Improving Information Retrieval Bug Localisation Using Contextual Heuristics

Thesis

Improving Information Retrieval Based Bug Localisation Using Contextual Heuristics

Tezcan Dilshener M.Sc.

A thesis submitted to

The Open University

in partial fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing

Department of Computing and Communications
Faculty of Mathematics, Computing and Technology
The Open University

September 2016

Abstract

Software developers working on unfamiliar systems are challenged to identify where and how high-level concepts are implemented in the source code prior to performing maintenance tasks. Bug localisation is a core program comprehension activity in software maintenance: given the observation of a bug, e.g. via a bug report, where is it located in the source code?

Information retrieval (IR) approaches see the bug report as the query, and the source files as the documents to be retrieved, ranked by relevance. Current approaches rely on project history, in particular previously fixed bugs and versions of the source code. Existing IR techniques fall short of providing adequate solutions in finding all the source code files relevant for a bug. Without additional help, bug localisation can become a tedious, time-consuming and error-prone task.

My research contributes a novel algorithm that, given a bug report and the application's source files, uses a combination of lexical and structural information to suggest, in a ranked order, files that may have to be changed to resolve the reported bug without requiring past code and similar reports.

I study eight applications for which I had access to the user guide, the source code, and some bug reports. I compare the relative importance and the occurrence of the domain concepts in the project artefacts and measure the effectiveness of using only concept key words to locate files relevant for a bug compared to using all the words of a bug report.

Measuring my approach against six others, using their five metrics and eight projects, I position an effected file in the top-1, top-5 and top-10 ranks on average for 44%, 69% and 76% of the bug reports respectively. This is an improvement of 23%, 16% and 11% respectively over the best performing current state-of-the-art tool.

Finally, I evaluate my algorithm with a range of industrial applications in user studies, and found that it is superior to simple string search, as often performed by developers. These results show the applicability of my approach to software projects without history and offers a simpler light-weight solution.

Acknowledgements

I express my thanks and sincere gratitude to my supervisors, Dr. Michel Wermelinger and Dr. Yijun Yu, for believing in my research and in my capabilities. Their dedicated support, direction and the guidance made my journey through the research forest a conquerable experience.

Simon Butler for his assistance in using the JIM tool and for the countless proof readings of my work. As my research buddy, he always managed to motivate and encourage me.

Prof. Marian Petre for sharing her valuable experience during online seminar sessions and organising the best departmental conferences.

Prof. Hongyu Zhang for kindly providing BugLocator and its datasets, Ripon Saha for the BLUiR dataset, Laura Moreno for the LOBSTER dataset and Chu-Pan Wong for giving me access to the source code of his tool BRTracer.

Jana B. at our industrial partner, a global financial IT solutions provider located in southern Germany, for providing the artefacts to one of their proprietary application that I used throughout my research and their input on diverse information required.

Markus S., Georgi M. and Alan E. for being the proxy at their business locations while I conducted my user study. Also to Markus S. at our industrial partner for his countless hours of discussions on patterns in software applications and source code analysis.

My wife Anja for her endless patience, understanding and respect. With her love, encouragement and dedicated coaching, I stood the emotional up and down phases of my research, thus completed my Ph.D. studies.

To my two daughters, Denise and Jasmin for allowing their play time with me to be spent on my research instead.

Last but not least, thank you to all those who wish to remain unnamed for contributing to my dedication.

Finally, there are no intended contradictory intentions implied to any individual or establishment.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As current software applications become more acceptable and their lifespan stretches beyond their estimated life expectancies, they evolve into legacy systems. These applications become valuable strategic assets to companies and require ongoing maintenance to continue providing functionality for the business. To reduce the cost of ownership, such companies may turn towards third party developers to take over the task of maintaining these applications. These consultants, due to the impermanent nature of their job, could introduce high-turn around and cause the knowledge of the applications to move further away from its source.

Each time a new developer is designated to perform a maintenance task, the associated high learning curve results in loss of precious time, incurring additional costs. As the documentation and other relevant project artefacts decay, to understand the current state of the application before implementing a change, the designated software developer has to go through the complex task of understanding the code.

In this Chapter, I describe the current challenges faced by software developers when performing maintenance tasks and my motivation for conducting this research. Henceforth, I show examples of the information retrieval process in software maintenance and identify the limitations of the existing approaches. Subsequently, I define the hypothesis that I investigate and the research questions I aim to answer to address some of the highlighted limitations. Finally, I summarise by giving an overview of the chapters contained in this thesis.

1.1 Academic Motivation: Software Maintenance Challenges

IEEE standards 1219 define software maintenance as the task of modifying a software application after it has been delivered for use, to correct any unforeseen faults, apply improvements or enhance it for other environments. Corbi (1989) identified that program comprehension — i.e. the process of understanding a program — is an important activity a developer must undertake before performing any maintenance tasks.

Program comprehension forms a considerable component of the effort and costs of software maintenance. Between 40–60% of software maintenance efforts are dedicated to understanding the software application (Pigoski, 1996), and the cost of change control — i.e. the process of managing maintenance — is estimated to make up 50–70% of the project life cycle costs, (Hunt *et al.*, 2008).

Prior to performing a maintenance task, the designated developer has to become familiar with the application in concern. If unfamiliar, she has to read the source code to understand how those program elements, e.g. source code class files (see Appendix A), interact with each other to accomplish the described use case. Even the experienced programmers face a tremendous learning curve while understanding the applications domain when they move to another area within their current projects, (Davison *et al.*, 2000). It is concluded that during maintenance there is a tendency of 60–80% of time being spent on discovering and argued that it is much harder to understand the problem than to implement a solution.

During application development, it would have been ideal for program comprehension to use the same words found in the requirements documentation when declaring program identifier names, i.e. source code entities (Appendix A). However, the developers often choose the abbreviated form of the words and names found in the text documentation as well as use nouns and verbs in compound format to capture the described actions. In addition, the layered multi-tier architectural guidelines, (Parnas, 1972), cause the program elements implementing the domain concept (Appendix A) to be scattered across the application and creates challenges during maintenance when linking the application source code to the text documentation.

Software maintainers can face further barriers during program comprehension because the knowledge recorded in source code and documentation can decay. Each change cycle causes degradation amongst the project artefacts like program elements, architecture and its documentation, if the changes are not propagated to these artefacts, (Lehman, 1980). The study conducted by Feilkas *et al.* (2009) report that between 70–90% decay of knowledge is caused by flaws in the documentation and 9–19% of the implementations failed to conform to the documented architecture.

1.2 Industry Motivation: Industry Needs in Locating Bugs

As an independent software development consultant, I perform maintenance tasks in business applications at different companies and sometimes rotate between projects within the same company. Each time I start work on a project new to me, I spend considerable amount of time trying to comprehend the application and come up to speed in locating relevant source code files prior to performing maintenance tasks.

In general, when I receive a new task to work on, e.g. documented as a bug report, I perform a search for the files to be changed using the tools available in my development environment. Firstly, these tools force me to specify precise search words otherwise the results contain many irrelevant files or no results are returned if I enter too many words. Secondly, even when the search words are precise, the results are displayed without an order.

As a new project team member, unfamiliar with the domain terminology (i.e. concepts) and project vocabulary, I am usually challenged to identify the key words or concepts to use on the search query as well as forced to analyse a long list of files retrieved from an application which I am unfamiliar with.

I often wonder whether the rich content available in bug reports can be utilised during the search and the resulting list of files are displayed in a ranked order based on relevance from most to least important. Subsequently, I can focus on the files listed at the top of the list and avoid the time consuming task of analysing all of the files. To reduce the time it takes in locating relevant source files and to improve the reliability of the search results, in my research I investigate to see how additional sources of information found in project artefacts can be leveraged and the search results are presented in an ordered list to provide benefit for the developers.

1.3 An Example of Information Retrieval Process

In a typical information retrieval (IR) approach for traceability between an application's source code and its textual documentation like the user guide, the current techniques first analyse the project artefacts and build an abstract high level representation of an application in a repository referred as the corpus where the program elements implementing the concepts can be searched. The existing IR techniques extract the traceability information from the program elements by parsing the application's source code files and store the extracted in-

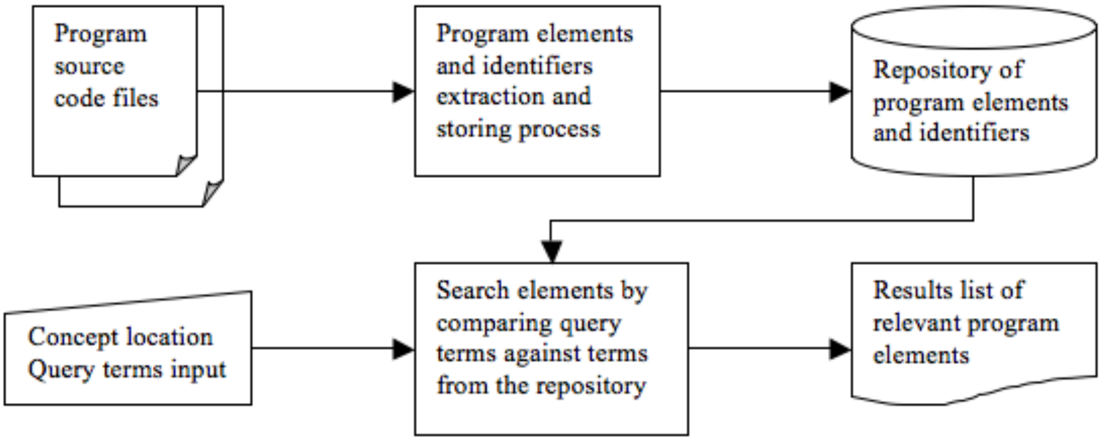

Program source code files
Program elements and identifiers extraction and storing process
Repository of program elements and identifiers
Concept location Query terms input
Search elements by comparing query terms against terms from the repository
Results list of relevant program elements

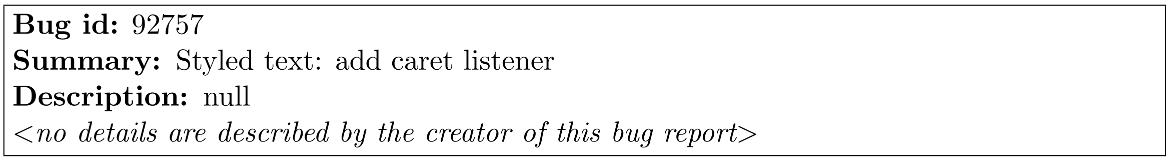
Figure 1.1: Information retrieval repository creation and search process

formation in the corpus. During the extraction process the identifier names are transformed into individual words (i.e. terms) according to known Object Oriented Programming (OOP) coding styles like the camel case naming pattern where, for example, identifier *standAloneRisk* is split into three single words as *stand*, *alone* and *risk*. After this transformation, the resulting terms are stored in the repository with a reference to their locations in the program elements as illustrated at the top half of Figure 1.1.

Once the corpus is created, a developer initiates the concept location (Appendix A) process by entering the query terms representing the concepts being searched in a user interface. The query terms may come from user guide documents or bug reports. The IR method performs the search by comparing the terms entered on the query against the terms found in the underlying repository. The matching score is calculated based on lexical similarity or probabilistic distance between the terms, explained in Chapter 2, depending on the underlying IR method. Finally the results are displayed to the developer in a list ranked by their relevance as defined in the IR model.

Now I will go through with examples to illustrate the limitations of the existing approaches. One of the projects I use to evaluate my approach, as we will see in Chapter 3 and Chapter 4, is a graphical user interface (GUI) project called SWT[1]. In project SWT, bug report #92757 has a very terse description (see Figure 1.2) and for a developer who is unfamiliar with SWT, exposes the question of what concept terms to use when searching for the relevant files prior to performing a change to resolve the reported bug. The only concept this bug report refers to is the GUI domain concept of *text*, which results in 142 source files

[1] http://www.eclipse.org/swt/



Bug id: 92757
Summary: Styled text: add caret listener
Description: null
<no details are described by the creator of this bug report>

Figure 1.2: Bug report with a very terse description in SWT project

to be returned when queried. To reduce the number of files the developer has to analyse, alternative terms like *style* or *caret* found in the bug report summary may be used. However *style* returns 55 and *caret* 13 files of which none of them is the relevant one already changed to resolve the reported bug.

Since this is a closed bug report, e.g. resolved, looking at the content of the affected file revealed that it implements the GUI domain concept *widget*. However, when the concept *widget* is used as the search query term, 130 files are returned on the resulting list.

To address the challenges highlighted by this example, i.e. selecting search terms that result in reduced number of files to be investigated, the current research approaches use all the vocabulary available in the bug report during the search (Zhou *et al.*, 2012a; Wong *et al.*, 2014; Saha *et al.*, 2013; Wang and Lo, 2014; Ye *et al.*, 2014). Then the results are enhanced by considering available historical data, e.g. previously changed source code files and similar bug reports. Finally, the results are presented in a list ranked by order of importance from high to low based on the relevance to the entered search terms.

However the existing studies conclude that such supplemental information sources, i.e. past history and similar bugs, do not guarantee improved results. For example, the two existing tools, BugLocator (Zhou *et al.*, 2012a) and BRTracer (Wong *et al.*, 2014), still rank the relevant file for this bug report at 88th and 75th position in the result list respectively. We will see in Chapter 4 that my proposed approach ranks the same file in the 5th position.

In general bug reports describe the observed erroneous behaviour and may contain detailed descriptions like stack trace information, which lists a sequence of files that were executing at the time of the error (Bettenburg *et al.*, 2008). To improve the performance of the bug localisation approaches, current research also considered evaluating stack trace information included in the bug reports (Wong *et al.*, 2014; Moreno *et al.*, 2014).

One of the ways to utilise the stack trace is to consider the file names listed in the trace. For example, in Figure 1.3 AspectJ[2] bug #158624 contains a detailed stack trace of an exception (*UnsupportedOperationException*) produced by an erroneous condition referred as *"the bug*

[2] http://eclipse.org/aspectj/

Bug id: 158624
Summary: Compiler Error: generics and arrays
Description: OK, not sure what to report here or what info you need, but here's the set up, message, and erroneous class. I don't understand the errors from the compiler enough to parse down the erroneous file to something that contains only the bug, but I could if direction were given. Here's my set up: Eclipse SDK Version: 3.2.0 Build id: M20060629-1905 With AJDT: Eclipse AspectJ Development Tools Version: 1.4.1.200608141223 AspectJ version: 1.5.3.200608210848 Here's **the bug dump** from the compiler inside Eclipse:
java.lang.**UnsupportedOperationException** at
org.aspectj.weaver.UnresolvedType.parameterize(UnresolvedType.java:221) at
org.aspectj.weaver.***ResolvedMemberImpl***.parameterize(ResolvedMemberImpl.java:680) at
org.aspectj.weaver.ResolvedMemberImpl.parameterize(ResolvedMemberImpl.java:690) at
org.aspectj.weaver.ReferenceType.getDeclaredMethods(ReferenceType.java:508) at
org.aspectj.weaver.ResolvedType$4.get(ResolvedType.java:226) at
...
<rest of the stack trace is intentionally not displayed>

Figure 1.3: AspectJ project's bug report description with stack trace information

dump" in the description. AspectJ is another project I use to evaluate the performance of my approach and compare it against the other tools that also used it. The source code file name, *ResolvedMemberImpl.java* (listed after the second *"at"* in the stack trace) is one of the the relevant files modified to solve this bug.

When the search results between the tool that takes advantage of the stack trace available in bug reports (Wong *et al.*, 2014) and the other tool that does not (Zhou *et al.*, 2012a) are compared, the ranking of the relevant file improves from 16th to 6th position in the result list. Nevertheless the developer still has to analyse 5 irrelevant files prior to finding the relevant one. The reasons for the low ranking in these tools is that existing approaches (Wong *et al.*, 2014; Moreno *et al.*, 2014) consider all the files found in the stack trace list, which eventually causes irrelevant files to be ranked more important than the relevant ones. Again we will see that my approach ranks *ResolvedMemberImpl.java* at 2nd position in the results list.

Further analysis showed that words in certain positions in the bug reports reveal file names. For example, as illustrated in Figure 1.4, the first word in the summary field of SWT bug #79268 is the relevant file name. The existing tools rank this file at 21st and 11th position respectively. One of the reasons for this is that the word *Program* is considered to be too ambiguous and gets a lower score. Although the current IR approaches also detect the file names available in the bug report, they continue to score other files based on word similarity, thus resulting in irrelevant files that have more matching words with the words from the bug report to get ranked higher. We will see that my approach ranks this file at the 1st position

Bug id: 79268
Summary: Program API does not work with GNOME 2.8 (libgnomevfs-WARNING)
Description: I200411170800-gtk Not sure what triggers it, neither who is doing it. I get the following stderr output once in a while: (&amp;lt;unknown&amp;gt;:27693): libgnomevfs-WARNING **: Deprecated function. User modifications to the MIME database are no longer supported. In my development workbench, the output reads: (Gecko:11501): libgnomevfs-WARNING **: Deprecated function. User modifications to the MIME database are no longer supported. So I suspect the mozilla/gecko library is doing this, hence I punted it to SWT. Probably nothing we can do much about, but here's the bug anyway.

Figure 1.4: Ambiguous bug report description found in SWT

in the result list.

1.4 Existing Work and The Limitations

Early attempts to aid developers in recovering traceability links between source code files and textual documentation used IR methods like the Vector Space Model (VSM), Salton and Buckley (1988), and managed to achieve high precision (the accuracy of the results, Marcus and Maletic 2003) or high recall (the completeness of the results, Antoniol *et al.* 2002). Nevertheless, the IR approaches do not consider words that are strongly related via structural information, e.g. OOP inheritance, available in software programs to be relevant and thus still perform poorly in some cases (Petrenko and Rajlich, 2013).

Further research recognised the need for combining multiple analysis approaches on top of IR to support program comprehension (Gethers *et al.*, 2011). To determine the starting points in investigating relevant source code files for maintenance work, techniques combining dynamic (Wilde and Scully, 1995) and static (Marcus *et al.*, 2005) analysis have been exploited (Poshyvanyk *et al.*, 2007; Eisenbarth *et al.*, 2001).

The application of dynamic analysis builds upon the trace information obtained by executing application usage scenarios. However, if a maintenance task describes a scenario that may not be executable, e.g. a new non-existing feature, then it is unsuitable for dynamic analysis (Poshyvanyk *et al.*, 2007).

On the other hand, to cover a broader set of program elements, one of the techniques the static analysis utilises is a static call-graph where the interactions, i.e. call relations, between the source code files are organised. However the call-graphs usually contain many dependencies and very deep subtrees (i.e. nodes/edges) that make them impractical for search and navigation purposes.

To compensate for the impractical call-graph weaknesses, the previous approaches utilised pruning logic to selectively remove the nodes during clustering (i.e. grouping) the call relations. However, one of the challenges pruning exposes is that a source code file may be moved to a different group/cluster due to the underlying pruning logic and cause inconsistencies when searching for the files implementing a concept.

Recent approaches utilise additional sources of information, e.g. previously fixed bug reports (i.e. similar bugs) and number of times a source file is fixed (i.e. version history), to boost the scoring of the underlying IR model. However, it is claimed that considering similar bug reports earlier than 14 days add no value (Nichols, 2010) and version history older than 20 days decrease the performance (Wang and Lo, 2014). Besides a large software project or one with a long history may require time-consuming analysis, making these approaches impracticable (Rao and Kak, 2011).

To overcome these limitations, recent studies utilised segmentation (Wong *et al.*, 2014) and stack trace analysis (Moreno *et al.*, 2014) as an effective technique to boost performance of IR models. During segmentation the source code files are divided into smaller chunks containing reduced number of words to provide more advantage during the search. However, the effectiveness of segmentation known to be sensitive to the number of words being included in each segment (Wong *et al.*, 2014), thus may reduce the relevance of a segment for the concept being searched. Moreover these approaches fail to consider non-application specific files found in the stack trace, hence the precision of the results deteriorate.

In general the limitations in the current literature can be summarised as follows.

1. Due to poor VSM performance combination of multiple techniques are required.

2. Dynamic analysis improves VSM but fails to consider all the relevant files.

3. Static analysis using application call-graph results in many irrelevant files.

4. Clustering first and then pruning cause relevant files to be removed.

5. Using similar bug reports earlier than 14 days provide no contribution.

6. Past history requires analysis of huge data thus time consuming and impractical.

7. Considering all files listed in the stack traces deteriorates the performance.

8. Segmentation causes files to be divided inefficiently thus results in loss of info.

In summary, current state-of-the-art IR approaches in bug localisation rely on project history, in particular previously fixed bugs and previous versions of the source code. Existing studies (Nichols, 2010; Wang and Lo, 2014) show that considering similar bug reports up to 14 days and version history between 15—20 days does not add any benefit to the use of IR alone. This suggests that the techniques can only be used where great deal of maintenance history is available, however same studies also show that considering history up to 50 days deteriorates the performance.

Besides, Bettenburg *et al.* (2008) argued that a bug report may contain a readily identifiable number of elements including stack traces, code fragments, patches and recreation steps each of which should be treated separately. The previous studies also show that many bug reports contain the file names that need to be fixed (Saha *et al.*, 2013) and that the bug reports have more terms in common with the affected files, which are present in the names of those affected files (Moreno *et al.*, 2014).

Furthermore, the existing approaches treat source code comments as part of the vocabulary extracted from the code files, but the comments tend to be written in sublanguage of English and due to their imperfect nature, i.e. terse grammar, they may deteriorate the performance of the search results (Etzkorn *et al.*, 2001; Arnaoudova *et al.*, 2013).

The hypothesis I investigate is that *superior results can be achieved without drawing on past history by utilising only the information, i.e. file names, available in the current bug report and considering source code comments, stemming, and a combination of both independently, to derive the best rank for each file.*

My research seeks to offer a more efficient and light-weight IR approach, which does not require any further analysis, e.g. to trace executed classes by re-running the scenarios described in the bug reports. Moreover I aim to provide a simple usability, which contributes to an *ab-initio* applicability, i.e. from the very first bug report submitted for the very first version and also be applied to new feature requests.

1.5 Research Questions

To address my hypothesis, I first undertake a preliminary investigation of eight applications to see whether vocabulary alone provides a good enough leverage for maintenance. More precisely I am interested in comparing the vocabularies of project artefacts (i.e. text documentation, bug reports and source code) to determine whether (1) the source code identifier names properly reflect the domain concepts in developers' minds and (2) identifier names can be efficiently searched for concepts to find the relevant files for implementing a given bug report. Thus my first research question (RQ) is follows.

RQ1: Do project artefacts share domain concepts and vocabulary that may aid code comprehension when searching to find the relevant files during software maintenance?

Insights to RQ1 revealed that despite good conceptual alignment among the artefacts, using concepts when searching for the relevant source files result in low precision and recall. I was able to improve the recall by simply using all of the vocabulary found in the bug report but precision remains low. To improve the suggestion of relevant source files during bug localisation, I conclude that heuristics based on words in certain key positions within the context of bug reports be developed.

Current state-of-the-art approaches for Java programs (Zhou *et al.*, 2012a; Wong *et al.*, 2014; Saha *et al.*, 2013; Wang and Lo, 2014; Ye *et al.*, 2014) rely on project history to improve the suggestion of relevant source files. In particular they use similar bug reports and recently modified files. The rationale for the former is that if a new bug report $\boldsymbol{x}$ is similar to a previously closed bug report $\boldsymbol{y}$, the files affected by $\boldsymbol{y}$ may also be relevant to $\boldsymbol{x}$. The rationale for the latter is that recent changes to a file may have led to the reported bug. However, the observed improvements using the history information have been small. I thus wonder whether file names mentioned in the bug report descriptions can replace the contribution of historical information in achieving comparable performance and ask my second research question as follows.

RQ2: Can the occurrence of file names in bug reports be leveraged to replace project history and similar bug reports to achieve improved IR-based bug localisation?

Previous studies performed by Starke *et al.* (2009) and Sillito *et al.* (2008) reveal that text-based searches available in current integrated development environments (IDE) are inadequate because they require search terms to be precisely specified otherwise irrelevant or no results are returned. It is highlighted that large search results returned by the IDE tools cause developers to analyse several files before performing bug-fixing tasks.

Henceforth, interested to find out how developers perceive the search results of my approach where a ranked list of candidate source code files that may be relevant for a bug report at hand during software maintenance, I ask my third research question as follows.

RQ3: How does the approach perform in industrial applications and does it benefit developers by presenting the results ranked in the order of relevance for the bug report at hand?

The added substantial value of my proposed approach compared to existing work is that it does not require past information like version history or similar bug reports that have been closed, nor the tuning of any weight factors to combine scores, nor the use of machine learning.

1.6 Overview of the Chapters

This thesis is organised as follows: Chapter 2 describes the current research efforts related to my work, Chapter 3 describes the work on analysing the occurrence of concepts in projects artefacts, Chapter 4 illustrates how file names can be leveraged to aid in bug localisation, Chapter 5 presents the results of a user study and Chapter 6 concludes with future recommendations.

Chapter 2. Landscape of code retrieval: Current research has proposed automated concept and bug location techniques. In this Chapter, I describe the relevant research approaches to introduce the reader into the landscape of code retrieval and summarise the limitations of the existing approaches that my research aims to address.

Chapter 3. Vocabulary analysis: In my previously published work (Dilshener and Wermelinger, 2011), which partially contributes to this Chapter, I have identified the key concepts in a financial domain and searched them in the application implementing these concepts using the concepts referenced in bug reports, however the number of relevant files found, i.e. precision, was very low. I have learnt the bug report document descriptions are very terse, action oriented and the unit of work often stretches beyond single concept implementations.

In this Chapter, I address my first research question by studying eight applications for which I had access to the user guide, the source code, and some bug reports. I compare the relative importance of the domain concepts, as understood by developers, in the user manual and in the source code of all eight applications. I analyse the vocabulary overlap of identifiers among the project artefacts as well as between the source code of different applications.

I present my novel IR approach, which directly scores each current file against the given bug report by assigning a score to a source file based on where the search terms occur in the source code file, i.e. class file names or identifiers. The logic also treats the comments and word stemming independently form each other when assigning a score.

Moreover, I search the source code files for the concepts occurring in the bug reports and vary the search to use all of the words from the bug report, to see if they could point developers to the code places to be modified. I compare the performance of the results and discuss the implications for maintenance.

Chapter 4. Heuristic based positional scoring: In my previously published work (Dilshener *et al.*, 2016), which partially contributes to this Chapter, I analysed to see how the file names occurring in bug reports can be leveraged to improve the performance without requiring past code and similar bug reports.

In this Chapter, I address my second research question by extending my approach presented in Chapter 3 to score files also based on where the search terms located in the bug report, i.e. the summary or stack trace. I treat each bug report and file individually, using the summary, stack trace, and file names only when available and relevant, i.e. when they improve the ranking. Subsequently, I compare the performance of my approach to eight others, using their own five metrics on their own projects and succeeded in placing an affected file among the top-1, top-5 and top-10 files for 44%, 69% and 76% of bug reports, on average.

I also look more closely at the contribution of past history, in particular of considering similar bug reports, and the contribution of IR model VSM, compared to a bespoke variant. I found that VSM is a crucial component to achieve the best performance for projects with a larger number of files that makes the use of term and document frequency more meaningful, but that in smaller projects its contribution is rather small.

Chapter 5. User case study: In this Chapter, I address my third research question by reporting on user studies conducted at 3 different companies with professional developers. My approach achieved to place at least one file on average for 88% of the bug reports into top-10 also with different commercial applications. Additionally developers stated that since most of the relevant files were positioned in the top-5, they were able to avoid the error prone tasks of browsing long result lists.

Chapter 6. Conclusion and future directions: After describing the key findings of my research and reflecting upon the results I presented, this Chapter highlights the conclusions I draw from my research and end by emphasising its importance for practitioners and researchers.

1.7 Summary

In my research, I embarked on a journey to improve bug localisation in software maintenance, which I describe in this thesis. Based on current literature and my industrial work experience, developers known to search for relevant source code files using the domain concepts when performing maintenance tasks to resolve a given a bug report. Although the concepts may occur in project artefacts and correlate strongly between the textual artefacts, e.g. user guide, and the source code, they may not be effective during bug localisation.

To improve the performance of the results, current research has recognised the need to use additional information sources, like project history. However, certain relevant files still can not be located. I argue that in addition to a bug report containing specific terms that support search (i.e. words that may match those used in source files), the bug reports also often include contextual information (i.e. file names) that further supports (refines) the search. Thus leveraging heuristics based on the contextual information, i.e. file names in certain positions of the bug report summary and of the stack trace, are the main contributions of my research.

Research shows that techniques to improve program understanding also have practical relevance and hence my work contributes to that area as well. Firstly, this research may benefit industrial organisations by reducing the programmer's time spent on program comprehension so that cost of implementing modifications is justifiable. Secondly, the published efforts can be used to educate the next generation of software developers who are designated to take on the challenges of maintaining applications.

Chapter 2

Landscape of Code Retrieval

In this Chapter, I describe in detail the existing research approaches in information retrieval and summarise their limitations. I highlight some of the challenges associated with identifying all the program elements relevant for a maintenance task and discuss how the challenges are addressed in current research.

According to existing literature, software maintenance tasks are documented as change requests, which are often written using the domain vocabulary or better known as domain concepts, (Rajlich and Wilde, 2002; Marcus and Haiduc, 2013). The developers who are assigned to perform a change task would need to perform concept location tasks to find the program elements implementing the concepts relevant for the task at hand.

Additionally, change requests may describe an unexpected and unintended erroneous behaviour, known as bugs of a software system. Identifying where to make changes in response to a bug report is called bug localisation where the change request is expressed as a bug report and the end goal is to change the existing code to correct an undesired behaviour of the software (Moreno *et al.*, 2014).

In this Chapter, Section 2.1 introduces the concept assignment process during program comprehension and the concept location activities while performing software maintenance tasks. In Section 2.2, I present some of the popular IR techniques used to locate the program elements implementing the concepts. Section 2.3 introduces the concept location techniques used in bug localisation. In Section 2.4, the current state-of-the-art tools against which I compare my approach are described. Section 2.5 summarises the existing literature on user studies relevant to my research. I conclude this Chapter in Section 2.6 by summarising the gaps which inspired me to conduct this research.

2.1 Concept Assignment Problem

A concept is a phrase or a single term that describes the unit of human knowledge (thought) existing within the domain of the software application (Rajlich and Wilde, 2002). Biggerstaff *et al.* (1993) argued that comparison between computer and human interpretation of concepts confronts us with human oriented terms, which may be informal and descriptive, e.g. *"deposit amount into savings account"*, against compilation unit terms which are formal and structured, e.g. *"if (accountType == 'savings') then balance += depositAmount"*. Biggerstaff *et al.* (1993) refer to discovering human oriented concepts and assigning them to their counterparts in the applications source code as "*the concept assignment problem*".

In addition to human vs. computational concept characteristics, Marcus *et al.* (2005) identified two types of implementation categories to describe the characteristics of program elements (i.e. source code class files, methods etc.) implementing the concepts in an OOP language like Java. The *composite* functionality refers to a group of source code files and their referenced code files that implement a concept. The *local* functionality refers to the files implementing the functionality of a concept in one file. For example, in a bank account management application, concept of "*account balance*" may be implemented in the account source code file, however the concept of "*balance transfer*" may be implement in multiple files.

A detailed study by Nunes *et al.* (2011), evaluated the implementation of the concepts by looking at their underlying characteristics with regards to their interaction (how they communicate with other class files), dependency (between other classes) and modularity (where they are located). The concept implementations are classified under the following six categories:

1. Multi partition: implementation is scattered in several modules (e.g., classes, methods).

2. Overly communicative: a set of class files implementing a concept heavily communicates with another set.

3. Sensitive: a method implementing a concept different to the one implemented by its class file.

4. Lattice: where concepts partially affect one another like logging and debugging.

5. Overlapping: sharing program elements.

6. Code clone: duplicating program elements.

Nunes *et al.* (2011) found these categories are correlated among each other based on two relations, (1) *can be related*; representing the perspective of where the concept is viewed from, e.g. overlapping vs overly communicative, and (2) *can influence*; referring to how dependency between each other can result in the outcome of the other, e.g. Multi partition vs. code clone. Finally, Nunes *et al.* (2011) observed the common mistakes made by software developers during program comprehension due to these characteristics of the concepts as follows:

1. Lack of explicit references between class files.

2. Strong dependency causes confusion over what class represents what concept.

3. Difficulty to assign a concept due to mixture of its implementation.

4. Confusion over a block of code in a class not related to the main concept of the class.

5. Classes implementing multiple concepts are mapped to only one concept.

6. Missing concept functionality in packaging, focus on behaviour of methods.

Although Nunes *et al.* (2011) acknowledged that certain assumptions like bias due to programmer experience as threats to the validity of the results, the study reveals that in certain applications due to naming conventions multiple program elements may seem to be implementing two independent concepts where as they are the same concept. Conversely, two program elements may seem to be implementing one concept but due to OOP language characteristics, e.g. overloading, they may actually be two independent concepts. In a further study, Deißenböck and Raţiu (2006) claimed that source code files may reflect only a fraction of the concepts being implemented accurately and argued that one cannot solely rely on the source code as the single source of information during concept location.

2.2 Information Retrieval as Conceptual Framework

Information retrieval deals with finding the relevant set of documents for a given query. In software maintenance, the project artefacts like the application's source code files, i.e. classes, and sometimes the search query are treated as the documents of the IR system. IR systems use internal models like the Vector Space Model (VSM) and the Latent Semantic Analysis (LSA) to aid in concept location during software maintenance. The internal model of an IR system defines certain configurable settings for example whether the relevance and ranking of the terms, i.e. words existing in the set of documents, is calculated based on the term frequency or term relations.

2.2.1 Evaluation Metrics

The results of an IR system are measured by precision, recall and in some cases by F-measure:

- The recall measures the completeness of the results and is calculated as the ratio of the total number of relevant documents over the number of documents retrieved.
- The precision measures the accuracy of the results and calculated as the ratio of number of relevant document assignments over the total number of documents retrieved.
- F-measure is used as a single quality measure and calculated as the harmonic mean of precision and recall since it combines both.

In VSM, each document and each query is mapped onto a vector in a multi-dimensional space, which is used for indexing and relevance ranking. The document vectors are ranked, according to some distance function, to the query vectors. The similarities are obtained by a well-known classic metric called the term frequency (tf) and inverse document frequency (idf) introduced by Salton and Buckley (1988). The tf is the number of times a term occurs in a document and the idf is the ratio between the total numbers of documents over the number of documents containing the term. The idf is used to measure if a term occurs more or less across a number of documents. The product of these two measures (i.e. tf * idf) is used to score the weighting of a term in a given collection of terms.

2.2.2 Vector Space Model and Latent Semantic Analysis

Antoniol *et al.* (2000a) demonstrated recovering traceability links between the source code files and its documentation by applying VSM. The aim of the study was to see if the source code class files could be traced back to the functional requirements. The study compared two different retrieval methods VSM and probabilistic. In VSM tf/idf score is used to calculate the similarity whereas probabilistic model computes the ranking scores as the probability that a document is related to the source code component (Antoniol *et al.*, 2000b). The authors concluded that semi-automatically recovering traceability links between code and documentation is achievable despite the fact that the developer has to manually analyse a number of information sources to obtain high recall values. Capobianco *et al.* (2009) argued that VSM has its limitations in that the similarity measure only takes into account the terms that precisely match between the search terms and the terms mapped onto the vector space. According to Capobianco *et al.* (2009), large size documents produce poor similarity values due to increased dimensions of the search space. Furthermore, in VSM, the possible semantic term relationships, like the synonymy between the terms *truck* and *lorry*, are not taken into account when calculating their similarities.

To address the described limitations of VSM, Kuhn *et al.* (2007) demonstrated the use of Latent Semantic Analysis (LSA) to handle the semantic relationships by comparing documents at their topical (conceptual) similarity level. LSA is referred as the Latent Semantic Indexing (LSI) in software maintenance and is known to be the application of LSA to document indexing and retrieval (Kuhn *et al.*, 2007). LSI organises the occurrences of project artefact terms in a repository of term by document matrix and applies the Single Value Decomposition (SVD) principle. SVD is a technique used to reduce noise while keeping the relative distance between artefacts intact. According to Capobianco *et al.* (2009), this allows the storage of semantically related terms based on synonymy, like the association that exists between the terms *engine* and *wheel* to the terms *truck* or *lorry*, in its repository.

Marcus and Maletic (2003) argued that LSI exclude less frequently occurring terms from its vector space by assuming them to be less related. However, if the applications are programmed using a subset of the English language then excluding terms will cause relations that may pose significance to be missed out during information retrieval. Furthermore, the model of language used in prose may not be the same as that found in source code files. According to the authors, the selective terminology used in programming, often at abstract level, introduces additional challenges in recovering traceability. So the issue that remains unanswered in the case study performed by Marcus and Maletic (2003). is how to map similarities between project artefacts when the pieces to link are described in broken language prose.

Additionally Marcus and Maletic (2003) found that when the underlying document space is large, improved results are obtained by using LSI because the main principle of LSI is to reduce this large corpus to a manageable size without loss of information. However, if the corpus is small where the terms and concepts are distributed sparsely throughout the space, reduction of the dimensionality results in significant loss of information. Hence, the authors articulated the need for improving the results by combining structural and semantic information extracted from the source code and its associated documentation. Furthermore, Marcus and Maletic (2003) proposed to support the semantic similarity measure defined in LSI to also consider the program structural information like call-hierarchies.

2.2.3 Dynamic Analysis and Execution Scenarios

Among other software analysis approaches to concept location, a popular technique known as dynamic analysis builds upon the trace information obtained by executing application usage scenarios (Liu *et al.*, 2007). The early work of Wilde and Scully (1995) pioneered concept location using the dynamic analysis technique. Their approach identifies the entry points of computational units for further analysis. However, the approach does not consider related elements within a group of concepts. Eisenberg and Volder (2005) extended the dynamic tracing approach by applying ranking heuristics to improve the accuracy of program elements' relevance to concepts being searched.

Although the dynamic analysis exposes a direct relation between the executed code and the scenario being run, it falls short of distinguishing between overlapping program elements, in that it does not guarantee that all the relevant elements are considered. On the other hand, the static analysis information organises and covers a broader spectrum of program elements though it may fail to identify the contributing elements that belong to a specific scenario. The current research has already recognised the need for combining both of these analysis approaches to aid in IR to support program comprehension. Eisenberg and Volder (2005) used static analysis (e.g. structural dependencies) and dynamic analysis (e.g. execution traces) as well as concept analysis techniques to relate concepts to program elements.

Poshyvanyk *et al.* (2007) combined LSI with Scenario Based Probabilistic (SBP) ranking to address the challenges of concept location and program comprehension. The aim was to improve precision of finding relevant program elements implementing a concept. SBP allows the tracing of execution scenarios and lists program elements (i.e. source code classes and methods) ranked according to their similarity for a given concept when it is executed during a particular scenario. However, LSI represents how the elements relate to each other and requires results to be analysed manually based on domain knowledge. Poshyvanyk *et al.* (2007) claimed that in SBP approach, it is possible to replace the manual knowledge-based analysing and argued that combining the results of both techniques improves the concept location and program comprehension process.

Based on three case studies conducted, Poshyvanyk *et al.* (2007) concluded that the combined techniques perform better than any one of the two (LSI, SBP) techniques independently. Recently, Gethers *et al.* (2011) also compared the accuracy of different IR methods, including VSM, Jensen and Shannon (JS) model, i.e. a similarity measurement representing the relations between data (Zhou *et al.*, 2012b), and Relational Topic Modelling (RTM), i.e. a model of documents with collection of words and the relations between them (Chang and Blei, 2009). Gethers *et al.* (2011) conclude that no IR method consistently provides superior recovery of traceability links. The authors argued that IR-based techniques expose opportunities to improve accuracy of finding program elements by combining different techniques.

Another approach to locating concepts in the source code is to apply Formal Concept Analysis (FCA). FCA defines two dimensions for a concept: extension, covering the entire objects (i.e. program elements) belonging to a concept, and intension, covering all the attributes that are shared by all the objects being considered. Poshyvanyk and Marcus (2007) combined the FCA technique with LSI based IR that organised the search results in a concept lattice to aid software developers in concept location activities. The authors articulated that the intentional description of FCA allows the grouping of concepts to be better interpreted and provides a flexible way of exploring the lattice. Poshyvanyk and Marcus (2007) also claimed that the use of concept lattices in the approach offers additional clues like the relationship between the concepts. The case study results indicate that FCA improves the concept ranking over those obtained by LSI. However, the trade off is when a higher number of attributes are used, it increases the size of the nodes in the generated concept lattices. Thus the sheer amount of information (i.e. the increased number of attributes and nodes) may cause extra burden on developers to comprehend the concept lattice during maintenance.

2.2.4 Source Code Vocabulary

Caprile and Tonella (1999) analysed the structure and informativeness of source code identifiers by considering their lexical, syntactical, and semantical structure. This was done by first breaking the identifiers into meaningful single words and then classifying them into lexical categories. The authors concluded that identifiers reveal the domain concepts implemented by an application, thus provide valuable information to developers when performing maintenance.

Since identifiers often communicate a programmer's intent when writing pieces of source code, they are often the starting point for program comprehension. To help new programmers understand typical verb usage, Høst and Østvold (2007) automatically extracted a verb lexicon from source code identifiers. The quality of identifiers was studied in relationship with other code quality measures, for example, the number of faults in the source code, cyclomatic complexity, and readability and maintainability metrics by Butler *et al.* (2010). Additionally, Abebe *et al.* (2009) illustrated inconsistencies referred to as *lexicon bad smells* existing in source code due to excessive use of contractions, e.g. shortened usage of spoken words like *arg* instead of *argument*. The authors claimed that such inconsistencies may cause problems during software maintenance, especially when searching the source code.

In addition to identifiers, developers use lexical information found in program comments. Even the comments, which are supposedly written in English, tend to gradually narrow into a lexically and semantically restricted subset of English (Etzkorn *et al.*, 2001). Recently, Padioleau *et al.* (2009) study conducted with three operating systems revealed that 52.6% or roughly 736,109[2] comments are to be more than just being as explanatory and conclude that the comments may be leveraged by various search techniques.

While above studies (Abebe *et al.*, 2009; Etzkorn *et al.*, 2001) focused on smells related to identifiers and comments, the lexical anti-patterns described by Arnaoudova *et al.* (2013) introduced a more abstract level of perspective. The authors revealed inconsistencies between source code method names, parameters, return types and comments as well as inconsistencies between source code attribute names, types, and comments, i.e. the comments did not correctly describe the intended purpose of the referenced program elements.

Existing code search approaches treat comments as part of the vocabulary extracted from the source code, however since comments are sublanguage of English, they can deteriorate the performance of the search results due to their imperfect nature, i.e. terse grammar, (Etzkorn *et al.*, 2001; Arnaoudova *et al.*, 2013). Therefore, the terms extracted from the comments should be considered independently from those extracted from the source code identifiers, e.g. method names.

In the absence of textual documentation, one of the techniques software developers utilise during maintenance tasks is to search for the concepts over the application's source code as Marcus *et al.* (2004) demonstrated by using LSI to discover the program elements implementing the concepts formulated on user queries. Kuhn *et al.* (2007) extended this approach to extract concepts from the source code based on the vocabulary usage and presented the semantic similarities that exist within the application's code. Both of these techniques focused on lexical similarities between the source code terms without considering the conceptual relations that exist in the application's domain, however neither of the techniques considers linguistic properties of the terms (i.e. nouns, verbs).

OOP guidelines encourage developers to use nouns to name source files and verbs to describe methods. Shepherd *et al.* (2006) found that natural language (NL) representation of an application's source code also plays an important role in program comprehension. On the role of nouns in recovering traceability, Capobianco *et al.* (2009) presented a method to index only the nouns extracted from the project artefacts to improve the accuracy of IR techniques based on VSM or Probabilistic models. The authors claimed that since software documentation is written using a subset of English, nouns play a semantic role and the verbs play a connector role to assist in recovering traceability links between use-cases and program elements. Capobianco *et al.* (2009) argued that some programming languages like Java are more noun oriented while others like Smalltalk are more verb oriented.

Shepherd *et al.* (2005) presented an approach to discover concepts from the source code by using the lexical chaining (LC) technique. LC groups semantically related terms in a document by computing the semantic distance (i.e. strength of relationship) between two terms. In order to automatically calculate the distance, the path between the two terms as defined in WordNet (Miller, 1995) dictionary and part of speech (PoS) tagging techniques are utilised. The approach in (Shepherd *et al.*, 2005) exploits naming conventions in the source code by using the LC technique to discover related program elements. The authors claimed that complex relations are also identified compared to other approaches by relying on the semantic of a term rather than lexical similarity.

One of the limitations of the approach presented in (Shepherd *et al.*, 2005) is that it fails to consider the terms extracted from the source code may occur in WordNet, but the implied meaning within the source code may differ. According to Sridhara *et al.* (2008), the terms used to represent similar meanings in computer science vocabulary, could have different meanings in the English vocabulary, e.g. the term *fire* extracted from the identifier *fireEvent* is used synonymously with the term *notify* extracted from identifier *notifyListener* but *fire* and *notify* do not commonly relate with one another in English text.

In a further work, Shepherd *et al.* (2006), like Parnas (1972), also argued that OOP promotes the decomposition of concepts into several source code files scattered across multiple layers of an application's architecture as compared to procedural programming languages where all of the implementation of a concept is usually done in one source file. Shepherd *et al.* (2006) introduced the Action Oriented Identifier Graph (AOIG) approach where the source code is formalised as a graphical *virtual* view so that relevant program elements implementing a concept can be viewed as being in one virtual file. It is stated that when concepts are grouped into a category, it is easier to understand and reason with them. One of the limitations of the AOIG approach is that it considers only those methods with noun and verb in their names, hence failing to process those that are declared using other combinations, like only nouns or indirect verbs.

To overcome the limitations of the AOIG approach, Hill *et al.* (2009) constructed a natural representation of source code by extending the AOIG principle and applying linguistic phrase structures like nouns and semantic usage information obtained from mining the comments and term similarities. Furthermore, the authors generalised the AOIG principle by introducing indirect object usage, like noun and preposition phrases, e.g. "*Calculate the interest rate*", to assist the concept location tasks. The introduced algorithm called Software Word Usage Model (SWUM) automatically extracts and generates noun, verb, and prepositional phrases from method and field signatures to capture the term context of NL queries (Hill, 2010).

During the information extraction to capture the word context, the approach evaluates method names and return type names with prepositions in the beginning, at the end and in the middle. By analysing the most frequently occurring identifiers in a set of 9,000 open source projects, common rules were created to determine if an identifier is a noun, verb or preposition phrase. Once the NL phrases have been extracted from the source code, searching the generated corpus to group associated program elements is achieved by applying regular expressions. Using the common rules, terms are extracted from the method names and the signatures are transformed into NL representations, e.g. given a class name "*OneGenericCouponAndMultipleProductVoucherSelector*" with a method signature as "*public List sortByActivationTime(Collections voucherCandidates)*" would get transformed into "*sort voucher candidates by activation time*".

A limitation of the generated context in the approach by Hill (2010) falls short of providing strong clues in detecting all the relevant program elements, especially those ones where the search terms are not used on the declaration of searched program elements. In a financial application for example, searching for the term "*covariance*" using the approach, detects the methods where the term covariance was present in the method signature, however fails to detect the related methods to calculate covariance where the term does not occur on the method signature.

Anquetil and Lethbridge (1998b), on assessing the relevance of program identifier names in legacy systems, stated that "*being able to rely on the names of software artefacts to detect different implementations of the same concept would be very useful*". To assist this, they proposed "*reliable naming*" conventions to enforce similarity between the program element names and the concepts they implement. However, Deißenböck and Pizka (2006) argued that conventions for program identifiers are misleading because they imply that an agreement between software developers exists and demonstrated that developers, on the contrary, follow one another to keep a unified standard during the development of a software project, which may impact programme comprehension (Feilkas *et al.*, 2009).

In addition, Haiduc and Marcus (2008) argued that the main challenge effecting program comprehension is the quality of program identifiers i.e. variables in source code files, being far from perfect. The authors found that searching for program elements implementing concepts are tied to the quality of the terms used to construct these elements. It is concluded that both comments and program identifiers present a significant source of domain terms to aid developers in maintenance tasks. Haiduc and Marcus (2008) articulated that in IR it is essential for any concept mapping and location activity to look at the source code comments and program identifiers since both provide clues to what is being programmed and what was on the developer's mind during implementation.

2.2.5 Ontology Relations

Ontology allows modelling a domain of knowledge by defining a set of basic elements, rendering shared vocabulary amongst them and describing the domain objects or concepts with their properties and relations (Arvidsson and Flycht-Eriksson, 2008). According to Gruber (1993), the definition of ontology within the context of computer science is the representation of knowledge to describe the concepts implemented in the source code files of the application's domain, the relations between the concepts and their attributes (or properties).

In a study, Hsi *et al.* (2003) demonstrated the recovery of core concepts implemented in an application's user interface by manually activating all the user interface elements to construct a list of terms and the interrelations among them. Subsequently, the concepts were discovered after manually analysing and identifying the nouns and indirect objects from the terms existing in the list. The approach established the relationships between the concepts by ontology relation types, *isA* (i.e. generalisation, hierarchical relations) and *hasA* (i.e. aggregation, part of a relation). The authors argued that ontology engineering contributes to many software engineering tasks, for example, during reverse engineering to locate domain concepts or to aid in semantic program comprehension. They claimed that understanding an application's ontology benefits the developers in software maintenance tasks. Petrenko *et al.* (2008) also demonstrated the use of manually constructed ontology fragments to partially comprehend a program and formulated queries to locate concepts in the source code.

Raţiu *et al.* (2008) argued that despite the implementation of concepts being scattered across the source code (Nunes *et al.*, 2011), it is possible to identify the concepts from the application and map them to predefined ontologies because the program elements reflect the modelled concepts and the relations in program identifier names. Raţiu *et al.* (2008) presented a framework to establish mappings between the domain concepts and program elements. The approach uses similarity and path matching to map concepts to the program elements. It assumes that given a set of program elements that refer to a concept, a single program element may also refer to multiple concepts such as a source file implementing more than one concept, which is also observed by Anquetil and Lethbridge (1998a). One of the limitations of the framework presented by Raţiu *et al.* (2008) is that it assumes the identifier names and the concept names are made up of terms that can be directly mapped to one another. This is justified by describing that the names are organised according to the meanings that they represent based on WordNet (Miller, 1995) synsets (i.e. *isA* and *hasPart* relations). However, a term extracted from a program identifier may not always have a synset (i.e. group of synonyms) representation.

Recently, Abebe and Tonella (2010) presented an approach that extracted domain concepts and relations from program identifiers using natural language (NL) parsing techniques. First, the terms were extracted from the program identifiers by performing common identifier splitting and NL parsing techniques. Second, linguistic sentences were constructed by feeding the extracted terms into the sentence templates. The templates were manually created after analysing the file names and method signatures (i.e. method and argument names). Subsequently, the concepts and the relationships were discovered from the linguistic dependencies established by parsing these sentences with a linguistic dependency tool Minipar. Finally, these dependencies were utilised to automatically extract concepts, which were used in building the ontology, by analysing the nouns found on the terms.

Similar to Abebe and Tonella (2010), also Hayashi *et al.* (2010) demonstrated the use of ontology in the concept location process to recover the tractability links between a natural language sentence and the relevant program elements. The approach presented makes use of relational information defined in an ontology diagram and in call-graphs. The results of the search using the approach with ontological relations were compared against those without the ontology. However, in 4 out of 7 cases the precision values have deteriorated when the search performed utilised the ontology relations. One of the reasons for this is due to the approach selecting false positives, possibly caused by OOP inheritance hierarchy where the irrelevant files overriding the originally identified ones were also being selected.

2.2.6 Call Relations

In the approach to assign program elements to concepts, Robillard and Murphy (2002) claimed that the structural information will only be useful when the program elements are directly related or linked to one another via call relations. The authors stated that structural information could be expressed in the source code but lost during dynamic compiler optimisations in the byte code. When performing static call analysis from byte code, to overcome issues like optimistic linking of class calls or resolving class calls by reflection, require that the source code analysis is performed over the complete code base of an application. However, in industrial environments due to the size of an application with multiple external modules linked only during full deployment stage, the complete source code analysis may not be feasible.

Robillard (2005) did further work on automatically generating suggestions for program investigation by extending the approach presented by Biggerstaff *et al.* (1993). The proposed technique ranks the program elements that may be of potential interest to the developers. It takes in a fuzzy set of elements (i.e. methods and variable names) and produces a set of methods and variables that might also be of interest. This is achieved by analysing the topology of the program's structural dependencies, like class hierarchies. The analysis algorithm works by calculating a suggestion set based on the relation type of the program elements. If the relation type specified at input is "*calls*" then the methods in the input set are analysed to produce a set of relevant methods that are "*called by*" from the input set. One of the limitations of this approach is if a novice developer has no experience with the system, he would not be in a position to provide a set of input elements. Hence the approach puts a lot of emphasis on the knowledge of the application prior to using the algorithm.

The existing approaches also combine IR methods like LSI, execution tracing, and prune dependency analysis to locate concepts. In general, these approaches function by first performing a search using query terms and then start navigating the call-graph of the application from the resulting list of methods to their neighbouring methods iteratively. Each method is assigned a weight to determine its relevance to the search query terms. The tf-idf score of a method is used to rank it in the results list. In addition, a weight is also assigned based on where the search term appears, i.e. on the method or on the class file name or both.

On the use of call relations to recover traceability links, Hill *et al.* (2007), Shao and Smith (2009) proposed approaches in which a query is created with search terms, then the source code is searched for the matching methods by using LSI and a score is obtained. Subsequently, for each method on the result list, the call-graph of the application is utilised to evaluate the relevance of the neighbouring methods and a relevance score is assigned based on where the query terms occur in the method names. Finally, both LSI and relevance scores are combined to rank the methods in the final search result list.

In their study, Petrenko and Rajlich (2013) presented a technique called DepIR that combines Dependency Search (DepS) with IR. The technique uses an initial query to obtain a ranked list of methods. The 10 methods with the highest ranks are selected as the possible entry points to explore the shortest path on the call-graph to the method relevant for the query. The shortest path is calculated using Dijkstra's algorithm (Dijkstra, 1959) and the effort needed is calculated as the number of edges (call-relations) on the shortest path plus the IR rank. The study aims to indicate a best-case scenario estimate of the effort needed to locate the relevant method for each bug report. The DepIR performance is compared against pure IR and DepS (a call-graph navigation technique) by using 5 systems and 10 bug report. It is claimed that on average, DepIR required a significantly smaller effort to locate the relevant methods for each bug report. The study only aims to evaluate the theoretical effort reduction in finding the target method by combining lexical ranking with dependency navigation. DepIR doesn't make use of the dependencies to improve the ranking, and thus to further reduce the effort in finding the right artefacts to change.

The IR approaches may consider two strongly related terms via structural information (i.e. OOP inheritance) to be irrelevant (Petrenko and Rajlich, 2013). In turn this may cause one method to be at the top of the result list and the other related one at the bottom. If the relevant methods, at the bottom of the list, are below the threshold used as cut-off point, they would be ignored when evaluating the performance of the approach. This may also cause additional burden on the developers because irrelevant files would need to be investigated before reaching the relevant ones at the bottom of the list. To cater for these structural relations, the static call-graph is utilised, however the call-graphs usually contain many dependencies and very deep nodes/edges that make them impractical for search and navigation purposes.

To compensate for the call-graph limitations, the presented approaches utilise pruning logic, which selectively removes the subtrees (i.e. nodes) during clustering (i.e. grouping) the call relations (i.e. links/edges) found in the call-graph. On exploring the neighbouring methods, Hill *et al.* (2007) introduced two types of call relations, (1) methods that call many others and (2) methods called by others. It is proposed that (1) can be seen as delegating and (2) as performing functionality. Hill *et al.* (2007) argued that the number of edges coming and going out of a method can also be used during clustering a call-graph to decide if the edge should be pruned when moving a node from one cluster to another. However, one of the challenges pruning exposes is that a source code class or method may be removed completely or may be moved to a different group/cluster due to the underlying pruning logic, e.g. simulated annealing, thus may cause inconsistencies when searching for the program elements implementing a concept.

2.3 Information Retrieval in Bug Localisation

Current software applications are valuable strategic assets to companies: they play a central role for the business and require continued maintenance. In recent years, research has focused on utilising concept location techniques in bug localisation to aid the challenges of performing maintenance tasks on such applications. In this section, I describe the existing research approaches and highlight their limitations.

In general, software applications consist of multiple components. When certain components of the application do not perform according to their predefined functionality, they are classified to be in error. These unexpected and unintended erroneous behaviours, also referred as bugs, are known to be often the product of coding mistakes. Upon discovering such abnormal behaviour of the software, a developer or a user report it in a document referred as bug report. Bug report documents may provide information that could help in fixing the bug by changing the relevant components, i.e. program elements of the application. Identifying where to make changes in response to a bug report is called bug localisation where the change request is expressed as a bug report and the end goal is to change the existing program elements (e.g. source code files) to correct an undesired behaviour of the software.

Li *et al.* (2006) classified bugs into three categories: root cause, impact and the involved software components. The categories are further explained as follows.

- The root cause category includes memory and semantic related bugs like improper handling of memory objects or semantic bugs, which are inconsistent with the original design requirements or the programmers' intention.
- The impact category includes performance and functionality related bugs, like the program keeps running but does not respond, halts abnormally, mistakenly changes user data or functions correctly but runs/responds slowly.
- The software components category includes bugs related to the components implementing core functionality, graphical user interfaces, runtime environment and communication, as well as data base handling.

The study in (Li *et al.*, 2006), performed with two OSS projects, showed that 81% of all bugs in Mozilla and 87% of those in Apache are semantics related. These percentages increase as the applications mature, and they have direct impact on system availability, contributing to 43–44% of crashes. Since it takes a longer time to locate and fix semantic bugs, more effort needs to be put into helping developers locate the bugs.

Although bug localisation is an instance of concept location (Moreno *et al.*, 2014) and previously described VSM or LSI information retrieval models are utilised to locate the relevant files, it differs compared to concept location with regards to the number of relevant files being identified. One of the main principles of concept location is to identify all source code files implementing a concept whereas in bug localisation only the files that are involved in the reported bug needs to be identified, which according to Lucia *et al.* (2012) may be in just a few of the thousands of files comprising an application.

Zhou *et al.* (2012a) proposed an approach consisting of the four traditional IR steps (corpus creation, indexing, query construction, retrieval & ranking) but using a revised Vector

Space Model (rVSM) to score each source code file against the given bug report. In addition, each file gets a similarity score (SimiScore) based on whether the file was affected by one or more closed bug reports similar to the given bug report. Similarity between bug reports is computed using VSM and combined with rVSM into a final score, which is then used to rank files from the highest to the lowest.

2.3.1 Evaluation Metrics

In bug localisation a search query is formulated from a bug report, which is substantially different in structure and content from other text documents, thus requiring special techniques to formulate the query (Dilshener *et al.*, 2016). Many affected program elements may not relate to the bug directly but they are rather results of the impact of the bug. For example, a bug report may describe an exception in the user interface component but its cause may lie in the component responsible for database access. Hence to measure the performance of bug localisation in precision and recall would be inadequate and unfair, instead the rank of the source file in the result list should be considered because it indicates the number of files a developer would need to go through before finding the relevant one to start with, e.g. if the relevant one is ranked at 5th position then the developer would have to investigate all four first (Marcus and Haiduc, 2013). Therefore the existing research techniques measure the performance of the results as top-N Rank of Files (TNRF), Mean Average Precision (MAP) and Mean Reciprocal Rank (MRR).

- TNRF is the top-N rank, which gives the percentage of bug reports whose related files are listed in top-N (which was set to 1, 5, and 10) of returned files.
- MAP provides a synthesised way to measure the quality of retrieved files, when there are more than one related file retrieved for the bug report.
- MRR metric measures the overall effectiveness of retrieval for a set of bug reports.

TNRF metric is used by Zhou *et al.* (2012a) as, given a bug report, if the top-N query results contain at least one file with which the bug should be fixed, the bug is considered to be located. The higher the metric value, the better the bug localisation performance. When a bug report has more than one related files, the highest ranked one among all the related files is used to evaluate the performance.

MAP has been shown to have especially good discrimination and stability (Manning *et al.*, 2008). The MAP is calculated as the sum of the average precision value for each bug report divided by the number of bug reports for a given project. The average precision for a bug report is the mean of the precision values obtained for all affected files listed in the result set. MAP is considered to be optimal for ranked results when the possible outcomes of a query are 5 or more (Lawrie, 2012).

MRR is a metric for evaluating a process that produces a list of possible responses to a query (Voorhees, 2001). The reciprocal rank for a query is the inverse rank of the first relevant document found. The mean reciprocal rank is the average of the reciprocal ranks of results of a set of queries. MRR is known to measure the performance of ranked results better when the outcome of a query is less than 5 and best when just equal to one (Lawrie, 2012).

2.3.2 Vocabulary of Bug Reports in Source Code Files

In general, a typical bug report document provides multiple fields where information pertaining to the reported issue may be described such as a summary of the problem, a detailed description of the conditions observed, date of the observed behaviour, name of the files changed to resolve the reported condition. Recent empirical studies provide evidence that many terms used in bug reports are also present in the source code files (Saha *et al.*, 2013; Moreno *et al.*, 2013). Such bug report terms are an exact or partial match of code constructs (i.e. class, method or variable names and comments) in at least one of the files affected by the bug report, i.e. those files actually changed to address the bug report. Moreno *et al.* (2013) show that (1) the bug report documents share more terms with the corresponding affected files than with other files in a system and (2) the shared terms are not randomly distributed in the patched source files instead the file names had more shared terms than other locations. The similarity between the terms extracted from the bug reports and those extracted from the source files shows that there is a correlation between each other and it gets stronger when the files are the affected ones. In addition, the terms were similar between the file names and comments rather then other parts of the source file, e.g. method names.

Subsequently, Moreno *et al.* (2013) discovered that source code file names revealed more of the terms extracted from the bug reports than any other location of a source files, e.g. the method names or variables. The authors evaluated their study using 6 OSS projects, containing over 35,000 source files and 114 bug reports, which were solved by changing 116 files. For each bug report and source file combination (over 1 million), they discovered that on average 75% share between 1–13 terms, 22% share nothing and only 3% share more than 13 terms. Investigating the number of shared terms between the bug reports and the changed set of files also revealed that 99% shared vocabulary.

Furthermore, Moreno *et al.* (2013) focused on the code location of the terms in the changed pair set and found that the largest contributor location to the source code vocabulary are the comments (27%), followed by the method names (20%) and then the identifiers names (13%). Investigating the contribution of each code location to the total shared terms between the bug reports revealed that comments contributed 27%, methods 17% and variables 13% respectively. Additionally, the study revealed that source code file names made up of a few terms but all of them contributed to the number of shared terms between a bug report and its affected files. For example, file names are made up of only 2 or 3 terms, e.g. in a financial application the source code file name *MarketRiskCalculator* is made up of 3 terms, *market, risk* and *calculator,* after camel case splitting (Butler *et al.*, 2011). Moreno *et al.* (2013) discovered that on average 54% of the cases all of those file name terms are present in the bug reports. Therefore the authors conclude that bug reports and source files share information, i.e. terms, and that the number of shared terms are directly related to the changed files.

A study done by Saha *et al.* (2013) also discovered that class file names are typically a combination of 2–4 terms and that the terms are present in more than 35% of the bug report summary fields and 85% of the bug report description fields of the OSS project AspectJ. Furthermore, the exact file name is present in more than 50% of the bug descriptions. The authors concluded that when the terms from these locations are compared during a search, the noise is reduced automatically due to reduction in search space. For example, in 27 AspectJ bug reports, at least one of the file names of the fixed files was present as-is in the bug report summary, whereas in 101 bug reports at least one of the terms extracted from the file name was present. Although the summary field contained only 3% of the total terms found in the bug report, at least one of the terms extracted from the comments, file, method or variable names was found in the bug report summaries in 82%, 35%, 72% or 43% of the cases respectively.

Finally, Moreno *et al.* (2013) have experimented by using the shared terms to search for relevant files using LSI, Lucene[1] and shared-terms only, which revealed that shared-terms search outperformed LSI in 63% of the cases. Although Lucene—a well known Java-based framework providing tools to index and search documents—out performed both LSI and shared-terms, there were a few cases where shared-terms performed 22% better than Lucene. One of the limitations of this approach is that since the bug reports already documented the affected files, the number of shared terms were easily identified and restrained to the reported average (i.e. between 1–13). Subsequently, performing search by utilising only those shared terms might have helped reduce the number of false positives, i.e. less relevant but accidentally matched source code class files.

While both studies (Saha *et al.*, 2013; Moreno *et al.*, 2013) have shown that there is some textual overlap between the terms used in bug reports and those in the source code files, this might not always be the case due to the different source code terms used by developers and the ones used by those creating the bug reports, for example, Bettenburg *et al.* (2008) showed that bug reports may be tersely described and may contain test scenarios rather then developer oriented technical details.

2.3.3 Using Stack Trace and Structure

It has been argued that existing approaches treat source code files as one whole unit and bug report as one plain text. However, bug reports may contain useful information like stack traces, code fragments, patches and recreation steps (Bettenburg *et al.*, 2008). A study done by Schröter *et al.* (2010) explored the usefulness of stack trace information found in the bug reports. Figure 1.3 in Chapter 1, shows a stack trace, which lists a sequence of files that were executing at the time of the error. Schröter *et al.* (2010) investigated 161,500 bug reports in the Eclipse project bug repository and found that around 8% (12,947/161,500) bug reports contained stack trace information. After manually linking 30% (3940/12,947) of those closed bug reports to their affected files, 60% (2321/3940) revealed that the file modified to solve the bug was one of the files listed in the stack trace or stack frame (ordered list of method/file names).

Schröter *et al.* (2010) note that stack traces identified in the study contained up to 1024 frames and a median of 25 frames. Out of 2321 bug reports with a stack trace, in 40% the affected file was in the very first position of the stack trace and in 80% it was within the first

[1] http://lucene.apache.org/java/docs/index.html

6 positions. It is not clear if non-project specific file names are ignored during the evaluation of the stack frames. Investigating the affected files revealed that 70% of the time the affected file is found in the first trace and in 20% it is found on the second or third trace. Finally, the authors investigated whether including stack trace in bug reports speeds up the development process and conclude that fixing bugs with stack trace information, requires 2–26 days and without stack trace 4–32 days.

Moreno *et al.* (2014) presented an approach that leverages the stack trace information available in bug reports to suggest relevant source code files. The authors argue that if 60% of the cases the stack trace contains at least one of the source code files that was changed to resolve the bug (Schröter *et al.*, 2010), then in rest of the cases (40%) the changed source file should be a neighbouring one to those present in the stack trace. Based on this argument, their approach calculates two scores: (1) a textual similarity between a bug report and a source file using VSM scoring and (2) a structural similarity score based on the call-graph shortest path between each file listed in the stack trace and the called/calling class file. Finally, both the VSM (1) and stack trace (2) scores are combined using an *alpha* factor, which determines the importance given to the stack trace score when considering it in the final score.

After evaluating the approach with only 155 bug reports containing a stack trace, the results reveal that 65% of the bugs were fixed in a file mentioned in the stack trace submitted in the bug report. Out of 314 files fixed to resolve the reported bugs, 35% (109/314) were at a distance 1 from the files listed in the stack trace, 11% (36/314) at a distance of 2 and 8% (25/314) at a distance between 3–7. Moreno *et al.* (2014) first varied the stack trace score's *lambda* factor (determines the number of neighbouring files to consider on call-graph) between 1–3 and the *alpha* factor (determines the weight given to the stack trace score) between 0–1, subsequently assessed the overall performance using MAP and MRR. The authors observed that increasing *lambda* value to consider up to 3$^{\text{rd}}$ generation of neighbours decrease the performance and recommend that up to 2 neighbours be considered. The best performance was gained when the *alpha* factor value was set between 0.7–0.9, which indicates that the VSM score needs to be aided by additional sources of information at 70%–90% ratios to achieve satisfactory results.

Furthermore, Moreno *et al.* (2014) noted that when two files are structurally related by call relations, and when one of them has a higher textual similarity against a particular query then the irrelevant file is ranked higher. So in these situations adjusting the alpha value allowed the relevant file to be ranked higher instead. The authors observed that when parts of the code that needs to be changed are not structurally related to the files listed in the stack trace then the effectiveness of the approach degrades. However, they conclude that overall 50% of the files to be changed were in the top-4 ranked files. Compared to a standard VSM (Lucene), the approach achieved 53% better, 29% equal and 18% worst performance results.

One of the limitations of the study in (Moreno *et al.*, 2014) is that it does not differentiate the two types of call-relations, i.e. *caller*: a class that calls others and *callee*: a class called by others (Hill *et al.*, 2007). The software developers may follow the call hierarchies to identify callers of the candidate class file and assign higher importance to them. While n-number of callee class files from the one listed on the stack trace are being evaluated by Moreno *et al.* (2014), no importance is given to the callers of the one found on the stack trace.

Wong *et al.* (2014) proposed a method that also considers the stack trace information and performs segmentation of source files to deal with the fact that source files may produce noise due to varying lengths (Bettenburg *et al.*, 2008). The method involves dividing each source file into sections so called segments, and matching each segment to a bug report to calculate a similarity score. After that for each source file a length score is calculated. In addition, the stack trace is evaluated to calculate a stack trace (ST) boost score. Subsequently, for each source file, the segment with the highest score is taken as the relevant one for the bug report and multiplied by the length score to derive a similarity score. Finally, the ST score is added on top of the similarity score to get the overall score.

The segmentation involves dividing the number of terms extracted into equal partitions, e.g. if a source file has 11 terms, and the segmentation is set as 4, then 3 segments are created with the first two having 4 terms and the last one having only 3 terms. The authors have varied the number of terms used in each segment between 400–1200 and conclude that 800 terms in a segment achieved an optimal effect in the performance of the results.

The length score is calculated by revising the logistic function used in (Zhou *et al.*, 2012a) with a *beta* factor to regulate how much favour is given to larger files. The authors have varied the *beta* values between 30 and 90 and conclude that 50 achieved an optimal effect.

The ST boost score is calculated by considering the files listed in the stack as well as the files referenced via import statements known as the closely referred ones, which was first introduced by Schröter *et al.* (2010). Additionally, the SimiScore function defined by Zhou *et al.* (2012a) is utilised to calculate the similarity of the bug report to the previously closed bugs and give favour to the files closed by those similar reports.

The approach is evaluated by measuring the performance against (Zhou *et al.*, 2012a) with three of the same datasets used in that study to see whether the segmentation and stack trace makes a difference. Three sets of search tasks are performed, first with both segmentation and stack trace enabled, second with only segmentation and third with only ST boosting. In all three cases, two variants of evaluation criteria are used, one with considering similar bug reports and the other without. It is concluded that the approach is able to significantly outperform the one introduced in (Zhou *et al.*, 2012a) regardless of considering similar bug reports or not. Authors claim that segmentation or stack trace analysis is an effective technique to boost bug localisation and both are compatible with the use of similar bug reports.

One of the limitations of this approach, also acknowledged, is that the effectiveness of segmentation is very sensitive to how the source files are divided, e.g. based on number of methods or based on number of lines. The number of terms to be included in each segment has an impact on similarity, in that a code section may get split into different segments thus reducing relevance, or including the section in one larger segment may produce false positives in certain cases. In addition, including closely referred source files, i.e. called files, via import statements in the search space may cause false positives as Moreno *et al.* (2014) study highlights. Furthermore, the study in (Wong *et al.*, 2014) fails to provide evidence on how much benefit the closely referred files bring into the performance of the results.

Furthermore, there is no indication in either of the studies (Moreno *et al.*, 2014; Wong *et al.*, 2014) on how the approach handles many non-application specific file names contained in the stack trace like those form third-party vendors or those belonging to the programming language specific API, which may be irrelevant for the reported bug.

2.3.4 Version History and Other Data Sources

Nichols (2010) argues that utilising additional sources of information would greatly aid IR models to identify relations between bug reports and source code files. One of the information sources available in well-maintained projects is the past bug details. To take advantage of the past bugs, the author proposed an approach that mines the past bug information automatically. The approach extracts semantic information from source code files like identifier and method names but excludes the comments, claiming that this would lead to inaccurate results. Subsequently, two documents (i.e. search repositories) are created, one containing a list of method signatures and the other containing all the terms extracted form those methods. Finally, information from previous bug reports (e.g. title, description and affected files) are collected and appended (augmented) into the document containing the method terms. The author also manually checked to see where exactly the code was changed by a previous bug report and appends its vocabulary on to the vocabulary of that code section.

Nichols has experimented with augmented and non-augmented search repositories and found that the search results from the repository augmented with up to 14 previous bug reports has produced same results as those from the non augmented one. The best performance is obtained when 26–30 previous bugs were used in the augmented repository. It is observed that there is no guarantee of considering previous bug information contributes towards improving the rank of a source file in the result list and concludes that quality bug reports, e.g. those with good descriptions, is more valuable than the number of bug reports included in the repository. One of the limitations of the presented approach is that augmenting the repository with bug reports requires manual effort, which may be prone to errors and the author fails to describe how its validity was verified.

Ratanotayanon *et al.* (2010) performed a study on diversifying data sources to improve concept location, and asked if having more diverse data in the repository where project artefacts are indexed always produce better results. They investigated the effects of combining information from (1) change set comments added by developers when interacting with source version control systems, (2) issue tracking systems and (3) source code file dependency relations, i.e. the call-graph. They experimented with four different combinations of using these three information sources together.

1. Change sets (i.e. commit messages and the fixed files).

2. Change sets and call-graph (only the calling class files of the fixed files).

3. Change sets and bug reports.

4. All of the above.

It is observed that (1) using change sets together with bug reports produces the best results, (2) including the referencing files, i.e. the calling class files available on the call-graph, increased recall but deteriorated precision and (3) adding bug report details into the commit comments had little effect on the performance of the results.

The authors argue that change sets provide vocabulary from the perspective of the problem domain because developers add, on each commit, descriptive comments which are more aligned with the terms used when performing a search task. However, success of utilising the change sets in search tasks are sensitive to the quality of these comments entered. One of the reasons for the third observation performing poorly, as stated above, could be due to the similar vocabulary being used in commit comments as in bug reports, hence the search was already benefiting from the commit message vocabulary and bug report descriptions did not provide additional benefit.

Additionally, Ratanotayanon *et al.* (2010) claimed that refactoring of the code base severely impacts the usability of commit comments in that they no longer match with the affected class files. Finally, the authors propose that when using a call-graph, the results be presented in ranked order based on a weighting scheme to compensate for any possible deterioration in precision due to false positives.

In the approach presented by Ratanotayanon *et al.* (2010), the change sets are augmented with the information extracted from the revision history, which depends on the consistency and conciseness of the descriptions entered by the developers on commit messages during interacting with source code control systems. Since different variations of code control systems exist, applicability of the approach in environments other then the one used may constrain the generalisability of the approach.

2.3.5 Combining Multiple Information Sources

One of the earlier works on combining multiple sources of information to aid bug localisation was to apply the concept of data fusion to feature location (Revelle, 2009). Data fusion involves integrating multiple sources of data by considering each source to be an expert (Hall and Llinas, 1997; Revelle, 2009). The proposed approach by Revelle (2009) combines the structural dependencies and textual information embedded in source code as well as dynamic execution traces to better identify a feature's source code files.

The approach by Revelle (2009), initially allows users to formulate queries and ranks all the source code files by their similarity to the query using LSI. Subsequently, using dynamic analyses, traces are collected by running a feature of the application. The source code files absent in the traces, i.e. not executed, are pruned from the ranked list. Finally, using a program call-graph both ranked list and dynamic information is optimised. For example, if a file's neighbour in the call-graph was not executed and did not meet a textual similarity threshold, then the dependency is not followed. The approach is later on implemented in a tool that integrates textual and dynamic feature location techniques called FLATTˆ3 (Savage *et al.*, 2010).

One of the limitations of the proposed approach is that it depends on the execution scenarios, which require recreational data that may no longer be available. In addition, pruning may unintentionally remove relevant files, thus negatively effect the performance of the results. Revelle *at al.* also claim that a change request, e.g. a bug report, may be relevant for only a small portion of a feature, hence may not fully represent all the files pertaining to the feature.

Saha *et al.* (2013) claim that dynamic approaches are more complicated, time consuming and expensive than static approaches where some sort of a recommendation logic is more appealing. The authors argued that existing techniques treat source code files as flat text ignoring its rich structures, i.e. file names, method names, comments and variables. Saha *et al.* (2013) proposed an approach where 8 scores are calculated based on the similarity of terms from 2 parts of a bug report (summary and description) with the terms from 4 parts of a source code file (file name, method names, variable names and comments).

The approach adopts the built in TF.IDF model of Indri[2], which is based on BM25 (Okapi) model[3], instead of VSM. The authors argue that different parts of text documents, e.g. bug descriptions, may cause noise due to matching lots of irrelevant terms. Based on a preliminary analysis, they have discovered 6 locations from where terms originate, bug summary and description as well as source file name, methods, comments and variables. One of the limitations the approach presented by Saha *et al.* (2013) assumes all features are equally important and ignores the lexical gap between bug reports and source code files, which is problematic due to potential contamination with bug-fixing information.

Davies and Roper (2013) also proposed a combined approach that integrates 4 sources of information and implemented by considering source code's method names instead of file name. Each individual information source is calculated as follows: (1) textual similarity between the bug description and the source code file's method; (2) probability output by a

[2] http://www.lemurproject.org

[3] http://nlp.stanford.edu/IR-book/html/htmledition/okapi-bm25-a-non-binary-model-1.html

classifier indicating a previously closed bug's similarity to the method, i.e. similar bug reports; (3) number of times a file's method was fixed, divided by the total number of bugs; and (4) inverse position of the method in stack traces contained in the bug report. The authors evaluated the approach using three OSS projects (Ant[4], JMeter[5], JodaTime[6]) and compared the performance of using each of the information sources (i.e. 1—4) as individual approach against the combined one. The results indicate that the combined approach outperforms any of the individual approaches in terms of both top-1 and top-10 results.

One of the limitations of the approach is that it depends upon the number of previously closed bug reports used to train the classifier. For example, Davies and Roper (2013) report poor performance results for JodaTime compared to the others (Ant, JMeter), which may be due to the small number of bug reports in that project. Additionally, the source code files may contain several methods and clutter the results list, hence burden developers when navigating the results. For example, FileA has 10 methods, FileB 20 and FileC 8. When 5 methods of FileA and 5 of FileC are relevant, but 5 methods of FileB are falsely placed in the top-10, then there is only room for 5 more methods to be placed into top-10, either from FileA or FileC, rest will be placed beyond top-10, thus negatively impacting precision of the results and forcing developers to investigate the irrelevant methods of FileB in top-10.

Wang and Lo (2014) proposed an approach that also consider the files found on version history commits, i.e. number of times a source file is fixed due to a bug based on commit messages. Instead of considering all the previous bug reports (Sisman and Kak, 2012), only recent version history commits are considered. The approach looks to see when the current bug was created and compare the number of days/hours between each past commit to assign a score to reflect the relevance of those previously modified files for the current bug at hand. For example, if a file was changed 17 hours ago, it gets a higher score then if it was changed 17 days ago.

Additionally, Wang and Lo (2014) adopted the similarity score (SimiScore) introduced by Zhou *et al.* (2012a), which basically considers the term similarity between a current bug and all previously reported bugs. Based on the SimiScore, the affected files of all previous bug reports are assigned a suspiciousness score to reflect their relevance for the current bug. Finally, the authors utilised the structure score introduced in Saha *et al.* (2013). Once all 3

[4] ant.apache.org

[5] jmeter.apache.org

[6] joda-time.sourceforge.net

scores (history, bug similarity and structure) are calculated, they are combined together in the following order.

The approach introduced by Wang and Lo (2014) first calculates a suspiciousness score by combining the structure score and the SimiScore using an *alpha (a)* value of 0.2, which is the same as the one used by Saha *et al.* (2013). So only 20% weight is given to files from previous bug reports, as follows.

SRscore(f) *= (1-a) * structure score of a file + a * SimiScore of a file*

Subsequently, the history score is combined with the SRscore in a similar way by using a *beta (b)* factor, which again determines how much consideration is given to history score compared to the combined score of structure and similar bug reports, as follows.

SRHscore(f) *= (1-b) * SRscore(f) + b * history score of a file*

Wang and Lo (2014) experimented with different values of *beta* and conclude that when it is incremented by 0.1, i.e. 10% weight is given to history, the performance steadily increases up to a *beta* value of 0.3 and beyond that the performance decreases. Compared to other history based bug localisation techniques, the performance of MAP is improved by 24% over the approach introduced in Zhou *et al.* (2012a) and 16% over the one introduced by Saha *et al.* (2013). The evaluation of the approach shows that considering historical commits up to 15 days increased the performance, 15 to 20 days did not make much difference and considering up to 50 days deteriorated the performance. Thus Wang and Lo (2014) claim that most important part of the version history is in the commits of the last 15 to 20 days.

Recently, Ye *et al.* (2014) claimed that if a source file is recently changed than it may still contain defects and if a file is changed frequently than it may be more likely to have additional errors. The authors argued that there is a significant inherent mismatch between descriptive natural language vocabulary of a bug report and the vocabulary of a programming language found in source code file. They propose a method to bridge this gap by introducing a novel idea of enriching bug descriptions with terms from the API descriptions of library components used in the source code file.

The ranking model of the approach combines six features, i.e. scores, measuring the relationship between bug reports and source code files, using a learning-to-rank (LtR) technique: (1) lexical similarity between bug reports and source code files; (2) API enriched lexical similarity, using API documentation of the libraries used by the source code; (3) collaborative filtering, using similar bug reports that got fixed previously; (4) bug fixing recency, i.e. time of last fix (e.g. number of months past since last fix); (5) bug fixing frequency, i.e. how often a file got fixed; (6) feature scaling, used to bring the score of all previous scores (1–6) into one scale. Their experimental evaluations show that the approach places a relevant file within the top-10 recommendations for over 70% of the bug reports of Tomcat[7].

The source files are ranked based on the score obtained by evaluating each features weight. Although improved results are obtained compared to existing tools, Ye *et al.* (2014) reported that in two datasets, AspectJ[8] and Tomcat, existing tools also achieved very similar results. One of the reasons is that in AspectJ affected files were very frequently changed and in Tomcat the bug reports had very detailed and long descriptions. Evaluating the effectiveness of each feature on the performance show that the best results are obtained by lexical similarity (1) and previous bug reports (2), hence this leaves the question whether considering features like version history and bug fix frequency is really worth the effort since other studies also report their effectiveness rather being poor.

One of the limitations LtR approach exposes is that it makes use of machine learning based on a set of training data to learn the rank of each document in response to a given query. However, training data is often unavailable in many cases particularly when developers work on a new software. Furthermore, the bugs in the training set may not be representative enough for the incoming buggy files, which could limit the effectiveness of the approach in localising bugs.

2.4 Current Tools

Zhou *et al.* (2012a) implemented their approach in a tool called BugLocator, which was evaluated using over 3,400 reports of closed bugs (see Table 2.1) and their known affected files from four OSS projects: the IDE tool Eclipse[9], the aspect-oriented programming library AspectJ, the GUI library SWT[10] and the bar-code tool ZXing[11]. Eclipse and AspectJ are well-known large scale applications used in many empirical research studies for evaluating various IR models (Rao and Kak, 2011). SWT is a subproject of Eclipse and ZXing is an Android project maintained by Google. The rVSM and similarity scores introduced by Zhou

[7] http://tomcat.apache.org/

[8] http:// www.st.cs.uni-saarland.de/ibugs/

[9] http://www.eclipse.org

[10] http://www.eclipse.org/swt/

[11] http://code.google.com/p/zxing/

Table 2.1: Project artefacts and previous studies that also used them

Project	**Source files**	**Bug reports**	**Period**	**Used also by**
AspectJ	6485	286	2002/07 - 2006/10	Zhou *et al.* (2012a), Saha *et al.* (2013), Wong *et al.* (2014), Wang and Lo (2014), Youm *et al.* (2015)
Eclipse	12863	3075	2004/10 - 2011/03	Zhou *et al.* (2012a), Saha *et al.* (2013), Wong *et al.* (2014), Wang and Lo (2014),
SWT	484	98	2004/10 - 2010/04	Zhou *et al.* (2012a), Saha *et al.* (2013), Wong *et al.* (2014), Wang and Lo (2014), Youm *et al.* (2015), Rahman *et al.* (2015)
ZXing	391	20	2010/03 - 2010/09	Zhou *et al.* (2012a), Saha *et al.* (2013), Wang and Lo (2014), Rahman *et al.* (2015)
Tomcat	2038	1056	2002/07 - 2014/01	Ye *et al.* (2014)
ArgoUML	1685	91	2002/01 - 2006/07	Moreno *et al.* (2014)
Pillar1	4355	27	2012/03 - 2013/01	-
Pillar2	337	12	2010/05 - 2011/01	Dilshener and Wermelinger (2011)

et al. (2012a) are each normalised to a value from 0 to 1 and combined into a final score: (1-w)*normalrVSM+w*normalSimiScore, where w is an empirically set weight.

The final score is then used to rank files from the highest to the lowest. For two of these projects w was set to 0.2 and for the other two w=0.3. The performance of BugLocator was evaluated with 5 metrics: Top-1, Top-5, Top-10, MAP and MRR. Across the four projects, BugLocator achieved a top-10 of 60–80%, i.e. for each project at least 60% of its bugs were affected by at least one of the first 10 suggested files. One of the limitations of this approach is that the parameter w is tuned on the same dataset that is used for evaluating the approach, which means that the results reported correspond to the training performance. It is therefore unclear how well their model may generalise with previous bug reports that are not used to train the model.

Saha *et al.* (2013) presented BLUiR tool, which leverages the structures inside a bug report and a source code file as discussed earlier in Section 2.3.5. The results were evaluated using BugLocator's dataset and performance indicators. For all but one indicator for one project (ZXing's MAP), BLUiR matches or outperforms BugLocator, hinting that a different IR approach might compensate for the lack of history information, namely the previously closed similar bug reports. Subsequently, BLUiR incorporated SimiScore, which did further improve its performance.

Wang and Lo (2014) implemented their approach (Section 2.3.5) in a tool called AmaLgam for suggesting relevant buggy source files by combining BugLocator's SimiScore and BLUiR's structured retrieval into a single score using a weight factor, which is then combined (using a different weight) with a version history score that considers the number of bug fixing commits that touch a file in the past k days. This final combined score is used to rank the file. The approach is evaluated in the same way as BugLocator and BLUiR, for various values of k. AmaLgam matches or outperforms the other two in all indicators except one, again the MAP for ZXing.

Wong *et al.* (2014) implemented their proposed method, which considers the stack trace information and performs segmentation of source files in BRTracer tool by extending the approach utilised in BugLocator. The approach is evaluated by measuring the performance against BugLocator as described in Section 2.3.3. It is concluded that the approach is able to significantly outperform the one introduced in (Zhou *et al.*, 2012a) for all the three projects regardless of considering similar bug reports or not.

Recently, Youm *et al.* (2015) introduced an approach where the scoring methods utilised in previous studies (Zhou *et al.*, 2012a; Wong *et al.*, 2014; Saha *et al.*, 2013; Wang and Lo, 2014) are first calculated individually and then combined together. The final scores is a product of all the scores and how much importance is given to each score is determined by varying *alpha* and *beta* parameters. The approach implemented in a tool called BLIA is compared against the performance of the other tools where the original methods were first introduced. For evaluation only the three smaller BugLocater datasets (i.e. excluding Eclipse) are used. Although BLIA improves the MAP and MRR values of BugLocator, BLUiR and BRTracer, it fails to outperform AmaLgam in AspectJ. The authors found that stack-trace analysis is the highest contributing factor among the analysed information for bug localisation.

In another recent study, Rahman *et al.* (2015) extended the approach introduced in Zhou *et al.* (2012a) by considering file fix frequency (*fileFixFrequency*) and file name match (*fileNameMatch*) scores as *BugLocator(rVSM+SimiScore)*fileFixFrequency+fileNameMatch.*

The file fix frequency score is calculated based on the number of times a ranked file is mentioned in the bug repository as changed to resolve another bug. The file name match score is based on the file names that appear in the bug report. The file names were extracted from the bug report using very simple pattern matching techniques and collected in a list. Subsequently during the search, if a ranked file is in the list of files, its score is boosted with a *beta* value.

The approach is evaluated with the SWT and ZXing projects from the BugLocator dataset and one of their own: Guava[12]. The authors show improved MAP and MRR values over BugLocator's. Rahman *et al.* (2015)'s (1) extract the file names from the bug report using very simple pattern matching techniques and (2) use a single constant value to boost a file's score when its name matches one of the extracted file names. One of the limitations of this approach is that treating all the positions equally important with only one constant value may be the reason why increasing its weight (*beta* factor) beyond 30% deteriorates the results because irrelevant files are being scored higher.

2.5 User Studies

Sillito *et al.* (2008) on asking and answering questions during implementing a change task, conducted two different studies, one in a laboratory setting with 9 developers who were new to the code base and the other in an industrial setting with 16 developers who were already working with the code base. In both studies developers were observed performing change tasks to multiple source files within a fairly complex code base using modern tools. In both cases the 30–45min long sessions were documented with an audio recording, screen shots and a post session interview was conducted. The outcome of the studies allowed authors to generate types of questions that developers ask organised under following 4 categories.

1. Finding focus points: identifying single classes as entry points
2. Expanding focus points: relating the single classes with their surrounding classes to find more information relevant to the task.
3. Understanding a sub-graph: the functionality provided with those related classes in (2) and than building an understanding of the concepts.
4. Correlating the sub-graphs with other sub-graphs: how the functionality is mapped to another sub-set of classes providing another type of functionality.

Subsequently, the authors describe the behaviours observed while answering these questions and highlight the gaps that current development tools have when providing support to developers within the context of these questions. The authors conclude that questions of types (1) through (3) are asked by mostly developers who are unfamiliar with the code base while type (4) questions are asked by developers who already have the expertise with the code base.

[12] https://github.com/google/guava/wiki

The questions asked in the 3$^{\text{rd}}$ category was aimed to build an understanding of the control flow among multiple classes as well as to understand the functionality provided to address a domain concept. Since current IDEs do not consider information flow during the search stage, a developer would need to first get a result list of candidate files and than search for call relations. These steps, as highlighted by the authors, caused developers to loose track of where they were and often ended up stating a fresh search.

The authors also mention that developers who are more experienced with the code base often ask questions of type (4), which evolves around how one set of functionality provided by a set of class files (e.g. call-relations) may map or relate to another functionality provided by another set of classes. This Indicates that the importance of control flow between these two sets may be of importance when ranking the class files.

Likewise, Starke *et al.* (2009) performed a study with 10 developers to find out how developers decide what to search for and decide what is relevant for the maintenance change task at hand. The study was conducted at a laboratory setting and involved 10 developers with prior Java development experience (of which 8 gained in industrial environments) using the Eclipse IDE. Participants were randomly assigned one of the 2 closed change tasks selected from the Sub-eclipse tool's issue repository. Each developer at each session was paired with a researcher (the second author) to collect data during the 30min. study session. The researcher acted as the observer (at the keyboard) performing the instructions given by the developer (the driver). This type of pairing allowed all dialogue exchange between the observer and the driver to be recorded. At the end of each session, another researcher (the first author) conducted a 10min interview with the developer. To ensure that 30min was sufficient two pilot studies were carried out.

In each session, participants were instructed to carry out search tasks in Eclipse using the 8 available search tools. Out of 96 search queries performed, 35 returned ten or more results, 22 more than fifty and 29 returned no results at all. Astonishingly, only 4 out of 10 developers has decided to open one of the files on the result list to inspect its content for relevance. All others briefly browsed the result list and decided to perform a new search.

The authors summarise their key observations of the developers in following 5 categories.

1. Forming hypotheses to what is wrong based on past experience.

2. Formulating search queries based on their hypotheses around naming conventions.

3. Looking for what might be relevant in the large search results with a very fuzzy idea.

4. Skimming through the result list rather than systematically investigate each result.

5. Opened a few files, briefly skimmed through and performed a new search rather than investigating the next item on the result list.

Starke *et al.* (2009) state that developers look at the code base and use their experience in the domain when forming their hypotheses to what might be the problem. It is highlighted that developers found formulating search queries challenging because Eclipse search tools require the search terms to be precisely specified otherwise no relevant results are returned. The authors found that when large search results are returned, the developers tend to loose confidence in the query and decide to search again rather than investigate what was returned.

Starke *et al.* (2009) propose future research on tool support for the developers to provide more contextual information by presenting the results in a ranked order, grouped within the context relevant for the search tasks at hand.

While developers ask focus-oriented questions to find entry points when starting their investigations, they use text-based searches available in current IDEs as highlighted in both studies (Sillito *et al.*, 2008; Starke *et al.*, 2009). The tools require the search terms to be precisely specified otherwise irrelevant or no results are returned, thus cause additional burden on developers due to repeatedly preformed discovery tasks in a trial and error mode. Sillito *et al.* (2008) also point out that additional challenges lie in assessing the high volume of information returned in the results because most tools treat search queries in isolation and require developers to assemble the information needed by piecing together the results of multiple queries.

Recently Kochhar *et al.* (2016) performed a study with practitioners about their expectation of automated fault localisation tools. The study explored several crucial parameters, such as trustworthiness, scalability and efficiency. Out of 386 responses, 30% rated fault localization as an “essential” research topic. The study further reveals that around 74% of respondents did not consider a fault localisation session to be successful if it requires developers to inspect more than 5 program elements and that 98% of developers indicated that inspecting more than 10 program elements is beyond their acceptability level. These findings show the importance of positioning relevant files in the top-10, especially in the top-5; otherwise the developers lose confidence.

Parnin and Orso (2011) studied users fixing 2 bugs with Tarantula, a tool that suggests a ranked list of code statements to fix a failed test. Users lost confidence if the ranked list was too long or had many false positives. Users didn't inspect the ranked statements in order, often skipping several ranks and going up and down the list. For some users, the authors manually changed the ranked list, moving one relevant statement from rank 83 to 16, and another from rank 7 to rank 35. There was no statistically significant change in how fast the users found the faulty code statements. This may be further indication that the top-5 ranks are the important ones.

Xia *et al.* (2016) record the activity of 36 professionals debugging 16 bugs of 4 Java apps, each with 20+ KLOC. They divide professionals into 3 groups: one gets the buggy statements in positions 1-5, the other in positions 6-10, the other gets no ranked list. On average, the groups fix each bug in 11, 16 and 26 minutes, respectively. The difference is statistically significant, which shows the ranked list is useful. Developers mention that they look first at the top-5, although some still use all of the top-10 if they're not familiar with code. Some do an intersection of the list with where they think the bug is and only inspect those statements. Although the bug localisation is based on failed/successful tests and the ranked list contains code lines, not files, the study emphasises the importance of the top-5 and how developers use ranked lists.

As I shall present in Chapter 4, my approach equals or improves the top-5 metric on all analysed projects, and as we shall see in Chapter 5, my user study on IR-based localisation of buggy files confirms some of the above findings on test-based localisation of buggy code lines.

2.6 Summary

Early attempts to recover traceability links using IR methods like Vector Space Model (VSM) resulted in high recall, however due to low precision these approaches required manual effort in evaluating the results (Antoniol *et al.*, 2000a). VSM has two limitations: (1) the similarity measure only takes into account the terms that precisely match between the search terms and the terms mapped onto the vector space, (2) large size documents produce poor similarity values because the distance between the terms in the search space increases.

To address the limitations of VSM, previous research has applied Latent Semantic Indexing (LSI) models that resulted in obtaining higher precision values compared to VSM

(Marcus and Maletic, 2003). Nevertheless, LSI is weak compared to VSM when the corpus created after parsing the source code is too small. Therefore, the need to improve results by considering additional sources of information like structural source code file hierarchies is suggested.

The current research also recognised the need for combining multiple analysis approaches to aid in IR to support program comprehension. Techniques like dynamic and static analysis in determining the entry points to investigate the other relevant program elements for maintenance work are exploited (Wilde and Scully, 1995). However, no single IR method consistently provides superior recovery of traceability links (Gethers *et al.*, 2011).

To improve the accuracy of IR models current research also exploited natural language (NL) processing techniques like indexing only the nouns extracted from the project artefacts by relying on the semantic of a term rather then lexical similarity (Shepherd *et al.*, 2005). Despite the advantages gained, one of the limitations of NL techniques is in differentiating between how the developers combine the terms during programming compared to their meaning found in English language vocabulary. Therefore, the need to establish semantic context is raised and the use of contextual (domain specific) representations is proposed (Fry *et al.*, 2008).

Furthermore, to address the limitations of NL in providing the needed context, existing research exploited prepositional phrases (Hill *et al.*, 2008) and use of reference dictionaries containing domain specific vocabulary (Hayase *et al.*, 2011). However, these approaches fail to consider relational clues based on structural information (i.e. OOP inheritance) among program elements. The term relations also point to similarities but fall short in recovering conceptual relations in the absence of domain specific context (Sridhara *et al.*, 2008).

The advantages of utilising ontologies to provide the required contextual information by modelling the knowledge to aid program comprehension in software maintenance are also exploited (Hsi *et al.*, 2003; Petrenko *et al.*, 2008). However, the existing approaches assume that terms extracted from program identifiers are made up of terms that can be directly mapped to one another without being further evaluated (Raţiu *et al.*, 2008). Furthermore, these approaches fall short on utilising the available information like navigating the call-graph efficiently (Hayashi *et al.*, 2010) and expect that applications are programmed by strictly following OOP guidelines (Abebe and Tonella, 2010).

In recent years, research has focused on utilising concept location techniques in bug localisation to aid the challenges of performing maintenance tasks. Concept location also depends on the experience and intuition of the developers, (Rajlich and Wilde, 2002). In case when there is shortage of either, to address the challenges of locating program elements, current research has proposed automated concept and bug location techniques.

Bug reports may describe a situation from the perspective of a problem and may contain the vocabulary of that domain. On the other hand, software applications are coded to address a business domain by providing a solution and contains the vocabulary of that domain (Ratanotayanon *et al.*, 2010).

As the current literature highlights, the IR approaches may consider two strongly related terms via structural information (i.e. OOP inheritance) to be irrelevant (Petrenko and Rajlich, 2013). In turn this may cause one source code file to be at the top of the result list and the other related one at the bottom. To cater for the structural relations, the static call-graph is utilised, however the call-graphs usually contain many dependencies and very deep nodes/edges that make them impractical for search and navigation purposes.

To compensate for the call-graph limitations, the presented approaches utilise pruning logic to remove nodes/edges by clustering the call relations found in the call-graph (Hill *et al.*, 2007). However, one of the challenges pruning exposes is that a source file may be removed or moved to a different group/cluster due to the underlying pruning logic causing inconsistencies when searching for the program elements implementing a concept.

On one hand, techniques like dynamic analysis requires recreating the behaviour reported on the bug report to collect execution trace, on the other hand the techniques like mining software repositories require collecting and analysing large quantities of historical data, making both of these techniques time consuming thus impractical in every-day industrial scale projects (Rao and Kak, 2011).

Recent approaches utilise additional sources of information, e.g. previously fixed bug reports (i.e. similar bugs) and number of times a source file is fixed (i.e. version history), to boost the scoring of the underlying IR model. However, it is claimed that considering similar bug reports earlier than 14 days add no value (Nichols, 2010) and version history older than 20 days decrease the performance (Wang and Lo, 2014).

In addition, recent studies utilised segmentation (Wong *et al.*, 2014) and stack trace analysis (Moreno *et al.*, 2014) as an effective technique to improve bug localisation. However, stack trace may often contain many non-application specific file names, e.g. programming language specific APIs or those belong to other vendors, thus deteriorate the precision of the results.

Furthermore, current studies (Zhou *et al.*, 2012a; Saha *et al.*, 2013; Moreno *et al.*, 2014; Wong *et al.*, 2014; Wang and Lo, 2014) exposed additional challenges associated with bug localisation due to the complexity of OOP (e.g. structure) and the nature of bug report descriptions (e.g. tersely described scenarios), which require weight factors like *alpha, beta* and *lambda* values to be constantly adjusted to obtain optimal results.

2.6.1 Conclusion

In summary, current state-of-the-art IR approaches in bug localisation rely on project history, in particular previously fixed bugs and previous versions of the source code. Existing studies (Nichols, 2010; Wang and Lo, 2014) show that considering similar bug reports up to 14 days and version history between 15—20 days does not add any benefit to the use of IR alone. This suggests that the techniques can only be used where great deal of maintenance history is available, however same studies also show that considering history up to 50 days deteriorates the performance.

Besides, Bettenburg *et al.* (2008) argued that a bug report may contain a readily identifiable number of elements including stack traces, code fragments, patches and recreation steps each of which should be treated separately. The previous studies also show that many bug reports contain the file names that need to be fixed (Saha *et al.*, 2013) and that the bug reports have more terms in common with the affected files, which are present in the names of those affected files (Moreno *et al.*, 2014).

Bettenburg *et al.* (2008) disagree with the treatment of a bug report as a single piece of text document and source code files as one whole unit by existing approaches Poshyvanyk *et al.* (2007); Ye *et al.* (2014); Kevic and Fritz (2014); Abebe *et al.* (2009). Furthermore, the existing approaches treat source code comments as part of the vocabulary extracted from the code files, but since comments are sublanguage of English, they may deteriorate the performance of the search results due to their imperfect nature, i.e. terse grammar (Etzkorn *et al.*, 2001; Arnaoudova *et al.*, 2013).

Therefore, as defined in Chapter 1, the hypothesis I investigate is that *superior results can be achieved without drawing on past history by utilising only the information, i.e. file names, available in the current bug report and considering source code comments, stemming,* *and a combination of both independently, to derive the best rank for each file.*

My research seeks to offer a more efficient and light-weight IR approach, which does not require any further analysis, e.g. to trace executed classes by re-running the scenarios described in the bug reports. Moreover to provide simple usability, which contributes to an *ab-initio* applicability, i.e. from the very first bug report submitted for the very first version and also be applied to new feature requests, without the need for the tuning of any weight factors to combine scores, nor the use of machine learning.

In this Chapter, I described the current IR based research efforts relevant to software maintenance and highlighted their limitations that motivated my research. In the following Chapter, I address my first research question by undertaking a preliminary investigation of eight applications to see whether vocabulary alone provides a good enough leverage for maintenance. I also introduce my novel approach, which directly scores each current file against the given bug report by assigning a score to a source file based on where the search terms occur in the source code file, i.e. class file names or identifiers.

Chapter 3

Relating Domain Concepts and Artefact Vocabulary in Software

In this Chapter, I address RQ1 by undertaking a preliminary study to compare the vocabularies extracted from the projects artefacts (text documentation, bug reports and source code) of eight applications to see whether vocabulary alone provides a good enough leverage for maintenance. More precisely, I investigate to determine whether (1) the source code identifier names properly reflect the domain concepts and (2) identifier names can be efficiently searched for concepts to find the relevant files for implementing a given bug report. I also introduce my novel approach, which directly scores each current file against the given bug report by assigning a score to a source file based on where the search terms occur in the source code file, i.e. class file names or identifiers.

Firstly, I compare the relative importance of the domain concepts, as understood by developers, in the user manual and in the source code. Subsequently, I search the code for the concepts occurring in bug reports, to see if they could point developers to files to be modified. I also vary the searches (using exact and stem matching, discarding stop-words, etc.) and present the performance. Finally, I discuss the implication of my results for maintenance.

In this Chapter, Section 3.1 gives a brief introduction and defines how RQ1 is going to be addressed. In Section 3.2, I briefly describe the current research efforts related to my work. I present my approach in Section 3.3 and the results of my evaluation in Section 3.4. I discuss the results and highlight the threats to validity in Section 3.5. Finally in Section 3.6, I end this Chapter with concluding remarks.

3.1 Introduction

Developers working on unfamiliar systems are challenged to accurately identify where and how high-level concepts are implemented in the source code. Without additional help, concept location can become a tedious, time-consuming and error-prone task.

The separation of concerns coupled with the abstract nature of OOP obscures the implementation and causes additional complexity for programmers during concept location and comprehension tasks (Shepherd *et al.*, 2006). It is argued that OOP promotes the decomposition of concepts into several files scattered across multiple layers of an application's architecture opposed to procedural programming languages where all of the implementation of a concept is usually done in one source file (Shepherd *et al.*, 2006).

Comparison between computer and human interpretation of concepts confronts us with human oriented terms, which may be informal and descriptive, for example, "*deposit the amount into the savings account*", as opposed to compilation unit terms that tend to be more formal and structured, e.g. "i*f (accountType == 'savings') then balance += depositAmount*".

Furthermore, if a program is coded by using only abbreviations and no meaningful words such as the ones from its text documentation, then searching for the vocabulary found in the supporting documentation would produce no results. In such circumstances, the developer — responsible for performing corrective task to resolve a reported bug — is now confronted with the task of comprehending the words represented by the abbreviations before being able to link them to those described in the text document.

Motivated by the challenges described so far, in Chapter 1, I asked my first research question as follows.

RQ1: Do project artefacts share domain concepts and vocabulary that may aid code comprehension when searching to find the relevant files during software maintenance?

To address the first RQ, I investigate if independent applications share key domain concepts and how the shared concepts correlate across the project artefacts, i.e. text documentation (user guide), bug reports and source code files. The correlation was computed pairwise between artefacts, over the instances of the concepts common to both artefacts, i.e. between the bug reports and the user guide, then between the bug reports and the source code, and finally between the user guide and the source code. I argue that identifying any agreement between the artefacts may lead to efficient software maintenance thus ask my first sub-research question as follows.

RQ1.1: *How does the degree of frequency among the common concepts correlate across the project artefacts?*

Furthermore, I am interested in discovering vocabulary similarity beyond the domain concepts that may contribute towards code comprehension. How vocabulary is distributed amongst the program elements of an application as well as recovering traceability links between source code and textual documentation has been recognised as an underestimated area (Deißenböck and Pizka, 2006). In addition to words, i.e. terms, extracted from source code identifiers, I investigate overall vocabulary of all the identifiers because my hypothesis is that despite an overlap in the terms, developers of one application might combine them in completely different ways when creating the identifiers that may confuse other developers, hence ask my following second sub-research question.

RQ1.2: *What is the vocabulary similarity beyond the domain concepts, which may contribute towards code comprehension?*

When searching for source code files implementing the concepts at hand prior to performing software maintenance, the source code file names or bug reports reflecting the domain concepts have a direct impact on the precision and recall performance of the search results. I aim to investigate this impact and tailor the search so that developers have fewer classes to inspect thus ask my third sub-research question as follows.

RQ1.3: *How can the vocabulary be leveraged when searching for concepts to find the relevant files for implementing bug reports?*

In answering these research questions, I compared the project artefacts of eight applications of which two are conceptually related and address the same Basel-II Accord (Basel-II, 2006) in the finance domain. One of these applications is propriety, whose artefacts vocabulary has been studied in my preliminary work (Dilshener and Wermelinger, 2011). The other application is open-source, which is fortunate, as datasets from the finance domain are seldom made available to the public. This enabled me to perform the designed study described in this Chapter.

3.2 Previous Approaches

The case study conducted by Lawrie *et al.* (2006) investigated how usage of identifier naming styles (abbreviated, full words, single letters) assisted in program comprehension. The authors concluded that although full words provide better results than single letters, use of abbreviations are just as relevant and report high confidence. Additionally, the work experience and the education of the developers also play an important role in comprehension. Lawrie *et al.* (2006) lobby for use of standard dictionaries during information extraction from identifier names and argue that abbreviations must also be considered in this process. I investigate further in an environment where recognisable names are used, if the bug report and domain concept terms result in higher quality of traceability between the source code and the text documentation.

Haiduc and Marcus (2008) created a list of graph theory concepts by manually selecting them from the literature and online sources. The authors first extracted identifier names and comments representing those concepts from the source code, and then checked if the words, i.e. terms, extracted from the comments are identifiable in the set of terms extracted from the identifiers. In addition they measured to see the degree of lexical agreement between the terms existing in both sets. The authors concluded that although comments reflect more domain information, both comments and identifiers present a significant source of domain terms to aid developers in maintenance tasks. I also check whether independently elicited concepts, in my case from different domains, occur in identifiers, but I go further in my investigation: I compare different artefacts beyond code, and I check whether the concepts can be used to map bug reports to the code areas to be changed.

In that, my work is similar to that of Antoniol *et al.* (2002). Their aim was to see if the source code files could be traced back to the functional requirements. The terms from the source code were extracted by splitting the identifier names, and the terms from the documentation were extracted by normalising the text using transformation rules. They created a matrix listing the files to be retrieved by querying the terms extracted from the text documents, i.e. functional requirements. The method relied on vector space information retrieval and ranked the documents against a query. The authors compared VSM against probabilistic IR model and measured the performance by applying precision/recall. In vector space model (VSM) tf/idf[1] score is used to calculate the similarity whereas probabilistic model computes the ranking scores as the probability that a document is related to the source code component. Antoniol *et al.* conclude that semi-automatically recovering traceability links between code and documentation is achievable despite the fact that the developer has to

[1] tf/idf (term frequency/inverse document frequency) is explained in Chapter 2, Sub-section 2.2.1

analyse a large number of source files during a maintenance task. My work differs to that introduced by Antoniol *et al.* (2000b) as I aim towards maintenance by attempting to recover traceability between bug reports and source code files, instead of between requirements and code.

Abebe and Tonella (2010) presented an approach that extracted domain concepts and relations from program identifiers using natural language parsing techniques. To evaluate the effectiveness for concept location, they conducted a case study with two consecutive concept location tasks. In the first task, only the key terms identified from the existing bug reports were used to formulate the search query. In the second, the most relevant concept and its immediate neighbour from a concept relationship diagram were used. The results indicate improved precision when the search queries include the concepts. In this study, I also perform concept location tasks using the identified concepts referred by the bug reports. However, I have found situations where the relevant concept terms are not adequate enough to retrieve the files affected by a bug report, thus I demonstrate the use of bug report vocabulary together with the relevant concept terms, additively rather than independently from each other, to achieve improved search results.

3.3 My Approach

The subjects of this study are eight applications (see Table 2.1) of which seven are OSS projects: the IDE tool Eclipse, the aspect-oriented programming library AspectJ, the GUI library SWT, the bar-code tool ZXing, the UML diagraming tool ArgoUML and the servlet engine Tomcat. These applications were chosen because other studies had used them (see Section 2.4 in Chapter 2), which allowed me to compare performance of my results in Chapter 4 against the others. Additionally, I used an OSS financial application Pillar1 and a proprietary financial application (due to confidentiality) referred as Pillar2. Both Pillar1 and Pillar2 implement the financial regulations for credit and risk management defined by the Basel-II Accord (Basel-II, 2006).

Pillar1[2] is a client/server application developed in Java and Groovy by Munich Re (a re-insurance company). The bug report documents are maintained with JIRA[3], a public tracking tool.

[2] http://www.pillarone.org

[3] https://issuetracking.intuitive-collaboration.com

Pillar2 is a web-based application developed in Java at our industrial partner's site. It was in production for 9 years. The maintenance and further improvements were undertaken by five developers, including in the past myself, none of us being part of the initial team. The bug report documents were maintained by a proprietary tracking tool. As Table 2.1 shows, for both Pillar1 and Pillar2 I only have a small set of closed bug reports for which I also have the source code version on which they were reported. Neither application is maintained anymore.

Prior to starting the maintenance task, a developer who is new to the architecture and vocabulary of these applications is faced with the challenge of searching the application artefacts to identify the relevant sections. In order to assist the developer, I attempt to see what information can be obtained from the vocabulary of the application domain.

3.3.1 Data Processing

I obtained the datasets for AspectJ, Eclipse, SWT and ZXing from the authors of Zhou *et al.* (2012a), Tomcat from the authors of Ye *et al.* (2014) and ArgoUML from the authors of Moreno *et al.* (2014). For Pillar1 and Pillar2, I have manually constructed the datasets from their online resources. The datasets are also available online: AspectJ, SWT and ZXing at BugLocator web-site[4], Eclipse at BRTracer web-site[5], ArgoUML at LOBSTER web-site[6] and Pillar1 at ConCodeSe web-site[7]. The Tomcat dataset is available directly from the authors of Ye *et al.* (2014). Pillar2 is a proprietary software application thus its dataset is not publicly available. Each dataset, consisting of source code files and bug reports with their known affected files identified as described in Zhou *et al.* (2012a), Ye *et al.* (2014) and Moreno *et al.* (2014).

I created a toolchain to process the dataset for all 8 applications. My tool, called ConCodeSe (Contextual Code Search Engine), consists of data extraction, persistence and search stages as illustrated in Figure 3.1 and discussed in detail below (Sub-sections 3.3.1.1, 3.3.1.2 and 3.3.1.3). ConCodeSe utilises state of the art data extraction, persistence and search APIs (SQL, Lucene, Hibernate[8]). Each dataset was processed to extract words from project artefacts, e.g source code files and bug reports, to create a corpus. Searches were then undertaken automatically for each dataset using the domain concepts of each dataset and vocabulary of

[4] https://code.google.com/archive/p/bugcenter/wikis/BugLocator.wiki

[5] https://sourceforge.net/projects/brtracer/files/?source=navbar

[6] http://www.cs.wayne.edu/~severe/icsme2014/#dataset

[7] http://www.concodese.com/research/dataset

[8] http://www.hibernate.org

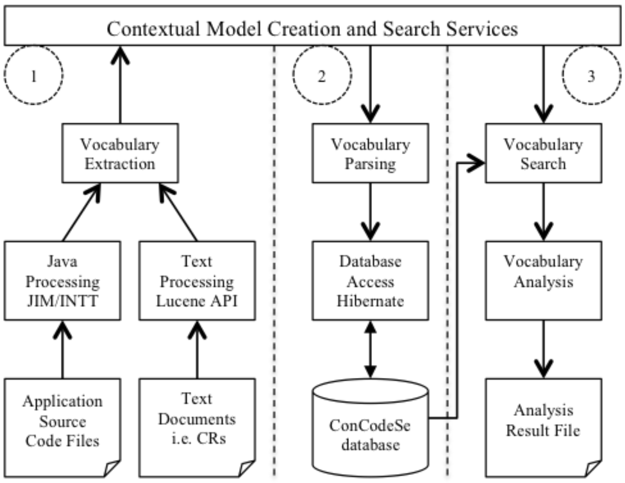


Figure 3.1: ConCodeSe Data Extraction, Storage and Search stages

the bug reports as search terms. I use the word 'term' because it also covers non-English words, prefixes, abbreviations and business terminology (Haiduc and Marcus, 2008).

Figure 3.1 shows that the left hand side (1) represents the extraction of terms from the source code files and from the textual documents, e.g. bug reports and user guide. The middle part (2) shows the storage of the extracted terms. Finally, in the search stage (3), two types of search are performed: one for the occurrence of concepts in the project artefacts i.e. bug reports, user guide and source files, and the other search is for the files affected by the bug reports.

3.3.1.1 Data Extraction Stage

In the first stage, information is extracted from the project artefacts, i.e. source code files and textual documentation, i.e. user guide, domain concepts and bug reports, in following process steps.

***Source code file processing*:** The Java source code files are parsed using the source code mining tool JIM[9] (Butler, 2016), which automates the extraction and analysis of identifiers, i.e. class, method and field names, from source files. It parses the source code files, extracts the identifiers and splits them into terms, using the INTT[10] tool (Butler *et al.*, 2011) within JIM. INTT uses camel-case, separators and other heuristics to split at ambiguous boundaries, like digits and lower case letters. For example, camel-case style identifier *financeEntityGrantFirCorrelation* extracted from the Pillar2 source file *K4MarketCapitalRisk.java*, gets split into 5 terms: *finance, entity, grant, fir* and *correlation.*

[9] https://github.com/sjbutler/jim
[10] https://github.com/sjbutler/intt

Many studies are done on identifier splitting techniques and conclude that accurate natural language information depends on correctly splitting the identifiers into their component words (Hill *et al.*, 2013). The empirical study of state-of-the-art identifier splitting techniques conducted by Hill *et al.* independently shows that INTT is on par with other splitting techniques.

Since Pillar1 is partially programmed in Groovy, I developed a module in Java to parse, extract and split the identifiers from the Groovy source code files using steps similar to JIM and INTT as described earlier.

***Bug reports processing*:** To process the text in the bug reports and extract terms from the vocabulary, I developed a text tokenisation module by reusing Lucene's *StandardAnalyzer* because it tokenises alphanumerics, acronyms, company names, email addresses, etc. The text tokenisation module includes stop-word removal to filter out those words without any significant meaning in English, e.g. 'a', 'be', 'do', 'for'. I used a publicly available stop-words list[11] to filter them out. For example, a bug report description as "*We need to create a component for calculating correlation.*", would get tokenised into 9 terms and after stop-word removal only 5 terms are considered: *need, create, component*, *calculating* and *correlation.*

***User guide processing*:** Additionally, I obtained the user guide of each application from their online annex[12] and saved it as a text file to ignore images, graphics, and tables. I next extracted the words from the resulting text documents by applying the same process as described above in the bug reports processing step. For Pillar2, since it is proprietary, confidential information, names, email addresses and phone numbers were then manually removed from its text file.

***Comments processing*:** The text tokenisation module (developed in bug reports processing step) is also used to extract terms (by tokenising words and removing stop-words) from the source code file comments. The comments are generally coded in two styles; header and inline (Etzkorn *et al.*, 2001). In addition, the OOP guidelines define two comment formats: Implementation comments (i.e. Block, Singe-Line, Trailing, End-of-Line) delimited by **"/*...*/"**, and **"//..."**. Documentation comments (known as "doc comments") delimited by **"/**...*/"** (Oracle, 1999). Based on these delimiter conventions a source code line is identified as comment and the terms are extracted similar to the process described above for the bug reports. For example, the terms *here, block* and *comment* are extracted from the

[11] http://xpo6.com/list-of-english-stop-words/

[12] http://www.concodese.com/research/dataset

```
/*
* Here is a block comment.
*/
if (customerAccountType == SAVINGS_ACCOUNT) {
.... <rest of the code is not shown>
```

Figure 3.2: Block style comment example used in a source code file

block comment illustrated in Figure 3.2.

Domain concepts processing**:** Furthermore, I compiled the following list of concepts used in each application's domain based on my experience and online sources. The full list of concepts are included in Appendix E.

1. For AspectJ, aspect oriented programming (AOP)[13]

2. For Eclipse, integrated development environment (IDE)[14]

3. For SWT, graphical user interface (GUI)[15]

4. For ZXing, barcode imaging/scanning[16]

5. For ArgoUML, unified modelling language[17]

6. For Tomcat, servlet container[18]

7. For Pillar1 and Pillar2, credit and risk management[19]

The concepts are made up of multiple words, e.g. "Investment Market Risk", "Market Value Calculation", "Lambda Factors". The list was distributed as a Likert type survey with a "strongly agree" - "strongly disagree" scale amongst seven other developers and two business analysts who were experts within the domains, to rate each concept and to make suggestions, in order to reduce any bias I might have introduced. Only five developers and one business analyst completed the survey. The turn around time for the whole process was less than 5 days. After evaluating the survey results and consolidating the suggestions, using the text tokenisation module, I extracted the unique single words (like 'market' and 'lambda')

[13] https://www.cs.utexas.edu/\%7erichcar/aopwisr.html

[14] http://www.qnx.com/download/feature.html?programid=21031

[15] http://www.ugrad.cs.ubc.ca/\%7ecs219/CourseNotes/Swing/intro-GUIConcepts.html

[16] http://www.amerbar.com/university/dictionary.asp

[17] http://www.uml-diagrams.org/uml-object-oriented-concepts.html

[18] https://docs.oracle.com/cd/E19276-01/817-2334/Glossary.html

[19] http://www.bis.org/publ/bcbs128.htm

occurring in the concepts as basis for further analysis. Henceforth, those words are called concepts in this Chapter.

***Word stem processing*:** During the previous processing steps, the text tokenisation module also performs stemming of the terms to their linguistic root in English using Lucene's implementation of Porter's stemmer (Porter, 1997). For example, term *calculating* extracted from the bug report description example given earlier in the bug report processing step and the term *calculation* extracted from the concept "Market Value Calculation" during the domain concepts processing step are stemmed to their root *calcu*.

3.3.1.2 Persistence Stage

In the second stage, the information gathered in the previously described data extraction steps are persisted in following steps.

Source code files: The information extracted from the source code files are stored in a Derby [20] relational database within JIM. This includes the terms extracted from the identifiers (i.e. class, method and field names), comments and their source code locations. For example, recall in the source code file processing step, the terms *"finance, entity, grant, fir* and *correlation*" were extracted from the identifier *"financeEntityGrantFirCorrelation"* found in the source file *"K4MarketCapitalRisk.java"*. These details, i.e. terms, identifier and source code file name, are stored in JIM Database tables Component Words, Identifiers and Files respectively (see Figure 3.3).

Textual documentation: The information about the textual artefacts, i.e. user guides, domain concepts and bug reports, are also stored into the same Derby database. For this, I extended the JIM database with additional tables as illustrated in Figure 3.3. The terms extracted from the textual documentation are stored into the existing JIM Component Words table and the details (e.g. name, content) of the document being processed are saved into the respective database tables User Guide, Domain Concepts or Change Requests (see Figure 3.3). For example, the financial domain concept *"Investment Market Risk"* is stored in the Domain Concepts table and its terms *investment, market* and *risk* are stored in the Component Words table. This table also contains the source code comment terms.

Referential integrity: Subsequently, all of the terms are linked to their stem words and to their source of origin, e.g. user guide, domain concepts, and/or bug reports. This

[20] http://db.apache.org/derby

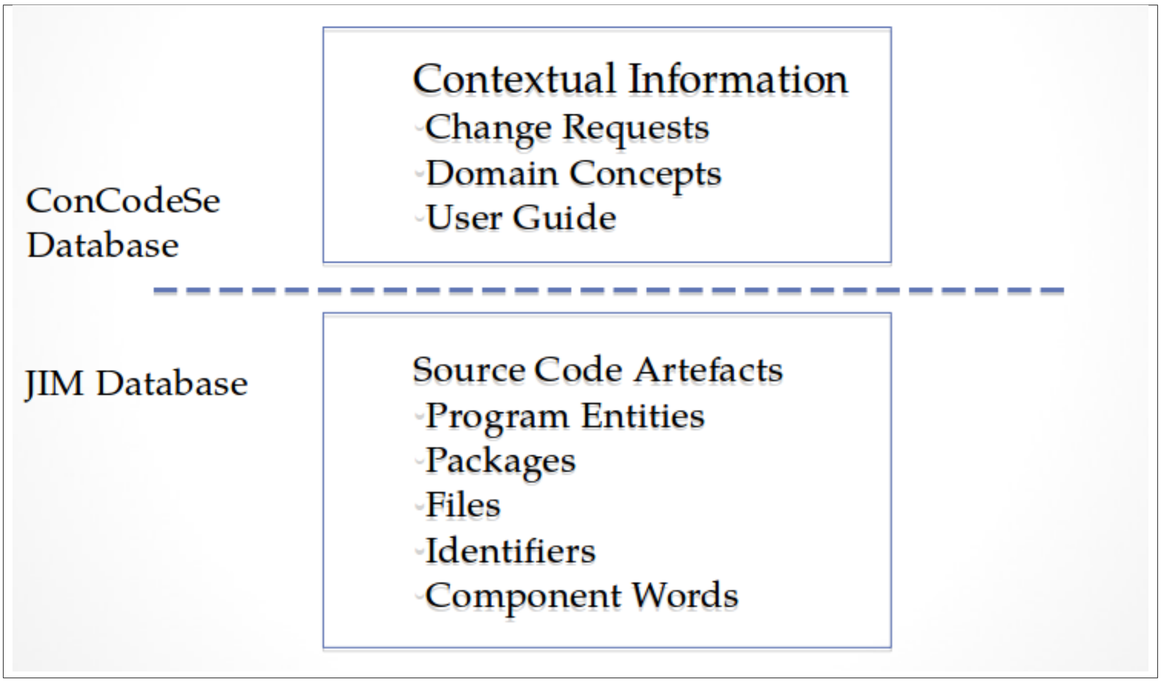


Figure 3.3: ConCodeSe database table extensions in addition to JIM database tables

allows the Component Words table to sustain referential links among the artefacts to provide strong relational clues during search tasks as seen in Figure 3.4. For example, to find out all the concept terms occurring in a change request, i.e. bug report, involves selecting all the terms of a bug report from the Component Words table and then reducing the resulting list of terms to those that have a referential link to the Domain Concepts table. Figure 3.5 illustrates the pseudo code of selecting only domain concept terms that occur in Pillar1 bug report #PMO-2012, which results in 2 concept terms instead of 18 terms.

Since the source code file names changed to resolve the reported bug are already documented in the bug reports processed, a referential link from the persisted bug reports in the Change Requests table to the relevant file name(s) stored in the Files table is also created during this step as well.

To establish the referential constraints, the contextual model, i.e. the underlying database tables (see Appendix A), is created in the following order.

1. JIM – Java source code terms processing
2. Contextual model table creation
3. Groovy - source code file terms processing
4. Change requests/Bug reports processing
5. Domain concepts processing

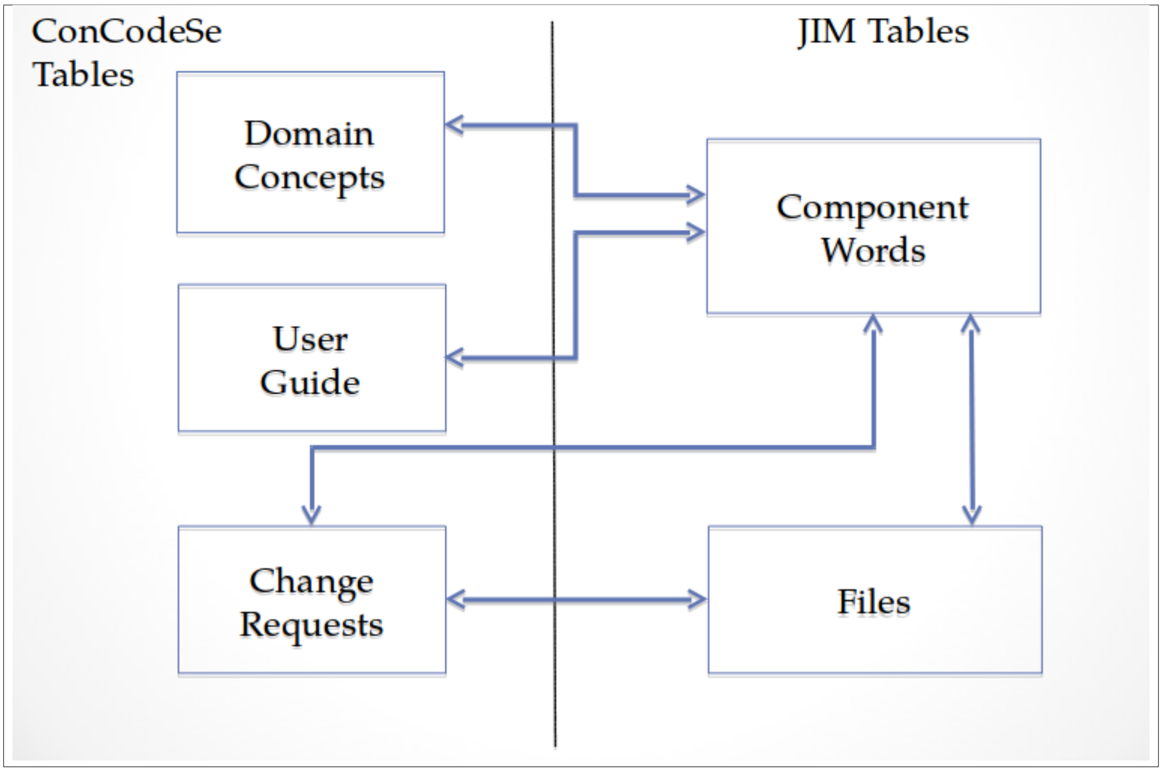


Figure 3.4: Referential relations linking the information stored in ConCodeSe and JIM database tables.

6. User guide processing

7. Comments processing

8. Word stem processing

3.3.1.3 Search Stage

Finally, for the third stage of the process, I developed a search module in Java to read the concept terms stored, and then run SQL queries to (1) search for the occurrences of the concepts in the three artefacts (bug reports, user guide and code) and (2) search for all files that included the terms matching the concepts found in bug reports. The second search variant ranks all the files for each bug report as explained in the next Sub-section 3.3.2.

For this stage, I also coded a module in Java to select all the file names and their terms from the JIM database and create a repository of terms per source code file, which is then

```
select change_request_key, component_word from Change_Requests_table
join Component_Words_table using component_word_key and change_request_key
join Domain_Concepts_table using domain_concept_key and component_word_key
where change_request_key = 'PMO-2012'
```

Figure 3.5: Pseudo code example of an SQL select statement for finding domain concepts that occur in bug report #PMO-2012

indexed by using Lucene's VSM. The idea behind VSM is that the more times a query term appears in a document relative to the number of times the term appears in all the documents in the collection, the more relevant that document is to the query. Vectors represent queries (concepts or bug reports) and documents (source code files). Each element of the vector corresponds to a word or term extracted from the query's or document's vocabulary. The relevance of a document to a query can be directly evaluated by calculating the similarity of their word vectors. I chose to use VSM over more advanced IR techniques such as LSI because it is simpler and known to be more effective as shown by Wang *et al.* (2011).

The ranked search results are saved in a spreadsheet for additional statistical analysis like computing the Mean Average Precision (MAP) and Mean Reciprocal Rank (MRR) values (Boslaugh and Watters, 2008).

MAP provides a single-figure measure of quality across recall levels. Among evaluation measures, MAP has been shown to have especially good discrimination and stability (Manning *et al.*, 2008). The MAP is calculated as the sum of the average precision value for each bug report divided by the number of bug reports for a given project. The average precision (AP) for a bug report is the mean of the precision values obtained for all affected files listed in the result set and computed as in equation 3.1. Then the MAP for a set of queries is the mean of the average precision values for all queries, which is calculated as in equation 3.2.

$$AP(Relevant) = \frac{\sum_{r \in Relevant} Precision(Rank(r))}{|Relevant|} \tag{3.1}$$

$$MAP(Queries) = \frac{\sum_{q \in Queries} AP(Relevant(q))}{|Queries|} \tag{3.2}$$

MAP is considered to be optimal for ranked results when the possible outcomes of a query are 5 or more (Lawrie, 2012). As Table 3.1 shows, very few bug reports have at least 5 affected files, but I still use the MAP to measure the performance of my approach for comparison with previous approaches in Chapter 4. As I will show in Sub-section 4.4.1.5, for all eight projects, I achieve the best MAP.

MRR is a metric for evaluating a process that produces a list of possible responses to a query (Voorhees, 2001). The reciprocal rank *RR(q)* for a query q is the inverse rank of the first relevant document found and computed as in equation 3.3. Then the MRR is the average of the reciprocal ranks of results of a set of queries, and calculated as in equation 3.4.

$$RR(q) = \begin{cases} if\ q\ retrieves\ no & \\ relevant\ documents & 0 \\ otherwise & {}^{1}/_{TopRank(q)} \end{cases} \tag{3.3}$$

$$MRR(Queries) = \frac{\sum_{q \in Queries} RR(q)}{|Queries|} \tag{3.4}$$

On the other hand, MRR is known to measure the performance of ranked results better when the outcome of a query is less than 5 and best when just 1 (Lawrie, 2012), which is the majority of cases for the datasets under study (Table 3.1). The higher the MRR value, the better the bug localisation performance as I will show in my approach.

Table 3.1: Number of affected files per bug report (BR) in each project under study

Project	**BRs with only 1 file**	**BRs with 2-4 files**	**BRs with >= 5 files**
AspectJ	71	149	66
Eclipse	1525	1066	484
SWT	59	32	7
ZXing	14	5	1
Tomcat	690	281	85
ArgoUML	42	32	17
Pillar1	10	6	11
Pillar2	0	1	11

3.3.2 Ranking Files

In addition to searching for the occurrence of concepts in the project artefacts, I also use the concepts that may be referred by a bug report to search for the files relevant for resolving the reported bug. The results of this search ranks the files in the order of relevance from most to least likely based on my scoring logic as follows.

Given a bug report and a source file, my approach computes two kinds of scores for the file: (1) a lexical similarity score and (2) a probabilistic score given by VSM, as implemented by Lucene. The two scorings are done with four search types, each using a different set of terms indexed from the concept and the source file:

1. Full terms from the file's code and the concept.

Table 3.2: Sample of ranking achieved by all search types in SWT project

BR #	**Affected Java file**	**Con CodeSe**	**1: Full/ code**	**2: Full/ all**	**3: Stem/ code**	**4: Stem/ all**
100040	Menu	**2**	484	3	29	**2**
79107	Combo	**3**	26	29	**3**	8
84911	FileDialog	**5**	**5**	39	6	56
92757	StyledTextListener	**3**	4	**3**	75	72

2. Full terms from the file's code and comments and the concept.

3. Stemmed terms from the file's code and the concept.

4. Stemmed terms from the file's code and comments and the concept.

For each of the 8 scorings, all files are ranked in descending order. Files with the same score are ordered alphabetically, e.g. if files F and G have the same score, F will be ranked above G. Then, for each file I take the best of its 8 ranks. If two files have the same best rank, then I compare the next best rank, and if that is tied, then their 3rd best rank etc. For example, if file B's two best ranks (of the 8 ranks each file has) is 1 and 1, and if file A's two best ranks are 1 and 2, B will be ranked overall before A.

The rationale for this approach is that during experiments, I noticed that when a file could not be ranked among the top-10 using the full terms from the file's code, it was often enough to use associated comments and/or stemming. As shown in Table 3.2, for SWT's bug report (BR) #100040, the affected Menu.java file has a low rank (484th) using the first type of search. When the search includes comments, stemming or both, it is ranked at 3rd, 29th and 2nd place respectively. The latter is the best rank for this file and thus returned by ConCodeSe (3rd column of the table).

There are cases when using comments or stemming could deteriorate the ranking, for example because it helps irrelevant files to match more terms with a bug report and thus push affected classes down the rankings. For example, in BR #92757, the affected file StyledTextListener.java is ranked much lower (75th and 72nd) when stemming is applied. However, by taking the best of the ranks, ConCodeSe is immune to such variations.

The lexical similarity scoring and ranking is done by a function, which takes as arguments a bug report and the project's files and returns an ordered list of files. The function assigns a score to the source file based on where the search terms (without duplicates) occur in the source file. Search terms can be the domain concepts referred by the bug report or all the

Algorithm 3.1 *scoreWithFileTerms*: Each matching term in the source file increments the score slightly

```
input: file: File, queryTerms: concept or bug report terms:
output: score: float
for each query_term in queryTerms do
  if (query_term = file.name) then
    score := $score$ + 2 return score
  else
    if (file.name contains query_term) then
      score := $score$ + 0.025
    else
      for each doc_term in file.terms do
        if (doc_term = query_term ) then
          score := $score$ + 0.0125
        end if
      end for
    end if
  end if
end for
return score
```

vocabulary of the bug report. The function is called 4 times, for each search type listed previously. Each occurrence of each search term in the file increments slightly the score, as shown in function *scoreWithFileTerms* (Algorithm 3.1). As explained before, the bug report or domain concept terms and the source file terms depend on whether stemming and/or comments are considered.

The rationale behind the scoring values is that file names are treated as the most important elements and are assigned the highest score. When a query term is identical to a file name, it is considered a full match (no further matches are sought for the file) and a relatively high score (adding 2) is assigned. The occurrence of the query term in the file name is considered to be more important (0.025) than in the terms extracted from identifiers, method signatures or comments (0.0125).

Pure textual similarity may not be able to distinguish the actual buggy file from other files that are similar but unrelated (Wang *et al.*, 2015). Therefore, the scoring is stopped upon full match because continuing to score a file using all its terms may deteriorate the performance of the results: if a file has more matching terms with the bug report, it is scored higher and falsely ranked as more relevant, which is also observed by Moreno *et al.* (2014). For example, one of the affected source code files (i.e. SWT.java) of SWT bug #103863 is ranked in the 2$^{\text{nd}}$ position but if I continue with scoring all the terms available in the source file, then SWT.java is ranked in the 3$^{\text{rd}}$ position. The reason is the irrelevant file Control.java has more matching terms with the bug report and is ranked in the 1st position causing the rank of the relevant file to degrade. This is prevented when the scoring is stopped because the bug report search term "*control*" matches the file name early in the iterative scoring logic, the current score is incremented by 2 and the file's low score remains unchanged. In the case of scoring SWT.java, the search term "*swt*" matches the file name much later, hence the accumulated current score is much higher prior to incrementing it by 2 and stopping. Thus the irrelevant file Control.java is ranked lower than the relevant file SWT.java.

The score values were chosen by applying the approach on a small sized training dataset, consisting of randomly selected 51 bug reports from SWT and 50 bug reports from AspectJ projects, i.e. (50+51)/4665=2.17% of all bug reports, and making adjustments to the scores in order to tune the results for the training dataset.

3.4 Evaluation of the results

In this Section, I present my analysis results obtained by searching the concept occurrences and the vocabulary coverage of the terms across the project artefacts. Using statistical Spearman tests, I demonstrate how the concepts correlate in the project artefacts. In addition, I show the results of searching for source code class file names matching the concepts referred in the bug reports as may be performed by developers while executing a maintenance tasks.

3.4.1 RQ1.1: How Does the Degree of Frequency Among the Common Concepts Correlate Across the Project Artefacts?

To answer my first research question, I first investigated whether independent applications share key domain concepts. Again identifying any agreement between the artefacts may lead to effective maintenance. Table 3.3 shows the number of total words and the unique words found in each project's artefacts. In case of domain concepts, as they are made up of multiple words, e.g. "Investment Market Risk", the table first reports the total number of concept words followed by the number of unique words, for example, the Pillar1 and Pillar2 applications implement the 50 financial domain concepts of Basel-II accord, which are made up of 113 words in total and contain 46 unique words.

Searching for the exact occurrences of the concepts in the artefacts, I found that the most frequently occurring (top-1) concept to be the same one in the bug reports (BR) as in the user guides (UG) for three projects: AspectJ, ArgoUML and Pillar2. However, in these

Table 3.3: Total and unique number of word occurrences in project artefacts

Number of words in	AspectJ	Eclipse	SWT	ZXing	Tomcat	ArgoU	Pillar1	Pillar2
Source code (SRC)	80480	460949	28322	14016	71743	29270	92499	14190
Unique SRC words	5153	7770	2477	1772	4111	2420	4143	701
Bug reports (BR)	26233	186032	5841	1867	41331	5717	1031	1652
Unique BR words	3491	12402	1496	765	5848	1345	433	189
Concepts (DC)	18	144	94	103	114	66	113	113
Unique DC words	18	104	67	84	63	40	46	46
User guide (UG)	2534	58132	4022	3383	41271	55799	22541	14197
Unique UG words	811	3029	1068	989	3883	4427	3630	722

projects the top-1 concept in the source code (SRC) was different compared to BR and UG. For example, in AspectJ the concept of ***aspect*** is the top-1 concept in BR and in UG but it is the 2nd frequently occurring concept in SRC. Similarly in Pillar2, ***market*** is the top-1 concept in BR and UG but moves to 2nd position in SRC. For ArgoUML ***diagram*** is the top-1 concept and becomes the 5th frequently occurring in SRC. In the case of other projects, all three artefacts (BR, UG, SRC) has different top-1 frequently occurring concept.

Subsequently, searching for stemmed occurrences of the concepts resulted the same top-1 concept in BR and UG also for Eclipse (***file***) and ZXing (***code***), however at the same time it changed the common top-1 concept for Pillar2. Analysing the reason why the concept of *market* was no longer the common top-1 concept in BR and UG revealed that stemming returned additional variations for the concept ***value*** in BR and ***calculation*** in UG, thus changed the relative ranking of the common concepts. For example, there are 513 exact occurrences of concept *calculation* in the UG, but searching for the concept's stem '***calcul***' returns 207 additional instances (for *calculate*, *calculated*, *calculating* and *calculation*), thus changes the relative ranking of *calculation* to top-1 and moves *market* to 2nd position.

Interestingly, I noticed that Pillar1 and Pillar2, two independent applications implementing the concepts of the same financial domain, have the same most frequently occurring concepts**,** ***value*** and ***label***, ranked at the 3rd and 18th position in their source codes (SRC) respectively. In Appendix B, Tables B.1 and B.2 list the top-5 concepts occurring across all the projects artefacts.

As the majority of projects share the same frequently occurring concepts between the BRs and the UGs indicate the vocabulary alignment among these documents written in natural language, i.e. English. On the other hand, it is acceptable to have different frequently occurring terms between the SRC and other artefacts as programming languages have their own syntax and grammar.

I found that while each concept occurred in at least one artefact, only a subset of the

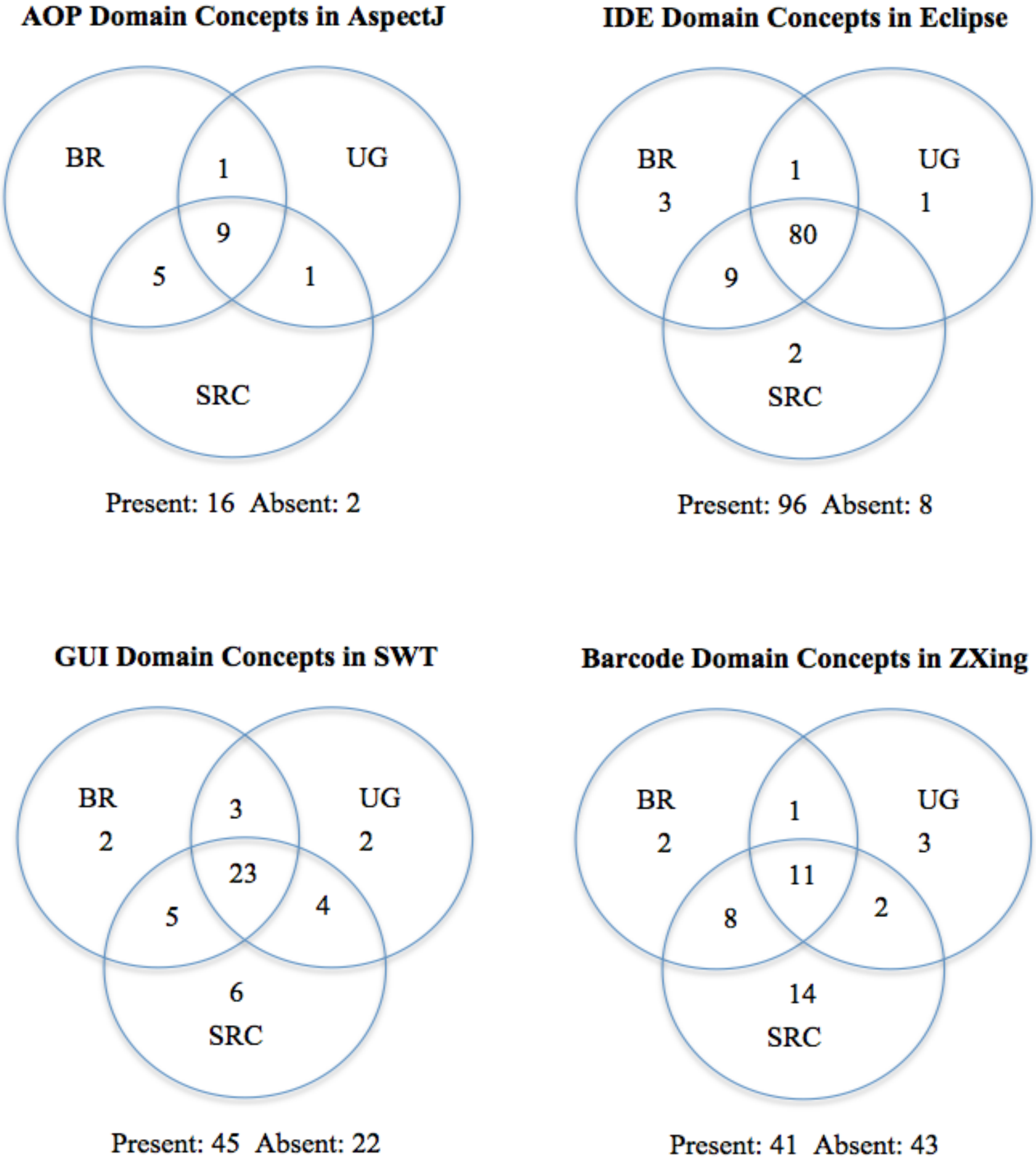


Figure 3.6: Distribution of concepts among AspectJ, Eclipse, SWT and ZXing projects

concepts occurred in all three artefacts (see Figures 3.6 and 3.7). The concept occurrences in single project artefacts, e.g. BR or UG on its own, is also very low and for some projects none at all. This also applies for the combination of two project artefacts, e.g. BR and UG. However, for Pillar1 and Pillar2 projects almost half of the concepts (21/46 or 46%, and 20/46 or 43%, respectively) occur in the intersecting set between the UG and the SRC (Figure 3.7). This is an indication towards good alignment between the source code and the user guide.

On the contrary, in ZXing more than half of the barcode domain concepts are absent from the artefacts. One of the reasons for this is that ZXing is a small size bar-code application with 391 source code files, thus does not implement all of the bar-code domain concepts. The

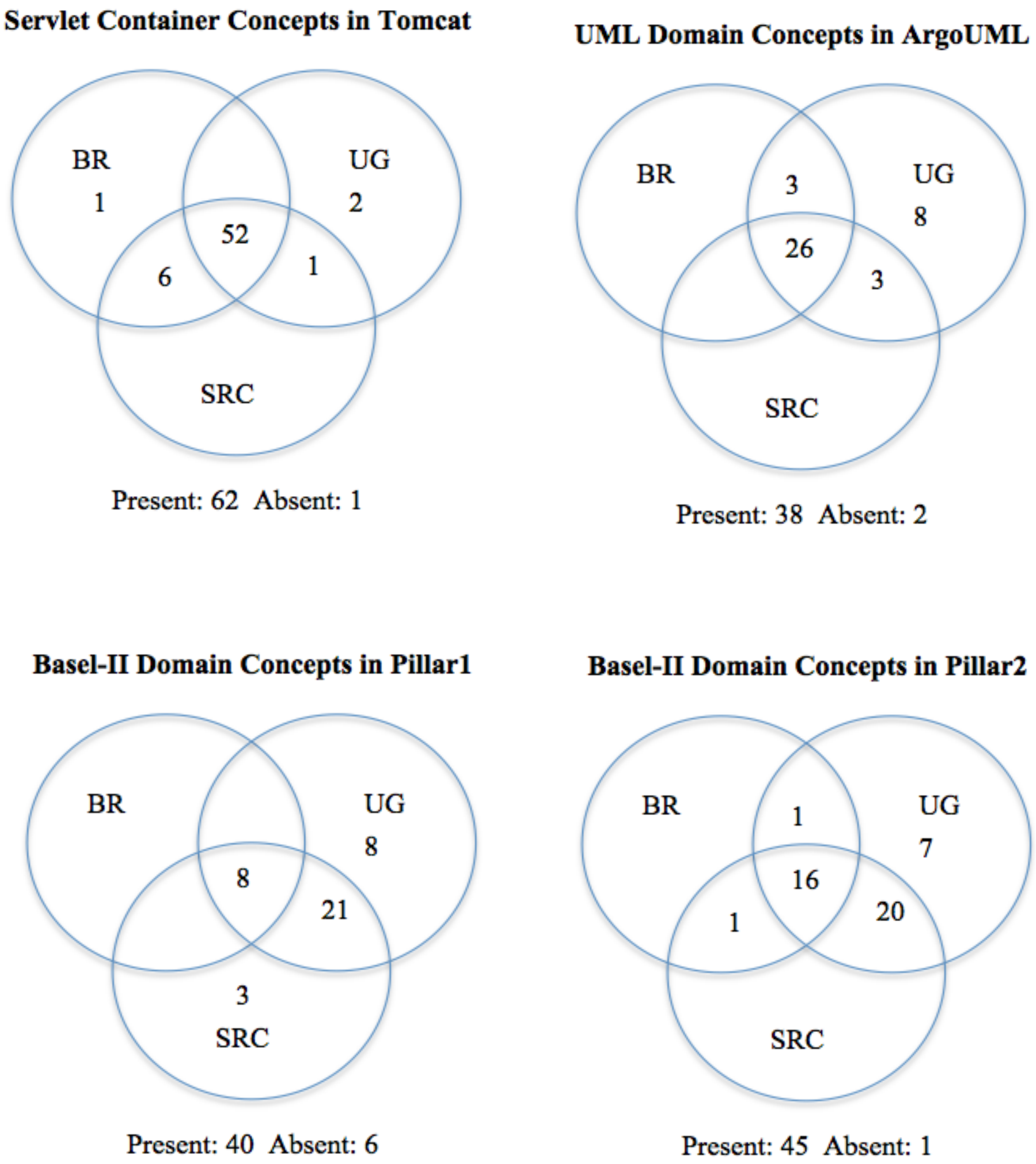

Servlet Container Concepts in Tomcat
BR
1
UG
2
52
6
1
SRC
Present: 62 Absent: 1
UML Domain Concepts in ArgoUML
BR
3
UG
8
26
3
SRC
Present: 38 Absent: 2
Basel-II Domain Concepts in Pillar1
BR
UG
8
8
21
3
SRC
Present: 40 Absent: 6
Basel-II Domain Concepts in Pillar2
BR
1
UG
7
16
1
20
SRC
Present: 45 Absent: 1

Figure 3.7: Distribution of concepts among Tomcat, ArgoUML, Pillar1 and Pillar2 projects

common concepts between the BR and the SRC alone is also very low, and for ArgoUML and Pillar1, it is none. This may lead to inconsistencies when searching for relevant source files unless the common concepts between those two artefacts are also the same ones as those between the UG, i.e. the intersecting concepts between the artefacts.

Additionally, I noticed that all three artefacts share a very high number of common domain concepts, e.g. 9/18 or 50% in AspectJ, 80/96 or 76% in Eclipse, 52/63 or 82% in Tomcat and 26/40 or 65% in ArgoUML. This is a good indicator for leveraging domain concepts to search for the source files that may be relevant for a bug report.

Furthermore, I noticed that 6 out of 46 Basel-II financial domain concepts are missing in Pillar1. Through investigating the reasons, I found that this is due to the naming conventions,

Table 3.4: Spearman rank correlation of common concepts between the project artefacts using exact concept terms

Project	**Between ==>**	**Exact Search (full concept term)**		
		BR and User Guide	**BR and Source**	**User Guide and Source**
AspectJ	Correlation	0.2590	0.5247	-0.0344
	p value	0.3327	**0.0369**	0.8995
Eclipse	Correlation	0.6174	0.7237	0.5380
	p value	**0**	**0**	**0**
SWT	Correlation	0.5351	0.4261	0.3783
	p value	**0.0002**	**0.0035**	**0.0104**
ZXing	Correlation	0.4910	0.4096	0.1404
	p value	**0.0011**	**0.0078**	**0.3814**
Tomcat	Correlation	0.4596	0.5938	0.6375
	p value	**0.0002**	**0**	**0**
ArgoUML	Correlation	0.7214	0.8037	0.6963
	p value	**0**	**0**	**0**
Pillar1	Correlation	0.1958	0.3136	0.4741
	p **value**	0.2260	**0.0488**	**0.0020**
Pillar2	Correlation	0.4146	0.4434	0.5064
	p value	**0.0046**	**0.0023**	**0.0004**

for example the financial concept '*pd/lgd*' is referenced as '*pdlgd*' and '*sub-group*' as '*subgroup*' in the projects artefacts. In Pillar2 only 36/45 or 80% of concepts also occur both in the code and the documentation. Since the latter is consulted during maintenance, this lack of full agreement between both artefacts may point to potential inefficiencies.

3.4.1.1 Correlation of common concepts across artefacts

Subsequently, I wanted to identify if the concepts also have the same degree of importance across artefacts, based on their occurrences. For example, among those concepts occurring both in the code and in the guide, if a concept is the n-th most frequent one in the code, is it also the n-th most frequent one in the guide? I applied Spearman's rank correlation coefficient, to determine how well the relationship between two variables in terms of the ranking within each artifactual domain can be described (Boslaugh and Watters, 2008). The correlation was computed pair-wise between artefacts, over the instances of the concepts common to both artefacts, i.e. between the bug reports and the user guide, then between the bug reports and the source code, and finally between the user guide and the source code. Table 3.4 shows the results using the online Wessa statistical tool (Wessa, 2016).

Except for AspectJ, the correlation between the artefacts is moderate and statistically significant (p-value < 0.05). In case of AspectJ, the correlation among all of the artefacts is

Table 3.5: Spearman rank correlation of common concepts between the project artefacts using stemmed concept terms

Project	**Between ==>**	**Stem Search (stemmed concept term)**		
		BR and User Guide	**BR and Source**	**User Guide and Source**
AspectJ	Correlation	0.1643	0.4382	0.0284
	p value	0.5431	0.0895	0.9169
Eclipse	Correlation	0.5392	0.5526	0.5264
	p value	**0**	**0**	**0**
SWT	Correlation	0.5208	0.4063	0.4549
	p value	**0.0002**	**0.0042**	**0.0012**
ZXing	Correlation	0.4585	0.4741	0.1495
	p value	**0.0020**	**0.0013**	0.3385
Tomcat	Correlation	0.4424	0.5775	0.5611
	p value	**0.0004**	**0**	**0**
ArgoUML	Correlation	0.7310	0.8291	0.7493
	p value	**0**	**0**	**0**
Pillar1	Correlation	0.3268	0.4225	0.5517
	p value	**0.0370**	**0.0059**	**0.0002**
Pillar2	Correlation	0.4544	0.5047	0.5897
	p value	**0.0020**	**0.0005**	**3.0E-05**

weak and not statistically significant (p-value > 0.05) except between BR and SRC where the correlation is moderate and statistical significant. I also note that for exact search type the correlation between BR and UG for Pillar1, and between UG and SRC for ZXing is weak and statistically not significant. This is because there are very few common concepts with exact occurrences in bug reports.

For stem search, Table 3.5, the correlation decreases in most cases except for ArgoUML, Pillar1 and Pillar2 where for exact search, the correlation is low, however for stem search, the correlation becomes stronger. This is because the stemming increased the number of matched words, i.e. the stemming of words into their root increases their instances in the artefacts, hence changed the relative ranking of the common concepts and the Spearman correlation became stronger and statistically more relevant as Table 3.5 shows compared to Table 3.4.

3.4.2 RQ1.2: What is the Vocabulary Similarity Beyond the Domain Concepts, which may Contribute Towards Code Comprehension?

By answering my first sub-research question in previous Sub-section, I identified that the projects artefacts share common domain concepts and the conceptual agreement between the artefacts correlate well. My second sub-research question investigates any vocabulary similarity beyond the domain concepts that may contribute towards code comprehension.

Table 3.6: Common terms found in source files of each application across all projects

Application	**Pillar2**	**Pillar1**	**ArgoU**	**Tomcat**	**ZXing**	**SWT**	**Eclipse**	**AspectJ**
Unique words	701	4143	2420	4111	1772	2477	7770	5153
AspectJ	531 76%	1899 46%	1503 62%	2166 53%	1117 63%	1333 54%	3128 40%	-
Eclipse	582 83%	2525 61%	1859 77%	2716 66%	1341 76%	2387 96%	-	3071 60%
SWT	416 59%	1172 28%	1030 43%	1280 31%	836 47%	-	2394 31%	1305 25%
Zxing	394 56%	530 13%	822 34%	1099 27%	-	837 34%	1344 17%	1096 21%
Tomcat	526 75%	1873 45%	1360 56%	-	1090 62%	1273 51%	2724 35%	2126 41%
ArgoUML	445 63%	1375 33%	-	1381 34%	832 47%	569 23%	1886 24%	1490 29%
Pillar1	520 74%	-	1374 57%	1903 46%	1048 59%	1193 48%	2563 33%	1889 37%
Pillar2	-	520 13%	437 18%	525 13%	390 22%	415 17%	577 7%	517 10%

To address RQ1.2, I searched the contextual model, i.e. repository of source code terms, of each application using the source code terms of other applications, excluding the concepts to see if there are some common vocabulary outside the concepts. I identified several common terms between each application with varying frequency (see Table 3.6).

The search results show that a high percentage of common terms exist between the applications especially when searching the contextual model of a large application with the source code terms of a smaller one. For example, out of 2477 unique SWT source code terms 2387 or 96% exists in Eclipse. Investigating the reasons revealed that the SWT code base is included in the Eclipse dataset because SWT is a sub-project of Eclipse. I also noticed that 83% of Pillar2 terms are present in Eclipse. Curious to find what kind of terms a proprietary application may share with a FLOSS application of a different domain, I performed a manual analysis and found that those common terms are the ones generally used by the developers, e.g. CONFIG, version, read, update, etc.

I also cross-checked the validity of the identified common terms by repeating the search over the contextual model of the other application (i.e. reverse search) and obtained varied number of common words (see the right cross section of Table 3.6) due to following two reasons.

1. When searching the repository of one application with the terms of another one, the number of common terms may be the same but the percentage value may differ due to the size of the applications. For example, Pillar1 and Pillar2 applications have 520

Table 3.7: Common identifiers found in source files of each application across all projects

Application	**Pillar2**	**Pillar1**	**ArgoU**	**Tomcat**	**ZXing**	**SWT**	**Eclipse**	**AspectJ**
Identifiers	4394	27174	11435	25300	5550	10979	140991	27970
AspectJ	164 4%	1399 5%	1199 10%	1987 8%	592 11%	746 7%	4413 3%	-
Eclipse	326 7%	2583 10%	2137 19%	3534 14%	1072 19%	10902 99%	-	4380 16%
SWT	117 3%	645 2%	564 5%	717 3%	416 7%	-	10905 8%	736 3%
Zxing	101 2%	542 2%	419 4%	618 2%	-	419 4%	1069 1%	580 2%
Tomcat	180 4%	1584 6%	1088 10%	-	619 11%	708 6%	3534 3%	1962 7%
ArgoUML	134 3%	989 4%	-	1100 4%	426 8%	574 5%	2149 2%	1192 4%
Pillar1	187 4%	-	988 9%	1600 6%	550 10%	655 6%	2608 2%	1395 5%
Pillar2	-	187 1%	130 1%	178 1%	101 2%	114 1%	323 0%	161 1%

common terms so the distribution is 73% (520/701) when Pillar1 code base is searched with Pillar2 terms and 13% (520/4143) when Pillar2 is searched with Pillar1 terms.

2. For certain applications, reverse search resulted in common term size to vary. For example, searching Pillar2 with Tomcat found 525 common terms but 526 when searching the other way around. This is because the term ***"action"*** is a concept in Tomcat and it is excluded from the search. I found the same irregularities also in other applications, which caused common term size to vary.

Subsequently, I wanted to find out if the common vocabulary also contributes towards ease of code understanding by developers. Since identifiers are made up of compound terms, I looked at overall vocabulary of all the identifiers and not just component terms that occur in the source. After all, even if there is an overlap, one application might combine them in completely different ways to make identifiers that other developers will find confusing. Indeed, all applications combine the words differently: only a small percentage of common identifiers exists between the applications as seen in Table 3.7.

Again, when searching Eclipse with SWT identifiers, 99% (10,902/10,979) of them are found in Eclipse because all of SWT source files are included as a sub-project within Eclipse. Conversely when the SWT code base is searched with Eclipse identifiers then 10,905 are found. The reason for finding 3 additional identifiers is because those identifiers are SWT concepts and are excluded when searching the Eclipse code base as explained previously.

Furthermore, during the search to find the number of common identifiers, I noticed that when two applications did not share any concept terms on their identifier names then none of the identifiers were excluded from the search, which resulted in the same number of common identifiers between both applications. For example, Tomcat and Eclipse do not share any concepts so no identifier was excluded during the search, which resulted in 3534 common identifiers. Conversely, when both applications share concept terms then exact number of identifiers are excluded from the comparison resulting also in the same number of common identifiers, e.g. Pillar1 and Pillar2 implement the same concepts of the financial domain, thus have 187 common identifiers.

I also found that the common identifiers between other applications outside the Eclipse and the financial domain, are made up of terms commonly used by developers during programming, e.g. array, index, config, insert, delete, etc. This is also the reason why Pillar2, a proprietary software not publicly available, share terms with publicly available applications. For example, ***entity*** is a common term shared between Eclipse and Pillar2, but in Pillar2 the *entity* is a financial concept that represents a risk object and in Eclipse the *entity* is a programming concept that represents a database entry.

3.4.3 RQ1.3: How can the Vocabulary be Leveraged When Searching for Concepts to Find the Relevant Files?

The results to my previous sub-research questions revealed that (1) the project artefacts and source code files have good conceptual alignment, which is a good indicator for leveraging domain concepts to search for the source files that may be relevant for a bug report, and that (2) there is vocabulary similarity beyond domain concepts, which may contribute towards code comprehension. Henceforth, my third sub-research question seeks to answer whether searching with domain concepts only is adequate enough to find the relevant files for a given bug report.

As described in Sub-section 3.3.1.2, ConCodeSe stores referential links to aid identifying relations among project artefacts like the concept terms occurring in each bug report. To address my third research question, I did a search using the domain concepts each bug report refers to among the terms extracted from the source files using my tool. The retrieved files were compared to those that should have been returned, i.e. those that were affected by implementing the bug report as listed in the datasets of the projects (see Table 2.1).

Looking at the search results in Table 3.8, my approach achieved the best performance

Table 3.8: Bug reports for which at least one affected file placed into top-N using concept terms only search

Application	**Top-1**	**Top-5**	**Top-10**	**MAP**	**MRR**
AspectJ	2%	13%	26%	0.04	0,.10
Eclipse	2%	6%	11%	0.03	0.06
SWT	19%	38%	45%	0.22	0.32
ZXing	0%	20%	25%	0.07	0.10
Tomcat	3%	11%	16%	0,06	0.08
ArgoUML	3%	11%	12%	0.04	0.10
Pillar1	4%	4%	7%	0.01	0.11
Pillar2	17%	58%	83%	0.18	0.63
On Average	**6%**	**20%**	**28%**	**0.08**	**0.19**

in SWT and Pillar2. In the remainder of the projects, the performance is very poor. On average I find one or more affected files in the top-10 ranked files for 28% of the bug reports due to very terse distribution of the concepts in the bug reports (see Figures 3.6 and 3.7).

To improve precision (MAP), so that developers have to inspect fewer source code files for their relevance to the bug report, I ran my approach using the actual words of the bug report, rather than its associated concepts, because they better describe the concept's aspects or actions to be changed.

Table 3.9 shows that in all projects for over 30% of the bug reports at least one relevant source file was placed in Top-1. The simple fact of considering all the vocabulary of a bug report during search ranked at least one or more relevant file in the top 10 ranks, on average, for 76% of the bug reports. In addition the MAP and MRR values are substantially improved (on average from 0.08 to 0.39 for MAP and from 0.19 to 0.69 for MRR) compared to Table 3.8. The improved results confirm that bug reports contain information that may boost the performance when properly evaluated.

Table 3.9: Bug reports for which at least one affected file placed into top-N using all of the bug report vocabulary

Project	**Top-1**	**Top-5**	**Top-10**	**MAP**	**MRR**
AspectJ	35%	67%	78%	0.30	0.62
Eclipse	35%	60%	70%	0.35	0.55
SWT	59%	89%	93%	0.62	0.85
ZXing	55%	75%	80%	0.55	0.68
Tomcat	49%	69%	75%	0.51	0.65
ArgoUML	32%	62%	66%	0.30	0.55
Pillar1	30%	59%	63%	0.22	0.69
Pillar2	33%	67%	83%	0.26	0.92
On average	**41%**	**68%**	**76%**	**0.39**	**0.69**

Table 3.10: Recall performance at top-N with full bug report vocabulary search

Project	Top-5		Top-10	
	Mean	Median	Mean	Median
AspectJ	36%	29%	46%	40%
Eclipse	44%	33%	53%	50%
SWT	73%	100%	83%	100%
Zxing	62%	75%	67%	100%
Tomcat	58%	100%	66%	100%
ArgoUML	39%	25%	43%	40%
Pillar1	35%	29%	43%	40%
Pillar2	19%	18%	31%	31%
On average	**46%**	**51%**	**54%**	**63%**

3.4.3.1 Performance

Although Table 3.9 shows that by using all the vocabulary of a bug report on average for 76% of the bug reports at least one file was placed in the top-10, it does not actually tell the true recall performance, which is calculated as the number of relevant files placed in the top-N out of all the effected relevant files. To find out the recall at top-N, I calculated the min, max, mean and median values of the results for each project. The recall is capped at 10 (N=10), which means that if a bug report has more than 10 relevant files only the 10 are considered. The min and max values were always 0 and 1 respectively because for each bug report my approach is either able to position all files in top-10 or none, therefore I only report the mean and median values in Table 3.10.

For Eclipse, SWT, ZXing and Tomcat my approach placed at least one of the relevant files in the top-10 when all of the bug report vocabulary is used during the search. Also on average for almost one quarter (46%) of the bug reports at least one relevant file is placed in the top-5. These two findings prove that my approach successfully positions on average 54% of the relevant files for a bug report in the top-10, thus provides a reliable focal point for developers to locate affected files without going through all of the files in the results list.

3.4.3.2 Contribution of VSM

As described in Section 3.3.2, out of 8 rankings of a file, 4 are obtained with the VSM probabilistic method available in the Lucene library, and the other 4 with my lexical similarity ranking method (Algorithm 3.1), which then take the best of the 8 ranks. To find out the added value of VSM, I performed two runs, one with VSM only and the other with lexical similarity scoring only using the domain concepts.

Table 3.11: Bug reports for at least one affected file placed into top-N using VSM vs lexical similarity scoring using concept terms only search

Project	Approach	Top-1	Top-5	Top-10	MAP	MRR
AspectJ	lexical	2%	12%	24%	0.04	0.10
	vsm	0%	1%	3%	0.02	0.03
Eclipse	lexical	1%	4%	7%	0.02	0.04
	vsm	1%	3%	5%	0.02	0.03
SWT	lexical	17%	36%	42%	0.20	0.30
	vsm	5%	11%	18%	0.09	0.11
ZXing	lexical	0%	10%	15%	0.05	0.06
	vsm	0%	10%	10%	0.04	0.06
Tomcat	lexical	2%	9%	12%	0.05	0.06
	vsm	1%	4%	6%	0.02	0.03
ArgoUML	lexical	2%	4%	7%	0.03	0.06
	vsm	1%	7%	7%	0.03	0.06
Pillar1	lexical	4%	4%	4%	0.01	0.08
	vsm	0%	4%	7%	0.01	0.04
Pillar2	lexical	8%	33%	50%	0.12	0.41
	vsm	8%	42%	58%	0.12	0.39
On average	**lexical**	**5%**	**14%**	**20%**	**0.06**	**0.14**
	vsm	**2%**	**10%**	**14%**	**0.04**	**0.09**

Table 3.11 shows that lexical scoring outperforms VSM in all projects except for Pillar1 in top-10 and Pillar2 in top-5. As to be expected, each scoring by itself has a poorer performance than the combined score, whilst the total number of bug reports located by ConCodeSe (Table 3.8) is not the sum of the parts. In other words, many files are ranked in the top-N by both scoring methods, and each is able to locate bug reports the other can't. ConCodeSe literally takes the best of both worlds, which shows the crucial contribution of lexical similarity scoring for the improved performance of my approach.

VSM is known to achieve a bigger contribution for projects with a large number of files, which makes the use of term and document frequency more meaningful. However, in my case using a handful of concepts that refer to a bug report (see Figures 3.6 and 3.7) as search terms, reveals the shortcomings of VSM over lexical search.

I repeated the search to compare the lexical match against VSM but this time using all of the vocabulary extracted from the bug report. As Table 3.12 shows the lexical search performed superior to VSM only in smaller size projects, i.e. SWT, ZXing, Tomcat and Pillar2. This confirms that VSM achieves a bigger contribution for projects with a large number of file only when the search terms are also more than just a couple of concept terms. I also confirm that the VSM advantage is paramount since it only has on average 2.4% improvement across all performance metrics, i.e. top-N, MAP and MRR, over the lexical

Table 3.12: Bug reports for at least one affected file placed into top-N using VSM vs lexical similarity scoring using full bug report vocabulary search

Project	Approach	Top-1	Top-5	Top-10	MAP	MRR
AspectJ	lexical	14%	44%	59%	0.17	0.34
	vsm	28%	47%	60%	0.22	0.49
Eclipse	lexical	18%	35%	48%	0.20	0.32
	vsm	23%	46%	57%	0.25	0.41
SWT	lexical	43%	77%	89%	0.50	0.69
	vsm	40%	71%	84%	0.46	0.66
ZXing	lexical	40%	65%	80%	0.45	0.53
	vsm	40%	60%	70%	0.41	0.50
Tomcat	lexical	34%	56%	64%	0.38	0.48
	vsm	28%	48%	58%	0.32	0.43
ArgoUML	lexical	12%	48%	56%	0.20	0.33
	vsm	24%	52%	58%	0.24	0.45
Pillar1	lexical	7%	33%	41%	0.10	0.36
	vsm	26%	48%	56%	0.18	0.59
Pillar2	lexical	25%	67%	75%	0.20	0.71
	vsm	8%	67%	75%	0.18	0.62
On average	**lexical**	**24%**	**53%**	**64%**	**0.27**	**0.47**
	vsm	**27%**	**55%**	**65%**	**0.28**	**0.52**

search alone, however, lexical search on its own still has a weak performance.

3.4.3.3 Searching for AspectJ and SWT in Eclipse

In Section 3.4.2, I found that Eclipse contains the SWT source files. Since AspectJ is also a sub-eclipse project and 60% of AspectJ terms are present in Eclipse (see Table 3.6), I searched the source files of Eclipse using the actual words extracted from the bug reports of SWT and AspectJ. The results for SWT are very close to those obtained by the search over the application's own source files (last row in Table 3.13). This indicates that given a large search space with many source files, e.g. 12,863 in Eclipse, my approach successfully retrieves the relevant files for the bug reports of a smaller sub-set of the search space, e.g. SWT with 484 files, without retrieving many false positives.

Table 3.13: Search performance for SWT and AspectJ bug reports within Eclipse code base

Project	Search	Top-1	Top-5	Top-10	MAP	MRR
AspectJ	Eclipse	2%	3%	4%	0.01	0.02
	AspectJ	35%	67%	78%	0.30	0.62
SWT	Eclipse	51%	81%	85%	0.55	0.71
	SWT	59%	89%	93%	0.62	0.85

In case of AspectJ, the search performance was poor compared to the search over the application's own source files (first row in Table 3.13). Analysing this revealed that some of the relevant files listed in the bug reports of Aspectj are source code class files that test the actual code, e.g. *MainTest.java* that tests *Main.java*, and the test files are excluded from the AspectJ version that is embedded in the Eclipse.

3.5 Threats to validity

The internal validity concerns claims of causality to verify that the observed outcomes are the natural product of the implementation. I only used single-word concepts, while business concepts are usually compound terms. This threat to internal validity is partially addressed by using tall the words found in the bug reports during the search task. My approach also relies on developers practicing good programming standards, particularly when naming source code identifiers, otherwise the performance of my approach may deteriorate. I minimised this threat by identifying the presence of common vocabulary beyond the domain concepts between different applications that developed independently.

The construct validity addresses biases that might have been introduced during the operationalisation of the theories. As software development consultant, I listed the concepts. This threat to construct validity was partly addressed by having the concepts validated by other stakeholders. During creation of the searchable corpus, I developed my own modules to interact with different tools: (1) parser to process source code comments and Groovy files; (2) JIM tool to extract the terms from the identifiers; (3) Lucene's Porter Stemmer to find the root of a word; and (4) Lucene's VSM module to index the repository of terms by source file. It is possible that other tools to (1) parse, (2) extract, (3) stem and (4) index the artefacts may produce different results. I catered for this threat by using the same dataset originally provided by Zhou *et al.* (2012a) and used by others (see Table 2.1), to which I added two of my own. I also used the same evaluations metrics that match those used by others in their prior work for fairness.

The external validity concerns generalisation, i.e. the possibility of applying the study and results to other circumstances. The characteristics of the projects (the domain, the terse bug reports (see Table C.1 in Appendix C) with sometimes misspelled text[21], the naming conventions, the kind of documentation available) are a threat to external validity, I addressed this by repeating the experiment with other projects and artefacts from different domains for comparison. Although ConCodeSe only works on Java projects for the purpose

[21] Pillar2 BR #2010 has incorrect spelling in the original document (see Appendix B)

of literature comparison, which may limit its generalisability, the principles are applicable to other object-oriented programming languages.

3.6 Discussion

From all the results shown, I can answer RQ1 affirmatively: project artefacts share domain concepts and vocabulary that may aid code comprehension when searching to find the relevant files during software maintenance. I offer further explanations as follows.

Regarding my first aim, I note that, together, the three artefacts (user guide, bug reports and source code) explicitly include large number of the domain concepts agreed upon by seven developers and a business analyst (see Figures 3.6 and 3.7). This indicates (a) full domain concept coverage, and (b) in these projects abbreviations are not required to retrieve such concepts from the artefacts. Those are two good indicators for maintenance. However, only a handful of concepts occur both in the code and the documentation. Since the latter is consulted during maintenance, this lack of full agreement between both artefacts, regarding the concepts in the developers' heads, may point to potential inefficiencies during maintenance.

On the other hand, and using stemming to account for lexical variations, except for AspectJ, those common concepts correlate well (with a high statistical significance of $p<0.05$ as per Table 3.4) in terms of relative frequency, taken as proxy for importance, i.e. the more important concepts in the user guide tend to be the more important ones in the code. Conceptual alignment between documentation and implementation eases maintenance, especially for new developers. The weak conceptual overlap and correlation between bug reports and the other two artefacts is not a major issue. Bug reports are usually specific for a particular unit of work and may not necessarily reflect all implemented or documented concepts.

Regarding my second aim, I found that mapping a bug report's wording to domain concepts and using those to search for files to be changed, achieved to position a relevant file into top-10 on average for only 28% of the bug reports and resulted in very poor MAP (0.08) and MRR (0.19) values as seen in Table 3.8. I found both MRR and MAP can be improved by using the actual bug report vocabulary (Table 3.8) and is enough to achieve very good recall of 54% on average (Table 3.10). I note that such project specific, simple, and efficient techniques can drastically reduce the false positives a developer has to go trough to find the files affected by a bug report.

My study showed that despite good concept coverage among the project artefacts and vocabulary similarity between the source code files, it is challenging to find the files referred by a bug report. Although reflecting the daily tasks of the developers, bug reports of action oriented nature are not a better source for traceability than user guides and domain concepts. I found that these two types of documentations, i.e. bug reports and user guides, describe the usage scenarios and may account for the improved traceability.

In this study, I also investigated the vocabulary of two independent financial applications, Pillar1 and Pillar2, covering the same domain concepts. I found that artefacts explicitly reflect the domain concepts and that paired artefacts have a good conceptual alignment, which helps maintenance when searching for files affected by a given bug report.

3.7 Concluding Remarks

This Chapter presented an efficient approach to relate the vocabulary of information sources for maintenance: bug reports, code, documentation, and the concepts in the stakeholders' minds. The approach consists of first extracting and normalising (incl. splitting identifiers and removing stop words) the terms from the artefacts, while independently eliciting domain concepts from the stakeholders. Secondly, to account for lexical variations, I did exact and stemmed searches of the concepts within the terms extracted from artefacts. I found that one can check whether (a) the artefacts explicitly reflect the stakeholders' concepts and (b) pairs of artefacts have good conceptual alignment. Both characteristics help maintenance, e.g. (b) facilitates locating code affected by given bug reports.

Although the investigation described in this Chapter is only a preliminary exploration of the vocabulary relationships between artefacts and the developers' concepts, it highlights that better programming guidelines and tool support are needed beyond enforcing naming conventions within code, because that by itself doesn't guarantee a good traceability between the concepts and the artefacts, which would greatly help maintenance tasks and communication within the team.

The importance of descriptive and consistent identifiers for program comprehension, and hence software maintenance, has been extensively argued for in the academic literature, e.g. Deißenböck and Pizka (2006), and the professional literature, e.g. Martin (2008). I applied my approach to OSS projects and to an industrial code that follows good naming conventions, in order to investigate whether they could be leveraged during maintenance. I observed that the conceptual alignment between documentation and code could be improved, and that descriptive identifiers support high recall of files affected by bug reports, but mean average precision (MAP) is low (on average 0.08), which is detrimental for maintenance. I found simple techniques to improve the MAP (on average 0.39), however further research is needed, for example, heuristics based on words in key positions within the context of a bug report could be developed.

In the next Chapter, I investigate the additional sources of information available in project artefacts that may be leveraged to improve the traceability between the artefacts during bug localisation. I extend my approach to consider words in certain positions of the bug report summary and of the stack trace (if available in a bug report) as well as source code comments, stemming, and a combination of both independently, to derive the best rank for each file. I compare the results of my approach to eight existing ones, using the same bug reports, applications and evaluation criteria to measure the improvements.

Chapter 4

Locating Bugs without Looking Back

In previous Chapter, I explored the conceptual alignment of vocabulary in project artefacts and introduced a novel approach to search for source code files that may be relevant for fixing a bug. I found good conceptual alignment among the artefacts. However, using domain concepts when searching for the source files revealed low precision and recall. I was able to improve the recall by simply using all of the vocabulary found in the bug report but precision remains low. I concluded that to improve the traceability between the artefacts during bug localisation, heuristics based on words in certain key positions within the context of bug reports may be developed.

Most state-of-the-art IR approaches rely on additional sources of information like project history, in particular previously fixed bugs and previous versions of the source code to improve the traceability. However, it is reported that neither of these sources guarantee to provide an advantage. In fact it is observed that considering past version history beyond 20 days or evaluating more than 30 previously fixed similar bug reports cause the performance of the search results to deteriorate (Wang and Lo, 2014; Nichols, 2010).

I take a more pragmatic approach to address RQ2 and hence investigate how file names found in bug reports can be leveraged during bug localisation to support software maintenance. I extend my approach presented in Chapter 3 to consider the file names in key positions when directly scoring each current file against the given report, thus without requiring past code or bug reports. The scoring method is based on heuristics identified through manual inspection of a small sample of bug reports.

I compare my approach to eight others, using their own five metrics on their own six open source and two new projects of my own. Out of 30 performance indicators, I improve 27 and equal 2. Over the projects analysed, on average my approach positions an affected file in the top-1, top-5 and top-10 ranks for 44%, 69% and 76% of the bug reports respectively. This is an improvement of 23%, 16% and 11% respectively over the best performing current state-of-the-art tool. These results show the applicability of my approach also to software projects without history.

In this Chapter, Section 4.1 gives a brief introduction and defines how RQ2 is going to be addressed. Section 4.2 revisits the IR-based approaches against which I compare mines. In Section 4.3, I explain how I extended my approach and present the results of addressing the research questions in Section 4.4. I discuss why the approach works and the threats to validity in Section 4.5, and conclude with remarks in Section 4.6.

4.1 Introduction

In Chapter 1, I defined the common view that the current state-of-the-art IR approaches see the bug report as the query, and the source files as the documents to be retrieved, ranked by relevance. Such approaches have the advantage of not requiring expensive static or dynamic analysis of the code. However, these approaches rely on project history, in particular previously fixed bugs and previous versions of the source code.

The study in (Li *et al.*, 2006), performed with two OSS projects, showed that 81% of all bugs in Mozilla and 87% of those in Apache are semantics related. These percentages increase as the applications mature, and they have direct impact on system availability, contributing to 43–44% of crashes. Since it takes a longer time to locate and fix semantic bugs, more effort needs to be put into helping developers locate the bugs.

Recent empirical studies provide evidence that many terms used in bug reports are also present in the source code files (Saha *et al.*, 2013; Moreno *et al.*, 2013). Such bug report terms are an exact or partial match of program elements (i.e. class, method or variable names and comments) in at least one of the files affected by the bug report, i.e. those files actually changed to address the bug report.

Moreno *et al.* (2013) showed that (1) the bug report documents share more terms with the corresponding affected files and (2) the shared terms were present in the source code file names. The authors evaluated six OSS projects, containing over 35K source files and 114 bug reports, which were solved by changing 116 files. For each bug report and source file combination (over 1 million), they discovered that on average 75% share between 1–13 terms, 22% share nothing and only 3% share more than 13 terms. Additionally, the study revealed that certain locations of a source code file, e.g. a file name instead of a method signature, may have only a few terms but all of them may contribute to the number of shared terms between a bug report and its affected files. The authors concluded that the bug reports have more terms in common with the affected files and the common terms are present in the names of those affected files.

Saha *et al.* (2013) claimed that although file names of classes are typically a combination of 2–4 terms, they are present in more than 35% of the bug report summary fields and 85% of the bug report description fields of the OSS project AspectJ. Furthermore, the exact file name is present in more than 50% of the bug descriptions. They concluded that when the terms from these locations are compared during a search, the noise is reduced automatically due to reduction in search space. For example, in 9% (27/286) of AspectJ bug reports, at least one of the file names of the fixed files was present as-is in the bug report summary, whereas in 35% (101/286) of the bug reports at least one of the file name terms was present.

Current state-of-the-art approaches for Java programs, e.g. BugLocator (Zhou *et al.*, 2012a), BRTracer (Wong *et al.*, 2014), BLUiR (Saha *et al.*, 2013), AmaLgam (Wang and Lo, 2014), LearnToRank (Ye *et al.*, 2014) BLIA (Youm *et al.*, 2015) and Rahman *et al.* (2015) rely on project history to improve the suggestion of relevant source files. In particular they use similar bug reports and recently modified files. The rationale for the former is that if a new bug report $\boldsymbol{x}$ is similar to a previously closed bug report $\boldsymbol{y}$, the files affected by $\boldsymbol{y}$ may also be relevant to $\boldsymbol{x}$. The rationale for the latter is that recent changes to a file may have led to the reported bug (Wang and Lo, 2014). However, the observed improvements using the history information have been small. Thus my hypothesis is that file names mentioned in the bug report descriptions can replace the contribution of historical information in achieving comparable performance and asked, in Chapter 1, my second research question as follows.

RQ2: Can the occurrence of file names in bug reports be leveraged to replace project history and similar bug reports to achieve improved IR-based bug localisation?

To address the second RQ in this Chapter, I aim to check if the occurrence of file names in bug reports can be leveraged for IR-based bug localisation in Java programs. I restrict the scope to Java programs, where each file is generally a class or interface, in order to directly match the class and interface names mentioned in the bug reports to the files retrieved by IR-based bug localisation. If file name occurrence can't be leveraged, I look more closely at the contribution of past history, in particular of considering similar bug reports, an approach introduced by BugLocator and adopted by others. Henceforth I ask in my first sub-research question:

RQ2.1: *What is the contribution of using similar bug reports to the results performance in other tools compared to my approach, which does not draw on past history?*

IR-based approaches to locating bugs use a base IR technique that is applied in a context-specific way or combined with bespoke heuristics. However, (Saha *et al.*, 2013) note that the exact variant of the underlying tf/idf[1] model used may affect results. In particular they find that the off-the-shelf model used in their tool BLUiR already outperforms BugLocator, which introduced rVSM, a bespoke VSM variant. My approach also uses an off-the-shelf VSM tool, different from the one used by BLUiR. In comparing my results to theirs I aim to distinguish the contribution of file names, and the contribution of the IR model used. Thus I ask in my second sub-research question:

RQ2.2: *What is the overall contribution of the VSM variant adopted in my approach, and how does it perform compared to rVSM?*

To address RQ2, I extend my approach introduced in Chapter 3 and then evaluate it against existing approaches (Zhou *et al.*, 2012a; Saha *et al.*, 2013; Moreno *et al.*, 2014; Wong *et al.*, 2014; Wang and Lo, 2014; Ye *et al.*, 2014; Youm *et al.*, 2015; Rahman *et al.*, 2015) on the same datasets and with the same performance metrics. Like other approaches, mine scores each file against a given bug report and then ranks the files in descending order of score, aiming for at least one of the files affected by the bug report to be among the top-ranked ones, so that it can serve as an entry point to navigate the code and find the other affected files. As we shall see, my approach outperforms the existing ones in the majority of cases. In particular it succeeds in placing an affected file among the top-1, top-5 and top-10 files for 44%, 69% and 76% of bug reports, on average.

My scoring scheme does not consider any historical information in the repository, which contributes to an *ab-initio* applicability of my approach, i.e. from the very first bug report submitted for the very first version. Moreover, my approach is efficient, because of

[1] tf/idf (term frequency/inverse document frequency) is explained in Chapter 2, Sub-section 2.2.1

the straightforward scoring, which doesn't require any further processing like dynamic code analysis to trace executed classes by re-running the scenarios described in the bug reports.

To address RQ2.1, I compare the results of BugLocator and BRTracer using SimiScore (the similar bug score), and the results of BLUiR according to the literature, showing that SimiScore's contribution is not as high as suggested. From my experiments, I conclude that my approach localises many bugs without using similar bug fix information, which were only localised by BugLocator, BRTracer or BLUiR using similar bug information.

As for RQ2.2, through experiments I found that VSM is a crucial component to achieve the best performance for projects with a larger number of files where the use of term and document frequency is more meaningful, but that in smaller projects its contribution is rather small. I chose the Lucene VSM, which performs in general better than the bespoke rVSM.

4.2 Existing Approaches

In this Section, I summarise the key points from the academic literature discussed in Chapter 2 that relate to my proposed approach. Nichols (2010) argued that utilising additional sources of information would greatly aid IR models to relate bug reports and source code files. One of the information sources available in well-maintained projects is the past bug details and to take advantage of this, the author proposes an approach that mines the past bug information automatically. The author concludes that search results from the repository augmented with up to 14 previous bug reports were the same as those from the non augmented one (Subsection 2.3.4).

Zhou *et al.* (2012a) proposed an approach consisting of the four traditional IR steps (corpus creation, indexing, query construction, retrieval & ranking) but using a revised Vector Space Model (rVSM) to score each source code file against the given bug report. In addition, each file gets a similarity score (SimiScore) based on whether the file was affected by one or more closed bug reports similar to the given bug report (Section 2.4). The approach, implemented in a tool called BugLocator, was evaluated using over 3,400 reports of closed bugs and their known affected files from four OSS projects (see Table 2.1). The authors report superior performance of the results compared to traditional VSM only approach.

Bettenburg *et al.* (2008) disagree with the treatment of a bug report as a single piece of text document and source code files as one whole unit by existing approaches, e.g. Poshyvanyk *et al.* (2007); Ye *et al.* (2014); Kevic and Fritz (2014); Abebe *et al.* (2009). Bettenburg *et al.*

(2008) argue that a bug report may contain a readily identifiable number of elements including stack traces, code fragments, patches and recreation steps each of which should be treated separately for analytical purposes (Sub-section 2.3.3).

Saha *et al.* (2013) presented BLUiR, which leverages the structures inside a bug report and a source code file by computing a similarity score between each of the 2 fields (**summary** and **description**) of a bug report and each of the 4 parts of a source file (class, method, variable names, and comments) as described in Sub-section 2.3.5. The results were evaluated using BugLocator's dataset and performance indicators. For all but one indicator for one project, ZXing's mean average precision (**MAP**), BLUiR matches or outperforms BugLocator, hinting that a different IR approach might compensate for the lack of history information, namely the previously closed similar bug reports.

Moreno *et al.* (2014) presented LOBSTER, which leverages the **stack trace** information available in bug reports to suggest relevant source code files. The approach first calculates a textual similarity score between the words extracted by Lucene, which uses VSM, from bug reports and files. Second, a structural similarity score is calculated between each file and the file names extracted from the stack trace. The authors conclude that considering stack traces does improve the performance with respect to only using VSM (Sub-section 2.3.3). I also leverage stack trace and compare the performance of both tools using their dataset for ArgoUML, a UML tool (Table 2.1).

Wong *et al.* (2014) proposed BRTracer, which also leverages the stack trace information and performs segmentation of source files to reduce any noise due to varying size of the file lengths (Bettenburg *et al.*, 2008). The stack trace score is calculated by evaluating the files listed in the stack trace and the files referenced in the source code file. The authors claim that the segmentation or the stack trace analysis is an effective technique to boost bug localisation (Sub-section 2.3.3). Since I also evaluate my approach with BugLocator 's dataset and leverage stack traces, I compare the performance of my approach against BRTracer and report on the improvements gained.

Wang and Lo (2014) proposed AmaLgam for suggesting relevant buggy source files by combining BugLocator's SimiScore and BLUiR's structured retrieval into a single score using a weight factor, which is then combined (using a different weight) with a version history score that considers the number of bug fixing commits that touch a file in the past k days. The evaluation shows that considering historical commits up to 15 days increased the performance, 15 to 20 days did not make much difference and considering up to 50 days deteriorated the performance. Thus they conclude that the most important part of the version history is between 15–20 days (Sub-section 2.3.5).

Ye *et al.* (2014) defined a ranking model that combines six features measuring the relationship between bug reports and source files, using a learning-to-rank (LtR) technique (Sub-section 2.3.5). Their experimental evaluations show that the approach places the relevant files within the top-10 recommendations for over 70% of the bug reports of Tomcat (Table 2.1). My approach is much more lightweight: it only uses the first of Ye *et al.*'s six features, lexical similarity, and yet provides better results on Tomcat.

Recently, Youm *et al.* (2015) introduced an approach where the scoring methods utilised in previous studies (Zhou *et al.*, 2012a; Saha *et al.*, 2013; Wong *et al.*, 2014; Wang and Lo, 2014) are first calculated individually and then combined together by varying alpha and beta parameter values. The approach, implemented in a tool called BLIA, is compared against the performance of the other tools where the original methods were first introduced. The authors found that stack-trace analysis is the highest contributing factor among the analysed information for bug localisation (Section 2.4).

In another recent study, Rahman *et al.* (2015) extended the approach introduced in Zhou *et al.* (2012a) by considering file fix frequency and file name match (Section 2.4). Independently of Rahman *et al.* (2015) (of which I became aware only recently), I decided to use file names because of the study Saha *et al.* (2013) that shows many bug reports contain the file names that need to be fixed. The two main differences are (1) in Rahman *et al.* (2015) the file names are extracted from the bug report using very simple pattern matching technique and (2) only one constant value is used to boost a file's score when its name match one of the extracted file names. On the contrary, my approach (1) uses a more sophisticated file matching regular expression pattern to extract file names from the bug report and (2) treat the extracted file name's position in the bug report with varying importance when scoring a file, as we will see later.

4.3 Extended Approach

Each approach presented in the previous section incorporates an additional information source to improve results, as shown in Table 4.1. I list the tools against which I evaluate my approach (ConCodeSe), by using the same datasets (Table 2.1) and metrics. So far my

Table 4.1: Comparison of IR model and information sources used in other approaches

Approach	**Underlying IR logic/model**	**Version History**	**Similar Report**	**Structure**	**File Name**	**Stack Trace**
BugLocator	rVSM	no	yes	no	no	no
BRTracer	rVSM + segmentation	no	yes	yes	no	yes
BLUiR	Indri	no	yes	yes	no	no
AmaLgam	Mixed	yes	yes	yes	no	no
BLIA	rVSM+ segmentation	yes	yes	yes	no	yes
Rahman	rVSM	yes	yes	no	yes	no
LtR	VSM	yes	yes	yes	no	no
LOBSTER	VSM	no	no	yes	no	yes
ConCodeSe	lexicalScore + VSM	no	no	yes	yes	yes

approach introduced in Chapter 3 ranks the source code files of an application based on lexical similarity between the vocabulary of the bug reports and the terms found in the source files. In this Section, I describe how I extend my approach to also consider additional information like file names and stack trace details available in bug reports during scoring.

4.3.1 Ranking Files Revisited

As explained in Chapter 3, Sub-section 3.3.2, given a bug report and a file, my approach computes two kinds of scores for the file: a lexical similarity score and a probabilistic score given by VSM, as implemented by Lucene.

The two scorings are done with four search types, each using a different set of terms indexed from the bug report and the file:

1. Full terms from the bug report and from the file's code.

2. Full terms from the bug report and the file's code and comments.

3. Stemmed terms from the bug report and the file's code.

4. Stemmed terms from the bug report, the file's code and comments.

Recall that for each of the 8 scorings, all files are ranked in descending order and then, for each file the best of its 8 ranks is taken (Sub-section 3.3.2). Henceforth I introduce a new function *searchAndRankFiles* (Algorithm 4.1), which encapsulates the function *scoreWithFileTerms* (Algorithm 3.1), and still performs the lexical similarity scoring and ranking by taking as

Algorithm 4.1 *searchAndRankFiles*: Main logic for scoring and ranking of project's files per bug report

```
input: files: List<File>, br: BR // one bug report
output: ranked: List<File>
for each file in files do do
    file.score := scoreWithKeyPositionWord(file.name, br.summary) // KP Score

    if file.score = 0 and br.stackTrace exists then then
        file.score := scoreWithStackTrace(file.name, br.stackTrace) // ST Score
    end if
    if file.score = 0 then then
        file.score := scoreWithFileTerms(file, br.terms) // TT Score
    end if
end for
return files in descending order by score
```

arguments a bug report and the project's files and returns an ordered list of files. The function is called 4 times, for each search type listed previously and goes through the following steps for each source code file.

1. Check if its name matches one of the words in key positions (KP) of the bug report's summary, and assign a score accordingly (Section 4.3.1.1).
2. If no score was assigned and if stack trace is available, check if the file name matches one of the file names listed in the stack trace and assign a score (ST) accordingly (Section 4.3.1.2).
3. If there is still no score then assign a score based on the occurrence of the search terms, i.e. the bug report text terms (TT) in the file (Algorithm 3.1 in Chapter 3).

Once all the files are scored against a bug report, the list is sorted in descending order so that the files with higher scores are ranked at the top. Ties are broken by alphabetical order.

4.3.1.1 Scoring with Key Positions (KP score)

By manually analysing all SWT and AspectJ bug reports and 50 random Eclipse bug reports, i.e. (98+286+50)/4665=9.3% of all bug reports (Table 2.1), I found that the word in first, second, penultimate or last position of the bug report summary may correspond to the affected file name. For example, Table 4.2 shows that in the summary sentence of bug report #79268, the first word is already the name of the affected source file, i.e. *Program.java*.

Overall, my manual analysis of SWT revealed that in 42% (42/98) of the bug reports the first word and in 15% (15/98) of the bug reports the last word of the summary sentence was

Table 4.2: Sample of words in key positions showing presence of source file names

BR#	Summary	Position
79268	Program API does not work with GNOME 2.8 (libgnomevfs-WARNING)	First
78559	[consistency] Slider fires two selection events before mouse down	Second
92341	DBR - Add SWT.VIRTUAL style to Tree widget	Pen-ultimate
100040	Slow down between 3.1 RC1 and N20050602 due to change to ImageList	Last

the affected source file (see Table 4.3). I found similar patterns in AspectJ: 28% (81/286) as the first word and 5% (15/286) as the last word. The frequency for the second and penultimate words being the affected file was 4% and 11% respectively.

Table 4.3: Summary of file names at key positions in the analysed bug reports

Project	# of BRs Analysed	First Position	Second Position	Pen-ultimate	Last Position
AspectJ	286	81 (28%)	12 (4%)	39 (11%)	15 (5%)
Eclipse	50	7 (14%)	4 (8%)	3 (6%)	4 (8%)
SWT	98	42 (42%)	7 (7%)	4 (4%)	15 (15%)

I also observed that some key position words come with method and package names in the bug report summary, e.g. Class.method() or package.subpackage.Class.method(). They hence required parsing using regular expressions. Based on these patterns I assign a high score to a source file when its name matches the words in the above described four key positions of the bug report summary sentence. The earlier the file name occurs, the higher the score: the word in first position gets a score of 10, the second 8, the penultimate 6 and the last 4 (see Algorithm 4.2).

Algorithm 4.2 *scoreWithKeyPositionWord*: Scores a file based on terms matching four key positions of the bug report summary sentence

```
input: fileName: String, summary: String // BR summary sentence
output: score: int
n := numWords(summary) //number of words
p := summary.find(fileName)
if (p = 1) return 10
if (p = 2) return 8
if (p = n -1) return 6
if (p = n) return 4
return 0
```

Note that the key positions are scored in a different order (1st, 2nd, penultimate, last) from their frequency (1st, penultimate, last, 2nd), because while experimenting with different score

Table 4.4: Stack trace information available in bug reports (BR) of each project

Project	**# of BRs**	**# of BRs with Stack Traces**	**% of BRs with Stack Traces**
AspectJ	286	67	23%
Eclipse	3075	435	14%
SWT	98	4	4%
ZXing	20	1	5%
Tomcat	1056	83	8%
ArgoUML	91	5	5%
Pillar1	27	1	4%
Pillar2	12	0	0%

values for SWT and AspectJ I found the 'natural' order to be more effective. Disregarding other positions in the summary prevents non-affected files that occur in those other positions from getting a high KP score and thus a higher ranking.

4.3.1.2 Scoring with Stack Traces (ST score)

Stack traces list the files that were executed when an error condition occurs. During manual analysis of the same bug reports as for KP score, I found several included a stack trace in the description field (see Table 4.4).

I noticed that especially for NullPointerException, the affected file was often the first one listed in the stack trace. For other exceptions such as UnsupportedOperationException or IllegalStateException, however the affected file was likely the 2nd or the 4th in the trace.

I first use regular expressions (see Figure D.1 in Appendix D) to extract from the stack trace the application-only source files, i.e. excluding third party and Java library files, and then put them into a list in the order in which they appeared in the trace. I score a file if its name matches one of the first four files occurring in the list. The file in first position gets a score of 9, the second 7, the third 5 and the fourth 3 (see Algorithm 4.3).

Algorithm 4.3 *scoreWithStackTrace*: Score a file based on terms matching one of the four files occurring in the stack trace

```
input: fileName: String, stackTrace: List<String> // BR stack trace
output: score: int
p := stackTrace.find(fileName)
if (p = 1) return 9
if (p = 2) return 7
if (p = 3) return 5
if (p = 4) return 3
return 0
```

4.3.1.3 Rationale behind the scoring values

As mentioned in the previous subsections, values for the KP scoring (10, 8, 6, 4), ST scoring (9, 7, 5, 3) and TT scoring (2, 0.025, 0.0125) were obtained heuristically, whilst reflecting the importance of the summary, the stack trace and the description, in this order, and of the different parts of each. The scoring values are weights that give more importance to certain positions in the summary and stack trace, or to certain kinds of words (class names). As such, they are in essence not different from other weights used by other authors, which they also obtained heuristically. For example, Hill *et al.* (2007) assigned weights 2.5 and 0.5 to terms based on whether they match those extracted from the file name or method name, respectively, and Uneno *et al.* (2016) summed the scores of three tools adjusted by weights 0.5, 0.3, and 0.1 or 1.0, based on manual experiments and observations. Methodologically, my approach therefore does not deviate from established practice.

Some authors automatically determine the best weight values, e.g. by using machine learning. However, the optimal weights obtained from a particular collection of projects are not necessarily the best weights for other projects. For example, in Tomcat the bug reports have very detailed and long descriptions but in Pillar2 they are tersely written. Optimising weights (via machine learning or some other technique) must therefore be done separately for each project, and only for projects that have sufficient history, i.e. sufficient past bug reports that can be used as training set.

Since I am interested in a specific approach that does not rely on history to provide the best results for the 8 projects at hand, I aim to see whether using the least information possible (bug reports and source code files) and using a light-weight IR approach, I can achieve similar performance to other approaches that use more data sources (similar past bug reports, past code versions). As I will show in Sub-section 4.4.1.5, my approach surpass the performance of other approaches.

4.4 Evaluation of the Results

In this Section, I address my research questions. Since they ask for the effects of various scoring components, I had to run ConCodeSe, BugLocator and BRTracer (the other tools were unavailable) in different 'modes', e.g. with and without stack trace information or with and without SimiScore (similar bug reports), to observe the effect on the ranking of individual files. I ran BugLocator and BRTracer without SimiScore by setting the alpha parameter to zero, as described in (Zhou *et al.*, 2012a; Wong *et al.*, 2014). I also ran both tools with SimiScore, by setting alpha to the value reported in the corresponding paper. I confirmed that I obtained the same top-N, MAP and MRR results as reported in the papers. This reassured that I was using the same tool versions, datasets and alpha values as the authors had, and that the results reported in the rest of this section for BugLocator and BRTracer are correct.

As I derived my heuristics by manually analysing the bug reports of AspectJ and SWT, and some from Eclipse, to avoid bias, I evaluated my approach using additional OSS and industrial projects: ArgoUML, Tomcat, ZXing, Pillar1 and Pillar2. As already described in Section 4.2, all but the last two projects were also used by the approaches I compare my tool's performance against.

4.4.1 RQ2: Scoring with File Names in Bug Reports

As described throughout Section 4.3, my lexical similarity scoring mainly considers whether the name of the file being scored occurs in the bug report, giving more importance to certain positions in the summary or in the description's stack trace. The rationale is of course that a file explicitly mentioned in the bug report is likely to require changes to fix the bug.

RQ2 asks whether such an approach, although seemingly sensible, is effective. To answer the question I compare my results to those of BugLocator, BRTracer, BLUiR, AmaLgam, LtR, BLIA and Rahman using the same 5 metrics (Top-1, Top-5, Top-10) and for LOBSTER using only the MAP and MRR metrics (as given in their paper). I look at the separate and combined effect of using file names for scoring.

4.4.1.1 Scoring with Words In Key Positions (KP score)

To see the impact of considering occurrences of the file name in certain positions of the bug report summary, I ran ConCodeSe with and without assigning a KP score, whilst keeping all the rest as described in Section 4.3.1. Table 4.5 shows how evaluating the words in key positions improved results for the affected classes of the bug reports given in Table 4.2. In the cases of BugLocator and BRTracer, I obtained the ranks by running their tool, and obtained the ones for BLUiR from their published results.

As Table 4.5 illustrates, in some cases (like for *Program.java*) the summary scoring can

Table 4.5: Example of ranking achieved by leveraging key positions in SWT bug reports compared to other tools

SWT BR#	**Affected file**	**Bug Locator**	**BR Tracer**	**BLUiR**	**ConCodeSe**	
					without	**with**
79268	Program.java	21	11	-	19	1
78559	Slider.java	2	4	1	5	1
92341	Tree.java	1	1	5	4	2
100040	ImageList.java	7	1	9	2	1

make the difference between the file not even making into the top-10 or making into the top-5. In other cases, the change in ranking is small but can be significant, making the affected file become the top ranked, which is always the most desirable situation, as the developer will not have to inspect any irrelevant files.

To have an overall view, I also ran ConCodeSe using just KP and TT scores together (KP+TT) against only using TT score, i.e. in both runs stack trace (ST score) and VSM scoring were not used. Table 4.6 shows that compared to TT score alone, KP+TT score provides an advantage in positioning files of a bug report in the top-1 for SWT and ZXing, and in the top-5 for AspectJ, Eclipse and Tomcat. On the contrary, in the cases of Pillar1 and Pillar2 using KP+TT score did not make a difference and the results remained the same as the TT score in all top-N categories. Further analysis revealed the reason: in Pillar 2 the bug report summaries do not contain source code file names and in Pillar1 only 3 bug reports contain file names in their summaries, but they are not the files changed to resolve the reported bug and the TT score of the relevant files is higher.

On average for 64% of bug reports a relevant file can be located in the top-10 by just assigning a high score to file names in certain positions of the bug report summary, confirming the studies cited in the introduction (Section 4.1) that found file names mentioned in a large percentage of bug reports (Saha *et al.*, 2013; Schröter *et al.*, 2010). The file name occurrences in other places of the bug report will also be scored by comparing bug report and file terms in function *scoreWithFileTerms* (see Sub-section 3.3.2 in Chapter 3), but irrelevant files that match several terms may accumulate a large score that pushes the affected classes down the ranking.

Recall Table 4.2, the first word in the summary field of the bug #79268 in SWT is the relevant file name (see Figure 1.4 in Chapter 1). The existing tools rank this file at 21st and 11th position respectively. One of the reasons for this is that the word *Program* is considered as being too ambiguous and gets a lower score. However based on KP score logic in Algorithm

Table 4.6: Performance comparison of scoring variations: Key Position (KP+TT) vs Stack Trace (ST+TT) vs Text Terms (TT only)

Project	**Scoring**	**Top-1**	**Top-5**	**Top-10**	**MAP**	**MRR**
AspectJ	KP+TT only	12.9%	43.0%	59.1%	0.17	0.33
	ST+TT only	**21.7%**	**45.1%**	**59.4%**	**0.20**	**0.40**
	TT only	13.6%	43.7%	59.1%	0.17	0.34
Eclipse	KP+TT only	19.5%	35.2%	48.0%	0.21	0.32
	ST+TT only	**19.9%**	**35.92%**	**48.0%**	**0.21**	**0.33**
	TT only	18.3%	34.7%	48.0%	0.20	0.32
SWT	**KP+TT only**	**62.2%**	**79.6%**	**89.8%**	**0.60**	**0.81**
	ST+TT only	43.9%	76.5%	88.8%	0.50	0.70
	TT only	42.9%	76.5%	88.8%	0.50	0.69
ZXing	**KP+TT only**	**40.0%**	**65.0%**	**80.0%**	**0.46**	**0.53**
	ST+TT only	n/a	n/a	n/a	n/a	n/a
	TT only	25.0%	60.0%	75.0%	0.38	0.42
Tomcat	**KP+TT only**	**36.2%**	**56.3%**	**64.1%**	**0.39**	**0.49**
	ST+TT only	34.8%	56.3%	64.3%	0.39	0.49
	TT only	34.0%	55.7%	64.0%	0.38	0.48
ArgoUML	KP+TT only	12.1%	48.4%	56.0%	0.19	0.32
	ST+TT only	**13.2%**	**48.4%**	**56.0%**	**0.20**	**0.33**
	TT only	12.1%	48.4%	56.0%	0.19	0.32
Pillar1	**TT only**	**7.4%**	**33.3%**	**40.7%**	**0.10**	**0.36**
Pillar2	**TT only**	**25.0%**	**66.7%**	**75.0%**	**0.20**	**0.71**

4.2, my approach matches the relevant file in the first position of the summary sentence and assigns a high score of 10, thus ranks it at the 1$^{\text{st}}$ position in the result list (see first row, last column in Table 4.2).

4.4.1.2 Scoring with Stack Trace Information (ST score)

To see the impact of considering file names occurring in a stack trace, if it exists, I ran ConCodeSe with and without assigning an ST score, but again leaving all the rest unchanged, i.e. using key position (KP) and text terms (TT) scoring. Table 4.7 shows results for some affected classes obtained by BugLocator, BRTracer and BLUiR.

Again, the rank differences can be small but significant, moving a file from top-10 to top-5 (ResolvedMemberImpl.java) or from top-5 to top-1 (EclipseSourceType.java). In some cases the file goes from not being in the top-10 to being in the top-1 (ReferenceType.java), even if it is in the lowest scoring fourth position in the stack.

Table 4.6 also shows the effect of running ConCodeSe just with ST and TT scores together (ST+TT) against only using TT score, i.e. without KP and VSM scoring methods, except for ZXing and Pillar1, which don't have any stack traces in its bug reports (Table 4.4). ST+TT scoring provides significant advantage over the TT score alone in positioning affected files of

Table 4.7: Example of ranking achieved by leveraging stack trace information in AspectJ bug reports compared to other tools

AspectJ BR#	**Exception Description**	**Affected file**	**Stack pos.**	**Bug Locator**	**BR Tracer**	**BLUiR**	**ConCodeSe**	
							without	**with**
138143	NullPointer Exception	EclipseSourceType	1st	1	2	5	5	1
158624	Unsupport edOperation	ResolvedMemberImpl	2nd	16	6	56	7	2
153490	IllegalState Exception	ReferenceType	4th	122	3	74	11	1

a bug report in top-1. In particular for AspectJ, Eclipse, Tomcat and ArgoUML, ST scoring places more bug reports in all top-N categories indicating that giving file names found in stack trace a higher score contributes to improving the performance of the results, which is also in line with the findings of previous studies (Schröter *et al.*, 2010; Moreno *et al.*, 2014; Wong *et al.*, 2014).

Note that there is no significant difference between ST+TT and TT scoring for SWT and ArgoUML. Only 4 of SWT's bug reports have a stack trace and it happens that in those cases the lower TT score value of 2 for the files occurring in the stack trace is still enough to rank them highly. For ArgoUML, 5 bug reports contain stack trace information and using ST+TT scoring adds only one more bug report to the top-1 compared to other two scoring variations. The small difference is due to the relevant file for the other 4 bug reports not being among the ones listed in the stack trace or being listed at a position after the 4th. Since ST scoring only considers the first 4 files, in that case the affected file gets a TT score, because its name occurs in the bug report description.

Again, for Pillar1 and Pillar2 using ST+TT scoring alone did not make a difference and the results remained constant in all top-N categories. None of the Pillar2 bug reports contains stack traces and in the case of Pillar1 only 1 bug report description contains stack traces but the relevant file is listed after the 4th position and gets a ST score of zero (Section 4.3.1.2).

Recall Figure 1.3 in Chapter 1, the AspectJ bug #158624 contains a detailed stack trace. When the search results between the tools that utilises the stack trace in bug reports (Wong *et al.*, 2014) and the other tool that does not (Zhou *et al.*, 2012a) are compared, the ranking of the relevant file improves from 16th to 6th position. However based on ST score logic in Algorithm 4.3, my approach finds the relevant file listed in the second position of the stack trace and assigns a high score of 7, thus ranks it at the 2nd position.

Table 4.8: Performance comparison between using combination of all scores vs only TT score during search with only concept terms

Project	Scoring	Top-1	Top-5	Top-10	MAP	MRR
AspectJ	TT Only	2%	13%	26%	0.04	0.10
	All (KP+ST+TT)	**9%**	**19%**	**31%**	**0.09**	**0.18**
Eclipse	TT Only	2%	6%	11%	0.03	0.06
	All (KP+ST+TT)	**11%**	**18%**	**22%**	**0.10**	**0.16**
SWT	TT Only	19%	38%	45%	0.22	0.32
	All (KP+ST+TT)	**45%**	**54%**	**59%**	**0.39**	**0.54**
Tomcat	TT Only	3%	10%	11%	0.06	0.08
	All (KP+ST+TT)	**15%**	**23%**	**27%**	**0.16**	**0.20**
Average	TT Only	7%	17%	23%	0.09	0.14
	All (KP+ST+TT)	**16%**	**27%**	**33%**	**0.16**	**0.24**

4.4.1.3 KP and ST Score improvement when Searching with Concepts Only vs all Bug Report Vocabulary

Table 3.8 in Chapter 3 showed that the TT score, i.e. function *scoreWithFileTerms* (Algorithm 3.1), placed at least one affected file into top-10, on average for 28% of the bug reports when only the domain concept terms are used during the search. To find out any advantage the KP and ST scores provide, I repeated the search performed in Sub-section 3.4.3 again using only the concept terms. Table 4.8 shows the improvements when the KP and ST scores are also utilised together with the TT score to rank files during the search. In the case of AspectJ, Eclipse, SWT and Tomcat, on average 10% more (23% vs 33%) bug report files are positioned into the top-10.

In the case of ArgoUML, ZXing, Pillar1 and Pillar2, the KP and ST scores did not provide any performance advantage over the TT score thus not listed in Table 4.8. Investigating the reasons revealed that the Pillar2 bug reports don't contain any stack trace thus can not take advantage of the ST scoring, and the bug reports of Pillar1 and ZXing contain only one stack trace where the relevant file already gets scored in the top-10 via the TT score. In the case of ArgoUML only 5 bug reports contain stack trace but the affected files are either not those listed in the stack trace or they are listed after the 4th position. Furthermore, the bug summary sentences in Pillar1 and Pillar2 do not refer to any source code file names, thus the KP score has no effect. In the case of ArgoUML and ZXing, the affected files of the bug reports are still ranked in the top-N via the TT score.

Recall in Sub-section 3.4.3, to improve the performance, I enhanced my approach to search using the actual words of the bug report rather than its associated concepts. Table 3.9 in Chapter 3 shows that the simple fact of considering all the vocabulary of a bug report during

Table 4.9: Performance comparison between using combination of Key Word and Stack Trace scoring On vs Off

Project	KP + ST	Top-1	Top-5	Top-10	MAP	MRR
AspectJ	**on**	**42.3%**	**68.2%**	**78.3%**	**0.32**	**0.67**
	off	35.0%	67.1%	78.3%	0.30	0.62
Eclipse	**on**	**37.6%**	**61.2%**	**69.9%**	**0.37**	**0.57**
	off	34.6%	59.6%	69.7%	0.35	0.55
SWT	**on**	**72.4%**	**89.8%**	**92.9%**	**0.68**	**0.94**
	off	59.2%	88.8%	92.9%	0.62	0.85
ZXing	**on**	**55.0%**	**75.0%**	**80.0%**	**0.55**	**0.68**
	off	35.0%	70.0%	80.0%	0.45	0.52
Tomcat	**on**	**51.5%**	**69.2%**	**75.4%**	**0.52**	**0.66**
	off	49.1%	68.6%	75.3%	0.51	0.65
ArgoUML	**any**	**31.9%**	**61.5%**	**65.9%**	**0.30**	**0.55**
Pillar1	**any**	**29.6%**	**59.3%**	**63.0%**	**0.22**	**0.69**
Pillar2	**any**	**33.3%**	**66.7%**	**83.3%**	**0.26**	**0.92**

the search ranked at least one or more relevant files in the top 10, on average for 76% of the bug reports. Thus, I end the analysis of the contributions of positional scoring of file names in bug report summaries (key position) and stack trace (ST) with Table 4.9, which shows the combined rank ‘boosting’ effect of positional scoring, i.e. using KP and ST scoring together vs not using it. For example, in the case of the SWT project, using summary and stack trace scoring places an affected source file in the top-1 for 72% of the bug reports compared to the base case (TT score) of 59%. This performance advantage remains noticeable high for all the projects except for Pillar1 and Pillar2 due to the stated reasons, i.e. setting the **KP+ST** scoring to **on** or **off** did not provide any advantage. Therefore only one set of values are presented for those projects (see row ***‘any’*** in Table 4.9).

4.4.1.4 Variations of Score Values

The KP and ST scores are computed in very simple ways, which may affect the performance of the outcome. Variations to those scores may or may not further enhance the performance. I experimented further to evaluate the effects of my scoring mechanism by assigning different scores to the four positions in the summary and in the stack trace. Table 4.10 shows the results obtained after performing the following changes:

1. The scores were halved, e.g. 10, 8, 6, 4 became 5, 4, 3, 2.

2. The scores were reversed, i.e. 4, 6, 8, 10 and 3, 5, 7, 9.

3. All 8 positions (4 in the BR summary and 4 in the ST) have a uniform score of 10.

Table 4.10: Performance comparison of using variable values for KP, ST and TT scores

Project	Approach	Top-1	Top-5	Top-10	MAP	MRR
Aspectj	**halved**	**42.0%**	**68.5%**	**78.3%**	**0.33**	**0.67**
	reversed	34.6%	65.0%	77.6%	0.30	0.61
	uniform	37.4%	68.2%	78.3%	0.31	0.64
	close2base	41.6%	68.2%	78.3%	0.33	0.67
Eclipse	halved	37.2%	61.1%	69.8%	0.37	0.57
	reversed	35.7%	61.2%	69.9%	0.36	0.56
	uniform	36.6%	61.2%	69.9%	0.36	0.56
	close2base	**37.8%**	**67.8%**	**78.3%**	**0.36**	**0.57**
SWT	halved	71.4%	89.8%	92.9%	0.68	0.93
	reversed	69.4%	88.8%	92.9%	0.66	0.92
	uniform	71.4%	89.8%	92.9%	0.68	0.94
	close2base	**72.4%**	**89.8%**	**92.9%**	**0.68**	**0.94**
Tomcat	halved	50.9%	68.8%	75.5%	0.52	0.65
	reversed	50.5%	68.8%	75.5%	0.52	0.65
	uniform	51.0%	69.2%	75.4%	0.52	0.66
	close2base	**51.7%**	**69.1%**	**75.4%**	**0.52**	**0.66**
ArgoUML	**any**	31.9%	61.5%	65.9%	0.30	0.55
ZXing	**any**	55.0%	75.0%	80.0%	0.51	0.63
Pillar1	**any**	29.6%	59.3%	63.0%	0.22	0.69
Pillar2	**any**	33.3%	66.7%	83.3%	0.26	0.92

4. The scores were made closer to those of TT (close2base):

   (a) for the summary positions: 3.00, 2.80, 2.60, 2.40

   (b) for the stack positions: 2.90, 2.70, 2.50, 2.30

Halving and close2base keeps the order of each set of 4 positions and the results obtained are similar to those obtained with the original score values, which are 10, 8, 6, 4 for summary position and 9, 7, 5, 3 for stack trace positions. The reversed and uniform scoring break that order and led to the worst results. This confirms that the relative importance of the various positions (especially the first position) found through inspection of SWT and AspectJ applies to most projects.

In the cases of Pillar1 and Pillar2 changing the KP and ST scoring doesn't make a difference because none of the Pillar2 bug reports contained stack traces and only 1 of the Pillar1 bug report description contained stack traces. In the case of ZXing neither of the scoring variations made a difference due to the small number of bug reports so only one set of performance results are reported (see row '***any***' in Table 4.10). In the case of SWT, it made little difference as it has only a few more bug reports than ArgoUML.

Looking closer at Table 4.10, I note that in the cases of Eclipse, SWT and Tomcat, **close2base** places more bug reports in top-1 than any other variation. Investigating more closely, I found that one additional bug report for SWT and two for Tomcat are placed in top-1. In the case of SWT, the only affected source code file, Spinner.java for bug report #96053, achieved a TT score of 2.56 by function *scoreWithFileTerms* (Algorithm 3.1) and is ranked in the 2$^{\text{nd}}$ position whereas the file Text.java achieved a KP score of 4 and is ranked 1$^{\text{st}}$. Analysing further revealed that the last word "text" in the bug report summary sentence matches the file name, thus assigning a high KP score value of 4 to the file Text.java. However, when close2base scores are used, the KP score value for the last word position is set to 2.40 (see point 4a in the variations list above), which is lower than the TT score (2.56), thus ranking Spinner.java as 1$^{\text{st}}$ and Text.java as 2$^{\text{nd}}$. Similar patterns were discovered in Eclipse and Tomcat.

Comparing Table 4.10 to the results achieved by other approaches (Figure 4.3, 4.4and 4.5 in the next Sub-section), I note that the halved and close2base variations outperform the other approaches in most cases, showing that the key improvement is the higher and differentiated scoring of the 4 positions in the summary and stack trace, independently of the exact score.

To sum up, the four systematic transformations of the score values and the better performance of the halved and close2base transformations provide evidence that the precise value of each score is not the main point but rather their relative values. Moreover, the heuristics (the more likely occurrence of relevant file names in certain positions) were based on the analysis of only 10% of the bug reports. Especially for large projects like Eclipse, with many people reporting bugs, one can reasonably expect that many bug reports will deviate from the sample. The similar behaviour of the variants across all projects and all bug reports (close2base and halved are better than uniform, which is better than reversed) therefore provides reassurance that the chosen values capture well the heuristics and that the heuristics are valid beyond the small sample size used to obtain them.

4.4.1.5 Overall Results

Finally, I compare the performance of ConCodeSe against the state-of-the-art tools using their datasets and metrics (Figures 4.1, 4.2, 4.3, 4.4 and 4.5).

As mentioned before, I was only able to obtain BugLocator and BRTracer[2], which meant that for the other approaches I could only refer to the published results for the datasets

[2] Although BLIA is available online, I was unable to run it on datasets other than the ones used by its authors (AspectJ, SWT and ZXing)

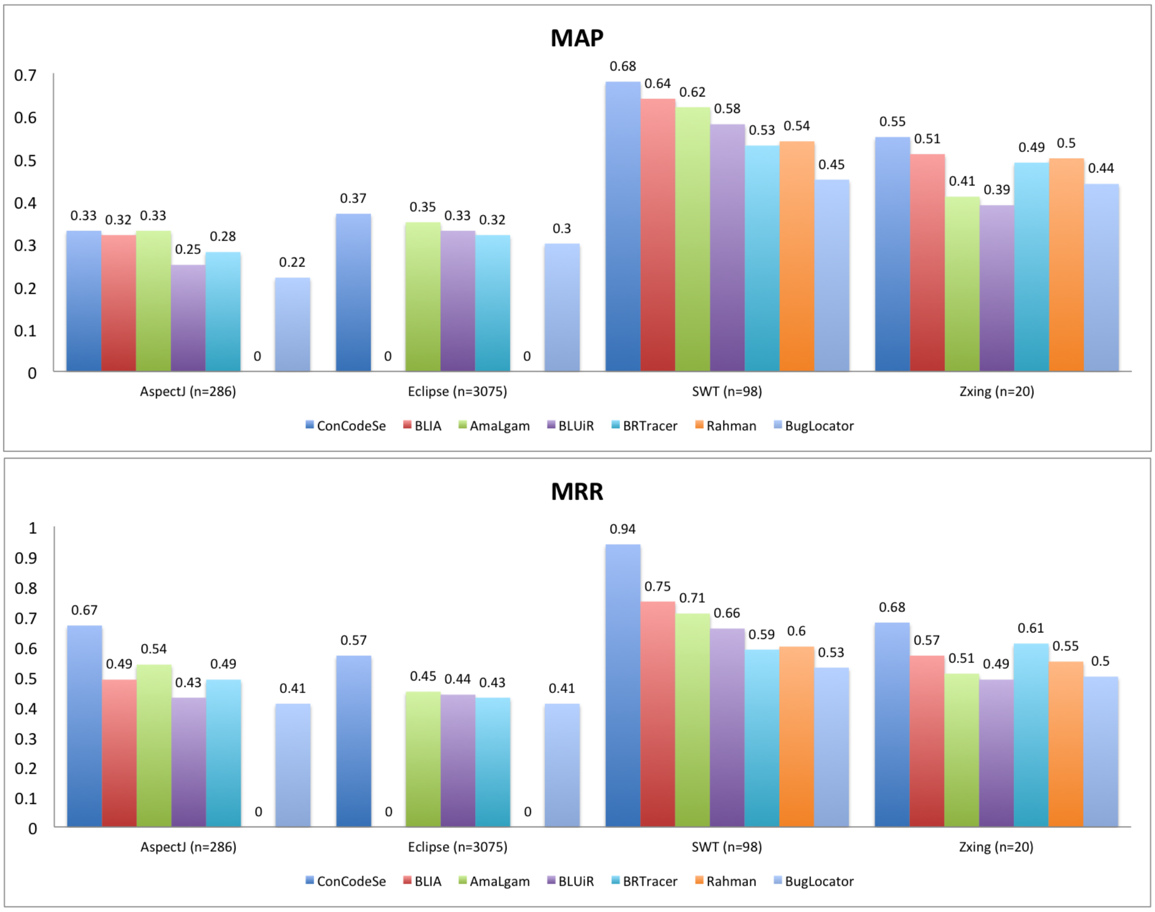


Figure 4.1: Performance comparison of MAP and MRR values per tool for AspectJ, Eclipse and SWT (n=number of BRs analysed)

they used. This means I could compare my Pillar1 and Pillar2 results only against those two approaches and couldn't for example run BLUiR and AmaLgam on the projects used by LOBSTER and LtR and vice versa. LtR's top-N values for Tomcat were computed from the ranking results published in LtR's online annex (Ye *et al.*, 2014). LtR also used AspectJ, Eclipse and SWT but with a different dataset to that of BugLocator. The online annex only included the Tomcat source code, so I was unable to rank the bug reports for the other projects with LtR.

Figures 4.1, 4.2, 4.3, 4.4 and 4.3 show that except for AmaLgam's top-1 performance on AspectJ, ConCodeSe outperforms or equals all tools on all metrics for all projects, including BugLocator's MAP for ZXing, which BLUiR and AmaLgam weren't able to match. For LOBSTER, the authors report MAP and MRR values obtained by varying the similarity distance in their approach, and I took their best values (0.17 and 0.25). LOBSTER only investigates the added value of stack traces so to compare like for like, I ran ConCodeSe on their ArgoUML dataset using only the ST scoring and improved on their results (Figure 4.2 ConCodeSe-(ST) row).

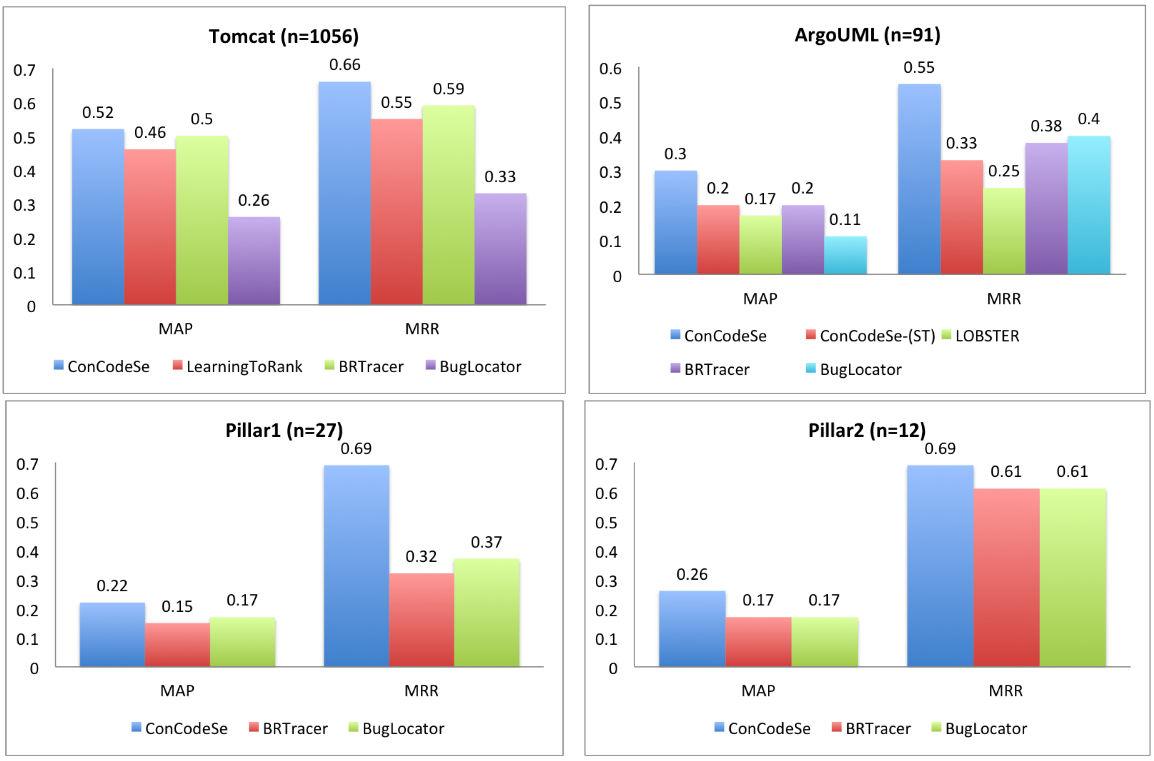


Figure 4.2: Performance comparison of MAP and MRR values per tool for Tomcat, ArgoUML, Pillar1 and Pillar2 (n=number of BRs analysed)

I note that ConCodeSe always improves the MRR value, which is an indication of how many files a developer has at most to go through in the ranked list before finding one that needs to be changed. User studies (Kochhar *et al.*, 2016; Xia *et al.*, 2016) indicate that developers only look at the top-10, and especially the top-5, results. ConCodeSe has the best top-5 and top-10 results across all projects.

I also get distinctly better results than Rahman *et al.* (2015), the only approach to explicitly use file names found in bug reports. Looking at the largest project Eclipse (Figure 4.3) reveals that even small 1.7% top-1 improvement over the second best approach (BLUiR) represents 52 more bug reports for which the first recommended file is relevant, thus helping developers save time.

I also notice that my tool performs almost 2% worse than AmaLgam (42.3% vs 44.4%) for AspectJ when placing relevant files in top-1. Investigating the reasons revealed that in 2 of AspectJ bug reports a FileNotFound exception is reported and the changed file is ranked in 2nd position despite being listed in the stack trace. This is because the stack trace lists a utility file in the 1st position but the affected file in the 2nd position. For example, in AspectJ bug report #123212, the file *AjBuildManager.java* uses *FileUtil.java* to write information to an external file and the FileNotFound exception is thrown by FileUtil first and then propagated

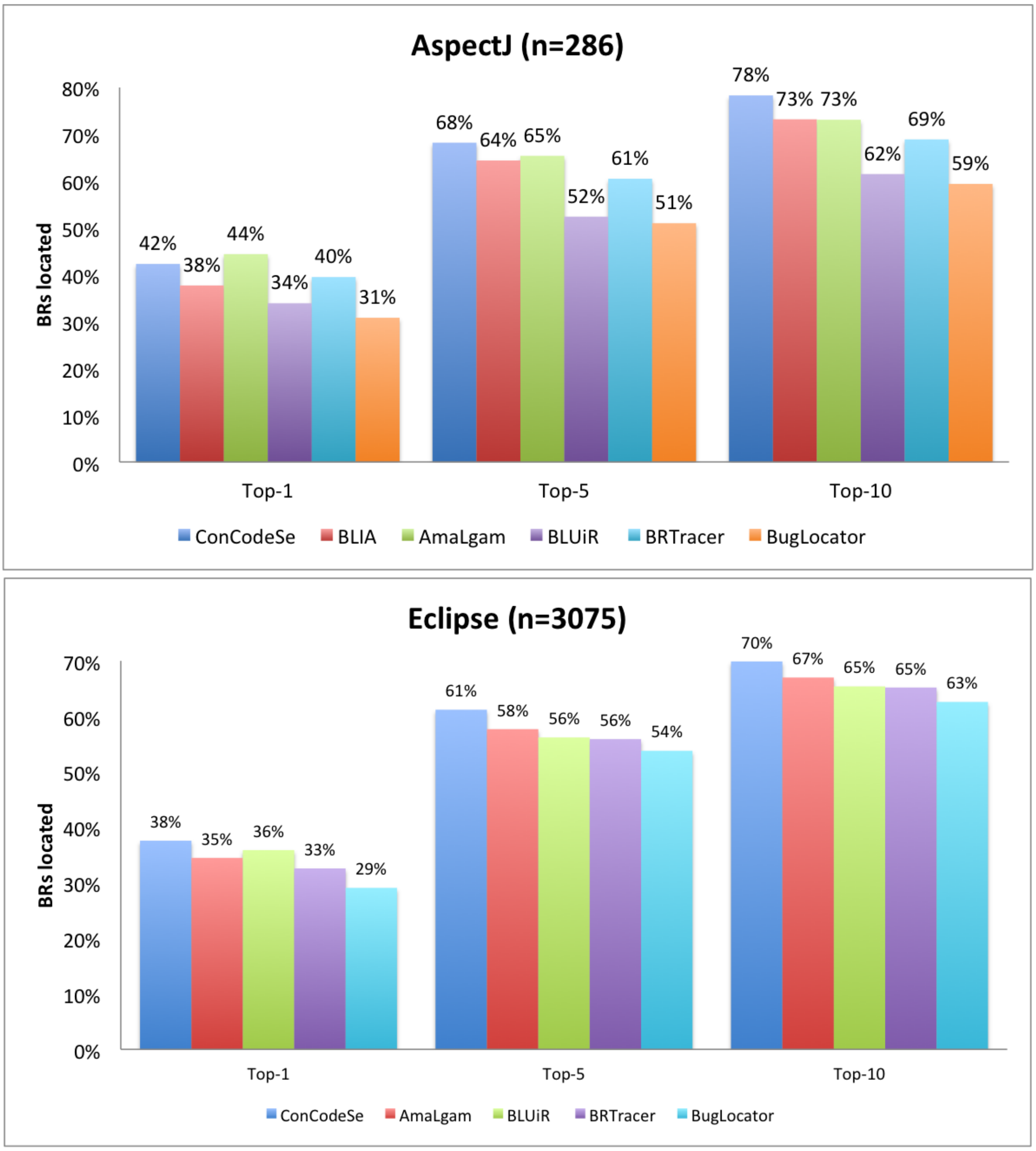


Figure 4.3: Performance compare of the tools for AspectJ and Eclipse (n=BRs analysed)

to its users like AjBuildManager. Since ST scores are assigned based on the order of each file appearing in the stack trace, in the case of AspectJ bug report #123212, FileUtil gets a higher score than AjBuildManager. To handle this scenario, I experimented by adjusting my ST scoring values but the overall results deteriorated.

In the case of Pillar1, my tool achieves a significantly higher performance over BugLocator and BRTracer in all top-N categories (Figure 4.5). It is also interesting to see that BugLocator outperforms BRTracer in the top-1 and top-5 metrics despite that BRTracer is reported to be an improvement over BugLocator. In the case of Pillar2, although my approach achieves identical performance to the second best in the top-5 and top-10 metrics, it is far superior

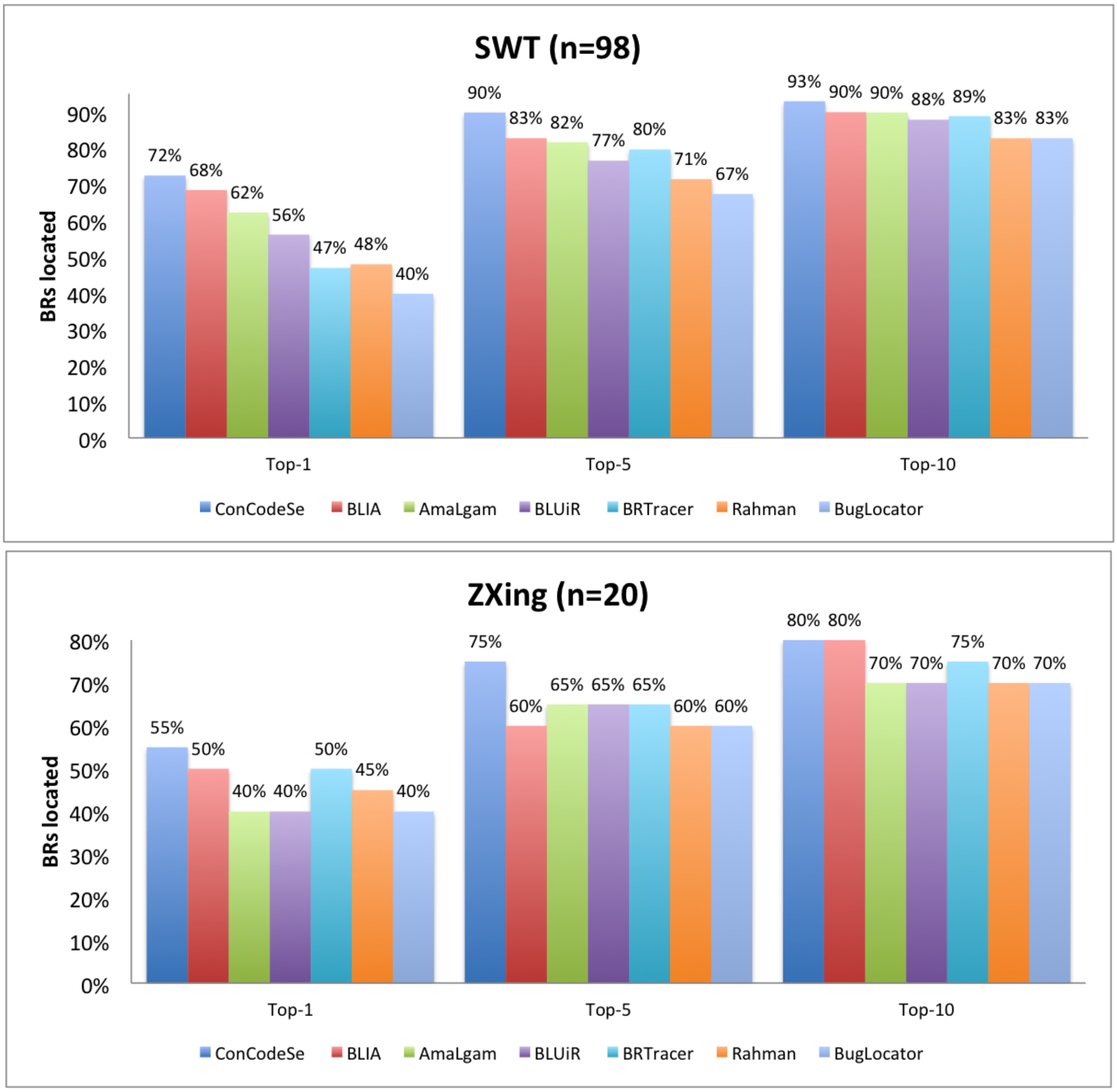


Figure 4.4: Performance compare of the tools for SWT and ZXing (n=BRs analysed)

in the top-1 metric and thus outperforms the other approaches in terms of MAP and MRR (Figures 4.1 and 4.2).

Recall Figure 1.2 in Chapter 1, the SWT bug report #92757 has a very terse description, BugLocator (Zhou *et al.*, 2012a) and BRTracer (Wong *et al.*, 2014) rank the relevant file at 88th and 75th positions respectively. However my approach matches the relevant file based on the stemmed terms from the bug report and the file's code, thus ranks it in the 5th position.

Pillar1 and Pillar2 are evidence that my approach performs well even if a bug report doesn't mention file names. As a further example, SWT bug report #58185 mentions no files and yet ConCodeSe places 3 of the 4 relevant files in the top-5, whereas BugLocator and BRTracer only place 2 files. Similarly, AspectJ bug report #95529 contains no file names but out of the 11 relevant files, ConCodeSe ranks 3 files in the top-5 and 1 in the top-10 whereas BugLocator and BRTracer only rank 1 relevant file in the top-5. In all these cases the KP

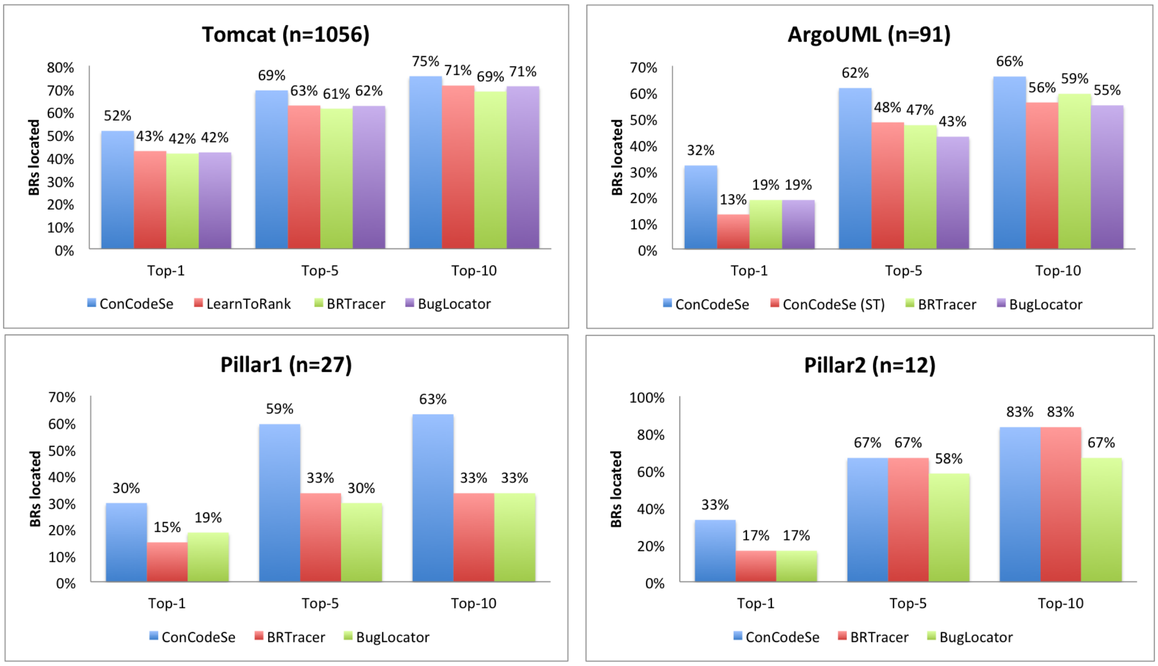


Figure 4.5: Performance compare of the tools for Tomcat, ArgoUML, Pillar1 and Pillar2 (n=number of BRs analysed)

and ST scores are zero and Algorithm 4.1 uses the lexical (TT) score. The TT and VSM scores are computed with and without using the file's comments, with and without stemming. Thus, even if a bug report doesn't mention file names, my approach still obtains 2*2*2 = 8 ranks for each file, to choose the best of them.

Having always 8 ranks to choose from also helps with the other extreme: the bug report includes many file names, but most are irrelevant. For example, SWT bug report #83699 mentions 14 files but only 1 is relevant. I rank it in the 4th position using the file's comments, whereas BugLocator and BRTracer rank it 9th and 14th, respectively. Similarly, AspectJ bug report #46298 has a very verbose description that mentions 9 files, none relevant. I list the only relevant file in the 2nd position using VSM and comments; BugLocator and BRTracer list it in 6th and 12th position respectively.

A bug report can be very terse, e.g. a short sentence in the summary and an empty description field, like SWT bug report #89533. In this example my tool ranks the only relevant file in 3rd position, by applying VSM and stemming to the bug report summary and the file's comments, whereas BugLocator and BRTracer rank the same file in 305th and 19th position respectively. Similarly, for AspectJ bug report #39436, the only relevant file is ranked in the top-5, based on the comments in the file, whereas BugLocator and BRTracer rank the same file below the top-10.

Table 4.11: Number of files for each bug report placed in the top-10 by ConCodeSe vs BugLocator and BRTracer

Project	**BugLocator**			**BRTracer**		
	Better	**Same**	**Worse**	**Better**	**Same**	**Worse**
AspectJ	45%	48%	7%	30%	55%	15%
Eclipse	23%	64%	13%	22%	65%	13%
SWT	24%	69%	6%	21%	74%	4%
ZXing	10%	85%	5%	20%	75%	5%
Tomcat	22%	66%	12%	56%	37%	7%
ArgoUML	25%	62%	13%	29%	59%	12%
Pillar1	48%	48%	4%	48%	48%	4%
Pillar2	58%	33%	8%	50%	33%	17%

4.4.1.6 Performance

Figures 4.3, 4.4 and 4.5 only counts for how many bug reports at least one affected file was placed in the top-N. The MAP and MRR values indicate that ConCodeSe tends to perform better for each bug report compared to other tools, so I additionally analysed the per bug report performance to measure the number of files for each bug report placed in the top-10. This analysis required access to per bug report results and the only publicly available tools that functioned once installed were BugLocator and BRTracer.

Table 4.11 shows, for example, that for 128/286=45% (resp. 85/286=30%) of AspectJ's bug reports, my tool placed more files in the top-10 than BugLocator (resp. BRTracer). This includes bug reports in which ConCodeSe placed at least one and the other tools placed none. The '***same***' columns indicate the percentage of bug reports for which both tools placed the same number of affected files in the top-10. This includes cases where all approaches can be improved (because neither ranks an affected file in the top-10), and where none can be improved (because all tools place all the affected files in the top-10).

Neither Figures 4.3,4.4, 4.5 nor Table 4.11 show the recall, i.e. the number of relevant files out of all the effected relevant files placed in the top-N. To find out the recall at top-N, similar to Sub-section 3.4.3.1 in Chapter 3, I have calculated the min, max, mean and median values of the results for each project (Table 3.10). Again, the recall is capped at 10 (N=10), which means that if a bug report has more than 10 relevant files only the 10 are considered.

The min and max values were always 0 and 1 respectively because my approach is either able to position all the relevant files in top-10 or none. I found that the top-10 recall performance remained the same between all combined scores (KP+ST+TT) compared to the TT score alone in all of the projects as reported in Table 3.10.

On the contrary, the recall at top-5 marginally improved (1%) for AspectJ, Eclipse, SWT and Tomcat. In the case of ZXing, ArgoUML, Pillar1 and Pillar2, the recall at top-5 between the combined scores and the TT score alone remains the same due to the reasons explained in Sub-section 4.4.1.3, i.e. absence of file names or stack trace in the summary or description of the bug reports.

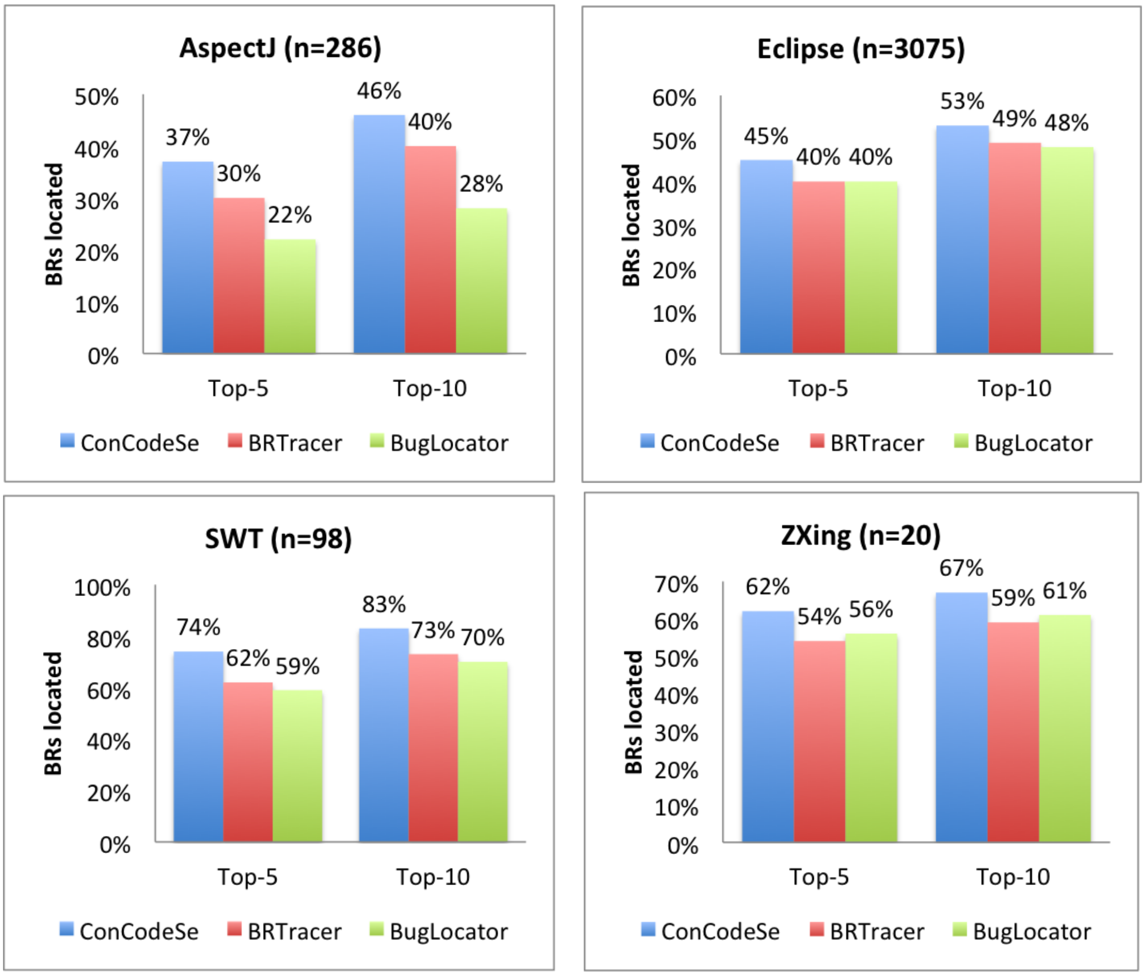


Figure 4.6: Recall performance of the tools for top-5 and top-10 for AspetcJ, Eclipse, SWT and ZXing (n=number of BRs analysed)

Furthermore, I compared the recall performance of ConCodeSe against BugLocator and BRTracer as they were the only available tools. Once again, the min and max values were always 0 and 1 respectively because the tools are either able to position all the relevant files in top-10 or not. Therefore in Figures 4.6 and 4.7, I report only the average (mean) for the top-5 and the top-10 metrics.

The results reveal that ConCodeSe performance is significantly superior to the other two current state-of-the-art tools even when a different metric, i.e. recall, is used to measure the performance instead of MAP and MRR as reported in their respective studies (Zhou *et al.*, 2012a; Wong *et al.*, 2014).

I also note that BRTracer performance is weaker than BugLocator in Tomcat, Pillar1 and ZXing. Since BRTracer improves upon BugLocator by using segmentation and stack trace, in the case of smaller size projects, i.e. Pillar1 and ZXing (Table 2.1) or due to fewer stack traces in the bug reports (Table 4.4), the improvements fall short over BugLocator.

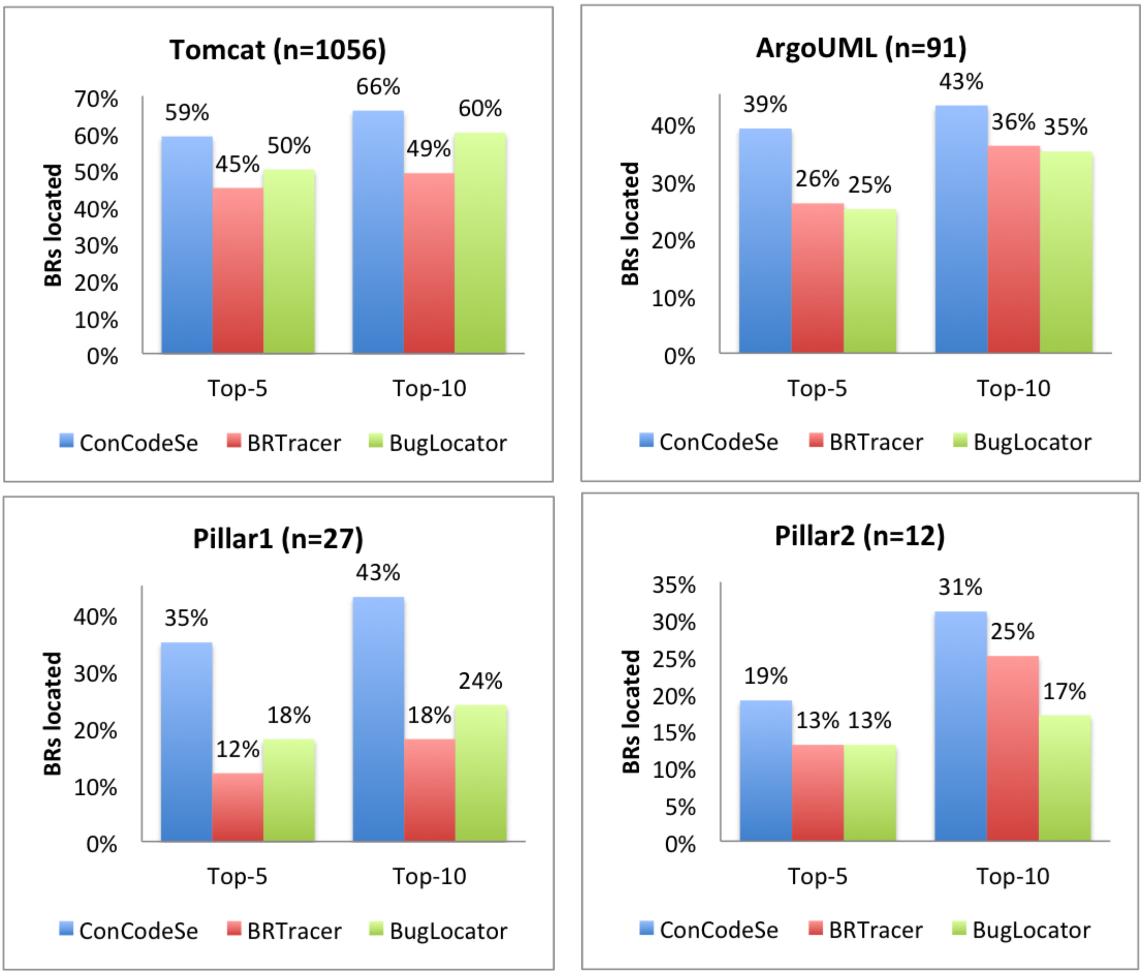


Figure 4.7: Recall performance of the tools for top-5 and top-10 for Tomcat, ArgoUML, Pillar1 and Pillar2 (n=number of BRs analysed)

From all the results shown I can answer RQ2 affirmatively: leveraging the occurrence of file names in bug reports leads in almost every case to better performance than using project history.

4.4.2 RQ2.1: Scoring without Similar Bugs

In previous Section, I confirmed that file names occurring in bug reports provide improved performance, thus in this Section, I seek to *evaluate the contribution of using similar bug reports to the results performance in other tools compared to my approach,* as asked by my first sub-research question in Section 4.1. Earlier I described that BugLocator, BRTracer, BLUiR, AmaLgam, BLIA and Rahman *et al.* (2015) utilise a feature called SimiScore, which

Table 4.12: Example of ranking achieved by leveraging similar bug reports in other tools vs not in ConCodeSe

BR#	Affected Java Files	BugLocator		BRTracer		BLUiR		Con CodeSe
		rVSM	Simi Score	no Simi Score	Simi Score	structure	Simi Score	
78856	OS.java	37	36	89	88	3	6	1
79419	Link.java	18	18	4	4	31	6	1
	OS.java	58	58	48	47	11	21	1
83262	RTFTransfer	224	224	214	214	43	79	35
	TextTransfer	202	202	198	197	-	-	36
87676	Tree.java	49	21	10	6	4	3	2

uses the bug report terms to find similar closed bug reports. The files changed to fix those bug reports are suggested as likely candidates for the current bug report being searched. To answer RQ2.1 I ran BugLocator and BRTracer with and without SimiScore, as explained at the start of Section 5.4.

Unfortunately, I was unable to obtain Rahman *et al*'s tool, BLUiR and AmaLgam to perform runs without SimiScore, but I do not consider this to be a handicap because from the published results it seems that SimiScore benefits mostly BugLocator and BRTracer.

I selected the SWT bug reports reported in the BLUiR paper (#78856, #79419, #83262 and #87676) and then ran BugLocator and BRTracer to compare their performance. As shown in Table 4.12, BugLocator placed the file Tree.java in the 49th and 21st positions in the ranked list by using their revised VSM (rVSM) approach first and then by considering similar bug reports. In the case of BRTracer, the introduced segmentation approach already ranked the file in the top-10 (10th position) and SimiScore placed the same file at an even higher position (6th). In the case of BLUiR, the same file is placed at 4th and 3rd positions respectively. For the other cases in the table, SimiScore doesn't improve (or only slightly so) the scoring for BugLocator. In the case of BLUiR, apart from the great improvement for Link.java, SimiScore leads to a lower rank than structural IR.

It can be seen in Table 4.13 that ConCodeSe achieves on average 46% (0.40/0.25) and 35% (0.57/0.40) superior results to BugLocator in terms of MAP and MRR respectively, thus allowing my approach to be more suitable in projects without closed bug reports similar to the new bug reports.

I run BugLocator and BRTracer without SimiScore on all projects, to have a more like-for-like comparison with ConCodeSe in terms of input (no past bug reports). Comparing Figures 4.3, 4.4, 4.5 (with SimiScore) and Table 4.13 (without) shows a noticeable performance decline in BugLocator and BRTracer when not using similar bug reports and thus even

Table 4.13: Performance of ConCodeSe compared to BugLocator and BRTracer without using similar bug reports score across all projects

Project	**Approach**	**Top-1**	**Top-5**	**Top-10**	**MAP**	**MRR**
AspectJ	BugLocator	22.7%	40.9%	55.6%	0.19	0.18
	BRTracer	38.8%	58.7%	66.8%	0.27	0.47
	ConCodeSe	**42.3%**	**68.2%**	**78.3%**	**0.33**	**0.67**
Eclipse	BugLocator	24.4%	46.1%	55.9%	0.26	0.35
	BRTracer	29.6%	51.9%	61.8%	0.30	0.40
	ConCodeSe	**37.6%**	**61.2%**	**69.9%**	**0.37**	**0.57**
SWT	BugLocator	31.6%	65.3%	77.6%	0.40	0.47
	BRTracer	46.9%	79.6%	88.8%	0.53	0.59
	ConCodeSe	**72.4%**	**89.8%**	**92.9%**	**0.68**	**0.94**
ZXing	BugLocator	40.0%	55.0%	70.0%	0.41	0.48
	BRTracer	45.0%	70.0%	75.0%	0.46	0.55
	ConCodeSe	**55.0%**	**75.0%**	**80.0%**	**0.55**	**0.68**
Tomcat	BugLocator	42.1%	62.4%	71.0%	0.26	0.33
	BRTracer	36.6%	57.3%	65.6%	0.45	0.56
	ConCodeSe	**49.9%**	**69.2%**	**75.4%**	**0.52**	**0.66**
ArgoUML	BugLocator	18.7%	42.9%	54.9%	0.11	0.48
	BRTracer	18.7%	46.2%	54.9%	0.20	0.38
	ConCodeSe	**31.9%**	**61.5%**	**65.9%**	**0.30**	**0.55**
Pillar1	BugLocator	18.5%	29.6%	33.3%	0.17	0.37
	BRTracer	14.8%	29.6%	29.6%	0.14	0.31
	ConCodeSe	**29.6%**	**59.3%**	**63.0%**	**0.22**	**0.69**
Pillar2	BugLocator	16.7%	58.3%	66.7%	0.17	0.61
	BRTracer	16.7%	66.7%	83.3%	0.17	0.61
	ConCodeSe	**33.3%**	**66.7%**	**83.3%**	**0.26**	**0.69**
Average	BugLocator	27%	50%	61%	0.25	0.40
	BRTracer	31%	57%	65%	0.33	0.50
	ConCodeSe	**44%**	**69%**	**76%**	**0.40**	**0.57**

greater improvement achieved by ConCodeSe. BLUiR without SimiScore also outperforms BugLocator and BRTracer with SimiScore. Interestingly BRTracer and ConCodeSe perform equally well in top-5 and top-10 for Pillar2.

I answer RQ2.1 by saying that although the contribution of similar bug reports significantly improves the performance of BugLocator and BRTracer, it is not enough to outperform ConCodeSe. The large contribution of SimiScore for BugLocator and BRTracer is mainly due to the lower baseline provided by rVSM, as reinforced by the results in the next section.

4.4.3 RQ2.2: VSM's Contribution

As described in Sub-section 4.3.1, 4 of a file's 8 rankings are obtained with the VSM probabilistic method available in the Lucene library, and the other 4 with my lexical similarity ranking method (Algorithm 4.1). To find out the added value of VSM, my second sub-research question aims to evaluate *the overall contribution of the VSM variant adopted in my approach,*

Table 4.14: Performance comparison between Lucene VSM vs lexical similarity scoring within ConCodeSe

Project	Approach	Top-1	Top-5	Top-10	MAP	MRR
AspectJ	VSM	28.0%	47.2%	60.1%	0.22	0.49
	lexical similarity	20.6%	44.4%	59.4%	0.20	0.39
Eclipse	VSM	23.3%	45.7%	56.6%	0.25	0.41
	lexical similarity	18.6%	36.3%	48.2%	0.21	0.32
SWT	VSM	39.8%	71.4%	82.7%	0.46	0.66
	lexical similarity	**63.3%**	**79.6%**	**89.8%**	**0.60**	**0.82**
ZXing	VSM	35.0%	55.0%	65.0%	0.37	0.45
	lexical similarity	**40.0%**	**70.0%**	**80.0%**	**0.47**	**0.54**
Tomcat	VSM	28.1%	47.7%	57.6%	0.32	0.43
	lexical similarity	**34.0%**	**56.6%**	**64.0%**	**0.38**	**0.48**
ArgoUML	VSM	24.2%	51.6%	58.2%	0.24	0.45
	lexical similarity	13.2%	48.4%	56.0%	0.20	0.33
Pillar1	VSM	25.9%	48.1%	55.6%	0.18	0.59
	lexical similarity	7.4%	33.3%	40.7%	0.10	0.36
Pillar2	VSM	8.3%	66.7%	75.0%	0.18	0.62
	lexical similarity	**25.0%**	**66.7%**	**75.0%**	**0.20**	**0.71**

and how it performs compared to rVSM. I performed two runs, one with VSM only and the other with lexical similarity scoring only (Table 4.14).

VSM outperforms the lexical scoring for the larger projects (AspectJ and Eclipse), i.e. with most code files (Table 2.1), and underperforms for small projects (SWT, ZXing and Pillar2). In cases of medium size projects (Tomcat and ArgoUML), lexical scoring outperforms VSM for Tomcat and underperforms for ArgoUML. As to be expected, each scoring by itself has a poorer performance than ConCodeSe, which combines both, whilst the total number of bug reports located by ConCodeSe (Figures 4.3, 4.4 and 4.5) is not the sum of the parts. In other words, many files are ranked in the top-N by both scoring methods, and each is able to locate bug reports the other can't. ConCodeSe literally takes the best of both worlds.

For all projects except ArgoUML, VSM by itself performs poorer than the best other approach (Figure 4.5), which shows the crucial contribution of lexical similarity scoring for the improved performance of my approach.

The VSM variant I adopt outperforms in many cases rVSM, introduced in BugLocator and also utilised in BRTracer.Even ConCodeSe's lexical similarity by itself outperforms rSVM in most cases. This can be seen by comparing the VSM (or lexical similarity) rows of Table 4.14 against the BugLocator rows of Table 4.13, where SimiScore is turned off to only use rVSM. Lexical similarity alone in ConCodeSe also outperforms rVSM for the two small and medium projects, except top-1 for ArgoUML.

I thus answer RQ2.2 by concluding that VSM provides a bigger contribution for projects with a large number of files, which makes the use of term and document frequency more meaningful. I also confirm that the exact IR variant used is paramount: Lucene's VSM and my simple lexical matching outperform BugLocator's bespoke rVSM in many cases as well as BLUiR's Okapi especially for SWT and ZXing. However, VSM on its own isn't enough to outperform other approaches.

4.5 Discussion

It has been argued that textual information in bug report documents is noisy (Zimmermann *et al.*, 2010). My approach is partly based on focussing on one single type of information in the bug report: the occurrence of file names. The assumption is that if a file is mentioned in the bug report, it likely needs to be changed to fix the bug. Manual inspection of only 9.4% of the bug reports, reinforced that assumption and revealed particular positions in the summary and in the stack trace where an affected file occurs. My improved results provide further evidence of the relevance of the assumption.

On the other hand, it has been argued that if bug reports already mention the relevant files, then automated bug localisation provides little benefit (Kochhar *et al.*, 2014). This would indeed be the case if almost all files mentioned in a bug report were true positives. As Table 3.1 shows, most bug reports only affect a very small number of files, and yet they may mention many more files, especially if they contain stack traces (Table 4.4). In Section 4.4.1.5 I gave two examples, both placed in the top-5 by ConCodeSe: one bug report mentioning 9 files, all irrelevant, and another mentioning 14, of which only one was relevant. Developers are interested in finding out focus points and how files relate to each other, especially when working in unfamiliar applications (Sillito *et al.*, 2008). Automated bug localisation can help developers to evaluate the information when looking at the files mentioned in bug reports, and even suggest relevant files not mentioned in the reports.

I offer some reasons why my approach works better.

First, other approaches combine scores using weight factors that are fixed. I instead take always the best of several ranks for each file. In this way, my approach is not a priori fixing for each file whether the lexical scoring or the probabilistic VSM score should take precedence. I also make sure that stemming and comments are only taken into account for files where it matters. The use of the best of 8 scores is likely the reason for improving the key MRR metric across all projects.

Second, I leverage structure further than other approaches. Like BLUiR, I distinguish the bug report's summary and description, but whereas BLUiR treats each bug report field in exactly the same way (both are scored by Indri against the parts of a file) I treat each field differently, through the key position and stack trace scoring.

Third, my approach simulates the selective way developers scan bug reports better than other approaches. It has been argued that developers may not necessarily need automated bug localisation tools when file names are present in the bug reports (Kochhar *et al.*, 2014) because they may be able to select which information in the bug report is more relevant, based on the structure of the sentences and their understanding of the reported bug (Wang *et al.*, 2015). ConCodeSe first looks at certain positions of the bug report and then, if unsuccessful, uses all terms in the bug report, stopping the scoring when a full file name match is found. However, the same cannot be said for most automated tools: they always score a file against all the bug report terms, which may deteriorate the performance if a file has more matching terms with the bug report, as it is scored higher and falsely ranked as more relevant (Moreno *et al.*, 2014).

Fourth, my approach addresses both the developer nature and the descriptive nature of bug reports, which I observed in the analysis of the bug reports for these projects and in particular for Pillar1 and Pillar2 (Dilshener and Wermelinger, 2011). Bug reports of a developer nature tend to include technical details, like stack traces and class or method names, whereas bug reports of a descriptive nature tend to contain user domain vocabulary. By leveraging file (class) names and stack traces when they occur in bug reports, and by otherwise falling back to VSM and a base lexical similarity scoring (Chapter 3, Sub-section 3.3.2), I cater for both types of bug reports. As the non-bold rows of Table 4.9 show, the fall-back scoring alone outperforms all other approaches in terms of MAP and MRR (except AmaLgam's MAP score for AspectJ).

Although LOBSTER doesn't use historical information either, it was a study on the value of stack traces over VSM, and thus only processes a subset of developer-type bug reports, those with stack traces.

To sum up, I treat each bug report and file individually, using the summary, stack trace, stemming, comments and file names only when available and relevant, i.e. when they improve the ranking. This enables my approach to deal with very terse bug reports and with bug reports that don't mention files.

As for the efficiency of my approach, creating the corpus from the application's source code and bug reports takes, on a machine with a 3GHz i3 dual-core processor and 8GB RAM, 3 hours for Eclipse (the largest project, see Table 2.1). Ranking (8 times!) its 12,863 files for all 3075 bug reports takes about 1.5 hours, i.e. on average almost 2 seconds per bug report. I consider this to be acceptable since my tool is a proof of concept and developers locate one bug report at a time.

4.5.1 Threats to Validity

During creation of the searchable corpus, I relied on regular expressions for extracting the stack trace from the bug report descriptions. It is possible that other techniques may produce different results. Also, the queries (i.e. search terms) in my study were taken directly from bug reports. The developers may formulate their queries differently when locating bugs in an IDE and that the use of different queries with vocabularies more in line with the source code would yield better results. However, I catered for all these threats to construct validity by using the bug report summaries and descriptions as queries instead of manually formulated queries avoided the introduction of bias on my behalf.

I addressed internal validity by comparing the search performance of ConCodeSe like for like (i.e., using the same datasets and the same criteria) with eight existing bug localisation approaches (Zhou *et al.*, 2012a; Saha *et al.*, 2013; Wong *et al.*, 2014; Moreno *et al.*, 2014; Wang and Lo, 2014; Ye *et al.*, 2014; Youm *et al.*, 2015; Rahman *et al.*, 2015) as well as assessing the contribution of the off-the-shelf Lucene library's VSM. Therefore, the improvement in results can only be due to my approach. I also used fixed values to score the files, obtained by manually tuning the scoring on AspectJ and SWT. I confirmed the rationale behind those values (namely, distinguish certain positions and assign much higher scores than base term matching) led to the best results, by trying out scoring variations.

I catered for the conclusion validity by using a non-parametric Wilcoxon matched pairs statistical test since no assumptions were made about the distribution of the results. This test is quite robust and has been extensively used in the past to conduct similar analyses (Schröter *et al.*, 2010; Moreno *et al.*, 2014). Based on the values obtained as shown in Table 4.15, I conclude that on average ConCodeSe locates significantly ($p \leq 0.05$) more relevant source files in the top-N, which confirms that the improvement in locating relevant files for a

Table 4.15: Wilcoxon test comparison for top-5 and top-10 performance of BugLocator and BRTracer vs ConCodeSe

Statistics	**BugLocator**		**BRTracer**	
	Top-1	**Top-10**	**Top-1**	**Top-10**
Z-value	-4.2686	-3.2351	-2.73934	-3.9656
W-value	254	576	487	543
p-value	0.047504	0.0326	0.043432	0.0067

bug report in the top-N position by my tool over the state of the art is significant.

The characteristics of the projects (e.g. the domain, the identifier naming conventions, and the way comments and bug reports are written, including the positions where file names occur) are a threat to external validity. I reduced this threat by repeating the search experiments with different applications, developed independently of each other, except for SWT and Eclipse. Although ConCodeSe only works on Java projects for the purpose of literature comparison, the principles (take the best of various rankings, score file names occurring in the bug report higher than other terms, look for file names in particular positions of the summary and of the stack trace if it exists) are applicable to other object-oriented programming languages.

4.6 Concluding Remarks

This Chapter contributes a novel algorithm that, given a bug report and the application's source code files, uses a combination of lexical and structural information to suggest, in a ranked order, files that may have to be changed to resolve the reported bug. The algorithm extends my previously proposed approach introduced in Chapter 3 by considering words in certain positions of the bug report summary and of the stack trace (if available in a bug report) as well as source code comments, stemming, and a combination of both independently, to derive the best rank for each file.

I compared the results to eight existing approaches, using their 5 evaluation criteria and their datasets (4626 bug reports, from 6 OSS applications), to which I added 39 bug reports from 2 other applications. I found that my approach improved the ranking of the affected files, increasing in a statistical significant way the percentage of bug reports for which a relevant file is placed among the top-1, 5, 10, which is respectively 44%, 69% and 76% , on average. This is an improvement of 23%, 16% and 11% respectively over the best performing current state-of-the-art tool. I also improved, in certain cases substantially, the mean reciprocal rank value for all eight applications evaluated, thereby reducing the number of files to inspect before finding a relevant file.

My approach not only outperforms other approaches, it does so in a simpler, faster, and more general way: it uses the least artefacts necessary (one bug report and the source codebase it was reported on), not requiring past information like version history or similar bug reports that have been closed, nor the tuning of any weight factors to combine scores, nor the use of machine learning.

Like previous studies, my approach shows that it is challenging to find the files affected by a bug report: in spite of my improvements, for larger projects, e.g. Eclipse, 30% of bugs are not located among the top-10 files. Adding history-based heuristics and learning-to-rank, as proposed by other approaches, will certainly further improve the bug location performance. In order to help develop new search approaches, I offer in an online companion to this thesis, the full results as an improved baseline for further bug localisation research [3].

Since I used only one industrial application (Pillar2) to evaluate my approach and to gain a broader understanding of how it may perform with other commercial applications, I conducted user studies in 3 different companies. In the following Chapter, I report on the performance of my tool in industrial settings and how the ranked result list sorted in the order of relevance is perceived by professional developers.

[3] http://www.concodese.com

Chapter 5

User studies

In the previous Chapter, I evaluated my approach with a range of OSS projects and showed it outperformed current state-of-the-art tools in a simpler, faster, and more general way that does not require history. To investigate the generalisability of my approach, I conducted user studies in three different companies with professional developers. The first aim of the study was to demonstrate the applicability of my approach in industrial environments. Commercial applications and bug reports may have different characteristics to the OSS applications investigated in Chapter 4, thus impacting the performance of my approach.

Previous user studies of reveal that large search results returned by the integrated development environment (IDE) tools cause developers to analyse several files before performing bug-fixing tasks (Sillito *et al.*, 2008; Starke *et al.*, 2009). Thus the second aim of the study was to know how developers perceive the search results of ConCodeSe, which presents a ranked list of candidate source code files that may be relevant for a bug report at hand during software maintenance.

In this Chapter, Section 5.1 gives an introduction by describing the challenges faced by developers when searching for relevant source code files as identified by current studies. Subsequently, I describe my study design in Section 5.2 and present the results of the study in Section 5.3. Finally, I evaluate the findings in Section 5.4, and conclude with remarks in Section 5.5.

5.1 Background

Sillito *et al.* (2008) conducted two different studies, one in a laboratory setting with 9 developers who were new to the code base and the other in an industrial setting with 16 developers who were already working with the code base. In both studies developers were observed performing change tasks to multiple source files within a fairly complex code base using modern tools. The findings reveal that text-based searches available in current IDEs are inadequate because they require search terms to be precisely specified, otherwise irrelevant or no results are returned. The study claims that developers repeatedly perform discovery tasks in a trial and /error mode, which causes additional effort and often results in several failed attempts.

Starke *et al.* (2009) performed a study with 10 developers to find out how developers decide what to search for and what is relevant for the maintenance change task at hand. Participants were randomly assigned one of 2 closed bug descriptions selected from the Sub-Eclipse tool's issue repository and instructed to carry out search tasks in the Eclipse IDE using the available search tools. The findings highlight that formulating a search query is the most challenging task for the developers since Eclipse search tools require the search terms to be precisely specified otherwise no relevant results are returned. The authors also state that when many search results are returned, developers tend to lose confidence in the query and decide to search again rather than investigate what was returned. They propose future research on tool support for the developers to provide more contextual information and to present results in a ranked order.

Both studies (Sillito *et al.*, 2008; Starke *et al.*, 2009) articulate that providing developers with automated code search tools that presents results in a ranked order would be of great benefit in performing their daily tasks. In addition, Xia *et al.* (2016) and Kochhar *et al.* (2016) found top-5 to be the magic number of results that developers deem acceptable to inspect. Furthermore, Kochhar *et al.* (2016) observed that almost all developers 'refuse' to inspect more than 10 items. None of those studies used IR-based bug localisation. Thus the second aim of my study was to see how developers perceive and use the ranked search results of ConCodeSe.

After obtaining better results than the state-of-the-art literature on bug localisation using open-source benchmark projects, I was interested in finding how my approach performs in industrial environments with proprietary business applications. Hence I asked my research question in Chapter 1 as follows.

RQ3: *How does my approach perform with industrial applications and does it benefit developers by presenting the search results in ranked order of relevance?*

In order to not require users to modify ConCodeSe to interface with their software repositories and issue trackers, I implemented a simple front-end (GUI) panel to my tool for users to paste the summary and description of a bug report and search for the candidate source code files. The results are displayed in the ranked order of relevance from most to least likely.

5.2 Study Design

Based on the works of Sillito *et al.* and Starke *et al.*, I designed a study with professional software developers with their a priori consent on the followings:

1. Data collected:
   (a) How useful was the ranked list?
   (b) How accurate was the ranking?
   (c) How much were you confident on the list?
   (d) How intuitive was it compared to an IDE search tool?
   (e) What did work best?
2. Data collection method;
   (a) Pre session interview
   (b) Each search session: choosing the bug reports, running the tool, evaluating the results.
   (c) Post session interview

I contacted five different companies where I formerly worked as a freelance software developer, explaining the study. I sent each company an information leaflet explaining that I was looking for participants to take part in a study where my tool will be used to search source code files of a software application with descriptions available in a bug report document. The leaflet informed interested parties how the study unfold:

1. I would conduct a 30-45min pre-session interview to explain how to use ConCodeSe.
2. Afterwards, the participants would be required to try the tool for 7–10 business days in their own time and document their experience on the relevance of the suggested source files for the search tasks performed.

Table 5.1: Business nature of the companies and professional experience of the participants involved in the user study

Company	Business Nature	Study Dates (2015)	Participant Id	Years of Experience	Years of IDE Experience	time spent on maintenance
U	Finance	29.06 - 03.07	1	20	10	30%
			2	15	10	70%
S	Logistic	06.07 - 04.08	3	9	9	20%
A	Software Services	05.06 - 16.10	4	11	11	40%

3. Finally, at the end of the trial period a post-session interview lasting for 30–45min would be conducted to collect details on their usage experience.

The leaflet further explained that the participation would be treated in strict confidence in accordance with the UK Data Protection Act and no personal information could be passed to anyone outside the research team. It also informed that I would aim to publish the findings from the study, but no individual would be identifiable. Participants were allowed to answer questions with "not-applicable" if they did not intend to provide an answer during both interview sessions.

Out of the five contacted companies, only three agreed to participate and two of them agreed to have the post-session interview session recorded as video. Upon receiving the participation agreement from each company, I obtained written consent from each participant prior to the study. Table 5.1 shows the participant profile information at each company and Table 5.2 details the artefacts used in the study: the size of the industrial applications is comparable to the medium-sized OSS projects (Table 2.1), All participants were professional software developers and did not require compensation since they were recruited on volunteering basis.

Although I had worked for the three companies in the past, I did not have prior knowledge of the applications used in the case study nor had I previous contact with the participants. I went to each company, and presented ConCodeSe and the aim of the study, which was to identify the benefits provided by a ranked list of candidate source code files that may be

Table 5.2: Project artefact details used in the study at the participating companies

Company	Application in production since	# of source code class files	# of bug reports used
U	2014	2840	10
S	2009	2240	10
A	2013	4560	10

relevant for a bug report at hand during software maintenance. Subsequently, I conducted a pre-session interview with the developers to collect information about their experience and their thought process on how they performed their tasks during daily work. One of the intentions of this pre-session interview was to make developers aware of their working habits so that they could document their experience with my tool more accurately.

5.3 Results

I present the information collected during the pre- and post-interviews. Throughout both sessions developers referred to a source code file as *class* and to a bug report as *defect*.

5.3.1 Pre-session Interview Findings

Based on the observations made by Starke *et al.*, I designed the pre-interview questions in order to obtain a summative profile of developers and their working context. Below I list the questions asked and the general answers given to each of these questions.

1. ***How do you go on about solving defects?*** All developers indicated that they read bug report descriptions and log files looking for what to fix based on experience. They also try to reproduce the bug to debug the execution steps and look at the source code to see what is wrong. Only one developer indicated that he compares the version of the application where the bug has occurred against a version where the bug hasn't been reported yet to detect any changes that might have caused the reported bug.

2. ***How would you go on about solving a defect in an unfamiliar application?*** In general developers responded that they search code by using some words (e.g. nouns and action words) from the bug description to pick a starting point. They also read the user guide to understand the behaviour of the application and technical guide to understand the architecture. One developer indicated that he would attempt to simulate the reported behaviour if he could understand the scenario described in the bug report and some test data is provided.

3. ***What about when you cannot reproduce or haven't got a running system?*** At least to get an idea of a starting point, developers look for some hints in class and method names. One developer stated that he would include logging statements in certain classes

to print out trace information and collect details of execution in production so that he could see what was happening during run time.

4. ***In order to find starting points to investigate further, what kind of search tools do you use?*** Although all developers said that they use search functions available in IDEs, e.g. full text, references, inheritance chain and call hierarchies, one said that he prefers to use the Mac OS Spotlight desktop search because in addition to source files, it indexes other available artefacts like the configuration files, GUI files (e.g. html, jsp and jsf) as well as the documentation of the application.

5. ***How do you evaluate a call hierarchy?*** Developers explained that they would start by performing a '***reference***' search of a class and browse through the results. One said that "*I also look to see if method and variable names that are surrounding the search words also make sense for the bug description that I am involved with*".

6. ***What do you consider as being important, callers of a class or the called classes of a class?*** Each reply started with "*That depends on...*". It seems that each participant has a different way of assessing the importance of the call hierarchy. For some developers the importance is based on the bug description, e.g. if the bug description indicated that some back-end modules are the culprit, then they would look to see where the control flows are coming from, while for others, it is based on architecture, e.g. if a certain class is calling many others then they would consider this to be a bad architecture and ignore the caller.

7. ***How do you decide which classes to investigate further, i.e. open to look inside, and which to skip?*** Almost all developers answered first by saying "*gut feeling*" and then went on to describe that they quickly skim through the results list and look for clues on package or file names to determine whether it makes sense to further investigate the file contents or not.

8. ***Do you consider descriptive clues?*** Once again developers replied that they rely on their experience of the project to relate conceptual words found in the bug reports to file names.

9. ***When do you decide to stop going through the path and start from the beginning?*** All participants indicated that after browsing through the search results looking

at file names, they may open 3 or 4 files to skim through their content and when no clues were detected, they would decide to start a new search with different words.

10. ***What kind of heuristics do you use when considering concepts implemented by neighbouring classes?*** In general, developers indicated that they look at the context of a source file within the bug that they are working on, i.e. relevance of a file based on the bug report vocabulary. One developer said that project vocabulary sometimes causes ambiguity because in the application he works with, certain file names contain the word *Exception*, which refers to a business exception rule and not to an application exception, i.e. error, condition.

The pre-session interview answers confirmed the challenges highlighted by previous studies (Sillito *et al.*, 2008; Starke *et al.*, 2009). I asked developers to try out ConCodeSe by downloading, installing and performing search tasks in their own time. I decided to perform an uncontrolled study because I felt that this would provide a more realistic environment and also allow developers to have adequate time to utilise my tool without negatively impacting their daily workload.

I asked them to collect screen shots showing the information they entered as search text and the results listed. I instructed them to use closed bug reports for search query where the affected files were also documented so that they can prove whether the results contained the relevant files or not.

I arranged to meet them in 7–10 business days for a 30–45min post-session interview to gather their experiences and to inspect the screen shots.

5.3.2 Post-session Findings

Below, I list the questions asked during the post-interview sessions and the general answers given to each of these questions.

1. ***How did you go on using the tool?*** All indicated that they first used a few bug reports with which they previously worked on to become familiar with my tool and its performance. Afterwards they have randomly chosen closed bug reports that they had not previously worked with, performed search by using the words found in the summary and description fields of the bug reports and then compared the search results against the files identified as affected in the issue-tracking tool. One developer

said, "*I performed search tasks by selecting a few words, i.e. 2–3, from the bug report, which I knew would lead to relevant results. However this did not work very well so I have gradually increased the search terms with additional ones found in the bug report descriptions. This provided more satisfactory results and the relevant classes started appearing in the result list*".

2. ***Did the tool suggest relevant classes in the ranked list?*** Inspecting the screenshots of the results, I found that in Company-U, for 8 out of 10 bug reports at least one file was in the top-10 and for the remaining 2 no file made it to top-10. In Company-S, for 9 out of 10 bug reports the tool ranked at least one affected file at top-1. Developer in Company-A said that "*The relevant file was always ranked among the top-3. I never needed to investigate files which were ranked beyond top-3*".

3. ***Were there relevant classes in the result list, which you might not have thought of without the tool?*** All participants replied with '***yes***'. One developer said that "*I have also opened up the files listed at top-2 and top-3, despite that they were not changed as part of the bug report at hand. However looking at those files would have also led me to the relevant one, assuming that the relevant file was not in the result list*". Another indicated that "*a bug report description had a file name which got ranked at top-1 but did not get changed. However it was important to consider that class during bug fixing*".

4. ***Did the ranked list provide clues to formulate search descriptions differently?*** Developers indicated that most of the time they had a clue to what they were looking for but were not always certain. However, seeing file names with a rank allowed them to consider those files with a degree of importance as reflected by the ranking. In case of one defect, the description caused a lot of noise and resulted in the top 5 files being irrelevant but 3 relevant files were placed in the top-10.

5. ***What did you add to or remove from the description to enhance the search?*** Developers indicated that the search led to better results when the query included possible file names, exceptions and stack trace information. However, in one case ignoring the description and searching only with the summary resulted in 4 out of 8 relevant files to be ranked at top-5 and one file between top-5 and top-10. One developer indicated

that in one case despite changing search words, the relevant file was still not found because the bug report did not provide any clues at all.

6. ***What would you consider to reduce false positives in search results?*** In general, all participants suggested that project specific vocabulary mapping should be used to cover cases when concepts in file names cause misleading results. For example, batch jobs are named as Controller, so when a bug report describes a batch job without using the word *controller*, then those classes are not found if there are no comments indicating additional clues. In such case, only 5 out of 18 affected files were found.

7. ***How comfortable were you with the ranked list?*** All developers indicated that the tool was easy to use and after performing 3–4 searches to become familiar with the tool, they were satisfied with the results since the files at the first 5 positions were most often relevant ones. Despite their comfort with the tool, all participants indicated that if a bug report description contained fewer technical details, e.g. fewer file names, and more descriptive information, e.g. test scenario steps, the tool was not useful. In fact they felt that in such cases any tool would fail to suggest the relevant file(s).

8. ***Were you able to get to the other relevant classes based on the suggested ones that were relevant?*** All participants expressed that the ranked list helped them to consider other focal points and gave them a feel for what else to look for that might be relevant. One said that "*Most of the cases I browsed through the results and opened only the top-1 and top-2 files to see if they were the relevant ones or to see if they can lead me to the file that may be more appropriate*".

9. ***Would you consider such a tool to support you in the current challenges you have?*** In general, all indicated that the ranked list would benefit anyone who is new to a project since it could guide novice developer to the parts of the application when searching for files that may be relevant to solve a defect. This would in turn allow the novice developers to rapidly build up application knowledge. One said "*The tool would definitely speed up the learning curve for a new team member who is not familiar with the architecture and code structure of our project. It would save him a lot of time during the first few weeks. After that due to the small size of the project (12000 LOC) it would not provide much significance because the developer would become familiar with the code anyway*". I was told that as a research product my prototype tool was very stable. They

experienced no failures, i.e. crashes or error exceptions. One developer said that even when running the tool in a virtual machine (VM) environment, suspending and then resuming the VM, my tool continued to function.

To my final 10th question, "***What would you suggest and recommend for future improvements?***", I have received a lot of valuable suggestions. First of all, I was told by all developers that the biggest help would be to write better bug descriptions and to introduce a defect template to solicitate this. They noticed that defect descriptions containing test steps entered by 1st level support is noise. For example, in case of one defect, the relevant file is ranked in top-5 and its test file in top-1 so the relevant files were obscured by test files.

I was also suggested to include in the search the content of configuration files, e.g. config.xml, DB scripts and GUI files. One developer noticed that the words on the first search field (bug report summary) are given more importance. He wished that the second field (bug report description) is treated with the same importance as the first search field.

In addition I have received several cosmetic suggestions for presenting the results and interacting in search fields, like proposing keywords, e.g. auto complete based on terms found in the source code. Participants also felt that it would be helpful to display, next to the ranked files, the words in those files that match the bug report so that the user can determine whether it really makes sense to investigate the content of the file or not. Interestingly one developer said that sometimes he did not consider the ranking as an important factor and suggested to group files into packages, based on words matching the package name and then rank the files within that group.

Most of the suggestions concern the developer interface and are outside the scope of ConCodeSe. I intend to investigate whether configuration files can be matched to bug reports in future work.

5.4 Evaluation of the Results

Modern IDE search tools offer limited lexical similarity function during a search. The developers are required to specify search words precisely in order to obtain accurate results, which may require developers to be familiar with the terminology of the application and the domain. The success of modern IDE search tools depend on the clarity of the search terms otherwise the results may contain many false positives. To compensate for these weaknesses, developers choose to specify on average 3 words (see post-session interview Answer 1) when searching for relevant files in IDEs instead of using all the words available in a bug report. Since current IDE search tools deprive developers from the advantage of utilising the full information available in bug reports, developers may search outside of the IDE (see pre-session interview Answer 4).

Furthermore, in current IDEs, the search results are not displayed in a ranked order of relevance causing developers to go through several irrelevant files before finding relevant ones as entry points for performing bug-fixing tasks. Since developers are faced with the challenge of manually analysing a possibly long list of files, they usually tend to quickly browse through the results and decide on its accuracy based on their gut feeling as revealed during my pre-session interview. They also prefer to perform a new search query using different words rather than opening some files to investigate their content. These repetitive search tasks cost additional effort and add burden on the productivity of the developers causing them to lose focus and introduce errors due to fatigue or lack of application knowledge.

I set out to explore whether my ranking approach would benefit developers. Based on the post-session interview answers provided by developers working in different industrial environments with different applications, I confirm that developers welcomed the ranked result list and stated that since most of the relevant files were positioned in the top-5, they were able to avoid the error prone tasks of browsing long result lists and repetitive search queries by focusing on the top-5 portion of the search results.

I was interested in finding out whether the ranked list would point developers to other files that might be of importance but were not initially thought to be relevant. After trying out my tool, at the post-session interview, the developers said that the result list contained other relevant files that they would not have thought of on their own without my tool (see post-session interview Answer 3). Developers stated that those additional and not thought of files would not have appeared in the result of the IDE search tool they use because those files do not contain the search terms. However my tool was able to localise those files because my approach combines VSM probabilistic scoring with lexical similarity match. Hence I advocate strongly for making a VSM based IR model as part of any modern search tool.

Finally, I wanted to see whether my tool, which leverages the textual information available in bug reports, encourages developers to use the full description of a bug report when formulating search queries. During the pre-session interview, developers told us that they use their gut feeling and experience when selecting words to use on the search query. At the post-session interview, I was told by developers that incrementing the search words with additional ones from the bug reports improved the results.

In software projects, a developer may get assistance from other team members or expert users when selecting the initial entry points to perform the assigned maintenance tasks. I was told that my tool complements this by providing a more sophisticated search during software maintenance. Thus the bug report vocabulary can be seen as the assistance provided by the expert team members and the file names in the search results can be seen as the initial entry points to investigate additional relevant files.

Furthermore, I was told that search results still depend on the quality of bug descriptions. In case of tersely described bug reports, even experienced developers find it challenging to search for relevant files. In addition, I found that bug reports can be of (1) developer nature with technical details, i.e. references to code files and stack traces or (2) descriptive nature with business terminology i.e. use of test case scenarios. Since bug report documents may come from a group of people who are unfamiliar with the vocabulary used in the source code, I propose that bug report descriptions contain a section for describing the relevant domain vocabulary. For example a list of domain terms implemented by an application can be semi-automatically extracted and imported into the bug report management tool. Subsequently, when creating a bug report, the user may choose from the list of relevant domain terms or the tool may intelligently suggest the terms for selection.

From all these results, I can answer my research question affirmatively: in case of business applications, my tool also achieves to place at least one file into top-10 for 88% of the bug reports (see post-session interview Answer 2 and Chapter 4, Figure 4.3: 83% in Pillar2, 80% in Company U, 90% in S and 100% in A) into top-10. My study also confirms that users will focus on the first 10 suggestions and that therefore presenting the search results ranked in the order of relevance for the task at hand benefits developers.

5.4.1 Threats to Validity

The queries (i.e. search terms) in my studies performed in Chapter 3 and Chapter 4 were taken directly from the bug reports. This threat to construct validity is addressed by the user studies in this Chapter, which showed that the developers formulate their queries differently when locating bugs in an IDE and that the use of different queries with vocabularies more in line with the source code would yield better results.

In the user study, the bug localisation was uncontrolled, to avoid disturbing the daily activities of the developers. This may be a potential threat to construct validity. I partially catered for this threat by asking developers to make screen shots of the results, which they showed during the post-session interview.

Bug location techniques poses a difficult challenge because of the inherent uncertainty and subjectivity. One programmer may think a source file is relevant to a feature while another may not. In my study I catered for this threat to internal validity by asking developers to use closed bugs that they did not work on previously and instructed them to document their results to avoid any bias.

The small size of the user study (4 participants from 3 companies) and the characteristics of the projects (e.g. the domain, the identifier naming conventions, and the way comments and bug reports are written, including the positions where file names occur) are a threat to external validity. I reduced threat to external validity by repeating the search experiments with developers at three different companies evaluating my tool using industrial applications, developed independently of each other.

5.5 Concluding Remarks

Chapters 3 and 4 have introduced a novel algorithm that, given a bug report and the application's source code files, uses a combination of lexical and structural information to suggest, in a ranked order, files that may have to be changed to implement the bug report. The algorithm considers words in certain positions of the bug report summary and of the stack trace (if available in a bug report) as well as source code comments, stemming, and a combination of both independently, to derive the best rank for each file.

In this Chapter, I evaluated the algorithm on four very different industrial applications (Pillar2 in Chapter 4 and from the three user studies), and placed at least one file in the top-10 for 83% of bug reports on average, thus confirming the applicability of my approach also in commercial environments.

Text-based searches available in current integrated development environments (IDE) are inadequate because they require search terms to be precisely specified otherwise irrelevant or no results are returned (Sillito *et al.*, 2008; Starke *et al.*, 2009). Developers stated that since most of the relevant files were positioned in the top-5, they were able to avoid the error prone tasks of browsing long result lists and performing repetitive search queries. This confirms that presenting the search results ranked in the order of relevance for the task at hand aids developers during maintenance.

Chapter 6

Conclusion and Future Directions

In Chapter 1, I argued that current state-of-the-art IR approaches in bug localisation rely on project history, in particular previously fixed bugs and previous versions of the source code. However, existing studies (Nichols, 2010; Wang and Lo, 2014) show that considering similar bug reports up to 14 days and version history between 15—20 days does not add any benefit to the use of IR alone. Furthermore, the techniques can only be used where great deal of maintenance history is available but the same studies also show that considering history up to 50 days deteriorates the performance.

In addition, Bettenburg *et al.* (2008) argued that a bug report may contain a readily identifiable number of elements including stack traces, code fragments, patches and recreation steps each of which should be treated separately. The previous studies also show that many bug reports contain the file names that need to be fixed (Saha *et al.*, 2013) and that the bug reports have more terms in common with the affected files, which are present in the names of those affected files (Moreno *et al.*, 2014).

In this Chapter, I revisit my hypothesis and describe how it was addressed by the arising research questions in Section 6.1. Subsequently, I highlight the contributions, and give recommendations to practitioners and future research in Section 6.2. Finally, I end this Chapter thus my thesis with concluding remarks in Section 6.3.

6.1 How the Research Problem is Addressed

Bettenburg *et al.* (2008) disagree with the treatment of a bug report as a single piece of text document and source code files as one whole unit by existing approaches (Poshyvanyk *et al.*, 2007; Ye *et al.*, 2014; Kevic and Fritz, 2014; Abebe *et al.*, 2009). Furthermore, the existing approaches treat comments as part of the vocabulary extracted from the source code, but since comments are sublanguage of English, they may deteriorate the performance of the search results due to their imperfect nature, i.e. terse grammar (Etzkorn *et al.*, 2001; Arnaoudova *et al.*, 2013).

The hypothesis I investigated is that *superior results can be achieved without drawing on past history by utilising only the information, i.e. file names, available in the current bug report and considering source code comments, stemming, and a combination of both independently, to derive the best rank for each file.*

In this section I summarise how I addressed my research hypothesis as follows.

6.1.1 Relating Domain Concepts and Vocabulary

Motivated to address my hypothesis, I first intended to discover whether vocabulary alone provides a good enough leverage for maintenance. I was determined to explore whether (1) the source code identifier names properly reflect the domain concepts in developers' minds and (2) identifier names can be efficiently searched for concepts to find the relevant files for implementing a given bug report. Thus asked my first research question as follows.

RQ1: Do project artefacts share domain concepts and vocabulary that may aid code comprehension when searching to find the relevant files during software maintenance?

To address RQ1, in Chapter 3 I undertook a preliminary investigation of eight applications and compared the vocabularies of project artefacts (i.e. text documentation, bug reports and source code) by asking three sub-research questions as follows.

RQ1.1: How does the degree of frequency among the common concepts correlate across the project artefacts? I investigated whether independent applications share key domain concepts and how the shared concepts correlate across the project artefacts. The correlation was computed pairwise between artefacts, over the instances of the concepts common to both artefacts, i.e. between the bug reports and the user guide, then between the bug reports and the source code, and finally between the user guide and the source code. I argued that identifying any agreement between the artefacts may lead to efficient software maintenance.

I found that while each concept occurred in at least one artefact, only a subset of the concepts occurred in all three artefacts (Sub-section 3.4.1). Also the three artefacts explicitly include all the domain concepts, however only a handful of the concepts occur both in the code and in the documentation, which may point to potential inefficiencies during maintenance.

On the other hand, I found that the common concepts correlate well in terms of relative frequency, taken as proxy for importance, i.e. the more important concepts in the user guide tend to be the more important ones in the code (Sub-section 3.4.1.1). This good conceptual alignment between documentation and implementation may ease maintenance, especially for new developers.

RQ1.2: What is the vocabulary similarity beyond the domain concepts, which may contribute towards code comprehension? Subsequently, I aimed at discovering any common terms, i.e. words extracted from the source code identifiers, other then the domain concepts between the source code files of different applications. I found that the application source files share common terms beyond the domain concepts (Sub-section 3.4.2) and manual analysis revealed that those common terms are the ones generally used by the developers during programming, e.g. CONFIG, version, read, update, etc., which may contribute towards code comprehension.

Additionally, I investigated overall vocabulary of all the identifiers because despite an overlap in the terms, developers of one application might combine them in completely different ways when creating the identifiers that may confuse other developers. Indeed, all applications combine the words differently: only a small percentage of common identifiers exists between the applications.

Furthermore, I discovered that Eclipse source code also contains the source files of two smaller applications, SWT and AspectJ. Searching the source code of Eclipse using the actual words extracted from the bug reports of the smaller sub-set revealed that given a large search space with many source files, e.g. 12,863 in Eclipse, my approach successfully retrieves the relevant files for the bug reports of a smaller sub-set, e.g. SWT with 484 files, within the same search space without retrieving many false positives (Sub-section 3.4.3.3). This indicates that like finding a needle in a hay stack, my scoring performs independently of any artefact's size.

RQ1.3: How can the vocabulary be leveraged when searching for concepts to find the relevant files for implementing bug reports? Finally, I investigated whether searching with domain concepts only is adequate enough to find relevant files for a given bug report. For this, I used my novel IR approach that I introduced in Sub-section 3.3.2, which directly scores each current file against the given bug report by assigning a score to a source file based on where the search terms occur in the source code file, i.e. class file names or identifiers.

The conceptual overlap and correlation between the bug reports and the other two artefacts revealed that searching with only the domain concepts referred by a bug report, achieved very poor MAP and MRR results (Sub-section 3.4.3). However, both can be improved by using all of the vocabulary found in the bug report and achieved on average 54% recall when positioning the relevant files into top-10 (Sub-section 3.4.3.1). Such a simple, and efficient technique can drastically reduce the false positives a developer has to go trough to find the files affected by a bug report, thus confirming that bug reports contain information that may boost the performance when properly evaluated.

In the data sets analysed on average only 3 files were listed as being changed but multiple files may implement a concept (Section 2.1). This suggests that bug reports are for a unit-of-work and searching for the relevant files using concepts find all the files implementing that concept. Nevertheless a bug report may require only a subset of those files to be changed. Hence in the context of a bug report this may lead to many false positives as seen by the poor MAP values obtained during concept-only search (Sub-section 3.4.3).

Based on the answers to all sub-research questions, I answer RQ1 affirmatively: project artefacts share domain concepts and vocabulary that may aid code comprehension when searching the relevant files during software maintenance.

6.1.2 Locating Bugs Without Looking Back

Answering RQ1 in Chapter 3 revealed that artefacts explicitly reflect the domain concepts and that paired artefacts have a good conceptual alignment, which should help maintenance when searching for files affected by given bug report. However, the studies I conducted to address RQ1.1–RQ1.3 showed that despite good vocabulary coverage, it is challenging to find the files referred by a bug report. To improve the suggestion of relevant source files during bug localisation, I argued that heuristics based on information available within the context of bug reports, e.g. words in certain key positions, be developed.

Current state-of-the-art approaches for Java programs (Zhou *et al.*, 2012a; Wong *et al.*, 2014; Saha *et al.*, 2013; Wang and Lo, 2014; Ye *et al.*, 2014) rely on project history to improve the suggestion of relevant source files. In particular they use similar bug reports and recently modified files. However, the observed improvements using the history information have been small (Sub-section 2.3.4). I thus argued that file names mentioned in the bug report descriptions can replace the contribution of historical information in achieving comparable performance and asked my second research question as follows.

RQ2: Can the occurrence of file names in bug reports be leveraged to replace project history and similar bug reports to achieve improved IR-based bug localisation?

To address RQ2 in Chapter 4, I extended my approach introduced in Sub-section 3.3.2 to consider words in certain positions of the bug report summary and of the stack trace (if available in a bug report).

Results showed that my approach located on average for 64% of the bug reports a relevant file in the top-10 by just assigning a high score to file names in the bug report summary (see row ***'KP only'*** in Table 4.6, Sub-section 4.4.1.1), confirming the studies cited in the introduction that found file names mentioned in a large percentage of bug reports (Saha *et al.*, 2013; Schröter *et al.*, 2010). In addition evaluating the stack trace further increased the number of relevant files placed in all top-N categories indicating that assigning a higher score to file names found in stack trace improves the performance of the results (Sub-section 4.4.1.2), which is also in line with the findings of previous studies (Schröter *et al.*, 2010; Moreno *et al.*, 2014; Wong *et al.*, 2014).

I compared the performance of my extended approach against the existing ones, i.e. BugLocator (Zhou *et al.*, 2012a), BRTracer (Wong *et al.*, 2014), BLUiR (Saha *et al.*, 2013), AmaLgam (Wang and Lo, 2014), LearnToRank (Ye *et al.*, 2014), BLIA Youm *et al.* (2015), and Rahman *et al.* (2015), on the same datasets, using the same performance metrics, and outperformed them in the majority of cases without considering any historical information. In particular my approach succeeded in placing an affected file among the top-1, top-5 and top-10 files on average for 44%, 69% and 76% of bug reports, improving the best performing current state-of-the-art tool by 23%, 16% and 11% respectively (Sub-section 4.4.1.5).

Additionally, I compared the recall[1] performance of ConCodeSe against BugLocator and BRTracer. The results were significantly superior to the other two current state-of-the-art tools (Sub-section 4.4.1.6) providing further evidence that even when a different metric (i.e. recall instead of MAP and MRR) is used to measure the performance, my approach still outperforms the existing ones.

[1] the number of relevant files placed in the top-N out of all the effected relevant files.

Furthermore in Chapter 4, I investigated the answers to the following two questions.

RQ2.1: ***What is the contribution of using similar bug reports to the results performance in other tools compared to my approach, which does not draw on past history?*** I looked more closely at the contribution of past history, in particular of considering similar bug reports, an approach introduced by BugLocator and adopted by others. I compared the results of BugLocator and BRTracer using SimiScore (the similar bug reports score), and the results of BLUiR according to the literature, showing that SimiScore's contribution is not as high as suggested.

Through my experiments, I found that ConCodeSe achieves on average superior results to BugLocator and BRTracer in terms of MAP and MRR respectively (Sub-section 4.4.2), thus allowing my approach to be more suitable in projects without closed bug reports similar to the new bug reports. Hence I concluded that my approach localises many bugs without using similar bug fix information, which were only localised by BugLocator, BRTracer or BLUiR using similar bug information, thus confirming the applicability of my approach also to software projects without history.

RQ2.2: ***What is the overall contribution of the VSM variant adopted in my approach, and how does it perform compared to rVSM?*** IR-based approaches to locating bugs use a base IR technique that is applied in a context-specific way or combined with bespoke heuristics. However, the exact variant of the underlying tf/idf[2] model used may affect results (Saha *et al.*, 2013). The off-the-shelf model VSM used in BLUiR already outperforms BugLocator, which introduced rVSM, a bespoke VSM variant.

Since my approach also uses an off-the-shelf VSM tool, different from the one used by BLUiR, I conducted experiments to determine the contribution of the file names, i.e. lexical similarity, and the contribution of the IR model, i.e. VSM. I found that VSM is a crucial component to achieve the best performance for projects with a larger number of files, which makes the use of term and document frequency more meaningful, but that in smaller projects it is rather small, which showed the crucial contribution of lexical similarity scoring for the improved performance of my approach.

I also confirm that the exact IR variant used is paramount: Lucene's VSM and my simple lexical matching outperform BugLocator's bespoke rVSM in many cases as well as BLUiR's Okapi (Sub-section 4.4.3). However, VSM on its own isn't enough to outperform

[2] tf/idf (term frequency/inverse document frequency) is explained in Chapter 2, Sub-section 2.2.1

other approaches.

6.1.3 User Studies

Evaluating my approach in Chapter 4, with one commercial and a range of open source projects showed it outperformed current state-of-the-art tools in a simpler, faster, and more general way that does not require history. To investigate the generalisability of my approach particularly in other commercial environments, I asked my third research question as follows.

RQ3: How does the approach perform in industrial applications and does it benefit developers by presenting the results ranked in the order of relevance for the bug report at hand?

To address RQ3 in Chapter 5, I conducted user studies in three different companies with professional developers. The first part of RQ3 aimed to demonstrate the applicability of my approach in industrial environments. Commercial applications and bug reports may have different characteristics to the OSS applications investigated in Chapter 4, thus impacting the performance of my approach. The results of the user study showed that in case of business applications, my tool also achieved to place at least one file in the top-10 for 90% of bug reports on average (see Answer 4 in Sub-section 5.3.2), thus confirming the first part of RQ3 in that my approach is applicable also in commercial environments.

The second part of RQ3 aimed to investigate how developers perceive the search results of ConCodeSe, which presents a ranked list of candidate source code files that may be relevant for a bug report at hand during software maintenance. Developers stated that since most of the relevant files were positioned in the top-5, they were able to avoid the error prone tasks of browsing long result lists and performing repetitive search queries (Sub-section 5.4). This confirms that presenting the results ranked in the order of relevance for the task at hand aids developers during maintenance, thus affirmatively answers the second part of RQ3.

6.1.4 Validity of My Hypothesis

From all the results obtained by answering my research questions as summarised in the previous Sub-sections, I conclude that leveraging the occurrence of file names in bug reports leads in most cases to better performance than using project history and contribution of similar bug reports.

Therefore I can answer my hypothesis affirmatively: *superior results can be achieved* *without drawing on past history by utilising only the information, i.e. file names, available in the current bug report and considering source code comments, stemming, and a combination of both independently, to derive the best rank for each file.*

6.2 Contributions

My research offers a more efficient and light-weight IR approach, which does not require any further analysis, e.g. to trace executed classes by re-running the scenarios described in the bug reports, and makes the following contributions.

1. A novel algorithm, which scores source code files relevant for a reported bug based on key positions and stack trace information found in the bug report without requiring any project history.

2. Implementation of the algorithm in a tool together with the datasets and the results, for example, the findings of the vocabulary study showing the concept correlation among project artefacts and vocabulary similarity for future research.

Given a bug report and the application's source code files, my approach uses a combination of lexical and structural information to suggest, in a ranked order, files that may have to be changed to resolve the reported bug. The algorithm considers heuristics based on contextual information, i.e. words in certain positions of the bug report summary and of the stack trace (if available in a bug report), as well as source code comments, stemming, and a combination of both independently, to derive the best rank for each file.

Moreover my approach not only outperforms other approaches, it does so in a simpler, faster, and more general way: it doesn't require past information like version history or similar bug reports that have been closed, nor the tuning of any weight factors to combine scores, nor the use of machine learning.

6.2.1 Recommendations for Practitioners

The search results still depend on the quality of bug descriptions. In case of tersely described bug reports, even experienced developers find it challenging to search for relevant files (Subsection 5.3.2). In addition to the previous studies which argue that the textual descriptions in bug reports are noisy (Zimmermann *et al.*, 2010), I also found that bug reports reveal two characteristics: (1) developer nature with technical details i.e. references to code files or (2) descriptive nature with business terminology i.e. use of test case scenarios.

Since bug report documents may come from a group of people who are unfamiliar with the vocabulary used in the source code, I propose that bug report descriptions contain a section for describing the relevant domain vocabulary. For example, a list of domain terms implemented by an application can be semi-automatically extracted and imported into the bug report management tool. Subsequently, when creating a bug report, the user may choose from the list of relevant domain terms or the tool may intelligently suggest the terms for selection.

To deal with crosscutting concerns — functionality implemented in multiple source code files of an application rather than in one isolated file — as illustrated by Shepherd *et al.* (2005), I recommend packaging the source files in architectural layers based on conceptual responsibility instead of technical functionality. This would first mediate improved communication with business users, since such communications take place at a conceptual level rather than technical level, and second assist in finding relevant files by considering the OOP architectural relations, e.g. abstraction, inheritance, when no call-relations exist.

In my user studies, the developers stated that some files would not have appeared in the result of the IDE search tool they use because those files do not contain the search terms. Hence I advocate combining lexical search with probabilistic methods like VSM should be an integral part of any modern search tool and that the search results should be presented in a ranked order of relevance from most to least important.

6.2.2 Suggestions for Future Research

Based on the results of my experiments conducted during my study, I suggest the following strands for future research.

Genetic algorithm: In my algorithm, I used arbitrary values to score the files obtained by manually tuning the scoring on two moderate size projects: AspectJ and SWT. I confirmed the rationale behind those values (namely, distinguish certain positions and assign much higher scores than base term matching) that led to the best results after trying out scoring variations (Sub-Section 4.4.1.4). Future research can apply a genetic algorithm (GA) to automatically derive the best fit combination of score values for each key position, stack trace and text term location.

Asadi *et al.* (2010) define GA as *"an iterative procedure that searches for the best solution to a given problem among a constant-size population, represented by a finite string of symbols, the genome"*. For example, an iterative search may start with an initial set of values (genomes) for the KP, ST and TT scores. At the end of each run, two utility functions, one for MAP and the other for MRR, could compare the performance of the results against the previous run and start a new search using a different combination of genomes. After a predetermined number of iterations, the utility functions could select the genome that resulted in the most performant MAP and MRR results. This would facilitate an automated generation of score values independently tailored for a project.

Feature requests: Although the evaluation datasets I used only include bug reports, my IR-based approach can be also applied to feature requests and not just bug reports, as it does not depend on past bug reports. Feature requests necessarily don't include a stack trace and they may not mention specific files. Hence future research investigating the performance of my approach with feature requests may benefit from evaluating the content of configuration files (e.g. config.xml, DB scripts, GUI files) as suggested during the user studies.

For example, in applications that utilise declarative configuration, the dependent class files are specified in a configuration file. This file could be analysed to discover how each file relates to one another within the context of the application's business domain. Also, in web based applications, the control flow between the GUI and the class files that process the submitted information is declared in the configuration files. Subsequently, the configurative relations and the control flow relations can be considered during the search when scoring a file. Finally, showing the relational flows, e.g. how a class file is associated with a GUI file, in the result list may point the developer directly to the location of the implementation for the new feature.

Term co-occurrences: I only used single-word concepts during the search, while business concepts are usually compound terms. Considering term co-occurances by evaluating also the words that are surrounding the search terms may lead to improved results. To facilitate this, future research may introduce, for example, project specific vocabulary dictionaries in the form of ontologies (Sub-Section 2.2.5) that define how concepts relate to each other. During the search, using the queried concept other related concepts can be considered to retrieve additional relevant files that may not have matching terms with the queried term but still may be relevant within the context of the business domain of the application for the bug report at hand.

Furthermore, it is conceivable that an IR engine using the LSI model (Sub-Section 2.2.2), for example, instead of VSM, to index the terms extracted from the source code identifiers, may produce more or less sensitive results to using file names in bug reports.

Call relations: Additional future research would be to identify the opportunities of using files in the results list as seed files to find other relevant files using call relations. It is acknowledged that during maintenance, developers perform search tasks using lexical information as well as navigate the structural information (Hill *et al.*, 2007), thus these two information sources can be combined to refine the search results by utilising any additional clues that call relations may provide.

For example, during the user studies, developers indicated that if the bug description revealed some back-end modules are the culprit, then they would look to see where the control flows are coming from, i.e. the callers (Sub-Section 5.3.1). They also said that if a certain class is calling many others then they would consider this to be a bad architecture and ignore the callees. Based on these two insights, call relations can be evaluated for importance and a score can be assigned to boost or penalise the rank of a file accordingly. Subsequently, the relevant files can be grouped together within the context of the search by leveraging the call-relations without deteriorating the performance.

Furthermore, in the case of bug reports requiring multiple files to be changed, investigating to see if the files that are scored in top-5 are also in the order of importance (e.g. high priority) for the developers compared to other tools would greatly benefit further research. For example, files A, B and C may be scored in the top-5 but file C may be the most important one within the context of the bug report at hand, thus positioned in the top-1.

GUI enhancements: Several cosmetic suggestions arose from the user studies for presenting the results and interacting in search fields, like proposing keywords, e.g. auto complete based on terms found in the source code (Sub-Section 5.3.1). Participants felt that it would be helpful to display, next to the ranked files, the words in those files that match the bug report so that the user can determine whether it really makes sense to investigate the content of the file or not. Interestingly one developer said that sometimes he did not consider the ranking as an important factor and suggested to group files into packages, based on words matching the package name and then rank the files within that group.

Future research may investigate different approaches to rank relevant files and present them in a user interface integrated into modern IDEs. Subsequently, for each ranked file, the matching words resulted in the ranking may be displayed as a teaser (preview) so that the developers would not need to open each file to examine its content.

Attachments in bug reports: Davies and Roper (2014) examined 1600 bug reports from Eclipse, Firefox[3], Apache[4] and Facebook API[5] to understand what information users provide. The authors found that three of the most important features are (1) observed behaviour: What the user saw happen in the application as a result of the bug; (2) expected behaviour: What the user expected to happen, usually contrasted with observed behaviour; and (3) steps to reproduce: Instructions that the developer can use to reproduce the bug.

However, the authors claim that these features are not sufficient enough for software maintenance, otherwise there would be no need for automated bug localisation because a developer would directly be able to utilise the information contained in the bug report. In my research, I successfully leveraged the file names and the stack traces found in the bug reports during automated bug localisation. A future research direction would be to also evaluate the other types of information contained in the bug reports, for example, attachments in the form of log files, screen shots or links to external websites containing code examples as identified by Davies and Roper (2014).

6.3 A Final Rounding Off

In my research, I focused on one single type of contextual information in the bug report: the occurrence of file names. The assumption is that if a file is mentioned in the bug report, it likely needs to be changed to fix the bug. Thus leveraging the file names in key positions and stack trace based on the contextual heuristics are the main contributions of my research.

As software applications become integral part of our lives, technological advancements demand existing applications to be maintained to meet with ongoing requirements. With information available at our finger tips, companies are required to deliver solutions in a timely manner so that they may remain competitive.

To reduce the time it takes to identify the location of a reported bug is a key challenge: wouldn't it be nice to to have a system where upon creating a bug description, the source code files of the application is automatically searched and a list of candidate files are identified with

[3] http://www.mozilla.org/firefox

[4] http://apache.org

[5] https://developers.facebook.com

a link to the bug report so that the developer may focus on implementing the solution instead of trying to figure out where to implement it first? My research presented an important step in this direction.

As the closing paragraph, thus the closing Chapter in my thesis, I advocate the importance of easing the daily tasks of software developers by providing them with a ranked list of search results where lexical similarity during search is complemented with probabilistic methods in a tool integrated in modern IDEs as well as in modern project management life cycle tools.

Bibliography

S. L. Abebe; S. Haiduc; P. Tonella; and A. Marcus (2009) Lexicon bad smells in software. In *2009 16th Working Conference on Reverse Engineering*, pp. 95–99.

S. L. Abebe; S. Haiduc; P. Tonella; and A. Marcus (2011) The effect of lexicon bad smells on concept location in source code. In *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*, pp. 125–134.

S. L. Abebe and P. Tonella (2010) Natural language parsing of program element names for concept extraction. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pp. 156–159.

N. Anquetil and T. Lethbridge (1998a) Assessing the relevance of identifier names in a legacy software system. In *Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research*, Toronto, Ontario, Canada, CASCON '98, pp. 4–. IBM Press.

N. Anquetil and T. Lethbridge (1998b) Extracting concepts from file names; a new file clustering criterion. In *Software Engineering, 1998. Proceedings of the 1998 International Conference on*, pp. 84–93.

G. Antoniol; G. Canfora; G. Casazza; and A. D. Lucia (2000a) Information retrieval models for recovering traceability links between code and documentation. In *Software Maintenance, 2000. Proceedings. International Conference on*, pp. 40–49.

G. Antoniol; G. Canfora; G. Casazza; A. D. Lucia; and E. Merlo (2000b) Tracing object-oriented code into functional requirements. In *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on*, pp. 79–86.

G. Antoniol; G. Canfora; G. Casazza; A. D. Lucia; and E. Merlo (2002) Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, pp. 970–983.

V. Arnaoudova; M. D. Penta; G. Antoniol; and Y. G. Guéhéneuc (2013) A new family of software anti-patterns: Linguistic anti-patterns. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pp. 187–196.

F. Arvidsson and A. Flycht-Eriksson (2008) Ontologies i. `http://www.ida.liu.se/janma/SemWeb/Slides/ontologies1.pdf`. Accessed: 2016-07.

F. Asadi; G. Antoniol; and Y. G. Guéhéneuc (2010) Concept location with genetic algorithms: A comparison of four distributed architectures. In *Search Based Software Engineering (SSBSE), 2010 Second International Symposium on*, pp. 153–162.

Basel-II (2006) International convergence of capital measurement and capital standards: A revised framework - comprehensive version. `http://www.bis.org/publ/bcbs128.htm`. Accessed: 2016-08.

R. Bendaoud; A. M. Rouane Hacene; Y. Toussaint; B. Delecroix; and A. Napoli (2007) Text-based ontology construction using relational concept analysis. In *International Workshop on Ontology Dynamics - IWOD 2007*. Innsbruck, Austria.

K. H. Bennett and V. T. Rajlich (2000) Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, Limerick, Ireland, ICSE '00, pp. 73–87. ACM, New York, NY, USA.

N. Bettenburg; R. Premraj; T. Zimmermann; and S. Kim (2008) Extracting structural information from bug reports. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, Leipzig, Germany, MSR '08, pp. 27–30. ACM, New York, NY, USA.

T. J. Biggerstaff; B. G. Mitbander; and D. Webster (1993) The concept assignment problem in program understanding. In *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, Maryland, USA, ICSE '93, pp. 482–498. IEEE Computer Society Press, Los Alamitos, CA, USA.

S. Boslaugh and P. Watters (2008) *Statistics in a nutshell*. O'Reilly Publishing, 1st edition.

S. Butler (2016) *Analysing Java Identifier Names*. Ph.D. thesis, The Open University.

S. Butler; M. Wermelinger; Y. Yu; and H. Sharp (2010) Exploring the influence of identifier names on code quality: An empirical study. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pp. 156–165.

S. Butler; M. Wermelinger; Y. Yu; and H. Sharp (2011) *Improving the Tokenisation of Identifier Names*, pp. 130–154. Springer Berlin Heidelberg, Berlin, Heidelberg.

G. Capobianco; A. D. Lucia; R. Oliveto; A. Panichella; and S. Panichella (2009) On the role of the nouns in ir-based traceability recovery. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pp. 148–157.

C. Caprile and P. Tonella (1999) Nomen est omen: analyzing the language of function identifiers. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pp. 112–122.

J. Chang and D. M. Blei (2009) Relational topic models for document networks. `https://www.cs.princeton.edu/~blei/papers/ChangBlei2009.pdf`. Accessed: 2016-08.

T. A. Corbi (1989) Program understanding: Challenge for the 1990's. *IBM Syst. J.*, pp. 294–306.

S. Davies and M. Roper (2013) Bug localisation through diverse sources of information. In *Software Reliability Engineering Workshops (ISSREW), 2013 IEEE International Symposium on*, pp. 126–131.

S. Davies and M. Roper (2014) What's in a bug report? In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, Torino, Italy, ESEM '14, pp. 26:1–26:10. ACM, New York, NY, USA.

J. W. Davison; D. M. Mancl; and W. F. Opdyke (2000) Understanding and addressing the essential costs of evolving systems. *Bell Labs Technical Journal*, 5(2):44–54.

F. Deißenböck and M. Pizka (2006) Concise and consistent naming. *Software Quality Journal*, pp. 261–282.

F. Deißenböck and D. Raţiu (2006) A unified meta-model for concept-based reverse engineering. In *In Proceedings of the 3rd International Workshop on Metamodels, Schemas, Grammars and Ontologies*.

E. W. Dijkstra (1959) A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271.

T. Dilshener (2012) Improving information retrieval-based concept location using contextual relationships. In *2012 34th International Conference on Software Engineering (ICSE)*, pp. 1499–1502.

T. Dilshener and M. Wermelinger (2011) Relating developers' concepts and artefact vocabulary in a financial software module. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pp. 412–417.

T. Dilshener; M. Wermelinger; and Y. Yu (2016) Locating bugs without looking back. In *Proceedings of the 13th International Conference on Mining Software Repositories*, Austin, Texas, MSR '16, pp. 286–290. ACM, New York, NY, USA.

M. Eaddy; A. V. Aho; G. Antoniol; and Y. G. Guéhéneuc (2008) Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pp. 53–62.

T. Eisenbarth; R. Koschke; and D. Simon (2001) Aiding program comprehension by static and dynamic feature analysis. In *Software Maintenance, 2001. Proceedings. IEEE International Conference on*, pp. 602–611.

A. D. Eisenberg and K. D. Volder (2005) Dynamic feature traces: finding features in unfamiliar code. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 337–346.

L. H. Etzkorn; C. G. Davis; and L. L. Bowen (2001) The language of comments in computer software: A sublanguage of english. *Journal of Pragmatics*, pp. 1731 – 1756.

M. Feilkas; D. Raţiu; and E. Jurgens (2009) The loss of architectural knowledge during system evolution: An industrial case study. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pp. 188–197.

Z. P. Fry; D. Shepherd; E. Hill; L. Pollock; and K. Vijay-Shanker (2008) Analysing source code: looking for useful verb-direct object pairs in all the right places. *IET Software*, pp. 27–36.

M. Gethers; R. Oliveto; D. Poshyvanyk; and A. D. Lucia (2011) On integrating orthogonal information retrieval methods to improve traceability recovery. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pp. 133–142.

T. R. Gruber (1993) A translation approach to portable ontology specifications. *Knowledge Acquisition*, pp. 199 – 220.

S. Haiduc and A. Marcus (2008) On the use of domain terms in source code. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pp. 113–122.

D. L. Hall and J. Llinas (1997) An introduction to multisensor data fusion. *Proceedings of the IEEE*, pp. 6–23.

Y. Hayase; Y. Kashima; Y. Manabe; and K. Inoue (2011) Building domain specific dictionaries of verb-object relation from source code. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pp. 93–100.

S. Hayashi; T. Yoshikawa; and M. Saeki (2010) Sentence-to-code traceability recovery with domain ontologies. In *2010 Asia Pacific Software Engineering Conference*, pp. 385–394.

E. Hill (2010) *Integrating Natural Language and Program Structure Information to Improve Software Search and Exploration.* Ph.D. thesis, Newark, DE, USA. AAI3423409.

E. Hill; D. Binkley; D. Lawrie; L. Pollock; and K. Vijay-Shanker (2013) An empirical study of identifier splitting techniques. *Empirical Software Engineering*, 19(6):1754–1780.

E. Hill; Z. P. Fry; H. Boyd; G. Sridhara; Y. Novikova; L. Pollock; and K. Vijay-Shanker (2008) Amap: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, Leipzig, Germany, MSR '08, pp. 79–88. ACM, New York, NY, USA.

E. Hill; L. Pollock; and K. Vijay-Shanker (2007) Exploring the neighborhood with dora to expedite software maintenance. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, Atlanta, Georgia, USA, ASE '07, pp. 14–23. ACM, New York, NY, USA.

E. Hill; L. Pollock; and K. Vijay-Shanker (2009) Automatically capturing source code context of nl-queries for software maintenance and reuse. In *2009 IEEE 31st International Conference on Software Engineering*, pp. 232–242.

E. W. Høst and B. M. Østvold (2007) The programmer's lexicon, volume i: The verbs. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pp. 193–202.

I. Hsi; C. Potts; and M. Moore (2003) Ontological excavation: unearthing the core concepts of the application. In *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on*, pp. 345–353.

B. Hunt; B. Turner; and K. McRitchie (2008) Software maintenance implications on cost and schedule. In *Aerospace Conference, 2008 IEEE*, pp. 1–6.

K. Kevic and T. Fritz (2014) Automatic search term identification for change tasks. In *Companion Proceedings of the 36th International Conference on Software Engineering*, Hyderabad, India, ICSE Companion 2014, pp. 468–471. ACM, New York, NY, USA.

G. Kiczales; E. Hilsdale; J. Hugunin; M. Kersten; J. Palm; and W. G. Griswold (2001) Getting started with ASPECTJ. *Commun. ACM*, 44(10):59–65.

P. S. Kochhar; Y. Tian; and D. Lo (2014) Potential biases in bug localization: Do they matter? In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, Vasteras, Sweden, ASE '14, pp. 803–814. ACM, New York, NY, USA. URL `http://doi.acm.org/10.1145/2642937.2642997`.

P. S. Kochhar; X. Xia; D. Lo; and S. Li (2016) Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, Saarbr&#252;cken, Germany, ISSTA 2016, pp. 165–176. ACM, New York, NY, USA. URL `http://doi.acm.org/10.1145/2931037.2931051`.

A. Kuhn; S. Ducasse; and T. Girba (2007) Semantic clustering: Identifying topics in source code. *Information and Software Technology*, pp. 230 – 243. 12th Working Conference on Reverse Engineering.

D. Lawrie (2012) Discussion of appropriate evaluation metrics, 1st workshop on text analysis in software maintenance. `https://dibt.unimol.it/TAinSM2012/slides/dawn.pdf`. Accessed: 2016-04.

D. Lawrie; D. Binkley; and C. Morrell (2010) Normalizing source code vocabulary. In *2010 17th Working Conference on Reverse Engineering*, pp. 3–12.

D. Lawrie; C. Morrell; H. Feild; and D. Binkley (2006) What's in a name? a study of identifiers. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pp. 3–12.

M. M. Lehman (1980) Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, pp. 1060–1076.

Z. Li; L. Tan; X. Wang; S. Lu; Y. Zhou; and C. Zhai (2006) Have things changed now?: An empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, San Jose, California, ASID '06, pp. 25–33. ACM, New York, NY, USA.

D. Liu; A. Marcus; D. Poshyvanyk; and V. Rajlich (2007) Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, Atlanta, Georgia, USA, ASE '07, pp. 234–243. ACM, New York, NY, USA.

Lucia; F. Thung; D. Lo; and L. Jiang (2012) Are faults localizable? In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, Zurich, Switzerland, MSR '12, pp. 74–77. IEEE Press, Piscataway, NJ, USA.

C. D. Manning; P. Raghavan; and H. Schütze (2008) *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA.

A. Marcus and S. Haiduc (2013) *Text Retrieval Approaches for Concept Location in Source Code*, pp. 126–158. Springer Berlin Heidelberg, Berlin, Heidelberg.

A. Marcus and J. I. Maletic (2003) Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*, Portland, Oregon, ICSE '03, pp. 125–135. IEEE Computer Society, Washington, DC, USA.

A. Marcus; V. Rajlich; J. Buchta; M. Petrenko; and A. Sergeyev (2005) Static techniques for concept location in object-oriented code. In *13th International Workshop on Program Comprehension (IWPC'05)*, pp. 33–42.

A. Marcus; A. Sergeyev; V. Rajlich; and J. I. Maletic (2004) An information retrieval approach to concept location in source code. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pp. 214–223.

R. C. Martin (2008) *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall, 1st edition.

G. A. Miller (1995) Wordnet: A lexical database for english. *Commun. ACM*, pp. 39–41.

L. Moreno; W. Bandara; S. Haiduc; and A. Marcus (2013) On the relationship between the vocabulary of bug reports and source code. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pp. 452–455.

L. Moreno; J. J. Treadway; A. Marcus; and W. Shen (2014) On the use of stack traces to improve text retrieval-based bug localization. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pp. 151–160.

B. D. Nichols (2010) Augmented bug localization using past bug information. In *Proceedings of the 48th Annual Southeast Regional Conference*, Oxford, Mississippi, ACM SE '10, pp. 61:1–61:6. ACM, New York, NY, USA.

C. Nunes; A. Garcia; E. Figueiredo; and C. Lucena (2011) Revealing mistakes in concern mapping tasks: An experimental evaluation. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pp. 101–110.

Oracle (1999) Code conventions for the java programming language, section 5 comments. `http://www.oracle.com/technetwork/java/codeconventions-141999.html`. Accessed: 2016-07.

Y. Padioleau; L. Tan; and Y. Zhou (2009) Listening to programmers taxonomies and characteristics of comments in operating system code. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pp. 331–341. IEEE Computer Society, Washington, DC, USA.

D. L. Parnas (1972) On the criteria to be used in decomposing systems into modules. *Commun. ACM*, pp. 1053–1058.

C. Parnin and A. Orso (2011) Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and*

Analysis, Toronto, Ontario, Canada, ISSTA '11, pp. 199–209. ACM, New York, NY, USA. URL `http://doi.acm.org/10.1145/2001420.2001445`.

M. Petrenko and V. Rajlich (2013) Concept location using program dependencies and information retrieval (depir). *Information and Software Technology*, pp. 651 – 659.

M. Petrenko; V. Rajlich; and R. Vanciu (2008) Partial domain comprehension in software evolution and maintenance. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pp. 13–22.

T. M. Pigoski (1996) *Practical Software Maintenance: Best Practices for Managing Your Software Investment.* Wiley Publishing, 1st edition.

M. F. Porter (1997) Readings in information retrieval. chapter An Algorithm for Suffix Stripping, pp. 313–316. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

D. Poshyvanyk; Y. G. Guéhéneuc; A. Marcus; G. Antoniol; and V. Rajlich (2007) Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, pp. 420–432.

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

D. Raţiu; M. Feilkas; and J. Jurjens (2008) Extracting domain ontologies from domain specific apis. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pp. 203–212.

S. Rahman; K. K. Ganguly; and K. Sakib (2015) An improved bug localization using structured information retrieval and version history. In *2015 18th International Conference on Computer and Information Technology (ICCIT)*, pp. 190–195.

V. Rajlich and N. Wilde (2002) The role of concepts in program comprehension. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pp. 271–278.

S. Rao and A. Kak (2011) Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, Waikiki, Honolulu, HI, USA, MSR '11, pp. 43–52. ACM, New York, NY, USA.

S. Ratanotayanon; H. J. Choi; and S. E. Sim (2010) My repository runneth over: An empirical study on diversifying data sources to improve feature search. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pp. 206–215.

M. Revelle (2009) Supporting feature-level software maintenance. In *2009 16th Working Conference on Reverse Engineering*, pp. 287–290.

M. Revelle; B. Dit; and D. Poshyvanyk (2010) Using data fusion and web mining to support feature location in software. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pp. 14–23.

M. Revelle and D. Poshyvanyk (2009) An exploratory study on assessing feature location techniques. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pp. 218–222.

M. P. Robillard (2005) Automatic generation of suggestions for program investigation. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Lisbon, Portugal, ESEC/FSE-13, pp. 11–20. ACM, New York, NY, USA.

M. R. Robillard and G. C. Murphy (2002) Concern graphs: finding and describing concerns using structural program dependencies. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pp. 406–416.

R. K. Saha; M. Lease; S. Khurshid; and D. E. Perry (2013) Improving bug localization using structured information retrieval. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pp. 345–355.

G. Salton and C. Buckley (1988) Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, pp. 513 – 523.

T. Savage; M. Revelle; and D. Poshyvanyk (2010) Flat3: Feature location and textual tracing tool. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, Cape Town, South Africa, ICSE '10, pp. 255–258. ACM, New York, NY, USA.

A. Schröter; N. Bettenburg; and R. Premraj (2010) Do stack traces help developers fix bugs?

In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pp. 118–121.

P. Shao and R. K. Smith (2009) Feature location by ir modules and call graph. In *Proceedings of the 47th Annual Southeast Regional Conference*, Clemson, South Carolina, ACM-SE 47, pp. 70:1–70:4. ACM, New York, NY, USA.

D. Shepherd; Z. P. Fry; E. Hill; L. Pollock; and K. Vijay-Shanker (2007) Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th International Conference on Aspect-oriented Software Development*, Vancouver, British Columbia, Canada, AOSD '07, pp. 212–224. ACM, New York, NY, USA.

D. Shepherd; L. Pollock; and T. Tourwé (2005) Using language clues to discover crosscutting concerns. In *Proceedings of the 2005 Workshop on Modeling and Analysis of Concerns in Software*, St. Louis, Missouri, MACS '05, pp. 1–6. ACM, New York, NY, USA.

D. Shepherd; L. Pollock; and K. Vijay-Shanker (2006) Towards supporting on-demand virtual remodularization using program graphs. In *Proceedings of the 5th International Conference on Aspect-oriented Software Development*, Bonn, Germany, AOSD '06, pp. 3–14. ACM, New York, NY, USA.

J. Sillito; G. C. Murphy; and K. D. Volder (2008) Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, pp. 434–451.

B. Sisman and A. C. Kak (2012) Incorporating version histories in information retrieval based bug localization. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pp. 50–59.

G. Sridhara; E. Hill; L. Pollock; and K. Vijay-Shanker (2008) Identifying word relations in software: A comparative study of semantic similarity tools. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pp. 123–132.

J. Starke; C. Luce; and J. Sillito (2009) Searching and skimming: An exploratory study. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pp. 157–166.

Y. Uneno; O. Mizuno; and E. H. Choi (2016) Using a distributed representation of words in localizing relevant files for bug reports. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 183–190.

E. M. Voorhees (2001) The trec question answering track. *Nat. Lang. Eng.*, pp. 361–378.

Q. Wang; C. Parnin; and A. Orso (2015) Evaluating the usefulness of ir-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, Baltimore, MD, USA, ISSTA 2015, pp. 1–11. ACM, New York, NY, USA. URL http://doi.acm.org/10.1145/2771783.2771797.

S. Wang and D. Lo (2014) Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22Nd International Conference on Program Comprehension*, Hyderabad, India, ICPC 2014, pp. 53–63. ACM, New York, NY, USA.

S. Wang; D. Lo; Z. Xing; and L. Jiang (2011) Concern localization using information retrieval: An empirical study on linux kernel. In *2011 18th Working Conference on Reverse Engineering*, pp. 92–96.

M. Wermelinger; Y. Yu; and A. Lozano (2008) Design principles in architectural evolution: A case study. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pp. 396–405.

Wessa (2016) Free statistics software, office for research development and education, version 1.1.23-r7. http://www.wessa.net/. Accessed: 2016-03.

N. Wilde and M. C. Scully (1995) Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*, pp. 49–62.

C. P. Wong; Y. Xiong; H. Zhang; D. Hao; L. Zhang; and H. Mei (2014) Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pp. 181–190.

X. Xia; L. Bao; D. Lo; and S. Li (2016) Automated debugging considered harmful - considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 267–278.

X. Ye; R. Bunescu; and C. Liu (2014) Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium* *on Foundations of Software Engineering*, Hong Kong, China, FSE 2014, pp. 689–699. ACM, New York, NY, USA.

K. C. Youm; J. Ahn; J. Kim; and E. Lee (2015) Bug localization based on code change histories and bug reports. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*, pp. 190–197.

J. Zhou; H. Zhang; and D. Lo (2012a) Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering*, Zurich, Switzerland, ICSE '12, pp. 14–24. IEEE Press, Piscataway, NJ, USA.

K. Zhou; K. M. Varadarajan; M. Zillich; and M. Vincze (2012b) Robust multiple model estimation with jensen-shannon divergence. In *Pattern Recognition (ICPR), 2012 21st International Conference on*, pp. 2136–2139.

T. Zimmermann; R. Premraj; N. Bettenburg; S. Just; A. Schröter; and C. Weiss (2010) What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5):618–643.

Appendix A

Glossary

This Appendix describes the terminology used throughout my thesis within the context of my research.

Bug localisation is the process of identifying where to make changes in response to a bug report (Moreno *et al.*, 2014).

Bug report describes an unexpected and unintended erroneous behaviour, known as bugs of a software system.

Concept is a phrase or a single word that describes the unit of human knowledge (thought) existing within the domain of the software application (Rajlich and Wilde, 2002), for example, in a financial domain dealing with risk the phrase standalone risk describes a type of risk concept.

Concept location is the task of locating the program identifiers implementing the concepts at hand prior to performing software maintenance.

Contextual model is the repository, i.e. database, containing the vocabulary (terms) extracted from an application's project artefacts, i.e. source code files, bug reports, concepts and user guide. It also provides referential information within the context of each artefact to determine the origin of the terms, for example, the source code file names from where the vocabulary was extracted.

Formal Concept Analysis (FCA) is a two dimensional model for grouping concepts. It consists of extension, covering the entire objects (i.e. program elements) belonging to a concept, and intension, covering all the at- tributes that are shared by all the objects being considered.

Free/Libre and open source software (FLOSS) refers to software developed through public collaboration and whose source code is publicly available for use and distribution without any charge.

Latent Semantic Indexing (LSI) is an information retrieval model. LSI organises the occurrences of project artefact terms in its repository of term by document matrix and applies the Single Value Decomposition (SVD) principle.

Lexical Chaining (LC) is technique to group semantically related terms in a document by computing the semantic distance (i.e. strength of relationship) between two terms.

Mean Average Precision (MAP) provides a synthesised way to measure the quality of retrieved files, when there are more than one related file retrieved for the bug report. The average precision for a bug report is the mean of the precision values obtained for all affected files listed in the result set. It is calculated as the sum of the average precision value for each bug report divided by the number of bug reports for a given project.

Mean Reciprocal Rank (MRR) measures the overall effectiveness of retrieval for a set of bug reports. The reciprocal rank for a query is the inverse rank of the first relevant document found. The mean reciprocal rank is the average of the reciprocal ranks of results of a set of queries.

Object Oriented Programming (OOP) is a paradigm for programming model.

Open Source Software(OSS) refers to software whose source code is publicly available for use and distribution.

Program identifiers (aka. identifier names) are the named source code entities that describe the classes, methods and fields of an OOP software application. In other words they are the given names of the program elements.

Project artefact is a result of a software application development, for example, the textual documentation of the application. In our research we use the project artefacts: (1) the source code, (2) the user guide and (3) the bug report documents describing the implementation of new or the modification of an existing business requirement in the software application.

Program elements are the programming language specific source code entities, like class files, methods and fields used in an OOP language like Java.

Relational Topic Modelling (RTM) is a model of documents with collection of words and the relations between them

Revised Vector Space Model (rVSM) is an information retrieval model that scores each source code file against the given bug report introduced by Zhou *et al.* (2012a). In addition, each file gets a similarity score (**SimiScore**) based on whether the file was affected by one or more closed bug reports similar to the given bug report.

Scenario Based Probabilistic (SBP) ranking allows the tracing of execution scenarios and lists of program elements (i.e. source code classes and methods) ranked according to their similarity for a given concept when it is executed during a particular scenario.

Single Value Decomposition (SVD) is a technique used to reduce noise in a repository of term by document matrix while keeping the relative distance between artefacts intact.

Structural relations are inheritance relationships (i.e. class hierarchies) or caller-callee relations between the program elements. In the caller-callee relation, the caller is the method calling the current method and the callee is the method being called from the current method.

Term frequency (tf) and inverse document frequency (idf) (TF.IDF) is used to score the weighting of a term in a given collection of terms. The **tf** is the number of times a term occurs in a document and the idf is the ratio between the total numbers of documents over the number of documents containing the term. The **idf** is used to measure if a term occurs more or less across a number of documents .

Vector Space Model (VSM) is an information retrieval model where vectors represent queries (bug reports in the case of bug localisation) and documents source code files. Each element of the vector corresponds to a word or term extracted from the query's or document's vocabulary. The relevance of a document to a query can be directly evaluated by calculating the similarity of their word vectors.

Appendix B

Top-5 Concepts

Tables B.1 and B.2 list the top-5 concepts occuring across all artefacts of the projects.

Table B.1: Top-5 concepts occuring across the project artefacts

Project	Search type	Bug reports (BR)	User guide (UG)	Source code (SRC)
AspectJ	normal	aspect	aspect	type
		pointcut	pointcut	point
		type	point	aspect
		advice	join	source
		object	crosscutting	target
	stemmed	aspect (aspect)	aspect (aspect)	type (type)
		pointcut (pointcut)	point (point)	aspect (aspect)
		type (type)	pointcut (pointcut)	point (point)
		weav (weaving)	crosscut (crosscutting)	sourc (source)
		return (returning)	join (join)	target (target)
Eclipse	normal	debug	file	page
		file	cvs	text
		line	workbench	file
		build	editor	source
		ant	ant	label
	stemmed	file (file)	file (file)	page (page)
		view (views)	view (views)	file (file)
		debug (debug, debugging)	project (projects)	text (text)
		line (line)	cv (cvs)	line (line)
		build (build)	editor (editor,editors)	label (label)
SWT	normal	display	widget	text
		text	display	menu
		widget	help	window
		tree	layout	list
		menu	label	file
	stemmed	displai (display)	widget (widget)	text (text)
		widget (widget)	displai (display)	menu (menu)
		text (text)	layout (layout)	window (window)
		window (window)	window (window)	list (list)
		menu (menu)	help (help)	file (file)

Table B.2: Top-5 concepts occuring across the project artefacts - cont.

Project	**Search type**	**Bug reports (BR)**	**User guide (UG)**	**Source code (SRC)**
ZXing	normal	code	scanner	data
		upc	code	size
		ean	source	matrix
		fixed	core	width
		size	reader	start
	stemmed	code (code)	code (code)	decod (decoder)
		fix (fixed)	scanner (scanner)	data (data)
		decod (decoder)	sourc (source)	size (size)
		upc (upc)	start (start)	width (width)
		ean (ean)	reader (reader)	matrix (matrix)
Tomcat	normal	http	web	context
		request	application	value
		servlet	context	request
		session	server	session
		jsp	value	servlet
	stemmed	http (http, https)	applic (application)	valu (value)
		request (request)	web (web)	context (context)
		servlet (servlet)	context (context)	request (request)
		session (session)	server (server)	session (session)
		jsp (jsp)	valu (value)	listen (listener)
ArgoUML	normal	diagram	diagram	action
		uml	use	state
		source	class	uml
		class	case	class
		activity	uml	diagram
	stemmed	diagram (diagram)	diagram (diagram)	action (action)
		uml (uml)	us (use)	state (state)
		class (class)	class (class)	uml (uml)
		sourc (source)	case (case)	class (class)
		us (use)	associ (association)	gener (generalization)
Pillar1	normal	time	risk	risk
		current	business	base
		capital	value	value
		aggregated	line	time
		risk	group	index
	stemmed	time (time)	risk (risk)	risk (risk)
		current (current)	busi (business)	base (base)
		aggreg (aggregated)	line (line)	valu (value)
		capit (capital)	valu (value)	time (time)
		label (label)	calcul (calculation)	index (index)
Pillar2	normal	market	market	index
		value	calculation	market
		calculation	scenario	value
		risk	investment	risk
		asset	index	scenario
	stemmed	valu (value)	calcul (calculation)	index (index)
		market (market)	market (market)	valu (value)
		calcul (calculation)	valu (value)	market (market)
		risk (risk)	scenario (scenario)	risk (risk)
		asset (asset)	index (index)	scenario (scenario)

Appendix C

Sample Bug Report Descriptions

Table C.1 shows sub-set of tersely described Pillar1 and Pillar2 bug reports. Note that Pillar2 BR #2010 has incorrect spelling of the word greater as **greather** in the summary of the original document.

Table C.1: Sub-set of bug report descriptions for Pillar1 and Pillar2

BR	**Description for Pillar1**
1619	unrecoverable error for error parameter poisson.
1733	Wiring concept for two-phase components should be extended
2024	Handling IllegalAcceessException in Packets and Collectors.
2081	Check usage of enum classes
2093	Quota event limit not implemented.
2163	GIRAModel not compatible with current master branch
2200	NPE when changing result views
BR	**Description for Pillar2**
2002	Roundup export data to an importable excel format.
2003	Pdlgd export data to an importable excel format.
2010	Allow volatility values **greather** than 1.
2063	New reallocation method ”Use Asset Diversified Risk”.
2068	Show in both sub systems Market and PD/LGD all calculation states
2074	Dialog to distribute lambda factors similar to other module.
2081	Show approx. group values and diversification effects.

Appendix D

Search Pattern for Extracting Stack Trace

The regular expression in Figure D.1 is used to search the application-only source files, i.e. excluding third party and Java library files, in the stack trace.

```
at [^java. | ^sun.]([packageName]*).  // exclude java packages
([className]*).($[innerClassName]*)?. // consider any inner class
([methodName]*) ( ((([fileName]*).java):
[lineNumber])*? (Unknown Source)? (Native Method)? )
```

Figure D.1: Search pattern for stack trace

Appendix E

Domain Concepts

Tables E.1 through E.7 list, in two columns, all the domain concepts used.

Table E.1: Graphical User Interface (GUI) Domain Concepts

CONCEPTS - I	**CONCEPTS - II**
Button	Balloon help
Context menu	Heads-up display in computing
Menu	Heads-up display in video games
Pie menu	Icon
Checkbox	Infobar
Combo box	Label
Cycle button	Loading screen
Drop-down list	Progress indicator
Grid view	Progress bar
List box	Splash screen
List builder	Throbber
Radio button	Sidebar
Scrollbar	Status bar
Inspector window	Palette window
Modal window	Spinner
Layout manager	Search box
Look and feel	Text box
Mouseover	Toast
Accordion	Tooltip
Menu bar	About box
Panel	Alert dialog box
Ribbon	Dialog box
Tab	File dialog
Toolbar	Widget toolkit
Window	WIMP
Address bar	Frame Fieldset
Breadcrumb	Slider
Hyperlink	Disclosure widget
Tree view	—

Table E.2: Integrated Development Environment (IDE) Domain Concepts

CONCEPTS	**CONCEPTS**
Workbench	Java Views
Perspectives	Java Editor
Editors	Quick Fix and Assist
Views	Templates
Toolbars	Java Search
Markers	Refactoring Support
Bookmarks	Debugger
Label decorations	Scrapbook
Ant & External tools	Local Debugging
Team programming with CVS	Remote Debugging
Accessibility features	Breakpoints
Features	String Externalization
Java Projects	Extensions and Extension Points
Java Builder	Feature
Java Perspectives	Fragment
Application Frameworks	Environment Configuration
Component Frameworks	Peer Review Support
File Managers	Collaboration Support
Text Editors	Deployment Support
Execution Environments	Role-Based Views
Design and Modeling	Graphical User Interfaces
Documentation Generators	Unit-Testing Frameworks
Test Management	Static Code Analyzers
Plug-in	JSP JSF Source Editing Tools
Product	Web Page Editor
Update Site	JSF Application Configuration
Rhino Debug	JSF Tag Registry
JSDT Features	JSF Component Tree
JSDT Known Limitations	Java API for XML-Based Web Services
JSF Specification	Command Line Interfaces
JSF Facets	Application Programming Interfaces
JSF Libraries	Compilation and Build
Component Templates	Task Lists Application Templates

Table E.3: Bar Code (imaging/scanning) Domain Concepts

CONCEPTS	**CONCEPTS**
Aspect Ratio	Dot Size (Printer)
Background	Dot Size (Scanner)
Bar Code Character	EAN
Bar Code Density	EAN International
Bearer Bars	EAN Bar Code (European Article Number)
Bi-Directional	Extended Code 39
Character Set	Flat Bed Scanner
Check Character	Guard Bars
Clear Area	Light Pen
Code 39	Human Readable
Codabar	Interleaved 2 of 5 code
Code 11	Inter-Character Space
Code 93	Keyboard Wedge Decoder
Color Scheme	Ladder Code
Minimum Reflectivity Difference	Spectral Band
Mil	Verifier
Stacked Codes	Void
Start/Stop Characters	Wand
TSR	Wide to Narrow Ratio
Verification	X Dimension
Slot Reader	Print Contrast Signal (PCS)
Mis-Read	Print Quality
Module	Quiet Zone
Modulo Check Character(s)	Resolution
Number System Character	Self-Checking
OCR	Serial Decoder
OCR-A	Space
Opacity	Zero Suppression
Picket Fence Code	2-Dimensional Symbology
Decoder	Data Identifier
Demand Printer	—

Table E.4: Servlet Container Domain Concepts

CONCEPTS	**CONCEPTS**
Application event listener	HTTP session
Bean	HTTPS
Client	J2EE
Cookie	J2EE application
Context root	J2EE web tier
Custom tag	JAR
Deployment	JavaBeans
Deployment descriptor	JSF
Dispatcher	JSTL
Document root	JDBC
Filter	JSP action
Front Controller	JSP element
HTTP	JSP expression
HTTP Monitor	JSP page
HTTP response	JSP scripting element
HTTP request	JSP tag
Web module	Web module group
Servlet mapping	Value object
Session	View Creation Helper
Tag	View Mapper
Tag attribute	WAR
TLD	Web application
URI	Web browser
JSP tag library	scriptlet
JSP technology	Server
listener	Server plugin
model object	Servlet
MIME	Servlet container
Scope	Servlet context
Scripting element	Servlet event listener
Scripting variable	Servlet filter
Web component	Web server
Web context	Web container
Web client	XML

Table E.5: Aspect Oriented Programming (AOP) Domain Concepts

CONCEPTS	**CONCEPTS**
Scattering	join point
Tangling	Advice
Crosscutting	Pointcut
inter-type	Introduction
Aspect	Target object
Around	Proxy
Weaving	After
Before	After-returning
After-throwing	—

Table E.6: Basel-II Domain Concepts

CONCEPTS	**CONCEPTS**
financial investment risk	scenario volatility
investment market scenario	scenario volatility rules
investment market risk	scenario correlation
index volatility	scenario correlation rules
index correlation	value rules
correction asset	stand alone (economic capital)
aggregated market value	intra diversified (economic capital)
investment market asset	inter diversified (economic capital)
investment market calculation	legal entity
time attribute	division
lambda factors	sub group
base calculation	business line
current calculation	sender
volatility scenario	receiver
portfolio granularity	inter risk
business line	scenario rule
business unit	holding receiver
legal entity	market value
legal entity receiver	portfolio unit
economic capital	scenario
financial investment pdlgd	asset
financial investment roundup	subgroup sender
public context label	subgroup receiver
correlation scenario	stand alone risk
value scenario	intra diversified risk

Table E.7: Unified Modelling Language (UML) Domain Concepts

CONCEPTS	**CONCEPTS**
Activity Diagram	Hierarchical Statechart Diagram
Action	Include Relationship
Actor	Mealy Machine
Analysis	Method (of a Class or Object)
Association Class	Moore Machine
Association	Object
Attribute (of a Class or Object)	Pane
Responsibility	Sequence Diagram
Scenario	State
System Statechart Diagram	Collaboration
Class	Collaboration Diagram
Class Diagram	Collaborator
System Sequence Diagram	Vision Document
Transition	Waterfall Design Process
UML	Statechart Diagram
Use Case	Stereotypes and Stereotyping
Use Case Diagram	Critic
Use Case Specification	Extend Relationship
Generalization Relationship	—