Information Integration for Software Maintenance and Evolution

Malcom Bernard Gethers II

Elizabeth City, North Carolina

Master of Science, University of North Carolina at Greensboro, 2007
Bachelor of Science, High Point University, 2005

A Dissertation presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

The College of William and Mary
August, 2012

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosphy

Malcom Bernard Gethers II

Approved by the Committee, July, 2012

Committee Chair
Assistant Professor Denys Poshyvanyk, Computer Science
The College of William and Mary

Associate Professor Peter Kemper, Computer Science
The College of William and Mary

Associate Professor Xipeng Shen, Computer Science
The College of William and Mary

Associate Professor Andrian Marcus, Computer Science
Wayne State University

Assistant Professor Huzefa Kagdi, Electrical Engineering and Computer Science
Wichita State University

ABSTRACT PAGE

Software maintenance and evolution is a particularly complex phenomenon in the case of long-lived, large-scale systems. It is not uncommon for such systems to progress through years of development history, a number of developers, and a multitude of software artifacts including millions of lines of code. Therefore, realizing even the slightest change may not always be straightforward. Clearly, changes are the central force driving software evolution. Therefore, it is not surprising that a significant effort has been (and should be) devoted in the software engineering community to systematically understanding, estimating, and managing changes to software artifacts. This effort includes the three core change related tasks of (1) expert developer recommendations - identifying who are the most experienced developers to implement needed changes, (2) traceability link recovery recovering dependencies (traceability links) between different types of software artifacts, and (3) software change impact analysis - which other software entities should be changed given a starting point.

This dissertation defines a framework for an integrated approach to support three core software maintenance and evolution tasks: expert developer recommendation, traceability link recovery, and software change impact analysis. The framework is centered on the use of conceptual and evolutionary relationships latent in structured and unstructured software artifacts. Information Retrieval (IR) and Mining Software Repositories (MSR) based techniques are used for analyzing and deriving these relationships. All the three tasks are supported under the framework by providing systematic combinations of MSR and IR analyses on single and multiple versions of a software system. Our approach to the integration of information is what sets it apart from previously reported relevant solutions in the literature. Evaluation on a number of open source systems suggests that such combinations do offer improvements over individual approaches.

Table of Contents

To my family and friends.

ACKNOWLEDGMENTS

I would first like to sincerely thank my advisor, Denys Poshyvanyk for his guidance during my doctoral studies. I would also like to thank my dissertation committee members, Huzefa Kagdi, Peter Kemper, Andrian Marcus, and Xipeng Shen. I am also grateful to all faculty and staff in the Computer Science department for cultivating a supportive environment.

I am grateful to Gabriele Bavota, Michael Collard, Andrea De Lucia, Massimiliano Di Penta, Maen Hammad, Huzefa Kagdi, and Rocco Oliveto for strong colloborations. Furthermore, I express my gratitude to all the members of the SEMERU group, namely Bogdan Dit, Daniel Graham, Mario Linares Vasquez, Collin McMillan, Evan Moritz, Derrin Pierret, Meghan Revelle, and Trevor Savage.

I would like to thank all my family and friends who supported me and took time out to review this work. I would especially like to thank my friend and love Laura for all of her support and encouragement throughout the years and my wonderful son Malachi.

The work described in dissertation was partially funded by the grants CCF-1016868 and CNS-0959924 from the U.S. National Science Foundation, Virginia Space Grant Consortium, and Southern Regional Education Board (SREB). Any opinions, findings and conclusions expressed herein are those of the author and do not necessarily reflect those of the sponsors.

List of Tables

List of Figures

Information Integration for Software Maintenance and Evolution

Chapter 1

Supporting Software Evolution

Software maintenance and evolution represents a phase in the software development life cycle, which accounts for approximately 50%-60% of time and money devoted towards developing software systems [21]. This dissertation presents techniques which alleviate the time and effort required by developers to perform software maintenance tasks. More specifically, we present techniques to assist software engineers when they address crucial software maintenance tasks, including (1) developer recommendation (DR) (2) traceability link recovery (TR) and (3) impact analysis (IA). The underlying theme of this dissertation is to leverage various sources of information and analysis techniques to reduce the effort required by software developers and project managers when faced with these duties. Through this research we advance the state-of-the-art in three important areas of software maintenance. Ideally, the time and effort required of developers should be minimal, and in its reduction organizations will see significant decrease in the expense of developing software systems.

In this dissertation, we introduce an approach for the analysis of software that integrates information and analysis techniques to directly support the core software maintenance tasks DR, TR, and IA. Conceptual information captures the extent to which domain concepts and software artifacts are related to each other. This information is derived using Information

Retrieval (IR) based analysis of textual software artifacts that are not limited to a single version of software (*e.g.*, comments and identifiers in a single snapshot of source code), but also across versions (*e.g.*, change logs and bug reports in the change history). Evolutionary information is derived from analyzing relationships and relevant information observed from past changes by utilizing Mining Software Repositories (MSR) techniques. Central to our approach are the information sources that are developer/human centric (*e.g.*, comments and identifiers, and commit practices), rather than (formal)language/artifact centric (*e.g.*, static and dynamic dependencies such as call graphs).

The core research philosophy is that the integration of information sources and analysis techniques will enhance our ability to perform crucial software maintenance tasks. For example, existing IA techniques have utilized both single and multiple version analysis of source code independently, but their combined use has not been previously investigated. Overall, a change management solution which integrates various types of analysis and not only helps with identifying expert developers, but also the extent the change will have on the system, is currently missing. Our solution is an addresses these open issues and support software maintenance using an integrated approach.

1.1 Research Goals and Contributions

In this dissertation we take an initial step towards answering our overarching research question as to what are the exclusive and potentially synergistic benefits of integrating conceptual and evolutionary information with regards to key software maintenance tasks. While both these sources have been studied independently before, their combined use for

tasks such as the ones studied here has not been scientifically investigated. Furthermore, the combination of different analysis techniques on a single information source as also been overlooked.

1.1.1 Information Integration

The core research philosophy behind this integrated approach is that the combination of different information sources and types of analysis overcomes their individual limitations. In this work we consider two cases, (1) integrating information of two different sources (namely, IR and MSR) and (2) integrating different types of analysis techniques for a single source of information. Each individual information source as well as type of analysis possesses a set of strengths and limitations. In the first case, complementary sources of information are combined. More specifically, it allows for the analysis of single (present) and multiple versions (past) of software systems. In the second case, the individual analysis techniques represent experts and both experts express complementary judgments based on the conceptual relationships between source code entities. In both cases, the integration helps alleviate the limitations of the individual techniques. Our principal research hypothesis is that such integrated approaches to performing software maintenance task will help alleviate the effort required by developers and project leads.

1.1.2 Contributions

This dissertation makes the following research contributions:

- **Developer Recommendation:** We present a novel approach that integrates IR and MSR techniques to recommend expert developers to assist in the implementation of

software change requests (e.g., bug reports and feature requests). An 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 60for 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.

- **Traceability Link Recovery:** We define an integrated approach to address the issue of traceability link recovery. Different IR methods have been proposed to recover traceability links among software artifacts. Until now there is no single method that sensibly outperforms the others, however, it has been empirically shown that some methods recover different, yet complementary traceability links. In this work we exploit this empirical finding and present an integrated approach to combine orthogonal IR techniques, which have been statistically shown to produce dissimilar results. Our approach combines the following IR-based methods: Vector Space Model

(VSM), probabilistic Jensen and Shannon (JS) model, and Relational Topic Modeling (RTM), which has not been used in the context of traceability link recovery before. The empirical case study conducted on six software systems indicates that the integrated method outperforms stand-alone IR methods as well as any other combination of non-orthogonal methods with a statistically significant margin.

- **Impact Analysis:** We have developed a novel approach that integrates conceptual and evolutionary techniques to support change impact analysis in source code. Information Retrieval is used to derive conceptual couplings from the source code in a single version (release) of a software system. Evolutionary couplings are mined from source code commits. The premise is that such combined methods provide improvements to the accuracy of impact sets. A rigorous empirical assessment on the changes of the open source systems Apache httpd, ArgoUML, iBatis, and KOffice is also reported. The results show that a combination of these two techniques, across several cut points, provides statistically significant improvements in accuracy over either of the two techniques used independently. Improvements in recall values of up to 20% over the conceptual technique in KOffice and up to 45% over the evolutionary technique in iBatis were reported.

1.2 Bibliographical Notes

This dissertation contains material which was previously published. This section highlights details related to collaborations with other researchers.

The material presented in Chapter 2 is based on a collaborative research effort with

Dr. Huzefa Kagdi at Wichita State University and Maen Hammad at Hashemite University in Zarqa, Jordan. The results of the research project were originally published in the Journal of Software: Evolution and Process (*formely the International Journal of Software Maintenance and Evolution*) [79].

The idea and findings presented in Chapter 3 appeared in the proceedings of the 18th IEEE International Conference on Program Comprehension (ICPC'10) [113] and the 27th IEEE International Conference on Software Maintenance (ICSM'11) [144] and was invited to the Journal of Software Maintenance and Evolution special issue among the best papers at ICSM'11. This project was a collaboration with Rocco Oliveto at the University of Molise, Italy and Andrea De Lucia at the University of Salerno, Italy.

Chapter 4 contains the results of a collaboration with Huzefa Kagdi at Wichita State University. The findings were previously published in the proceedings of the 17th IEEE Working Conference on Reverse Engineering (WCRE'10) [78]. The publication was invited to the Empirical Software Engineering special issue among the best papers at WCRE'10.

Chapter 2

Developer Recommendation

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 many factors; all of which if handled manually can be quite tedious [10]. 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 tens of 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 email the project team or developers, or discuss via issue tracking system, and rely on them for suggestions or advice on who has the helpful knowledge about a certain part of source code, 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).

We developed an approach that integrates two existing techniques to address the task of developer assignments to change requests [76]. 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 IR technique, specifically Latent Semantic Indexing (LSI) [46, 102], is first used to locate 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 [74]. 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 research philosophy is that the present+past of a software system leads to its better future evolution [73]. 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 the source code level, as in our previous work [74]. 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 integrated with a technique based on Mining Software Repositories for the expert developer recommendation task. This integrated 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 [10, 25, 27, 26, 74]. 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. This chapter includes a comprehensive evaluation of the approach 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.

2.1 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).

2.1.1 Locating Concepts with Information Retrieval

Concept refers to a human-oriented expression of the intent of implemented software or its parts [16]. 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 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 [117]. 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 [95, 157], identification of abstract data types [96], detection of high level concept clones [99, 141], recovery of traceability links between software artifacts [8, 43, 72, 97], identifying topics in source code [87], classifying and clustering software [82], measuring cohesion [98] and coupling [119], browsing relevant software artifacts [42] and tracing requirements [67, 91]. Using advanced IR techniques, such as those based on LSI [102, 120], allow 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 [137]. Marcus et al. [102] introduced a methodology to index and search source code using IR methods. Subsequently, Poshyvanyk et al. [120] 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 source code. In addition, we also created the corpus builder for large C++ projects, using srcML [40] and Columbus [55].

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 include the robustness of LSI with respect to outlandish identifier names and stop words (which are eliminated), and no need of a predefined vocabulary or morphological rules. The finer details of the inner workings of LSI used in this work are similar to its previous usages; we refer the interested readers to [102, 120].

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 4,756 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

Table 2.1: Top five classes extracted and ranked by the concept location tool that are relevant to the description of bug# 173881 reported in KOffice.

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

location tool. The partial results of the search (i.e., a ranked list of relevant classes) are summarized in Table 2.1.

2.1.2 Recommending Developers from Mining Software Repositories

We use the xFinder approach to recommend expert developers by mining version archives of a software system [74]. 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-

Figure 2.1: Part of a *KOffice* subversion log message.

```
<?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>
```

set. Figure 2.1 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 [74]. We used the measures to determine developers that were likely to be experts in a specific source code file, i.e., developercode map. The developer-code map is represented by the developer-code vector DV for the developer d and file f. $DV_{(d,f)} =< C_f, A_f, R_f >$, where:

- C_f is the number of commits, *i.e.*, commit contributions that include the file f and are committed by the developer d.
- A_f is the number of workdays, *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: $FV_{(f)} =< C'_f, A'_f, R'_f >$, where

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

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 [74]). Distance is an 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,

$$xFactor(d, v) = \frac{1}{DV_{(d,v)} - FV_{(f)}} \tag{2.1}$$

$$xFactor(d, v) = \frac{1}{\sqrt[2]{(C_f - C'_f)^2 + (A_f - A'_f)^2 + (R_f - R'_f)^2}} \tag{2.2}$$

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, with second highest value is ranked second, and so forth. Multiple developers with the same value 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

Figure 2.2: The procedure used in *xFinder* to give a ranked list of developer candidates to assist with a given source code entity.

```
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( ))
```

(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 2.2. 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 recommendation 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.

Table 2.2: 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

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 2.1, 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 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 2.2. Further details on xFinder are provided in our previous work on this topic [74].

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.

2.2 Case Study

We conducted a case study to empirically assess our approach according to the design and reporting guidelines presented in [132].

2.3 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 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 comprised of 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 cycle 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 Jun 2002, version 3.0 was released on 25 Jun 2004, and 3.3.2 was released on 21 Feb 2008. For each version we studied the change

history before the release date. *Eclipse* project contains millions 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 an 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?

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.

2.3.1 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 the step 2.

4. Compare the results of the 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 on 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 Table 2.5 and Table 2.3). We configured the tool to report only the top ten relevant source code files. Our previous

Table 2.3: 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 (Rf), package (Rp), system (Rs) and overall (Ro) 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	Rf	Rp	Rs	Ro
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 setpagenumber 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

work on the analysis of commits that are associated with bug fixes guided the choice of this particular cut point [3]. This satisfies the 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 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 the 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: where bt is the change request and dt is the developer who contributed at least one commit, ct, that includes the bug id of bt 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 the step 4 of our evaluation procedure. Additionally, we also sampled five feature and one refactoring requests. 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 2.3 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 a 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.

2.3.2 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 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.

2.3.3 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 2.3.

We explain the accuracy computation with the help of the bug# 167009, shown in Table 2.4. 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 2.5) 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 2.4 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%

Figure 2.3: Procedure for computing recommendation accuracy of the approach in the evaluation process.

```
# 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)
```

(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 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

Table 2.4: Accuracies of developer recommendations for bug# 167009 from KOffice at the File, Package (Pack), System (Sys), and overall (Ovl) levels. The 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

developers with a high accuracy.

Table 2.5 and Table 2.3 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

Table 2.5: 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 (Rf), package (Rp), system (Rs) and overall (Ro) 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	Rf	Rp	Rs	Ro
124527	display of calculation results incorrect for number or scientific format cells but correct for text format cells	0	0	0	0	–	–	–	–
125306	cant 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	cant 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 doesnt start from first page of custom slideshow	20	90	100	100	7	1	1	1
173630	format page layout does nothing	10	70	100	100	10	2	1	1
173882	cant 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

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).

2.3.4 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

Table 2.6: Automatic queries: Summary of an average ranking effectiveness for bugs of KOffice using different granularities (file (Rf), package (Rp), system (Rs) and overall (Ro)) for top 1, 3, 5, 10 recommendations.

	Avg. Ranking Effectiveness for Bugs				
	Top 1	Top 3	Top 5	Top 10	No rec.
Rf	11.11%	27.78%	33.33%	61.11%	38.89%
Rp	38.89%	66.67%	83.33%	83.33%	16.67%
Rs	72.22%	72.22%	72.22%	72.22%	27.78%
Ro	77.78%	83.33%	88.89%	88.89%	11.11%
Avg	50.00%	62.50%	69.44%	76.39%	23.61%

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 does manually selected value of ten classes fare with regards 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 2.5. As mentioned in the 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

Table 2.7: Automatic queries: Summary of an average ranking effectiveness for features of KOffice using different granularities (file (Rf), package (Rp), system (Rs) and overall (Ro)) for top 1, 3, 5, 10 recommendations.

	Avg. Ranking Effectiveness for Features				
	Top 1	Top 3	Top 5	Top 10	No rec.
Rf	50.00%	50.00%	50.00%	66.67%	33.33%
Rp	33.33%	50.00%	50.00%	66.67%	33.33%
Rs	33.33%	33.33%	33.33%	33.33%	66.67%
Ro	50.00%	50.00%	50.00%	66.67%	33.33%
Avg	41.67%	45.83%	45.83%	58.33%	41.67%

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 Table 2.6 and Table 2.7). On average across the three different granularities, which also includes overall 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 [117] 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 top most 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 (Ro) 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.

2.3.5 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

Table 2.8: 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 (Rf), package (Rp), system (Rs) and overall (Ro) levels are provided.

			Avg. Accuracy (%)				Avg. Ranking Effectiveness			
System	xFinder: Max Num of Dev	Num of Bugs	File	Pkg	Sys	Ovl	Rf	Rp	Rs	Ro
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

histories of $KOffice$. In other words, what is the impact on accuracy if different amounts of previous commits 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 2.8 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 be also 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 2.8 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 $KOffice$. The accuracies at the different levels of expertise granularities and ranking effectiveness were computed using the same process described for $KOffice$. Table 2.8 shows only the average accuracy results, which correspond, for example, to the last rows in Table 2.5 and Table 2.9. 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 in that $ArgoUML$ had the least number of active developers among our considered subject systems.

Table 2.9: 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 (Rf), package (Rp), system (Rs) and overall (Ro) 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	Rf	Rp	Rs	Ro
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 slideshow 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

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, concluding the first half of our study.

Table 2.10: 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 (Rf), package (Rp), system (Rs) and overall (Ro) 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	Rf	Rp	Rs	Ro
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 slideshow custom enable presentation view	50	20	0	50	3	7	–	3
	Accuracy	28	28	20	60	2.8	4	5	1.8

2.4 Comparative Study on $KOffice$

2.4.1 Effectiveness of automatic vs. manual LSI queries

The ranked lists of files generated by LSI are somewhat sensitive to the input query [58, 89, 102]. 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 choice 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.

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 Table 2.9 and Table 2.10). 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. Table 2.9 and Table 2.10 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 2.11-2.12). 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 a bit 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 similarly to package granularity for the top one and top ten. Similarly to package granularity, 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 we used words directly from bug reports, whereas we used words from commit messages for features. In our case, we found 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 the Table 2.11-2.12.

Table 2.11: Manual queries: Summary of an average ranking effectiveness for bugs in KOffice using different granularities (file (Rf), package (Rp), system (Rs) and overall (Ro)) for top 1, 3, 5, 10 recommendations

	Avg. Ranking Effectiveness for Bugs				
	Top 1	Top 3	Top 5	Top 10	No rec.
Rf	27.27%	36.36%	40.91%	54.55%	45.45%
Rp	45.45%	68.18%	77.27%	81.82%	18.18%
Rs	31.82%	45.45%	59.09%	59.09%	40.91%
Ro	63.64%	72.73%	77.27%	81.82%	18.18%
Avg	42.05%	55.68%	63.64%	69.32%	30.68%

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 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 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% vs. 52.2% and 72.2% vs. 32.2%. Automatic queries also improve the overall accuracy over manual queries by 2.8% (80% vs. 77.2%). On the other hand, the ranking effectiveness is better for manual queries at the file level granularity (3.3 for manual vs. 5.1 for automatic). These results indicate that manual queries on average obtain the correct recommendation quicker

Table 2.12: Manual queries: Summary of an average ranking effectiveness for features in KOffice using different granularities (file (Rf), package (Rp), system (Rs) and overall (Ro)) for top 1, 3, 5, 10 recommendations

	Avg. Ranking Effectiveness for Features				
	Top 1	Top 3	Top 5	Top 10	No rec.
Rf	16.67%	50.00%	66.67%	66.67%	33.33%
Rp	16.67%	33.33%	50.00%	66.67%	33.33%
Rs	16.67%	16.67%	16.67%	33.33%	66.67%
Ro	33.33%	66.67%	66.67%	66.67%	33.33%
Avg	20.83%	41.67%	50.00%	58.33%	41.67%

(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% vs. 23.3%) at the file level granularity. We also observed that the package-level accuracy is higher for manual queries (28% vs. 23.3%), whereas the system level accuracy yields better results for automatic queries (33% vs. 20%). Overall accuracy is higher for manual queries (60% vs. 43.3%), whereas 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

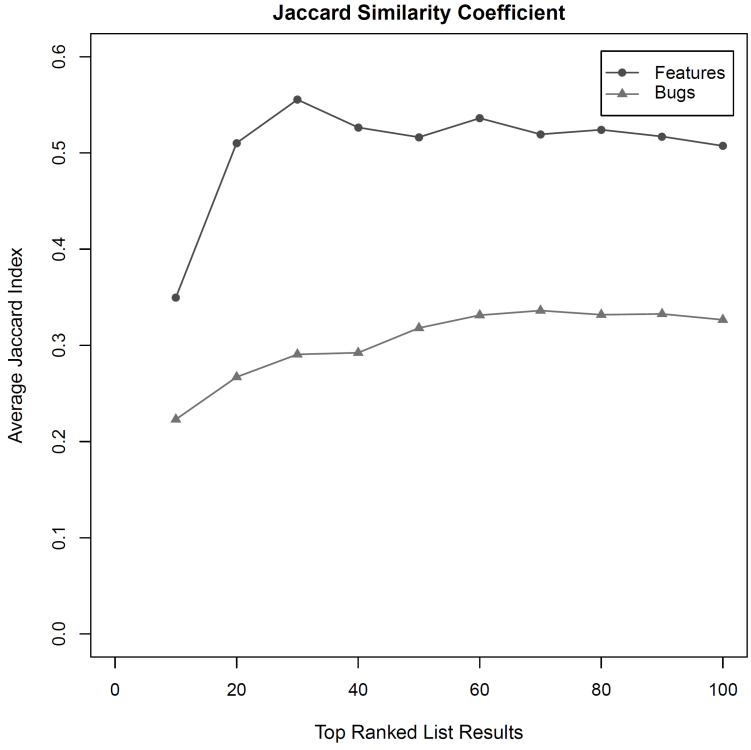
all granularities, i.e., file, package and system. While using automatic queries, fewer files on average are used from the ranked list to obtain the pertinent recommendations.

While analyzing average ranking effectiveness for manual vs. 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 outperform manual queries at the ranked list sizes for one (50% vs. 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 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 the following. 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 $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$.

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

Figure 2.4: Comparing the similarity and diversity of files for automatic and manual queries obtained from concept location in KOffice.



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 2.4 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 those for bugs. Again, these observations strengthen our conclusions that our approach works comparably 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 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.

2.4.2 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 $KOffice$ (partially because it contained the largest number of developers among the datasets considered).

To answer the 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 2.8 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 6 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 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 the 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 ap-

proach outperformed both randomly selected and maintainer based methods of developer recommendations on the considered $KOffice$ dataset by a substantial margin.

The details of the bug and accuracy data for $ArgoUML$, $Eclipse$, and $KOffice$ corresponding to Table 2.8 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, concluding the second half of our study.

2.5 Threats to Validity

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

2.5.1 Construct Validity

We discuss threats to construct validity that concerns to 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 [12]. 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

Table 2.13: 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?

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 Table 2.13). 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 [10]. 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 that was used in our method does not provide a comprehensive picture, i.e., an incomplete and monolithic view of accuracy from the considered dataset. 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, 3.4, and 3.5.

2.5.2 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 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 developers contributions to it in the form of past changes, was found to hold well. However, we do not clam 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 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 [129], the accuracy of the result will be affected. In our evaluation, we came 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 also should 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 open source development model, one cannot and should not police as to what is being reported and risk discouraging project participation and success. There is 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 not a 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 [19, 110], 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 [63, 161] in the mining software repositories community, it was observed that recent history was a better predictor of the future.

2.5.3 External Validity

We discuss threats to external validity that concerns with factors that are associated with generalizing the validity of our results to datasets 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 remains 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 [131] and other noisy information in bug/issue databases. Not accounting for such issues may lead to biased results positively or negatively or both. The considered sample sizes in our evaluation, however, is not uncommon, for example, Anvik et al. [10] 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).

2.5.4 Reliability

We discuss threats that would risk the replication of our evaluation study. Dataset not available: One of the main difficulties in conducting empirical studies is the access (or lack of it) to the dataset of interest. In our study, we used open source datasets 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 on 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 dataset.

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.

2.6 Background and 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.

2.6.1 Concept Location

Wilde et al. [154] 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 [7] and selecting execution scenarios [54]. Biggerstaff et al. [16] introduced the problem of concept assignment in the context of static analysis. They extracted identifiers from the source code and clustered them to support 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. Information-retrieval techniques [58, 102, 120, 118] 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 [65] and bugs [93], categorize software systems [147] and measure software cohesion [90]. Natural language processing techniques have been recently utilized to augment concept location approaches [69, 137].

Chen et al. [33] 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 [128]. Zhao et al. [160] proposed a technique that combines information

retrieval with branch-reserving call-graph information to automatically assign features to respective elements in the source code. Gold et al. [64] 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. [53] 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. [133, 134] use static and dynamic data to identify feature interaction in Java source code. Kothari et al. [86] use dynamic and static analysis for identifying canonical feature sets in software systems. Greevy et al. used dynamic analysis to analyze evolving features throughout software evolution [66]. Hill et al. [68] combined static and textual information to expedite traversal of program dependence graphs for impact analysis. Poshyvanyk et al. [89, 117] combined an information retrieval based technique with a scenario-based probabilistic ranking of the execution traces to improve the precision of feature location. Eaddy et al. [52] 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 [125]. Summaries of static, dynamic, and other approaches are available in [7, 34, 89, 101], while an overview of industrial feature location tools is available in [138]. To the best of our knowledge, no other work besides ours [76] has applied concept location techniques to the problem of expert developer recommendation.

2.6.2 Developer Contributions and Recommendation

McDonald and Ackerman [104] 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 [106] 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. [30]. EEL helps 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) [108] is another tool to locate people with 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 [11] 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 who fixed the bug. They found that both approaches have 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 [10]. 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 [140]. A recent approach to improve bug triaging uses graph based model based on Markov chains, which capture bug reassignment history [71]. 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. [103] 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 [60] 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. [149] analyzed the developers' working time of open source software. The email sent time was used to identify developers' working time. Bird et al. [18] mined email archives to analyze the communication and co-ordination activities of the participants. Del Rosso [48] 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. [94] proposed an approach for identifying developers using implementation expertise (i.e., using functionality by calling API methods). Yu and Ramaswamy [158] mined CVS repositories to identify developer roles (core and associate). The interaction between authors is used as clustering criteria. The KLOC and number of revisions are used to study the development effort for the two groups. Weissgerber et al. [153] 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 [59] studied the modification records (MRs) of CVS logs to visualize who are the people who tend to modify certain files. Fischer et al. [56] 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.

2.7 Discussion

The main contribution of our work is the first use of a concept location technique integrated 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.

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. [10] reported the precision of their approach as 57% and 64% on *Eclipse* and Firefox systems (albeit a different experiment setup and execution). At the very least, the presented approach did outperform two straightforward first choices that maybe 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 http://www.cs.wm.edu/semeru/data/jsme09-bugs-devs/) and hope that this data, including the specific bugs used, will provide the first steps towards 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).

Chapter 3

Traceability Link Recovery

Traceability links between software artifacts represent an important source of information, if available, for different stakeholders and provides important insights during software development [8]. Unfortunately, establishing and maintaining traceability links between software artifacts is an error prone and person-power intensive task [123]. Consequently, despite the advantages that can be gained, effective traceability is rarely established.

Extensive effort in the software engineering community has been brought forth to improve the explicit connection of software artifacts. Promising results have been achieved using Information Retrieval techniques [14, 46] to recover links between different types of artifacts (see e.g., [8, 97]). IR-based methods propose a list of candidate traceability links on the basis of the textual similarity between the text contained in the software artifacts. The conjecture is that two artifacts having high textual similarity share similar concepts, thus they are good candidates to be traced on each other. Several IR methods have been employed for traceability recovery, such as Vector Space Model [14] and Latent Semantic Indexing [46].

The experiments conducted to evaluate the accuracy of all these methods highlight that there is no clear technique able to sensibly outperform the others. In a recent study

[111] it has been empirically proved that widely used IR-based methods, such as VSM and LSI, are nearly equivalent, while Latent Dirichlet Allocation (LDA) [20]—a topic modeling technique recently used for traceability link recovery [13]—is able to capture some important information missed by the other exploited IR methods, while its accuracy is lower than that of the other IR methods.

This recent empirical result motivates our work. In particular, orthogonality of IR-based techniques may present the opportunity to improve accuracy through the integration of different techniques. In addition, topic modeling techniques should be further analyzed since they seem to capture a dimension missed by canonical IR methods. Thus, in this chapter we present (i) a novel method for traceability link recovery that exploits Relational Topic Model [31] for extracting and analyzing topics and relationships among them from software artifacts; and (ii) an approach to efficiently integrate different IR methods for traceability recovery. The results of the case study conducted on six software repositories indicate the benefits achieved while combining RTM with canonical IR techniques, in particular a technique based on VSM [14] and a technique based on probabilistic model, namely Jensen and Shannon [1]. The combination is highly valuable only when canonical methods are integrated with the topic modeling technique based on RTM. This is because RTM is orthogonal to VSM and JS, while the latter two canonical methods provide similar results, thus confirming the finding achieved in [111]. In the context of our case study, we also analyzed the impact on the recovery accuracy of the natural language (i.e., English versus Italian) and the type of software artifacts (i.e., use cases, UML diagrams, and test cases) to be traced on source code classes. The data used in the evaluation is made freely available

online, encouraging other researchers to replicate this work[1].

Summarizing, the specific contributions of the chapter are:

- the definition of a novel traceability recovery method based on RTM;
- an hybrid approach for traceability recovery that combines different IR methods. The integration of orthogonal techniques provides a tangible improvement in recovery accuracy;
- an analysis on how the language and the type of the software artifacts to be traced interact with the IR method and influence the recovery accuracy

Structure of the chapter. Section 3.1 presents background information to our work. Sections 3.2 and 3.3 present RTM and the hybrid traceability recovery method, respectively. Section 3.4 provides details on the design of the case study and presents the results achieved. Section 3.5 discusses the results achieved, while Section 3.6 concludes the chapter.

3.1 Background and Related Work

This section provides background notions and state of the art on IR-based traceability recovery.

3.1.1 IR-based Traceability Recovery

An IR-based traceability recovery tool uses an IR technique to compare a set of source artifacts (used as a query) against another (even overlapping) set of target artifacts and

[1] http://www.cs.wm.edu/semeru/data/icsm2011-traceability-rtm

rank the similarities of all possible pairs of artifacts. The textual similarity between two artifacts is based on the occurrences of terms (words) within the artifacts contained in the repository. The extraction of the terms from the artifact contents is preceded by a text normalization for removing most non-textual tokens (e.g., operators, special symbols, some numbers) and splitting into separate words source code identifiers composed of two or more words separated by using the under_score or CamelCase separators. Common terms (e.g., articles, adverbs) that are not useful to capture semantics of the artifacts are also discarded using a stop word function, to prune out all the words having a length less than a fixed threshold, and a stop word list, to cut-off all the words contained in a given word list. In our study, we also performed a morphological analysis, i.e., stemming [116], of the extracted terms to remove suffixes of words to extract their stems.

The extracted information is generally stored in a $m \times n$ matrix (called *term-by-document* matrix), where m is the number of all terms that occur in all the artifacts, and n is the number of artifacts in the repository. A generic entry $w_{i,j}$ of this matrix denotes a measure of the weight (i.e., relevance) of the i^{th} term in the j^{th} document [14]. In our study we adopted a standard term weighting scheme known as *term frequency – inverse document frequency (td-idf)* [14]. Term frequency awards terms appearing in an artifact with a high frequency, while inverse document frequency penalizes terms appearing in too many artifacts, i.e., non-discriminating terms. This means that a term is considered relevant for representing the artifact content and is assigned a relatively high weight if it occurs many times in the artifact, and is contained in a small number of artifacts.

Based on the term-by-document matrix representation, different IR methods can be used to rank conceptual similarities between pairs of artifacts. In our study we use a probabilistic

model, i.e., the JS model, VSM, and a topic model, i.e., RTM.

The JS similarity model is an IR technique driven by a probabilistic approach and hypothesis testing techniques. As well as other probabilistic models, it represents each artifact through a probability distribution. This means that an artifact is represented by a random variable where the probability of its states is given by the empirical distribution of the terms occurring in the artifact (i.e., normalized columns of the *term-by-document matrix*). The empirical distribution of a term is based on the weight assigned to the term for the specific artifact [1]. In the JS method the similarity between two artifacts is represented by the "distance" of their probability distributions measured by using the Jensen-Shannon Divergence [1]. The JS method does not take into account relations between terms. This means that having "automobile" in one artifact and "car" in another artifact does not contribute to the similarity measure between these two documents. Thus, the method suffers of the synonymy and the polysemy problems.

In the VSM, artifacts are represented as vectors of terms that occur within artifacts in the repository [14]. In particular, each column of the *term-by-document* matrix can be considered as an artifact vector in the m-space of the terms. Thus, the similarity between two artifacts is measured by the cosine of the angle between the corresponding vectors (i.e., columns of the *term-by-document* matrix). Such a similarity measure increases as more terms are shared between the two artifacts. In particular, as well as the JS method, VSM does not take into account relations between terms and it suffers of the synonymy and the polysemy problems.

Other than canonical IR-based recovery methods, we also propose the use of RTM as traceability recovery method. Details on such a technique are provided in Section 3.2.

3.1.2 State of the art

Antoniol *et al.* [8] are the first to apply IR methods to the problem of recovering traceability links between software artifacts. They use both the probabilistic and vector space models to trace source code onto software documentation. The results of the experimentation show the two methods exhibit similar accuracy. Marcus and Maletic [97] use LSI to recover traceability links between source code and documentation. They perform case studies similar in design to those in [8] and compare the accuracy of LSI with respect to the vector space and probabilistic models. The results show that LSI performs at least as well as the probabilistic and vector space models combined with full parsing of the source code and morphological analysis of the documentation. Abadi *et al.* [1] compare several IR techniques to recover traceability links between code and documentation. They compare dimensionality reduction methods (e.g., LSI), probabilistic and information theoretic approaches (i.e., JS), and the standard VSM. The results achieved show that the techniques that provide the best results are VSM and JS. Recently, Asuncion *et al.* [13] applied LDA for traceability link recovery between text-based artifacts (such as requirements and design documents). The authors monitor the operations (e.g., opening a requirements specification or visiting a Wiki page) performed by the software engineers during software development identifying a list of potentially related artifacts. Such relationships are then used to extract a set of topics that can be subsequently used to infer other relationships between code and documentation.

Heuristics [28, 38] and variants of basic IR methods [38, 43, 92, 136] have been proposed to improve the retrieval accuracy of IR-based traceability recovery tools. Promising

results have also been achieved using the relevance feedback analysis [6, 44, 67] that aims at improving the accuracy of the tool by learning from user feedback provided during the link classification. Recently, the use of the coverage link analysis has also been proposed to increase the amount of correct links traced by the software engineer with respect to a traditional process [45].

A issue which hinders the performance of IR techniques when applied to traceability recovery is the presence of vocabulary mismatch between source and target artifacts. Recently, a technique attempts to alleviate such an issue has been introduced [36, 62]. The proposed approach uses search engines to identify a set of terms related to the query and expand the query in an attempt to improve recovery accuracy. Empirical studies indicate that using web mining to enhance queries improves retrieval accuracy.

3.2 Relational Topic Model

Relational Topic Model [31] is a hierarchical probabilistic model of links and document attributes. RTM defines a comprehensive method for modeling interconnected networks of documents. There exist other models for explaining network link structure (see related work by Chang *et al.* [31]), but what separates RTM from those prior methods of link prediction is its ability to account for both document context and links between documents when making predictions. Prediction of links, which are modeled as binary random variables, is dependent on the topic assignments of the documents modeled. Another distinction, beneficial to our application, is that RTM does not require any prior observed links to make these predictions.

Generating a model consist of two steps (1) modeling the documents in a corpus and (2) modeling the links between pairs of documents. Established with a foundation on LDA, step one is identical to the LDA generative process. In the context of LDA, each document has a corresponding multinomial distribution over T topics and each topic has a corresponding multinomial distribution over the set of words in the vocabulary of the corpus. LDA assumes the following generative process for each document d_i in a corpus D:

1. Choose $N \sim$ Poisson distribution (ξ)
2. Choose $\theta \sim$ Dirichlet distribution (α)
3. For each of the N words w_n:
   (a) Choose a topic $t_n \sim$ Multinomial (θ).
   (b) Choose a word w_n from $p(w_n|t_n, \beta)$, a multinomial probability conditioned on topic t_n.

The second phase for the generation of the model exploited by RTM is as follows:

For each pair of documents d_i, d_j:

(a) Draw binary link indicator $y_{d_i,d_j}|t_i, t_j \sim \psi\left(\eta \cdot |t_i, t_j, \right)$ where $t_i = \{t_{i,1}, t_{i,2}, \ldots, t_{i,n}\}$

The link probability function ψ_ϵ is defined as:

$$\psi_\epsilon(y = 1) = \texttt{exp}(\eta^T(\overline{\texttt{t}}_{d_i} \circ \overline{\texttt{t}}_{d_j}) + v).$$

where links between documents are modeled by logistic regression. The $\circ$ notation represents the Hadamard product, $\overline{\texttt{t}}_d = \frac{1}{N_d}\sum_n z_{d,n}$ and `exp()` is an exponential mean function parameterized by coefficients η and intercept v.

Table 3.1: Characteristics of the software systems used in the experimentation.

System	Description	Source Artifact (#)	Target Artifact (#)	Correct links
eAnci	A system providing support to manage Italian municipalities	Use cases (139)	Classes (55)	567
		Use cases (30)	Classes (37)	93
EasyClinic*	A system used to manage a doctor's office	UML Diagrams (20)	Classes (37)	69
		Test Cases (63)	Classes (37)	204
eTour*	An electronic touristic guide developed by students.	Use cases (58)	Classes (174)	366
SMOS	A system used to monitor high school students (e.g., absence, grades)	Use cases (67)	Classes (100)	1,044

* A complete version of the software system is available in both English (ENG) and Italian (ITA).

Proposed applications of RTM [31] include assisting social network users in identifying potential friends, locating relevant citations for a given scientific paper, pinpointing related web pages of a particular web page, and computing coupling among source code classes in software [61]. Our intuition leads us to believe this model may serve well for traceability link recovery. In the context of traceability recovery, RTM is used to estimate topic distribution in the term-by-document matrix in order to define the link probability function. Such a function plays the same role of the artifact vectors in canonical vector-based IR methods, e.g., VSM. In particular, it is used to topically compare pairs of artifacts in order to obtain a list of candidate links.

One key distinction between establishing link probabilities in RTM and the canonical LDA is the underlying data used. Here, RTM uses topic assignments to make link predictions whereas to compute document similarities we use topic proportions for each document. This difference is discussed in more detail in the original work by Chang *et al.* [31].

3.3 The Hybrid Approach

Besides proposing to use RTM as a traceability recovery method, we also propose a new approach to improve the accuracy of recovery methods by integrating orthogonal IR methods, i.e. methods that provide different sets of recovered links. Our conjecture is supported by a preliminary study [111] that provides some evidence of (i) the equivalence (in terms of links recovered) of canonical IR methods, such as VSM, LSI, and JS and (ii) the presence of orthogonality between canonical IR methods and topic modeling techniques, in particular LDA. The proposed combined method is based on affine transformation [70], a technique used to combine experts' judgments previously used to combine orthogonal feature location techniques [117].

The basic idea behind our approach is that two IR methods can be viewed as two experts who provide their expertise to solve the problem of identifying links between a set of source artifacts and a set of target artifacts. The two experts, e.g., a canonical IR method and a topic modeling technique, express their judgments based on different observations. Both experts express judgments based on the textual similarity between two artifacts. However, canonical methods analyze the terms shared by two artifacts, while topic modeling techniques utilize probabilistic topic distributions and word distributions across all the artifacts. This allows two techniques to capture different information, as highlighted in [111] and confirmed in our study (see Section 3.4). Thus, the proposed approach integrates valuable (orthogonal) expertise of both experts to obtain a more accurate list of candidate links and minimize the effort of software developers.

Formally, the combination is obtained in two steps. In the first step, the judgments (i.e.,

similarities) of the two experts are mapped to a standard normal distribution as follows:

$$sim_{m_i}(x,y) = \frac{m_i(x,y) - mean(m_i(X,Y))}{stdev(m_i(X,Y))}$$

where X, Y are sets of software artifacts, $x \in X, y \in Y$ and $sim_{m_i}(x,y)$ is the normalized similarity of $m_i(x,y)$ where m_i is an IR method. The functions $mean()$ and $stdev()$ return the mean and standard deviation respectively, for the similarity values of all pairs of artifacts (x_a, y_b) using m_i. Note that the normalization phase is required because different experts may express judgments that are not commensurable.

In the second step, the normalized judgments are combined through a weighted sum:

$$sim_{combined}(x,y) = \lambda \times sim_{m_i}(x,y) + (1-\lambda) \times sim_{m_j}(x,y)$$

where $\lambda \in [0,1]$ expresses the confidence in each technique. The higher the value the higher the confidence in the technique. In Section 3.4.4 we experimentally identify two heuristics to define the value of λ.

3.4 Case Study

In this section we describe in detail the design and the results of the case study carried out to evaluate the proposed approach. The description of the study follows the Goal−Question−Metric [15] guidelines.

3.4.1 Definition and Context

The *goal* of the experiment was to analyze (i) the support given by RTM during traceability link recovery; (ii) whether RTM is orthogonal to VSM and JS canonical IR methods;

and (iii) whether the accuracy of IR-based traceability recovery methods improves when combining RTM with other canonical methods. The *quality focus* was on ensuring better recovery accuracy, while the *perspective* was both (i) of a researcher, who wants to evaluate the accuracy improvement achieved using a hybrid recovery method; and (ii) of a project manager, who wants to evaluate the possibility of adopting the hybrid technique within her software company.

The context of our study is represented by six software repositories, namely eAnci, EasyClinic (English and Italian versions), eTour (English and Italian versions), and SMOS. All the systems have been developed by final year students at the University of Salerno (Italy). Use cases and code classes are available for eAnci, eTour, and SMOS, while for EasyClinic those two types of artifacts as well as UML interaction diagrams and test cases are available. Note that EasyClinic and eTour were recently used as data set for the traceability challenge organized at TEFSE 2009[2] and 2011[3].

Table 3.1 shows the characteristics of the considered software systems in terms of type and number of source and target artifacts. The language of the artifacts for all the systems is Italian, while for the EasyClinic and eTour repositories both Italian and English versions are available. On each system links between source and target artifacts are recovered to analyze the accuracy of the experimented IR methods. The table also reports the number of correct links between source and target artifacts. The traceability links were derived from the traceability matrix provided by the original developers. Such a matrix was used as the oracle for evaluating the accuracy of the studied traceability recovery methods.

[2]http://web.soccerlab.polymtl.ca/tefse09

[3]http://www.cs.wm.edu/semeru/tefse2011

3.4.2 Research Questions

In the context of our study the followings research questions (**RQ**) were formulated:

- **RQ**$_1$: *Does RTM-based traceability recovery outperform other canonical IR-based approaches?*
- **RQ**$_2$: *Is RTM orthogonal as compared to canonical IR techniques?*
- **RQ**$_3$: *Does the combination of RTM and canonical IR methods outperform standalone methods?*

To respond to our research questions, we recovered traceability links between source code and documentation of EAnci, EasyClinic, eTour, and SMOS (see Table 3.1 for details).

To have a good benchmark for the proposed traceability recovery methods and cover a large number of IR methods, we selected and considered as canonical method the JS method and VSM based on the results of our previous study [111]. The selected techniques are widely used for traceability recovery and are accepted as state of the art for IR-based traceability recovery [35]. In the context of our study, IR methods were provided identical term-by-document matrices as an input in order to eliminate all pre-processing related biases.

We were also interested in analyzing how the proposed approach interacts with the types and the language of the artifacts to be traced. Thus, two more research questions were formulated:

- **RQ**$_4$: *Does the type of the artifacts to be traced interact with the IR method and affect the recovery accuracy?*

- **RQ$_5$**: *Does the language of the artifacts to be traced interact with the IR method and affect the recovery accuracy?*

To analyze the effect of the type of the artifacts to be traced, only EasyClinic (English and Italian) repositories were considered because it is the only repository in our dataset with different types of artifacts. Regarding the influence of the language, we used both the EasyClinic and eTour repositories as for these repositories we had versions of the artifacts written in both Italian and English.

3.4.3 Metrics

To evaluate the accuracy of each IR method the number of correct links and false positives were collected for each recovery activity performed. Indeed, the number of correct links and false positives were automatically identified by a tool. The tool takes as an input the ranked list of candidate links and classifies each link as correct link or false positive until all correct links are recovered. Such a classification is automatically performed by the tool exploiting the original traceability matrix as an oracle.

Method comparison. A preliminary comparison of different IR methods—i.e., research questions RQ$_1$ and RQ$_3$—is obtained using two well-known IR metrics, namely recall and precision [14]:

$$recall = \frac{|cor \cap ret|}{|cor|}\% \qquad precision = \frac{|cor \cap ret|}{|ret|}\%$$

where *cor* and *ret* represent the sets of correct links and links retrieved by the tool, respectively. Other than recall and precision, we also use average precision [14], which returns a single value for each ranked lists of candidate links provided.

A further comparison of the IR-based recovery methods exploits statistical analysis. In particular, we used a statistical significance test to verify that the number of false positives retrieved by one method is significantly lower than the number of false positives retrieved by another method. In other words, we compared the false positives retrieved by method m_i with the false positives retrieved by method m_j to test the following null hypothesis:

> *H_0: there is no difference between the number of false positives retrieved by m_i and m_j*

Thus, the dependent variable of our study is represented by the number of false positives retrieved by the traceability recovery method for each correct link identified. Since the number of correct links is the same for each traceability recovery activity (i.e., the data was paired), we decided to use the Wilcoxon Rank Sum test [41] to test the statistical significance difference between the false positives retrieved by two traceability recovery methods. The results were intended as statistically significant at $\alpha = 0.05$.

Other than testing the null hypothesis, it is of practical interest to estimate the magnitude of the difference between accuracy achieved with different IR methods (e.g., combined *vs.* stand-alone). To this aim, we used the Cohen d effect size [81], which indicates the magnitude of the effect of the main treatment on the dependent variables [81]). For dependent samples (to be used in the context of paired analysis) it is defined as the difference between the means, divided by the standard deviation of the (paired) differences between samples, i.e., false positive distributions. The effect size is considered small for $0.2 \leq d < 0.5$, medium for $0.5 \leq d < 0.8$ and large for $d \geq 0.8$ [39]. We chose the Cohen d effect size as it is appropriate for our variables (in ratio scale) and given the different levels (small,

medium, large) defined for it, it is quite easy to be interpreted.

Orthogonality Checking. To analyze the orthogonality of different IR methods ($\mathbf{RQ}_2$), we uses Principal Component Analysis (PCA), a statistical technique capable of identifying various orthogonal dimensions captured by the data (principal components) and which measure contributes to the identified dimensions. The analysis identifies variables (in our case IR-based techniques) which are correlated to principal components and which techniques are the primary contributors to those components. This information provides insights on the orthogonality between similarity metrics.

Moreover, to have a further analysis of orthogonality between traceability recovery methods we used the following overlap metrics [111]:

$$correct_{m_i \cap m_j} = \frac{|correct_{m_i} \cap correct_{m_j}|}{|correct_{m_i} \cup correct_{m_j}|}\%$$

$$correct_{m_i \setminus m_j} = \frac{|correct_{m_i} \setminus correct_{m_j}|}{|correct_{m_i} \cup correct_{m_j}|}\%$$

where $correct_{m_i}$ represents the set of correct links identified by the IR method m_i. It is worth noting that $correct_{m_i \cap m_j}$ captures the overlap between the set of correct links retrieved by two IR methods, while $correct_{m_i \setminus m_j}$ measures the correct links retrieved by m_i and missed by m_j. The latter metric gives an indication on how an IR method contributes to complementing the set of correct links identified by the other method.

Interaction of Artifact Types and Language. The interaction of the type and the language of the artifacts to be traced with the IR method ($\mathbf{RQ}_4$ and $\mathbf{RQ}_5$) was analyzed by using the Two-Way Analysis of Variance (ANOVA) [49] and interaction plots. The latter are simple line graphs where the means on the dependent variable (number of false positives)

Table 3.2: Principal Component Analysis. Results are for tracing use cases onto code classes.

	PC_1	PC_2	PC_3
% variance	79.74%	20.15%	0.11%
Cumulative %	79.74%	99.89%	100%
JS	**0.98**	-0.19	0.03
VSM	**0.97**	-0.19	-0.03
RTM	0.68	**0.73**	0.00

(a) EasyClinic-ENG

	PC_1	PC_2	PC_3
% variance	75.78%	24.11%	0.11%
Cumulative %	75.78%	99.89%	100%
JS	**0.99**	-0.09	0.04
VSM	**0.99**	-0.10	-0.03
RTM	0.54	**0.83**	0.00

(b) EasyClinic-ITA

	PC_1	PC_2	PC_3
% variance	68.51%	31.18%	0.31%
Cumulative %	68.51%	99.69%	100%
JS	**0.99**	0.11	0.07
VSM	**0.99**	0.08	-0.06
RTM	0.29	**0.95**	0.00

(c) eTour-ENG

	PC_1	PC_2	PC_3
% variance	67.12%	32.63%	0.25%
Cumulative %	67.12%	99.75%	100%
JS	**0.97**	0.21	0.06
VSM	**0.98**	0.18	-0.05
RTM	0.31	**0.94**	0.00

(d) eTour-ITA

	PC_1	PC_2	PC_3
% variance	63.79%	35.55%	0.66%
Cumulative %	63.79%	99.34%	100%
JS	**0.96**	0.24	0.10
VSM	**0.97**	0.18	-0.09
RTM	0.17	**0.98**	0.00

(e) SMOS

	PC_1	PC_2	PC_3
% variance	70.03%	29.65%	0.32%
Cumulative %	70.03%	99.68%	100%
JS	**0.98**	-0.19	-0.06
VSM	**0.98**	-0.18	0.06
RTM	0.42	**0.90**	0.00

(f) EAnci

for each level of one factor are plotted over all the levels of the second factor. When there is no interaction the resulting profiles are parallel, otherwise they are non-parallel [49].

3.4.4 Analysis of the Results

RQ_1: Accuracy of RTM. We first investigate whether RTM provides accuracy superior to that of other IR-based traceability recovery techniques. Fig. 3.1 provides the precision/recall curves achieved when tracing use cases onto code classes of eTour_{ENG}. From the results we are unable to identify an approach, which consistently exceeds the performance of all the others. As the figure shows, we have cases where RTM outperforms most other techniques for certain levels of recall, but there are also cases (e.g., on EasyClinic_{ITA}) where the performances of RTM are not consistently better than that acquired by other techniques.

The statistically analysis indicates that the RTM-based technique is capable of providing

Table 3.3: Overlap analysis. Results are for tracing use cases onto code classes.

	EasyClinic$_{ITA}$			EasyClinic$_{ENG}$			eTour$_{ENG}$			eTour$_{ITA}$			EAnci			SMOS		
	Cut points μ			Cut points μ			Cut points μ			Cut points μ			Cut points μ			Cut points μ		
	25	50	100	25	50	100	25	50	100	25	50	100	25	50	100	25	50	100
$correct_{JS \cap VSM}$	100	92	95	83	85	88	91	94	85	89	91	91	72	80	82	100	79	76
$correct_{JS \setminus VSM}$	0	3	4	16	15	8	4	5	7	10	5	7	11	7	10	0	8	6
$correct_{VSM \setminus JS}$	0	3	0	0	0	2	4	0	7	0	2	1	16	11	8	0	12	17
$correct_{JS \cap RTM}$	19	40	52	19	19	36	23	28	35	25	36	36	20	23	29	16	18	23
$correct_{JS \setminus RTM}$	42	22	21	38	36	21	41	35	22	34	29	25	41	26	24	26	24	23
$correct_{RTM \setminus JS}$	38	37	26	42	44	42	35	36	41	40	34	38	37	50	45	56	57	52
$correct_{VSM \cap RTM}$	19	40	51	15	17	33	23	29	32	26	34	35	15	26	31	16	17	26
$correct_{VSM \setminus RTM}$	42	22	20	35	32	21	41	32	24	30	29	23	46	26	22	26	25	25
$correct_{RTM \setminus VSM}$	38	37	28	50	50	45	35	38	42	43	36	40	38	47	45	56	56	47

statistically significant improvement over other canonical techniques only when tracing use cases onto code classes and interaction diagrams onto code classes of eTour$_{ENG}$ (p-values are lower than 0.01 with a high effect size). In all the other cases, the Wilcoxon tests indicate that RTM does not provide any statistically significant improvement over other stand-alone methods.

RQ$_2$: Orthogonality checking. Regarding the orthogonality of the experimented techniques, Table 3.2 reports the results of PCA, which indicate the prevailing characteristics of the analysis for all the results). Results of the six systems evaluated are in agreement with regards to both the number of principal components, which capture most of the variance, and the main contributors of those principal components. From the results, we can conclude that RTM is orthogonal to other canonical IR methods, that on the other hand are not orthogonal between them.

We also evaluated the degree of overlap amongst correct links for candidate sets provided by pairs of the techniques (one of the two techniques is the RTM-based traceability recovery

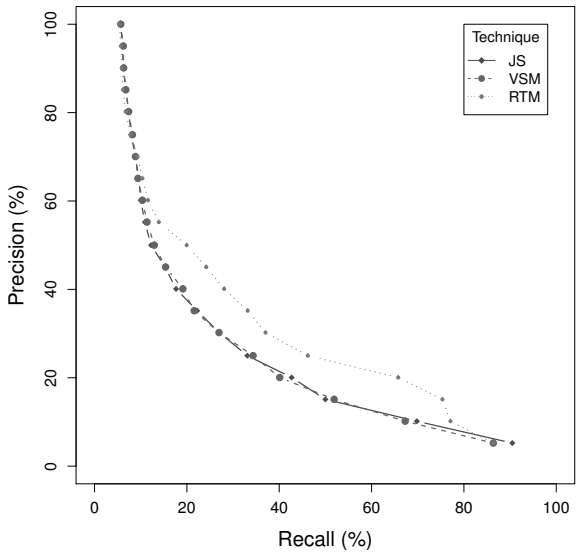


Figure 3.1: RTM vs VSM and JS: use cases onto code classes of eTour$_{ENG}$.

technique). Given the top μ candidate links we describe two aspects of the data. Provided the set of correct links obtained from both candidate sets we determine the percentage of correct links (1) identified by both techniques ($correct_{RTM \cap JS}$) and (2) distinctly revealed by RTM ($correct_{RTM \setminus JS}$). Once again, Table 3.3 shows a subset of the results achieved (among the average results). As we can see, the overlap between RTM and other techniques is relatively low, while the percentage of links identified by RTM and not identified by other canonical methods is high. In the case of recovering traceability links between use cases and code classes for EasyClinic$_{ENG}$ the results show that RTM provides a significant number of unique correct links. When ranked lists of 100 links are returned for the two techniques JS and RTM, JS is capable of identifying 33 correct links while RTM identifies 45 correct links. Among the correct links identified, 37% of them are common to both techniques while 42% are unique to RTM. Similar results are obtained for various systems, tracing links between different artifacts and for artifacts in various natural languages. These results confirm

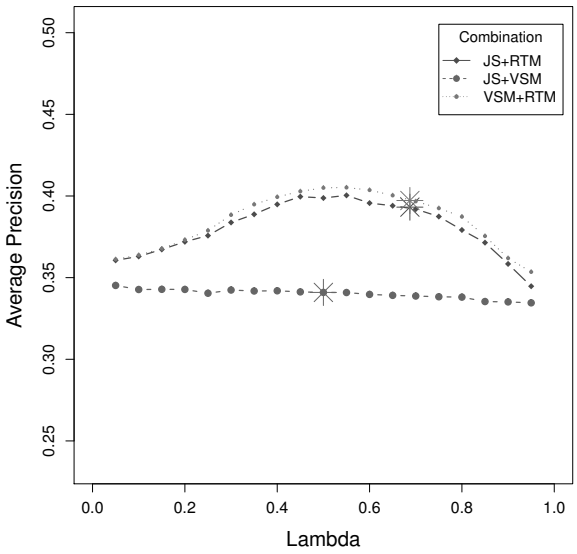


Figure 3.2: Average precision in eTour$_{ENG}$ using various values of lambda. Lambda represents the weight of the first method in the combination, while the asterisk indicates the accuracy of the PCA-based weighting technique.

Table 3.4: Comparing RTM-based combinations with stand-alone methods: Wilcoxon test results (p-values).

	EasyClinic$_{ENG}$	EasyClinic$_{ITA}$	eTour$_{ENG}$	eTour$_{ITA}$	SMOS	EAnci
RTM+JS vs JS	< **0.001**	< **0.001**	< **0.001**	< **0.001**	1	< **0.001**
RTM+VSM vs VSM	< **0.001**	< **0.001**	< **0.001**	< **0.001**	1	< **0.001**
RTM+JS vs RTM	< **0.001**	< **0.001**	< **0.001**	< **0.001**	< **0.001**	0.47
RTM+VSM vs RTM	< **0.001**	< **0.001**	< **0.001**	< **0.001**	< **0.001**	**0.04**

the findings of the PCA, indicating that RTM is a technique orthogonal to the other IR canonical methods.

RQ$_3$: Evaluation of the hybrid approach. Our goal is to improve traceability recovery accuracy by exploiting the orthogonality of IR methods. The proposed hybrid approach uses a parameter (λ) to assign a weight to the IR method to be combined (see Section 3.3). We analyze the effect of such a parameter on the accuracy (in terms of average precision) of the proposed approach using various values to lambda (0.05 through 0.95 with a step of 0.05) to combine techniques. Figure 3.2 shows the results achieved on ETour$_{ENG}$.

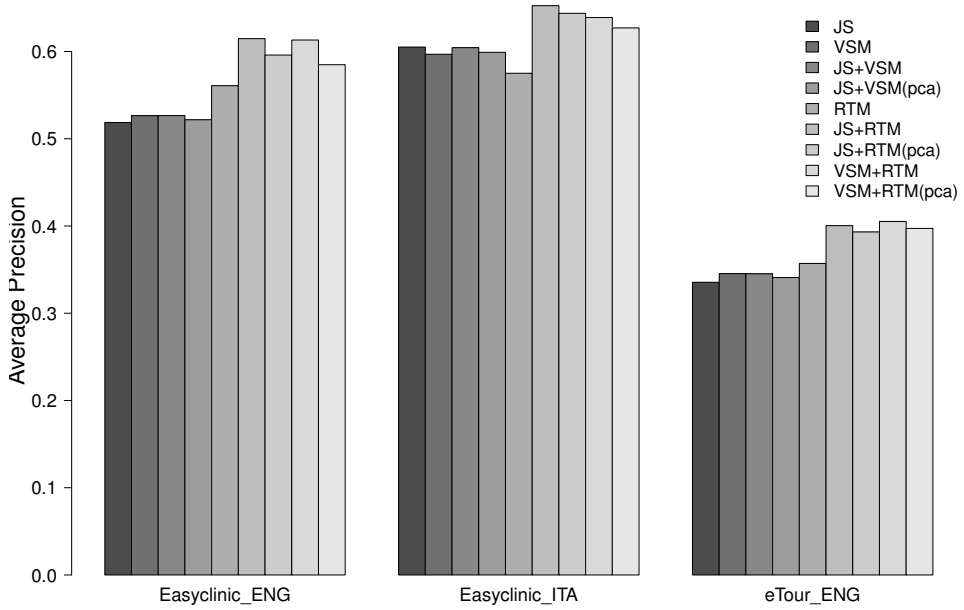


Figure 3.3: Results of average precision for retrieving all correct links for each EasyClinic$_{ENG}$ (left), EasyClinic$_{ITA}$ (middle), and eTour$_{ENG}$ (right). Results are presented the best performing combination and combinations obtained using the PCA-based weighting technique.

As expected the value of λ affects the accuracy of the proposed approach. Defining a "good" value for λ *a priori* is challenging. However, from the analysis of the results we identify two possible heuristics: (i) assign the same weight $\lambda = 0.5$ to the IR methods to be combined; (ii) use the proportion of variance obtained by PCA to weight the different IR methods. The former is a constant heuristic that generally provides good results, while the latter is an heuristic that is context-dependent and provides a more accurate estimation of λ. Such an heuristic is based on the observation that PCA identifies the different dimensions that describe a phenomenon, e.g., the similarity between pairs of artifacts, and gives an indication of the importance of each dimension (captured by one or more IR methods) in the description of this phenomenon, i.e., the proportion of variance. We conjecture that the higher the amount of variance captured by a particular dimension the higher should be the weight for the IR technique that best correlates to that dimension. The accuracy

obtained weighting the IR methods exploiting the proportion of variance obtained by PCA is highlighted in Figure 3.2 with an asterisk. Note that the PCA-based weighting technique provides better results than approximately 75% of combinations considered for the complete analysis). Such a result suggests that the PCA-based technique provides an acceptable means of combining IR methods for recovering traceability links.

Figure 3.2 also highlights the benefits provided by combining orthogonal IR methods. In particular, the accuracy of RTM+JS (or VSM) sensibly overcomes the accuracy of JS+VSM. To have further evidence of the benefits provided by the combination of different IR methods (using for *lambda* the best and the PCA-based values), Figure 3.3 shows the average precision achieved with stand-alone methods and different combinations of IR methods. As we can see the combination of RTM with other IR techniques results in significant improvement in average precision. In addition, the results achieved applying our proposed PCA-based weighting technique to combine orthogonal IR methods yields results, which consistently exceed the results of standalone techniques. Such a result confirms the usefulness of the proposed heuristic.

All these findings were also confirmed by the results of the Wilcoxon tests (see Table 3.4). In all the repositories, but SMOS and in one case for EAnci, the RTM combined method is able to statistically outperform the stand-alone methods. However, even if in the other cases the results did not reveal a statistically significant difference between techniques, the average precision of the combination is higher than any other standalone method.

$RQ_{4,5}$: Interaction of Artifact Type and Language. The ANOVA analysis confirmed the influence of the IR method, and highlighted the influence of both types and language of the artifacts to be traced. ANOVA also revealed a statistically significant in-

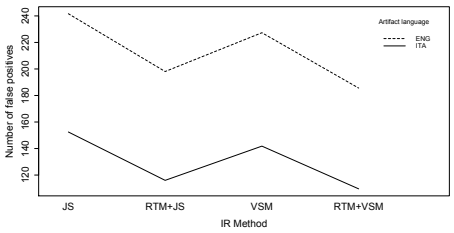
teraction between IR method and artifact language (on the ETour repositories), as well as between IR method and artifact type. The interactions investigated are statistically significant based on our dataset with $p < 0.001$. To better understand the interaction between factors, Figure 3.4 shows (a) the interaction plot between IR method and artifact language and (b) between IR method and artifact type. Regarding the influence of the artifact language, we observe that on EasyClinic better recovery accuracy is achieved on the Italian version, while on ETour better accuracy is generally achieved on the English version. The reason is that in the Italian version of the ETour repository identifiers in the source code are written in English. This negatively impacts the accuracy of the IR methods. As for the influence of the artifact type, we observe that the combination is highly valuable when tracing UML diagrams onto source code, while in the other cases the improvement is not so evident.

3.5 Discussion and Threats to Validity

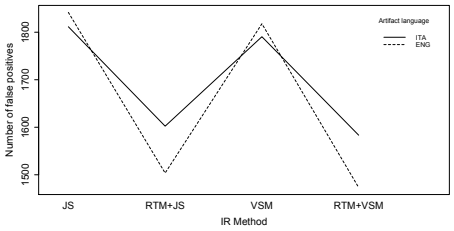
This section discusses the achieved results focusing the attention on the threats that could affect their validity [15].

3.5.1 Evaluation Method

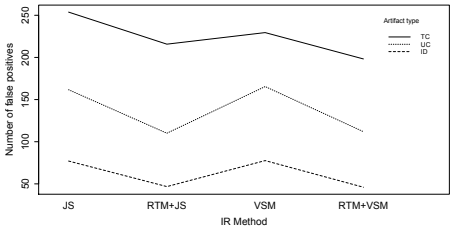
Recall, precision, and average precision are widely used metrics for assessing an IR technique and the number of false positives retrieved by a traceability recovery tool for each correct link retrieved reflects well its retrieval accuracy. The overlap metrics give a good indication on the overlap of the correct links recovered by the different IR methods. Moreover, the



(a) EasyClinic - Method and Language



(b) eTour - Method and Language



(c) EasyClinic - Method and Artifact Type

Figure 3.4: Interaction between Method and Artifact Types and between Method and Language.

similarity measures provided by each IR method are also statistically analyzed using PCA to verify the presence of IR methods that provide orthogonal similarity measures.

We also performed statistical analysis of the achieved results. Attention was paid not to violate assumptions made by statistical tests. Whenever conditions necessary to use parametric statistics did not hold (e.g., analysis of each experiment data), we used non-parametric tests, in particular Wilcoxon test for paired analysis. We also used a parametric test, i.e., ANOVA, to analyze the effect of different factors even if the distribution was not

normal. According to [156] this can be done since the ANOVA test is a very robust test. In addition, even if the distribution is not normally distributed we can relax the normality assumption applying the law of large numbers. In particular, according to [139] having a population higher than 100 it is possible to safety relax the normality assumption.

3.5.2 Object Systems and Oracle Accuracy

An important threat is related to the repositories used in the case study. EAnci, EasyClinic, eTour, and SMOS are not industrial projects, since they were developed by students. However, they are comparable to (or greater than) repositories used by other researchers [20, 8, 67, 91, 100] and both EasyClinic and ETour have been used as benchmark repositories in the last two editions of the traceability recovery challenge organized at TEFSE. In addition, to the best of our knowledge in this dissertation we reported the largest empirical study to evaluate and compare different IR methods for traceability recovery.

The investigated traceability recovery methods are based on IR techniques. Thus, the language of the artifacts may play an important role and affect the achieved results. To mitigate such a threat we performed the experimentation on two versions of the same artifact repository, one written in Italian and the other one in English. Analyzing the performance achieved on the same repository written in two different natural languages we had possibility to focus our investigation on the only difference between two versions of the repository, i.e., artifact language. The same considerations hold for the types of the artifacts to be traced.

Finally, the accuracy of the oracle we used to evaluate the tracing accuracy could also affect the achieved results. To mitigate such a threat we used original traceability matrices provided by the software developers. The links were also validated during review meet-

ings made by the original development team together with PhD students and academic researchers.

3.5.3 RTM Configuration and Number of Topics

RTM is a probabilistic topic model method which uses sampling techniques to infer underlying topics and topic/word distributions. When generating topic models, using an R project implementation[4], we performed a large number of sampling iterations to stabilize the set of topics extracted from a software system. In addition, the choice of the number of topics is critical and the proper way to make such a choice is still an open issue. For this reason we experimented different number of topics and for each repository we used the value that provides the best accuracy. Future work will be devoted to try to identify an heuristic to estimate the number of topics.

3.5.4 Heuristics to Weight the IR methods to be Combined

The proposed hybrid approach uses a parameter (λ) to assign a weight to the IR method to be combined. Defining a "good" value for λ *a priori* is challenging. For this reason, we experimentally identified two possible heuristics to weight the IR methods to be combined: (i) assign the same weight $\lambda = 0.5$ to the IR methods to be combined; (ii) use the proportion of variance obtained by PCA to weight the different IR methods. Both the heuristics are able to approximate the optimal λ. This means that the software engineer can initially use the value provided by the heuristic and then work around it by slightly increasing or decreasing it, within an incremental classification process.

[4]http://cran.r-project.org/web/packages/lda/

3.5.5 Orthogonality is a Key Point for Improving Accuracy

In our study we compare the accuracy of different IR methods, namely RTM, JS, and VSM. No IR method consistently provides superior recovery accuracy when compared to all other IR-based techniques considered. In particular, there are several cases where applying different IR-based traceability recovery techniques result in comparable accuracy and there are also cases where a particular technique yields better accuracy than any other technique considered.

The results achieved also highlight that JS and VSM are almost equivalent, while RTM captures a unique dimension in the data, i.e., it identifies correct links overlooked by JS and VSM. Across all systems evaluated, PCA reveals that there exists a principal component (typically accounting for 20%-35% of variance in data) with RTM as its main contributor. That is, RTM tends to contribute 73%-98% of the variance captured by that particular principal component. Through our analysis of overlap of links between pairs of IR methods we confirm that RTM is able to provide correct links omitted by other techniques for particular cut points.

Orthogonality of IR-based techniques is a key point for improving accuracy through combining different techniques. In our study we show that the combination of RTM with orthogonal IR techniques results in accuracy which surpasses that of either stand-alone technique. That is, in our results the improvements in precision exceed 30% in certain cases. Although improvements of that magnitude do not occur across all the systems evaluated, we do obtain acceptable increases in virtually all the scenarios.

3.6 Discussion

In this chapter we presented a novel traceability recovery method based on RTM and an hybrid approach for traceability recovery that integrates different IR methods. We also analyzed (i) the orthogonality of RTM as compared to other IR methods and (ii) the recovery accuracy improvement provided by the combination of RTM with other canonical IR methods. The empirical case study conducted on six software systems indicated that the hybrid method outperforms stand-alone IR methods as well as any other combination of non-orthogonal methods with a statistically significant margin.

Chapter 4

Impact Analysis

According to Arnold and Bohner [22] software-change impact analysis, or simply impact analysis (IA), is defined as the determination of potential effects to a subject system resulting from a proposed software change. The premise of impact analysis is that a proposed change may result in undesirable *side effects* and/or *ripple effects*. A side effect is a condition that leads the software to a state that is erroneous or violates the original assumptions/semantics as a result of a proposed change. A ripple effect is a phenomenon that affects other parts of a system on account of a proposed change. The task of an impact analysis technique is to estimate the (complete closure of) ripple effects and prevent side effects of a proposed change. The scope of the analyzed and estimated software artifacts may include requirements, design, and source code.

IA is a key task in software maintenance and evolution. Decades of research efforts have produced a wide spectrum of approaches, ranging from the traditional static and dynamic analysis techniques [24, 88, 114, 115, 124, 127] to the contemporary methods such as those based on Information Retrieval [25, 68, 119] and Mining Software Repositories [161]. Although ample progress has been made, there still remains much work to be done in further improving the effectiveness (*e.g.*, accuracy) of the state-of-the-art IA techniques. Our goal

is to develop a new and improved IA approach by reusing some of the existing contemporary solutions. Central to our approach are the information sources that are developer centric (*e.g.*, comments and identifiers, and commits and commit practices), rather than artifact centric (*e.g.*, static and dynamic dependencies such as call graphs).

In this chapter, we present an approach that integrates conceptual and evolutionary information to support IA in source code. The two sources of information are used to capture couplings between source code entities. Conceptual couplings capture the extent to which domain concepts and software artifacts are related to each other. This information is derived using. This analysis focused on a single version is consistent with its previous usages in IA [5, 119]. Evolutionary couplings capture the extent to which software artifacts were co-changed. This information is derived from analyzing patterns, relationships, and relevant information of source code changes mined from multiple versions in software repositories.

The research philosophy behind the integration of the two information sources is that *present+past* of software systems leads to better IA. For IA, both single (present) and multiple versions (past) analysis methods have been utilized independently, but their integrated use has not been previously investigated. The approaches presented in this chapter are a fundamental and necessary baseline step in this direction. We investigate two different integration approaches, *i.e.*, disjunctive and conjunctive, and compute impact sets at varying source code granularity levels (*e.g.*, files and methods). Our principal research hypothesis is that such combined methods provide improvements to the accuracy of impact sets.

An extensive empirical study on hundreds of changes from the open source systems, such as *Apache httpd*, *ArgoUML*, *iBatis*, and *KOffice* was conducted to test the research hypothesis. The results of the study show that the disjunctive combination of IR and MSR

techniques, across several cut points (impact set sizes), provides statistically significant improvements in accuracy over either of the two standalone techniques. For example, the disjunctive method reported improvements in recall values of up to 20% over the conceptual technique in *KOffice* and up to 45% improvement over the evolutionary technique in *iBatis*. These results are encouraging considering that the combinations do not require an overly complex blending of two separate approaches.

4.1 Background and Related Work

This chapter presents a novel approach for a key maintenance task, namely software change impact analysis, by involving conceptual and evolutionary couplings. There is a rich volume of literature covering each of these areas. Our intention is not to cover every individual work exhaustively, but to provide a breadth of the solutions offered to the problem and by the solutions.

4.1.1 Software Change Impact Analysis (IA)

Dependency analysis and *traceability* analysis are the two primary methodologies for performing impact analysis. Broadly, dependency analysis refers to impact analysis of software artifacts at the same level of abstraction (*e.g.*, source code to source code or design to design). Traceability analysis refers to impact analysis of software artifacts across different levels of abstractions (*e.g.*, source code to UML). Various dependency-analysis methods based on call graphs, program slicing [57], hidden dependency analysis [32, 150, 159], lightweight static analysis approaches [109, 115], concept analysis [148], dynamic analysis [88], hypertext systems, documentation systems, UML models [23], and Information Retrieval [5] are

already investigated in the literature. Queille et al. [122] proposed an interactive process in which the programmer, guided by dependencies among program components (*i.e.*, classes, functions), inspects components one-by-one and identifies the ones that are going to change – this process involves both searching and browsing activities. This interactive process was supported via a formal model, based on graph rewriting rules [33].

Coupling measures have been also used to support impact analysis in OO systems [24, 155]. Wilkie and Kitchenham [155] investigated if classes with high CBO (Coupling Between Objects) coupling metric values are more likely to be affected by change ripple effects. Although CBO was found to be an indicator of change-proneness in general, it was not sufficient to account for all possible changes. Briand et al. [24] investigated the use of coupling measures and derived decision models for identifying classes likely to be changed during impact analysis. The results of an empirical investigation of the structural coupling measures and their combinations showed that the coupling measures can be used to focus the underlying dependency analysis and reduce impact analysis effort. On the other hand, the study revealed a substantial number of ripple effects, which are not accounted for by the highly coupled (structurally) classes.

More recent work appears in [68, 127], where proposed tools can help navigate and prioritize system dependencies during various software maintenance tasks. The work in [68] relates to our approach in as much as it also uses lexical (textual) clues from the source code to identify related methods. Several recent papers presented algorithms that estimate the impact of a change on tests [85, 130]. A comparison of different impact analysis algorithms is provided in [114].

4.1.2 Conceptual Information in Software

Identifiers used by programmers for names of classes, methods, or attributes in source code or other artifacts contain important information and account for approximately half of the source code in software [47]. These names often serve as a starting point in many program comprehension tasks [29], hence it is essential that these names clearly reflect the concepts that they are supposed to represent, as self-documenting identifiers decrease the time and effort needed to acquire a basic comprehension level for a programming task [9].

The software maintenance research community recently recognized the problem of extracting and analyzing conceptual information in software artifacts. IR-based methods have been applied to support practical tasks. For instance, IR methods have been successfully used to support feature location [117, 120], traceability link recovery [8, 43] and impact analysis [5, 119]. We do not discuss other applications of IR-based techniques in the context of software maintenance due to space limitations, however, interested readers are referred to [17] for such an overview.

4.1.3 Evolutionary Information in Software Repositories

The term MSR has been coined to describe a broad class of investigations into the examination of software repositories (*e.g.*, *Subversion* and *Bugzilla*). The premise of MSR is that empirical and systematic investigations of repositories will shed new light on the process of software evolution, and the changes that occur over time, by uncovering pertinent information, relationships, or trends about a particular evolutionary characteristic of the system.

We now briefly discuss some representative works in MSR for mining of evolutionary

couplings. Zimmerman et al. [161] used *CVS* logs for detecting evolutionary coupling between source code entities. Association rules based on itemset mining were formed from the change-sets and used for change-prediction. Canfora et al. [25] used the bug descriptions and the *CVS* commit messages for the purpose of change prediction. An information retrieval method is used to index the changed files, and commit logs, in the *CVS* and the past bug reports from the *Bugzilla* repositories.

In addition, conceptual information has been utilized in conjunction with evolutionary data to support several other tasks, such as assigning incoming bug reports to developers [10, 71, 76], identifying duplicate bug reports [151], estimating time to fix incoming bugs [152] and classifying software maintenance requests [50]. Finally, we conducted a comprehensive literature survey on MSR approaches during the prologue of this work [73]. Xie's online bibliography and tutorial[1] on MSR is another well-maintained source.

The above discussion shows that both IR and MSR have been used for impact analysis. Also, IR techniques have been applied to software repositories. Our work differs in that we limit the use of IR to a single snapshot (*i.e.*, to derive conceptual couplings) of source code and data mining techniques are used on past commits of source code (*i.e.*, to derive evolutionary couplings). To the best of our knowledge, such a combined use of IR and MSR has not been presented elsewhere or empirically investigated before in the research literature. Our approach builds on existing solutions, but synergizes them in a new holistic technique.

[1] https://sites.google.com/site/asergrp/dmse

4.2 A Integrated Approach to Impact Analysis

A typical IA technique takes a software entity in which a change is proposed or identified and estimates other entities that are also potential change candidates, referred to as an estimated impact set. Our general approach computes the estimated impact set with the following steps:

Step 1: **Select** the first software entity, e_s, for which IA needs to be performed. For example, this first entity could be a result of a feature location activity. Note that IA starts with a given entity.

Step 2: Compute conceptual couplings with IR methods from the release of a software system in which the first entity is selected. Let $EM(e_s)$ be the set of entities that are conceptually related to the entity from Step 1.

Step 3: Mine a set of commits from the source code repository and compute evolutionary couplings. Here, only the commits that occurred before the release in the above step are considered. Let $EM(e_s)$ be the set of entities that are evolutionary coupled to the entity from Step 1.

Step 4: Compute the estimated impact set, $EM(e_s)$, from the combinations of couplings computed in steps 3 and 4.

We now discuss the details of these steps, especially conceptual and evolutionary couplings, and their combinations.

4.2.1 Conceptual Couplings

We use *conceptual similarity* as a primary mechanism of capturing conceptual coupling among software entities. This measure is designed to capture the conceptual relationship among documents. Formally, the conceptual similarity between software entities e_k and e_j (where e_k and e_j can be methods), is computed as the cosine between the vectors ve_k and ve_j, corresponding to e_k and e_j in the vector space constructed by an IR method (*e.g.*, Latent Semantic Indexing):

$$CSE(e_k, e_j) = \frac{\sum ve_k \times ve_j}{\sqrt{\sum (ve_k)^2} \times \sqrt{\sum (ve_j)^2}} \tag{4.1}$$

As defined, the value of $CSE(e_k, e_j) \in [-1, 1]$, as CSE is a cosine in the Vector Space Model. For source code documents, the entities can be attributes, methods, classes, files, etc. Computing attribute-attribute or method-method similarities, CSE is straightforward (*e.g.*, e_k and e_j are substituted by a_k and a_j in the CSE formula), while deriving method-class or class-class CSE requires additional steps. We define the conceptual similarity between a method m_k and a class c_j ($CSEMC$) with t number of methods as follows:

$$CSEMC(m_k, c_j) = \frac{\sum_{q=1}^{t} CSE(m_k, m_{jq})}{t} \tag{4.2}$$

which is an average of the conceptual similarities between method m_k and all the methods from class c_j . Using $CSEMC$ we define the conceptual similarity between two classes ($CSEBC$) $c_k \in C$ with r number of methods and $c_j \in C$ (where C is a set of classes in software) as:

$$CSEBC(c_k, c_j) = \frac{\sum_{l=1}^{r} CSEMC(m_{kl}, c_j)}{r} \tag{4.3}$$

which is the average of the similarity measures between all unordered pairs of methods from class c_k and class c_j . The assumption, which is used in defining CSE, $CSEMC$, and $CSEBC$, is that if the methods of a class relate to each other, then the two methods or classes are also related. For more details and examples, please refer to our preliminary work on conceptual coupling measures [118, 119].

To analyze conceptual information in a given release of a software system, the source code is parsed using a developer-defined granularity level (*i.e.*, methods or files). A corpus is created, so that each software artifact will have a corresponding document in it. We rely on *srcML* [40] for the underlying representation of the source code and textual information. *srcML* is an *XML* representation of source code that explicitly embeds the syntactic structure inherently present in source code text with *XML* tags. The format preserves all the original source code contents including comments, white space, and preprocessor directives, which are used to build the corpus.

4.2.2 Evolutionary Couplings

We mine the change history of a software system for evolutionary relationships. In our approach, evolutionary couplings are essentially mined patterns of changed entities. We employ *itemset* mining [2], as the specific order of change between artifacts is not considered. This unordered set allows the computed evolutionary couplings to be consistent with the conceptual couplings (with no change order between coupled artifacts).

Formally, a *software change history*, *SCH*, is a set of change-sets (commits) submitted to the source-control repository during the evolution of the system in the time interval λ. Also, let $E = \cup_{i=1}^{m} cs_i$ be the set of m entities, each of which was changed in at least one

change-set. An unordered evolutionary coupling is a set of source code entities that are found to be recurring in at least a given number (σ_{min}) of change-sets, $ec_u = e_p, e_q, \ldots, e_o$ where each $e \in E$ and there exists a set of related change-sets, $S(ec) = \{c \in SCH | ec \subseteq c\}$ with its cardinality, $\sigma(ec) = |S(ec)| \geq \sigma_{min}$. The $\sigma(ec)$ value of a mined pattern is termed its *support* value in the data mining vocabulary. Similarly, the σ_{min} value is termed as *minimum support value*. Also, let $EC = \cup_{i=1}^{k} ec_i$ be a set of all the evolutionary couplings observed in *SCH*.

For any given software entity from E, which could be the first point e_s for impact analysis, we compute all the association rules from the mined evolutionary couplings where it occurs as an antecedent (*lhs*) and another entity from E as a consequent (*rhs*). Simply put, an association rule gives the conditional probability of the *rhs* also occurring when the *lhs* occurs, measured by a *confidence* value. That is, an association rules is of the form $lhs \Rightarrow rhs$. When multiple rules are found for a given entity, they are first ranked by their confidence values and then by their support values; both in a descending order (higher the value, stronger the rule). We allow a user specified cut-off point to pick the top n rules. Thus, $EM(e_s)$ is the set of all consequents in the selected n rules.

Broadly, the presented approach for mining fine-grained evolutionary couplings and prediction rules consists of three steps: [Warning: Draw object ignored]

4.2.2.1 Extract Change-sets from Software Repositories

Modern source-control systems, such as *Subversion*, preserve the grouping of several changes in multiple files to a single change-set as performed by a committer. This information can be easily obtained (*e.g.*, *svn log* and *pysvn*).

4.2.2.2 Process to Fine-grained Change-sets

The differences in a file of a change-set can be readily obtained at a line-level granularity (*e.g.*, with *diff* utility). In this case, the line differences need to be mapped to the corresponding fine-grained differences in the syntactic constructs. Our approach employs s*rcDiff*, a lightweight methodology for fine-grained differencing of files in a change-set. *srcDiff* extends the *srcML* representation by also marking the regions of changes to the code in a collection of difference elements. Information about the syntactic changes to the code is found using an XPath query. [Warning: Draw object ignored]

4.2.2.3 Mine Evolutionary Couplings

A mining tool, namely *sqminer,* was previously developed to uncover evolutionary couplings from the set of commits (processed at fine-granularity levels with *srcDiff* should the need be). The basic premise of *sqminer* is if the same set of source code entities frequently co-changes then there is a potential evolutionary coupling between them. *sqminer* supports mining of both unordered and ordered patterns. These patterns are used to generate association rules that serve as prediction rules for source code changes. *sqminer* has already been applied previously to mine co-changes at the file level [80], uncover/discover traceability links [75], and mine evolutionary couplings of localized documents [73].

4.2.3 Disjunctive and Conjunctive Combinations

With regards to combining conceptual and evolutionary dependencies, there is a pertinent research question. *Should the union or intersection of the two estimations be considered*, *i.e.*, $EI(e_s) \cup EM(e_s)$ or $EI(e_s) \cap EM(e_s)$? This question may not be an issue, if both

$EI(e_s)$ and $EM(e_s)$ predict the same estimation set. If the estimation sets differ, taking their union could result in increased recall; however, at the expense of decreased precision (if a large number of false-positive are estimated). Alternatively, taking only the intersection imposes a stricter constraint that could result in increased precision; however, at the expense of decreased recall.

The combined approaches for IA that use the union and intersection of estimations of conceptual and evolutionary estimations are termed as *disjunctive approach* and *conjunctive approach* respectively. That is, $E(e_s)_{\cup} = EI(e_s) \cup EM(e_s)$ and $E(e_s)_{\cap} = EI(e_s) \cap EM(e_s)$. Our approach supports both of these combinations. Both approaches require the user to specify a starting entity as well as a cut point for deriving an estimated impact set. For a given cut point, μ, provided by the user, we compute the impact set of the disjunctive method $E(e_s)_{\cup}$ by determining $EI(e_s)$ and $EM(e_s)$ such that the cardinality of each set is equal (or the cardinality $EI(e_s)$ is larger by one entity) and the cardinality of their union equals μ. A similar approach is taken to obtain the impact set of the conjunctive method; however, in this case we ensure the cardinality of the intersection equals μ. They use typical sets of parameters needed for LSI and itemset mining algorithms.

4.2.4 Examples

In order to explain what each technique finds and the issues that arise in the integration of the techniques, we present an example from a real system. In Apache httpd commit# 888310 addresses the bug#47087[2] regarding *"Incorrect request body handling with Expect: 100-continue if the client does not receive a transmitted 300 or 400 response prior to send-*

[2] $https://issues.apache.org/bugzilla/show_bug.cgi?id = 47087$

Table 4.1: Example showing the accuracy gains of the disjunctive impact analysis method on the bug# 47087 in *Apache httpd*.

	Conceptual	Evolutionary	Disjunctive
1	**/server/protocol.c**	/modules/http/byterange_filter.c	**/server/protocol.c**
2	/modules/proxy/mod_proxy_http.c	**/modules/http/http_protocol.c**	/modules/proxy/mod_proxy_http.c
3	/modules/debugging/mod_bucketeer.c	/modules/proxy/mod_proxy_ftp.c	**/modules/http/byterange_filter.c**
4	/server/core_filters.c	/server/core.c	/modules/http/http_protocol.c

ing its body". In this revision to fix the bug there were three source code files which needed to be changed (/modules/http/http_filters.c, /modules/http/http_protocol.c, and /server/protocol.c). In order to perform impact analysis, the developer must have a starting entity. For this example, let us assume the developer discovers, through feature location, that fixing the problem requires modifying /modules/http/http_filters.c. From this point the developer can perform impact analysis to discover other entities which also require modification. Using conceptual and evolutionary couplings for impact analysis, we obtain the results in 4.1. As standalone techniques neither conceptual nor evolutionary coupling are capable of establishing 100% recall. Conceptual coupling ranks /server/protocol.c as first in the ranked list, but ranks /modules/http/http_protocol.c as 91^{st}, whereas evolutionary coupling ranks /modules/http/http_protocol.c second in the ranked list, but ranks /server/protocol.c as 16^{th}. We can combine the results using our disjunctive approach. This results in the set of entities that also appear in 4.1. Here we can see that when combined, the couplings are capable of identifying all methods requiring modification within an impact set, i.e., cut point, of five methods. Note that our disjunctive and conjunctive approaches result in sets as opposed to ranked lists (*i.e.,* the entities are unordered).

4.3 Case Study

In this section we describe the empirical assessment of our approach. We describe our study following the Goal-Question-Metrics paradigm [15], which includes *goals*, *quality focus, and context*. In the context of our case study we aim at addressing the research questions (RQs):

- **RQ1**: Does combining conceptual and evolutionary couplings improve the accuracy of IA when compared to the two standalone techniques?
- **RQ2**: Does the choice of granularity, *i.e.*, file or method, affect the accuracy of IA of standalone techniques and their combination?

The *goal* of the case study is to investigate these research questions. The *quality focus* is on providing improved accuracy, while the *perspective* was of a software developer performing a change task, which requires extensive impact analysis of related source code entities. Our two research questions directly address the effectiveness and expressiveness of an IA solution. With regards to effectiveness, it is desirable to have a technique that provides all, and only, the impacted entities, i.e., prevents false positives and false negatives in the estimated impact set as much as possible. Additionally, it is desirable to provide the developers with the ability to apply the IA technique at various source code granularities. Our approach offers this feature; however, an important issue is to assess the change in effectiveness at different levels of granularity.

4.3.1 Accuracy Metrics

4.3.1.1 Precision and Recall

Impact analysis techniques are typically assessed with the two widely used metrics *precision* (*i.e.*, inverse measure of false positives) and *recall* (*i.e.,* inverse measure of false negatives). These metrics are computed from the estimated impact set produced from a technique and the actual impact set from the established ground truth (*e.g.*, change-sets/patches after the proposed change is actually implemented or developer verification).

For a given entity e_s (*e.g.,* file and method) let $EI(e_s)$ be the set of entities that are conceptually related to the entity e_s. Let R_i be the set of actual or correctly changed entities with the entity e_s. The precision of conceptual couplings, P_{EI}, is the mean percentage of correctly estimated changed entities over the total estimated entities. The recall of conceptual couplings, R_{EI}, is the mean percentage of correctly estimated changed entities over the total correctly changed entities.

$$P_{EI} = \frac{1}{n}\sum_{i=1}^{n}\frac{|EI_i \cap R_i|}{EI_i} \times 100\% \tag{4.4}$$

$$R_{EI} = \frac{1}{n}\sum_{i=1}^{n}\frac{|E_i \cap R_i|}{R_i} \times 100\% \tag{4.5}$$

The precision and recall values for evolutionary couplings, disjunctive, and conjunctive methods can be similarly computed. The set $EM(e_s)$ would indicate the set of entities that are related to a known entity e_s based on evolutionary couplings. The sets $E(e_s)_{\cup}$ and $E(e_s)\cap$ would indicate the couplings from the disjunctive and conjunctive methods respectively.

Table 4.2: Characteristics of the subject systems considered in the empirical evaluation.

System	Version	LOC	Files	Methods	Terms
Apache(httpd)	2.2.3	311K	782	n/a	6583
ArgoUML	0.28	367K	1,995	n/a	9384
iBatis	3.0.0-216	70K	774	n/a	3772
KOffice 2.0.91	2.0.91	231K	6.5K	n/a	48513
KOffice 2.0.1	2.0.1	257K	6.7K	68.4K	32212

4.3.2 Evaluated Subject Systems

4.3.3 Examples

The *context* of our study is characterized by a set of four open source software systems, namely *Apache httpd*, *ArgoUML*, *iBatis*, and *KOffice*. The selected set of systems represents different primary implementation languages (*e.g.,* C/C++ and Java), size, development environment, and application domain. *Apache httpd* is an open source implementation of an HTTP server, which focuses on providing a robust and commercial-grade system. *ArgoUML* is a Java implementation of a UML diagramming tool. The *iBatis Data Mapper* framework provides a mechanism that simplifies the use of relational database systems with *Java* and *.NET* applications. *KOffice* is an application suite that includes various office productivity applications such as word (*i.e.*, *KWord*) and spreadsheet (*i.e.*, *KSpread*) processing. Specifics of various system characteristics appear in Table 4.3.3.

4.3.4 Evaluation Procedure

The source code changes in software repositories, *i.e.*, commits, are used for the evaluation purpose. Our general evaluation procedure consists of the following steps:

Table 4.3: Evolutionary training and (testing) datasets used for the empirical evaluation.

System	History	# of Commits	# of Entities
Apache(httpd)	2.2.9-2.3.5	1736 (287)	2086 (982)
ArgoUML	0.24-0.28	3375 (773)	4217 (621)
iBatis	3.0.0-190_b1 -3.0.0-240_b10	108 (40)	461 (118)
KOffice 2.0.91	2.0.0-2.0.91	2749 (522)	5580 (1072)
KOffice 2.0.1	2.0.0-2.0.2	763 (255)	1233 (533)
KOffice 2.0.1*	2.0.0-2.0.2	577 (192)	5530 (1438)

1. Compute conceptual couplings on a release (*e.g.*, *KOffice 2.0.91*) of a subject system – *Conceptual Training Set*.

2. Mine evolutionary couplings (and association rules) from a set of commits in a history period prior to the selected release in Step 1 – *Evolutionary Training Set*.

3. Select a set of commits in a history period after the selected release in Step 1 – *Testing Set*. Each commit in the testing set is considered as an actual impact set, *i.e.*, the ground truth, for evaluation purposes.

4. Derive disjunctive and conjunctive impact sets from the two training sets for each commit in the testing set.

5. Compute accuracy metrics for the two standalone techniques and their two combinations.

6. Compare standalone and combination accuracy results.

7. Repeat the above steps for all the considered subject systems and releases.

The details of the training and testing sets are detailed next.

4.3.4.1 Conceptual training sets - Corpora

We generated two sets of corpora from the subject systems corresponding to the granularity of *documents* at the *file* and *method* levels. The process of generating a corpus consisted of extracting textual information, *i.e.*, identifiers and comments, from the source code for the specific granularity level. The identifiers and comments, *i.e.*, *terms*, from each file (or a method if that is the chosen granularity) formed a *document*, whereas a complete collection of these documents formed a corpus. Once a corpus was built, LSI was used to index its *term-by-document* co-occurrence matrix. Conceptual couplings between source code documents, *i.e.*, files or methods, were then computed (see section 4.2). Details of the corpora, including the releases indexed, are provided in Table 4.3.3. The associated computing time was consistent with the previous uses [5, 119].

4.3.4.2 Evolutionary training sets

In order to obtain evolutionary training sets we selected a period of history, which preceded the version of the system used to build the corpus. For example, the corpus created for *Apache httpd* used the source code from version 2.2.3. The commit history from releases 2.2.9 to 2.2.3 was considered for the evolutionary training set. Commits with more than ten files were discarded. This type of filtering is a common heuristic used in mining techniques to mitigate factors such as updating the license information on every file or performing merging and copying [24, 46]. Furthermore, because commits may contain non source code files, only source code files were considered and other types discarded.

The tool *sqminer* was employed to mine evolutionary couplings (and association rules) in the *itemset mining* mode with minimum support values of 1, 2, 4, and 8. Also, we

considered all the possible association rules with the confidence values greater than zero. Mining was performed at both file and method levels of granularity. The mining time was in the order of a few seconds.

4.3.4.3 Testing set

The testing sets were extracted similar to training sets; however, the periods of history used were different from the training set. The testing set consists of commits extracted from a period of history after the release date of the version of the system used to build the corpus. For example, the commit history of *Apache httpd* after the release 2.2.3 and up to the release 2.2.5 was considered for the testing set. The testing set provides a way to evaluate our proposed approach. Similar approaches for the training and testing sets are previously reported in the literature, for example in [75, 161].

Table 4.3.4 shows the details of the evolutionary training and testing sets considered at the file and method levels. The entries corresponding to the method level are suffixes with a * symbol (same notation in other tables). They include a range of releases corresponding to different history periods. Also, the numbers of commits and files (methods) during those periods of history are provided. The (larger) training sets and (smaller) testing sets were extracted from the *History* (Table 4.3.4) periods before and after the Versions used to index with LSI. For the method level, the number of commits corresponds to commits that contained method changes (and so differs from those at the file level).

Table 4.4: Orthogonality check for various cut points of conceptual (C), evolutionary (E), and their combination. The results show that conceptual and evolutionary couplings provide orthogonal information, and support a strong case for combining them.

System	Metric	5	10	20	30	40	50
Apache	$C \setminus E$	32	33	35	35	35	37
	$E \setminus C$	39	36	28	23	20	17
	$C \cap E$	29	32	37	42	45	46
ArgoUML	$C \setminus E$	59	51	44	41	41	40
	$E \setminus C$	28	26	28	25	24	22
	$C \cap E$	13	23	29	34	35	38
iBatis	$C \setminus E$	67	65	69	70	70	70
	$E \setminus C$	15	21	14	14	13	12
	$C \cap E$	18	13	16	16	17	18
KOffice 2.0.91	$C \setminus E$	60	62	64	64	63	64
	$E \setminus C$	26	22	16	13	12	11
	$C \cap E$	14	16	20	23	24	26
KOffice 2.0.1	$C \setminus E$	42	41	40	42	43	44
	$E \setminus C$	43	38	36	35	33	32
	$C \cap E$	16	21	23	23	23	24
KOffice 2.0.1	$C \setminus E$	47	48	46	47	46	46
	$E \setminus C$	52	51	52	50	51	51
	$C \cap E$	1	1	2	3	3	3

4.3.5 Results

4.3.5.1 RQ1:Does combining conceptual and evolutionary couplings improve accuracy of IA?

Prior research efforts have investigated the performance of coupling metrics that use specific sources of information (*e.g.*, structural and textual) to capture couplings in source code. Our first research question focuses on determining if we can improve the accuracy of IA by

augmenting metrics based on complementary underlying information.

As a step toward determining the potential benefits of combining conceptual and evolutionary couplings, we analyze the orthogonality of the two standalone couplings. One situation where the combination of the techniques is beneficial is when techniques provide complementary sets of correct entities. If the standalone techniques considered for combination provide identical or very similar information, combining them may not be a worthwhile effort. In order to measure the degree to which the techniques could potentially complement one another we use the following metrics:

$$correct_{m_i \cap m_j} = \frac{correct_{m_i} \cap correct_{m_j}|}{correct_{m_i} \cup correct_{m_j}|}\% \tag{4.6}$$

$$correct_{m_i m_j} = \frac{correct_{m_i}\ correct_{m_j}|}{correct_{m_i} \cup correct_{m_j}|}\% \tag{4.7}$$

where $correct_{m_i}$ represents the set of source code entities correctly identified when using coupling metric m_i for IA. The two metrics capture the overlap between the set of correct source code entities and the percentage of correct entities identified only by m_i respectively.

The results of orthogonality metrics between the two metrics for the various systems are given in Table 4.4. Based on our datasets, the overlap between the set of correct links for the two approaches did not exceed 46%. Minimal overlap indicates potential orthogonality between the two techniques. One exception is the case where virtually all correct entities identified by one technique make up a small subset of the correct entities identified by the other technique. A similar scenario is where one technique performs inadequately and returns very few correct entities. Both cases are captured by our metric $correct_{m_i m_j}$. Our results contain cases where conceptual couplings are capable of identifying a large portion of

correct entities not identified by evolutionary couplings, and vice versa. In case of *KOffice 2.0.1* both techniques are capable of capturing a similar portion of correct entities. These findings support our premise that combining conceptual and evolutionary couplings could identify a larger set of correct entities.

Based on our datasets, conceptual and semantic couplings identify correct entities orthogonally. With this knowledge we direct our attention to our second step towards demonstrating the benefits of combining the couplings. Table 4.3.5.2 provides precision and recall results for the subject systems under study. These results are obtained by using the various couplings for IA. Only a subset of the cut points (μ) we considered are shown in Table 4.3.5.2. The cut points represent the sizes of the impact set considered with our combinations. For example, a cut point of 5 indicate that the estimated impact set with our approach contained 5 entities.

We considered both disjunctive and conjunctive approaches to combining couplings. The disjunctive approach outperforms the conjunctive approach in all cases considered (see Table 4.3.5.2). Additionally, the conjunctive approach is generally unable to provide improvement over either technique. This is somewhat expected because the two couplings appear complementary (see Table 4.4). The orthogonality between the sets of correct entities identified by the two couplings appears to contribute to the performance of the conjunctive approach. The utility of the conjunctive approach is probably better suited for scenarios where a pair of couplings identifies similar sets of correct entities, but varying sets of false positives. For such a scenario the conjunctive approach may serve as a useful filtering mechanism for false positives. The disjunctive approach better leverages the orthogonality between the couplings. The rest of the discussion about the combinations of two couplings

refers to the disjunctive approach.

[Warning: Draw object ignored]The prevailing pattern of our results demonstrates that the combination of conceptual and evolutionary couplings improves the performance over either standalone technique. Consider a case in Table 4.3.5.2 where $\mu = 30$ for *Apache httpd*. Conceptual and evolutionary couplings in this instance yield recall values of 58% and 51% respectively, while the combination of the two increases recall to 70%. Similar improvements are apparent throughout all the datasets considered in our evaluation. Another example is where $\mu = 50$ for *KOffice 2.0.1* (file-level granularity). In this case both conceptual and evolutionary couplings result in recall of 37% while their combination gives recall of 57%. Within our results a few cases surface that illustrate the importance of both techniques. For example, in the case where $\mu = 5$ for *iBatis* combining conceptual and evolutionary couplings does not improve accuracy. This can be partially attributed to the accuracy of the evolutionary coupling metric. In this case, the inadequate individual performance of a technique limits the gain acquired when they are combined.

Our results for combining conceptual and evolutionary couplings are promising. To further ascertain our conclusions on our initial dataset, we carried out a statistical test. We developed four testable null hypotheses:

$\mathbf{H}_{0CP}$: Combining conceptual and evolutionary couplings *does not significantly* improve precision results of impact analysis compared to conceptual couplings.

$\mathbf{H}_{0CR}$: Combining conceptual and evolutionary couplings *does not significantly* improve recall results of impact analysis compared to conceptual couplings.

$\mathbf{H}_{0EP}$: Combining conceptual and evolutionary couplings *does not significantly* improve precision results of impact analysis compared to evolutionary couplings.

Table 4.5: Results of Wilcoxon signed-rank test (= 30). The p values indicate that the disjunctive approach provided improvement is not by chance.

System	H_{0CP}	H_{0CR}	H_{0EP}	H_{0ER}	Null Hypothesis
Apache(httpd)	0.0002	0.0003	0.0001	0.0003	Rejected
ArgoUML	0.0050	0.0039	< 0.0001	< 0.0001	Rejected
iBatis	0.0126	0.0126	0.0001	0.0002	Rejected
KOffice 2.0.91	< 0.0001	< 0.0001	< 0.0001	< 0.0001	Rejected
KOffice 2.0.1	< 0.0001	< 0.0001	< 0.0001	< 0.0001	Rejected
KOffice 2.0.1*	< 0.0001	< 0.0001	< 0.0001	< 0.0001	Rejected

$\mathbf{H}_{0ER}$: Combining conceptual and evolutionary couplings *does not significantly* improve recall results of impact analysis compared to evolutionary couplings.

We also developed alternative hypotheses for the cases where the null hypotheses can be rejected with relatively high confidence. For example:

$\mathbf{H}_{aCP}$: Combining conceptual and evolutionary couplings *significantly* improve precision results of impact analysis compared to conceptual couplings.

The remaining three alternative hypotheses are formulated in a similar manner and are left out for brevity.

To test for statistical significance we used the Wilcoxon signed-rank test, a non-parametric paired samples test. Our application of the test determines whether the improvement obtained using the combination of conceptual and evolutionary couplings compared to standalone approaches is statistically significant.

Table 4.3.5.1 presents the results of performing the Wilcoxon signed-rank test. We performed the test for each of the four hypotheses for each system to determine whether the improvements for precision and recall when combining the techniques are statistically

Table 4.6: Precision (P) and recall (R) percentages results of conceptual coupling (Conc), evolutionary coupling (Evol), disjunctive (Disj), and conjunctive (Conj) approaches to impact analysis for all systems using various cut points. ImpC and ImpE show the improvement obtained by the disjunctive approach compared to conceptual and evolutionary couplings respectively. The disjunctive approach outperforms with statistical significance.

		5		10		20		30		40		50			5		10		20		30		40		50	
		P	R	P	R	P	R	P	R	P	R	P	R		P	R	P	R	P	R	P	R	P	R	P	R
Apache	Conc	15	28	11	38	7	49	6	58	5	63	4	67	KOffice 2.0.91	13	27	9	35	6	46	5	53	4	56	3	59
	Evol	18	38	11	43	6	48	4	51	3	53	3	54		9	19	6	22	4	24	3	26	2	28	2	28
	Disj	21	43	14	54	9	64	6	70	5	73	4	78		17	34	12	44	8	55	6	60	5	63	4	65
	Conj	16	34	10	40	6	47	4	47	3	48	2	48		8	16	5	21	3	22	2	22	2	23	1	23
	ImpC	6	15	3	16	2	15	0	12	0	10	0	11		4	7	3	9	2	9	1	7	1	7	1	6
	ImpE	3	5	3	11	3	16	2	19	2	20	1	24		8	15	6	22	4	31	3	34	3	35	2	37
ArgoUML	Conc	11	17	8	22	5	27	4	32	4	35	3	38	KOffice 2.0.1	10	19	7	26	4	30	3	33	3	35	2	37
	Evol	6	10	5	15	4	20	3	24	3	27	2	29		13	26	9	30	5	35	4	36	3	37	2	37
	Disj	11	19	9	27	6	33	5	38	4	41	4	44		16	34	11	41	7	49	5	53	4	55	4	57
	Conj	10	16	7	18	4	21	3	25	3	25	2	25		8	15	5	17	3	18	2	18	2	18	1	18
	ImpC	0	2	1	5	1	6	1	6	0	6	1	6		6	15	4	15	3	19	2	20	1	20	2	20
	ImpE	5	9	4	12	2	13	2	14	1	14	2	15		3	8	2	11	2	14	1	17	1	18	2	20
iBatis	Conc	17	27	13	37	10	56	7	59	6	61	5	63	KOffice 2.0.1*	5	4	3	4	2	5	2	6	1	7	1	7
	Evol	7	11	6	17	3	19	2	21	2	24	2	24		11	7	10	12	7	17	6	17	5	18	4	19
	Disj	18	27	14	40	10	60	8	66	6	68	5	68		14	10	12	15	8	21	6	22	5	24	4	25
	Conj	8	12	5	13	3	15	2	15	1	15	1	15		2	1	1	1	1	1	1	1	1	1	1	1
	ImpC	1	0	1	3	0	4	1	7	0	7	0	5		9	6	9	11	6	16	4	16	4	17	3	18
	ImpE	11	16	8	23	7	41	6	45	4	44	3	44		3	3	2	3	1	4	0	5	0	6	0	6

significant over the accuracy of standalone conceptual and evolutionary couplings. In all cases considered for our dataset we obtained a p-value less than 0.05, indicating that the improvement in accuracy obtained is not by chance.

4.3.5.2 RQ2: Does the choice of granularity (i.e., file vs. method) impact standalone techniques and their combinations?

Our second research question focuses on the impact of granularity on the accuracy of the standalone techniques, as well as their combinations. We examined the impact of different

granularities on the accuracy of the couplings when they are used for IA. Here, we focused on the accuracy of the various couplings on the system *KOffice 2.0.1*. For this system we obtained results at both file and method levels of granularity. Accuracy results of the techniques for IA are shown in Table 4.3.5.2. There is a noticeable decrease in accuracy when method level granularity is used. Conceptual coupling is affected by the difference in granularity more than evolutionary coupling. Regardless of the decrease in accuracy of the standalone techniques, when the two are combined there exists a statistically significant improvement in accuracy. In certain cases the improvement achieved is 6%. Generally, only a small portion of correct methods identified by both techniques overlap, *i.e.*, they exhibit orthogonality. This allows their combination to provide an enriched set of correct methods.

Our results show that the level of granularity does impact the accuracy of both standalone techniques and their combinations. Although finer granularity decreases accuracy of all approaches, it does not prevent the combination of the two from outperforming the standalone techniques. That is, the gain acquired by combining conceptual and evolutionary coupling exists regardless of the granularity considered in this study. For both file-level and method-level granularity levels, combining conceptual and evolutionary information delivers accuracy superior to either standalone technique.

4.3.6 Threats to validity

We address some of the threats to validity that could have impacted our empirical study and results. The uses of LSI and itemset mining algorithms are sensitive to a set of user-defined parameters. It is a viable risk that the improvements gained by our approach are valid only for a particular set of these parameter values. To address this risk, we experimented with

different parameter values. For example, the accuracy of evolutionary couplings decreases with an increase in the minimum support value; however, the trend of accuracy gains continued with our approach. We will continue our quest to obtain the optimal values with other studies in the future.

We measured the accuracy of IA with precision and recall metrics. It is possible that a different accuracy metric may produce a different result; however, both these metrics are widely used and accepted in the community, including for IA. We tried with F-measure, which is based on precision and recall, and also noticed statistically significant improvements with our disjunctive approach. We considered (later) commits as the gold standard for computing our accuracy metrics. It is reasonable to assume that not all the entities in a commit are related to a single change request, and a single commit may not capture all the entities related to a change request. Therefore, they may not be an accurate representation of the actual change-sets and could have compromised our accuracy basis. However, commits have been used as a basis for accuracy assessment previously (*e.g.*, see Zimmerman et al. [161]). We did some manual inspection and plan to conduct a user study with developer established actual impact sets in the future. We reported our findings at the granularity of file and method levels. A possible issue here could be how well our results hold for other granularity levels besides the two considered. We concur with previous studies [161] that file and method granularity levels provide a realistic balance of coarse and fine granularity levels for IA. The accuracies of the two standalone techniques, however low in certain cases to raise a practicality concern, are comparable to other previous results [161]. Our work shows how to improve accuracy by forming effective combinations.

We evaluated on datasets from four open source systems that represent a wide spectrum

of domains, programming languages (C/C++ and Java), sizes, and development processes. However, we do not claim that our combined approach would operate with equivalent improvement in accuracy on other systems, including closed source.

4.4 Discussion

The empirical assessment on four open source systems provides support for our approach with several conclusions in the context of change impact analysis. Combining conceptual and evolutionary couplings improves accuracy. Our findings indicate that in certain cases an improvement of 20% in recall is achieved when conceptual and evolutionary coupling is combined. The overall improvement obtained when combining the two techniques is statistically significant for the dataset used in our evaluation. Although our combining methods of couplings may appear straightforward, it did provide promising improvements in accuracy. Our findings show that the disjunctive approach clearly outperforms the conjunctive approach in accuracy. We conjecture that the difference in performance is, in part, an attribute of the orthogonal nature of the correct entities revealed by the two couplings in our empirical analysis.

Chapter 5

Conclusion

The dissertation establishes an approach which integrates information as well as integrates orthogonal analysis techniques to support three core software maintenance and evolution tasks. The approach is focused on (1) the integration of information, i.e., conceptual and evolutionary relationships found in structured and unstructured software artifacts and (2) the integration of analysis techniques, i.e., analyzing a single information source using orthogonal analysis techniques. Information Retrieval and Mining Software Repositories based techniques are leveraged to acquire insight necessary to reveal useful relationships. In this dissertation, we provide results of our approach when used to address crucial software maintenance task, namely developer recommendations, traceability link recovery, and impact analysis.

The main contributions of this dissertation include:

- **Developer Recommendation:** We present a novel approach to recommend expert developers for software change requests (e.g., bug reports and feature requests) of interest. Our approach integrates IR with MSR. 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. Expert developers are then mined

from the previous commits from version control repositories related to the identified relevant source code entities. The results of our evaluation on three open-source systems, namely $KOffice$, $Eclipse$ and $ArgoUML$, show that our approach can identify relevant developers to work on change requests with fairly high accuracy and in an effective ranked order. The results show that the overall accuracies of the correctly recommended developers are between 47% and 96% for bug reports, and between 43 and 60for feature requests. Through the integration of information, we provide project leads and developers with an approach to automatically suggest relevant developers for a given change request in free-form text.

- **Traceability Link Recovery:** We developed and evaluated a novel approach to TR that integrates orthogonal analysis IR techniques. In the literature, different Information Retrieval methods have been proposed to recover traceability links among software artifacts. We conduct an empirical study and observe that no individual IR technique sensibly outperforms the others, however, some methods recover different, yet complementary traceability links. In this work we leverage this empirical finding and present an integrated approach to combine orthogonal IR techniques, which have been statistically shown to produce dissimilar results. We consider the following IR techniques, Vector Space Model, probabilistic Jensen and Shannon model, and Relational Topic Modeling, which has not been used in the context of traceability link recovery before. We investigated (i) the orthogonality of RTM as compared to other IR methods and (ii) the recovery accuracy improvement provided by the combination of RTM with other canonical IR methods. Based on our evaluation,

conducted on six software systems, we conclude that the hybrid method outperforms stand-alone IR methods as well as any other combination of non-orthogonal methods with a statistically significant margin.

- **Impact Analysis:** We have developed a novel approach to support the software change impact analysis. Our approach integrates conceptual and evolutionary information. Information Retrieval is used to identify couplings based on conceptual relationships from a single release of the software system's source code. MSR is used to identify couplings based on evolutionary relationships mined from source code commits. An empirical study is conducted on four open source systems. Our finding indicate that integrating conceptual and evolutionary information improves accuracy. In certain cases, we observe an improvement as much as 20%. Furthermore, the improvement obtained when integrating the two sources of information is statistically significant for the dataset used in our evaluation.

Our findings indicate that our integrated approach outperforms individual techniques which are currently considered "state-of-the-art". Furthermore, integrating multiple information sources as well as integrating orthogonal analysis techniques to address software maintenance task sets our approach apart from previously reported relevant solutions in the literature.

Bibliography

[1] A. Abadi, M. Nisenson, and Y. Simionovici. A traceability technique for specifications. In *16th IEEE International Conference on Program Comprehension (ICPC'08)*, pages 103–112, Amsterdam, The Netherlands, 2008.

[2] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *11th International Conference on Data Engineering*, Taipei, Taiwan, 1995. IEEE Computer Society: Los Alamitos CA.

[3] A. Alali, H. Kagdi, and J.I. Maletic. What's a typical commit? a characterization of open source software repositories. In *16th IEEE International Conference on Program Comprehension (ICPC'08)*, Amsterdam, The Netherlands, 2008.

[4] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guhneuc. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *18th conference of the Centre for Advanced Studies on Collaborative research (CASCON '08)*, 2008.

[5] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. Identifying the starting impact set of a maintenance and reengineering. In *4th European Conference on Software Maintenance (CSMR2000)*, pages 227–230, Zurich, Switzerland, 2000. CS Press.

[6] G. Antoniol, G. Casazza, and A. Cimitile. Traceability recovery by modeling programmer behavior. In *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 240–247, Brisbane, Australia, 2000.

[7] G. Antoniol and Y.G. Guhneuc. Feature identification: An epidemiological metaphor. *IEEE Transactions on Software Engineering*, 32(9):627–641, 2006.

[8] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970 – 983, 2002.

[9] Giuliano Antoniol, Yann-Gael Gueheneuc, Ettore Merlo, and Paolo Tonella. Mining the lexicon used by programmers during software evolution. In *23rd IEEE International Conference on Software Maintenance (ICSM'07)*, pages 14–23, Paris, France, 2007. IEEE Computer Society Press.

[10] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *28th International Conference on Software Engineering (ICSE'06)*, pages 361–370, 2006.

[11] John Anvik and Gail Murphy. Determining implementation expertise from bug reports. In *Fourth International Workshop on Mining Software Repositories (MSR'07)*, Minneapolis, MN, 2007.

[12] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *31st IEEE/ACM International Conference on Software Engineering (ICSE'09)*, pages 298–308, Vancouver, British Columbia, Canada, 2009.

[13] H. Asuncion, A. Asuncion, and R. Taylor. Software traceability with topic modeling. In *32nd International Conference on Software Engineering (ICSE'10)*, 2010.

[14] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[15] V. R. Basili, G. Caldiera, and D. H. Rombach. *The Goal Question Metric Paradigm*. John W and S, 1994.

[16] T.J. Biggerstaff, B.G. Mitbander, and D.E. Webster. The concept assignment problem in program understanding. In *15th IEEE/ACM International Conference on Software Engineering (ICSE'94)*, pages 482–498, 1994. hard copy.

[17] D. Binkley and D. Lawrie. *Information Retrieval Applications in Software Maintenance and Evolution*. Encyclopedia of Software Engineering. Taylor & Francis LLC, 2010.

[18] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks. In *International Workshop on Mining Software Repositories (MSR'06)*, pages 137–143, Shanghai, China, 2006.

[19] C. Bird, D. S. Pattison, R. M. D'Souza, V. Filkov, and P. T. Devanbu. Latent social structure in open source projects. In *ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'08)*, pages 24–35, Atlanta, GA, 2008.

[20] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[21] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1981.

[22] S. Bohner and R. Arnold. *Software Change Impact Analysis*. IEEE Computer Society, Los Alamitos, CA, 1996.

[23] L. Briand, Y. Labiche, and G. Soccar. Automating impact analysis and regression test selection based on uml designs. In *International Conference on Software Maintenance (ICSM'02)*, pages 252–261, Montreal, Quebec, Canada, 2002. Briand02.pdf.

[24] Lionel Briand, Jurgen Wust, and Hakim Louinis. Using coupling measurement for impact analysis in object-oriented systems. In *IEEE International Conference on Software Maintenance (ICSM'99)*, pages 475–482. IEEE Computer Society Press, 1999.

[25] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *11th IEEE International Symposium on Software Metrics (METRICS'05)*, pages 20–29, 2005.

[26] Gerardo Canfora and L. Cerulo. Fine grained indexing of software repositories to support impact analysis. In *International Workshop on Mining Software Repositories (MSR'06)*, pages 105 – 111, 2006.

[27] Gerardo Canfora and L. Cerulo. Supporting change request assignment in open source development. In *21st Annual ACM Symposium on Applied Computing (SAC'06)*, pages 1767 – 1772, 2006.

[28] Giovanni Capobianco, Andrea De Lucia, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. On the role of the nouns in ir-based traceability recovery. In *17th IEEE International Conference on Program Comprehension (ICPC'09)*, pages 148 – 157, Vancouver, British Columbia, Canada, 2009.

[29] C. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *6th IEEE Working Conference on Reverse Engineering (WCRE'99)*, pages 112–122, Atlanta, Georgia, USA, 1999.

[30] Marcelo Cataldo, Patrick Wagstrom, James Herbsleb, and Kathleen M. Carley. Identification of coordination requirements: Implications for the design of collaboration and awareness tools. In *20th anniversary conference on Computer supported cooperative work (CSCW'06)*, pages 353 – 362, Alberta, Canada, 2006.

[31] Jonathan Chang and D. M. Blei. Hierarchical relational models for document networks. *Annals of Applied Statistics*, 2010.

[32] K. Chen and V. Rajlich. Ripples: Tool for change in legacy software. In *International Conference on Software Maintenance (ICSM'01)*, pages 230–239, Florence, Italy, 2001. Chen01.pdf.

[33] Kunrong Chen and Vaclav Rajlich. Case study of feature location using dependence graph. In *8th IEEE International Workshop on Program Comprehension (IWPC'00)*, pages 241–249, Limerick, Ireland, 2000.

[34] B. Cleary, C. Exton, J. Buckley, and M. English. An empirical analysis of information retrieval based concept location techniques in software comprehension. *Empirical Software Engineering, An International Journal*, 14(1):93–130, 2009.

[35] J. Cleland-Huang, B. Berenbach, S. Clark, R. Settimi, and E. Romanova. Best practices for automated traceability. *IEEE Computer*, 40(6):27–35, 2007. 1271947.

[36] J. Cleland-Huang, A. Czauderna, M. Gibiec, and J. Emenecker. A machine learning approach for tracing regulatory codes to product specific requirements. In *32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, pages 155–164, Cape Town, South Africa, 2010.

[37] J. Cleland-Huang, Y. Shin, E. Keenan, A. Czauderna, G. Leach, E. Moritz, **Gethers, M.**, D. Poshyvanyk, J. H. Hayes, and W. Li. Toward actionable, broadly accessible contests in software engineering. In *Proc. of 34rd ACM/IEEE International Conference on Software Engineering (ICSE'12), New Ideas and Emerging Results (NIER) Track*, Zurich, Switzerland, 2012.

[38] Jane Cleland-Huang, R. Settimi, C. Duan, and X. Zou. Utilizing supporting evidence to improve dynamic requirements traceability. In *International Requirements Engineering Conference (RE'05)*, pages 135–144, Paris, France, 2005.

[39] J. Cohen. *Statistical power analysis for the behavioral sciences*. 1988.

[40] Michael L. Collard, Huzefa H. Kagdi, and Jonathan I. Maletic. An xml-based lightweight c++ fact extractor. In *11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pages 134–143, Portland, OR, 2003. IEEE-CS.

[41] W. J. Conover. *Practical Nonparametric Statistics*. Third Edition, Wiley, 1998.

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

[43] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artefact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology*, 16(4), 2007.

[44] A. De Lucia, R. Oliveto, and P. Sgueglia. Incremental approach and user feedbacks: a silver bullet for traceability recovery. In *IEEE International Conference on Software Maintenance (ICSM'06)*, pages 299–309, Philadelphia, Pennsylvania, 2006.

[45] Andrea De Lucia, Rocco Oliveto, and Genoveffa Tortora. The role of the coverage analysis during ir-based traceability recovery: A controlled experiment. In *25th IEEE International Conference on Software Maintenance (ICSM'09)*, pages 371–380, Edmonton, Alberta, Canada, 2009.

[46] Scott Deerwester, Susan T. Dumais, G. W. Furnas, Thomas K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41:391–407, 1990.

[47] F. Deissenboeck and M. Pizka. Concise and consistent naming. In *13th IEEE International Workshop on Program Comprehension (IWPC'05)*, pages 97–106, St. Louis, Missouri, USA, 2005.

[48] Christian Del Rosso. Comprehend and analyze knowledge networks to improve software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 21:189 – 215, May 2009.

[49] J. L. Devore and N. Farnum. *Applied Statistics for Engineers and Scientists*. 1999.

[50] G. A. Di Lucca, M. Di Penta, and S. Gradara. An approach to classify software maintenance requests. In *IEEE International Conference on Software Maintenance (ICSM'02)*, pages 93–102, Montral, Qubec, Canada, 2002.

[51] B. Dit, M. Revelle, **M. Gethers**, and D. Poshyvanyk. Feature location in source code: A taxonomy and survey. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, to appear.

[52] M. Eaddy, A. V. Aho, G. Antoniol, and Y.G. Guhneuc. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *16th IEEE International Conference on Program Comprehension (ICPC'08)*, pages 53–62, Amsterdam, The Netherlands, 2008.

[53] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210 – 224, 2003.

[54] Andrew D. Eisenberg and Kris De Volder. Dynamic feature traces: Finding features in unfamiliar code. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 337–346, Budapest, Hungary, 2005.

[55] Rudolf Ferenc, Arpad Beszedes, and T. Gyimthy. Extracting facts with columbus from c++ code. In *8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 4–8, Tampere, Finland, 2004.

[56] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *IEEE Working Conference on Reverse Engineering (WCRE'03)*, pages 90–101, 2003.

[57] K. Gallagher and J. Lyle. Using program slicing in software maintenance. *Transactions on Software Engineering*, 17(8):751–762, 1991. Gallagher91.pdf.

[58] G. Gay, S. Haiduc, M. Marcus, and T. Menzies. On the use of relevance feedback in ir-based concept location. In *25th IEEE International Conference on Software Maintenance (ICSM'09)*, pages 351–360, Edmonton, Canada, 2009.

[59] Daniel M. German. An empirical study of fine-grained software modifications. *Empirical Software Engineering, An International Journal*, 11(3):369–393, 2006.

[60] Daniel M. German. A study of the contributors of postgresql. In *2006 International Workshop on Mining Software Repositories (MSR '06)*, pages 163 – 164, Shanghai, China, 2006.

[61] M. Gethers and D. Poshyvanyk. Using relational topic models to capture coupling among classes in object-oriented software systems. In *26th IEEE International Conference on Software Maintenance (ICSM'10)*, pages 1–10, Timioara, Romania, 2010.

[62] M. Gibiec, A. Czauderna, and J. Cleland-Huang. Towards mining replacement queries for hard-to-retrieve traces. In *25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*, pages 245–254, Antwerp, Belgium, 2010.

[63] T. Girba, T. Ducasse, and M. Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 40–49, Chicago, IL, 2004.

[64] N. Gold, M. Harman, Z. Li, and K. Mahdavi. Allowing overlapping boundaries in source code using a search based approach to concept binding. In *22nd IEEE International Conference on Software Maintenance (ICSM'06)*, pages 310–319, Philadelphia, PA, 2006.

[65] S. Grant, J. R. Cordy, and D. B. Skillicorn. Automated concept location using independent component analysis. In *15th Working Conference on Reverse Engineering (WCRE'08)*, pages 138–142, Antwerp, Belgium, 2008.

[66] O. Greevy, S. Ducasse, and T. Girba. Analyzing software evolution through feature views. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(6):425 – 456, 2006.

[67] J.H. Hayes, A. Dekhtyar, and S.K. Sundaram. Advancing candidate link generation for requirements tracing: the study of methods. *IEEE Transactions on Software Engineering*, 32(1):4–19, 2006.

[68] E. Hill, L. Pollock, and K. Vijay-Shanker. Exploring the neighborhood with dora to expedite software maintenance. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, pages 14–23, 2007.

[69] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *31st IEEE/ACM International Conference on Software Engineering (ICSE'09)*. Vancouver, Canada, 2009.

[70] R. Jacobs. Methods for combining experts' probability assessments. *Neural Computation*, 7(5):867–888, 1995.

[71] G. Jeong, S. Kim, and T. Zimmermann. Improving bug triage with bug tossing graphs. In *7th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2009)*, Amsterdam, The Netherlands, 2009.

[72] H. Jiang, T. Nguyen, I. X. Che, H. Jaygarl, and C. Chang. Incremental latent semantic indexing for effective, automatic traceability link evolution management. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*, L'Aquila, Italy, 2008.

[73] H. Kagdi, M.L. Collard, and J.I. Maletic. 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)*, 19(2):77–131, 2007.

[74] H. Kagdi, M. Hammad, and J. I. Maletic. Who can help me with this source code change? In *IEEE International Conference on Software Maintenance (ICSM'08)*, Beijing, China, 2008.

[75] H Kagdi, J. I. Maletic, and Bonita Sharif. Mining software repositories for traceability links. In *IEEE International Conference on Program Comprehension (ICPC'07)*, pages 145–154, Banff, Canada, 2007.

[76] H. Kagdi and D. Poshyvanyk. Who can help me with this change request? In *17th IEEE International Conference on Program Comprehension (ICPC'09)*, pages 273–277, Vancouver, British Columbia, Canada, 2009.

[77] H. Kagdi, **M. Gethers**, and D. Poshyvanyk. SE^2 model to support software evolution. In *Proc. of 27th IEEE International Conference on Software Maintenance (ICSM'11), Early Research Achievements Track*, pages 512–515, Williamsburg, Virginia, USA, 2011.

[78] Huzefa Kagdi, **M. Gethers**, Denys Poshyvanyk, and Michael Collard. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *Proc. of 17th Working Conference on Reverse Engineering (WCRE'10)*, pages 119–128, Beverly, Massachusetts, USA, 2010.

[79] Huzefa Kagdi, **M. Gethers**, Denys Poshyvanyk, and Maen Hammad. Assigning change requests to software developers. *Journal of Software: Evolution and Process*, 24(1):3–33, 2012.

[80] Huzefa Kagdi, Shehnaaz Yusuf, and J. I. Maletic. Mining sequences of changed-files from version histories. In *3rd International Workshop on Mining Software Repositories (MSR'06)*, pages 47–53, Shanghai, China, 2006. ACM Press: New York NY.

[81] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. K. Sjøberg. A systematic review of effect size in software engineering experiments. *Information and Software Technology*, 49(11-12):1073–1086, 2007.

[82] Shinji Kawaguchi, Pankaj K. Garg, Makoto Matsushita, and Katsuro Inoue. Automatic categorization algorithm for evolvable software archive. In *6th International Workshop on Principles of Software Evolution (IWPSE'03)*, pages 195–200, 2003.

[83] Ed Keenan, Adam Czauderna, Greg Leach, Jane Cleland-Huang, Yonghee Shin, Evan Moritz, **M. Gethers**, Denys Poshyvanyk, Jonathan Maletic, Jane Huffman Hayes, Alex Dekhtyar, Daria Manukian, Shervin Hussein, and Derek Hearn. Tracelab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions. In *Submitted to 34th ACM/IEEE International Conference on Software Engineering (ICSE'12), Formal Research Tool Demonstration Track*, Zurich, Switzerland, 2012.

[84] Sam Klock, **M. Gethers**, Bogdan Dit, and Denys Poshyvanyk. Traceclipse: An eclipse plug-in for traceability link recovery and management. In *Proc. of 6th ICSE 2011 International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'11)*, pages 24–30, Honolulu, Hawaii, USA, 2011.

[85] R. Kosara, C. G. Healey, V. Interrante, D. H. Laidlaw, and C Ware. Visualization viewpoints. *Computer Graphics and Applications*, 23(4):20–25, 2003.

[86] J. Kothari, T. Denton, S. Mancoridis, and A. Shokoufandeh. On computing the canonical features of software systems. In *13th IEEE Working Conference on Reverse Engineering (WCRE'06)*, October 2006.

[87] Adrian Kuhn, Stephane Ducasse, and Tudor Grba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.

[88] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *25th International Conference on Software Engineering (ICSE'03)*, pages 308–318, Portland, Oregon, 2003.

[89] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, pages 234–243, Atlanta, Georgia, 2007.

[90] Y. Liu, D. Poshyvanyk, R. Ferenc, T. Gyimthy, and N. Chrisochoides. Modeling class cohesion as mixtures of latent topics. In *25th IEEE International Conference on Software Maintenance (ICSM'09)*, pages 233–242, September 20-26 2009.

[91] M. Lormans and A. Van Deursen. Can lsi help reconstructing requirements traceability in design and test? In *10th European Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 47–56, 2006.

[92] Marco Lormans, Arie Deursen, and Hans-Gerhard Gross. An industrial case study in reconstructing requirements views. *Empirical Software Engineering, An International Journal*, 13(6):727–760, 2008. 1466714.

[93] S. Lukins, N. Kraft, and L. Etzkorn. Source code retrieval for bug location using latent dirichlet allocation. In *15th Working Conference on Reverse Engineering (WCRE'08)*, pages 155–164, Antwerp, Belgium, 2008.

[94] D. Ma, D. Schuler, T. Zimmermann, and J. Sillito. Expertise recommendation with usage expertise. In *25th IEEE International Conference on Software Maintenance (ICSM'09)*, Edmonton, Alberta, Canada, 2009.

[95] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, 1991.

[96] Jonathan I. Maletic and Andrian Marcus. Supporting program comprehension using semantic and structural information. In *23rd International Conference on Software Engineering (ICSE'01)*, pages 103–112, Toronto, Ontario, Canada, 2001. IEEE. hard copy - .pdf.

[97] A. Marcus, J.I. Maletic, and A. Sergeyev. Recovery of traceability links between software documentation and source code. *International Journal of Software Engineering and Knowledge Engineering*, 15(4):811–836, 2005.

[98] A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, 2008.

[99] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code. In *Automated Software Engineering (ASE'01)*, pages 107–114, San Diego, CA, 2001.

[100] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *25th IEEE/ACM International Conference on Software Engineering (ICSE'03)*, pages 125–137, Portland, OR, 2003.

[101] Andrian Marcus, Vaclav Rajlich, Joseph Buchta, Maksym Petrenko, and Andrey Sergeyev. Static techniques for concept location in object-oriented code. In *13th IEEE International Workshop on Program Comprehension (IWPC'05)*, pages 33–42, St. Louis, Missouri, USA, 2005.

[102] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan Maletic. An information retrieval approach to concept location in source code. In *11th IEEE Working Conference on Reverse Engineering (WCRE'04)*, pages 214–223, Delft, The Netherlands, 2004.

[103] Dominique Matter, Adrian Kuhn, and Oscar Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *6th IEEE Working Conference on Mining Software Repositories (MSR'09)*, pages 131 – 140, 2009.

[104] David McDonald and Mark Ackerman. Expertise recommender: A flexible recommendation system and architecture. In *2000 ACM Conference on Computer Supported Cooperative Work (CSCW '00)*, pages 231–240, Philadelphia, PA, 2000.

[105] C. McMillan, D. Poshyvanyk, and M. Revelle. Combining textual and structural analysis of software artifacts for traceability link recovery. In *International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'09)*, pages 41–48, Vancouver, Canada, 2009.

[106] Shawn Minto and Gail Murphy. Recommending emergent teams. In *Fourth International Workshop on Mining Software Repositories (MSR '07)*, Minneapolis, MN, 2007.

[107] A. Mockus and L.G. Votta. Identifying reasons for software changes using historic databases. In *IEEE International Conference on Software Maintenance (ICSM'00)*, pages 120–130, 2000.

[108] Audris Mockus and James Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *24th International Conference on Software Engineering (ICSE '02)*, pages 503–512, Orlando, FL, 2002.

[109] L. Moonen. Lightweight impact analysis using island grammars. In *10th International Workshop on Program Comprehension (IWPC'02)*, pages 219–228, Paris, France, 2002. Moonen02.

[110] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. Evolution patterns of open-source software systems and communities. In *International Workshop on Principles of Software Evolution (IWPSE'02)*, pages 76–85, Orlando, Florida, 2002.

[111] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery. In *18th IEEE International Conference on Program Comprehension (ICPC'10)*, pages 68–71, Braga, Portugal, 2010.

[112] Rocco Oliveto, **M. Gethers**, Gabriele Bavota, Denys Poshyvanyk, and Andrea De Lucia. Identifying method friendships to remove the feature envy bad smell (nier track). In *Proc. of 33rd ACM/IEEE International Conference on Software Engineering (ICSE'11), New Ideas and Emerging Results (NIER) Track*, pages 820–823, Honolulu, Hawaii, USA, 2011.

[113] Rocco Oliveto, **M. Gethers**, Denys Poshyvanyk, and Andrea De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery. In *Proc. of 18th IEEE International Conference on Program Comprehension (ICPC'10)*, pages 68–71, Braga, Portugal, 2010.

[114] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M.J. Harrold. An empirical comparison of dynamic impact analysis algorithms. In *IEEE/ACM International Conference on Software Engineering (ICSE'04)*, pages 776–786, 2004.

[115] M. Petrenko and V. Rajlich. Variable granularity for improving precision of impact analysis. In *17th IEEE International Conference on Program Comprehension (ICPC'09)*, pages 10–19, Vancouver, BC, Canada, 2009.

[116] Martin Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[117] D. Poshyvanyk, Y.G. Guhneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007.

[118] D. Poshyvanyk and A. Marcus. The conceptual coupling metrics for object-oriented systems. In *22nd IEEE International Conference on Software Maintenance (ICSM'06)*, pages 469 – 478, Philadelphia, PA, 2006.

[119] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimthy. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering, An International Journal*, 14(1):5–32, 2009.

[120] D. Poshyvanyk and D. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *15th IEEE International Conference on Program Comprehension (ICPC'07)*, pages 37–48, Banff, Alberta, Canada, 2007.

[121] D. Poshyvanyk, **M. Gethers**, and A. Marcus. Concept location using formal concept analysis and information retrieval. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, to appear.

[122] Jean-Pierre Queille, Jean-Francois Voidrot, Norman Wilde, and Malcolm and Munro. The impact analysis task in software maintenance: A model and a case study. In *International Conference on Software Maintenance ('94)*, pages 234 – 242, 1994.

[123] B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 27(1):58–93, 2001.

[124] X. Ren, F. Shah, F. Tip, B.G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications(OOPSLA '04)*, pages 432–448, Vancouver, BC, Canada, 2004.

[125] M. Revelle and D. Poshyvanyk. An exploratory study on assessing feature location techniques. In *17th IEEE International Conference on Program Comprehension (ICPC'09)*, pages 218–222, Vancouver, British Columbia, Canada, 2009.

[126] Meghan Revelle, **M. Gethers**, and Denys Poshyvanyk. Using structural and textual information to capture feature coupling in object-oriented software. *Empirical Software Engineering, An International Journal (EMSE)*, 16(6):773–811, 2011.

[127] M. Robillard. Automatic generation of suggestions for program investigation. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'05)*, pages 11 – 20, Lisbon, Portugal, 2005.

[128] M. P. Robillard. Topology analysis of software dependencies. *ACM Transactions on Software Engineering and Methodology*, 17(4), 2008.

[129] Gregorio Robles and Jesus M. Gonzlez-Barahona. Developer identification methods for integrated data from various sources. In *2nd International Workshop on Mining Software Repositories (MSR'05)*, pages 106–110, St. Louis, Missouri, 2005. ACM Press: New York NY.

[130] Atanas Rountev, Ana Milanova, and G. Ryder, Barbara. Points-to analysis for java using annotated constraints. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, pages 43–55, Tampa Bay, FL, USA, 2001.

[131] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *29th IEEE/ACM International Conference on Software Engineering (ICSE'07)*, pages 499–510, Minneapolis, MN, 2007.

[132] P. Runeson and M. Hst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering, An International Journal*, 14(2):131–164, 2009.

[133] M. Salah and S. Mancoridis. A hierarchy of dynamic software views: from object-interactions to feature-interactions. In *20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 72–81, Chicago, IL, 2004.

[134] M. Salah, S. Mancoridis, G. Antoniol, and M. Di Penta. Scenario-driven dynamic analysis for comprehending large software systems. In *10th IEEE European Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 71–80, 2006.

[135] Trevor Savage, Bogdan Dit, **M. Gethers**, and Denys Poshyvanyk. Topic$_{XP}$: Exploring topics in source code using latent dirichlet allocation. In *26th IEEE International Conference on Software Maintenance (ICSM'10), Formal Research Tool Demonstration*, pages 1–6, Timisoara, Romania, 2010.

[136] R. Settimi, J. Cleland-Huang, O. Ben Khadra, J. Mody, W. Lukasik, and C. DePalma. Supporting software evolution through dynamically retrieving traces to uml artifacts. In *7th International Workshop on Principles of Software Evolution (IWPSE)*, pages 49–54, Kyoto, Japan, 2004.

[137] D. Shepherd, Z. Fry, E. Gibson, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *6th International Conference on Aspect Oriented Software Development (AOSD'07)*, pages 212–224, 2007.

[138] S. Simmons, D. Edwards, N. Wilde, J. Homan, and M. Groble. Industrial tools for the feature location problem: an exploratory study. *Journal of Software Maintenance: Research and Practice*, 18(6):457–474, 2006.

[139] R. M. Sirkin. *Statistics for the social sciences*. Sage Publications, 2005.

[140] X. Song, B. Tseng, C. Lin, and M. Sun. Expertisenet: Relational and evolutionary expert modeling. In *10th International Conference on User Modeling (UM'5)*, Edinburgh, UK, 2005.

[141] R. Tairas and J. Gray. An information retrieval process to aid in the analysis of code clones. *Empirical Software Engineering, An International Journal*, 14(1):33–56, 2009.

[142] **M. Gethers**, Bogdan Dit, Huzefa Kagdi, and Denys Poshyvanyk. Integrated impact analysis for managing software changes. In *Proc. of 34th ACM/IEEE International Conference on Software Engineering (ICSE'12)*, Zurich, Switzerland, 2012.

[143] **M. Gethers**, H. Kagdi, B. Dit, and D. Poshyvanyk. An adaptive approach to impact analysis from change requests to source code. In *Proc. of 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*, pages 540–543, Lawrence, Kansas, USA, 2011.

[144] **M. Gethers**, R. Oliveto, D. Poshyvanyk, and A. De Lucia. On integrating orthogonal information retrieval methods to improve traceability link recovery. In *Proc. of 27th IEEE International Conference on Software Maintenance (ICSM'11)*, pages 133–142, Williamsburg, Virginia, USA, 2011.

[145] **M. Gethers** and Denys Poshyvanyk. Using relational topic models to capture coupling among classes in object-oriented software systems. In *Proc. of 26th IEEE International Conference on Software Maintenance (ICSM'10)*, pages 1–10, Timisoara, Romania, 2010.

[146] **M. Gethers**, Trevor Savage, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. Codetopics: Which topic am i coding now? In *Proc. of 33rd ACM/IEEE International Conference on Software Engineering (ICSE'11), Formal Research Tool Demonstration*, pages 1034–1036, Honolulu, Hawaii, USA, 2011.

[147] K. Tian, M. Revelle, and D. Poshyvanyk. Using latent dirichlet allocation for automatic categorization of software. In *6th IEEE Working Conference on Mining Software Repositories (MSR'09)*, pages 163–166, Vancouver, British Columbia, Canada, 2009.

[148] P. Tonella. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Transactions on Software Engineering*, 29(6):495–509, 2003. Tonella03.pdf.

[149] Masateru Tsunoda, Akito Monden, Takeshi Kakimoto, Yasutaka Kamei, and Ken-ichi Matsumoto. Analyzing oss developers' working time using mailing lists archives. In *2006 International Workshop on Mining Software Repositories (MSR '06)*, pages 181 – 182, Shanghai, China, 2006.

[150] R. Vaclav. A model for change propagation based on graph rewriting. In *International Conference on Software Maintenance (ICSM '97)*, pages 84–91, Bari, ITALY, 1997. IEEE. Vaclav97.pdf.

[151] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *30th International Conference on Software Engineering (ICSE08)*, pages 461–470, Leipzig, Germany, 2008.

[152] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *4th IEEE International Workshop on Mining Software Repositories (MSR'07)*, pages 1–8, Minneapolis, MN, 2007.

[153] Peter Weissgerber, Mathias Pohl, and Michael Burch. Visual data mining in software archives to detect how developers work together. In *Fourth International Workshop on Mining Software Repositories (MSR'07)*, Minneapolis, USA, 2007.

[154] N. Wilde, J.A. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *IEEE International Conference on Software Maintenance (ICSM'92)*, pages 200–205, Orlando, FL, 1992.

[155] F.G. Wilkie and B.A. Kitchenham. Coupling measures and change ripples in c++ application software. *The Journal of Systems and Software*, 52:157–164, 2000.

[156] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnel, and A. Wesslen. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Press, Amsterdam, 1999.

[157] Y. Ye and Gerhard Fischer. Reuse-conducive development environments. *Journal Automated Software Engineering*, 12(2):199–235, 2005.

[158] L. Yu and S. Ramaswamy. Mining cvs repositories to understand open-source project developer roles. In *4th International Workshop on Mining Software Repositories (MSR'07)*, page 8, 2007.

[159] Z. Yu and V. Rajlich. Hidden dependencies in program comprehension and change propagation. In *9th IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 293–299, Toronto, Canada, 2001. Yu01.pdf.

[160] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. Sniafl: Towards a static non-interactive approach to feature location. *ACM Transactions on Software Engineering and Methodologies (TOSEM)*, 15(2):195–226, 2006.

[161] T. Zimmermann, A. Zeller, P. Weigerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.