. Comparing this result with the request-level accuracies in Section 3.3 shows that our approach provides substantially better accuracy as compared to a random selection method (approximately thrice accurate). We repeated a similar experiment to see how the results compared when a random selection was made for each pair of a bug and a relevant file obtained using LSI (essentially the same setup is used in our approach to obtain the results). We obtained an accuracy of 22% for overall expertise-level granularity (refer Section 3.4). Once again, a comparison of this result with the *KOffice* results in Table XI shows that our approach achieves substantially better accuracy as compared to a random selection method for each history period considered (approximately thrice accurate). None of the six features had the correct developer recommended from the random selection sets (i.e., a 0% accuracy). We realize that comparing our approach with a random method is probably not sufficient or a realistic depiction; however, we believe that it is a reasonable litmus

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table XI. Summary of developer recommendation average accuracies obtained using automatic queries for all systems at the File, Package (Pkg), System (Sys), and Overall (Ovl) levels. The average accuracy values are obtained from the accuracy values of all of the bugs evaluated in the particular system. The average ranks for the first relevant recommendation on file (R_f), package (R_p), system (R_s) and overall (R_o) levels are provided.

System	xFinder: Max Num of Dev	Num of Bugs	Average Accuracy (%)				Average Ranking Effectiveness			
			File	Pkg	Sys	Ovl	R_f	R_p	R_s	R_o
KOffice (17 days history)	10	18	1.1	12.8	16.7	22.2	3	3.5	1	3.1
KOffice (1 month history)	10	18	2.8	25.6	33.3	42.2	6.5	2.9	1	2.1
KOffice (2 months history)	10	18	4.4	36.7	55.6	61.1	6	2.4	1	1.3
KOffice (4 months history)	10	18	5.0	39.4	55.6	62.8	6	2.6	1	1.6
KOffice (All history)	10	18	15.6	55.0	72.2	80.0	5.1	2	1	1.3
Eclipse 2.0	10	14	13.6	34.3	57.1	75.0	3.5	2.9	1	1.4
Eclipse 3.0	10	14	15.7	27.1	35.7	47.1	3.4	2.7	1	1.8
Eclipse 3.3.2	10	14	27.9	36.4	71.4	82.1	2.9	2.9	1	1.6
ArgoUML 0.26.2	5	15	23.3	59.3	20.0	64.7	3.4	1.7	1	1.7
ArgoUML 0.26.2	10	23	17.4	75.2	69.1	95.7	3.6	1.3	1.4	1

test. After all, this test provides an answer to why need a sophisticated method when a random works equally well or even better?

To answer question RQ4.2, we first collected the maintainers listed for every *KOffice* application[§§]. In order to get the 'best' accuracy, we did not restrict the recommendations to only the maintainer of the application (e.g., *KWord*) in which a given bug was reported (and fixed). We considered all the maintainers. That is, any maintainer is equally likely to fix any given bug in any application of *KOffice*. The setup was similar to that of the first question above except that now the maintainers were recommended as potential bug fixers (and not the randomly selected developers). A bug was considered to have an accurate recommendation, if the correct developer, i.e., the one who ended by fixing the bug, was one of the recommended maintainers. A request-level accuracy of 33% (six correctly recommended/18 total bugs) was obtained from this maintainer model. Comparing this result with the request-level accuracies in Section 3.3 shows that our approach provides substantially better accuracy than a random selection method (approximately twice as accurate). We observed similar results for overall expertise-level granularity accuracy. Two of the six features had the correct developer recommended from the maintainer set. Overall, our approach outperformed both randomly selected and maintainer-based methods of developer recommendations on the considered *KOffice* data set by a substantial margin.

The details of the bug and accuracy data for *ArgoUML*, *Eclipse*, and *KOffice* corresponding to Table XI are available at http://www.cs.wm.edu/semeru/data/jsme09-bugs-devs/ and excluded here for brevity.

The above discussion provides empirical answers to the research questions RQ3 and RQ4 of our case study. Thus, concludes the second half of our study.

5. THREATS TO VALIDITY

We discuss some threats that may affect the construct, internal, external validities, and reliability of the approach and the case study conducted for its evaluation.

[§§] http://wiki.koffice.org/index.php?title=Junior_Jobs.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

5.1. Construct validity

We discuss threats to construct validity that concerns the means that are used in our method and its accuracy assessment as a depiction of reality. In other words, do the accuracy measures and their operational computation represent correctness of developer recommendations?

Our developer recommendation method does not entirely reflect the true reality of real systems:

We realize that the activity or process of who resolves the change requests, and histories of even simple bugs, is dependent on social, organizational, and technical knowledge that cannot be automatically extracted from solely software repositories [36]. In our case studies we had access only to the qualitative and quantitative data from the bug tracking and revision control systems. We did contact some of the developers (contact persons for components or specific project parts listed on the project web sites) of the three systems considered in our study to gain an understanding of their current practices and potential benefits of our method. We requested them to respond to a questionnaire we prepared (see, Appendix A). We received only a single response in which it was stated that the current practice of change requests to developer matching was largely manual. The response also stated that an automatic method that could save time and is not a deterrent would be useful. Also, it should be noted that the bug history databases have been used in the literature for validating the accuracy of developer recommendations. For example, using the developer-related information in the 'fixed' bug report to compare with the recommended developers by a given approach [37]. In the sense, the accuracy assessment procedure that we used in our study is not uncommon.

Concept location may not find source code exactly relevant to the bug or feature:

In a few cases, the concept location tools did not exactly return the classes (files) that were found in the commits related to the bug fixes or feature implementations. However, it is interesting to note from the accuracy results that the classes that were recommended were either relevant (but not involved in the change that resolved the issue) or conceptually related (i.e., developers were also knowledgeable in these parts). This point is elaborated in Section 4.1.

Accuracy measures may not precisely measure the correctness of developer recommendations:

A valid concern could be a single measure of accuracy may not provide a comprehensive picture, i.e., an incomplete and monolithic view of accuracy from the considered data set. To mitigate this concern, we defined three different viewpoints of accuracy that assess the core components of our method and analyzed the data from the prospective of each of these viewpoints on the studied systems. The three corresponding measures are discussed at length in Sections 3.3–3.5.

5.2. Internal validity

We discuss threats to internal validity that concerns with factors that could have influenced our results.

Factors other than expertise are responsible for the developers ending up in resolving the change requests:

In our case study, we showed that there is a positive relationship between the developers recommended with our approach to work on change requests and the developers entered as the ones who fixed them in the software repositories (i.e., considered baseline). Therefore, the basic premise of our approach, i.e., relevant source code with LSI and developer's contributions to it in the form of past changes, was found to hold well. However, we do not claim that past contributions or expertise alone is a necessary and sufficient condition to correctly assigning developers to change requests. At best, our results allude to a strong correlation between the recommendations and the baseline, and not causality. It is possible that other factors, such as schedule, work habits, technology fade or expertise, and project policy/roles are equally effective or better. A definitive answer in this regard would require another set of studies.

Developer Identities could have influenced the accuracy results:

xFinder uses the committer ID, which represents the developer's identity. We do not exactly know from the repository data who changed the file, but only who committed. Also, if the developer has more than one ID [38], the accuracy of the result will be affected. In our evaluation, we came

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

across several cases where developers had one form of identification in the *Bugzilla* repository (names or email addresses) and another (username) in the source code repository, and even a manual examination could not conclusively assert that they were the same developers. It should be noted that such cases were discarded from our accuracy computation. This identity issue is one of the main reasons why we did not report the accuracy for all the bug reports submitted and only considered samples during the evaluation periods in our assessment study.

Specific query formulation could have influenced the accuracy results:

It should also be noted that the LSI queries to retrieve initial files used in the study to compare manual and automatic queries were formulated by one of the authors. While the automatic queries were solely based on the words from the actual change requests, a different choice of words for the manual queries could have produced a different set of ranking results. Nonetheless, automatic queries provide a reliable, realistic, non-biased assessment on the lower bound of possible accuracy measure results. For example, in the true spirit of the open-source development model, one cannot and should not police as to what is being reported and risk discouraging project participation and success. There are bound to be variations in the textual description due to the diverse backgrounds and skills of project participants.

User-specified parameters could influence the accuracy results:

Another potential issue is the variation in the accuracy with the change in the number of relevant classes and maximum number of developer recommendations, both of which are user specified. We found that the values used in our evaluation were sufficient to provide a reasonable level of accuracy. In fact, the average number of developers recommended at the file and package levels in our evaluated systems was well within the specified maximum limits, which suggests that the value ten is a reasonable upper bound.

History periods could influence the accuracy results:

Our tool needs source code version history in order to give recommendations. If there is no 'good' portion of development history, it will most likely not be able to function with a high accuracy (or in the worst case provide no recommendation, e.g., a new developer to a project making the first contribution to a bug fix). The accuracy of the recommended list seems to improve with an increase in the training set size (also observed in our conducted evaluation results); however, not to a conclusively significant limit. This could be attributed to the fact that when open source projects evolve, their communities also evolve [39, 40], so the relationship between the length of the historical period of time and the accuracy of the recommendation is not very succinct and decisive. In other related studies [41, 42] in the mining software repositories community, it was observed that recent history was a 'better' predictor of the future.

5.3. External validity

We discuss threats to external validity that concerns with factors that are associated with generalizing the validity of our results to data sets other than considered in our study.

Assessed systems are not representative:

The accuracy was assessed on three open-source systems, which we believe are good representatives of large-scale, collaboratively developed software systems. However, we cannot claim that the results presented here would equally hold on other systems (e.g., closed source).

Sampled sets of change requests are not sufficient:

The evaluation was performed on randomly chosen bug reports, features, and even a refactoring change that fit our sampling criteria. While the bug reports and feature requests used are representatives of the considered software systems used in the evaluation (we picked them from more than one release, history period), similar studies on other systems are necessary to confirm that conclusions would hold in general.

The size of the evaluation sample and the number of systems remain a difficult issue, as there is no accepted 'gold standard' for the problem of the developer recommendation problem, i.e., how many change requests and systems are considered to be a sufficient evaluation data set? The approach of 'more, the better' may not necessarily yield a rigorous evaluation, as they are known issues of bug duplication [43] and other noisy information in bug/issue databases. Not accounting

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

for such issues may lead to biased results positively or negatively or both. The considered sample sizes in our evaluation, however, are not uncommon, for example, Anvik *et al.* [1] also considered 22 bug reports from Firefox in their evaluation. Nonetheless, this topic remains an important part of our future work.

Accuracy offered by our method may not be practical:

We compared the accuracy results of our approach with two obvious null models that use a random set of developers and a maintainer list available in project documentation. But we certainly do not claim that these two models define the gold standard for comparison. We plan to pursue avenues such as a case study on the use of our approach in the actual triage process of the considered open-source projects and the actual developers' feedback (on arguably non-trivial tasks).

5.4. Reliability

We discuss threats that would risk the replication of our evaluation study.

Data set not available: One of the main difficulties in conducting empirical studies is the access (or lack of it) to the data set of interest. In our study, we used open-source data sets that are publicly available. Also, we detailed the specifics of change requests that we used. The details of the bug and accuracy data for *ArgoUML*, *Eclipse*, and *KOffice* are available at http://www.cs.wm.edu/semeru/data/jsme09-bugs-devs/. We also provide the appropriate context to our study, e.g., parameters used for *xFinder* and concept location tools. Therefore, our case studies can be reliably reproduced by other researchers.

Evaluation protocol not available:

A concern could be that the lack of sufficient information about the evaluation procedure and protocol may limit the replicability of the study. We believe that our accuracy measures along with the evaluation procedure are sufficiently documented to enable replication on the same or even different data sets.

Instruments not available:

Lack of access to the instruments and tools used in the study could limit the replication of the study. The concept location tool uses LSI-based algorithm, which is available in many statistical open-source packages, and *xFinder* tool uses well-documented mining methods, which should allow for easy implementation and/or access. We are working on making an open-source version of our tool chain available in the near future.

6. RELATED WORK

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

6.1. Concept location

Wilde *et al.* [44] were the first to address the problem of feature location using the Software Reconnaissance method, which uses dynamic information. Several researchers recently revisited this approach with the goal of improving its accuracy and provided new methods for analyzing execution traces [45] and selecting execution scenarios [46]. Biggerstaff *et al.* [10] introduced the problem of concept assignment in the context of static analysis. They extracted identifiers from the source code and clustered them to support the identification of concepts. The most straightforward and commonly used static technique for searching the source code is based on regular expression matching tools, such as the Unix utility *grep*. IR techniques [4, 28, 35, 47] bring a significant improvement over regular expression matching and related techniques, and allow more general queries and rank the results to these queries. Latent Dirichlet Allocation and independent component analysis have been recently applied to locate concepts [48] and bugs [49], categorize software systems [50], and measure software cohesion [51]. Natural language processing techniques have been recently utilized to augment concept location approaches [29, 52].

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Chen *et al*. [53] proposed a static-based technique for concept location that is based on the search of abstract system dependence graphs. This approach has been recently improved by Robillard [54]. Zhao *et al.* [55] proposed a technique that combines IR with branch-reserving call-graph information to automatically assign features to respective elements in the source code. Gold *et al.* [56] proposed an approach for binding concepts with overlapping boundaries to the source code, which is formulated as a search problem and uses genetic and hill climbing algorithms.

Eisenbarth *et al.* [57] combined static dependencies and dynamic execution traces to identify features in programs and used Formal Concept Analysis (FCA) to relate features together. Salah *et al.* [58, 59] use static and dynamic data to identify feature interaction in Java source code. Kothari *et al.* [60] use dynamic and static analyses for identifying canonical feature sets in software systems. Greevy *et al.* used dynamic analysis to analyze evolving features throughout software evolution [61]. Hill *et al.* [62] combined static and textual information to expedite traversal of program dependence graphs for impact analysis. Poshyvanyk *et al.* [11, 34] combined an IR-based technique with a scenario-based probabilistic ranking of the execution traces to improve the precision of feature location. Eaddy *et al.* [63] combined static, dynamic, and textual analyses to trace requirements (concepts) to target source code elements.

A comparison of different approaches for feature location is presented in [64]. Summaries of static, dynamic, and other approaches are available in [34, 45, 65, 66], while an overview of industrial feature location tools is available in [67]. To the best of our knowledge, no other work besides ours [2] has applied concept location techniques to the problem of expert developer recommendation.

6.2. Developer contributions and recommendation

McDonald and Ackerman [68] developed a heuristic-based recommendation system called the *Expertise Recommender* (ER) to identify experts at the module level. Developers are ranked according to the most recent modification date. When there are multiple modules, people who touched all the modules are considered. Vector-based similarity is also used to identify technical support. For each request, three query vectors (symptoms, customers, and modules) are created. These vectors are then compared with the person's profile. This approach depends on user profiles that need to be explicitly collected upfront. This approach has been designed for specific organizations and not tested on open-source projects.

Mino and Murphy [69] produced a tool called *Emergent Expertise Locator* (EEL). Their work is adopted from a framework to compute the coordination requirements between developers given by Cataldo *et al.* [70]. EEL helps to find the developers who can assist in solving a particular problem. The approach is based on mining the history of how files have changed together and who has participated in the change. In our approach, we also include the activities, i.e., days on which they contributed changes, of the developers and identify experts at the package and system levels, and not only at the file level.

Expertise Browser (ExB) [71] is another tool to locate people with the desired expertise. The elementary unit of experience is the Experience Atom (EA). The number of these EAs in a specific domain measures the developer experience. The smallest EA is a code change that has been made on a specific file. In our approach, the number of EAs corresponds to the commit contributions. Again, we included more than one parameter in determining file experts. We also used two different measures to identify experts: one measure for file experts and another for package and system experts.

Anvik and Murphy [72] did an empirical evaluation of two approaches to locate expertise. As developers work on a specific part of the software, they accumulate expertise. They term this expertise as *implementation expertise*. The two approaches are based on mining the source and bug repositories. The first approach examines the check-in logs for modules that contain the fixed source files. Recently, active developers who did the changes are selected and filtered. In the second approach, the bug reports from bug repositories are examined. The developers are selected from the CC lists, the comments, and those who fixed the bug. They found that both approaches have

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

relative strengths in different ways. In the first approach, the most recent activity date is used to select developers. This study focuses on identifying experts to fix bugs or to deal with bug reports.

A machine-learning technique is used to automatically assign a bug report to the right developer who can resolve it [1]. The classifier obtained from the machine-learning technique analyzes the textual contents of the report and recommends a list of developers. Another text-based approach is used to build a graph model called *ExpertiseNet* for expertise modeling [73]. A recent approach to improve bug triaging uses graph-based model based on Markov chains, which capture bug reassignment history [74]. Our approach uses expertise measures that are computed in a straightforward manner from the commits in source code repositories and does not employ any machine learning like techniques. In another recent work, Matter *et al.* [75] used the similarity of textual terms between source code changes (i.e., word frequencies of the *diff* given changes from source code repositories) and the given bug report to assign developers. Their approach does not require indexing of past bug reports, one of the rare ones similar to ours; however, it is purely text based. Our approach does not use textual analysis of source code changes and is based on a number of non-text-based contribution measures.

There are also works on using MSR techniques to study and analyze developer contributions. German [76] described in his report some characteristics of the development team of *PostgreSQL*. He found that in the last years only two persons were responsible for most of the source code. Tsunoda *et al.* [77] analyzed the developers' working time of open-source software. The e-mail sent time was used to identify developers' working time. Bird *et al.* [78] mined e-mail archives to analyze the communication and co-ordination activities of the participants. Del Rosso [79] used collaborations and interactions between knowledge-intensive software developers to build a social network. By analyzing this network, he tries to understand the meaning and implications to software and software development. Some implications are locating developers with a wide expertise on the project and determining where the expertise is concentrated in the software development team. Ma *et al.* [80] proposed an approach for identifying developers using implementation expertise (i.e., using functionality by calling API methods). Yu and Ramaswamy [81] mined CVS repositories to identify developer roles (core and associate). The interaction between authors is used as clustering criteria. The KLOC and the number of revisions are used to study the development effort for the two groups. Weissgerber *et al.* [82] 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 [83] studied the modification records (MRs) of CVS logs to visualize who are the people who tend to modify certain files. Fischer *et al.* [84] analyzed and related bug report data for tracking features in software.

In summary, to the best of our knowledge, no other work besides ours has used a combination of a concept location and mining software repositories techniques to address the problem of assigning expert developers to change requests. Also, our approach does not need to mine past change requests (e.g., history of similar bug reports to resolve the bug request in question), but does require source code change history. The single-version source code analysis with IR (and not the past reports in the issue repositories) is employed to reduce the mining space of the source code change history of only selective entities.

7. CONCLUSIONS AND FUTURE WORK

The main contribution of our work is the first use of a concept location technique combined with a technique based on MSR for the expert developer recommendation task. While both these techniques have been investigated and used independently before, their combined use for tasks such as the one studied here has not been systematically investigated. We showed the application of a concept location technique beyond merely concept or feature location in source code. Also, our work provides an interesting horizon to bring together single-version analysis of traditional software engineering (i.e., concept location) with multi-version analysis based on mining software repositories.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

The results of our systematic evaluation on *KOffice*, *Eclipse* and *ArgoUML* indicate that our approach can identify relevant developers to work on change requests with fairly high accuracy and in an effective ranked order. We recorded the highest overall accuracies of 95%, 82%, and 80% in *ArgoUML*, *Eclipse*, and *KOffice* when all the prior commit histories were considered (and the lowest of 22% overall accuracy with only about two weeks of commit history in *KOffice*). These results are comparable to other approaches in the literature. For example, Anvik *et al.* [1] reported the precision of their approach as 57% and 64% on *Eclipse* and *Firefox* systems (although a different experiment setup and execution). At the very least, the presented approach did outperform two straightforward first choices that maybe as readily available to recommend developers to work on change requests. Our approach required mining histories of only between top three and five ranked files relevant to a concept to get the first accurate developer recommendations on the evaluated systems. We make our evaluation data publicly available (see online appendix[¶¶] and hope that these data, including the specific bugs used, will provide the first steps toward creating a benchmark for evaluating developer recommendation approaches. Also, we show the value of the package and system levels of expertise considered by *xFinder* in developer recommendations. We believe that our approach has merits in time, effort, and quality improvements when dealing with change requests during software maintenance (a rigorous validation of which would require a field case study, and is a subject of future work).

Our immediate future work includes assessing the accuracy impact of analyzing developer expertise at finer granularity than system, package, and file levels (e.g., class and method). We believe that this work would result in improved accuracy of recommendations. Another area of investigation is the use of other similarity measures for vectors (e.g., *Cosine* or *Manhattan* distance) for computing the *xFactor* values and their impact on the recommendation accuracy. We are also investigating ways of integrating existing approaches [85–87] for automatically classifying change requests based on their types into our approach. We are also evaluating our approach on other open-source systems such as *Apache httpd* and *jEdit* and conducting empirical studies comparing of our approach with other *pure* history-based approaches. We are also developing open-source plug-ins for popular environments such as *Eclipse* and *KDevelop*. Finally, we are planning on integrating existing approaches for identifying duplicate bug reports [43] for effectively triaging similar bug reports that have been previously addressed.

APPENDIX A

Questions posed to the developers of *ArgoUML*, *Eclipse*, and *KOffice*:

(1) What is the current practice/process of allocating bug reports/feature requests to developers?
(2) Is the current practice/process mostly manual or involves automatic tool support?
(3) What are the specific manual and automatic parts of the process?
(4) What are the criteria, if any, used in the current process?
(5) Would it be potentially useful to your project or contributing developers in your opinion to have a tool that automatically identifies and favors developers who have previously contributed source code related to a bug request (or a feature request) in question to work on?
(6) Would it be potentially useful to your project or contributing developers in your opinion to have a tool that favors developers who have previously worked on similar bugs (or features) to a bug request (or a feature request) in question to work on?
(7) Do you have any comments or suggestions or advise about our work that you would like to share?

[¶¶] http://www.cs.wm.edu/semeru/data/jsme09-bugs-devs/ (created and verified on 10/21/09)).

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

ACKNOWLEDGEMENTS

This work is supported by NSF CCF-1016868, NSF CCF-1063253, and University of Missouri Research Board grants. Any opinions, findings, and conclusions expressed herein are of the authors' and do not necessarily reflect those of the sponsors.

We would like to thank anonymous reviewers for their helpful comments on improving earlier versions of this paper.

REFERENCES

1. Anvik J, Hiew L, Murphy GC. Who should fix this bug? *The 28th IEEE/ACM International Conference on Software Engineering* (*ICSE '06*), 2006; 361–370.
2. Kagdi H, Poshyvanyk D. Who can help me with this change request? *The 17th IEEE International Conference on Program Comprehension* (*ICPC'09*), 2009; 273–277.
3. Deerwester S, Dumais ST, Furnas GW, Landauer TK, Harshman R. Indexing by latent semantic analysis. *Journal of the American Society for Information Science* 1990; **41**:391–407. DOI: Electronic Resource Number.
4. Marcus A, Sergeyev A, Rajlich V, Maletic J. An information retrieval approach to concept location in source code. *The 11th IEEE Working Conference on Reverse Engineering* (*WCRE'04*), 2004; 214–223.
5. Kagdi H, Hammad M, Maletic JI. Who can help me with this source code change? *IEEE International Conference on Software Maintenance* (*ICSM'08*), 2008; 157–166.
6. Kagdi H, Collard ML, Maletic JI. 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* (*JSME*) 2007; **19**(2):77–131.
7. Canfora G, Cerulo L. Impact analysis by mining software and change request repositories. *The 11th IEEE International Symposium on Software Metrics* (*METRICS'05*), 2005; 20–29.
8. Canfora G, Cerulo L. Fine grained indexing of software repositories to support impact analysis. *International Workshop on Mining Software Repositories* (*MSR'06*), 2006; 105–111.
9. Canfora G, Cerulo L. Supporting change request assignment in open source development. *ACM Symposium on Applied Computing* (*SAC'06*), 2006; 1767–1772.
10. Biggerstaff TJ, Mitbander BG, Webster DE. The concept assignment problem in program understanding. *The 15th IEEE/ACM International Conference on Software Engineering* (*ICSE'94*), 1994; 482–498.
11. Poshyvanyk D, Guéhéneuc YG, Marcus A, Antoniol G, Rajlich V. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering* 2007; **33**(6):420–432.
12. Ye Y, Fischer G. Reuse-conducive development environments. *Journal Automated Software Engineering* 2005; **12**(2):199–235.
13. Maarek YS, Berry DM, Kaiser GE. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering* 1991; **17**(8):800–813.
14. Maletic JI, Marcus A. Supporting program comprehension using semantic and structural information. *The 23rd International Conference on Software Engineering* (*ICSE'01*). IEEE: Toronto, ON Canada, 2001; 103–112.
15. Marcus A, Maletic JI. Identification of high-level concept clones in source code. *Automated Software Engineering* (*ASE'01*), 2001; 107–114.
16. Tairas R, Gray J. An information retrieval process to aid in the analysis of code clones. *Empirical Software Engineering* 2009; **14**(1):33–56.
17. Antoniol G, Canfora G, Casazza G, De Lucia A, Merlo E. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering* 2002; **28**(10):970–983.
18. Marcus A, Maletic JI, Sergeyev A. Recovery of traceability links between software documentation and source code. *International Journal of Software Engineering and Knowledge Engineering* 2005; **15**(4):811–836.
19. De Lucia A, Fasano F, Oliveto R, Tortora G. Recovering traceability links in software artefact management systems. *ACM Transactions on Software Engineering and Methodology* 2007; **16**(4):13(1)–13(50). DOI: 10.1145/1276933.1276934.
20. Jiang H, Nguyen T, Che IX, Jaygarl H, Chang C. Incremental latent semantic indexing for effective, automatic traceability link evolution management. *The 23rd IEEE/ACM International Conference on Automated Software Engineering* (*ASE'08*), 2008.
21. Kuhn A, Ducasse S, Gîrba T. Semantic clustering: Identifying topics in source code. *Information and Software Technology* 2007; **49**(3):230–243.
22. Kawaguchi S, Garg PK, Matsushita M, Inoue K. Automatic categorization algorithm for evolvable software archive. *The 6th International Workshop on Principles of Software Evolution* (*IWPSE'03*), 2003; 195–200.
23. Marcus A, Poshyvanyk D, Ferenc R. Using the conceptual cohesion of classes for fault prediction in object oriented systems. *IEEE Transactions on Software Engineering* 2008; **34**(2):287–300.
24. Poshyvanyk D, Marcus A, Ferenc R, Gyimóthy T. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering* 2009; **14**(1):5–32.
25. Cubranic D, Murphy GC, Singer J, Booth KS. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering* 2005; **31**(6):446–465.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

26. Hayes JH, Dekhtyar A, Sundaram SK. Advancing candidate link generation for requirements tracing: the study of methods. *IEEE Transactions on Software Engineering* 2006; **32**(1):4–19.
27. Lormans M, Van Deursen A. Can LSI help reconstructing requirements traceability in design and test? *The 10th European Conference on Software Maintenance and Reengineering* (*CSMR'06*), 2006; 47–56.
28. Poshyvanyk D, Marcus D. Combining formal concept analysis with information retrieval for concept location in source code. *The 15th IEEE International Conference on Program Comprehension* (*ICPC'07*), 2007; 37–48.
29. Shepherd D, Fry Z, Gibson E, Pollock L, Vijay-Shanker K. Using natural language program analysis to locate and understand action-oriented concerns. *International Conference on Aspect Oriented Software Development* (*AOSD'07*), 2007; 212–224.
30. Collard ML, Kagdi HH, Maletic JI. An XML-based lightweight C++ fact extractor. *The 11th IEEE International Workshop on Program Comprehension* (*IWPC'03*). IEEE-CS: Portland OR, 2003; 134–143.
31. Ferenc R, Beszedes A, Gyimóthy T. Extracting facts with columbus from C++ code. *The 8th European Conference on Software Maintenance and Reengineering* (*CSMR 2004*), 2004; 4–8.
32. Runeson P, Höst M. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 2009; **14**(2):131–164.
33. Alali A, Kagdi H, Maletic JI. What's a typical commit? A characterization of open source software repositories. *The 16th EEE International Conference on Program Comprehension* (*ICPC'08*), 2008; 182–191.
34. Liu D, Marcus A, Poshyvanyk D, Rajlich V. Feature location via information retrieval based filtering of a single scenario execution trace. *The 22nd IEEE/ACM International Conference on Automated Software Engineering* (*ASE'07*), 2007; 234–243.
35. Gay G, Haiduc S, Marcus M, Menzies T. On the use of relevance feedback in IR-based concept location. *The 25th IEEE International Conference on Software Maintenance* (*ICSM'09*), 2009.
36. Aranda J, Venolia G. The secret life of bugs: Going past the errors and omissions in software repositories. *The 31st IEEE/ACM International Conference on Software Engineering* (*ICSE'09*), 2009; 298–308.
37. Anvik J, Hiew L, Murphy GC. Who should fix this bug? *The 28th International Conference on Software Engineering* (*ICSE'06*), 2006; 361–370.
38. Robles G, Gonzalez-Barahona JM. Developer identification methods for integrated data from various sources. *2005 International Workshop on Mining Software Repositories* (*MSR'05*), 2005; 1–5.
39. Nakakoji K, Yamamoto Y, Nishinaka Y, Kishida K, Ye Y. Evolution patterns of open-source software systems and communities. *International Workshop on Principles of Software Evolution* (*IWPSE'02*), 2002; 76–85.
40. Bird C, Pattison DS, D'Souza RM, Filkov V, Devanbu PT. Latent social structure in open source projects. *ACM SIGSOFT Symposium on Foundations of Software Engineering* (*FSE'08*), 2008; 24–35.
41. Girba T, Ducasse T, Lanza M. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. *The 20th IEEE International Conference on Software Maintenance* (*ICSM'04*), 2004; 40–49.
42. Zimmermann T, Zeller A, Weißgerber P, Diehl S. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering* 2005; **31**(6):429–445.
43. Runeson P, Alexandersson M, Nyholm O. Detection of duplicate defect reports using natural language processing. *29th IEEE/ACM International Conference on Software Engineering* (*ICSE'07*), 2007; 499–510.
44. Wilde N, Gomez JA, Gust T, Strasburg D. Locating user functionality in old code. *IEEE International Conference on Software Maintenance* (*ICSM'92*), 1992; 200–205.
45. Antoniol G, Guéhéneuc YG. Feature identification: An epidemiological metaphor. *IEEE Transactions on Software Engineering* 2006; **32**(9):627–641.
46. Eisenberg AD, De Volder K. Dynamic feature traces: Finding features in unfamiliar code. *The 21st IEEE International Conference on Software Maintenance* (*ICSM'05*), 2005; 337–346.
47. Poshyvanyk D, Petrenko M, Marcus A, Xie X, Liu D. Source code exploration with Google. *The 22nd IEEE International Conference on Software Maintenance* (*ICSM'06*), 2006; 334–338.
48. Grant S, Cordy JR, Skillicorn DB. Automated concept location using independent component analysis. *The 15th Working Conference on Reverse Engineering* (*WCRE'08*), 2008; 138–142.
49. Lukins S, Kraft N, Etzkorn L. Source code retrieval for bug location using latent dirichlet allocation. *The 15th Working Conference on Reverse Engineering* (*WCRE'08*), 2008; 155–164.
50. Tian K, Revelle M, Poshyvanyk D. Using latent dirichlet allocation for automatic categorization of software. *The 6th IEEE Working Conference on Mining Software Repositories* (*MSR'09*), 2009.
51. Liu Y, Poshyvanyk D, Ferenc R, Gyimóthy T, Chrisochoides N. Modelling class cohesion as mixtures of latent topics. *The 25th IEEE International Conference on Software Maintenance* (*ICSM'09*), 2009.
52. Hill E, Pollock L, Vijay-Shanker K. Automatically capturing source code context of NL-queries for software maintenance and reuse. *The 31st IEEE/ACM International Conference on Software Engineering* (*ICSE'09*), 2009.
53. Chen K, Rajlich V. Case study of feature location using dependence graph. *The 8th IEEE International Workshop on Program Comprehension* (*IWPC'00*), 2000; 241–249.
54. Robillard MP. Topology analysis of software dependencies. *ACM Transactions on Software Engineering and Methodology* 2008; **17**(4):18(1)–18(36). DOI: 10.1145/13487689.13487691.
55. Zhao W, Zhang L, Liu Y, Sun J, Yang F. SNIAFL: Towards a static non-interactive approach to feature location. *ACM Transactions on Software Engineering and Methodologies* (*TOSEM*) 2006; **15**(2):195–226.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

56. Gold N, Harman M, Li Z, Mahdavi K. Allowing overlapping boundaries in source code using a search based approach to concept binding. *The 22nd IEEE International Conference on Software Maintenance* (*ICSM'06*), 2006; 310–319.
57. Eisenbarth T, Koschke R, Simon D. Locating features in source code. *IEEE Transactions on Software Engineering* 2003; **29**(3):210–224. DOI: Electronic Resource Number.
58. Salah M, Mancoridis S. A hierarchy of dynamic software views: from object-interactions to feature-interactions. *The 20th IEEE International Conference on Software Maintenance* (*ICSM'04*), 2004; 72–81.
59. Salah M, Mancoridis S, Antoniol G, Di Penta M. Scenario-driven dynamic analysis for comprehending large software systems. *The 10th IEEE European Conference on Software Maintenance and Reengineering* (*CSMR'06*), 2006; 71–80.
60. Kothari J, Denton T, Mancoridis S, Shokoufandeh A. On computing the canonical features of software systems. *The 13th IEEE Working Conference on Reverse Engineering* (*WCRE'06*), 2006.
61. Greevy O, Ducasse S, Girba T. Analyzing software evolution through feature views. 2006; **18**(6):425–456. DOI: Electronic Resource Number.
62. Hill E, Pollock L, Vijay-Shanker K. Exploring the neighborhood with Dora to expedite software maintenance. *22nd IEEE/ACM International Conference on Automated Software Engineering* (*ASE'07*), 2007; 14–23.
63. Eaddy M, Aho AV, Antoniol G, Guéhéneuc YG. CERBERUS: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. *The 17th IEEE International Conference on Program Comprehension* (*ICPC'08*), 2008.
64. Revelle M, Poshyvanyk D. An exploratory study on assessing feature location techniques. *The 17th IEEE International Conference on Program Comprehension* (*ICPC'09*), 2009.
65. Marcus A, Rajlich V, Buchta J, Petrenko M, Sergeyev A. Static techniques for concept location in object-oriented code. *The 13th IEEE International Workshop on Program Comprehension* (*IWPC'05*), 2005; 33–42.
66. Cleary B, Exton C, Buckley J, English M. An empirical analysis of information retrieval based concept location techniques in software comprehension. *Empirical Software Engineering* 2009; **14**(1):93–130.
67. Simmons S, Edwards D, Wilde N, Homan J, Groble M. Industrial tools for the feature location problem: An exploratory study. *Journal of Software Maintenance*: *Research and Practice* 2006; **18**(6):457–474.
68. McDonald D, Ackerman M. Expertise recommender: A flexible recommendation system and architecture. *2000 ACM Conference on Computer Supported Cooperative Work* (*CSCW '00*), 2000; 231–240.
69. Minto S, Murphy G. Recommending emergent teams. *Fourth International Workshop on Mining Software Repositories* (*MSR '07*), 2007.
70. Cataldo M, Wagstrom P, Herbsleb J, Carley KM. Identification of coordination requirements: Implications for the design of collaboration and awareness tools. *The 20th Anniversary Conference on Computer Supported Cooperative Work* (*CSCW'06*), 2006; 353–362.
71. Mockus A, Herbsleb J. Expertise browser: a quantitative approach to identifying expertise. *The 24th International Conference on Software Engineering* (*ICSE '02*), 2002; 503–512.
72. Anvik J, Murphy G. Determining implementation expertise from Bug reports. *Fourth International Workshop on Mining Software Repositories* (*MSR'07*), 2007.
73. Song X, Tseng B, Lin C, Sun M. ExpertiseNet: Relational and evolutionary expert modeling. *The 10th International Conference on User Modeling* (*UM'5*), 2005.
74. Jeong G, Kim S, Zimmermann T. Improving bug triage with bug tossing graphs. *The 7th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (*ESEC/FSE 2009*), 2009.
75. Matter D, Kuhn A, Nierstrasz O. Assigning Bug reports using a vocabulary-based expertise model of developers. *The 6th IEEE International Working Conference on Mining Software Repositories* (*MSR'09*), 2009; 131–140.
76. German DM. A study of the contributors of PostgreSQL. *2006 International Workshop on Mining Software Repositories* (*MSR '06*), 2006; 163–164.
77. Tsunoda M, Monden A, Kakimoto T, Kamei Y, Matsumoto K-i. Analyzing OSS developers' working time using mailing lists archives. *2006 International Workshop on Mining Software Repositories* (*MSR '06*), 2006; 181–182.
78. Bird C, Gourley A, Devanbu P, Gertz M, Swaminathan A. Mining Email social networks. *2006 International Workshop on Mining Software Repositories* (*MSR '06*), 2006; 137–143.
79. Del Rosso C. Comprehend and analyze knowledge networks to improve software evolution. *Journal of Software Maintenance and Evolution*: *Research and Practice* (*JSME*) 2009; **21**(3):189–215.
80. Ma D, Schuler D, Zimmermann T, Sillito J. Expertise recommendation with usage expertise. *The 25th IEEE International Conference on Software Maintenance* (*ICSM'09*), 2009.
81. Yu L, Ramaswamy S. Mining CVS repositories to understand open-source project developer roles. *Fourth International Workshop on Mining Software Repositories* (*MSR'07*), 2007.
82. Weissgerber P, Pohl M, Burch M. Visual data mining in software archives to detect how developers work together. *Fourth International Workshop on Mining Software Repositories* (*MSR'07*), 2007.
83. German DM. An empirical study of fine-grained software modifications. *Empirical Software Engineering* 2006; **11**(3):369–393.
84. Fischer M, Pinzger M, Gall H. Analyzing and relating bug report data for feature tracking. *IEEE Working Conference on Reverse Engineering* (*WCRE'03*), 2003; 90–101.
85. Antoniol G, Ayari K, Di Penta M, Khomh F, Guéhéneuc Y-G. Is it a bug or an enhancement? A text-based approach to classify change requests. *CASCON'08*, 2008.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

86. Di Lucca GA, Di Penta M, Gradara S. An approach to classify software maintenance requests. *IEEE International Conference on Software Maintenance* (*ICSM'02*), 2002; 93–102.
87. Mockus A, Votta LG. Identifying reasons for software changes using historic databases. *IEEE International Conference on Software Maintenance* (*ICSM'00*), 2000; 120–130.

AUTHORS' BIOGRAPHIES

Huzefa Kagdi is an Assistant Professor in the Department of Computer Science at Winston-Salem State University in North Carolina, U.S.A. He received the PhD and MS in Computer Science from Kent State University, U.S.A. and the BE in Computer Engineering from Birla Vishwakarma Mahavidyalaya, India. His research interests are in Software engineering: software evolution/maintenance, mining software repositories, empirical software engineering, source code analysis, and software visualization.

Malcom Gethers is a PhD Candidate in the Computer Science Department at William and Mary where he is a member of the SEMERU research group. He is advised by Dr Denys Poshyvanyk. Malcom obtained his BS from High Point University and his MS from the University of North Carolina at Greensboro. His research interests include software engineering, software maintenance and evolution, mining of software repositories, feature location, software measurement, and traceability link recovery and management. He is a student member of the IEEE and ACM.

Denys Poshyvanyk is an Assistant Professor at the College of William and Mary in Virginia. He received his PhD degree in Computer Science from Wayne State University in 2008. He also obtained his MS and MA degrees in Computer Science from the National University of Kyiv-Mohyla Academy, Ukraine and Wayne State University in 2003 and 2006, respectively. His research interests are in software engineering, software maintenance and evolution, program comprehension, reverse engineering, software repository mining, source code analysis and metrics. He is a member of the IEEE and ACM.

Maen Hammad completed his PhD at Kent State University in the Department of Computer Science in May 2010 and is now Assistant Professor at Hashemite University, Jordan. He received his Master's in Computer Science from Al-Yarmouk University—Jordan and his BS in Computer Science from the Hashemite University—Jordan. His research interest is software engineering with focus on software evolution, program comprehension and mining software repositories.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr