The Dissertation Committee for Ripon Kumar Saha

certifies that this is the approved version of the following dissertation:

Effective Bug Detection and Localization Using Information Retrieval

Committee:

Dewayne E. Perry, Supervisor

Sarfraz Khurshid

Christine Julien

Milos Gligoric

Premkumar Devanbu

Julia Lawall

Effective Bug Detection and Localization Using Information Retrieval

by

Ripon Kumar Saha, B.S. C.S.; M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2016

Dedicated to my wife Mitu, and my family.

Acknowledgments

First of all, I would like to express my heart-felt and most sincere thanks to my Ph.D. advisor Dr. Dewayne Perry for his constant guidance, advice, encouragement and extraordinary patience during this dissertation work. Over the past four years, he tried his best to train me to be an independent researcher. He is not only a great researcher, teacher, advisor, but also one of the best persons I have ever met. In addition to supervising me with the Ph.D. work, he always went out of his way to help me out in my daily things. A simple example could be that probably I am the only student who even learned how to drive a car from his advisor. I am very fortunate to have him as my advisor.

I would like to give my deepest thanks to Dr. Sarfraz Khurshid. From the very first project of my Ph.D., I collaborated with him. Whenever I needed any help, he was always there. Although officially he is not my co-advisor, in my mind, he is my co-advisor.

I would like to give my sincere thanks to Dr. Julia Lawall. I collaborated with her in a couple of projects. She is a great mentor, and spent a lot of time and energy to help me with various problems. I really enjoyed working with her, and really appreciate all her help during my Ph.D. study.

I am grateful to Dr. Christine Julien. She encouraged me a lot for the bug localization project when she interviewed me for my pre-qualification examination. She also guided me a lot in the early days of my Ph.D. as the Software Engineering Track Advisor.

I am privileged to have an esteemed dissertation committee and would like to thank Dr. Sarfraz Khurshid, Dr. Christine Julien, Dr. Milos Gligoric, Dr. Premkumar Devanbu,

and Julia Lawall for their timely feedback that always gave me a new perspective to my research.

I would like to thank Dr. Matthew Lease from School of Information, The UT Austin, for mentoring me on the bug localization project. The idea of structured retrieval came to my mind when I was taking his Information Retrieval course, and laid the foundation of my dissertation.

I would like to thank my B.Sc. advisor, Dr. Rameswar Debnath, and my M.Sc. advisor, Dr. Chanchal Roy and Dr. Kevin Schneider, for teaching me how to do research, and for motivating and preparing me for a Ph.D.

Thanks to my friends and past colleagues Dr. Baishakhy Ray, Dr. Suman Jana, and Dr. Lingming Zhang, who always helped me in many ways academically and socially. I also collaborated with Lingming in one project.

Thanks to all of my friends and colleagues in the Software Engineering Track with whom I have had the opportunity to grow as a researcher. In particular, I would like to thank Yuqun Zhang, Rui Qiu, Lisa Hua, Divya Gopinath, Allison Sullivan, Nima Dini, Kaiyuan Wang, Alyas Mohammed, Yen Jung Chang, Meiru Che, Kate McArdle, and Sung Min.

I thank the anonymous reviewers for their valuable comments and suggestions in improving the papers produced from this thesis.

I would like to thank all the staff members of the Software Engineering Track who have helped me in one way or another along the way. In particular I would like to thank Cayetana Garcia, and RoseAnna Goewey.

I express my gratefulness to my family members and relatives, especially, my mother Sreelekha Saha, my father Dilip Saha, my mother-in-law Bela Sikder, my father-in-law Sukumar Sikder, my brother Avigit, my brother- and sister-in-laws Shopno, Tithi, and Aparna, my uncles Jibon, Nil Kamal, and Nil Ratan, my aunts Malabika, Swapna, Tapati, and Mukty, my cousins Ananda, Prokash, Mithun, and Pinky, and my friends Pronab, Sabuj, Shashanka, Tushar, Naresh, Emiko, Sudip, Vishma, and Suman who did not get the

share of my time that they deserved.

Finally, I cannot express my gratitude in words to my beloved wife Mitu for sharing this journey with me. This work would not have been possible without her love, support, and encouragement. Wherever I go, or regardless how difficult the time is, she is always there, supporting me unconditionally.

Invariably, acknowledgements always miss someone important. For those that I have not listed explicitly, thank you for being a part of this dissertation and helping me grow as a person and a researcher.

The projects included in my research were funded in part by National Science Foundation (NSF Grant Nos. CCF-1117902 and CCF-0845628).

RIPON KUMAR SAHA

The University of Texas at Austin
May 2016

Effective Bug Detection and Localization Using Information Retrieval

Publication No. _____

Ripon Kumar Saha, Ph.D.
The University of Texas at Austin, 2016

Supervisor: Dewayne E. Perry

Software bugs pose a fundamental threat to the reliability of software systems, even in systems designed with the best software engineering (SE) teams using the best SE practices. Detecting bugs early and fixing them quickly are extremely important. However, they are very expensive and challenging, especially at-scale. While the sciences of bug detection (e.g., software testing) and localization via static and dynamic program analyses have been explored considerably, text-based Information Retrieval (IR) techniques for bug detection and localization are interesting and promising new approaches for these problems. One advantage of text-based approaches is that it can utilize a lot of (implicit) semantic information about a program's functionality from the program text, which is almost impossible to extract using program analysis based techniques.

This dissertation builds a deeper understanding of current bug triaging and fixing processes via mining software repositories, and introduces new techniques for effective bug detection and localization. The dissertation has three main parts. First, we perform a number of empirical studies to investigate the extent of and reasons for long lived bugs, their severities, and time spent in different phases of bug fixing process. We demonstrate that many bugs remain unfixed for inordinate period of time due to numerous reasons, including difficulties in detecting, localizing, and fixing them. Second, we demonstrate that developers use very similar program text in source code and their corresponding test cases, which could be utilized to implement powerful test prioritization techniques. We introduce a novel IR based regression test prioritization technique called REPiR that embodies our insight, and show that REPiR is more efficient than program analysis based or dynamic coverage based techniques. Third, we demonstrate that fine grained program text such as class names, method names, variable names, and comments carry different levels of information, and it can be utilized to improve IR based bug localization. We introduce a structured retrieval technique called BLUiR that embodies our insights and show that BLUiR outperforms the existing state-of-the-art IR-based bug localization approaches. Finally, we further improve BLUiR by natural language processing.

We make four contributions in this dissertation. One, we provide empirical evidence that there are considerable numbers of non-trivial bugs in software projects that survive for a long time. We describe the reasons for delay in fixing, the nature of fixes, and overall fixing process of these long lived bugs in a great detail. Two, we introduce the notion of IR-based regression test prioritization based on program changes. Three, we introduce the notion of structured retrieval for bug localization. Four, we provide an in-depth analysis of the extent to which natural languages processing can play an important role in improving IR-based bug localization further. The central ideas are embodied in a suite of prototype tools. Rigorous empirical evaluation is performed to validate the efficacy of the proposed techniques using datasets containing a variety of real-world Java and C programs.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With software systems in charge of flying planes, running financial markets, and regulating key functions in the human body, the importance of developing dependable and reliable systems has never been greater. Unfortunately, such dependability and reliability is often compromised in practice by the presence of software bugs (also known as "faults"). Such bugs seem inevitable even in software systems designed and evolved with the best software engineering (SE) teams using the best SE practices. The causes of such bugs are plentiful and varied, ranging from Brooks' essential characteristics of software systems to other accidental characteristics, from lack of experience and knowledge to impossible schedules and scarce resources. The consequences range from annoying to disastrous, from lost time to staggering costs.

1.1 Problem Statement

Detecting bugs and localizing them as early as possible is extremely important. However, the processes of bug detection and localization are both challenging and expensive, especially for large software systems. To help developers effectively perform these tasks during software maintenance and evolution, researchers have proposed various techniques and tools to speed up the process. For example, regression test prioritization is a well known

technique to detect regression bugs early in the testing phase [35, 62, 158, 167]; automatic bug localization, depending on the granularity, helps identify buggy source code either at line level or at file level [1, 52, 173], and so on.

Since we are concerned about the current bug fixing process, we performed an empirical study on more than 94,000 fixed bugs in seven popular open source projects to understand their lifetime. We observe that there are a considerable number of *long lived bugs* (bugs surviving more than one year) in each repository, and majority of them are non-trivial. Therefore, we performed more studies to understand the extent and reasons for long lived bugs, their severities, the time spent in different phases of bug fixing process, and nature of bug fixes. One of the important findings of our study is that a bug surviving for a year or more does not necessarily mean that it requires a large fix. In fact, 40% of long-lived bug fixes require only a few changes to one file. The implication is that many bugs persist in software *not* due to the effort required to fix them, but instead due to difficulty in determining *where* source code needs to be fixed. *This also presents an opportunity*: if we could help developers to more easily localize bugs (i.e., find which source code needs to be considered to make a fix), bugs might be corrected significantly faster than today.

While the science of bug *detection* and *localization* via static and dynamic program analysis is a well-studied research area, an exciting new direction of research has begun in investigating the use of text-based Information Retrieval (IR) techniques [18, 47, 77, 82, 90, 114]. Virtually all the human-centric SE documents are text-based, including requirements specifications, architectural prescriptions or descriptions, design documents, code, test scripts, test documents, and test systems. Given such an overwhelming presence of text, IR approaches to test prioritization and bug localization represent a promising and largely unexplored new territory for investigation, providing an opportunity to gain new traction on these old and entrenched problems. By surveying the most up-to-date literature, we identified the following research problems:

1. **Regression test prioritization** (RTP) is a well known technique to expose bugs early

in the testing phase. Although a number of RTP techniques (specifically coverage-based ones) have been widely used, they have two key limitations [95]. First, coverage profiling overhead (in terms of time and space) can be significant. Second, in the context of program changes (that modify behavior significantly) the coverage information from the previous version can be imprecise to guide test prioritization for the current version. Although the static techniques [95, 171] address the coverage profiling overhead, they simulate the coverage information via static analysis, and thus can be also imprecise.

2. **IR-based Bug Localization** is a well known technique to identify the source code that is relevant to a particular bug report, and has gained significant attention due to its relatively low computational cost and minimal external dependencies (e.g., requiring only source code and a bug report in order to operate). Despite the empirical success of prior work, we identify the following limitations.

   (a) Existing IR-based bug localization techniques treat source code as flat text lacking structure. In fact, source code's rich structure distinguishes code constructs such as comments, names of classes, methods, and variables, etc. While ignoring such code structure simplifies the system, it also sacrifices an opportunity to exploit this structural information to improve localization accuracy.

   (b) While previous studies have shown that these IR-based bug localization approaches give good results, a limitation of these studies is that they focus on software written in object-oriented languages, primarily Java. On the other hand, much of the most critical and widely used software, such as operating systems, compilers, and programming language runtime environments, is written in C. Indeed, as of May 2014, C was the most popular programming language according to the TIOBE programming language popularity index [147]. Nevertheless, there is a lack of an established dataset of large-scale, widely used C software, and a lack of easy-to-use tools for manipulating C code. There-

fore, we yet do not know the efficiency of IR-based bug localization tools for C code. Most previous bug localization studies have also acknowledged this limitation [28, 127, 138].

(c) Although bug reports are mostly written in natural English and natural language processing (NLP) is concerned with the interactions between computers and human (natural) languages, we still lack the empirical knowledge whether or how we can leverage the NLP techniques to improve IR-based bug localization.

1.2 Theses

Based on several empirical investigations, we believe that the limitations identified in the area of regression test prioritization and bug localization could be mitigated by either taking additional information, which is already available in the software repository, into account or introducing new techniques. Our theses are as follows:

1. We observe that program test cases have a strong relationship with the source code under test in terms of textual similarity. We can effectively leverage this relationship to enable IR-based regression test prioritization technique based on program changes.

2. We observe that different program constructs (e.g. classes, methods, and variables) carry different information and thus their importance level is not also the same. We can effectively leverage these program constructs to enable more accurate IR-based bug localization. Furthermore, natural language processing may play an important role to improve the approach even further. We also hypothesize that IR-based bug localization is not only effective in Java programs, but also may be effective in C programs.

1.3 Solution

Our solution involves three major steps: i) a set of empirical studies that enrich current knowledge regarding the overall bug fixing process, especially for long lived bugs (i.e., bugs that lived in a program more than one year), ii) the design and implementation of two IR-based approaches: REPiR and BLUiR for regression test prioritization and bug localization respectively, and iii) rigorous experimental evaluation of REPiR and BLUiR on real world software projects.

1.3.1 Bug Repository Analysis

The first part of our thesis is to gain more understanding about software bugs and their fixing processes overall. To this end, we performed two empirical studies:

An Empirical Study of Long Lived Bugs: One of the main objectives of this study is that we would like to know the extent and reasons for long lived bugs (bugs that survived more than one year in a system). There are quite a few studies that investigated the overall factors related to bug fix time. However, if we automatically analyze all the bug reports using a standard data mining technique, it is highly likely that the main factors behind long lived bugs will get lost due to the well-known "imbalanced dataset" problem. Therefore, in this dissertation, we conduct an exploratory study focused solely on long lived bugs in seven popular open source projects. We analyzed them from five different perspectives: their proportion, severity, assignment, reasons for delay in fix, and the change effort of fixes. To this end, we extracted all the bug reports and their detailed information such as opening time, assignment time, fixing time, and so on, from the associated bug repositories. Then we only analyzed those bugs that were eventually fixed to make sure that all the analyzed bugs are valid. Furthermore, we removed all the duplicate bug reports so that we do not overestimate long lived bugs. To assess the importance of long lived bugs, we analyzed the severity field of the bug repository and the number of duplicated bugs for a given bug report. To investigate the reasons for delay in fixing, we manually analyzed a random sample of

long lived bugs. Finally to investigate the change effort of bug fixes, we identified the bug fixing commits, where it was possible. Then we used the change of lines, hunks, and files metrics to get an approximate idea about change effort.

Are These Bugs Really "Normal": There is a widespread confusion regarding the severity level tagged in bug repository, especially for "normal" bugs. Since the primary objective of the severity field is to express the importance of bugs and the majority of bugs in any bug repository are "normal", in this dissertation, we investigate the extent to which "normal" bug reports actually have the "normal" severity. To this end, we designed an approach based on manual investigation. First, we took a random sample of bug reports that are labeled as "normal" in the bug repository. Then each bug report was assessed independently by two assessors. Assessors gave us their opinion about the actual severity of those bug reports and the rationale for their decision. If the two assessors had different opinion about a given bug report, we analyzed both the report and the assessments to make a decision.

1.3.2 An Information Retrieval Approach for Regression Test Prioritization Based on Program Changes

We introduce a new IR-based based approach, REPiR, to address the problem of regression test prioritization. Our key insight is that in addition to writing good identifier names and comments in the code, developers use very similar terms for test cases, and we can utilize these textual relationships by reducing the RTP problem to a standard IR problem such that program changes constitute the query and the test cases form the document collection. Our tool REPiR embodies our insight. We build REPiR on top of the state-of-the-art Indri [140] toolkit, which provides an open-source, highly optimized platform for building solutions based on IR principles. An empirical evaluation using eight open-source Java projects shows that REPiR is computationally efficient and performs better than existing (dynamic or static) techniques for the majority of subject systems.

1.3.3 Improving Bug Localization using Structured Information Retrieval

Our key insight in this study is that structured information retrieval based on Java code constructs, such as class and method names, enables more accurate bug localization. We present BLUiR, which embodies this insight, requires only the source code and bug reports. We build BLUiR on a proven, open source IR toolkit. Our work provides a thorough grounding of IR-based bug localization research in fundamental IR theoretical and empirical knowledge and practice. We evaluate BLUiR on four open source projects with approximately 3,400 bugs.

1.3.4 On the Effectiveness of Information Retrieval Based Bug Localization for C Programs

In this study, we create a benchmark dataset consisting of more than 7,500 bug reports from five popular C projects and rigorously evaluate our recently introduced IR-based bug localization tool using this dataset. We also adapted our tool BLUiR so that it leverages the structured retrieval for C programs as well.

1.3.5 Natural Language Processing to Improve IR-based Bug Localization: An Exploratory Study

In this study, we investigate the extent to which natural language processing of bug reports can improve the accuracy of the core IR model for bug localization. In particularly, we investigate how the variations in words due to parts of speech, synonyms, and tenses affect bug localization. Then we incorporate the most effective of these techniques into our bug localization tool BLUiR to improve it further.

1.4 Contributions

This dissertation makes the following contributions.

- **Empirical knowledge about long lived bugs:** Our work is the first comprehensive analysis of long lived bugs. Our study finds that there are a considerable number of bugs in well known, widely used software projects that survived more than one year, and these bugs are not trivial. The reasons for the long life of these bugs are diverse including long assignment time, not understanding their importance in advance, difficulties in localization, etc. So we need better tool support to speed up the bug fixing process.

- **Actual severity of bugs:** Although there is a widespread confusion regarding the severity level tagged in bug repository, before our study the proportion of misclassifications was unknown. Our study first gives concrete statistics that many bugs, tagged as "normal" in the bug repository, are not actually "normal" and 25% of them are actually severe.

- **REPiR:** We introduce a new approach for regression test prioritization (RTP) based on program changes. We define a reduction from the regression test prioritization problem to a standard information retrieval problem and present our approach, REPiR, based on this reduction. We embody our approach in a prototype tool that leverages the off-the-shelf, state-of-the-art Indri toolkit for information retrieval. We present a rigorous empirical evaluation using version history of eight open-source Java projects and compare REPiR with 10 RTP strategies. We also present different variants of REPiR and provide detailed results on how REPiR can be used more effectively depending on test or program differencing granularities.

- **BLUiR:** We present a new technique for increasing bug localization accuracy, particularly by modeling the source code structure. We also present a new state-of-the-art accuracy for bug localization on a public community benchmark. We built our technique on a proven, open source IR toolkit. Based on an evaluation with four open source Java projects with approximately 3,400 bugs, we show that BLUiR achieves

better accuracy than the state-of-the-art bug localization tool.

- **Effectiveness of IR-based bug localization for C systems:** We introduce a new dataset consisting of more than 7500 bug reports with their location in the source code at file level and function level for C programs and a prototype to localize bugs in C systems. We produce more generalizable results than the previous state-of-the-art evaluations on the effectiveness of IR-based bug localization.

- **Role of NLP in bug localization:** We provide an in-depth analysis of the extent to which natural languages processing can play an important role in improving IR-based bug localization further. Our results show that bug summary and bug description based term weighting and POS-based term weighting individually improve the results considerably. On the other hand, expanding queries using synonyms, which showed promising results for code search, is not very effective for bug localization. We have incorporated our findings into BLUiR, which improves the state-of-the-art accuracy by 10%.

In summary, we believe our results and tools will give researchers and developers more insight about the current bug fixing process and will advance the current practice in the area of bug detection and localization.

1.5 Organization

The rest of the proposal is organized as follows.

In Chapter 2, we provide the necessary background for this work.

In Chapter 3, we briefly describe our study on long lived bugs.

In Chapter 4, we investigate whether the bugs tagged as normal in the bug repository are actually normal.

In Chapter 5, we introduce a new approach for regression test prioritization that is information retrieval based, and is based on program changes.

In Chapter 6, we introduce a structured retrieval approach to improve information retrieval based bug localization.

In Chapter 7, we investigate whether IR-based bug localization is effective for C programs.

In Chapter 8, we investigate whether or how natural language techniques improve information retrieval based bug localization.

Finally, in Chapter 9, we conclude our dissertation.

Chapter 2

Background

This chapter provides background information relevant to the research of this dissertation. Since this dissertation is concerned with understanding the current bug fixing process in more detail, and with introducing or improving techniques related to test prioritization and bug localization, we begin by describing bug tracking systems and the overall bug fixing process. Then, we describe more specific techniques, such as regression test prioritization and automatic bug localization.

2.1 Bug Tracking System

Generally project stakeholders maintain a bug database for tracking all the bugs associated with their project. Several online bug tracking systems are available, such as Bugzilla,[1] JIRA[2], Mantis[3], etc. Different repositories may have different data structures and follow different life cycles of bugs. The dataset we created and used in our work was extracted from Bugzilla, a popular online bug tracking system. Therefore, the rest of the discussion regarding bug tracking systems is limited to Bugzilla.

[1] https://www.bugzilla.org/

[2] https://www.atlassian.com/software/jira

[3] https://www.mantisbt.org/

Any person having legitimate access to a project's bug database can post a change request through Bugzilla. A change request could be either a bug or an enhancement. In Bugzilla, however, both bugs and enhancements are represented similarly and are referred to as bugs, with the only difference being that for an enhancement the severity field is set to "enhancement". Generally bug reporters provide a bug summary, bug description, the affected product, and the component name with the bug severity.

Bugzilla allows the developers of a particular project to define their own severity levels. According to the Eclipse Bugzilla documentation,[4] the severity level can be one of the following values, which represent the degree of potential harm.

`Blocker`: These bugs block the development and/or testing work. There exists no workaround.

`Critical`: These bugs cause program crashes, loss of data, or severe memory leaks.

`Major`: These bugs result in a major loss of function.

`Normal`: These are regular issues. There is some loss of functionality under specific circumstances.

`Minor`: These bugs cause minor loss of functionality, or other problems where an easy workaround is present.

`Trivial`: These are generally cosmetic problems such as misspelled words or misaligned text.

The developers of WineHQ also follow the same severity levels. However, the GDB community recognizes three levels of severity: critical, normal, and minor. On the other hand, the Linux community has their own severity level: blocking, high, normal, and low.

In addition to providing the severity level, reporters also specify the software version, the platform and operating system where they encountered the bug so that developers

[4] http://wiki.eclipse.org/Eclipse/Bug_Tracking

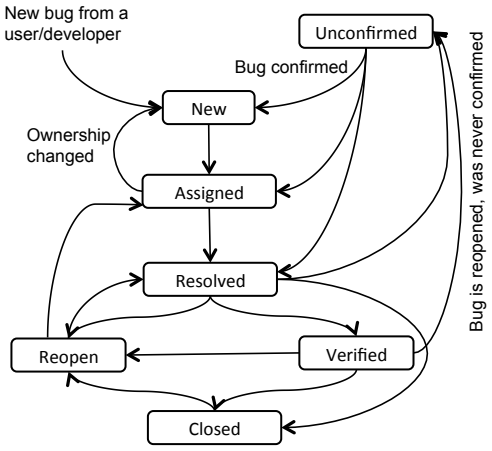

Figure 2.1: Life Cycle of a Bug in Bugzilla

can productively attempt to reproduce it. Bug reporters can also attach files to the bug report such as screen shots, failing test cases etc. Once a bug is posted, all other related developers can make comments regarding the bug to discuss different issues. Therefore, a bug repository contains a rich set of information that can be analyzed to gain insight about bugs.

2.2 Bug Life Cycle

The overall bug fixing process in a system is directly related to the bug life cycle maintained by the bug tracking system. Although different projects may have different strategies for using Bugzilla, a common life cycle for a bug is as follows: [5]

Validation: Each project/component team leader triages `NEW` bugs to verify if the bug is really a bug and if the provided information is correct. In case of any inconsistencies, the bug triager can correct them. The bug triager also can request further information to validate a bug if it is necessary. If there is no response within a week, the team leader closes the bug marking it as `RESOLVED`, `INVALID`, or `WONTFIX`. However, the reporter

[5] http://wiki.eclipse.org/Development_Resources/HOWTO/Bugzilla_Use

can reopen the bug anytime if she has more information.

Prioritization: In this stage, the triager first determines whether a bug is a feature request. If so, the triager changes the severity of the bug to `enhancement`. Otherwise, she checks the severity level of the bug to make sure that it is consistent with the bug description. Then, the priority of the bug is set based on following guidelines: [6]

`P1`: These bugs are a must fix for the indicated target milestone.

`P2`: These bugs are very important for the indicated target milestone. Generally developers try to resolve all the `P2` bugs.

`P3`: These bugs are normal and thus labeled as the default priority. Furthermore, if the bug triager is uncertain about the priority of a bug or if it is actually a normal bug, she can set the priority to `P3`. Then the assigned developer can adjust it if appropriate.

`P4`: These bugs should be fixed if time permits.

`P5`: These are valid bugs, but there are no specific plans to fix them. The `P5` priority also indicates that help is wanted.

Fixing: At this point, a bug remains in the component's "inbox" account until a developer takes the bug, or the team leader assigns it to a developer. After fixing the bug, the developer marks it as `RESOLVED-FIXED`.

Verification: Once a bug is fixed, it is assigned to another committer on the team to verify. Ideally, all bugs should be verified before the next integration build. Once the verifier tests that the bug is completely resolved, she changes the bug status to `VERIFIED`. Figure 2.1 represents all possible state transitions of a bug in Bugzilla.

2.3 Regression Test Prioritization

Regression testing is a widely used methodology for validating program changes. More specifically, whenever a change is made to a program, the existing test suite is run to make sure that the new change has not broken any existing functionalities. However, regression

[6] http://wiki.eclipse.org/WTP/Conventions_of_bug_priority_and_severity

testing can be time consuming and expensive [10, 74]. Executing a single regression suite can take even weeks for some software [122]. Regression testing is even more challenging in continuous or short-term delivery processes, which are now common practices in industry [53]. Regression test prioritization (RTP) is a widely studied technique that ranks the tests based on their likelihood of revealing faults. RTP defines a test execution order based on this ranking so that tests that are more likely to find (new, unknown) faults are run earlier [35, 95, 158, 167, 171].

Rothermel et al. [122] formally defined the test case prioritization problem as finding $T' \in PT$, such that $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$. In this definition, PT denotes the set of all possible permutations of a given test suite T, and f denotes a function from any member of PT to a real number such that a larger number indicates better prioritization. In this dissertation, we focus on regression test prioritization (RTP), which is basically a specific case of general test prioritization. RTP increases the likelihood of revealing regression errors related to specific code changes earlier in the regression testing process.

Existing RTP techniques are largely based on dynamic code coverage, where the coverage from the previous program version is used to order, i.e., *rank*, the tests for running against the next version [35, 62, 158, 167]. A few recent techniques utilize static program analysis in lieu of dynamic code coverage [95, 171]. RTP techniques (whether dynamic or static) are broadly classified into two categories, *total* or *additional*, depending on how they calculate the rank [122]. *Total* techniques do not change values of test cases during the prioritization process, whereas *additional* techniques adjust the values of the remaining test cases taking into account the influence of the already prioritized test cases.

2.4 Automatic Bug Localization

Generally, bug fixing starts with finding relevant buggy source code, i.e., bug localization. However, performing this process manually for many bugs is time consuming and expen-

sive. Therefore, effective methods for locating bugs automatically from bug reports are highly desirable. There are two general approaches for bug localization: i) dynamically locating the bug via program execution together with such technologies as execution and data monitoring, breakpoints etc. [1]; and ii) statically locating bugs via various forms of analyses using the bug reports together with the code [52]. The dynamic approach is often time consuming and expensive.

2.5 Information Retrieval

Information retrieval (IR) is concerned with finding relevant documents (material) from within a (large) collection of documents based on a query, which is basically the information need. Although an IR system can be designed to retrieve a variety of documents such as, text, audio, videos, and so on, in this dissertation, we limit our discussion to text-based automatic IR systems, where both the documents and queries are text. An IR system typically begins with three-step preprocessing phase, performing *text normalization*, *stopword removal*, and *stemming*. Normalization involves removing punctuation, performing case-folding, tokenizing terms, etc, ultimately defining the initial vocabulary in which queries and documents will be represented. Next, a set of extraneous terms identified in a stopword list (e.g., "to", "the", "be", etc.) are filtered out in order to improve efficiency and reduce spurious matches. Finally, stemming conflates variants of the same underlying term (e.g., "ran", "running", "run") to improve term matching between query and document.

While these three preprocessing steps are often given short shrift in describing IR approaches, they embody important tradeoffs that can significantly influence the ultimate success or failure of the retrieval model. For example, normalization can increase matches between query and document by case-folding (improving recall), but this can also introduce spurious matches (hurting precision). Similarly, while stopword removal can reduce unhelpful term matching (e.g., "to"), any stopword removed is almost certain to hurt matching for some particular query (e.g., "to be or not to be"). Finally, stemming will similarly

increase recall by conflating variants of the same underlying term, but this may also introduce false matches. For reproducible experimentation, preprocessing methods should be fully described along with other details of the IR model formulation.

Once queries and documents have been pre-processed, documents are *indexed* by collecting and storing various statistics, such as *term frequency* (TF, the number of times a term occurs in a given document), and *document frequency* (DF, the number of documents in which the term appears). IDF refers to inverse (dampened) DF, most simply formulated as $log(\frac{N}{DF})$, where N is the number of documents in the collection. For a broad overview of IR, see [86] online.

2.6 Information Retrieval Techniques in Software Engineering

In recent years, IR techniques have been applied to over two dozen different software engineering problems, many of which are highlighted in two surveys on the application of IR to SE problems [16, 17]. There are two predominant tasks today. The first task is feature (or concept) location, which consists of locating features described in maintenance requests, such as enhancements or faults [18, 47, 77, 82, 90, 114]. The second task is traceability, which links or recovers links between software engineering artifacts [81, 89]. Another closely related task is software reuse, where IR is used to identify the reusable software artifacts [39]. There are also a diverse set of other tasks such as quality assessment [72], change impact analysis in source code [25], restructuring and refactoring [9], defect prediction [15], clone detection [88] and duplicate bug detection [141].

Chapter 3

An Empirical Study of Long Lived Bugs

This chapter is based on our paper, "An Empirical Study of Long Lived Bugs", published in the Proceeding of the IEEE CSMR-18/WCRE-21 Software Evolution Week [123]. This paper was also invited to a journal as one of the best papers in that conference. Therefore, the extended version of the paper, "Understanding the triaging and fixing processes of long lived bugs" is published in Information and Software Technology [124].[7]

3.1 Context and Problem Statement

Although developers and testers try their best to make their software error free, in practice software ships with bugs. The number of bugs in software is a significant indicator of software quality since bugs can adversely affect users' experience directly. Therefore, developers are generally very active in finding and removing bugs.

To ensure high software quality for each release, developers try to fix bugs very actively. However, there are still many bugs that live for a long time. We believe the impact of these *long lived* bugs (for our study, bugs that are not fixed within one year after they are reported) is even more critical since the users may experience the same failures version

[7]Please note that Dr. Sarfraz Khurshid and Dr. Dewayne Perry are the co-authors of this paper. They both helped me design the empirical study and improve the presentation of the paper.

after version. Therefore, it is important to understand the extent and reasons of these long lived bugs so that we can improve software quality.

A number of previous studies have investigated the overall factors affecting bug fix time. Giger et al. [41] empirically investigated the relationships between bug report attributes and the time to fix. Zhang et al. [166] predicted overall bug fix time in commercial projects. Canfora et al. [24] used survival analysis to determine the relationship between the risk of not fixing a bug within a given time frame and specific code constructs changed when fixing the bug. Zhang et al. [165] examined factors affecting bug fixing time along three dimensions: bug reports, source code involved in the fix, and code changes that are required to fix the bug.

While these studies are useful in understanding the overall factors related to bug fix time, we know of no study that has specifically investigated long lived bugs to understand why they take such a long time to be fixed and how important they are. We point out that analyzing entire bug datasets using various machine learning or data mining techniques (as done in previous work) is not sufficient to understand long lived bugs due to the imbalanced dataset.[8] Imbalanced datasets are a major problem in most data mining applications since machine learning algorithms can be biased towards the majority class due to over-prevalence [55]. We expect (and our results also support) that the proportion of long-lived bugs would be lot less than 50% of the total bugs, thus resulting an imbalanced dataset. Therefore, if we automatically analyze all the bug reports using a standard data mining technique, it is highly likely that the main factors behind long lived bugs will get lost. In this dissertation, we conduct an exploratory study focused solely on long lived bugs to understand their extent and reasons with respect to following research questions:

1. **What proportion of the bugs are long lived?** The answer to this question is important since if there are few long lived bugs, there may be little reason to worry.

2. **How important are long lived bugs in terms of severity?** It is important to under-

[8] A dataset is imbalanced if the classification classes are not approximately equally represented.

stand how crucial these bugs are. If they are minor or trivial bugs, their impact would be less on overall software quality.

3. **Where is most of the time spent in the bug fixing process?** The answer to this question is important to identify the time consuming phases so that developers as well as researchers can work on improving the processes involved in this phase.

4. **What are common reasons for long lived bugs?** To make the bug fixing process faster, first we need to understand the underlying reasons for delays. Delineating the common reasons for long lived bugs will help researchers deal with the problem more systematically.

5. **What is the nature of fixes of long lived bugs?** The answer to this question will help us in better understanding the bug fixing process, estimating change efforts, and so on, which will be useful in exploring potential approaches for improving overall bug fixing processes.

3.2 Study Setup

This section provides a brief description of the subject systems that we studied, and the metrics and process we used to understand the extent and reason of long lived bugs.

3.2.1 Subject Systems

We use seven open source projects for our study. Among them, we choose four projects from the Eclipse product family, namely, JDT, CDT, PDE, and Platform, which are written in Java programming language. The other three projects are the Linux Kernel, WineHQ, and GDB, which are written in C programming language. There are three main reasons for choosing these projects. First, these projects are highly successful and have been widely used in software engineering research. Second, each project has a long development history. Third, these projects are from different domains.

- JDT and CDT provide a fully functional Integrated Development Environment based on the Eclipse platform for developing Java, and C and C++ applications. [9,10]
- The Plug-in Development Environment (PDE) provides tools to create, develop, test, debug, build and deploy Eclipse plug-ins, fragments, features, update sites and RCP products. [11]
- The Eclipse Platform defines the set of frameworks and common services that collectively make up infrastructure required to support the use of Eclipse.[12]
- Linux Kernel: The kernel of the Linux operating system.[13]
- WineHQ: A compatibility layer, making it possible to run Windows applications on POSIX compliant operating systems.[14]
- GDB: A debugger for programs written in C, C++, and many other programming languages.[15]

We have created the dataset for C projects. This dataset includes all the bug reports and their histories from their inception to May 2014. For Java projects, we have used Lamkanfi et al's [71] bug dataset to extract the bug information. The Java dataset includes all the bug reports and their histories from their inception to March 2011 for these four projects (extracted from Eclipse Bugzilla database). More detailed descriptions of the dataset is presented in Table 7.3.1. In the Table, the last row represents the number of bugs (excluding enhancement and duplicated bugs) that got eventually fixed, which is the actual dataset of this study.

[9] http://projects.eclipse.org/projects/eclipse.jdt
[10] http://www.eclipse.org/cdt/
[11] http://www.eclipse.org/pde/
[12] https://projects.eclipse.org/projects/eclipse.platform
[13] https://www.kernel.org/
[14] http://www.winehq.org
[15] http://www.sourceware.org/gdb/

Table 3.1: Data Set

System	#CR	#Bugs	# Enh.	# Bug Fixed
JDT	46,308	38,520	7,788	18,873
CDT	14,871	12,854	20,17	7,260
PDE	13,677	11,958	1,719	6,854
Platform	90,691	78,120	12,571	33,738
Linux	23,618	23,387	231	5,784
WineHQ	36,691	34,490	2,201	14,338
GDB	17,038	15,562	1,476	7,667

3.2.2 Terms and Metrics

We make use of bug tracking and version control systems' information to calculate metrics that we were interested in. This section defines different terms and metrics that we use in the rest of the chapter.

Bug Introduction Time (T_I): This is the timestamp when the buggy code is committed for the first time for a given bug.

Bug Reporting Time (T_R): This is the timestamp when a bug is reported to the Bugzilla system by a user/developer.

Bug Assignment Time (T_A): This is the timestamp when a bug was officially assigned to the right developers through Bugzilla. If a bug is assigned to multiple developers, we use the assignment time of the developer who fixed the bug. If a bug is fixed by multiple developers, we use the assignment time of the developer who committed the last changes.

Bug Severity Realization Time (T_S): This is the timestamp when the actual severity of a given bug was understood by the developers and thus the severity field of that bug was changed for the last time.

Bug Fix Time (T_F): This is the timestamp when a developer officially marked a bug as `FIXED` in Bugzilla through the `resolution` field.

Bug Assignment Period (AP): This is the lapse time between when the bug was opened and when it was assigned to the right developer. Mathematically, $AP = T_A - T_R$

Bug Fixing Period (FP): This is the period of time that developers took to fix a bug. It should be noted that it is not the actual coding time of the bug-fix. Instead, it is the time period between the bug assignment time and the bug fix time. Mathematically, $FP = T_F - T_A$. It should be further noted that we do not deduct the time when a bug is temporarily closed. A bug is temporarily closed when the developers think that the bug is fixed but actually it is not. Therefore, we do not deduct that time, since the bug is still (at least partially) present during the time when the bug reports was closed.

Pre-Severity Realization Period (Pre-SRP): This is the period of time developers took to understand the actual severity of the bug. Therefore, pre-severity realization time is the time between bug reporting time and the time when the severity was changed for the last time. Mathematically, $Pre\text{-}SRP = T_S - T_R$.

Post-Severity Realization Period ($Post$-SRP): This is the time developers took to fix the bug after realizing the actual severity time. Mathematically, $Post\text{-}SRP = T_F - T_S$.

Bug Verification Period (VP): This is the period of time that a developer took to verify a bug after it is marked as `FIXED` in Bugzilla. Mathematically, $VP = T_V - T_F$.

Bug Survival Period (SP): This is the period that a bug exists in the system. Ideally it should be the time period between the bug introduction time (T_I) and bug fixing time (T_F). However, in our study it is the time period between T_R and T_F. The timestamps of bug introduction (T_I) and bug reporting (T_R) can be certainly different, since a bug can remain dormant for a long time [27]. Although there are some algorithms [65] to identify bug introducing changes, it is difficult to map those changes to the associated bug reports. Therefore, we preferred bug reporting time over bug introduction time. Furthermore, since T_R is always later than T_I (i.e., a bug is always reported after the bug introducing changes are committed), our calculated bug survival period (SP) never overestimates the actual SP. However, we do not subtract the time period from SP when a bug was temporarily closed. Figure 3.1 visually presents all the terms and metrics in a timeline.

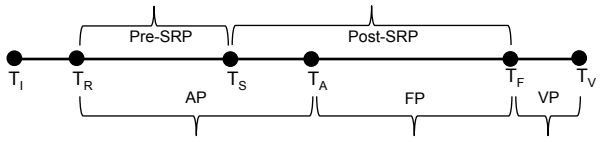

Figure 3.1: An Example Timeline of a Bug

```
commit 768b107e4b3be0acf6f58e914afe4f337c00932b
Date:   Fri May 4 11:29:56 2012 +0200

    drm/i915: disable sdvo hotplug on i945g/gm

v2: While at it, remove the bogus hotplug_active read,
and do not mask hotplug_active[0] before checking
whether the irq is needed, per discussion with Daniel
on IRC.
Bugzilla:
https://bugzilla.kernel.org/show_bug.cgi?id=38442
```

Figure 3.2: A bug fixing commit for #38442 in Linux Kernel

3.2.3 Identification of Faulty Source Code

Previous studies [63] showed that when developers fix bugs they often put the bug id in their commit message. Therefore, to get the version histories and commit messages of these four projects, first we accessed their git repositories. Then using JGit APIs, we extracted all the commit messages from the histories and searched all numbers.[16,17,18,19] Then we matched each number with the bug IDs. To further ensure that those are indeed bug IDs, we only accepted those commits that contain additional information. For example, in Java projects, the term `bug(s)` (case insensitive) was present. In the Linux Kernel, we found that developers referred to the Bugzilla URL (as shown in Figure 3.2), whereas in GDB the bug was referred by the term `PR`. In this way, we reduced the chance of getting false positives, although we might missed some true mappings.

[16] http://www.eclipse.org/jgit/

[17] git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

[18] git://source.winehq.org/git/wine.git/

[19] git://sourceware.org/git/binutils-gdb.git

For one of our considered projects, WineHQ, however, the above process gave no results. We thus consulted with a developer from the WineHQ community who informed us that in this community the convention is for the bug report to refer back to the commit, rather than the commit referring to the bug report. Indeed, in the WineHQ Bugzilla, there is a dedicated field for a git commit id. However, many of these fields were empty since the field is not required. In this way, we identified the bug fixing commits for those long-lived bugs, where the information was available in the commit messages. Then we used git *diff* to compute following metrics for bug fixes:

Number of Changed Files: It is the number of files that were changed in the bug fixing commit. If a bug was fixed in multiple commits, it is the total number of distinct files in all commits.

Number of Hunks: A hunk is a chunk of adjacent lines that was changed. For a bug fix spanning over multiple commits, it is total number of hunks in all commits. This is useful to understand how many times developers had to move around in the code to fix a bug.

Code Churn: This is the total number of changed lines. Since we use git *diff* itself, the changes in comments were counted as well. For multiple commits, it is the total number of changed lines in all commits. It should be noted that if a line is changed, it is considered as a line deletion first and then addition of another line. Thus the value of code churn for a line change is two.

3.3 Analysis and Results

In this section, we present our analysis and the experimental results which answer our research questions.

3.3.1 RQ1: What proportion of the bugs are long lived?

Motivation. The first question of any empirical study is how large is the population that we want to study. The answer is important since if the population is small, there may be little reason to worry about them. In this cases, our population of interest is long lived bugs.

Long Lived Bugs. The definition of long lived bugs is subjective since the time threshold for deciding whether a bug is long lived or short lived could vary across projects, persons, or studies. In this research question, we report the survival time of all the fixed bugs in each subject system and define the long lived bugs more concretely for our study.

Although many of us believe that a bug could be considered as long lived if it survives more than six months, in this study we have considered only those bugs as long lived that survive more than one year. There are two main reasons behind this decision. First, the average release cycle length of most of the subject systems considered vary from nine months to one year. Second, we wanted to be more conservative so that we can investigate really long lived bugs. Therefore, if a bug was not fixed in one year, it is expected that the bug propagated through at least two major releases. And it would not be a pleasant experience for a user if s/he experiences the same bug in subsequent major versions of a software.

Methodology: To investigate long lived bugs, we first calculated the survival time of each bug as described in Section 3.2.2. Then we divided the bug survival time into six categories: bugs fixed within one day, one week, one month, six months, one year, and more than one year. Finally, we count the number of bug reports in each category and compute the proportion to understand the overall distribution of bug survival time in the dataset.

Used Metric. Survival time (SP).

Results. Results presented in Table 3.2 show that around 50%(+/-4%) of the total (fixed) bugs in Java projects were fixed within a week. In C projects, the bug fixing rate was slower than that of Java projects; it took a month to fix around 50% of the total bugs (except WineHQ). This indicates that even in open source projects, developers are active in

Table 3.2: Bug Fix Time

Time	JDT		CDT		PDE		Platform		Linux		WineHQ		GDB	
	#Bugs	[%]	#Bugs	[%]	#Bugs	[%]	#Bugs	[%]	#Bugs	[%]	#Bugs	[%]	#Bugs	[%]
<1 day	5,009	26.54	1,911	26.32	2,154	31.43	8,244	24.44	578	9.99	673	4.69	1,591	20.75
1-7 days	4,788	25.37	1,485	20.45	1,570	22.91	7,420	21.99	881	15.23	1,809	12.62	1,470	19.17
8-30 days	3,704	19.63	1,230	16.94	1293	18.86	6,442	19.09	1,221	21.11	1,656	11.55	1,358	17.71
1-6 mon.	3,604	19.10	1,426	19.64	1,212	17.68	7,101	21.05	1,573	27.2	2,837	19.79	1,654	21.57
6-12 mon.	855	4.53	567	7.81	353	5.15	2,173	6.44	578	9.99	2,062	14.38	511	6.66
>1 year	913	4.84	641	8.83	272	3.97	2,358	6.99	953	16.48	5,301	36.97	1,083	14.13
Total	18,873		7,260		6,854		33,738		5,784		14,338		7,667	

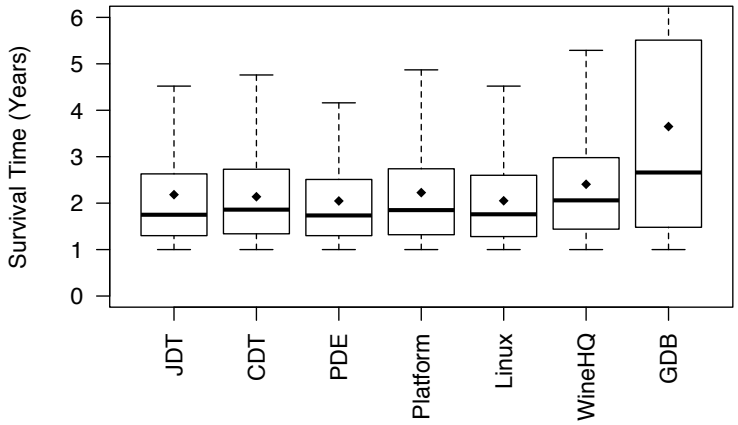

Figure 3.3: Suvival Time of Long Lived Bugs

fixing bugs. However, as the results show, 10% to 17% of bugs in Java projects and 20% to 50% of bugs in C projects took more than six months to be fixed.

Finally, we found that even after considering such a conservative definition, there are more than 4,184 and 7,337 long lived bugs in the Java and C projects respectively. It should be noted that all these bugs eventually got fixed. Therefore, they all are valid bug reports. We believe these are large numbers and thus it is important to investigate them quantitatively and qualitatively.

Figure 3.3 presents the survival time distribution of long lived bugs. It shows the average (dot in the box), median (line in the box), upper/lower quartile, and 90th/10th percentile of survival time. We limit the values of the Y axis to 6 years to better represent

the figure. From the figure, we see that at least 25% (upper quartile) of long lived bugs took more than 2.5 years to be fixed. For GDB, it is close to 5.5 years.

Among the total number of bugs that developers fixed, 5%-9% in Java projects and 14%-37% in C projects took more than one year to be fixed.

3.3.2 RQ2: How important long lived bugs are in terms of severity?

Motivation. In this research question, we understand how crucial these long-lived bugs were from the perspective of both developers and users. If they are minor or trivial bugs, their impact would be less on the overall software quality.

Methodology. There are two fields in Bugzilla that indicate the importance of a bug: i) severity and ii) priority. However, based on their usage, severity is more important than priority to understand the importance, since severity represents the degree of the impact of the bug on the operation of the system. On the other hand, priority often describes the relative work schedule of fixing a bug set by the developers for a given milestone. For example, if there are 10 critical bugs in a system but developers have time to fix only five bugs, they can set higher priority to any five bugs based on some consideration and set a relatively lower priority to others. Sometimes, developers can set high priority to even a less severe bug, if it is expected to be fixed more easily than a critical bug. Therefore, for this research question, we emphasize on severity over priority. Our initial hypothesis was that *most of the long lived bugs are either minor or trivial, which do not have serious effects.*

To understand the overall distribution of bug reports in terms of severity level, we extract the severity field information from bug repository and count the number of bug reports in each category.

As a part of this research question, we also investigated how long it takes to understand the severity of the bugs. Our initial hypothesis was that *perhaps it took long time to realize the severity of these important (`critical` or `major`) bugs. But once the severity was realized it should not take long time to fix them since they are important problems.*

Table 3.3: Importance of Long Lived Bugs

System	Blkr.	Critical	Major	Normal	Minor	Trivial
JDT	1	5	49	712	119	27
CDT	2	4	61	523	37	14
PDE	0	6	20	224	13	9
Platform	6	42	221	1856	170	63
Linux	32	130	N/A	743	48	N/A
WineHQ	24	39	185	4200	632	221
GDB	N/A	53	N/A	977	53	N/A

For this analysis, we have considered only those bugs that have severity level of `major` or higher because they are the most important ones.

Finally, we understand the importance of bug reports by investigating the number of duplicate bug reports. Severity is certainly the most reliable information to understand the importance of a bug since it is determined by the bug reporters and supported by developers. However, a large number of duplicate bugs also may express their importance since they often indicate that the scope of the master bug is large and/or the affected users/other developers are getting frustrated [12].

Used Metrics. The distribution of bug reports in terms of severity, Pre Severity Realization Period ($Pre\text{-}SRP$), Post Severity Realization Period ($Post\text{-}SRP$), and the distribution of bug reports in terms of duplicate bug reports.

Bug Severity

Table 3.3 presents the importance of long lived bugs based on their severity. Our results show that almost 90% of the long lived bugs have severity level of `normal` or above both for Java and C projects. Project-wise the proportion varied from 84% to 95%. According to the Eclipse Bugzilla documentation, only `minor` and `trivial` bugs do not interfere with normal work or use, which means that any bugs having severity level `normal` and above adversely affect user experiences. Taking that information into account and assuming similar interpretation applies to C projects, we believe that the delay in the long bug fixing

Table 3.4: Analysis of Severity for Critical and Major Bugs

System	# Bugs	Severity Changed	Proportion	Maximum Changed
JDT	54	23	42.59%	3
CDT	65	21	32.31%	4
PDE	26	11	42.31%	3
Platform	263	115	43.73%	5
Linux	152	14	8.64%	1
WineHQ	248	68	27.42%	3
GDB	53	10	18.89%	1

process was not due to the fact that they were trivial.

Now let us a take closer look into more severe bugs: `critical` and `major` (`blocker` bugs generally do not interfere users directly). Our results show that for Java projects, only 1% to 2% of long lived bugs were `critical`, whereas 5% to 10% of long lived bugs were `major` in each system. The absolute number ranged from 4 to 42 for `critical` and 20 to 221 for `major` bugs. For C projects, there are 463 bugs in total that are either `major` or above.

Considering that a `critical` bug causes program crashes and/or data loss and a `major` bug causes major loss of function, these numbers are high, especially since all of them took more than one year to be fixed.

Severity Realization Period (SRP)

Table 3.4 represents the number of `critical` and `major` long lived bugs in each system, the number of bugs whose severity level was changed, and the maximum number of times the severity level changed for a bug. Results show that for Java projects the severity level of 32%-43% of such bugs was corrected later. For C projects, the change in severity level is only from 8% to 27%. This indicates that the bug reporters could understand the actual severity level of more than 50% of the bugs for Java projects and 70%-90% of the bugs for C projects at the time of bug posting. Therefore, it is evident that developers took more than one year to fix a large number of bugs even after they realized that the bugs are very

Table 3.5: Time Needed for Understanding Bug Severity

System	Pre-SRP (Days)			Post-SRP (Days)		
	Avg.	Med.	Max.	Avg.	Med.	Max
JDT	374	163	1890	338	320	1712
CDT	80	7	590	760	700	1442
PDE	348	388	1274	300	34	1201
Platform	351	164	2208	498	410	2730
Linux	150	50	1228	644	572	1204
WineHQ	227	55	1756	649	560	2199
GDB	1090	756	2375	377	230	1394

important.

Now we analyze the bugs whose severity has been corrected later. Table 3.5 presents the average, median, and maximum Pre-SRP and Post-SRP (defined in Section 3.2.2) values. Our results show that it took almost a year on average to realize the correct severity level of the bug in three of the four Java projects. The only exception is CDT, where the average $Pre\text{-}SRP$ was 80 days. The maximum $Pre\text{-}SRP$ of each system shows that for some bug it took several years to realize the severity. On the other hand, for these bugs it took another year on average to be fixed. For CDT, which was the best in terms of average $Pre\text{-}SRP$, $Post\text{-}SRP$ was more than two years. From the maximum $Post\text{-}SRP$, we see that some bugs took even three to eight years to be fixed after developers realized the actual severity level. Therefore, our results indicate that for most long lived bugs in Java projects, $Post\text{-}SRP$ was high regardless of their $Pre\text{-}SRP$.

We see a varying Pre-SRP in C projects. For Linux and WineHQ, severity levels were fixed within six months and a year respectively. For GDB, it took more than three years to understand the actual severity level. However, it should be noted that the severity level of most of the important bugs in C project was known at the time bug posting. However, the Post-SRP was more than a year regardless of whether the actual severity level was understood in advance or later. From the maximum Post-SRP, we see that some `major` or higher severe bugs were fixed after six years.

Table 3.6: Duplicate Bugs

System	# Bugs	# Duplicate Bugs	# Duplicate Bugs (NOD)						Max NOD
			1	2	3	4	5	>5	
JDT	913	210	101	42	27	10	13	17	26
CDT	641	52	36	8	4	3	0	1	6
PDE	272	50	32	8	2	0	2	6	15
Platform	2,358	495	271	102	51	23	15	33	20
Linux	953	63	46	11	2	2	0	2	10
WineHQ	5,301	603	386	91	41	22	21	42	46
GDB	1,083	102	87	13	1	0	1	0	5

Duplicate Bugs

Table 3.6 presents an overview of duplicated bugs of long lived bugs. Results show that for 9% to 23% of long lived bugs in Java projects, users/developers submitted multiple bug reports. For C projects, the proportion of duplicate bugs varied between 6% and 11%. From the maximum number of duplicate bugs, we see that some bugs have more than 20 duplicated bug reports in Java projects. In WineHQ, there is a bug (id #6971), for which 46 duplicate bug reports were submitted. The middle columns present more fine grained results of duplicated bugs.

More than 90% of long lived bugs affect users' normal working experiences and thus are important to fix. However, it took a long time to fix these bugs even after realizing their severity. Moreover, there are multiple bug reports for these long lived bugs, which indicate the users' demand for fixing them.

3.3.3 RQ3: Where was most of the time spent in the bug fixing process?

Motivation. A bug fixing process can be divided into three main phases in terms of activity: i) assignment phase ii) fixing phase, and iii) verification phase. In this research question, we analyze the time taken by team leads/developers in each phase. The answer to this question is important to identify the time consuming phases so that developers as well as researchers can work on improving the processes involving that phase.

Table 3.7: Bug Assignment Time Vs. Bug Fixing Time

System	Assign. Period (AP)			Fixing Period (FP)		
	Avg.	Med.	Max.	Avg.	Med.	Max
JDT	463	374	2745	407	376	2854
CDT	603	552	2035	330	97	1815
PDE	484	482	2728	437	393	1622
Platform	459	373	3326	407	409	2854
Linux	347	275	1617	413	363	2472
WineHQ	489	327	3144	606	478	2563
GDB	915	605	4732	269	78	3224

Methodology. We extracted the resolution field information from the bug tracking systems to find when a bug report was assigned to someone, when that person fixed it and when the fix was finally verified. There are many bugs that are fixed multiple times. For those cases, we consider the last successful fix as the main fix. Then we compute the time period spent in each phase in terms of number of days as described in Section 3.2.2. Our initial hypothesis was that *perhaps it took a long time to assign long lived bugs to the appropriate developers. But once the bugs are assigned, it should not take too long to fix them.*

Results. Table 3.7 presents the average, median, and maximum time of both assignment period (AP) and fixing period (FP) in terms of days for all long lived bugs. Our results show that it took more than 1.5 years on average to assign the bugs to the appropriate developers in Java projects. The median AP also shows that the data is fairly normally distributed. The maximum AP shows that it can take more than six years to assign sum bugs to the correct developers. From Table 3.8, we can also observe that most of the long lived bugs in Java projects are reassigned at least once. The proportion of bugs reassigned ranged from 64% to 88%. More than 10% of the long lived bugs were reassigned 5 times or more.

For the C projects, the information regarding the first bug assignment was not present for all bugs. We found the bug assignment time only for those bugs that were

Table 3.8: Reassignments of Long-lived Bugs

System	Reassigned	Proportion	Max. Reassn.
JDT	771	84.45%	14
CDT	478	74.57%	8
PDE	174	63.97%	10
Platform	2066	87.61%	12
Linux	437	45.86%	8
WineHQ	399	6.33%	5
GDB	367	33.89%	6

reassigned (439, 401, and 366 bugs in Linux, WineHQ, and GDB respectively). Based on the results from those bugs, we see that the average bug assignment time in C projects varies from one year to three years.

Guo et al. [44] have conducted a study to investigate the reasons for bug reassignment. They observed that reassignments are not always harmful. In fact many reassignments happened to find the appropriate developers. However, they also observed that the required time for bug fixes increased with the increase of number of reassignments. Therefore, they concluded that excessive reassignments are harmful. They delineated five reasons for bug reassignments: finding the root cause, determining ownership, poor bug report quality, hard to determine proper fix, and workload balancing. Therefore, taking the aforementioned findings into account, our results indicate that the assignment of these long lived bugs was complex and time consuming, supporting our initial hypothesis.

However, unlike our expectations, the average FP for all systems was quite high: around a year. By seeing the median FP for CDT, we understand the data is skewed. But for the other three subjects, it is not the case. Also the maximum FP shows that, like the bug assignment, it took more than five years for some bugs to be fixed after they were assigned to the right developers.

On the other hand, for the verification period, we found that most of the bugs were never verified, at least according to the Bugzilla data. However, if they do get verified, the verification time is pretty small: less than a month for most of the subject systems.

Bug assignment and bug fixing are still time intensive processes, despite the availability of automatic bug assignment tools that could have been used.

3.3.4 RQ:4 What are common reasons for long lived bugs?

Motivation. To improve the bug fixing process, first we need to understand the underlying reasons for delays. Delineating the common reasons of long lived bugs will help researchers deal with the problem more systematically.

Methodology. To answer this question, we first manually analyzed all the `critical` and `major` bug reports from JDT. We have intentionally chosen the highly severe bugs, since they should be taken seriously by the developers and thus, we will be able to identify the actual reasons for the delay. We also analyzed 50 recent (`critical` or `major`) long lived bugs from PDE and Platform. Since JDT and CDT are from similar domains, we did not take any bugs from CDT. Finally, we manually analyze 20 bugs of the Linux kernel (10 oldest + 10 recent long lived bugs with `high` severity) to check if we find any new category. In this way, we identified a set of 125 (= 55+25+25+20) bug reports for manual analysis.

In order to identify the underlying reasons, first, we read the bug summary and description to understand the nature of bugs. Second, we carefully analyze developers' comments to understand the reasons for any delays since developers often discuss different problems associated with a given bug through comments. For most of the cases, the actual reasons were easily identifiable.

To categorize the reasons for delays, we followed an open-ended taxonomy. We incrementally analyzed all the bug reports. For any given bug report, first we identified the high level reason and checked if the reason already fits into any of the existing categories. Otherwise, we create a new category. We have quoted several key comment(s) for most of the categories to better understand the tagging procedure. In the few cases where the reasons were ambiguous, we relied on contextual information.

Results. The following summarizes a taxonomy of common reasons for long lived

bugs that we found in the subject systems.

1) Hard to understand: Understanding/locating buggy statements/files in a software project is hard. Sometimes, identifying even the buggy component can be hard. For example, there is a bug (#128563) in JDT, where developers had hard time in understanding if it is a VM or JDT bug. The following comments explain the situation:

"I found something quite interesting. If you move the classes from the two output folders into the same directory and you run from there, it works fine. We generate exactly the same bytecodes in both cases. The VM should behave the same. Might be a VM bug."

After two years, another developer commented–*"I believe this is our bug, we should not reference a non accessible type in our bytecode. The fact it works at times feel like unspecified behavior from the VM."*

2) Uncertain how to fix: Sometimes developers may know how to solve a bug, but need to wait for making the solution consistent/robust with other parts of the software. The following comments in bug # 3849 represent such a scenario:

"*I would like to defer this until we know how we will implement the new Code Manipulation Infrastructure. This is only possible if we get a better undo story. Currently we can only push undo commands on the refactorings undo stack if a file is save. Otherwise the next save would flush the current undo stack which would remove the undo object for extract method.*"

3) Hard to fix: This kind of bug is hard to fix. There were lots of group discussions for a long time regarding different alternative solutions and finally the group agreed on some specific solution.

4) Risky to fix: Sometimes, bugs are caught just before the release. Then if developers think that it would be risky to change the relevant code, they generally defer it for the next release even if the bug is important. Then it takes a long time to fix the bug. The following developers' comments on bug #80,000 in JDT represents such a scenario:

"Will investigate during RC2 whether there's a low risk fix for this."

Table 3.9: Reasons of Sampled Long Lived Bugs

Reason	# Bugs	Bug IDs
1)	13	113870 (PDE), 128563 (JDT), 241241 (Platform), 245008 (Platform), 247766 (PDE), 268833 (Platform), 278598 (PDE), 3022 (Linux), 5534 (Linux), 5637 (Linux), 9905 (Linux), 13484 (Linux), 38442 (Linux)
2)	3	3849 (JDT), 36204 (JDT), 133072 (PDE)
3)	16	3849 (JDT), 24951 (PDE), 36204 (JDT), 38746 (JDT), 40243 (JDT), 46407 (JDT), 67425 (JDT), 82850 (JDT), 99137 (JDT), 233643 (PDE), 233773 (Platform), 266651 (JDT), 273450 (Platform), 295200 (JDT), 38442 (Linux), 44161 (Linux),
4)	2	80000 (JDT), 102780 (JDT)
5)	6	1766 (JDT), 33035 (JDT), 36204 (JDT), 136135 (PDE), 3410 (Linux), 46171 (Linux),
6)	14	36204 (JDT), 46216 (JDT), 50735 (JDT), 109636 (JDT), 117698 (JDT), 156168 (JDT), 175226 (JDT), 224880 (Platform), 243894 (Platform), 257202 (Platform), 266651 (JDT), 267649 (Platform), 273450 (Platform), 278598 (PDE)
7)	11	1766 (JDT), 39222 (JDT), 54831 (JDT), 82850 (JDT), 83473 (JDT), 195183 (JDT), 262032 (Platform), 294650 (Platform), 298795 (Platform), 2979 (Linux), 31602 (Linux),
8)	7	3920 (JDT), 19251 (PDE), 46216 (JDT), 67425 (JDT), 224880 (Platform), 235572 (Platform), 277638 (Platform)
9)	12	6437 (JDT), 19248 (PDE), 28637 (JDT), 44035 (JDT), 61744 (PDE), 132333 (PDE), 158589 (PDE), 271373 (Platform), 14563 (Linux), 43981 (Linux), 45031 (Linux), 46161 (Linux)
10)	2	34033 (PDE), 130874 (JDT)
11)	8	12955 (JDT), 16686 (JDT), 20919 (PDE), 24951 (PDE), 29799 (PDE), 34399 (PDE), 231936 (PDE), 290324 (PDE)
12)	34	21100 (PDE), 26556 (JDT), 38288 (PDE), 39803 (JDT), 51862 (PDE), 89347 (JDT), 95288 (JDT), 97541 (JDT), 111419 (JDT), 128303 (PDE), 129689 (PDE), 149316 (JDT), 154823 (JDT), 175133 (JDT), 181954 (JDT), 209537 (JDT), 226595 (Platform), 234623 (Platform), 235554 (Platform), 236104 (Platform), 237025 (PDE), 238943 (JDT), 258952 (Platform), 262032 (Platform), 267173 (Platform), 275910 (Platform), 277638 (Platform), 279781 (Platform), 285101 (Platform), 5637 (Linux), 11509 (Linux), 38312 (Linux), 41682 (Linux), 43153 (Linux)

Two weeks later, the same developer commented: *"Sorry, too risky to touch at this point."*

5) Incomplete fix: This is considered as one of the common problems for taking a long time to fix bugs. Developers often miss corner cases while bug fixing and need to re-fix again until the problem is fully solved. Here is a developer's comment regarding a bug fix for #38746 in JDT.

"The fix for this problem is not sufficiently robust. Please see Bug 75454 for more information as to how things can go wrong. Not only does the situation described there happen once, but it happens 60 times on start-up (10 minutes of start-up time)."

There are also lots of other reasons for incomplete bug fixes. For a comprehensive set of reasons for incomplete bug fixes, please refer to [106].

6) Importance was not realized until duplicate bugs were reported: We found many bugs where there were some activities around the bug for some time, which we observed by reading developers' comments. After that there was no activity for a long time. Then somebody pointed out some duplicate bugs and everybody started talking again; the bug was fixed quickly. The following comment on bug # 16114 in PDE represents such an example:

"*this one is experienced by several users (see the duplicates for more info). Looks like something causes certain fragment files on the disk to be in use and when we try to delete the project (even with 'force' option), we fail. This leave us with a partially deleted project that causes more trouble after that."*

7) Reproducibility: There are some bugs that take a long time to reproduce, but once the bug is reproduced, it is fixed quickly. For example, it took 1 year and 4 months to reproduce the bug #268833 in Platform but took only one day to fix. This problem often happens from low quality bug reports, execution difference due to platforms, and so on.

There are some interesting bugs where users know how to reproduce the bug but it happens for some special cases and thus needs some time to reproduce. For this kind of

bug, if a user submits the bug without concrete data, it takes a long time to reproduce the required data that developers need to analyze the bug. Therefore, the bug fix gets delayed even though the responsible developer is ready to fix it. For example, to debug an "out of memory" problem in JDT (# 54831), developers needed a heap dump, which was not submitted when the bug was posted. When the assigned developer asked for it, the bug reporter (who is actually another developer of JDT) was busy with his own work and could not submit the heap dump on time. As a result, it took a long time to fix the bug.

8) Schedule issue: Sometimes developers also feel that a bug is important to fix. However, they have more important bugs at hand that should be fixed earlier. Therefore, although the other bugs are important, they are generally deferred. For example, there is a `blocker` bug (#10800) in JDT that prevented users from putting a space in VM arguments. `Blocker` bugs are considered as the most severe bugs. However, such a severe bug was deferred due to scheduling issues. Certainly, other developers were not very happy about that. The following developers' comments illustrate the scenario more clearly.

"Can more explanation be given as to why this issue has been marked as LATER? Does this mean it will not be fixed any time soon? If so, I find it very unfortunate as this is a very serious bug and requires nasty work arounds. If not, then my apologies..."

In reply, the responsible team leader said: *"In this case, 'LATER' means probably not for the final 2.0 release (tentatively scheduled for sometime in May). Quite simply, this problem was not deemed as critical as a lot of other problems that need to be solved for 2.0. The debug committers have A LOT to do before 2.0. But the beauty of an open source project is that if someone feels strongly about a particular feature or bug, they can make a contribution. If you would like to contribute a fix, I would be happy to review it."*

In the end, it took more than two years to fix the bug.

9) Not aware of fix or reopened due to misunderstanding: We are not completely certain if these bugs are really long lived. In this category, some bugs perhaps were fixed earlier but developers have not changed the status in Bugzilla. Therefore, the reporters or

Table 3.10: Analysis of Bug Fixes

System	# Bugs	#Number of Files (NOF)						Med	Max
		1	2	3	4	5	>5	NOF	NOF
JDT	223	80	41	37	10	15	40	2	47
CDT	185	50	32	19	18	8	58	3	237
PDE	105	34	13	13	12	4	29	3	211
Platform	740	349	114	72	47	36	122	2	91
Total	1253	513	200	141	87	63	249	-	-
(%)	-	40.94	15.96	11.25	6.94	5.03	19.87	-	-
Linux	171	117	28	12	4	3	7	1	19
WineHQ	609	231	203	70	37	29	39	2	51
GDB	33	0	10	2	1	0	20	7	222
Total	813	348	241	84	42	32	66	-	-
(%)	-	42.8	29.64	10.33	5.17	3.94	8.12	-	-

other developers were not aware of the fix. Later some other developers just closed the bugs mentioning that probably the bugs have been already fixed. Another case is that sometimes reporters misunderstood something and reopened a given bug again. But then some other developer clarified the mistakes the reporter was making and finally again marked it as `FIXED` and `RESOLVED`.

10) Infrequent use case: This kind of bug is important to fix considering their destructive ability. However, they are not very frequent use cases. Therefore, developers just defer it for next milestone. For example, due to the bug # 130874 in JDT, a user can lose his/her Java code template references. However, the developer deferred it by making following comment.

"We should definitely fix this during 3.5. Too late for 3.4 and really not a very common case."

11) Others: There are also other reasons for delay in bug fixing such as expert developers are on vacation, dependency on other bugs to be fixed, and various document fixing.

12) As-usual delay: We have not found any specific reasons for these bugs by

analyzing developers' comments and thus we considered them as as-usual delay. If there are some specific reasons (mentioned above) to make delay, it is highly likely that developers will discuss it like the other bugs. However, it is also possible that the fixes were deferred due to scheduling issues. The following comment for bug #149316 in JDT can be served as an example of as-usual delay.

"*Thanks for the good examples, sorry for the wait. fixed > 20080422.*"

One may think that since these bug-fixes were delayed without any specific reasons, they are probably not that important or relevant. However, it should be noted that for this investigation we analyzed only `critical` or `major` bugs. So they are already marked as important by the reporters or developers. Furthermore, since we ourselves are users of these software systems, we understand that many such bugs can be frustrating. The following represent two summaries of such bugs:

Bug # 26556 (JDT): "*PDE Junit does not read plugin info from the plugins directory.*"

Bug # 43153 (Linux): "*Random SATA drives on PMPs on sata_sil24 cards not being detected at boot since 3.2/3.4.*"

We encountered an interesting finding while analyzing the bug reports manually. We started our manual investigation with JDT and listed all the common reasons from there. We have not found any new common reason when analyzing the bug reports for PDE, Platform, or the Linux Kernel. Therefore, we believe that this is a comprehensive list of reasons for long lived bugs. It should be noted that these reasons are not mutually exclusive.

Reasons for long lived bugs are diverse. While problem complexity, problems in reproducing errors, and not understanding the importance of some of the bugs in advance are the common reasons, we observed there are many bug-fixes that were delayed without any specific reason.

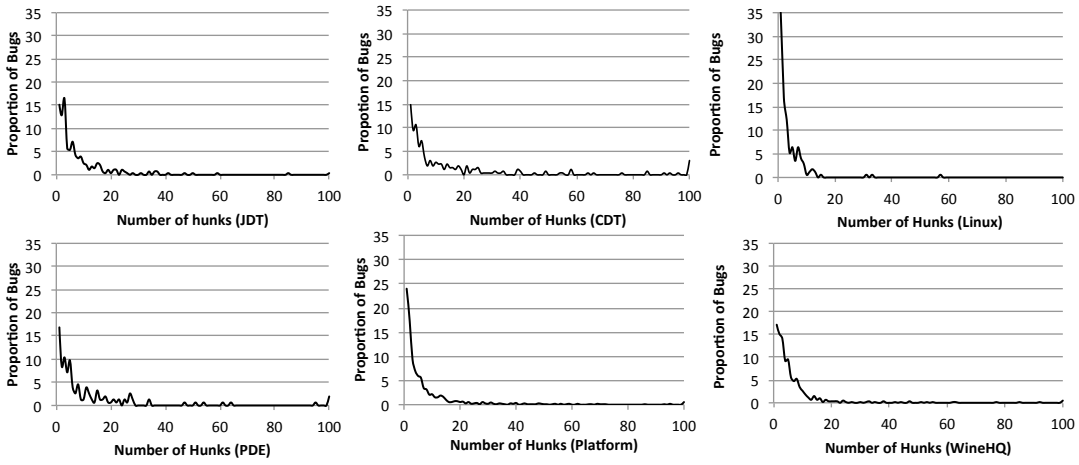

Figure 3.4: Number of Hunks Vs. Proportion of Bugs

3.3.5 RQ5: What is the nature of bug fixes?

Motivation. In this research question, we investigate the nature of bug fixes in terms of source code changes. The answer to this question will help us in better understanding the bug fixing process, estimating change efforts, and so on, which will be useful in exploring potential approaches for improving the overall bug fixing process.

Methodology. In order to identify the bug fixing changes in the source code for the long lived bugs, we followed the *methodology* described in Section 3.2.3. We were able to identify fixed files for 223, 185, 105, and 740 bugs in JDT, CDT, PDE, and Platform respectively, and 171, 609, and 33 bugs in Linux, WineHQ, and GDB respectively. Then, we compute the number of changed files, number of hunks, and code churn for each bug-fix, as described in Section 3.2.2. These metrics are often used to get a rough idea about change effort, although understanding the actual change effort is difficult and depends additionally on the implemented algorithm and code complexity itself. Analogous to previous study results [165], our initial hypothesis was that *the required source code changes to fix most of the long lived bugs would be large.*

Used Metrics. Number of Changed Files, Hunks, and Code Churn.

Changes at File Level

To understand the nature of bug fixes, we first analyze the source code changes in terms of number of changed files. As we stated in our initial hypothesis, we expected a large number of changed files for most long lived bugs. Surprisingly, from the results in Table 3.10, we see that for both Java and C projects, more than 40% of the fixes involved only one source code file. This proportion varied from 27% to 47% among the Java projects, whereas it varied from 38% to 68% in C projects (except GDB). For only 30% of long lived bugs, the required changes spanned over more than three files in Java projects, whereas it was only 17% for C projects. From our results it is also noticeable that the maximum number of changed files in each system is quite high. However, when we manually investigated such changes, we found they are often moving files from one directory to another, or adding a test suite to test a specific or multiple bugs.

Number of Hunks

Now we analyzed the changes in terms of hunk size to get more fine grained results. A large number of hunks indicates that developers needed to modify a lot of different places to fix the bugs. Figure 3.4 presents the number of bugs for each hunk size in Java projects. Our results show that 43% to 53% of bugs in Java projects were fixed by changing five hunks or fewer. In C projects, the proportion was even higher. 76% and 67% of bug fixes in the Linux Kernel and WineHQ respectively involved 5 hunks or fewer. More than 70% of the long lived bugs for JDT and Platform were fixed within 10 hunks, whereas the numbers are 16 and 17 for PDE and CDT respectively. The median number of hunks for all long lived bugs is only 5 or less for all projects except GDB. Considering that this is an overestimated measurement of source code changes since we have analyzed textual *diff* results (so changes in comments have been also counted), we believe the number of hunks is low. It should be noted that we presented all the bugs that involved more than 100 hunks in the graph at the end. Therefore, there is a spike at the end of some graphs.

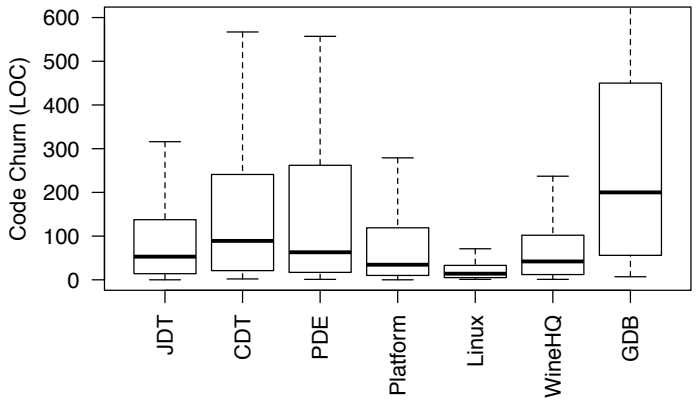

Figure 3.5: Code Churn of Long Lived Bugs

Code Churn

We investigate further low level changes (at line level). Figure 3.5 presents the distribution of code changes in terms of code churn (defined in Section 3.2.2). From the figure, we see that a considerable proportion of bugs required major changes. More specifically, for 23% of bugs, the value of code churn was more than 100. However, there is even a larger proportion of bugs that required changes of less that 20 lines. For example, for 8-14% of bug fixes in Java projects, the value of code churn was from 1 to 5, for 7-12% the value of code churn was from 6 to 10, and for 9-13% the value of code churn was from 11 to 20. For the Linux Kernel and WineHQ, the proportion was even higher. 40% of long lived bugs in the Linux Kernel and 20% of bugs in WineHQ required changes in only 10 or fewer lines of code. Recalling Section 3.2.2, it should be noted that the code churn value of a one line change is 2, whereas an addition or deletion of one line is 1. Therefore, a value of code churn value of 10 may be changes in only five lines. We understand that some smaller bug fixes can be complex. But at the same time, we also stress that many long lived bugs could be fixed quickly through careful prioritization.

It should be noted that the overall bug fixing changes in GDB seems to be larger than other projects. However, as Table 3.10 shows, we were able to identify the bug fixing changes for only 33 bugs in GDB. Therefore, it is very difficult to draw any conclusion from GDB.

A Qualitative Analysis

Now we present three bug fixes to discuss how a simple fix can take a long time to be fixed. To select these example-fixes we considered the following criteria: i) they are from different subject systems, ii) they are important, i.e., critical or major (or high for the linux kernel), and iii) their descriptions are concise enough to present in the dissertation.

Bug # 38260 (SWT): This is a bug in the SWT component of Eclipse Platform. The bug was first reported as having a `normal` severity level. However, within 15 days, it was reconsidered to be `critical`. The following is the bug description provided by the reporter:

"*When I use the CCombo with the dialog, the dropdown list shown in the back of the Dialog.*

I did as following.

1. Create one sample application with a Group.

2. The parent of group is one shell.

3. I create a dialog.

4. I put my application to this dialog. and change the Group's parent as the dialog's parent.

Now the above describped pbm occured."

From the bug comments, we found that the bug was reproduced within three days and the actual problem was identified within a month. However, it took more than ten months to fix the bug. Figure 3.6 presents the changed code for fixing this bug and it was only a one line change.

Bug # 195183 (JDT): This is a `major` bug in the Debug component in JDT. The bug summary and description (condensed) are as follows: *"JavaClassPath.performApply()*

```
--- a/bundles/org.eclipse.swt/Eclipse SWT Custom Widgets/
                  common/org/eclipse/swt/custom/CCombo.java
+++ b/bundles/org.eclipse.swt/Eclipse SWT Custom Widgets/
                  common/org/eclipse/swt/custom/CCombo.java
@@ -76,7 +76,7 @@ public CCombo (Composite parent, int style) {
          if ((style & SWT.READ_ONLY) != 0) textStyle |= SWT.READ_ONLY;
          if ((style & SWT.FLAT) != 0) textStyle |= SWT.FLAT;
          text = new Text (this, textStyle);
-         popup = new Shell (getShell (), SWT.NO_TRIM);
+         popup = new Shell (getDisplay(), SWT.NO_TRIM | SWT.ON_TOP);
          int listStyle = SWT.SINGLE | SWT.V_SCROLL;
          if ((style & SWT.FLAT) != 0) listStyle |= SWT.FLAT;
          if ((style & SWT.RIGHT_TO_LEFT) != 0) listStyle |= SWT.RIGHT_TO_LEFT;
```

Figure 3.6: Bug Fixing Changes for # 38260 in SWT Component of Platform

uses original instead of working copy causes NPE"

"Steps To Reproduce: I use JavaClassPath in my custom launch config and has some code like:" .....[some code snippet] *"and as soon as performApply() is called isDefaultClasspath() fails since it is passed in a null as a launchconfig even though I passed in a newly created one. This worked fine in eclipse 3.2 and it seem the culprit is that wc.getOriginal() is used instead of just wc. Resulting in a NPE."*

From the bug description, we can see that the bug report was very specific. The reporter clearly pointed out that there are some problems with the $getOriginal()$ API. Interestingly, from the bug-fix (Figure 3.7), we found that only one line was changed and the change was the removal of the $getOriginal()$ API. But by that time, more than two years had passed.

Bug # 3410 (Linux Kernel): This is a bug with `high` severity in the Linux Kernel. The bug summary and description provided by the reporter are as follows:

Summary: "passive mode is not left, once entered"

Description: "Steps to reproduce:

echo "xx:xx:low_value:xx:xx:" >/proc/acpi/

thermal_zone/∗/trip_points echo "xx:xx:high_value:xx:xx:" >/proc/acpi/

thermal_zone/∗/trip_points

passive mode still active, even the temperature is far below the trip point

```
--- a/org.eclipse.jdt.debug.ui/ui/org/eclipse/jdt/debug/ui/launchConfigurations/JavaClasspathTab.java
+++ b/org.eclipse.jdt.debug.ui/ui/org/eclipse/jdt/debug/ui/launchConfigurations/JavaClasspathTab.java
@@ -272,7 +272,7 @@ public class JavaClasspathTab extends AbstractJavaClasspathTab {
  public void performApply(ILaunchConfigurationWorkingCopy configuration) {
          if (isDirty()) {
                  IRuntimeClasspathEntry[] classpath = getCurrentClasspath();
-                 boolean def = isDefaultClasspath(classpath, configuration.getOriginal());
+                 boolean def = isDefaultClasspath(classpath, configuration);
                  if (def) {
                         configuration.setAttribute(IJavaLaunchConfigurationConstants.ATTR_DEFAULT_CLASSPATH,
                                                    (String)null);
                         configuration.setAttribute(IJavaLaunchConfigurationConstants.ATTR_CLASSPATH,
                                                    (String)null);
```

Figure 3.7: Bug Fixing Changes for # 195183 in Debug Component of JDT

```
--- a/drivers/acpi/processor_thermal.c
+++ b/drivers/acpi/processor_thermal.c
@@ -102,8 +102,8 @@ static int cpu_has_cpufreq(unsigned int cpu)

        struct cpufreq_policy policy;
        if (!acpi_thermal_cpufreq_is_init || cpufreq_get_policy(&policy, cpu))
-                return -ENODEV;
-        return 0;
+                return 0;
+        return 1;
```

Figure 3.8: Bug Fixing Changes for # 3410 in Linux Kernel

With this bug, there was an attempt to fix it on the same day the bug was reported. However, the fix was incomplete, which was identified on the next day. Although there was some discussion regarding the bug around that time, it remained unfixed for almost one and half years. Figure 3.8 shows the second fix of the bug, which made the first fix complete. The patch involved changes in only two lines of code.

We believe that one year or more is too long time to fix the bugs like these examples, especially considering that they were considered as very important.

Unlike previous studies, we found that a bug surviving for a year or more does not necessarily mean that it requires a large fix. We found that 40% of long-lived bug fixes involved only a few changes in only one file.

3.3.6 Impact of Different Definitions for Long Lived Bugs

As we noted in RQ1, the definition of long lived bugs is subjective. Therefore, it is subject to criticism for any specific definition. In this study, we emphasized the length of release cycle to define long lived bugs. Table 3.11 presents a detailed overview regarding release cycle length of the subject systems.[20,21,22,23] Our rationale was that if a valid bug exists in two or more *major releases*, it would be unsatisfactory for users. However, setting a time

[20] http://wiki.eclipse.org/Simultaneous_Release

[21] http://en.wikipedia.org/wiki/Linux_kernel

[22] http://wiki.winehq.org/TimeBasedReleases

[23] http://www.gnu.org/software/gdb/schedule

Table 3.11: Time Difference between Two Major Releases (Months)

System	Min	Max	Mean	Median
JDT/CDT/ PDE/Platform	12	12	12	12
Linux	2	32	9	5
WineHQ	2	31	12	8
GDB	3	18	9	8

threshold for long lived bugs is difficult, since from the results, we observe that different projects follow different release schedules and the length of release cycles varies within and across projects. In this study, we focused on the average release cycle length in the considered subject systems, which varied from nine months to one year. Since we wanted to take a conservative definition and make the results comparable, we considered those bugs as long lived that existed in the systems for more than one year (which is the average release cycle length of five out of seven subject systems). Certainly, one could use a different or more sophisticated definition of long lived bugs such as a variable threshold for different projects (min, max, or median), even a variable threshold within a project. For instance, one can set a time threshold for each release to consider only those bugs that actually existed in two or more major releases.

The threshold for defining long lived bugs can have a significant impact on the results. Certainly, for a different threshold, we would get a different set of long lived bugs in each system, for which we would get different statistics for all research questions. However, since we have set the maximum average release cycle length, we do not overestimate long lived bugs. Furthermore, if one wants to set a variable threshold—e.g., average or median release cycle length for each project (which is less than one year), one can estimate the number of such bugs easily from Table 3.2. For a threshold more than one year, Figure 3.3 can be used.

3.4 Developers' Survey

Since we have not found any specific reasons for a significant proportion of `critical` or `major` long lived bugs (classified as "as-usual delay"), and many long lived bugs involved small fixes, we believe that many such delays could have been avoided if developers could predict the severity and change effort in advance. To investigate what developers think about our assumptions, we conducted a survey. Since we were looking for developers' opinion on our findings, we preferred open questions rather than multiple-choice questions. In the survey, we briefly explained our results and possible actions (e..g. predicting severity, change effort, prioritization, etc.), and asked developers if they agree with us or not. We also asked what actions could be taken to minimize the number of long lived bugs, if they think otherwise. The following is the main excerpt from our email:

"Currently we are conducting an empirical study on the long-lived bugs (that took more than one year to get fixed) in four Eclipse projects (JDT, CDT, PDE, Platform).

We found although developers are very active in removing bugs (around 50% of bugs fixed within 1 week), 5-9% bugs took get fixed more than 1 year. 90% of these long-lived bugs are normal, critical, or major, NOT minor or trivial.

We manually inspected 100 bugs (reading bug summary, description, and developers comments) and found that there are diverse reasons for delays including hard to understand, hard to locate, hard to fix, risky to fix, reproducibility, infrequent use-case, schedule issue, and so on. However, we found a significant amount of bugs as well where we did not find any specific reasons.

We also found that many long lived bugs involved small fixes–a couple of lines changes in one file. So we assume that many such a long delay could be avoided if developers had proper tool support for task prioritization, predicting change effort etc.

Our question is:

Do you think many long lived bugs can be fixed faster through careful task prioritization (by knowing who should fix that bug, predicting the severity and change effort in

advance)?

If not, what solutions you think that researchers can work on to assist developers to minimize long lived bugs?"

We sent the survey to all the 104 developers who contributed to JDT, CDT, PDE, and Platform from year 2010. In total, we sent the survey to 104 developers. Among them, 38 developers do not use their email addresses anymore. We got responses from five developers. Among them, four developers said that tool support may be helpful to avoid long lived bugs but they focused on different functionalities. Here are the developers' comments:

Developer 1: "I think that if we could predict the change effort that would help a lot in getting long-lived bugs resolved. Often a bug will be created at a time when committers are busy, and that bug then falls through the cracks. If committers could know the bug was easy to fix, Im sure they would look at it again."

Developer 2: "I think task prioritization helps, especially when the existing backlog is large. A big problem going over the old backlog when the backlog is huge... prioritization of issues that come in are usually handled well."

Developer 3: "I personally would be more interested in tools that would help us getting less duplicate bug reports and bug reports of better quality (steps to reproduce, readable English, ...). "

Developer 4: "What helps is figuring out duplicates. This is done to some extent already, but could be largely improved, e.g. by also scanning attachments. The more duplicates a bug gets, the higher could be its severity."

Furthermore, some of them think that understanding the impact of change is even more critical and often a major reason for delay in making bug fixing commits.

Developer 5: " imagine a situation which was a major holder for me in many cases: I knew the code and knew the fix, but was aware that there are thousands of plugins that will use it, and somebody may be unhappy with the change. It took ages for a commiter to find

out what is the true effect of the patch even despite it was simple (you know, the butterfly effect).

I think that long lived bugs could be avoided mainly by using architectures that do not limit further choices (like microservices, but maybe there is more)."

Besides these positive comments, we have also got some arguments. For example, one developer asked, "When a bug is not touched in a long time (no pings, no duplicates), isn't that a sign that the bug is really not that important? As you know, also a 2-line fix needs a lot of code reading to ensure it doesn't break other parts of the project. Our resources are very limited, so we try to focus on bugs that are real blockers or where we think a lot of users are affected ". However, he emphasized that duplicate bugs are an indicator of importance level. In response, when we said "Around 20%-25% of long lived bugs (in JDT, PDE, Platform) have duplicated bug reports", we have not received any further response.

The most important observation in our survey is that the developers did not say that long lived bugs are irrelevant. Rather they talked about various kinds of tool support such as predicting change-effort and change impact, detecting duplicates, etc., which could play an important role in minimizing long lived bugs.

3.5 Threats to Validity

This section discusses the validity and generalizability of our findings. In particular, we discuss Construct Validity, Internal Validity, and External Validity.

Construct Validity: We used two artifacts: bug reports from the bug tracking system and source code changes from the version history, which are generally well understood. We have used also well known metrics in our data analysis such as various time periods, the number of changed files, the number of hunks, code churns, which are straightforward to compute. Both the used dataset and version histories are also publicly available, which enable the replication of this study. Therefore, we argue for a strong construct validity.

Internal Validity: In our study, we relied on the information from the bug track-

ing system and version histories. However, the information in these systems may not be completely accurate. For example, a change request can be actually an enhancement but it could be misclassified as a bug [46]; the severity level associated with some bugs may not reflect the actual severity levels. Furthermore, a developer may commit a bug fixing change a long time after she actually fixed the bug. Similarly, a tester may change the bug status from `FIXED` to `VERIFIED` a long time after she actually verified the bug. Although it is very difficult to completely eliminate these threats, we performed extensive manual investigations and qualitative analyses, and provided many concrete examples to minimize these threats.

To delineate the common reasons of long lived bugs, we manually analyzed bug reports. There might have been some unintentional misinterpretations during the manual verification due to the lack of domain knowledge or the lack of useful contextual knowledge. However, we held extensive discussions to minimize this threat.

We used traditional heuristics to find mappings between bug fixing changes and associated bug reports. Although, in this way, we missed many bug fixing changes, the precision of our result is very high, which is important for our study. ReLink [160] is a more advanced algorithm that improves the recall quite a bit for finding the mappings. However, it also sacrifices some precision. In future, we would like to use ReLink to see how it affects our results.

The phenomena studied had major releases. Systems with more frequent release cycles may well exhibit different phenomena, although there will still be long lived bugs. The number of release cycles and the lapse times for long lived bugs in this context are likely to be different.

External Validity: We have used seven subject systems in our experiment and all of them are open source projects. Although, they are very popular projects, our findings may not be generalizable to other open source projects or industrial projects. However, the Java dataset that we used has more than 165,000 bug reports, and the C dataset we created

contains more than 77,000 bug reports, which is large. Furthermore, the consistent findings from both Java and C projects make our results more generalizable for large open source projects. However, additional confidence could be achieved by adding more subject systems (both open source and industrial).

3.6 Related Work

The study of software bugs/faults has been an active research area for nearly two decades. Perry and Stieg [107] were among the first to analyze software faults in a large evolving software system. Since then, researchers analyzed various software artifacts relevant to bugs (e.g. bug report, bug fixing changes) to understand and to improve different steps (e.g. bug reporting, triaging, localizing, fixing) of the bug fixing processes.

Thung et al. [143] investigated when a bug should be reported. Bettenburg et al. [11] studied the qualities of a good bug report. In another study, Bettenburg et al. [12] investigated the extents and reasons of duplicated bug reports. They presented empirical evidence that duplicate bug reports are not necessarily bad. They often provide additional information, which is important for automatic bug triaging, bug assignment, and localization. Guo et al. [43] characterized and predicted which bugs get fixed. They noted that in addition to the importance of bugs, there are several other factors that affect whether a bug would get fixed—e.g., the reputation of bug reporters, influence of seniority, personal relations and trust, etc. In another study [44], the same authors investigated the reasons for bug reassignment (described in more detail in Section 3.3.3). Lamkanfi et al. [69] and Tian et al. [146] predicted the severity and priority of bugs respectively. Anvik et al. [4] and Shokripour et al. [135] proposed approaches for automatic bug assignment. Saha et al. [127] and Zhou et al. [173] proposed different approaches for automatic bug localization. To complement these studies, in this dissertation, we focused on long lived bugs to understand their characteristics and reasons.

The work closest to ours is the study of bug-fix time prediction, since these studies

also identify the factors that are correlated to bug fixing time. Weiss et al. [156] considered the text (summary and description) in the bug report as the prime factor and used that to predict bug fix time. Panjer [105] observed that commenting activity, bug severity, product, component, and version are the most influential factors in predicting bug fix time. Giger et al. [41] found that the assigned developer, the bug reporter, and the month when the bug was reported have the strongest influence on the bug fixing time. Zhang et al. [166] also found the same results for commercial projects. Anbalagan et al. [2] found a strong relationship between bug fixing time and the number of people participating in the bug report. Marks et al. [94] observed different results for different projects. They found that bug fixing time is important for the Mozilla project, whereas, bug severity is the key for Eclipse.

While the aforementioned studies vary in terms of the analysis and techniques used, some of the common approaches used in these studies are that researchers used various machine learning or data mining techniques to analyze the whole bug dataset in identifying the overall factors affecting bug fixing time. Bhattacharya and Neamtiu [14] pointed out that most attributes used by prior work do not correlate with bug-fix time when analyzed in isolation, and thus they emphasized on finding new attributes that correlate with bug-fix time in isolation. We stress that it is also important to analyze various kinds of bug-fixes in isolation to gain better insight about specific group of bugs. For example, Shihab et al. [134] studied and predicted reopened bugs, Park et al. [106] investigated supplementary bug-fixes, and Ngyuen et al. [102] analyzed recurring bug-fixes. In this study, we analyzed long lived bugs to advance empirical knowledge further regarding long-term delays in the bug fixing process.

There is another group of studies that investigated the actual source code changes for bug fixes to study bug fixing time. Canfora et al. [24] found relationships between different program constructs and bug survival time. For example, exception handling leads to low bug survival time. Zhang et al. [165] found that bug fixing time increases with the increase of code churns. In our study, we have also analyzed the source code changes for

long lived bug-fix and showed that many long lived bugs involved only a few changes in only one file.

3.7 Summary

Bug fixing is a fundamental and critical activity in the software development and maintenance phases since buggy behavior may cause not only costly failures but also can affect the user's overall experience with the software product. In this work, we showed that although the software development and maintenance processes have advanced a lot, there are still a significant number of bugs in each project that survive for more than a year. More than 90% of these long lived bugs may have affected users' normal working experience. The average bug assignment time was more than one year and the bug fix time after the assignment was another year on average. When we analyzed the bug descriptions and the developers' comments around these bugs, we found that the reasons for long lived bugs are diverse. While problem complexity, problems in reproducing, and not understanding the importance of some of the long lived bugs in advance are the common reasons, we observed there are many bugs that were delayed without any specific reasons. Finally, by investigating the actual source code changes for these long lived bugs, we noted that a bug surviving for a year or more does not necessarily mean that it requires a large fix. In fact, we found 40% of long-lived bug fixes that involved only a few changes in only one file. Most importantly, all of the findings are consistent across the projects that we considered regardless of their domains or programming languages.

In summary, our results indicate that the overall bug fixing time of many, if not all, long lived bugs can be reduced through careful prioritization, and by predicting their severity, change effort, and change impact. The responses from the developer survey also support our observation. Our findings also indicate that although there are a number of tools for supporting bug triaging and fixing (e.g. automatic bug assignment, bug fix time prediction), we appear to realize very few benefits from them. There may be two possible

reasons: i) developers are not aware that these tools exist, or ii) the tools do not meet developers needs or expectations. In the future, we plan to conduct a separate study to understand the reasons for this phenomenon. We believe all of these findings together will play an important role in developing new and more effective approaches for bug triaging as well as improving the overall bug fixing process.

Chapter 4

Are These Bugs Really "Normal"?

This chapter is based on our paper, "Are These Bugs Really "Normal"", published in the Proceedings of the 12th Working Conference on Mining Software Repositories [126].[24]

4.1 Context and Problem Statement

Bug tracking systems are among the most frequently used resources for research in mining software repositories [11, 12, 24, 27, 44, 46, 134, 165]. They are also often used in developing new techniques for automated software engineering such as automatic bug triaging [21], bug assignment [4], bug-fix time prediction [41, 166], severity prediction [69], bug prioritization [146], and bug localization [127, 173]. A critical element of much of this work is to understand the importance of the bug reports found in these bug tracking systems. As this information is difficult to accurately infer, and may depend on the priorities and point of view of the bug reporter, studies typically rely on the "severity" label provided in the bug report [123]. While the labels vary by project, they typically amount to some variant of *Severe*, *Normal*, and *Minor*.

[24] Please note that Dr. Julia Lawall, Dr. Sarfraz Khurshid and Dr. Dewayne Perry are the co-authors of this paper. They all helped me brainstorm the idea, design the empirical study, and improve the presentation of the paper.

In many bug tracking systems, Normal is provided as the default. This may raise questions about the validity of a Normal severity label. Indeed, the person who files a bug report may be an ordinary user who has no expertise in the implementation of the affected software, or even no technical expertise at all. Such a person may find it difficult to accurately assess the severity of a bug. Thus, the bug reporter might not fill in the severity field, leaving it at its default Normal value. As a result, studies that use the severity field to investigate if there exists any relationship between bug severity and factors such as bug-fix time, amount of discussion, etc. are open to criticism that the results found in the Normal case may be invalid. Simply excluding the Normal reports, however, may distort the results in the opposite direction, if the Normal reports represent a large percentage of the available data.

These issues have been highlighted in a number of research studies. For example, in their two studies of severity prediction, Lamkanfi et al. [69, 70] excluded all the normal bugs stating: "*In our case, the normal severity is deliberately not taken into account. First of all because they represent the grey zone, hence might confuse the classifier. But more importantly, because in the cases we investigated this "normal" severity was the default option for selecting the severity when reporting a bug and we suspected that many reporters just did not bother to consciously assess the bug severity"*. Similarly, Tian et al. [145] excluded Normal bugs stating: "*Following the work of Lamkanfi et al., we do not consider the severity label normal as this is the default option*". Along the same lines, in the submission of our previous work on "long lived bugs" [123], when we drew our conclusions that most long lived bugs were important and adversely affected users' normal working experience, all three reviewers expressed their concerns:

Reviewer 1: *"since you used data from the severity field, I would suggest to discuss the fact that the level of this field could be somewhat subjective."*

Reviewer 2: *"In most cases, 'Normal' is the 'default' value of the severity, thus most of the users reporting a bug leave the default value since they simply don't know or are not*

interested in precisely defining the value."

Reviewer 3: *"Firstly we have to agree with the finding that eclipse severity is meaningful. If it is then you can cite other work that shows it to be meaningful, otherwise this claim does not hold up."*

A researcher is thus faced with a dilemma: either include information that may be unreliable, or discard potentially valuable information. To the best our knowledge no study has investigated either the amount of noise in the severity data (except *w.r.t.* enhancements [3, 46]) or the amount of value in this information.

In this dissertation, to better understand how severity information can be used, we investigate the following hypotheses, summarizing the apparent current consensus, as reflected by the above citations:

H1: Normal bugs do not reflect the actual severity level.

H2: Bug reports are mislabeled as Normal because reporters do not bother to change the default (Normal) severity.

Furthermore, we investigate the reasons for these problems and their impact in a representative software-engineering application. Our analysis is carried out in the context of four systems from the Eclipse product family. These are open source systems, that have publicly available bug databases, and that have been used in a number of previous software engineering studies [69, 70, 123, 145].

Our findings indicate that:

- Around 80% of the bugs reported in the studied software projects are classified as Normal. Excluding them from any automatic software engineering techniques could substantially distort the results.

- A manual reclassification of 500 Normal bugs in the studied software projects by pairs of students showed that 65% of the Normal bugs are not normal. Indeed, almost 25% of the Normal bugs are severe. These results support Hypothesis 1.

- Contradicting Hypothesis 2, we find the main reason for misclassifications in the Normal bugs is not that it is the default severity level. Rather, this field is very subjective and thus users may follow different criteria. Indeed, the pairs of students provided different opinions for more than half of the Normal bugs. We provide a taxonomy of the most common rationales used in these dissimilar assessments.

- The presence or absence of Normal bugs in training and test sets can significantly affect the actual and measured effectiveness of automatic software engineering techniques that rely on bug severity information. In our experiment with a basic bug severity predictor, we find that misclassification in the training data can reduce the accuracy of the severity prediction considerably. On the other hand, a tool accuracy excluding Normal bugs from both training and testing data is likely to be an overestimation if the tool is intended to be used on unlabeled data containing Normal bugs.

We conclude that while the classification of Normal reports is not very accurate, excluding them from software engineering studies can significantly distort the results.

4.2 Study Setup

4.2.1 Research Questions

Our study investigates the following research questions:

RQ1. What proportion of the bugs are Normal in the bug repository?

Motivation: The first question of any empirical study is how large is the population that we want to study. Indeed, if the population is small, there may be little reason to worry about it. For this study, our population of interest is the set of bug reports having the severity level Normal.

RQ2. What proportion of bug reports classified as Normal are actually "normal"?

Motivation: This is one of the main research questions of our study. The fewer bug reports classified as Normal that are actually "normal," the more effect this will have on the validity of any study that somehow depends on the bug report severity classification. This research question also addresses Hypothesis 1: *Normal bugs may not represent the actual severity level.*

RQ3. What are the main sources of misclassifications?

Motivation: To reduce misclassifications, we first need to understand the reasons behind them. Delineating the common reasons for misclassifications will help researchers or practitioners deal with the problem more systematically. This research question also addresses Hypothesis 2: *Bug reports are mislabeled as Normal because reporters do not bother to change the default (Normal) severity level.*

RQ4. Can misclassification or exclusion of Normal bugs affect previous study results?

Motivation: We investigate whether the noise in Normal bugs can affect tool results. If there is no impact, we would have little reason to worry about the issue.

4.2.2 Subject Systems

Our study focuses on four open-source projects, JDT, CDT, PDE, and Platform, from the Eclipse product family. A brief description of these systems was provided in Section 3.2.1. These projects are widely used in the real world, and have also been extensively used in software engineering research [69, 70, 123]. Furthermore, although these projects belong to the same product family, they are from various domains.

We have used Lamkanfi et al.'s [71] bug dataset, obtained from the Eclipse Bugzilla database,[25] to obtain the bug information associated with these projects. This dataset includes all the bug reports and their histories from the project inception to March 2011 for these four projects. Table 4.1 describes the dataset in more detail. Although this dataset is few years old, it was part of the MSR data showcase in 2013 and similar datasets have been

[25] https://bugs.eclipse.org/bugs/

Table 4.1: Data Set

System	#Change Requests (Bugs+Enh.)	#Bugs	#Enhancements	#Bugs Fixed
JDT	46,308	38,520	7,788	18,873
CDT	14,871	12,854	2,017	7,260
PDE	13677	11,958	1,719	6,854
Platform	90,691	78,120	12,571	33,738
Total	165,547	141,452	24,095	66,725

used in many studies [69, 70, 98, 123, 145]

4.3 Proportion of Normal Bugs

In this section, we investigate our first research question: *What proportion of bugs have Normal severity level?*

4.3.1 Methodology

A straightforward methodology would be to just compare the number of reports labelled Normal with the total number of reports. However, a bug tracking system may contain many invalid and duplicate bug reports, as well as feature requests or enhancements. There are also some bug reports that developers think are not worth fixing. In Bugzilla, the *status* and *resolution* fields together keep track of the current status of each bug. More specifically, the status field holds at most one of the values: `UNCONFIRMED`, `CONFIRMED`, `IN_PROGESS`, `RESOLVED`, and `VERIFIED`. The resolution field holds at most one of the values: `FIXED`, `INVALID`, `WONTFIX`, `DUPLICATE`, and `WORKSFORME`. Using this information, we extracted all the unique and valid bug reports, specifically those whose status field was set to either `RESOLVED` or `VERIFIED` and the resolution field was set to `FIXED`. We note that, as our bug reports date from 2011 at the latest, almost all of the reports have either been classified as uninteresting (`INVALID`, `WONTFIX`, `DUPLICATE`, or `WORKSFORME`) or have

Table 4.2: Proportion of Bugs by Severity

System	Blocker	Critical	Major	Normal	Minor	Trivial	Total
JDT	116	572	1,647	14,856	1,090	592	18,873
	0.6%	3.0%	8.7%	78.7%	5.8%	3.1%	100%
CDT	83	155	698	5,946	288	90	7,260
	1.1%	2.1%	9.6%	81.9%	4.0%	1.2%	100%
PDE	64	220	567	5,631	246	126	6,854
	0.9%	3.2%	8.3%	82.2%	3.6%	1.8%	100%
Platform	424	1,306	3,535	26,289	1,245	939	33,738
	1.3%	3.9%	10.5%	77.9%	3.7%	2.8%	100%

been fixed. We also removed all the reports marked as enhancements. We use the resulting set of reports in this and all of the subsequent research questions. Then, we counted all the bugs for each severity level.

4.3.2 Results

Table 4.2 provides detailed results regarding severity. For all of the considered systems, Normal is the dominant severity category, with 78-82% of the bug reports. The next most dominant category is Major, representing only 8-10% of the reports. Blocker bugs are rarest, at around 1%. The proportions of other types of bugs (Critical, Minor, and Trivial) are between 2% and 4% in most cases. Our results suggest that any research based on bug severity that ignores Normal bugs faces a severe threat to validity, since a large percentage of bug reports would likely not be taken into account.

4.4 Actual Severity of "Normal"-labeled Bugs

In the previous section, we saw that a large proportion of bugs are classified as Normal. However, we do not yet know if these Normal bugs are really normal according to the Eclipse Bugzilla definition. In this section, we investigate the second research question: *What proportion of bugs having Normal severity level is actually normal?*

4.4.1 Methodology

Since to the best of our knowledge, there is no clean dataset of bug reports that have actual severity levels, classifying bug reports using automated machine learning techniques would likely be inaccurate. Therefore, we conduct a manual investigation of their actual severity. Our methodology takes into account the fact that bug severity may be subjective, as well as the high cost of doing such an analysis.

Design

The severity field represents the impact of a given bug on the operation of the software, and thus it may be subjective. Indeed, even the Eclipse documentation mentions that the bug reporter's perspective on the severity can depend on how the bug reporter wants to use the software.[26] Therefore, to get reliable results, we have each bug report assessed by multiple users. In such a study, the cost depends on two factors: 1) the number of bug reports to be assessed, and 2) how many assessments are made.

To keep the cost reasonable, we made two decisions. First, we randomly selected a sample of 500 bug reports labelled Normal, representing 125 bug reports from each project, from within the last five years of our dataset, *i.e.*, from 2006 to 2011. Second, we recruited a group of assessors, such that each report would have at least two assessors. If the two assessors had different opinion about a given bug report, we analyzed both the report and the assessments to make a decision.

Users/Assessors Selection

All the assessors in our study are either graduate or undergraduate students of the University of Texas at Austin. We sent a general email to all the students in the senior "Software Engineering" class and to some graduate students in the software engineering track of the Electrical and Computer Engineering department. Good programming knowledge and sub-

[26] http://wiki.eclipse.org/Eclipse/Bug_Tracking

Table 4.3: Student's Qualifications

Description	Mean	Median	Min	Max
Coding Experience (in Years)	6.1	5.0	3.0	13.0
Experience in Java (in Years)	5.0	4.0	2.0	11.0
Experience with Eclipse (in Years)	4.6	3.5	2.0	11.0
Industrial Experience (in Months)	10.0	9.0	2.0	24.0

stantial experience in working with the Eclipse IDE were requirements to participate in the study. Based on these criteria, we selected 6 graduate students and 4 senior undergraduate students. 9 of them have experience in industry either as an intern or as a full-time programmer. Table 4.3 gives the students' qualifications in more detail.

Procedure

Our study was divided into two sessions: training and assessment. In the training session, we conducted a 30-minute tutorial. The tutorial included:

1. Providing a brief overview of our study,
2. Explaining a real Eclipse bug report and giving a brief overview of Bugzilla,
3. Showing how to submit a bug report in Eclipse, to show that Normal is the default severity level,
4. Explaining the definition of each severity level from the Eclipse documentation (described in Chapter 2),
5. Showing a representative example in each severity level to deepen their understanding about bug severity,
6. Explaining the structure of the expected feedback.

We divided the 500 bug reports into five sets of 100 bug reports each. Sets 1-4 had 100 bug reports from the same project and Set 5 had 25 bug reports from each of the four

projects. This strategy allowed most of the students to focus on a single project. We then assigned the students to five groups, pairing a graduate student with an undergraduate, when possible. The group members were not informed of each other's identity. Then we assigned each set of bug reports to a group randomly. The students were given 10 days to complete the assessments. We recommended to the students that they carefully read each part of the bug report, including at least the bug summary, the bug description, and the developers' comments, to make their decision. All of the students completed their task. After the study, each student was rewarded with a *$50 Amazon Gift Card*.

We note that the students have access to more information than the original bug reporter, as the students have access to the comments that were added after the original bug report was made. Our goal, however, is not to simulate the conditions under which bugs are reported, but to obtain accurate severity information. Furthermore, in the total of 1,000 assessments produced by the students, the students used the bug summary, bug description, and comments in 68%, 62%, and 39% of the cases, respectively. For only 15% of the bug reports did both students report that the comment information was helpful to make a decision, and for only 2% of the bug reports did the students only use the comments to make a decision. Therefore, for most bug reports, the students were able to make a decision based on the information available to the original reporter.

Feedback

We designed a Google form to receive students' feedback.[27] Our form contains: 1) the bug id, 2) the actual severity of the bug, 3) the specific reason for the decision (free text), 4) the parts (summary, description, and comments) of the bug report that helped make a decision, and 5) the assessor's name. All of the fields were required to submit a response. Students could provide "Not Sure" for the actual severity if they really were not sure about it.

[27] http://goo.gl/XP03JZ

Table 4.4: Similarity of Students' Response

System	Similar	Different	Proportion of Agreement
JDT	52	73	42%
CDT	62	63	50%
PDE	53	72	42%
Platform	43	81	35%
Total	210	289	42%

4.4.2 Results

We got 1000 responses from the students, comprising two responses for each bug report. There were only 16 responses where at least one student was undecided and thus chose "Not Sure". For only one bug report, from the Platform project, were both responses "Not Sure" due to insufficient information. We investigated all these 16 bug reports and were able to assign a severity level in 15 cases. However, we were not able to assign any severity level to the bug report where both responses were not "Not Sure". We eliminated this report, leaving the responses for 499 bug reports for analysis.

Among the 500 bug reports, there were only 164 reports, *i.e.*, 33%, for which both students gave the same severity level. To refine the results, we focused on the difference between Normal and the other severity levels. Like other work [69, 70], we merged the Blocker, Critical, and Major categories into a higher-level category, *Severe*, and the Minor and Trivial categories into a higher-level category, *Non-Severe* (*NS*). We refer to two responses that are in the same higher-level category as *similar*. After merging the categories, as shown in Table 7.3.1, we obtain 210 similar responses, amounting to 42% of the reports, leaving 289 where the students disagree. For Platform, the proportion of similar responses is only 35%. The highest rate of agreement (50%) is found for CDT. These results confirm that severity is highly subjective.

Given the high rate of dissimilarity among the severity levels provided by the students, we cannot use the results directly to obtain the proportion of Normal bug reports that

are actually normal. Instead, we consider the results from three perspectives: *best case*, *worst case*, and *optimal*. For the best case, *i.e.*, the most optimistic view of the state of the software, we take the lowest level of severity between the two students' responses. For example, if one student categorized a bug as Major and another categorized it as Normal, we would consider the actual severity to be Normal. On the other hand, for the worst case, *i.e.*, the most pessimistic view of the state of the software, we take the highest level of severity between the two responses. Finally, for optimal case, we investigated all the 289 bug reports where students' responses differed and tried to come to a consensus. We read all the bug reports (summary, description, and comments) and students' responses (assigned severity level and specific reason for assigning that level). Then, we made a decision by either taking the one of the students' responses that we agreed with, which was possible in most cases, or assigned a different severity level.

Tables 4.5 and 4.6 present the proportion of Normal reports that are classified by the students into the different high-level categories for the best, worst, and optimal cases. In all cases, the Enhancement columns are the same; since enhancements are not bugs, in both tables we use the optimal-case results. From the results, we can see that the proportion of Severe, Normal, and Non-Severe bugs could vary between 7%-34%, 32%-40%, and 15%-34% respectively, depending on how the results are calculated. More specifically, from Table 4.6, representing the optimal classification, we see that the actual proportion of Normal bugs among those originally labelled Normal is only 35%, and that 19% of the reports originally labelled as Normal are not reports of bugs at all. Furthermore, among the bugs originally labelled Normal, 24% are Severe and 22% are Non-Severe. For JDT, the proportion of Severe bugs is even higher, 33%. Among the 109 reports that are Non-Severe, 84 are Minor and only 25 are Trivial. Therefore, our overall results suggest that the dataset of Normal bugs has serious noise, validating Hypothesis 1: *Normal bugs may not represent the actual severity level*.

Table 4.5: Proportion of Normal Bugs at each Severity Level from the Best and Worst Case Perspectives

System	Best Case				Worst Case			
	Sev.	Norm.	NS	Enh.	Sev.	Norm.	NS	Enh.
JDT	14	48	48	15	50	40	20	15
	11%	38%	38%	12%	40%	32%	16%	12%
CDT	14	66	21	24	45	52	4	24
	11%	53%	17%	19%	36%	42%	3%	19%
PDE	5	46	45	29	40	31	25	29
	4%	37%	36%	23%	32%	25%	20%	23%
Platform	4	38	54	28	36	36	24	28
	3%	30%	43%	22%	29%	29%	19%	22%
Total	37	198	168	96	171	159	73	96
	7%	40%	34%	19%	34%	32%	15%	19%

Table 4.6: Proportion of Normal Bugs at each Severity Level from the Optimal Case Perspective

System	Same Response				Different Response				Total			
	Sev.	Norm.	NS	Enh.	Sev.	Norm.	NS	Enh.	Sev.	Norm.	NS	Enh.
JDT	13	19	20	0	28	21	9	15	41	40	29	15
	10%	15%	16%	0%	22%	17%	7%	12%	33%	32%	23%	12%
CDT	12	32	3	15	22	27	6	9	34	58	9	24
	10%	26%	2%	12%	18%	22%	5%	7%	27%	46%	7%	19%
PDE	4	16	25	8	12	30	9	21	16	46	34	29
	3%	13%	20%	6%	10%	24%	7%	17%	13%	37%	27%	23%
Platform	3	13	20	7	26	17	17	21	29	30	37	28
	2%	10%	16%	6%	21%	14%	14%	17%	23%	24%	30%	22%
Total	32	80	68	30	88	95	41	66	120	174	109	96
	6%	16%	14%	6%	18%	19%	8%	13%	24%	35%	22%	19%

4.5 Sources of Misclassification

In the previous section, we showed that 65% of Normal bugs are not actually normal according to the definition of the Eclipse severity levels. There may be numerous reasons for these misclassifications, from "leaving the severity field at its default value" to "too subjective to decide". In this section, we answer RQ3: *What are the main sources of misclassifications?*

4.5.1 Methodology

To understand the reasons for misclassifications, first we investigate the main severity levels that confused assessors. Then, we read all the bug reports where students' responses differed by high-level category (Severe, Normal, or Non-Severe). In almost all of the cases, it was possible to determine why the student chose a particular severity level by reading the reasons the student provided.

To categorize the common reasons for different responses, we analyzed all the bug reports, following an open-ended taxonomy. For a given bug report, first we identified the high-level reason for the difference, and then we checked whether the reason fits into any of the existing categories. If it did not, we created a new category. We provide concrete examples for each category below, to better understand the categorization procedure. We selected examples that: i) cover the range of subject systems, and ii) have a summary or description in the bug report that is concise enough to present in the dissertation.

4.5.2 Results

Table 4.7 shows the number of bug reports for each pair of dissimilar responses. For example, the value in the rightmost cell indicates that for 13 bug reports, one student's response is Trivial but the other's response is Enhancement. From the results we see that students were mostly confused between the Normal and Critical, Normal and Major, and Minor and Normal severity levels. Interestingly, Normal bugs are present in each confusion pair. Therefore, it is evident that even after careful assessment, users can be confused between

Table 4.7: Different Response Matrix

Blocker	0						
Critical	2	2					
Major	1	0	16				
Normal	6	3	43	67			
Minor	1	2	4	19	54		
Trivial	2	0	1	2	27	29	
Enhancement	4	1	1	6	19	11	13
	Not Sure	Blocker	Critical	Major	Normal	Minor	Trivial

Normal and other categories. The following summarizes our taxonomy of common reasons for the different responses.

Focusing on different aspects of a bug report

A bug report may describe several aspects of a bug. Different persons can focus on different aspects, causing them to map the bug to different severity levels. Consider the following example:

Platform # 210946, Description: *A caught Throwable is not written to the Eclipse log. It is just written on the console.*

One student thought that this bug is Severe since it involves losing data from the Eclipse log. Their rationale was that since information is normally written to both the console and the log, the user may close the console and rely only on the log file. Such a user would not even realize that some data were missing. On the other hand, the other student assigned a severity level Minor thinking that there is an easy workaround, since the Throwable is at least written on the console.

Same aspect but different perception

In some bug reports both students focused on the same aspect of the bug but their perception of it was different. For example:

CDT # 332915, Summary: *[tracepoint] Refreshing the Trace Control view blocks the UI thread.*

Bug Description: *We've noticed that when heavily using the tracepoint interface, deadlocks can happen due to the UI thread being blocked. Once [*sic*] case is that the refresh operation of the Trace Control view is done within a Query, which locks the UI thread.*

One student responded, "One feature enabled ends up impeding another feature - even though both features work in isolation." Thus, she chose Severe. The second student responded, "GUI and refresh are on the same thread". Thus, it is a regular issue and the student assigned a Normal severity.

Different acceptance or tolerance level

Sometimes users may have the same perception about the problem but a different level of tolerance to deal with it. For example:

Platform # 172321, Summary*[Commands] [GTK] Handler activation in editor when a dialog is closed is delayed*

One student thought that this is a "loss of functionality (delay time of feature) only on linux platform" and thus tagged it as Normal. The other student responded, "delay issue for activating handler is a major issue to me". So he chose Major.

Impact

Some bugs can seem to fall into the category Minor if the definitions of severity levels are strictly followed. However, the impact of the bug can be annoying enough to make the bug Normal or even Severe. For example:

JDT # 231887, Summary: *[actions] cannot refresh working sets through Package Explorer*

Bug Description: *Steps To Reproduce: 1. Import some Java projects and put them into some working sets. Change the Top Level Elements in the Package Explorer to Working Sets 2. Externally modify some of the files from different working sets 3. Select the working sets in the Package Explorer, right-click, and choose Refresh. Nothing happens. (Verify by opening files that have been modified - instead of opening the file, you get the "This file is out of sync" editor) 4. If you expand all the working sets and refresh the individual projects, it works.*

From the bug summary and description we can see that the user has provided a workaround. However, every time the user changes something, he must refresh each project related to the working set, which is annoying. Thus, one student marked the bug as Severe saying that it hinders the workflow. However, the other student responded, "Easy workaround. Not so much important bug." We also noticed these dissimilar responses when keyboard shortcuts do not work properly (e.g., Platform # 262593). Again, there is in principle an easy workaround, using menu commands, but some users may be so used to keyboard-shortcuts that they do not feel comfortable with the menu, causing them to view the bug as Severe.

Different cost of the same bug: development perspective vs. release perspective

This is the most frequent category in our sample, especially for those dissimilar responses where students are confused between Critical and another category. In our study, in most cases, when a report indicates a program crash, *e.g.*, due to a null pointer exception, the students marked it Critical, which is appropriate according to the definition of the severity levels. Examples include:

JDT # 325523, Summary: *NPE when deleting resource*

PDE # 275921, Summary: *NPE with update classpath*

However, in some cases, the students analyzed the bug from a developer's perspective and marked it as Normal. For example, they said that this was an easy fix or occurred in infrequent cases. Indeed, when an exception is thrown during development and testing, this can be considered as normal since this is a common mistake that developers can fix quickly. However, if a stable version crashes for a given task and the user must wait for the next update to get it resolved, the effect is a lot costlier than the development scenario.

Bug vs. Enhancement

Whether a given issue is a bug or an enhancement is subjective, and is thus itself a reason for misclassification. Furthermore, even if a reporter correctly classifies an enhancement, s/he may end up representing it incorrectly in the Bugzilla database due to the tricky Bugzilla configuration used by Eclipse. Indeed, in the configuration of Bugzilla used by the Eclipse projects there is no separate field for distinguishing bugs from enhancements. Instead, Enhancement is just one possible bug severity level. Therefore, if a change request is an enhancement, the reporter should set the severity label to Enhancement regardless of the request's importance. In practice, however, we found many cases where reporters marked enhancements as Major, Normal, or Minor depending on their perceived importance, thus implicitly misclassifying the change request as a Bug, not as an Enhancement. We discuss some examples:

Platform # 185067, Summary: *[KeyBindings] New Keys pref page: cannot sort 'User' column*

PDE # 330943, Description: *[plug-in registry] View initialization takes too much time*

In the first example, the issue reporter is asking for sort functionality to be added to the "new keys" preference page. Since what is asked for is a new feature, it is an enhancement, not a bug, even if the reporter finds it inconvenient or inconsistent that the feature is not currently available. Likewise, in the second example, there is no error in "View initial-

ization." Instead, the reporter is requesting that the performance be improved. However, in the students' assessment, one response for each bug was Minor, since the students thought that these issues would not affect users much.

4.5.3 Discussion for Hypothesis 2

We now investigate the existing Hypothesis 2, whether "leaving the severity field at its default value" is the main reason for severity misclassification. For this, we try to infer possible motivations from the experiences of our student assessors and from the bug report characteristics themselves.

Experiences of the student assessors: All the 500 bug reports in our manual investigation are marked as Normal in the Eclipse Bugzilla. However, for 65% of these reports, the student assessors had a different opinion of the severity from the original labeller. We try to understand why differences of opinion can occur by studying the differences of opinion that occurred within our manual study. Specifically, for 58% of the reports, each of the students evaluating the report assigned it a severity in a different higher-level category (e.g., severe, normal or NS). Careful investigation into the students' responses reveal that most of these discrepancies were due to the subjective nature of the assessment. There are indeed many factors to consider, such as minor or major loss of functionality, frequent or infrequent use cases, the convenience of any workaround, etc. Each of these factors is itself subjective, and our analysis in Section 6.5 shows that different choices by the students often resulted from their putting different weights on these different subjective criteria. As the students were told to pay careful attention to the evidence available in choosing a bug severity, but still often came up with different labels, it is possible that the original reporters were also paying attention to the choice of severity, but made choices that were different than those of the students.

Furthermore, we observe that our optimal strategy classifies only 3 of the selected 500 Normal reports as Blocker and only 25 of these reports as Trivial, implying that most

of the differences of opinion between the original reporter and the students was among the severity levels closer to Normal. These differences are again likely to be more subjective.

Report characteristics: We also performed a simple automatic analysis on all the fixed bug reports to get an idea about the proportion of bugs that should be Trivial but are categorized as Normal. We found that many of the reports classified as Trivial by our optimal strategy contain the keywords *typo*, *spell*, and *documentation*, either in the report or in the students' comment, and that these words appear rarely in the other reports in our sample. Text search of the summary and description of all the fixed Normal bugs in the complete dataset showed that only 1% of such bug reports contain these words. Again, we find little overlap between levels that are far apart, suggesting that misclassifications are between similar categories for which the differences are more subjective.

4.6 Misclassification or Exclusion of Normal Bugs: Do They Matter?

In this section, we investigate RQ4: *Can misclassification or exclusion of Normal bugs affect previous study results?*

4.6.1 Methodology

As we already noted, many previous studies use the bug severity field as a feature in various techniques such as bug-fix time prediction, modeling bug report quality, severity prediction, etc. A number of previous studies have ignored Normal bug reports, on the assumption that Normal does not correctly reflect the severity level. In this study, we have confirmed this assumption. However, a tool that has not been trained on Normal bugs may subsequently give meaningless results on Normal input. Such a tool is thus unusable in a real-world setting unless accurate severity information is already available. Therefore, we investigate two phenomena: 1) whether there can be any effect of misclassification on previous study

results, and 2) whether there can be any impact on the results if the Normal bugs are eliminated from the study.

To investigate the impact of these phenomena, we chose bug severity prediction as a representative application. Generally a bug severity prediction algorithm takes a set of bug reports known to be from various categories (e.g., Severe and Non-Severe) as training data and uses the properties inferred from this training data to predict the severity of bug reports in a test set. Lamkanfi et al. [69] showed that taking into account bug summaries is sufficient to get accurate results. Since our objective is to investigate the effect of misclassification or exclusion of Normal bugs on accuracy, not to propose new techniques for bug severity prediction, we have just implemented a simple approach. Our bug severity prediction system takes a set of bug summaries labelled with severity as a training set and predicts the severity of an input report represented by its summary. Our predictions are coarse-grained: Severe, Normal, and Non-Severe. We use Mallet's implementation of Naive Bayes out-of-the-box as our underlying classifier. Then we measure accuracy in terms of the proportion of correctly classified items. Specifically, the accuracy of our classifier is $m/n * 100\%$ if it classifies m instances correctly out of n instances. For a comprehensive description of bug severity prediction, please see [69, 70, 145].

Training and Test Set Creation: We distinguish between a *clean* dataset, in which we have good confidence that the severity labels are accurate, and a *noisy* dataset, in which it is not known whether the severity labels are accurate. Either kind of dataset furthermore may or may not contain Normal bugs. This leads to four training sets, TR_{clean}, TR_{noisy}, $TR_{clean} - Normal$, and $TR_{noisy} - Normal$, having various permutations of these properties. We train our severity prediction algorithm on each of these training sets, resulting in four instances of the tool. To assess the impact of noisy data and of excluding Normal bugs, we then test each of these tool instances on a clean test set, TE, and compare the accuracy of the resulting predictions with the known labels.

A challenge in our experimental methodology is obtaining sufficient clean data.

Indeed, our training and test sets must respect a number of constraints. First, the training set and the test set should be disjoint. Furthermore, previous research has shown that to avoid bias due to the over-prevalence of data in one class, all of the training sets (clean or otherwise) and the test set should have the same number of bugs at each severity level [55]. Specifically, if one class has few instances in the training set, any learning algorithm would know less about that class. Likewise, if a one class is very dominant in a test set, the evaluation result would be biased toward that class. For example, in a two-class (C_A and C_B) test set , if one class (C_A) represents 90% of instances and if a naive classifier says all the instances are C_A, its accuracy would be still 90%. Finally, Lamkanfi et al. [69] have found that a training set of 500 bug reports in each category gives stable results.

In addressing our previous research questions, we have manually investigated only 500 bug reports, and among them 120, 174, and 109 are classified as Severe, Normal, and Non-Severe, respectively. Using this dataset, and respecting the constraint that there should be the same number of bugs at each severity level, we can obtain a data set of at most only slightly over 300 elements. Concretely, we take the 100 most recent reports in each severity level, resulting in a dataset of 300 elements. As this does not satisfy the requirement of 500 reports in each category, we cannot use this as a training set. Thus, we use it as the test set, TE.

For the training sets, we need a low-cost way to obtain more data with reliable severity labels. For this, we focus on the bug reports in which the severity information has been changed at least once in any way. This may not result in a completely clean set, but it should be acceptable, since developers, who we assume to be experts in the software, have reviewed those bug reports and adjusted their severity level. Starting from the year 2006, we take the first 500 bug reports having this property in each severity category. For example, the training set of Normal bugs consists of those bug reports that ended up being Normal after one or more severity changes. The resulting set of 1500 reports makes up TR_{clean}. To create TR_{noisy}, we follow the same procedure, without the requirement of a change in

the severity information. Then, from TR_{clean} and TR_{noisy}, we obtain $TR_{clean}-Normal$ and $TR_{noisy}-Normal$, respectively, by removing the Normal reports. As we have taken the training data TR from the start of the time period and the test data TE from the end of the time period, they are likely disjoint. We have furthermore verified this in practice.

Finally, we note that all of the datasets contain bug reports from all four subject systems.

4.6.2 Results

We present our results in terms of the confusion matrix and accuracy. Precision, recall, and F-measure can all be calculated from the confusion matrix. Table 4.8 shows the impact of misclassification on the accuracy of the severity predictor trained on the full clean dataset TR_{clean}. In the results, each row is the number of predicted results for a given category. For example, the first row represents the 100 actual Severe bugs in TE. Of these, 57 are predicted to be Severe, 26 are predicted to be Normal, and 17 are predicted to be Non-Severe. Based on the actual severity level of bug reports in TE, the accuracy of our classifier is 49%. However, if we consider all the bug reports in TE to be Normal, as they are classified in bug repository, then the accuracy is only 29%.

We next perform the same experiment, but where all Normal bugs have been removed from the clean training set, producing $TR_{clean}-Normal$. Since the training dataset contains no Normal data, no bugs in the test set TE will be classified as Normal (see Table 5.7) and the accuracy of the resulting classifier is only 47%. However, if we also exclude Normal bugs from TE, as done in previous studies, the accuracy increases to 71%. Therefore, the accuracy reported in the existing literature is likely overestimated if the tool is intended to be used on unlabeled data that may contain Normal bugs.

Next, we repeat both experiments for the case of noisy training data. First, we consider TR_{noisy}, containing all categories of bugs. Table 4.10 shows that we obtain an accuracy of 41% for TE with this training data. This value is 8% less than that obtained

Table 4.8: Accuracy for Severity Prediction Classifier Trained from TR_{clean}

		Predicted			
		Severe	Normal	Non-Severe	
Actual	Severe	57	26	17	Accuracy: 49%
	Normal	35	38	27	
	Non-Severe	22	27	51	

Accuracy: 29% if we consider all the bug reports in TE to be Normal, as indicated in the bug repository.

Table 4.9: Accuracy for Severity Prediction Classifier Trained from $TR_{clean} - Normal$

		Predicted			
		Severe	Normal	Non-Severe	
Actual	Severe	75	.	25	Accuracy: 47%
	Normal	45	.	55	
	Non-Severe	33	.	67	

Accuracy: 71% if we exclude Normal bugs from TE

when we trained our classifier with TR_{clean}. Therefore, misclassification in the training data can reduce the accuracy of the severity prediction considerably.

Next, we consider $TR_{noisy} - Normal$, in which the reports labelled Normal have been removed. As compared to the use of the clean training set $TR_{clean} - Normal$, the accuracy only slightly declines, from 47% to 45% (Table 4.11), which is less than the decline between the results obtained using TR_{clean} and TR_{noisy}. Furthermore, as compared to TR_{noisy}, the accuracy actually improves, from 41% to 45%. This result indicates that the most noise is in Normal bugs. Finally, the reported accuracy again increases greatly if our test set does not include Normal bugs, reaching 67%. But knowing in advance whether

Table 4.10: Accuracy for Severity Prediction Classifier Trained from TR_{noisy}

		Predicted			
		Severe	Normal	Non-Severe	
Actual	Severe	34	49	17	Accuracy: 41%
	Normal	36	40	24	
	Non-Severe	22	29	49	

Table 4.11: Accuracy for Severity Prediction Classifier Trained from $TR_{noisy}-Normal$

		Predicted			
		Severe	Normal	Non-Severe	
Actual	Severe	63	.	37	Accuracy: 45%
	Normal	52	.	48	
	Non-Severe	29	.	71	

Accuracy: 67% if we exclude Normal bugs from TE

a bug is Normal is not a reasonable assumption for the input of a bug severity prediction tool.

Therefore, our overall results suggest that both misclassification and exclusion of Normal bugs may significantly affect any results based on bug severity.

4.7 Threats to Validity

Construct Validity: The set of bug reports is the only artifact used in our study; bug reports are generally well understood. We have also used well known metrics in our data analysis such as proportion and classification accuracy, which are straightforward to compute. We use a publicly available dataset, which enables the replication of this study. Therefore, we argue for a strong construct validity.

Internal validity: A bug report may be filed by either an Eclipse developer or a real user. The students who participated in our study are not involved in Eclipse development. However, they frequently use Eclipse for their own software development, e.g., research and class projects. Furthermore, 9 out of the 10 students had previous experience in industrial software development. Therefore, we believe that the students have the background necessary to assess bug severity.

All the bug reports used in our study are extracted from Bugzilla. There are many other bug tracking systems such as Jira, Mantis, etc. Since the severity levels may vary across projects and bug tracking systems, we may get different results for the bug reports in other bug tracking system.

To assess the severity of bugs, the students manually analyzed bug reports. To delineate the common reasons for misclassification, we also manually analyzed bug reports and students' responses. There might have been some unintentional misinterpretations during the manual verification due to the lack of domain knowledge. However, we held extensive discussions to minimize this threat.

To construct a clean training set, we selected bug reports whose severity had been changed at least once. Although we did not manually check that these bug reports have the actual level of severity, we believe the dataset should be fairly accurate since either bug triagers or developers examined those bug reports and adjusted the severity level accordingly.

External Validity: Eclipse may not be representative of all software. Still, it has been used in a number of studies, and so the conclusions drawn from it are at least applicable to those studies. Furthermore, we find similar results across the different Eclipse sub-projects.

Since manual investigation of bug reports is expensive, we investigated only 500 bug reports. We plan to conduct a more large-scale manual investigation in the future. Still, it has been shown that a sample of size 500 has suffient power to detect all but the smallest effects [119]. In our manual investigation, each report was assessed by two students. More assessments may further increase the confidence in the results. However, we separately investigated all the dissimilar responses. Therefore, this threat should have little impact on our results.

4.8 Related Work

Our work is related to the study of bug reports, and more specifically bug severity. We review some recent work that has relied on this information. Our work specifically investigates a source of bias or noise in bug reports. We also review some work that has investigated other such issues.

Bug reports are one of the main artifacts in software maintenance research. They have been used to understand various phenomena about software bugs and to design tools to help developers in the bug fixing process. Bettenburg et al. [11] investigated what kind of information developers think is the most helpful in a bug report. They also investigated the extents and reasons of duplicated bug reports [12]. Bortis et al. [21] proposed to tag bug reports automatically to help with bug triaging. Tian et al. [146] proposed a machine-learning based approach for assigning a priority to each bug report. Anvik et al. [4] and Shokripour et al. [135] proposed approaches for automatic assignment of bug reports to developers. Saha et al. [127] and Zhou et al. [173] proposed approaches for automatic bug localization. Huo et al. [54] investigated the role of expert and non-expert knowledge in bug reports and its impact on the results of bug localization tools.

Bug Severity is one of the key features of a bug report, to understand the bug's importance. Researchers have used this attribute in numerous software engineering studies. Menzies and Marcus [96] and Lamkanfi et al. [69, 70] proposed a text mining and machine learning based approach to predict bug severity. Tian et al. [145] also predicted bug severity, based on information retrieval. Bhattacharya et al. [13] proposed a graph-based approach to estimate bug severity. Hooimeijer and Weimer [51] used bug severity to investigate and model bug report quality. They concluded that self-reported severity is an important factor in the model's performance. Giger et al. [41] and Zhang et al. [166] used bug severity (with several other bug report features) to predict bug-fix time. Saha et al. [123] used severity to understand the importance of long lived bugs.

Bias or noise in bug-relevant datasets is a well-known problem in software engineering research. Bird et al. [19] investigated the potential biases in defect datasets in terms of bug features and commit features. They evaluated a popular defect prediction algorithm and showed that bug feature bias (e.g., unequal proportion of bug reports in terms of bug severity and developers' reputation) affects the performance of the algorithm. Later Nguyen et al. [101] confirmed that the bias in the bug-fix dataset exists not only in open-source

projects but also in the datasets of commercial projects. Kim et al. [64] proposed an algorithm to detect such noisy instances in bug datasets so that they can be eliminated. However, these studies did not investigate noise in bug severity. Rahman et al. [113] showed that consideration of the sample size of a dataset is as important as bias in the dataset. Antoniol et al. [3] showed that not all the bug reports in bug tracking systems are actually bugs. Later, based on a comprehensive manual investigation on 7,000 issue reports, Herzig et al. [46] reported that one-third of the bugs in the issue tracking systems are not actually bugs and this misclassification affects bug prediction algorithms. Kochhar et al. [66] investigated the potential biases in the dataset of mappings between bug reports and corresponding fixed files, and described their impact on bug localization.

4.9 Summary

In this Chapter, we have studied the mislabeling of Normal bugs, and the impact that it can have on tools that rely on bug severity. Based on the studied software projects, we confirm that the bugs labeled Normal are often not normal according to the bug repository criteria. Furthermore, we find that the inclusion or exclusion of these reports, as well as their consideration as Normal bugs or according to their actual severity, can have a major impact on the accuracy of tools that rely on bug severity values. This raises a real dilemma for the software engineering researcher. Normal reports are very prevalent, around 80% of the reports in our study, but cannot be relied on and are damaging to tool evaluations.

A partial solution is to create a clean dataset. Indeed, our results show that a bug severity prediction tool gives better results when trained on clean data than when trained on noisy data. We have proposed two approaches to creating a clean dataset: manual inspection and selecting only reports where the severity has been changed after the original submission. The former, however, is time-consuming, and the latter is more approximate. The wide use of bug reports by the software engineering community thus suggests that the community may want to invest resources into creating larger clean datasets.

We have also observed that misclassification of bugs and enhancements is a severe problem, which also may affect many studies. It appears that distinguishing enhancements from bugs through the severity field is not effective, because it does not allow the user to express the urgency of the enhancement request. Users could be less tempted to create noisy data if bug tracking systems such as Bugzilla would provide a dedicated field to separate bugs from enhancements. Our future work includes manually validating more Normal bug reports to create a large-scale clean dataset, and improving the state-of-the-art of severity prediction algorithms.

Data. Our data and results are publicly available at:
http://www.riponsaha.com/data/severity-assessments.csv.

Chapter 5

An Information Retrieval Approach for Regression Test Prioritization Based on Program Changes

This chapter is based on our paper, "An Information Retrieval Approach for Regression Test Prioritization Based on Program Changes", published in Proceedings of the 37th International Conference on Software Engineering [128].[28]

5.1 Context

Programs commonly evolve due to feature enhancements, program improvements, or bug fixes. Regression testing is a widely used methodology for validating program changes. However, regression testing can be time consuming and expensive [10, 74]. Executing a single regression suite can take even weeks [122] for some software. Regression testing is even more challenging in continuous or short-term delivery processes, which are now

[28] Please note that Dr. Lingming Zhang, Dr. Sarfraz Khurshid and Dr. Dewayne Perry are the co-authors of this paper. Dr. Zhang contributed in the experiments reproducing the results of existing tools. Dr. Khurshid and Dr. Perry both helped me brainstorm the idea, design the empirical study, and improve the presentation of the paper.

common practices in industry [53]. Therefore, early detection of regression faults is highly desirable.

Regression test prioritization (RTP) is a widely studied technique that ranks the tests based on their likelihood in revealing faults. RTP defines a test execution order based on this ranking so that tests that are more likely to find (new, unknown) faults are run earlier [35, 95, 158, 167, 171]. Existing RTP techniques are largely based on dynamic code coverage where the coverage from the previous program version is used to order, i.e., *rank*, the tests for running against the next version [35, 62, 158, 167]. A few recent techniques utilize static program analysis in lieu of dynamic code coverage [95, 171]. RTP techniques (whether dynamic or static) are broadly classified into two categories, *total* or *additional*, depending on how they calculate the rank [122]. *Total* techniques do not change values of test cases during the prioritization process, whereas *additional* techniques adjust values of the remaining test cases taking into account the influence of already prioritized test cases.

Although a number of RTP techniques (specifically coverage-based ones) have been widely used, they have two key limitations [95]. First, coverage profiling overhead (in terms of time and space) can be significant. Second, in the context of certain program changes (which modify behavior significantly) the coverage information from the previous version can be imprecise to guide test prioritization for the current version. Although the static techniques [171, 95] address the coverage profiling overhead, they simulate the coverage information via static analysis, and thus can be also imprecise.

This dissertation presents REPiR, an information retrieval (IR) approach for regression test prioritization based on program changes. Traditional IR techniques [87] focus on the analysis of natural language in an effort to find the most relevant *documents* in a collection based on a given *query*. Even though the original focus of IR techniques was on documents written in natural language, recent years have seen a growing number of applications of IR to effectively solve software engineering problems by extracting useful information from source code and other software artifacts [18, 77, 82, 90, 114]. The effectiveness of

these solutions relies on the use of meaningful terms (e.g., identifiers and comments) in software artifacts, and such use is common in most real world software projects. In the context of testing and debugging, the application of IR has been primarily focused on bug localization [114, 127, 173].

Our key insight is that in addition to writing good identifier names and comments in the code, developers use very similar terms for test cases, and we can utilize these textual relationships by reducing the RTP problem to a standard IR problem such that program changes constitute the query and the test cases form the document collection. Our tool REPiR embodies our insight. We build REPiR on top of the state-of-the-art Indri [140] toolkit, which provides an open-source, highly optimized platform for building solutions based on IR principles.

We compare REPiR against ten traditional RTP strategies [34, 36, 95] using a dataset consisting of eight open-source software projects. The experimental results show that for the majority of subjects REPiR outperforms all program-analysis-based and coverage-based strategies at both test-method and test-class levels. Thus, REPiR provides an effective alternative approach to addressing the RTP problem without requiring any dynamic coverage or static analysis information. Furthermore, unlike traditional techniques, REPiR can be made oblivious to the programming language at the expense of only 2% of accuracy, and may be directly applied to programs written in different languages. For reproducibility and verification, our experimental data is available online.[29] This chapter makes the following contributions:

- **REPiR**. We introduce a new approach for regression test prioritization (RTP) based on program changes. We define a reduction from the regression test prioritization problem to a standard information retrieval problem and present our approach, REPiR, based on this reduction.

- **Tool**. We embody our approach in a prototype tool that leverages the off-the-shelf,

[29] https://utexas.box.com/repir

state-of-the-art Indri toolkit for information retrieval.

- **Evaluation**. We present a rigorous empirical evaluation using the version history of eight open-source Java projects and compare REPiR with 10 other RTP strategies. We also present different variants of REPiR and provide detailed results on how REPiR can be used more effectively depending on test or program differencing granularities.

5.2 REPiR Approach

Regression test prioritization (RTP) and Information Retrieval (IR) both deal with a *ranking* problem, albeit in different domains. While RTP is concerned with test cases written in a programming language, IR is concerned with documents written in natural language. However, many human-centric software engineering documents are text-based, including source code, test scripts, and test documents. Furthermore, in real world software projects, developers often use meaningful identifier names and write comments, which allow solving a number of software engineering problems using information retrieval. Our key insight is that in addition to writing good identifier names and comments in the code, developers use very similar terms for test cases, and we can utilize these textual relationships using IR to develop effective and efficient RTP techniques.

5.2.1 Problem Formulation

We reduce RTP to a standard IR problem where the program difference between two software revisions or versions is the *query* and the test cases or test classes are the *document collection*. Therefore, for a given test suite TS, the prioritized test suite TS' is defined by ranking tests in TS based on the similarity score between the program differences and test cases. Reducing the RTP technique to a standard IR task enables us to exploit a wealth of prior theoretical and empirical IR methodology, providing a robust foundation for tackling RTP. This work primarily focuses on projects with JUnit tests.

5.2.2 Construction of Document Collection

The process of constructing documents from test cases varies depending on the information granularity and the choice of information retrieval techniques. Generally a test suite is a collection of source code files, where each source code file consists of one or test more test methods/functions. For example, JUnit has two levels of test cases: test classes and test methods. Prior research [167, 171] on RTP focused on prioritizing both test methods and test classes. In this section, we describe the construction of three types of document collections. Hereafter, we use test class to denote test class/file and test method to denote test method/function.

At Test Class Level

To make a document collection at test class level, we first build the abstract syntax tree (AST) of each source code file using Eclipse Java Development Tools (JDT), and traverse the AST to extract all the identifier names (class names, attribute names, method names, and variable names) and comments. The identifier names and comments are particularly important from the information retrieval point of view, since these are the places where developers can use their own natural language terms. Alternatively, a document collection at the test class level can be also constructed without any knowledge of underlying programming language. In this case, we do not construct any AST for test classes. Rather, we read each term in the test class, remove all mathematical operators using simple text processing, and tokenize them to construct the document-collection.

At Test Method Level

To make a document collection at the test method level, we extract all the methods from the AST using JDT and store all the identifiers and comments related to a given method as a text document.

5.2.3 Query Construction

As we defined in the problem formulation, in an IR-based RTP, the differences between two program versions comprise the query. How to choose the best query representation (e.g., succinct vs. descriptive) is a very well-known problem in traditional IR [73]. While the succinct representation often provides the most important keywords, it may lack other terms useful for matching. In contrast, although the more verbose descriptions may contain many other useful terms to match, it may also contain a variety of distracting terms. In our work, we experiment with three representations of program differences that can affect the overall results of RTP.

Low-Level Differences

By low level differences, we mean the program differences between two versions at the line level. We compute the low level differences by applying UNIX *diff* recursively while ignoring spaces and blank lines. The low level differences can also be directly obtained from version-control systems (e.g. cvs, svn, git) without additional computation. We denote this representation of a query as `LDiff` and quantify it in terms of the number of lines.

High-Level Differences

Since LDiff is expected to be noisy (e.g., sensitive to changes in formatting, or local changes), our goal is to summarize LDiff by abstracting local changes and ignoring formatting differences. To this end, we consider nine types of atomic changes (Table 5.1) that have been used in various studies for change impact analysis [42, 168, 169, 170]. We use FaultTracer [165], a change impact analysis tool, to extract these changes. We denote this high-level query as `HDiff` and quantify it in terms of the number of atomic changes.

Table 5.1: High Level Change Types

No.	Type	Change Description
1	CM	Change in Method
2	AM	Addition of Method
3	DM	Deletion of Method
4	AF	Addition of Field
5	DF	Deletion of Field
6	CFI	Change in Instance of Field Intializer
7	CSFI	Change in Static Field Initializer
8	LC_m	Look-up Change due to Method Changes
9	LC_f	Look-up Change due to Field Changes

Compact LDiff or HDiff

Since the difference between two program versions is often too long (e.g. thousands of lines), it is highly likely that it would have many duplicated terms. In this representation, we construct a compact version of LDiff and HDiff by removing all duplicated words. We denote these compact forms of LDiff and HDiff as `LDiff.Distinct` and `HDiff.Distinct`, respectively.

5.2.4 Tokenization

Since we are dealing with program source code rather than natural language written in English, similar to other IR systems for software engineering, our tokenization is different than that of standard IR tasks. Generally identifier names are a concatenation of words. Dit et al. [8] compared simple camel case splitting to the more sophisticated Samurai [10] system and found that both performed comparably in concept location. Therefore, in addition to splitting terms based on periods, commas, and white space, we also split identifier names based on the camel case heuristic.

5.2.5 Retrieval Model

Researchers in software engineering have experimented with a number of different IR models including latent semantic indexing (LSI) [29], the vector space model (VSM) [87], and Latent Dirichlet Allocation (LDA) [155]. However, recent research shows that the TF.IDF term weighted VSM (briefly TF.IDF model) works better than others [127, 173]. Another study shows that although there is a widespread debate on which of three (TF.IDF [130], BM25 (Okapi) [117], or language modeling [108]) traditionally-dominant IR paradigms was best, all three approaches utilize the same underlying textual features, and empirically perform comparably when well-tuned [38]. Therefore, we chose the TF.IDF model for our study. We elaborate on this TF.IDF formulation below.

Let us assume that test cases (documents) and a program difference (query) are represented by a weighted term frequency vector $\vec{d}$ and $\vec{q}$ respectively of length n (the size of the vocabulary, i.e., the total number of terms).

$$\vec{d} = (x_1, x_2, ..., x_n) \tag{5.1}$$

$$\vec{q} = (y_1, y_2, ..., y_n) \tag{5.2}$$

Each element x_i in $\vec{d}$ represents the frequency of term t_i in document d (similarly, y_i in query $\vec{q}$). However, the terms that occur very frequently in most of the documents are less useful for search. Therefore, in a vector space model, generally query and document terms are weighted by a heuristic TF.IDF weighting formula instead of only their raw frequencies. Inverse document frequency (IDF) diminishes the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely. Thus, weighted vectors for $\vec{d}$ and $\vec{q}$ are:

$$\vec{d_w} = (tf_d(x_1)idf(t_1), tf_d(x_2)idf(t_2), ..., tf_d(x_n)idf(t_n)) \tag{5.3}$$

$$\vec{q_w} = (tf_q(y_1)idf(t_1), tf_q(y_2)idf(t_2), ..., tf_q(y_n)idf(t_n)) \tag{5.4}$$

The basic formulation of IDF for term t_i is $idf(t_i) = log\frac{N}{n_{t_i}}$, where N is the total number of documents in C and n_{t_i} is the number of documents with term t_i. Therefore, in the simplest TF.IDF model, we would simply multiply this value by the term's frequency in document d to compute the TF.IDF score for (t, d). However, actual TF.IDF models used in practice differ greatly from this to improve accuracy [117, 136]. To date IR researchers have proposed a number of variants of the TF.IDF model. We adopt Indri's built-in TF.IDF formulation, based upon the well-established BM25 model [117, 164]. This TF.IDF variant has been actively used in IR community over a decade and rigorously evaluated in shared task evaluations at the Text REtrieval Conference (TREC). In this variant, the *document*'s tf function is computed by Okapi as:

$$tf_d(x) = \frac{k_1 x}{x + k_1(1 - b + b\frac{l_d}{l_C})} \tag{5.5}$$

where k_1 is a tuning parameter (≥ 0) that calibrates document term frequency scaling. For a small value of k_1, the *term frequency* value quickly saturates (i.e., dampens or diminishes the effect of frequency), whereas, a large value corresponds to using the raw term frequency. b is another tuning parameter between 0 and 1, which is the document scaling factor. When the value of b is 1, the term weight is fully scaled by the document length. For a zero value of b, no length normalization is applied. l_d and l_C represents the document length and average document length for the collection respectively. The IDF value is smoothed as $log\frac{N+1}{n_t+0.5}$ to avoid division by zero for the special case when a particular term appears in all documents.

The term frequency function of query, tf_q is defined similarly as tf_d. However, since the query is fixed across documents, normalization of query length is unnecessary. Therefore, b is simply set to zero.

$$tf_q(y) = \frac{k_2 y}{y + k_2} \tag{5.6}$$

Now the similarity score of document $\vec{d}$ against query $\vec{q}$ is given by Equation 6.7.

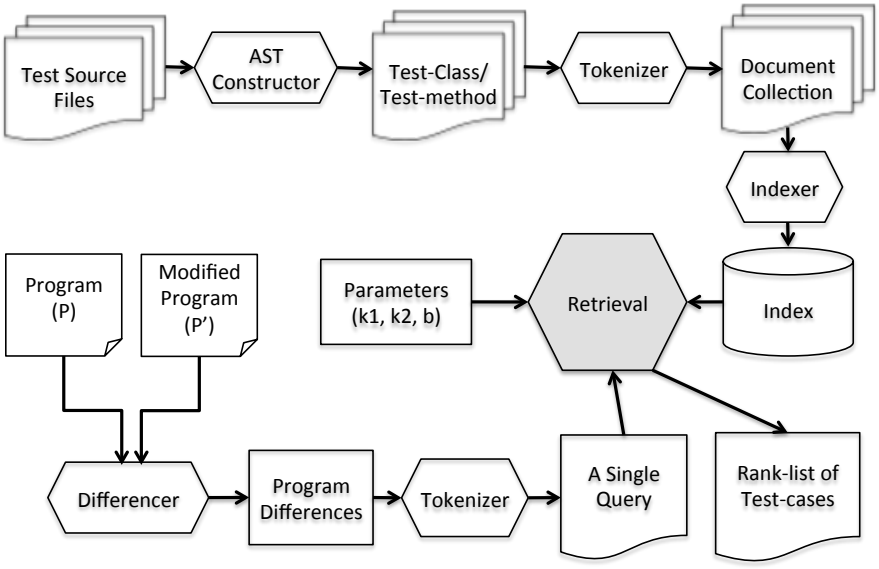

Figure 5.1: REPiR architecture

$$s(\vec{d}, \vec{q}) = \sum_{i=1}^{n} tf_d(x_i) tf_q(y_i) idf(t_i)^2 \tag{5.7}$$

5.2.6 Architecture

Figure 5.1 shows the overall architecture of our IR-based RTP prototype, REPiR (REgression test Prioritization using information Retrieval). First, REPiR takes the source code files of tests as input that we would like to prioritize. Next, it extracts information from test cases, tokenizes the terms, and constructs a document collection for a given level of granularity, as described in Section 5.2.2. REPiR also extracts program changes (LDiff, HDiff, or compact) and tokenizes terms in the same way as tokenizing document terms to construct the query.

We adopt the Indri toolkit [140], a highly efficient open-source library, for indexing and developing our retrieval model. After documents and queries are created above, they are handed off to Indri for stopword removal, stemming, and indexing. Note that we use the default stopword list provided with Indri and the Krovetz stemmer for stemming. We also

set the values of k_1, k_2, and b to 1.0, 1000, and 0.3 respectively, which have been found to perform well empirically [127].

5.3 Empirical Evaluation

To investigate the effectiveness of REPiR, we performed an empirical evaluation in terms of five research questions.

- **RQ1:** Is REPiR more effective than untreated or random test orders?

  This research question aims to understand whether REPiR reveals regression faults earlier than when there is no RTP or when test cases are ordered at random.

- **RQ2:** Do the high-level program differences help improve the performance of REPiR?

  Low-level program differences are expected to produce noisy results since changes to even a single character of a line would be interpreted as deletion of a line followed by an addition of line. Therefore, in this research question, we investigate whether the high-level program differences based on an AST improve the accuracy of REPiR.

- **RQ3:** How does REPiR perform compared to existing RTP techniques?

  To date researchers have focused on various program analysis (either static or dynamic) based techniques to propose or improve RTP. In this research question, we are interested in investigating how well REPiR performs compared to those existing techniques.

- **RQ4:** How does REPiR perform when it is oblivious to language-level information?

  Since REPiR utilizes only textual information, identification of specific programing language constructs is not needed. Lightweight language-specific parsing is used only for identifier-name extraction for constructing the document collection at the method-level. However, if we use LDiff to prioritize test-classes, REPiR can be

made completely oblivious of the underlying programming language. In this research question, we investigate how REPiR performs for this configuration.

5.3.1 Subject Systems

We studied eight open source software systems for our evaluation. These systems are from diverse application domains and have been widely used in software testing research [33, 95, 131]. We obtained Xml-Security and Apache Ant from the well-known Software-artifact Infrastructure Repository (SIR) [32] and download the other subject systems from their host website. The sizes of these systems vary from 5.7K LOC (Time and Money, 2.7K LOC source code and 3K LOC test code) to 96.9K LOC (Apache Ant, 80.4K LOC source code and 16.5K LOC test code).

For each subject system, we first extract all the major releases with their test cases and consider each pair of consecutive versions as a version-pair. For each pair, we run the old regression test suite on the new version to find possible regression test failures. Then, we treat the changes causing those test failures as the regression faults. In this way, we were able to identify 24 version-pairs with regression faults, which are all used in our study. Table 5.2 provides all the details regarding each version-pair, including the statistics for test methods, test classes, fault-revealing test methods, program edits (in terms of both LDiff and HDiff), and failure-inducing edits (in terms of HDiff). It also represents the textual properties of the first version in each version-pair, such as the number of distinct tokens extracted from source code and test cases, as well as the number and proportion of test-case tokens that also appear in source code. The high proportion (i.e., $>$48.8% for all subjects) of test-case tokens in source code confirms REPiR's motivation that developers use similar terms for tests and source code.

5.3.2 Independent Variable

In this study, we are interested in investigating the performance of REPiR for different granularities of test cases and different representations of program differences. We further

Table 5.2: Description of the Dataset

No.	Project	Version Pair	#TMeth	#TClass	#FTMeth	#LDiff	#HDiff	#FEdits	#SrcToken	#TstToken	# ComToken
P_1	Time and Money	3.0-4.0	143	15	1	1,200	215	1	276	303	183 (60.4%)
P_2	Time and Money	4.0-5.0	159	16	1	658	246	1	303	318	207 (65.1%)
P_3	Mime4J	0.50-0.60	120	24	8	4,377	2,862	3	494	293	194 (66.2%)
P_4	Mime4J	0.61-0.68	348	57	3	2,967	3,160	4	535	472	301 (63.8%)
P_5	Jaxen	1.0b7-1.0b9	24	12	2	2,788	204	3	368	117	93 (79.5%)
P_6	Jaxen	1.1b6-1.1b7	243	41	2	5,020	92	1	458	295	207 (70.2%)
P_7	Jaxen	1.0b9-1.0b11	645	69	1	1020	92	1	476	361	235 (65.1%)
P_8	Xml-Security	1.0-1.1	91	15	5	6,025	329	2	905	396	317 (80.1%)
P_9	XStream	1.20-1.21	637	115	1	833	209	1	698	928	470 (50.6%)
P_{10}	XStream	1.21-1.22	698	124	2	1,079	222	2	712	967	492 (50.9%)
P_{11}	XStream	1.22-1.30	768	133	11	5,920	540	11	740	1,029	512 (49.8%)
P_{12}	XStream	1.30-1.31	885	150	3	2,630	416	3	780	1,122	547 (48.8%)
P_{13}	XStream	1.31-1.40	924	140	7	6,744	1,225	7	796	1,146	559 (48.8%)
P_{14}	XStream	1.41-1.42	1200	157	5	828	136	5	835	1,206	595 (49.3%)
P_{15}	Commons-Lang	3.02-3.03	1698	83	1	1,757	221	1	906	925	603 (65.2%)
P_{16}	Commons-Lang	3.03-3.04	1703	83	2	3,003	172	2	913	924	603 (65.3%)
P_{17}	Joda-Time	0.90-0.95	219	10	2	8,653	5,976	2	450	236	130 (55.1%)
P_{18}	Joda-Time	0.98-0.99	1932	71	2	13,735	1,254	2	573	401	289 (72.1%)
P_{19}	Joda-Time	1.10-1.20	2420	90	1	1,348	793	1	606	501	358 (71.5%)
P_{20}	Joda-Time	1.20-1.30	2516	93	3	1,979	571	3	618	516	369 (71.5%)
P_{21}	Apache Ant	0.0-1.0	112	28	5	7,766	2,071	3	1,169	170	144 (84.7%)
P_{22}	Apache Ant	3.0-4.0	219	52	3	40,102	5,752	3	1,895	327	287 (87.8%)
P_{23}	Apache Ant	4.0-5.0	520	101	2	7,760	586	3	2,419	646	522 (80.8%)
P_{24}	Apache Ant	6.0-7.0	558	105	12	36,749	5,019	7	2,458	671	545 (81.2%)

investigate how REPiR works compared to other RTP strategies. Therefore, we have mainly three independent variables:

- **IV1**: Test-case Granularity
- **IV2**: Program Differences, and
- **IV3**: Prioritization Strategy

In Section 5.2, we discussed different test cases granularities and program differences. Now we briefly describe the 10 test prioritization strategies that we considered for comparison.

Untreated test prioritization keeps the original sequence of test cases as provided by developers. In our discussion, we denote the untreated test case prioritization as `UT`. We consider this to be the *control* treatment.

Random test prioritization rearranges test cases randomly. Since the results of the random strategy may vary a lot for each run, we applied random test prioritization 1000 times for each subject according to Arcuri et al.'s guidelines to evaluate randomized algorithms [6]. In our discussion, we denote the random test prioritization technique as `RT`.

Dynamic coverage-based test prioritization varies depending on the types of coverage information (e.g., the method or statement coverage) and prioritization strategies (e.g., the *total* or *additional* strategy). We used the four most-widely used variants of coverage-based RTP: `CMA`, `CMT`, `CSA`, and `CST`. For example, `CMA` denotes test prioritization based on *Method* coverage using the *Additional* strategy, and `CST` denotes test prioritization based on *Statement* coverage using the *Total* strategy.

JUPTA [95, 171] is a static-analysis-based test prioritization approach that ranks tests based on test ability (`TA`). TA is determined by the number of program elements relevant to a given test case (`T`), which is computed from the static call graph of T to simulate coverage information. TA can be calculated based on two levels of granularity: fine granularity and coarse granularity. TA at the fine-granularity level is calculated based on the

number of statements contained by the methods transitively called by each test, whereas TA at the coarse-granularity level is calculated based on the number of methods transitively called by each test. Similar to coverage-based prioritization techniques, we also used four variants of JUPTA: `JMA`, `JMT`, `JSA`, and `JST`.

Note that we implemented all the static and dynamic RTP techniques using byte-code analysis. More specifically, we used the *ASM byte-code manipulation framework*[30] to extract all the static and coverage information for test prioritization.

5.3.3 Dependent Variable

We use the Average Percentage Faults Detected (APFD) [122], a widely used metric in evaluating regression test prioritization techniques, as the dependent variable. This metric measures prioritization effectiveness in terms of the rate of fault detection of a test suite, and is defined by the following formula:

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{n \times m} + \frac{1}{2 \times n} \tag{5.8}$$

where n denotes the total number of test cases, m denotes the total number of faults, and TF_i denotes the smallest number of test cases in sequence that need to be run in order to expose the i^{th} fault. The value of APFD can vary from 0 to 1. Since n and m are fixed for any given test suite, a higher APFD value indicates a higher fault-detection rate.

5.3.4 Study Results

In this section, we present the experimental results which answer our research questions.

RQ1: REPiR vs. UT and RT

First, to understand the performance of REPiR compared to UT and RT at the test-method level, REPiR is set to construct the document collection at test-method level and use LDiff

[30] http://asm.ow2.org/

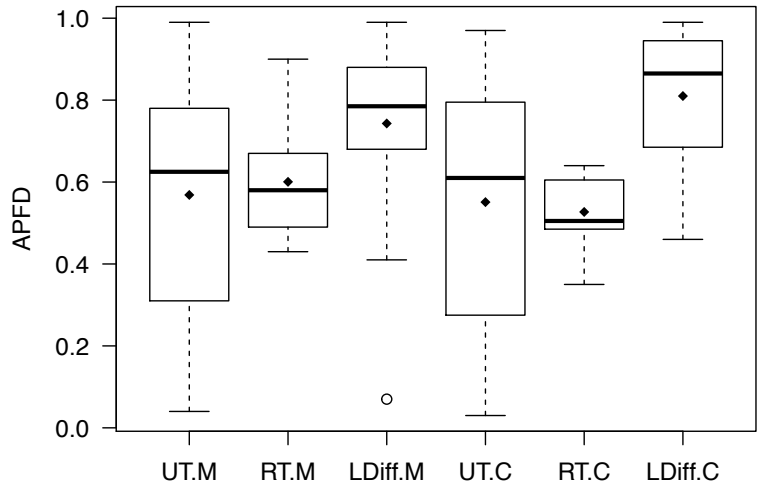

Figure 5.2: Accuracy of REPiR (LDiff) at test-method and test-class levels

as a query. We select the unstructured retrieval model and run REPiR on all version pairs (P_1-P_{24}). We also run RT and UT for the same dataset. Each RTP technique provides a ranked list of test-methods for each version pair, which we use to calculate APFDs. We also perform the same experiment for the test class level.

Figure 5.2 presents the results for all version-pairs in the form of boxplot. In each plot, the X-axis shows the strategies that we compared and the Y-axis shows the APFD values. To name RTP techniques, we used M to denote method-level and C to denote class-level test-cases. Each boxplot shows the average (dot in the box), median (line in the box), upper/lower quartile, and 90th/10th percentile APFD values achieved by a strategy. From the figure, we see that the mean, median, first and third quartiles APFD of (UT, RT, REPiR) at test-method level are (0.52, 0.60, **0.73**), (0.49, 0.55, **0.77**), (0.25, 0.50, **0.64**), and (0.72, 0.67, **0.87**) respectively, which clearly indicates that REPiR overall performs better than UT and RT.

We also investigate whether the accuracy of REPiR varies with the length of program differences or the number of test-methods, since these are the two main inputs for

REPiR. We compute the Spearman correlation between the size of LDiff (quantified by number of changed lines) and APFD, and between the number of test-methods and APFD. The low correlation values for both cases (0.23 and 0.2) indicate that the accuracy of REPiR is fairly independent of the length of program differences and the number of test-methods.

Similarly for the test-class level, we see that the mean, median, first and third quartiles APFD of REPiR (**0.79**, **0.79**, **0.66**, **0.94**) are higher than those of UT (0.52, 0.55, 0.3, 0.66) and RT (0.52, 0.5, 0.49, 0.53). These results show that REPiR performs much better than UT and RT at test-class level. The low Spearman correlations between the number of test-classes and APFD (0.24) and between the length of program differences and APFD (-0.49) indicate that the accuracy of REPiR is not dependent either on the number of test-classes or the size of program differences.

RQ2: Impact of Program Differencing Strategies

To answer RQ2, we run REPiR with four forms of program differences (LDiff, LDiff.Distinct, HDiff, HDiff.Distinct) separately for both at test-method level and at test-class level. Figure 5.3 presents the summary of APFD values for test-methods and test-classes respectively. Results show that at method-level, HDiff slightly works better than LDiff in terms of both mean (HDiff: 0.75 vs. LDiff: 0.73) and median (HDiff: 0.79 vs. LDiff: 0.77). When we take a closer look at our data for individual program versions, we find that HDiff improves the APFD values for 14 version-pairs, but decreases it for 7 version-pairs. However, if we further condense the query by removing duplicate terms, the accuracy decreases for both HDiff and LDiff, and the decrease rate is larger for HDiff than for LDiff. We believe that since HDiff is already condensed, it is affected more by the removal of duplicated terms than LDiff.

On the other hand, we see that LDiff works slightly better than HDiff, on average, at test-class level. The mean value of APFDs for HDiff are 0.75, whereas it is 0.79 for LDiff, although interestingly the median APFD for HDiff is 0.01 higher than that of LDiff.

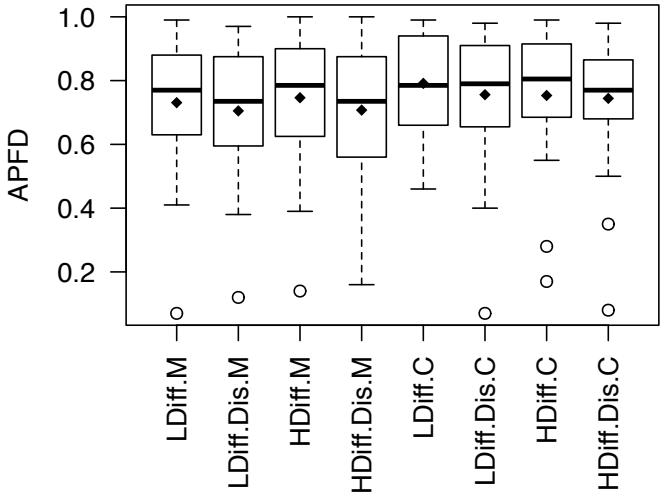

Figure 5.3: Impact of program differences at test-method (M) and test-class (C) levels (Dis=Distinct)

We think the reason for getting better accuracy with HDiff at test-method level is that since test-methods are typically small, they are more affected by the noise of LDiff. From the results of LDiff.Distinct and HDiff.Distinct, we see that like test-method level, the removal of duplicated terms also slightly hurts the results. However, it should be noted that REPiR runs faster when it uses the compact representation of the query and thus this representation may be suitable when the program change is large.

RQ3: REPiR Vs. JUPTA or Coverage-based RTP

To answer RQ3, first we ran all the eight techniques (four variants of coverage-based technique and four variants of JUPTA based on call graphs) described in Section 5.3.2 on each program-pair in our dataset. Figures 5.4 and 5.5 show the summary of APFD values for each strategy at method-level and class-level respectively. Results show that for both static and dynamic techniques, *additional* strategies are overall more effective than *total* strategies. This is consistent with prior studies [59, 167].

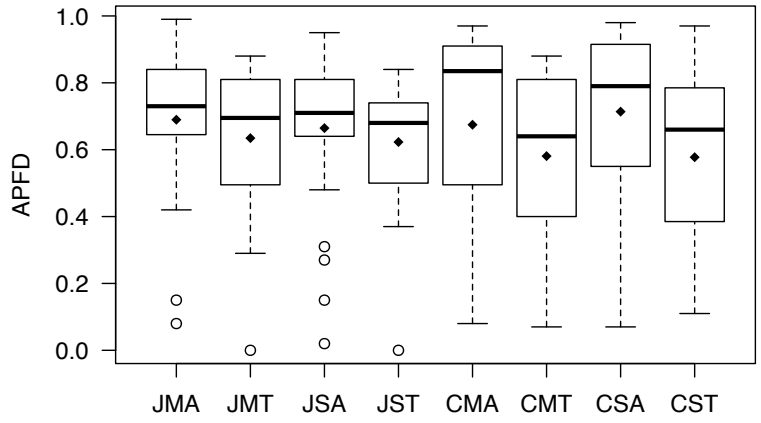

Figure 5.4: Accuracy of JUPTA and coverage-based RTP at test method level

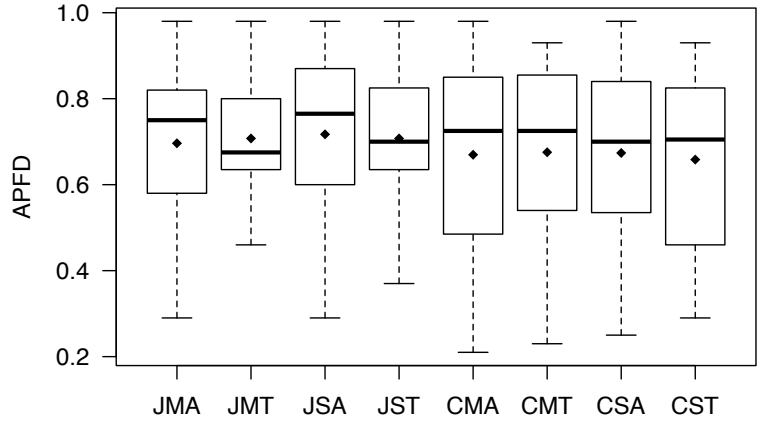

Figure 5.5: Accuracy of JUPTA and coverage-based RTP at test class level

Now we compare all the mean APFDs of REPiR (from Figure 5.3) with that of JUPTA and coverage-based techniques (from Figures 5.4 and 5.5) as summarized in Table 5.3. From the results, we see that REPiR equipped with either LDiff or HDiff overall outperforms all the JUPTA and coverage-based approaches (*total* or *additional*) regardless of test case granularities (test method/test class). At test method level the mean APFD achieved by REPiR are 0.73 using LDiff and 0.75 using HDiff, whereas the best variants of JUPTA (JMA) and coverage-based technique (CSA) achieve 0.69 and 0.71, respectively. At Test class level, REPiR achieves the mean APFD of 0.79 using LDiff and 0.75 using HDiff,test whereas the best variants of JUPTA (JSA) and coverage-based technique (CMT) achieve 0.72 and 0.68, respectively. Even with the compact representation of queries (LDiff.Dis and HDiff.Dis), REPiR performs better than all *total* strategies, and performs equally well or better than the *additional* strategies.

We further investigate how REPiR performs for each subject system. For conciseness, we present the *average* APFD values for each system achieved by REPiR using HDiff, JMA (JUPTA based on method coverage using additional strategy), which is the best among all the JUPTA strategies, and CSA (based on statement coverage using additional strategy), which is the best among all the coverage-based strategies at method level. Similarly, for test class level, we present the average APFD values for each subject system achieved by REPiR using LDiff, JSA, and CMT, which are the best in IR, JUPTA, and coverage-based approach respectively. Table 5.4 shows that REPiR achieved the best APFD for six/five out of eight subjects at test method/test class level.

RQ4: Accuracy when REPiR is oblivious of the Programming Language

For this experimental setting, REPiR does not build any AST for source code or test classes while constructing document collections and queries. Documents are made at test-class level by simply removing mathematical operators and tokenizing any text that is in the test-classes. The resulting documents are expected to be noisy because of the presence

Table 5.3: Comparison of Mean APFDs achieved by Different Strategies (TM=Test Method,TC=Test Class)

	TM	TC		TM	TC		TM	TC
LDiff	0.73	0.79	JMA	0.69	0.70	CMA	0.67	0.67
LDiff.Dis	0.71	0.76	JMT	0.63	0.71	CMT	0.58	0.68
HDiff	0.75	0.75	JSA	0.66	0.72	CSA	0.71	0.67
HDiff.Dis	0.71	0.74	JST	0.62	0.71	CST	0.58	0.66

Table 5.4: Comparison by Subjects

	Test Method			**Test Class**		
Subject	**HDiff**	**JMA**	**CSA**	**LDiff**	**JSA**	**CMT**
Time and Money	0.50	0.47	0.19	0.82	0.91	0.41
Mime4J	0.68	0.68	0.59	0.89	0.79	0.65
Jaxen	0.67	0.67	0.94	0.61	0.57	0.82
XML-Security	0.80	0.42	0.69	0.90	0.77	0.37
XStream	0.84	0.68	0.79	0.87	0.83	0.76
Commons-Lang	0.95	0.79	0.86	0.96	0.62	0.83
joda	0.75	0.87	0.78	0.63	0.66	0.72
Apache Ant	0.7	0.54	0.65	0.7	0.51	0.54

of programming language keywords. We used LDiff as the query, which is also program language independent. Then we run REPiR for all version-pairs and calculate the APFD values. The results show that the mean APFD across all version-pairs is 0.77, while it was 0.79 when we used only identifiers and comments as document terms. Interestingly, it turns out that the median APFD of the language oblivious approach is 0.01 higher: 0.8 for language-oblivious configuration vs. 0.79 when we used only identifiers and comments of test classes. Therefore, our results show that REPiR, even in its simplest form when it is oblivious of the programming language, does not lose any significant accuracy and outperforms all the JUPTA and coverage-based approach (highest mean is 0.72 achieved by JSA).

5.3.5 Qualitative Analysis

Our quantitative results already show that developers tend to use very similar terms in source code and corresponding test cases, which is one of our main motivations for developing an IR-based RTP. In this section, we illustrate a concrete example to show the usefulness of this information.

When Commons-Lang evolved from version 3.02 to 3.03, the test method `FastDateFormatTest.testLang538` failed since the developer incorrectly removed a conditional block for updating the time zone in the method `FastDateFormat.format()` (shown in Figure 5.6, highlighted in red). If we extract the program differences from this change, LDiff produces the following terms: time, zone, forced, calendar, get etc., while HDiff produces `CM:FastDateFormat.format`. It should be noted there were also many other (non-faulty) changes in the query. Now let us take a look at the test method that reveals this fault in Figure 5.7. Interestingly, we see many of the terms from faulty edits in the test method (highlighted in bold). Furthermore, the source code class (`FastDateFormat`) and the corresponding test class (`FastDateFormatTest`) have similar names. As a result, REPiR with HDiff ranked this method at 7th and LDiff ranked at 17th position among 1,698 test methods. On the other hand, the best variants of JUPTA and coverage-based technique, JMA and CSA ranked it at the 367th and 370th position respectively.

In spite of such good results, there was one occasion, where REPiR performed unsatisfactorily–when Time&Money evolved from version 4.0 to version 5.0. We found that

```
public StringBuffer format(Calendar calendar,
         StringBuffer buf) {
-        if (mTimeZoneForced){
-                calendar.getTimeInMillis();
-                calendar = (Calendar) calendar.clone();
-                calendar.setTimeZone(mTimeZone);
-        }
         return applyRules(calendar, buf);
}
```

Figure 5.6: A failure-inducing edit in Commons-Lang 3.03

```
public void testLang538() {
    final String dateTime = "2009-10-16T16:42:16.000Z";
    // more commonly constructed with: cal = new GregorianCalendar(2009, 9, 16, 8, 42, 16)
    // for the unit test to work in any time zone, constructing with GMT-8 rather than default locale time zone
    GregorianCalendar cal = new GregorianCalendar(TimeZone.getTimeZone("GMT-8"));
    cal.clear();
    cal.set(2009, 9, 16, 8, 42, 16);
    FastDateFormat format = FastDateFormat.getInstance("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'", TimeZone.getTimeZone("GMT"));
    assertEquals("dateTime", dateTime, format.format(cal));}
```

Figure 5.7: A fault-revealing test method for Commons-Lang 3.02

the fault revealing test method was `MoneyTest.testPrint` where there was apparently no information of use to IR, since the only line in the test method is `assertEquals("USD 15.00",d15.toString())`. However, our overall results show that the number of such occasions is very small (1 out of 24 cases in our study).

5.3.6 Time and Space Overhead

The running time of REPiR depends on three parameters: the size of vocabulary, the number of test cases, and the length of program differences. REPiR works most efficiently when we use a compact representation of the query. It takes only a fraction of a second for each version pair to prioritize its test cases. For example, REPiR took only 0.18 second to prioritize all the test methods of Joda-Time 1.20, which has the largest number of test methods (2,520 test methods) in our study. The preprocessing and indexing took only three seconds. When we used the full representation of query, REPiR took 20 seconds. On the other hand, the *additional* strategy based on statement level coverage information took 40 seconds, only for test prioritization (*excluding instrumentation and coverage collection*). The time complexity of REPiR and total strategy grows *linearly* when the test suite size increases, while the additional strategy grows *quadratically* [95]. Thus, REPiR is a more cost-effective approach. Furthermore, note that coverage-based approach is not useful if the coverage is not available from the old version because developers can simply run all the tests instead of spending time for recollecting the coverage.

The space overhead of REPiR is determined by the requirement of indexing test cases for IR. For most of the subjects, the index size is around 1MB. The index size of Joda-Time 1.20, which is the project that has the largest number of test methods, is 3MB.

On the contrary, the data required by the traditional techniques for the same system was 11.6MB for method coverage matrix and 31MB for the statement coverage matrix. The time and space overhead of JUPTA is very similar to that of coverage-based approaches since JUPTA tries to simulate code coverage. All the experiments were performed on a MacBook Pro running OS X 10.8 with Intel Core i7 CPU (2.8GHz) and 4GB RAM.

5.3.7 Threats to Validity

This section discusses the validity and generalizability of our findings.

Construct Validity: We used two artifacts of a software repository: program source code and test cases, which are generally well understood. Our evaluation uses subject systems with both real and seeded (for the projects from SIR) regression faults. Also we applied all the prioritization techniques on the same dataset, enabling fair comparison and reproducible findings. To evaluate the quality of prioritization, we chose APFD, which has been extensively used in the field of regression test prioritization, and is straightforward to compute. APFD expresses the quality of prioritized test cases based on how early the faulty test cases are positioned in the prioritized suite. However, APFD does not consider either the execution time of the individual test cases or the severity of the faults. Therefore, it may not accurately estimate how much we are gaining from the prioritized test suite in terms of costs and benefits.

Internal Validity: The success of REPiR vastly depends on the usage of meaningful and similar terms in source code and corresponding test cases, which is consistent with programming best practices. Another threat to internal validity is the potential faults in our implementations as well as the used libraries and frameworks. To reduce it, we have used mature libraries and frameworks that have been widely used in various software engineering and information retrieval applications (e.g., FaultTracer [165] and Indri [140]).

External Validity: Our experimental results are based on 24 versions of programs from eight software projects, all of which are open source projects written in Java with JUnit

tests. Although they are popular projects and widely used in regression testing research, our findings may not be generalizable to other open-source or industrial projects with other test paradigms. Note however that our technique does not have to rely on language specific features and therefore we expect it to handle programs in other languages. For example, in another study [125], we showed that the information retrieval based bug localization is equally effective in C programs. The risk of insufficient generalization could be mitigated by applying REPiR on more subject systems (both open source and industrial). This will be explored in our future work.

5.4 Related Work

Reducing the time and cost of regression testing has been an active research area for near two decades. Researchers have already proposed various regression testing techniques, such as regression test selection [8, 121], prioritization [35, 95], and reduction [120]. Since our work is for regression test prioritization (RTP), this section is limited to the relevant work in this area. For related work regarding IR in software engineering, please refer to Section 2.

Wong et al. [158] introduced the notion of RTP to make regression testing more effective. They made use of program differences and test execution coverage from the previous version, and then sorted test cases in order of increasing cost per additional coverage. Rothermel et al. [122] empirically evaluated a number of test prioritization techniques, including both the *total* and *additional* test prioritization strategies using various coverage information. In that work, they also proposed the widely used APFD metric for test prioritization. Along the same line, Elbaum et al. [36] investigated more code coverage information, and incorporated the cost and the severity of each test case for test prioritization [35]. Jones and Harrold [60] argued that there are important differences between statement-level coverage and modified condition/decision coverage (MC/DC) for regression testing, and proposed test reduction and prioritization using the MC/DC information. Jeffrey and Gupta [56] introduced the notion of relevant slices in RTP. Their approach as-

signs higher weight to a test case that has larger number of statements (branches) in its relevant slice of the output. However, a common limitation of these techniques is that they require coverage information for the old version, which can be costly to collect or may not be available in the repository.

Besides investigating different types of coverage information, researchers have also proposed various other strategies for RTP. Li et al. [75] used search-based algorithms, such as hill-climbing and genetic programming, for test prioritization. Jiang et al. [59] used the idea of adaptive random testing for test prioritization. Zhang et al. [167] recently proposed a spectrum of test prioritization strategies between the traditional *total* and *additional* strategies based on statistical models. However, according to the reported empirical results [59, 167], the traditional *additional* strategy remains one of the most effective test prioritization strategies.

There are also some approaches that do not require dynamic coverage information. Srikanth et al. [139] proposed a value-driven approach to system-level test case prioritization based on four factors: requirements volatility, customer priority, implementation complexity, and fault proneness of the requirements. Tonella et al. [148] used relative priority information from the user, in the form of pairwise test case comparisons, to iteratively refine the test case ordering. Yoo et al. [162] further used test clustering to reduce the manual efforts in pairwise test comparisons. Ma and Zhao [84] distinguished fault severity based on both users knowledge and program structure information, and prioritized tests to detect severe faults first. All these approaches require inputs from someone who is familiar with the program under test, which may be costly and not always available. To avoid manual effort, Zhang et al. [171] proposed a static test prioritization approach, JUPTA, which extracts static call graph of a given test case to estimate its coverage. Later Mei et al. [95] extended the study and proposed more variants of JUPTA along the same lines. Recently, Jiang and Chan [58] proposed a static test prioritization approach based on static test input information. However, all these approaches try to use static information to simulate

code coverage, and thus may be imprecise. In contrast, REPiR is a fully automated and lightweight (does not require coverage collection or static analysis) test prioritization approach based on information retrieval, and we have shown it to be more precise and efficient than many existing techniques.

Like our approach, there are also some test prioritization techniques that utilize the presence of natural English in source code artifacts. Arafeen and Do [5] first clustered test cases based on requirements, which are written in English and then prioritized each cluster based on code metrics. Nguyen et al. [100] used an IR-based approach for prioritizing audit tests in evolving web services where they use service change descriptions to query against test execution traces. Both of these approaches require requirement-documents or manual change descriptions with past test execution traces, which may not be available. Furthermore, they are designed and intended to be used when requirements or services change and may not be well suited for day-to-day regression testing. Thomas [142] proposed a test prioritization approach using a topic model approach where test cases are ordered based on their edit-distances. This technique does not utilize any change information and thus may be imprecise.

5.5 Summary

To reduce the regression testing cost, researchers have developed various techniques for *prioritizing* tests such that the higher priority tests have a higher likelihood of finding bugs. However, existing techniques require either dynamic coverage information or static program analysis, and thus can be costly or imprecise. In this dissertation, we have introduced a new approach, REPiR, to address the problem of regression test prioritization by reducing it to a standard IR problem. REPiR does not require any dynamic profiling or static program analysis. We rigorously evaluated REPiR using a dataset consisting of 24 version-pairs from eight projects with both real and seeded regression faults, and compared it with 10 RTP strategies. The results show that REPiR is more efficient and outperforms

the existing strategies for the majority of the studied subjects. We also show that REPiR can be made oblivious of the underlying programming language for test-class prioritization, seldom losing accuracy. We believe that this alternative approach to RTP represents a promising and largely unexplored new territory for investigation, providing an opportunity to gain new traction on this old and entrenched problem of RTP. Moreover, further gains might be achieved by investigating such IR techniques in conjunction with traditional static and dynamic program analysis, integrating the two disparate approaches, each exploiting complementary and independent forms of evidence regarding RTP.

Chapter 6

Improving Bug Localization Using Structured Retrieval

This chapter is based on our paper, "Improving Bug Localization Using Structured Retrieval", published in Proceedings of the 28th International Conference on Automated Software Engineering [127].[31]

6.1 Context

Frederick Brooks wrote that "Software entities are more complex for their size than perhaps any other human construct because no two parts are alike (at least above the statement level)" [23]. Due to this inherent complexity of software construction, software bugs remain frequent. For a large software system, the number of bugs may range from hundreds to thousands.

[31] Please note that Dr. Matthew Lease, Dr. Sarfraz Khurshid and Dr. Dewayne Perry are the co-authors of this paper. Dr. Lease helped me brainstrom the idea and design the empirical study from the information retrieval perspective. In addition, he helped improve the writing of the paper significantly. Dr. Khurshid and Dr. Perry both helped me brainstorm the idea and design the empirical study from the software engineering perspective.

Large, widely used software projects typically combine many modules with different, interrelated functionalities and involve many developers. In this environment, it is difficult for a user that encounters a bug to know precisely where the bug occurs. Even a developer who reads a bug report may not immediately know what files are relevant, particularly if the report does not relate to his own code. If the code relevant to a bug report is not immediately apparent, the report can be ignored for a long time, or can be assigned to the wrong developer, wasting developer time [123]. Therefore, effective methods for locating bugs automatically from bug reports are highly desirable.

There are two general approaches for bug localization: i) dynamically locating the bug via program execution together with technologies such as execution and data monitoring, breakpoints etc. [1]; and ii) statically locating bugs via various forms of analyses using the bug reports together with the code [52]. The dynamic approach is often time consuming and expensive. The ease of the static approach, together with its immediate recommendation, make it appealing.

In recent years, information retrieval (IR) based bug localization techniques have gained significant attention due to their relatively low computational cost and minimal external dependencies (e.g., requiring only the source code and the bug report in order to operate). In these IR approaches, each bug report is treated as a *query*, and all the source files in the project comprise the *document collection*. IR techniques then rank the documents by predicted relevance, returning a ranked list of candidate source code files that may contain the bug. The fundamental assumption underlying these techniques is that some terms in a given bug report will be found in the source files needing to be fixed for that bug. Figure 6.1 presents a real world bug report from Eclipse 3.1 and corresponding source code fix, taken from Zhou et al. [173]. The Figure shows matching words (in bold font) found in both the bug report and one of the corresponding source code files that was ultimately fixed for that bug. The better an IR system can interpret the bug report and source files, the more accurately it is expected to highly rank the source files needing to be fixed. While deep

Bug ID: 80720
Summary: **Pinned console** does not remain on top
Description:
*Open two **console views**, ... **Pin** one **console.** Launch another program that produces output. Both **consoles display** the last launch. The **pinned console** should remain **pinned.***

Source code file: **ConsoleView.java**

```
public class ConsoleView extends PageBookView
    implements IConsoleView, IConsoleListener {...
      public void display(IConsole console) {
          if (fPinned && fActiveConsole != null) { return;}
      } …
      public void pin(IConsole console) {
            if (console == null) {  setPinned(false);
            } else {
               if (isPinned()) { setPinned(false); }
              display(console);
              setPinned(true);
```

Figure 6.1: An example of a bug localization [173]

semantics remain elusive, shallow matching often works quite well.

Researchers have previously evaluated a number of IR models for bug localization. Lukins et al. [83] proposed a Latent Dirichlet Allocation (LDA) approach, while Rao et al. [115] compared a range of IR techniques: Unigram, Vector Space, Latent Semantic Analysis (LSA), LDA, Cluster Based, and various combinations. Both used a relatively small number of bugs in evaluation. Ngyuen et al. proposed *BugScout* [100], which customized LDA for bug localization. Results on several large-scale public datasets showed good performance. Recently, Zhou et al. [173] proposed *BugLocator*, which combined a sophisticated TF.IDF formulation, a modeling heuristic for file length, and knowledge of previously fixed similar bugs. In a large scale evaluation of approximately 3,400 bugs over four open source projects, BugLocator showed even stronger performance. Moreover, datasets and BugLocator's executable were made available, providing an invaluable benchmark for testing and comparing alternative IR approaches to bug localization.

Despite the empirical success of prior work, we perceive a gap today between IR community practices and techniques being applied to bug localization. For example, exist-

ing IR-based bug localization treats source code as flat text lacking structure. In fact, source code's rich structure distinguishes code constructs such as comments, names of classes, methods, and variables, etc. While ignoring such code structure simplifies the system, it also sacrifices an opportunity to exploit this structural information to improve localization accuracy. While we believe modeling source code structure is novel for bug localization, we also note that the concept of modeling document structure in IR is quite old (e.g., Google in 1998 [22] and more recent BM25F [116]).

Whereas recent prior work [173] devised a heuristic to model program length, we discuss how the importance of length normalization was actually recognized in IR two decades ago [137] and is built-into today's baseline IR models. In the same vein, we discuss how the use of bug similarity data to improve localization is closely related to the established IR use of relevance feedback data [118]. Generalizing from this, we suspect that our idea for modeling code structure is only one of the many ways in which IR-based bug location could benefit from greater interaction with the IR community. Beyond our technical contributions, our approach strives to forge stronger conceptual ties between ongoing work in bug localization and proven practices from the IR community.

We introduce BLUiR (Bug Localization Using information Retrieval), an automatic bug localization tool based on the concept of structured information retrieval. Rather than build BLUiR's IR indexing and retrieval system from scratch, we instead build upon an existing, highly-tuned, open source IR toolkit [140]. While we use an off-the-shelf IR tool, we simultaneously stress the importance of using it effectively, i.e., recognizing and addressing domain-specific particulars of bug localization. In our work, we extract and model code constructs like structured documents, and we show how a seemingly trivial change to how camel case identifiers are indexed yields significantly improved localization accuracy.

We evaluate BLUiR using the same large-scale benchmark on which BugLocator was evaluated. When bug similarity data is not used, the off-the-shelf IR toolkit (unmodi-

fied) already exceeds BugLocator's accuracy. With our enhancements (e.g., structural modeling and indexing camel case identifiers as-is), accuracy is significantly improved further. Modeling additional bug similarity data provides yet a further gain. Finally, even if BugLocator is given bug similarity data and BLUiR is not, BLUiR still outperforms BugLocator on three of the four code repositories in the benchmark and matches its accuracy on the fourth.

Contributions. We present: 1) new techniques for increasing localization accuracy, particularly modeling of source code structure; 2) new state-of-the-art accuracy for bug localization on a public community benchmark, built on a proven, open source IR toolkit anyone can use; and 3) thorough grounding of IR-based bug localization research in fundamental IR theoretical and empirical knowledge and practice.

6.2 Presence of Source Code Terms in Bug Reports: An Empirical Study

The success of IR-based bug localization is dependent on effectively matching the bug report to the source files needing to be fixed. As discussed in Section 2, even preprocessing issues can significantly impact IR accuracy. The classic IR challenge lies in effectively recognizing important terms in the query and document, and assigning each a greater weight for matching. With regard to text length, long queries (e.g., when using the bug report's `description` field) can obscure key search terms [73]. Document length also merits special attention [137]. Both topics are further discussed in Section 6.8.3.

Another classic IR approach distinguishes and separately models different fields when text is structured [22, 116]. For example, while searching documents, Google considers page title, different anchor texts, and the body separately [22]. We investigate this structured approach to IR-based bug localization. With queries, a bug report contains separate `summary` and `description` fields; whereas the summary provides essential keywords,

the description is more verbose with additional terms. As discussed in the next section, source code files are even more structured. We perform preliminary analysis here to assess the degree to which source code terms appear in bug reports, potentially providing an opportunity for better IR.

We distinguish six types of terms. Query terms come from different bug report fields: the concise `summary` and verbose `description`). Parsing source code structure also lets us distinguish four different document fields: `class`, `method`, `variable`, and `comments`. These fields are extracted by constructing and traversing the abstract syntax tree (AST) of the subject program (Section 6.3.1). For each bug report, we separately search for terms from each document field in source files that were fixed for the corresponding bug. We collect two separate sets of statistics: matching terms "as is" in their original form vs. splitting identifier names based on the camel case heuristics and searching for each token.

To illustrate, for the example given in Figure 6.1, first we search for `ConsoleView`, and then the separate terms `console` and `view`, in both the bug summary and bug description. For each search, we exclude those tokens that either are stop words or have fewer than three characters. For example, if a variable name is `isBalancedTree`, we do not search for "is".

Table 6.1 provides empirical evidence that terms from source files to fix are present in the corresponding bug reports. Each entry represents the number of bug reports in which different term types (class, method, variable, or comment) were found. For each bug report section (`summary` vs. `description`), we count the number of bug reports containing an exact match or token match for at least one of the files to be fixed. For example, the first two numbers in the "class" row of Table 6.1 represent that in 27 bug reports in `AspectJ`, at least one of the class names of the fixed files was present as-is in the bug summary, whereas in 101 bug reports at least one of the class name's split tokens was present. We intentionally restrict the analysis here to `AspectJ`, reserving the other three source code repositories for later blind evaluation to maximize the generality of our findings.

Table 6.1: Presence of Different Term Types in Bug Reports for AspectJ

Term Type	Summary		Description	
	Exact match	Token match	Exact mach	Token match
Class	27 (9.44%)	101 (35.31%)	148 (51.74%)	244 (85.31%)
Method	43 (15.03%)	205 (71.67%)	187 (65.38%)	277 (96.85%)
Variable	107 (37.41%)	125 (43.70%)	230 (80.42%)	252 (88.11%)
Comments	N/A	235 (82.16%)	N/A	278 (97.20%)

From the table, we see that although the summary contains only 3% of the total terms in the bug report, at least one of the class, method, variable, and comment terms was found in 35%, 72%, 43%, and 82% of the bug summaries, respectively. Similarly, although a class name is typically a combination of 2-4 terms per source code file, class names' terms are present in more than 35% of the bug summaries and 85% of the bug descriptions. Furthermore, the exact class name is present in more than 50% of the bug descriptions. We can observe a similar phenomenon for method names as well.

While the bug description has many more matches than the bug summary, the more verbose `description` likely matches many irrelevant terms as well. Similarly, Table 6.1 only shows matches from the source files needing to be fixed. The bug reports also include terms matching many other source files not needing to be fixed. Consequently, this table provides suggestive rather than conclusive evidence for our approach; evaluation later in the chapter will demonstrate the empirical effectiveness of modeling this information.

6.3 Approach

In the previous section, we showed that program structure, i.e., important program constructs such as class names and method names are present in many bug reports and thus might be effectively used to improve bug localization. This section describes our structured IR-based approach for localizing bugs.

6.3.1 BLUiR Architecture

Figure 6.2 shows the overall architecture of BLUiR. First BLUiR takes as input the source code files in which we would like to localize the bugs. Next, it builds the abstract syntax tree (AST) of each source code file using Eclipse Java Development Tools (JDT), and traverses the AST to extract different program constructs such as class names, method names, variable names, and comments. Then BLUiR tokenizes all the identifier names and comment words, as described in Section 6.3.2. This information for each source file is then stored as a structured XML document.

Reducing bug localization to a standard IR task enables us to exploit a wealth of prior theoretical and empirical IR methodology, providing a robust foundation for tackling bug localization. We adopt the Indri toolkit [140] for efficient indexing and developing our retrieval model. After XML documents are created as described above, they are handed off to Indri for stopword removal, stemming, and indexing. We used the default stopword list provided with Indri.

Each bug report is similarly tokenized, then handed off to Indri for stopping, stemming, and retrieval (Section 6.3.3).

6.3.2 Source Code Parsing & Term Indexing

In comparison to prior approaches, we make two improvements in our preprocessing. First, prior work has indexed all source code terms except English stopwords and programming language keywords. However, some keywords like `String` or `Class` are used in identifier names and may be found in bug reports. For example, in `AspectJ`, many identifiers use Java language keywords, e.g., `if`, `else`, etc via camel case. Therefore, instead of pruning all language keywords, we instead build the Abstract Syntax Tree (AST) of each source file and extract all identifier names (class name, method name, variable name etc.). In this way, we exclude language keywords without losing their presence in identifiers.

Secondly, identifiers are typically split into tokens for indexing to improve recall.

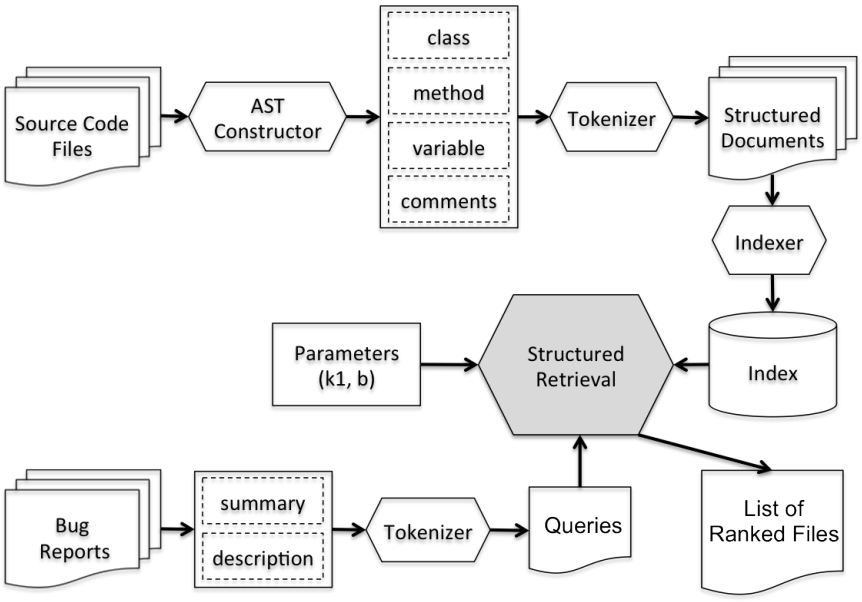

Figure 6.2: BLUiR Architecture

Dit et al. [31] compared simple camel case splitting to the more sophisticated *Samurai* [37] system and found that both performed comparably in concept location. This suggests that either could be used interchangeably, and so we adopt camel case splitting for its simplicity. However, since our analysis in Table 6.1 reveals that full identifiers are often present in bug reports in the form of execution traces of exceptions, test cases or code snippets, we index full identifiers as well as split tokens. Although it is a very simple extension, we will see that it yields significant improvement.

6.3.3 Retrieval Model

TF.IDF is not a well-defined model, and different TF.IDF variants can achieve vastly different empirical performance in practice. We adopt Indri's built-in TF.IDF formulation (from its parent project *Lemur*), based upon the well-established BM25 (Okapi) model [117]. This TF.IDF model has been rigorously evaluated over a decade of widespread use in IR. We elaborate below.

Assume that a document and a query are represented by a weighted term frequency vectors $\vec{d}$ and $\vec{q}$ respectively of length n (the total number of terms or the size of vocabulary).

$$\vec{d} = (x_1, x_2, ..., x_n) \tag{6.1}$$

$$\vec{q} = (y_1, y_2, ..., y_n) \tag{6.2}$$

Each element of x_i of $\vec{d}$ represents the frequency (count) of term t_i in document d (similarly, y_i in query $\vec{q}$).

Generally, in a vector space model, query and document terms are weighted by a heuristic TF.IDF weighting formula instead of only their raw frequencies. Inverse document frequency (IDF) diminishes the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely. Weighted vectors for $\vec{d}$ and $\vec{q}$ are thus:

$$\vec{d_w} = (tf_d(x_1)idf(t_1), tf_d(x_2)idf(t_2), ..., tf_d(x_n)idf(t_n)) \tag{6.3}$$

$$\vec{q_w} = (tf_q(y_1)idf(t_1), tf_q(y_2)idf(t_2), ..., tf_q(y_n)idf(t_n)) \tag{6.4}$$

Given a collection C of source files, the simplest, classic IDF formulation for term t is given by $idf(t_i) = log\frac{N}{n_t}$, where N is the total number of documents in C and n_t is the number of documents with term t. In the simplest TF-IDF model, we would simply multiply this value by the term's frequency in document d to compute the TF-IDF score for (t, d), then sum over all terms in the query to arrive at the d's TF-IDF score. As mentioned above, however, actual TF-IDF models used in practice differ greatly from this for improved accuracy [117, 136]. We adopt Indri's TF.IDF model [164], which is summarized below.

To begin with, the IDF value is smoothed as follows to avoid division by zero, which would otherwise occur whenever a particular term appears in all documents: $idf(t_i) = log\frac{N+1}{n_t+0.5}$.

The *document*'s tf function is defined by Okapi:

$$tf_d(x) = \frac{k_1 x}{x + k_1(1 - b + b\frac{l_d}{l_C})} \tag{6.5}$$

where k_1 is a tuning parameter (≥ 0) that calibrates document term frequency scaling. The *term frequency* value quickly saturates for a small value of k_1, whereas, a large value corresponds to using raw term frequency. b is another tuning parameter between 0 and 1, which is the document scaling factor. Recall that BugLocator introduces a heuristic for modeling document length, whereas this is already built into IR models today (Section 6.1). Here, when the value of b is 1, the term weight is fully scaled by the document length. For a zero value of b, no length normalization is applied. l_d and l_C represent the document length and average document length for the collection respectively.

The *query*'s TF function tf_q is defined similar to tf_d though $b = 0$ is fixed since the query is fixed across documents being compared, and thus normalization of the query length is unnecessary:

$$tf_q(y) = \frac{k_3 y}{x + k_3} \tag{6.6}$$

In Equation, 6.6, the value of k_3 is fixed to 1000 to obtain almost the raw query term frequency because in a query the probability of having the same term many times is rare. Now the similarity score of document $\vec{d}$ against query $\vec{q}$ is given by Equation 6.7.

$$s(\vec{d}, \vec{q}) = \sum_{i=1}^{n} tf_d(x_i) tf_q(y_i) idf(t_i)^2 \tag{6.7}$$

6.3.4 Incorporating Structural Information

The TF.IDF model presented in Equation 6.7 does not consider source code structure (program constructs)—i.e., each term in a source code file is considered to have the same relevance with respect to the given query. Therefore, important information like class names and method names often gets lost in the relatively large number of variable names and comments terms due to the term weighting function (Equation 6.5). For example, if a source code file with class name "A" also contains 10 other variable names having the term "A", then the class name "A" does not add much weight. Thus, if there is a bug report related to class "A", it will rank another file higher if the file has the term "A" more than 11 times even in the local variable names or comments. Our proposed model distinguishes different

code constructs to overcome this problem.

As we described in Section 6.2, we distinguish two alternative query representations coming from different fields of the bug report (the *summary* and the more verbose *description*). Parsing source code structure also lets us distinguish four different document fields:[32] *class, method, variable, comments*. To exploit all of these different types of query and document representations, we perform a separate search for each of the eight (query representation, document field) combinations and then sum document scores across all eight searches.

$$s'(\vec{d}, \vec{q}) = \sum_{r \in Q} \sum_{f \in D} s(d_f, q_r) \tag{6.8}$$

where r is a particular query representation and f is a particular document field. The benefit of this model is that terms appearing in multiple document fields are implicitly assigned greater weight, since the contribution from each term is summed over all fields in which it appears. While our method of integrating structural information is quite simple, more sophisticated methods for integrating structural information could be explored in future work, e.g., doing a weighted combination rather than a simple sum, or better yet, weighting term frequencies rather than document fields to better control for term frequency saturation [116].

6.4 Evaluation Setup

6.4.1 Data Set

We have used the same dataset that Zhou et al. [173] used to evaluate BugLocator. This dataset contains 3,379 bug reports in total from four popular open source projects–Eclipse, AspectJ, SWT, and ZXing along with the information about which files were fixed for those bugs. Table 8.1 describes the dataset in more detail. Since we would like to compare BLUiR with BugLocator, using the same dataset allows us to get comparable results. Among the

[32] In IR, each term's type in a structured document called field

Table 6.2: Details of Benchmark

Project	Description	Period	#Bugs	#Files
SWT 3.1	Widget toolkit for Java	10/04-04/10	98	484
Eclipse 3.1	Popular IDE for Java	10/04 03/11	3075	12863
AspectJ	Aspect-oriented extension to Java	07/02-10/06	286	6485
ZXing	Barcode image processing library for Android	03/10-09/10	20	391

four subject systems in the dataset, we always use AspectJ for learning (to tune the parameters) so that we do not overfit our retrieval model. We chose the AspectJ system as a training dataset because it has 298 bugs, which is neither too large nor too small, compared to the number bugs in other subject systems. We have also compared our results with a similar version of the dataset (for AspectJ and Eclipse) that was used in evaluating BugScout (Table 6.7).

6.4.2 Evaluation Metrics

Since an IR system's value is in direct proportion to how well it serves its users, the design and selection of appropriate evaluation metrics has been a topic of considerable study in IR. We should select a sufficient yet minimal set of metrics to ensure that what we measure provides an appropriate and comprehensible yardstick for assessing the most pertinent aspects of performance. As Lord Kelvin remarked, "if you cannot measure it, you cannot improve it." We err on the side of excess and comparative evaluation, including all five metrics considered by Zhou et al. [173]; other systems we compare to use a subset of these metrics. All metrics are based on gain rather than loss (larger values indicate better performance).

Recall at Top N: This metric reports the number of bugs with at least one buggy source file found in the top N (= 1, 5, 10) ranked results (once the first buggy file is located, it may become easier for the developer to find the rest). Since we are only considering the top few ranks, and only require finding one of the buggy files per bug, this metric emphasizes early precision over total recall.

Mean Reciprocal Rank (MRR): Like "Recall at Top N", MRR emphasizes early

precision over recall. The reciprocal rank for a query is the inverse rank of the first relevant document found. MRR is the reciprocal rank averaged over all queries:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \tag{6.9}$$

Mean Average Precision (MAP): MAP is by far the most commonly used, traditional IR metric. MAP emphasizes recall over precision, and thus is favored in scenarios in which users will go deep in a ranked list to find many relevant results. The Average Precision of a single query is computed as:

$$AP = \sum_{k=1}^{M} \frac{P(k) \times pos(k)}{number\ of\ positive\ instances} \tag{6.10}$$

where k is the rank, M is the number of retrieved source files, and $pos(k)$ is a binary indicator of whether or not the item at rank is a buggy file. $P(k)$ is the precision at the given cut-off rank k. The MAP for a set of queries is simply the mean of the average precision values for all queries.

Note that all metrics above compute an arithmetic mean over the query set to measure average performance. This may not be appropriate if developer satisfaction is driven by worst-case performance rather than average performance, in which case a geometric mean may be more appropriate. Figure 6.3 presents a per-query analysis of results, inspecting the performance of each query instead of only the average.

6.4.3 System Tuning

As we described in Section 6.4.1, we use AspectJ as a training dataset to choose the stemmer and to tune two parameters of our model: the term weight scaling parameter k_1 and the document normalization parameter b. Table 6.3 compares the system performance with no stemmer vs. using two popular stemmers: Krovetz and Porter. For this experiment (only), we use approximately raw TF.IDF with $k_1 = 1000$, and $b = 0$. We observe no significant difference among the three methods. In prior work, Hill et al. [48] also observed that

Table 6.3: Effect of Different Stemmers and Parameters on AspectJ

Term Weighting	Stemmer	Top 1	Top 5	Top 10	MAP	MRR
	none	29	93	134	0.12	0.22
$k_1 = 1000, b = 0$	Krovetz	29	97	134	0.12	0.22
	Porter	27	99	135	0.12	0.21
$k_1 = 1.2, b = 0.75$	Krovetz	77	130	162	0.20	0.37
$k_1 = 1.0, b = 0.3$	Krovetz	79	131	168	0.20	0.37

no single stemmer is better for all kinds of queries. While we choose Krovetz somewhat arbitrarily, as the more conservative of the two stemming algorithms, closer analysis here appears to be warranted to provide a fuller explanation.

Table 6.3 shows results of tuning k_1 and b. These experiments exclude modeling of source code structure. Traditional wisdom is to set $k_1 = 1.2$ and $b = 0.75$. However, since bug localization is different from traditional text retrieval, we did a linear sweep of all values between 0 and 2 for k_1 and between 0 and 1 for b (with step-size 0.1), selecting $k_1 = 1.0$ and $b = 0.3$ as optimal.

6.5 Results

This section presents the evaluation results of BLUiR while performing bug localization on the four subject systems described in Table 6.2. We mainly answer five research questions that show the effectiveness of different improvements that we made in developing BLUiR, and compare the results of BLUiR with other information retrieval models and tools.

RQ1: Does indexing the exact identifier names improve bug localization? In Section 6.2, we observed that in many bug reports of AspectJ, different kinds of source code entity names (e.g., class name, method name) are present exactly as-is. In this research question, we investigate the effectiveness of adding full identifier names as well as tokenized identifiers to the index. Our experiments in this section exclude source code structure. Results are reported for all four subject systems, first with only the tokenized

Table 6.4: Effect of Indexing Full Idenifier Names

System	Indexed	Top 1	Top 5	Top 10	MAP	MRR
SWT	Tokens	29	72	82	0.41	0.48
	Both	37	71	84	0.47	0.54
Eclipse	Tokens	529	1121	1415	0.20	0.27
	Both	746	1378	1647	0.26	0.34
AspectJ	Tokens	79	131	168	0.20	0.37
	Both	87	147	175	0.22	0.41
ZXing	Tokens	8	11	12	0.35	0.48
	Both	7	11	12	0.35	0.45

identifier names and then with the combination of tokenized and full identifier names.

Table 6.4 presents the result of indexing both. The evaluation results show that the addition of full identifier names improves the accuracy for three of four subject systems. In Eclipse, using exact identifier names, BLUiR localized 217 (7.05%) more bugs in the Top 1 file, whereas, the increases are 8.16% and 2.79% for SWT and AspectJ respectively. The consistently higher MAP and MRR for the first three subject systems show the overall improvements of the ranking due to adding exact identifiers. In ZXing, the Top 1 and MRR metrics were a little bit lower than the traditional one, while other metrics were exactly the same. However, it is difficult to derive any useful conclusions from ZXing because the dataset has only 20 bugs for this subject system.

RQ2: Does modeling source code structure help improve accuracy? In Section 6.2, we argued that source code structure, i.e., distinguishing different code constructs could be effectively used to find more important terms in both source code and bug reports and thus improve the overall bug localization accuracy. In this research question, we investigate whether this improves the accuracy of bug localization and, if so, how much. To this end, we ran BLUiR on all the subject systems to localize bugs with and without modeling source code structure.

Table 6.5 results show that in most cases, BLUiR performed better in terms of all the metrics when it considers different program constructs. More specifically, structured re-

Table 6.5: Effect of Modeling Source Code Structure

System	Structure	Top 1	Top 5	Top 10	MAP	MRR	ET/$Q_i(s)$
SWT	N	37	71	84	0.47	0.54	0.05
	Y	54	75	85	0.56	0.65	0.21
Eclipse	N	746	1378	1647	0.26	0.34	0.44
	Y	952	1636	1933	0.32	0.42	5.45
AspectJ	N	87	147	175	0.20	0.37	0.57
	Y	92	146	173	0.24	0.41	4.22
ZXing	N	7	11	12	0.35	0.45	0.08
	Y	8	13	14	0.38	0.49	0.25

trieval is more effective for Top 1. Using structured retrieval, BLUiR localized 17 (17.35%), 206 (6.70%), 5 (1.74%), and 1 (5%) more bugs in SWT, Eclipse, AspectJ, and ZXing respectively within the Top 1 file. In AspectJ, for Top 5 and Top 10, BLUiR localized a few less bugs when using structured retrieval. However, the high MAP and MRR shows that the overall ranking is much better when BLUiR uses structured retrieval. Further qualitative analysis is presented in Section 6.6.

Runtime Overhead: Since the structured information retrieval involves more computation than the normal text retrieval, it should add some runtime overhead. In order to investigate this issue, we computed the average execution time per query (ET/Q_i) of BLUiR both for traditional retrieval and structured retrieval. From Table 6.5, we see that structured retrieval is more costly than the normal analysis, depending on the size of *document collection*. However, since all the execution times remain in a range of a few seconds, the added cost should be almost negligible from the developers perspective. In addition, structured information retrieval may save quite a bit of developers' time since it has higher accuracy.

RQ3: Does BLUiR outperform other bug localization tools and models? While evaluating BugLocator, Zhou et al. [173] compared their model with other prior work, and showed that BugLocator consistently performed best. We therefore compare BLUiR with BugLocator, which is, to the best of our knowledge, the most accurate tool at present.

Table 6.6 shows results of BLUiR and BugLocator for the given dataset, using and

without using similar bug report data. BugLocator results are copied verbatim from [173]. It should be noted that we use the same datasets used to evaluate BugLocator. In this section, we restrict our discussion to results without using bug similarity data.

Comparing the various metrics for each system, we can see that, for ZXing, the results produced by both tools are almost the same. As we explained earlier, it is very difficult to derive any useful conclusions from ZXing because its bug dataset has only 20 bugs. However, looking into the results of other systems, which have more bug reports (98 for SWT, 3075 for Eclipse, and 286 for AspectJ), we can clearly see the that BLUiR outperformed BugLocator by a great margin. BLUiR localized 23 (23.47%) more bugs in SWT, 203 (6.60%) more bugs in Eclipse, and 27 (9.44%) more bugs in AspectJ ranked within the Top 1 file. The same trend is observed for other metrics as well. The consistently higher MAP and MRR for BLUiR also suggest that the overall ranking of the buggy files produced by BLUiR is better than that of BugLocator.

Since the higher number of bugs located in Top 1, 5, and 10 files retrieved by BLUiR than that of BugLocator does not necessarily mean that BLUiR performed well for all queries, now we investigate the number of queries for which BLUiR actually performed better than BugLocator . **Figure 6.3** shows per-query performance of BLUiR compared to BugLocator on SWT, where the X axis represents the query number and the Y axis represents the difference between the best rank of the buggy files by BLUiR and that of BugLocator. The negative value represents the query where BugLocator performs better than BLUiR. We can see that for 47 queries BLUiR performed better, for 14 queries BugLocator performed better, and for 35 queries both tools perform exactly the same. This results suggest that BLUiR performed better than BugLocator for most of the queries. Interestingly, we also observe that for 12 bug reports BLUiR improved the rank of buggy files by more than 10 positions, whereas there were only two bug reports where BugLocator improved the rank by more than 10 positions (64 and 85 positions). As a result, BLUiR places more buggy files within the Top 1, 5, and 10 files in the rank list than BugLocator.

Table 6.6: BLUiR vs BugLocator

System	Method	SB	Top 1	Top 5	Top 10	MAP	MRR
SWT	BugLocator	N	31	64	76	0.40	0.47
	BLUiR	N	54	75	85	0.56	0.65
	BugLocator	Y	39	66	80	0.45	0.53
	BLUiR	Y	55	75	86	0.58	0.66
Eclipse	BugLocator	N	749	1419	1719	0.26	0.35
	BLUiR	N	952	1636	1933	0.32	0.42
	BugLocator	Y	896	1653	1925	0.30	0.41
	BLUiR	Y	1013	1729	2010	0.33	0.44
AspectJ	BugLocator	N	65	117	159	0.17	0.33
	BLUiR	N	92	146	173	0.24	0.41
	BugLocator	Y	88	146	170	0.22	0.41
	BLUiR	Y	97	150	176	0.25	0.43
ZXing	BugLocator	N	8	11	14	0.41	0.48
	BLUiR	N	8	13	14	0.38	0.49
	BugLocator	Y	8	12	14	0.44	0.50
	BLUiR	Y	8	13	14	0.39	0.49

This analysis required access to per-query results from BugLocator, made possible by its executable being publicly available. Unfortunately, it crashed when run on the other three collections, and we could not reach the authors for assistance. This analysis is therefore limited to SWT only.

We also compare our results to *BugScout* [100] and BugLocator in **Table 6.7**. To evaluate BugScout, Nguyen et al. used AspectJ and Eclipse as their subject systems. Since our datasets are not exactly the same as theirs, we present the differences between the two datasets in terms of number of bugs in Table 6.7. We have also guessed the recall at Top 1, Top 5, and Top 10 of BugScout results from a figure in their paper, which may slightly differ from their actual value. The results show that BLUiR outperforms BugScout consistently.

Recently, Sisman and Kak [138] incorporated version histories into IR-based bug localization. They proposed two models, namely the *Modification History based Prior* and the *Defect History based Prior* models, to estimate a prior probability for each file in a

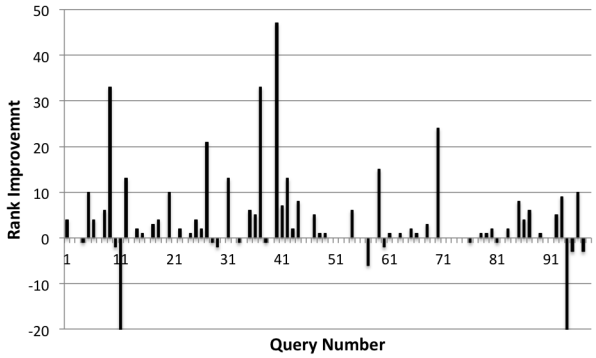

Figure 6.3: Query wise comparison of BugLocator and BLUiR for SWT

Table 6.7: Comparison of BugScout and BugLocator with BLUiR

System	Description	BugScout	BugLocator	BLUiR
AspectJ	Number of Bug Reports	271	286	286
	Top 1	11%	23%	32%
	Top 5	26%	41%	51%
	Top 10	35%	56%	60%
Eclipse	Number of Bug Reports	4,136	3,075	3,075
	Top 1	14%	24%	31%
	Top 5	24%	46%	53%
	Top 10	31%	56%	63%

project having bugs. Then used these priors to rank documents in addition to different models. Based on a case study on AspectJ, they showed that version histories improved the MAP by as much as 30%. However, without considering any version history information, BLUiR (MAP: 0.2396) performed better than their best results (MAP: 0.2258).

RQ4: Does our approach compensate for the lack of similar bug information? Although the performance of BugLocator improved a lot after using similar bug fixes information, one of our main objectives is improving bug localization without using the similar bug information, since most real world projects do not explicitly have that information. In addition, reconstructing the similar bug fix information is not a trivial task. Therefore, we investigate how BLUiR performs in locating bugs compared to BugLocator when BugLocator uses similar bug information. The comparative results presented in Figure 6.6 show that in most cases, the MAP and MRR of BLUiR are higher than that of BugLocator, which indicates that BLUiR overall performed better even when BugLocator considered similar bug information. For example, BLUiR localized 17 more bugs in SWT, 56 more bugs in Eclipse, and 4 more bugs in AspectJ than BugLocator within the Top 1 file.

We were also curious to see if we could capture those bugs that were localized by BugLocator using similar bug information. To this end, we ran BugLocator on SWT using ($\alpha = 0.2$) and not using ($\alpha = 0$) similar bug information. We found 10 such bugs in total, that have been placed within top 1, 5, or 10 files by BugLocator after using similar bug information. We found that BLUiR could localize all of these bugs without using similarity information.

RQ5: Does similar bug fix information further improve our model? We implemented the same technique for incorporating similar bug information to BLUiR that Zhou et al. [173] did in developing BugLocator. By comparing the results of BLUiR in Table 6.6, we can see that the similar bug information further improved our results. For example, it helps BLUiR localize 61, 93, and 77 more bugs ranked in the top 1, 5, and 10 files respectively for Eclipse. The result is also improved 1 bug localization in SWT and 5 in

AspectJ within top 1 file. However, the overall improvement due to using similar bug information was not as large as that of BugLocator. Therefore, here we can conclude that BLUiR can compensate for the lack of similar bug information partially because it already localized many bugs without using similar bug fix information, which were only localized by BugLocator using similar bug information. BLUiR can also make use of the similar bug fix information to improve the model further, if it is available.

Other Results. We briefly report preliminary experiments with pseudo-relevance feedback (PRF, Section 6.8.2) using Indri. The primary advantage of using PRF is that we do not need to know any prior information (e.g., similar bugs) about relevant documents while running a query. In PRF mode, Indri basically performs the general retrieval first, and then augments the original query by taking the m most frequent words from the top r documents. There is also a tuning parameter α for weighting the original query and augmented terms. Finally, the augmented query is run again to get the final rank list. We experimented with different values of m, n, and α, but did not observe improved accuracy. In future work, we would like to explore this idea further.

6.6 Qualitative Analysis

The previous section presented quantitative results showing BLUiR's improvement on average over BugLocator. In this section, we dig into several queries in detail to better understand why BLUiR performs better in most cases. Consider SWT bug report #87676:

Summary: *Double-click only works on a tree's column0*

Description: *Using the log view as an example, double-clicking on column0 brings up the event dialog as it should. double-clicking on column1, column2 results in no notification to our double-click listener.*

Only `org.eclipse.swt.widgets.Tree.java` was fixed for this bug. By reading the bug report and seeing the file name of the fixed file, one might think at a glance

that this file can be identified easily. However, identifying a buggy file in a real world project is not that easy, especially where there are many other such similar files. For example, in SWT there are at least 10 other files (TreeColumn.java, TreeEvent.java, TreeListener.java, TreeAdapter.java, TreeItem.java etc.) that deal with `tree` and have this word in their file names. Thus, for a developer who did not originally implement the functionality of tree might think TreeColumn and TreeListener are more important because the bug report contains the words `column` and `listener`. Furthermore, the bug report has some other words such as `double click`, `event`, `dialog`, which are contained many times in more than 30 other files such as Text.java, Widget.java, Button.java, and so on. Therefore, finding the desired files from the IR perspective is also very challenging. Relying on only the length of the files is certainly not the solution of this problem.

As a result BugLocator placed the file, Tree.java, at the 50th position in the rank list. Fortunately, BLUiR first performs all the field retrievals using both the bug summary and the bug description, and then aggregates all the scores to finally rank all the source code files. This results in the summary words (e.g. `tree`) being used more advantageously. Furthermore, documents that have the search words (e.g. `column, double click`) spread over more fields produce better results than documents having the search words found in one field. In this way, BLUiR focuses on more important words in the documents. As a result, BLUiR placed Tree.java at the 3rd position in the rank list. In this way, BLUiR improved the rank of buggy files by more than 10 positions for 12 bug reports (e.g., bugs #78856, #79419, #83262, and so on).

6.7 Threats to Validity

This section discusses the validity and generalizability of our findings. In particular, we discuss Construct Validity, Internal Validity, and External Validity.

Construct Validity. We have used two primary artifacts of a software repository: source code and bug reports, which are generally well understood. Our evaluation uses

the same benchmark dataset of bug reports and source code shared by Zhou at al. [173], enabling fair comparison and reproducible findings. Metrics used for evaluation match those of Zhou at al. and other prior work, are standard in IR, and are straightforward to compute. Therefore, we argue for a strong construct validity.

Internal Validity. We utilize program constructs to rank source code documents with respect to a given bug. Since all the subject systems we have used are written in Java, these are mainly object-oriented (OO) constructs such as class names, method names, and so on. In this sense, our approach is language dependent. However, in the next chapter, we show that our approach is easily adaptable to the procedural language, such as C, as well.

Since we are matching terms between bug reports and source code, we assume meaningful identifier names and inclusion of comments, consistent with programming best practices. That said, poorly written source code would make bug localization more difficult (for both IR or non-IR approaches). Similarly, we also depend upon the quality of the bug report, and poorly written reports would likely also hurt IR and non-IR methods. Our structural modeling approach matching source code terms in bug reports likely benefits significantly from the bug reports having been written by developers knowledgeable of the underlying source code. Bug reports written by end-users would likely show a far less pronounced effect.

We have used the same dataset as of Zhou et al. [173]. While the possibility exists of errors in their data, this seems quite low since they have manually validated the dataset. Also, three of our four subject systems represent system-specific projects. As Hyrum et al. [159] noted, system-domain software may have its own set of development biases. Therefore, we may not capture some unique concerns, which are only present in the software development targeted toward other domains.

External Validity. We have used only four subject systems in our experiment and all of them are open source projects. Although, they are very popular projects, our findings may not be generalizable to other open source projects or industrial projects. However, to

maximize generalizability of findings and minimize risk of over-fitting, we developed and tuned BLUiR on only one subject system (AspectJ), reserving the remaining three systems for a final blind evaluation. This risk of insufficient generalization could be mitigated by expanding the benchmark to include more subject systems (both open source and industrial). This will be explored in our future work.

6.8 Related Work

6.8.1 Automatic Bug Localization

Automatic bug localization or automatic debugging has been an active research area for over two decades [132, 133]. Existing techniques can be broadly categorized into two categories: dynamic [1] and static [52]. Generally, dynamic fault localization techniques can localize bugs very precisely (such as at statement level). However, they require a test case suite and need to execute the program for gathering passing and failing execution traces. Furthermore, the approaches are computationally expensive. Spectrum based bug localization [1, 61, 76], dynamic slicing [172], and delta debugging [163] are some of the well known techniques in this category.

Static approaches, on the other hand, do not require any program test cases or execution traces. In most cases, they need only program source code and bug reports. They are also computationally efficient. The static approaches can be also divided into two categories: i) program analysis based approaches ii) IR-based approaches. FindBug [52] is a popular bug localization tool based on static program analysis that can detect bugs by identifying buggy patterns frequently happened in practice. Therefore, FindBug does not even need a bug report. However, it cannot detect semantic bugs.

6.8.2 Information Retrieval (IR)

While historical arguments debated which of three traditionally-dominant IR paradigms was best (TF.IDF [130], the "probabilistic approach" known as BM25 (Okapi) [117], or more recent language modeling [108], all three approaches have been shown theoretically to utilize the same underlying textual features, and empirically to perform comparably when well-tuned [38]. Consequently, while one formalism or another might make it easier to integrate useful additional features, use of one formalism or another is not particularly important when it simply comes to baseline IR performance.

In contrast with shallow "bag-of-words" models, research has also explored deeper methods matching "concepts" (often poorly defined). While latent semantic indexing (LSI) induces latent concepts, it is rarely used in practice today due to errors in induced concepts introducing more harm than good. While a probabilistic variant of LSI has been devised [50], its probability model was found to be deficient. This led to now ubiquitous latent dirichlet allocation (LDA) modeling [20]. While many studies have shown LDA can usefully infer latent topics underlying a document collection, LDA is both computationally expensive and operates without reference to the input query. It has been shown that far simpler IR models based on pseudo-relevance feedback (PRF) can efficiently induce better topics on the fly for each query, tailored to the query vs. query-independent LDA topics [161]. Consequently, LDA models appear less useful for IR than simpler models until this fundamental problem can be meaningfully addressed.

One issue considered in this chapter was how to best utilize multiple representations of the same bug report (i.e., its `summary` and `description`). While the summary is very succinct and likely provides the most important keywords, it may lack other terms useful for matching (suggesting high precision but possibly low recall). In contrast, the more verbose description may contain many other useful terms to match, it likely contains a variety of distracting terms as well. This is a very well-known problem in traditional IR [73]. For over two decades, data from the Text REtrieval Conference (TREC) has provided queries

at three levels of verbosity, with researchers devising various methods to maximally exploit these different representations. For example, simply concatenating the two representations (e.g., for our case they are bug summary and description) together provides an easy way to emphasize keywords while also including more verbose terms as well. In this work, our method of performing separate summary and description searches and summing results is roughly equivalent to such concatenation. Future work could explore a wide variety of more sophisticated IR methods for exploiting these alternative query representations with varying verbosity.

6.8.3 IR-based Bug Localization

The value of document length normalization was recognized in IR nearly two decades ago [137]. Empirical data compared the length of documents predicted relevant by TF.IDF vs. the length of actual relevant documents, showing that traditional IR models are actually biased against longer documents. An empirical correction for this bias was developed, it was realized that this correction was already built-into BM25, and it has been further shown that IR's language modeling paradigm performs implicit length normalization as well.

Several prior studies have investigated use of bug similarity data in order to improve localization accuracy [28, 173]. This idea can be seen as a close cousin to long-established methods for incorporating relevance feedback (RF) data in IR [118]. While RF exploits the fact that knowing one or more documents relevant to the current query makes it much easier to find other relevant documents, this knowledge is often seldom available in practice. A "trinity" of related variants has been theoretically and empirically established, showing that similar queries should retrieve similar documents (and vice-versa), and that similar documents should receive similar relevance scores for the same query (*score regularization*) [30]. In fact, the idea that the same documents should be relevant for similar queries provides the foundation for search community question and answer forums today [57]. Consequently, while use of bug similarity data for localization represents a very valuable adap-

tation of RF methods from traditional IR, there is a wide spectrum of similar techniques and existing methodology that might be further explored as well (e.g., the aforementioned PRF, which infers relevant documents for feedback rather requiring the user to supply them explicitly).

Concept location or feature location represents another task closely related to bug localization. Generally, concept location or feature location aims to identify the relevant parts of a software system that implement a specific concept or functionality. Thus, it is one of the most common activities in program comprehension. Researchers have used a variety of information retrieval techniques in feature location and concept location as well. Marcus et al. [91] used LSI to find modules related to a given feature in form of a user query. Poshyvanyk et al. [112] used LSI first to rank source code elements based on a given feature or bug reports, and then used a Formal Concept Analysis to cluster the results. In another work, Poshyvanyk et al. [110] formulated the feature location problem as a decision-making problem in the presence of uncertainty. The decision is taken based on the opinions from two experts. The first expert is LSI, which enables users to search static documents relevant to a feature. The second expert is the Scenario Based Probabilistic ranking, which helps user rank a list of entities, given a feature of interest, by analyzing dynamic traces from the execution of different scenarios. Gay et al. [40] incorporated RF in IR-based concept location. Although bug reports were used as a concept/feature in some of these studies, they were few in number.

6.9 Summary

Locating bugs is important, difficult, and expensive, particularly for large-scale software projects. To address this, natural language information retrieval (IR) techniques are increasingly being used to suggest potential faulty source files given bug reports. While these techniques are very scalable, in practice their effectiveness remains low in accurately localizing bugs to a small number of files.

Our key insight is that structured information retrieval based on code constructs, such as class and method names, enables more accurate bug localization. We present BLUiR, which embodies this insight, builds on an open source IR toolkit [140], requires only the source code and bug reports, and takes advantage of bug similarity data if available. We evaluate BLUiR on four open source projects with approximately 3,400 bugs. When bug similarity data is not used, the off-the-shelf IR tookit (unmodified) already exceeds state-of-the-art tool, BugLocator's accuracy. With our enhancements (e.g., structural modeling and camel case indexing), accuracy is significantly improved further. Modeling additional bug similarity data provides yet a further gain. Finally, even if BugLocator is given bug similarity data and BLUiR is not, BLUiR still outperforms BugLocator on three of the four code repositories in the benchmark.

Beyond our technical contributions, our presentation also strives to forge stronger conceptual ties between ongoing work in bug localization and proven practices from the IR community, via a thorough discussion of IR-based bug localization research in relation to fundamental IR theoretical and empirical knowledge and practice.

Chapter 7

On the Effectiveness of Information Retrieval Based Bug Localization for C Programs

This chapter is based on our paper, "On the Effectiveness of Information Retrieval Based Bug Localization for C Programs", published in Proceedings of the International Conference on Software Maintenance and Evolution [125].[33]

7.1 Context

In the previous chapter, we discussed that researchers have already evaluated many IR models for bug localization. We have also introduced a new IR-based technique, BLUiR, which takes into account program structure, distinguishing between different kinds of terms in source code based on program constructs [127]. BLUiR outperforms previous techniques on standard Java benchmarks. However, a limitation of these studies is that they focus on

[33] Please note that Dr. Julia Lawall, Dr. Sarfraz Khurshid and Dr. Dewayne Perry are the co-authors of this paper. They all helped me brainstorm the idea, design the empirical study, and improve the presentation of the paper.

software written in object-oriented languages, primarily Java. On the other hand, much of the most critical and widely used software, such as operating systems, compilers, and programming language runtime environments, is written in C. Indeed, as of May 2014, C is the most popular programming language according to the TIOBE programming language popularity index [147]. Nevertheless, there is a lack of an established dataset of large-scale, widely used C software, and a lack of easy-to-use tools for manipulating C code. Therefore, we yet do not know the efficiency of IR-based bug localization tools for C code. Most previous bug localization studies have also acknowledged this limitation [28, 127, 138].

In this study, we perform a large-scale experiment to investigate the efficiency of IR-based bug localization for C systems. To this end, we have created a dataset consisting of more than 7,500 bug reports from five popular C projects, and tested BLUiR on this dataset. We focus on the following research questions:

RQ1. How do the IR-related properties of C software compare to those of Java software?

RQ2,3. How does the accuracy of bug localization compare between C and Java software, at the file level (RQ2) and at the function level (RQ3)?

RQ4. How does the use of English words in software affect the accuracy of C and Java bug localization?

RQ5. How do preprocessor directives and macros in C code affect the accuracy of bug localization?

RQ6. How much do the different structural elements of C and Java code contribute to the accuracy of bug localization and how does this contribution vary between C and Java code?

Our results show that:

- While structured IR-based bug localization gives comparable accuracy for C code

and for Java code, the benefit for C code over language-independent IR-based bug localization is less than for Java code.

- The rate of English words in methods and identifiers differs greatly between C code and Java code; the fact that IR-based bug localization gives good results on both suggests that the rate of English words is not a good predictor of bug localization success across programming languages. However, we did find that for C programs, there is a correlation between the use of English words in the code and success of bug localization.

- Adequate parsing technology exists such that macros are not a major obstacle to IR-based bug localization.

- Bug localization for both C and Java code mostly relies on similar information: names of defined methods/functions and names of referenced identifiers. Bug localization for Java also benefits from the name of the defined class, while the C counterpart, *i.e.*, the file name, provides less information.

Our contributions include: 1) a dataset consisting of more than 7500 bug reports with their location in the source code at file level and function level for C programs, 2) a prototype to localize bugs in C systems; and 3) more generalizable results on the efficiency of IR-based bug localization.

7.2 Methodology

We now describe the methodology that we use to set up our experiments. This includes creating a large-scale dataset for C programs and adapting BLUiR for C code.

7.2.1 Creating a Dataset

To evaluate an IR-based bug localization tool retrospectively on a software project, we need to have the project's bug reports, program source code, and a means of identifying the files that were eventually fixed for the given bugs. Although getting bug reports and source code for various open source projects is fairly straightforward, determining the fixed files for a given bug is more challenging, since typically the bug tracking system and the version control system are independent of each other. Although there are many commits where developers indicate that a bug has been fixed, projects vary in the degree to which a reference to the bug tracker is provided. Furthermore, different projects have different conventions for how bug tracker references are indicated. We now describe how we map bug reports to the commits and to the affected code, at both the file and the function level.

At the File Level

Most of the software projects in our dataset, presented in Section 8.1, use `git` for version control and `Bugzilla` for bug tracking. Git commit messages are free form, and thus developers may reference bug reports in any manner. To determine how the developers of a given project typically refer to bug reports, we first searched through all the commit messages for the keywords `bug`, `issue`, and `fix`. If we found any of them, we then searched for any number, that could be bug numbers. Then, we manually analyzed a number of commits selected in this manner from each system. This analysis revealed some common patterns, including a complete Bugzilla URL for the Linux kernel and the keyword `PR` followed by a bug number for GDB and GCC. For one of our considered projects, WineHQ, however, the above process gave no results. We thus consulted with a developer from the WineHQ community who informed us that in this community the convention is for the bug report to refer back to the commit, rather than the commit referring to the bug report. Indeed, in the WineHQ Bugzilla, there is a dedicated field for a git commit id. However, many of these fields are empty since the field is not required. After identifying the bug

fixing commits, we extracted the names of the files that were changed and stored the bug report id and corresponding changed files in a JSON file for each project.

One of our considered projects, Python, uses `mercurial` rather than `git` and uses a dedicated bug tracking system rather than Bugzilla. Nevertheless, the process is essentially the same. By following the above process, we have found that Python bug report numbers are indicated by # followed by a number in the mercurial commit messages.

At Function Level

To construct the dataset at function level, we need to know the name of the function in which each bug fix occurs. For this, we use the command `git show -U0` to list the differences between the state of each file before and after the bug fix. `git show` produces the result in "unified diff format" [85], which normally shows the function header preceding each hunk, as illustrated by the following:

```
@@ -71,6 +71,17 @@ static int acpi_sleep_prepare(u32 acpi_state)
```

The `-U0` option furthermore tells `git show` to use no context information, which reduces the chance that a hunk will cross function boundaries. This approach, however, is still not completely reliable, *e.g.*, it may produce a recent label rather than a function header. We consider only those cases where the hunk header contains an open parenthesis, which has a high probability of indicating a function name.

Collecting information from bug reports

From the mapping we created in step 1, we have bug identifiers. Then, we use the Bugzilla Java API to download the summary and description for the bug report associated with each bug identifier. For Python, which does not use Bugzilla, we download the corresponding page from the dedicated repository[34] and parse it in an ad hoc manner to extract the summary and description. In each case, the description contains only the original report text,

[34] http://bugs.python.org/

and does not contain any subsequent discussions, subsequently proposed patches, lists of fixed files, etc.

7.2.2 Adapting BLUiR

Our experiment uses BLUiR for C code. However, since BLUiR was designed for Java code, we have had to adapt it for experiments involving the C programming language.

Collecting information from C code

From the C code, we need to obtain the names of the defined functions and the identifiers used in each file. For this, we must parse the C code. A challenge in parsing C code is that such code may use C preprocessor directives, to include header files, to express conditional compilation and to use macros. One strategy would be to apply the C preprocessor before processing, resulting in a file that conforms to the standard C grammar. This approach, however, has numerous disadvantages. It would duplicate commonly used header files in every C file that uses them, which would explode the code size and could dilute the information that is relevant to bug localization. Furthermore, preprocessing the code would eliminate macro names, which are often more informative than the code that they expand into. In the case of software in which variability is expressed using conditional compilation, it would result in discarding the code that does not correspond to a chosen configuration. Finally, in the case of the Linux kernel, we have found that the result of preprocessing is so large that collecting and processing the relevant terms from the expanded code is impractical, in terms of both computing time and disk usage.

To avoid these problems, we use a parser for C code, developed as part of the program matching and transformation tool Coccinelle [104], that does not require preliminary processing by the C preprocessor. Instead, the Coccinelle parser makes an effort to parse macro references, to parse around conditional compilation directives, and to parse other preprocessor directives, such as `#ifdef` and `#define` directly [103]. When parsing fails,

the parser recovers at the next top-level program unit, *e.g.*, function, variable, or type definition, thus minimizing the impact of the failure. The Coccinelle parser also has the ability to give feedback to the user about the most common parsing problems. Typically, these problems can be solved by providing a few artificial macro definitions in a configuration file. This configuration file typically needs to be created only once per software project, for use across multiple versions, as in our experience the set of problematic macros changes rarely. Except where noted, all of our experiments use these dedicated macro definition files whenever parsing is required. We examine these files in more detail in Section 7.4.5.

Our extension of BLUiR, built on the Coccinelle C parser, collects function names, identifiers, and words appearing in comments. Function names and identifiers are collected both in their entirety and are split at underscores and according to Camel Casing. Identifiers are collected from variable names, function names, type names, structure field names, function parameter names, and goto label names.

Retrieval

BLUiR supports *language-independent* retrieval, *flat-text* retrieval and *structured* retrieval. Since language-independent and flat-text retrieval are not concerned with distinguishing terms, and differ only in the strategy for extracting them, retrieval is the same for both C and Java code. However, since structured retrieval distinguishes between different types of terms based on program constructs (class, methods, variables, and comments) we have to classify C constructs within these categories. We consider C file names to be equivalent to Java class names and C function names to be equivalent to Java method names. Identifier names and comments are the same for both languages. Then, we apply the same underlying technique to compute the similarity score between a query and a collection of documents, to rank C files as described for Java language in the previous chapter.

7.3 Datasets and Metrics

Because there is no standard dataset for C software, we had to create one. In this section, we motivate our choice of software, and present the metrics that we use to compare bug localization for C code with bug localization for Java code.

7.3.1 Datasets

Generally, bug localization is more useful for large-scale systems, where developers could have trouble localizing bugs manually. Therefore, we have selected a number of C projects that are well known and large, that have a long development history, and that have a dedicated bug tracking system containing a large number of bug reports. In this way, we have chosen five open source projects, ordered below from smallest to largest in terms of the number of lines of code:

- Python 3.4.0: The runtime of the Python programming language.[35]
- GDB 7.7: A debugger for programs written in C, C++, and many other programming languages.[36]
- WineHQ 1.6.2: A compatibility layer, making it possible to run Windows applications on POSIX compliant operating systems.[37]
- GCC 4.9.0: A compiler for programs written in C, C++, and many other programming languages.[38]
- Linux Kernel 3.14: The kernel of the Linux operating system.[39]

[35] https://www.python.org/, http://hg.python.org/cpython

[36] http://www.sourceware.org/gdb/, git://sourceware.org/git/binutils-gdb.git

[37] http://www.winehq.org, git://source.winehq.org/git/wine.git/

[38] http://gcc.gnu.org/, git://gcc.gnu.org/git/gcc.git

[39] https://www.kernel.org/, git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

Table 7.1: Dataset Description for C Systems

Subjects Systems	Oldest patch	SLOC	For File Level		For Function Level	
			#Bugs	#Files	#Bugs	#Functions
Python	1990	380K	3,407	488	-	-
GDB	1988	1,982K	195	2,655	177	41,298
WineHQ	1993	2,340K	2,350	2,815	2,218	89,430
GCC	1988	2,571K	216	22,678	193	75,746
GCC_NT	-	2,062K	-	2,473	-	40,684
Linux kernel	2005	11,829K	1,548	19,853	1,178	347,057

Table 7.2: Dataset Description for Java Systems

Project	Description	Oldest Patch	SLOC	#Bugs	#Files
SWT 3.1	Widget toolkit for Java	2004	78K	98	484
AspectJ	Aspect-oriented extension to Java	2002	323K	286	6485
Eclipse 3.1	Popular IDE for Java	2002	1,579K	3,075	12,863

Table 7.1 presents some properties of these projects. For GCC we have created two versions since GCC has a large testsuite that consists of more than 20,000 files in the form of C code. GCC_NT (no test) represents the version with these test cases removed. For comparison with Java, we use Zhou et al.'s [173] dataset, presented in Table 7.2, which we have used in previous work on BLUiR. We have not included the project ZXing from this dataset, because this project has only 20 bug reports.

In terms of size, the C projects fall into three groups: Python at under 400,000 lines of code, GDB, WineHQ, and GCC at around 2 million lines of code, and the Linux kernel at over 11 million lines of code. For both C and Java software, we computed the size using David Wheeler's SLOCcount,[40] which includes only the number of lines of C or Java code, respectively, not whitespace, comments, or code written in other languages. The C projects are substantially larger than the Java projects, with the second smallest C project, GDB,

[40] http://www.dwheeler.com/sloccount/

being 25% larger than the largest Java project, Eclipse.

In terms of development history (column **Oldest patch**, Table 7.1) all of our C projects date from around 1990. The current git repository of the Linux kernel, however, only contains commits going back to 2005, when git was adopted by the Linux kernel developers. Other projects imported their previous version control history into git, and thus we have commits from a wider time span for these projects.

Finally, the number of bug reports available for the different C projects varies widely, but remains within the same order of magnitude as the number of bug reports available for the different Java projects. We have followed the procedure described in Section 7.2.1 for identifying bug reports that can be linked to commits. We take only the bug reports for which the fixes touch at least one C file, and for which at least one of the affected files still exists in the considered version of the software. At the function level, we have only considered the bug reports for which at least one name of an affected function can be identified from the associated patch, as described in Section 7.2.1. For Python, we have no function-level information, as Python uses `mercurial`, whose patch viewer does not make function header information available.

7.3.2 Evaluation Metrics

To evaluate the efficiency of BLUiR for C systems, we used the same set of metrics that we used for Java programs in Chapter 6: Recall at Top N, Mean Average Precision, and Mean Reciprocal Rank.

7.4 Results

We now present our experimental results. First, we consider some properties of the C and Java software that may affect the applicability of IR-based bug localization. Then, we consider our research questions, as defined in Section 7.1, related to the accuracy of various BLUiR-based approaches and to several details of the bug localization process.

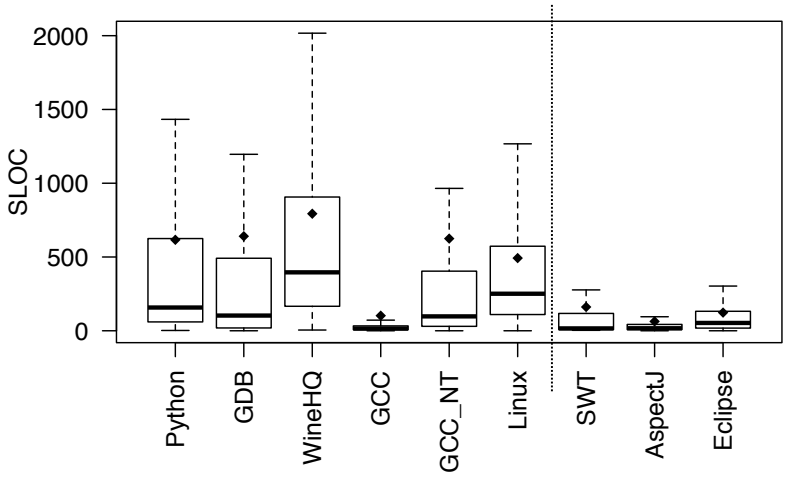

Figure 7.1: Comparison of File Size

7.4.1 RQ1: IR-related properties of C and Java software

We first investigate two IR-relevant properties: file length and nature of terms used in C and Java programs, which have an impact on the results of any IR system. If these properties of C and Java software are different, then the effectiveness of IR-based bug localization may be different as well.

File Size. Figure 7.1 displays the file size of each software project in terms of the number of lines of code without comments (SLOC). We observe that the SLOC distribution of GCC files is completely different from that of the other projects. As we discussed in Section 7.3.1, the source code of GCC contains a large suite of test cases, amounting to more than 20,000 small .c files. As we expect that few bug reports relate to bugs in test cases, we also include in Figure 7.1 information for GCC with all test cases excluded (GCC_NT).

Figure 7.1 shows that the median sizes of C files, indicated by the thickest horizontal line, vary from 97 (GCC_NT) to 396 (WineHQ) SLOC, whereas they vary from 16 (SWT) to 53 (Eclipse) SLOC for Java files. However, for the C projects, the average, marked by the diamond, is typically much higher than the median, indicating that the file

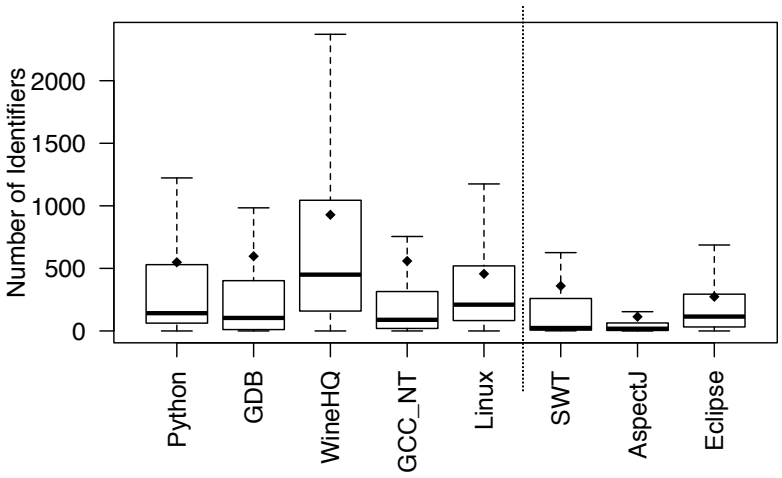

Figure 7.2: Comparison of Number of Terms

sizes are highly skewed. The average size of the C files varies from 492 (Linux) to 793 (WineHQ) SLOC, whereas it varies from 63 (AspectJ) to 161 (SWT) SLOC for the Java projects. Therefore, overall, C files are, on average, substantially larger than Java files, for our considered projects.

As an alternate measure of size, we also investigate the total number of terms in the source code that would be actually used for IR, as presented in Figure 7.2. We see that the number of terms present in the C files is also considerably higher than that of Java files. The average number of terms in the C files varies from 456 (Linux) to 928 (WineHQ), whereas for Java it varies from 114 (AspectJ) to 360 (SWT).

Terms in Source Code.. Bug reports are generally written in natural English. IR-based bug localization generally focuses on identifier (class, method, and variable) names and comments, because these are the places where developers can use natural English. In object-oriented programming languages, developers are strongly encouraged to use meaningful words in identifier names. For example, the Eclipse Foundation has very specific naming conventions.[41] Therefore, IR-based bug localization is expected to work well for Java

[41] http://wiki.eclipse.org/Naming_Conventions

projects. Since the C programming language is used by a different group of people and is generally used for different types of software (e.g. systems software rather than applications) than Java, the programming style of C may be considerably different.

Our study of the use of English words focuses on method and identifier names, omitting comments on the assumption that comments almost always contain natural English text, regardless of the programming language. To have the greatest chance of finding English words, we split method and identifier names according to the conventions of CamelCase and additionally split C names at underscores (_), following the conventions commonly used in our software projects. Finally, we exclude the words `get` and `set` when counting English words in Java programs, since these words are very frequent, due to the use of getter and setter methods. In our C projects, we have found that developers use underscores in 88%-97% of method names and 43%-55% of identifier names, and in our Java projects, we have found that developers use CamelCase in 62%-80% of method names and 38%-46% of identifier names.

Once the identifier names have been split into tokens (or terms), we match each result against a comprehensive list of 354,983 English words.[42] Then, we calculate the percentage of terms that are found in the list of English words. Since a term can appear multiple times in the code, we also calculate the percentage of unique terms that are also English words. That is, if E is the set of terms that are found in the dictionary, with all duplicates removed, and T is the complete set of split words, again with duplicates removed, we calculate the unique word percentage as $\frac{|E|}{|T|} \times 100$. For function (or method for Java) names, T is the set of split words obtained from the names of defined functions, while for identifier names, T is the set of split terms obtained from all identifier names, including the names of called functions. From the results, we see that developers tend to use English terms in both function and identifier names. However, the higher unique percentages for function names show that there are more non-English words in identifier names than

[42] http://www.infochimps.com

Table 7.3: Presence of English words in Source Code

Term Type	Function/Method-Terms*		Identifier-Terms	
	Actual %	Unique %	Actual %	Unique %
Python	66%	44%	67%	33%
GDB	67%	32%	59%	18%
WineHQ	72%	20%	59%	12%
GCC	71%	30%	69%	21%
GCC_NT	76%	46%	72%	25%
Linux	59%	20%	59%	10%
SWT	95%	85%	84%	48%
AspectJ	92%	67%	91%	52%
Eclipse	97%	75%	94%	48%

function names. From the results we also see that the presence of English words in Java programs is considerably higher than in C programs, both in terms of actual and unique percentages.

Therefore, the overall results show that C and Java are not only different due to programming paradigms (procedural vs. OOP) but also different from an IR perspective. Our next research questions investigate how IR-based bug localization, which has previously mostly been evaluated for Java programs, performs in practice for C programs.

7.4.2 RQ2: Accuracy of Bug Localization at the File Level

Figures 7.3 and 7.4 show the accuracy of bug localization for the C and Java projects, using language-independent retrieval, flat-text retrieval, and structured retrieval, in terms of Recall at Top 1, Top 5, and Top 10, and MAP and MRR. In Figure 7.3, the Recall at Top 5 and Recall at Top 10 bars represent the increase in recall as compared to taking into account the Top 1 or Top 5 files, respectively. We first assess the overall results, and then compare the results for C and Java projects as the number of files considered from the top of the ranked list changes, and in terms of the bug localization strategy.

For all but the smallest projects (Python and SWT), whether C or Java, bug local-

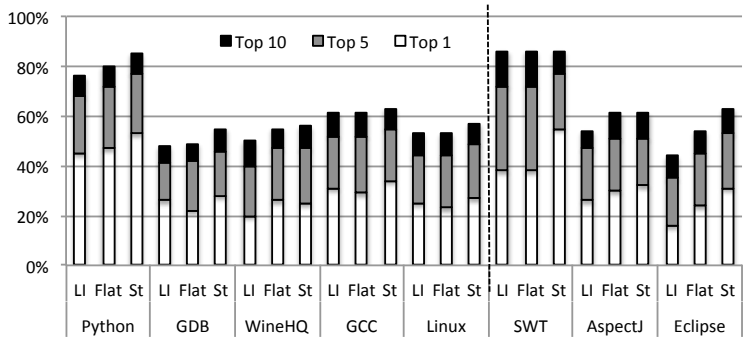

Figure 7.3: Comparison of Recall at Top N for Different Strategies

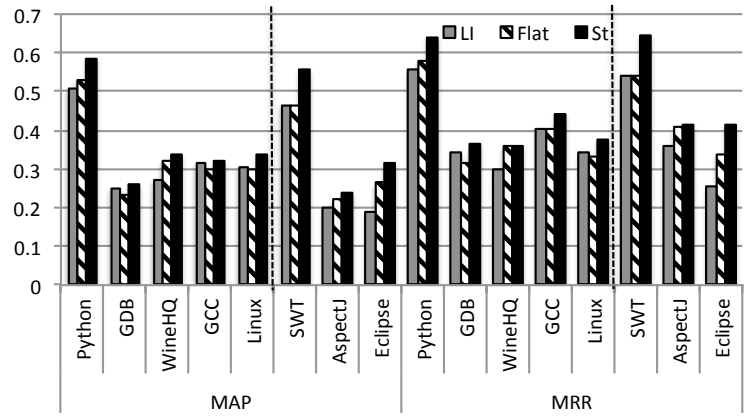

Figure 7.4: Comparison of MAP and MRR for Different Strategies

ization gives roughly the same accuracy, with Recall at Top 1 values of 20-34% for C and 16-32% for Java, and Recall at Top 10 values of 48-63% for C and 44-63% for Java. We see the same similarity in the MRR scores. The C projects, however, have higher MAP scores than the Java projects, ranging from 0.249 to 0.337 for C (with 0.586 for Python) and from 0.190 to 0.316 for Java (with 0.557 for SWT). Thus, bug localization is more successful in finding all of the files that should be changed for the C projects.

Next, we consider the impact on accuracy of considering more files in the ranked list, by comparing the result for Recall at Top 1 with the results for Recall at Top 5. For language-independent retrieval, considering more files gives less of an improvement for the C projects (51-76%, except for WineHQ, where there is a 100% improvement) than for the Java projects, where the improvement is always over 80%. On the other hand, for flat-text retrieval, the improvement is about the same, being 70-90% for all projects except Python. For structured retrieval, considering more results gives more of an improvement for C (62-88% for all C projects except Python) than for Java (59-71% for all Java projects except SWT). The smallest projects, Python and SWT, achieve an improvement of 45% and 40%, respectively, for Recall at Top 5 as compared to Recall at Top 1, but their Recall at Top 1 rates were already much higher, at 53% and 55%, respectively, than for the other projects.

Finally, we consider the improvement provided by flat-text retrieval and structured retrieval as compared to language-independent retrieval. In the case of flat-text retrieval, there is again a major difference between C and Java projects. For three of the C projects, GDB, GCC, and Linux, flat-text retrieval gives a *worse* Recall at Top 1, by up to 15% for GDB, while for Java, the result is always the same (SWT) or better. Indeed, flat-text retrieval increases Recall at Top 1 by 50% for Eclipse as compared to language-independent retrieval. However, the results for flat-text retrieval for C projects are not all negative; for WineHQ, which has the lowest accuracy for language-independent retrieval, at 20%, flat-text retrieval gives a 30% improvement, resulting in an accuracy that is comparable to that of the other larger projects. Structured retrieval gives an improvement over language-

independent retrieval for all projects. Nevertheless, the improvement is quite small for some C projects: from 26% to 28% for GDB, from 31% to 34% for GCC, and from 25% to 27% for Linux. In these cases, structured retrieval mostly just reverses the losses observed for flat-text retrieval. Indeed, WineHQ, which benefited most from flat-text retrieval, obtains a slightly worse Recall at Top 1 result with structured retrieval than with flat-text retrieval, from 26% to 25%. These results contrast with the results for Java, where structured retrieval gives a substantial improvement in accuracy, including an improvement of 94% in Recall at Top 1 for Eclipse as compared to language-independent retrieval.

In summary, we find that for both C and Java, the accuracy is roughly similar, except in the case of MAR, where bug localization is more successful for the C projects. We also find that the impact of taking into account more reported files varies between C and Java, depending on the retrieval strategy, and that structured retrieval provides less benefit for C projects.

7.4.3 RQ3: Accuracy of Bug Localization at the Function Level

Thus far, our results have been expressed at the file level. We have seen in Section 7.4.1, however, that for our projects, the C files are much larger, on average, than the Java files. While knowing the affected file may permit the developer to hone in directly on the problem, in the worst case, the C developer has on average, *e.g.*, approximately 2500 to 4000 lines of code to inspect when considering the Recall at Top 5 results, while the Java developer has only at worst on average 300 to 800 lines of code to inspect. Thus, we consider whether BLUiR can be effective at the function level on C projects, to further narrow down the search space of developers. Over all of the C projects, the average function size varies from 28 to 42 lines of code.[43] Thus, Recall at Top 5 at the function level for our C projects would be roughly comparable to Recall at Top 1 on average for our Java projects at the file level

[43] Function size is computed from the difference between the line number of the last line of the function and the line number of the first line, and thus may include lines containing only comments or whitespace.

Table 7.4: Function-Level Retrieval Accuracy

Project	Top 1	Top 5	Top 10	MAP	MRR
GDB	7%	21%	27%	0.073	0.145
WineHQ	8%	16%	21%	0.085	0.122
GCC	9%	18%	25%	0.081	0.144
Linux	11%	19%	24%	0.127	0.159

based on the number of lines to be inspected.

To investigate the accuracy of BLUiR at the function level, we constructed a document collection containing one document per function and applied BLUiR to it. Our results (in Table 7.4) show that the accuracy of BLUiR is much lower at the function level than at the file level. The recall in Top 1 ranges from 7% to 11%, whereas the recall in Top 10 ranges from 21% to 27%. We think this result is not surprising since an individual function provides much less information than a complete file. Furthermore, retrieval at the function level can be more expensive than retrieval at the file level since the number of functions can be much greater than the number of files. For example, in Linux Kernel, BLUiR ranks all the source code files in 5 seconds on average for a given bug report, whereas it takes 55 seconds per query at function level, on a machine having an Intel Core i7 @ 3.50GHz processor and 16GB memory.

To improve the performance of BLUiR at the function level, we then tried a two-step approach. First, we ran BLUiR at file level and took the top k files for function retrieval. From our previous results, we found that for all projects BLUiR can localize more than 80% of bugs within the Top 100 files. Thus, if we consider only the functions from these files, the number of functions for retrieval would be reduced a lot, without losing the buggy functions for more than 80% of the bugs. Therefore, reducing the candidate functions in this way should reduce the retrieval time but have little impact on accuracy. Our results show that this alternative approach indeed reduced the retrieval time considerably, while maintaining almost the same accuracy as considering all functions. The function-level retrieval now

requires only a fraction of a second after selecting the Top 100 files, reducing the total time from 55 seconds to 5 seconds.

7.4.4 RQ4: Impact of the use of English Words

We next investigate whether there is any relationship between the use of English words in method and identifier names and the accuracy of BLUiR. We noted previously that the Java projects use a substantially higher rate of English words than the C projects. Nevertheless, particularly in terms of Recall at Top 1, MAP, and MRR, Figures 7.3 and 7.4 show that we get equal or better results for the C projects than for the Java projects. Among the C projects, we have the highest rates of unique method names and unique identifier names for Python, and we obtain the highest accuracy for this project as well. We also observe that the Recall at Top N gradually increases with the increase in the unique percentages of English words in method names and identifiers, except for GDB. To show that this correlation is statistically strong, we calculated Pearson product-moment correlation coefficient between different accuracies and unique percentages. The correlation coefficient for method unique percentage and Recall at Top 1, and identifier unique percentage and Recall at Top 1 are 0.92 and 0.95, respectively. We also get 0.84 and 0.92 for Recall at Top 5. Figure 7.5 presents this trend with scatter plots and best fit regression lines computed by R.[44] But we do not observe the same trend in the case of Java, where the project with the highest accuracy, SWT, has the highest rate of unique method names, but does not have the highest rate of unique identifiers. We also have not found any systematic relationship for other Java projects. Therefore our overall results show that the greater usage of English words only increases the accuracy of bug localization for C projects.

[44] http://www.r-project.org

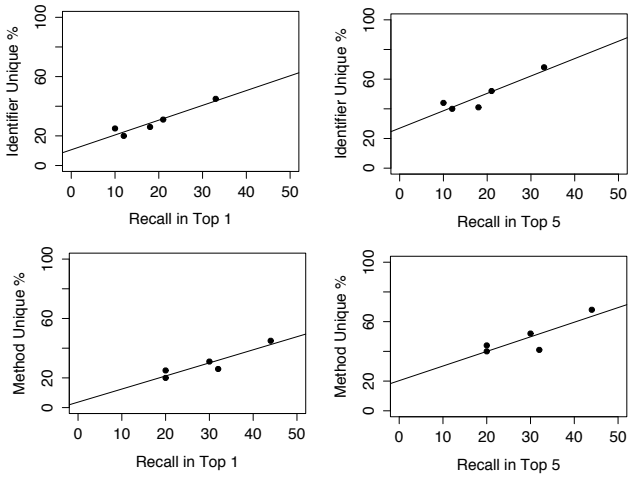

Figure 7.5: Correlation between Percentages of English Words and Accuracy

7.4.5 Impact of tool features

In this section, we study our results in more detail, to better understand the relationship between the performance of IR-based bug localization on C and on Java programs. First, we consider the effect of macros, which complicate the processing of C code and which are not present in Java code. Then, we consider the contribution of each kind of information used by structured retrieval to the final result.

RQ5: Effect of Macros. In Section 7.2.2, we explained why the presence of preprocessor directives and macros in C code can affect the analysis results. While our C parser tries to cope with unknown macro uses, in some cases, its heuristics are not successful, and some top-level variable or function definitions are not taken into account, potentially reducing the amount of information that is available. Better results can be obtained by providing a configuration file giving definitions for a few macros that are difficult to parse, based on feedback from the parser about common parsing problems. The time required for creating this configuration file mostly depends on the parsing time.

Table 7.5 shows the parsing time on our Intel Core I7 machine when no macro

Table 7.5: Properties of macro definition files

	No macro definitions		Custom macro definitions		
Project	**Parse time**	**Success rate**	**Macro definitions**	**Parse time**	**Success rate**
Python	51s	62%	24	39s	88%
GDB	8m 45s	78%	18	8m 34s	84%
WineHQ	6m 51s	70%	13	6m 3s	89%
GCC_NT	6m 14s	59%	41	4m 26s	77%
Linux	24m 43s	61%	234	19m 51s	84%

definitions are available, the percentage of files that are entirely successfully parsed in this case, the number of macro definitions in our customized macro definition file for each project, the parsing time when these definitions are available, and the percentage of files that are entirely successfully parsed in this case. For the Linux kernel, we use the existing default macro configuration file of Coccinelle [104], which targets Linux kernel code.

Figure 7.6 compares the accuracy of BLUiR when we use the project-specific macro definitions and when we do not use them. Results show that even though the case without specific macro definitions results in *e.g.*, only 59% of the files being successfully parsed in their entirety for GCC, the results are essentially the same, with a difference of at most 0.002 for MAP and 0.003 for MRR. Indeed, our C parser recovers at the start of the next top-level definition that it is able to identify, and thus in practice a parse error in one part of a file has no impact on the parsing of the rest of the file. Thus, plenty of information is available for bug localization, and BLUiR is able to localize bugs despite the parse errors.

RQ6: Importance of Information Sources. To better understand the previous results, we investigate which kinds of program terms (file/class names, function/method names, variable names, and comments) are more important. To this end, we run BLUiR on each type of term separately. Since this produces four sets of results for each project, we present only MAP for conciseness. MAP takes all the buggy files into account, and thus is the most comprehensive metric.

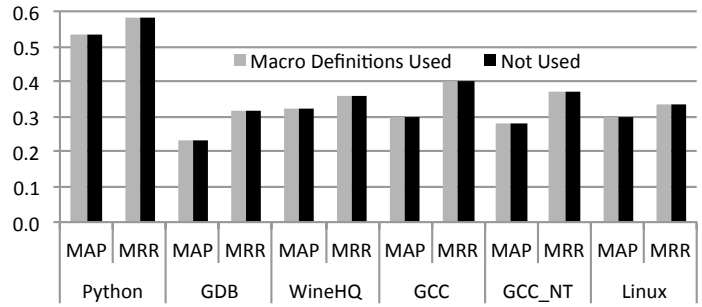

Figure 7.6: Effect of Macro Definitions on Flat-Text Accuracy

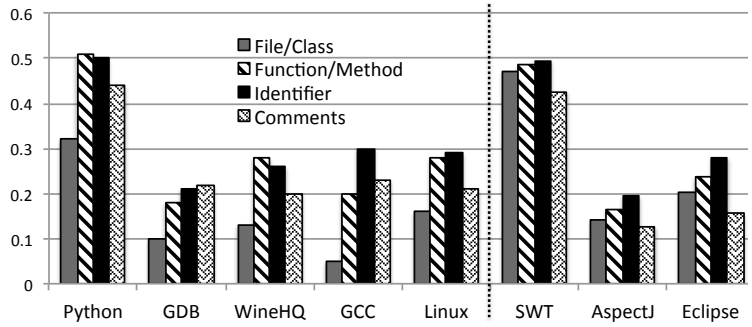

Figure 7.7: Mean Average Precision (MAP) for different kinds of program terms

The results show that Java class names are more important than C file names. For C programs, we see a large gap between the accuracy for file names and the accuracy for other terms for each project, whereas such gaps are small for Java programs. Also, we observe that although the number of method names is far smaller than the number of identifier names, method names carry a lot of information. For both C and Java programs, the MAP value based on only method names is very close to that of identifier names except for GCC. For some projects (Python, WineHQ, Linux Kernel, and all the Java projects), method names contribute more than comments. The overall results show that although all kinds of terms help localize bugs, for both languages, method/function names and identifier names are important for every project.

7.5 Threats to Validity

This section discusses the validity and generalizability of our findings.

Construct Validity: We used two artifacts of a software repository: source code and bug reports, which are generally well understood. We have used three popular metrics: Recall at Top N, MAP, and MRR, which are standard in IR, have been used in previous IR-based bug localization studies, and are straightforward to compute.

Internal Validity: To create the benchmark for C projects, we have relied on the information in version histories and bug tracking systems. However, for some cases this information may be inaccurate or incomplete, which may affect our results. Indeed, for GDB, WineHQ, and GCC, which are about the same size, we have widely varying numbers of linked bug reports, which may indicate that the developers of some projects do not mention such links systematically.

We have used a single release for bug localization in each system. Bugs that were previously fixed are no longer present in that code, and for old bug reports, the code may have changed substantially since the bug was encountered. Ideally, for each bug report we would extract the version from when the bug was reported to get the actual buggy code.

However, this approach is impractical for a large-scale experiment.

Most of our studied projects represent systems code rather than applications. Systems software may have its own set of development biases [159]. We may not capture concerns that are only present in software targeting other domains.

Like other IR-based bug localization studies, our results are intrinsically sensitive to the quality of the bug reports. An issue is the possible presence of "too well written" bug reports, *e.g.*, where a maintainer of the code has already solved the problem, and is using the bug repository to record his activities. Such reports could make bug localization unrealistically easy, as compared to reports from ordinary users, for which localization is actually needed. Indeed, Kochhar et al. [67] have found, in work concurrent with ours, that for three Java projects different from the ones considered here, around half of the bug reports contain the name of at least one of the classes that should be fixed, and that the sets of reports that contain the names of all of the classes that should be fixed have MAP scores 2.5-3 times higher than those that contain no such class names. In our dataset, we have found that 10%-19% of the bug reports of C projects contain the name of at least one file that was fixed, and likewise 5% to 29% for Java projects. If we ignore the extension (.c or .java) of the file name, the ranges vary from 25% to 29% for C projects and from 32% to 62% for Java projects. We furthermore observe that 19% of Python reports contain the name of at least one file that should be fixed, while only 10% of the WineHQ reports do, which may account for some of the difference in the success of bug localization on these two projects (see Figure 7.3). Nevertheless, it is hard to determine what proportion of these bug reports are actually bias in the dataset, since file names or class names may coincide with natural English words. We also found 34 Linux Kernel bug reports that contain the name of at least one file that was not fixed. Thus, file name information may not always make bug localization trivial. We leave a more detailed study of the kinds of information present in bug reports and how this information impacts the success of manual or automatic bug localization to future work.

Finally, our tools may contain errors. We have carefully inspected our code and rigorously tested it on a known dataset.

External Validity: We have used five C software projects and three Java software projects in our experiments. All are open source. Although, they are popular projects, our findings may not be generalizable to other open source projects or to closed source projects. However, to the best of our knowledge, this is the largest experiment for IR-based bug localization. The risk of insufficient generalization could be mitigated by expanding the benchmark to include more software projects (both open source and closed source). This will be explored in our future work.

7.6 Related Work

The literature on finding bugs and other features of source code is enormous. We thus focus on related work on matching some form of user-provided query to regions of source code, as well as studies that compare results for C to results for Java.

Bug localization: IR-based bug localization techniques have recently gained attention from the software engineering research community. Researchers have proposed a number of retrieval techniques to improve the rank of buggy files for a given bug report query. Lukins et al. [83] use the Latent Dirichlet Allocation (LDA), a generative statistical model widely used for topic modeling, for bug localization. Rao et al. [115] compare a number of techniques such as the Unigram Model (UM), the Vector Space Model (VSM), the Latent Semantic Analysis Model (LSA), the Latent Dirichlet Allocation (LDA), the Cluster Based Document Model (CBDM), and various combinations to investigate their relative performance for bug localization. Based on their evaluation, they concluded that sophisticated models such as LSA, LDA, or CBDM are not necessarily better than simpler models such as UM or VSM.

Recent bug localization techniques go beyond traditional IR by using additional information from software repositories. Sisman and Kak [138] incorporate version histories

in an IR model. Nguyen et al. [100] introduce BugScout, a topic model-based tool for narrowing the search space for buggy files. Zhou et al. [173] incorporate program file length and similar information into the TF.IDF term weighted VSM. BLUiR incorporates structural information into an IR model [127]. AmaLgam takes into account not only structure information, but also the set of files that have recently been subject to bug fixes and the set of files fixed by recent similar bug reports [150].

However, all of the above techniques have been evaluated on datasets containing object-oriented programs, particularly Java. Our focus is on whether techniques that work for object oriented languages also work for imperative languages. Thus, we evaluate IR-based bug localization with C programs.

Other IR problems and C code: Several other works have considered other kinds of localization problems for C code. Wang et al. [154] study the effectiveness of a wide range of information retrieval techniques on localizing concerns in Linux kernel source code. Rather than bug localization, they study the problem of feature localization, mapping a feature, expressed as a preprocessor flag, to the relevant source code, defined as the function whose definition is somehow affected by the flag's value. Their experiment did not involve bug reports and was limited to the Linux Kernel. Poshyvanyk et al. [109] formulated the feature location problem as a decision-making problem in the presence of uncertainty and evaluated their approach by localizing bugs in Mozilla. Mozilla contains both C and C++ code, but only the C++ code was taken into account in the evaluation.

Marcus et al. [92] use an old version of the NCSA Mosaic web browser, written in C, to test their approach to concept location, where the goal is to find code relevant to a developer-provided code search request. Developer searches may have different textual properties than bug reports. They use latent semantic indexing (LSI), which is different than the TF-IDF based approach used by BLUiR. Finally, the source code size is small (95KLOC) and there is no comparison between C and Java.

C vs. Java: Lucia et al. [80] compare the result of spectrum-based fault localiza-

tion, in which probable locations of faults are identified based on succeeding and failing execution traces, on C and Java programs, using a variety of metrics. They find that the results are overall better on C code than on Java code, and that the set of metrics that perform best is different in the two cases. Nevertheless, they consider a different kind of localization problem than the one considered here (execution trace based rather than IR-based), the C software projects considered are much smaller than the ones considered here, being at most 6,218 lines of code, and the issues of preprocessor directives and macros, which are critical to treating large C software projects, are not addressed.

7.7 Summary

In this study, we have compared the results of IR-based bug localization on large, widely used C and Java software, thus giving a richer perspective on the effectiveness of bug localization than that provided by previous studies, which were primarily limited to Java. The main technical challenge in applying IR-based bug localization to real C projects is to cope with the use of C preprocessor directives and macros. We have shown that this issue can be addressed in a lightweight way using existing technology. Another main lesson of our work is that even though C developers use English words substantially less often in their method and identifier names than Java developers, IR-based bug localization can still be effective on C code, comparably to Java code. On the other hand, our considered C projects benefit less than our considered Java projects from taking into account information from program structure, an extension to IR-based bug localization that has been proposed in a number of recent techniques. This suggests that a greater understanding may be needed of the properties of bug reports and source code to make IR-based bug localization more effective in practice.

Dataset. Our dataset of C projects is publicly available at https://utexas.box.com/icsme2014-dataset

Chapter 8

Natural Language Processing to Improve IR-based Bug Localization: An Exploratory Study

8.1 Context and Motivation

There has been a great number of endeavors from researchers to improve the accuracy of IR-based bug localization. Along the same line, we have also introduced BLUiR to improve the the accuracy based on structured retrieval. The accuracy of IR-based bug localization has been improved further by integrating information from version control systems and bug repositories, e.g., to identify files fixed for similar or recent bugs [151], as depicted on the right side of Figure 8.1.

Although having the information about similar bug reports and the mappings between previous bug reports and the corresponding source code files fixed is an added advantage, many projects have no formal mechanisms for indicating what files are modified to fix which bugs. For example, while in Jira there is a dedicated field for making a connection between a bug report and the corresponding commit, Bugzilla does not have such an explicit field. Some approaches [63, 173] have thus instead relied on bug report links identified in

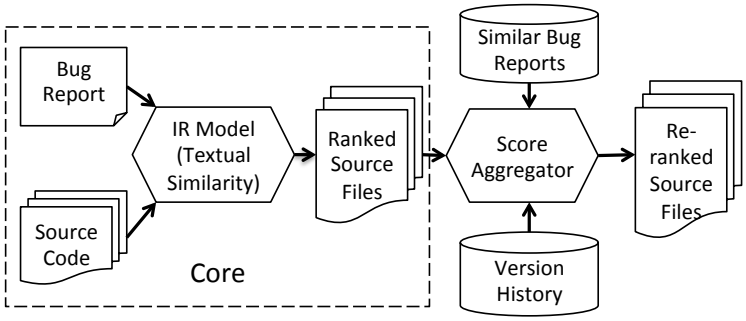

Figure 8.1: High level structure of a state-of-art IR-based bug localization tool

developer-entered commit logs. Developers, however, do not always write which commits are bug-fixes or which bugs have been fixed [7]. Therefore, it remains important to improve the core (left side of Figure 8.1) of IR based bug localization, i.e., without the help of any additional information that is not directly available in a software repository. This leaves us focusing on only bug reports and source code.

Since bug reports are mostly written in natural English and natural language processing (NLP) [129] is concerned with the interactions between computers and human (natural) languages, the goal of this proposal is to explore the use of NLP systematically. NLP has already been used in various software engineering applications such as requirements engineering [99], concept location, and code search subfields [79], where developers deal with natural English. Therefore, in this proposal, we plan to investigate the extent to which natural language processing of bug reports can improve the accuracy of the core IR model for bug localization. In particular, we plan to investigate the following possibilities:

- **Textual strength of the bug report:** Before exploring different NLP techniques, we first investigate the extent to which bug report text can increase the accuracy of bug localization, at least theoretically. Indeed, if the terms required to accurately localize a bug are not present in the bug report, no natural language technique will effectively help to improve the accuracy.

- **Parts of speech:** Different parts of speech carry different information. Previously, researchers have found that considering only nouns improves accuracy for other kinds of traceability recovery [26]. We investigate whether the same is true for bug localization.

- **Various kinds of bug report text:** A bug report has various kinds of text, such as the bug summary, the bug description, the log, quoted terms, etc. We investigate whether some of these kinds of text are more important than others.

- **Synonyms:** Bug reporters and developers may use different, but synonymous, terms in a bug report and in the corresponding source code files. Normalizing such terms in the bug report and the source code, or expanding bug reports with synonyms could improve the accuracy of bug localization. Query expansion using synonyms has been found to improve the accuracy in the area of code search [79]. In this proposal, we plan to investigate whether synonyms are useful to improve the accuracy of bug localization.

- **Related forms of words:** Like synonymous terms, developers and bug reporters may use different forms of parts of speech of the same word (e.g., "went" vs. "go") in bug reports and in the corresponding source code files. We have observed that stemmers in wide use, such as Porter and Krovetz, do not eliminate all such differences. Therefore, we investigate whether eliminating such differences improves bug localization.

- **Irrelevant files:** In the ranked list of probable files, there may be files that are textually similar to the bug report, but are semantically irrelevant. As class and package names may provide a lot of semantic information about a file, we investigate whether we can filter out some of the irrelevant files according to this information.

We believe that our results can give researchers more insight about the textual perspective on bug reports to improve bug localization.

8.2 Study Setup

In this section, we define our research questions, discuss how we create the dataset, and present the evaluation metrics we used to interpret our results.

8.2.1 Research Questions

To investigate the possible opportunities that we mentioned in Section 8.1 to improve bug localization, we focus on the following research questions:

RQ0: What is an approximate upper bound on the accuracy of IR-based bug localization given only bug report text?

RQ1: Do different parts of speech carry different weights for bug localization?

RQ2: Does the bug summary provide more important information than the bug description?

RQ3: Does expanding queries with synonyms improve bug localization?

RQ4: Does expanding queries with different forms of parts of speech improve bug localization?

RQ5: Can we somehow filter out irrelevant files from the ranking to improve bug localization?

8.2.2 Dataset

We have used Zhou et al.'s [173] dataset for our study since this dataset has been used in the evaluation of many IR-based bug localization tools [67, 127, 151, 152, 173]. However, we made three adjustments to the dataset to ensure a fair evaluation for our experiments. First, in the original dataset, three separate Eclipse projects (PDE, JDT, and Platform) are combined into one. It seems improbable, however, that a developer would e.g., search for a JDT bug in PDE, and thus we have separated the three projects. Other studies[71, 123]

Table 8.1: Overview of Dataset

Project	#Bugs	#Files
SWT 3.1	91	484
PDE 3.1	195	1,031
JDT 3.1	488	3,648
Platform 3.1	1,602	7,598

on bug reports have considered these projects separately as well. Second, there are some bugs in the dataset that were misclassified in terms of the tagged project. We say a bug report is misclassified when it is tagged as a bug of project X but only files in other projects have been fixed. To create a clean dataset, we have removed those bug reports. Third, we remove all the bug reports that contain stack traces. Recently, researchers have proposed special retrieval techniques [97, 157] to improve bug localization for bug reports that have stack traces. In this work, we are interested in the bug reports that are written mostly in natural language. In this way, we obtained 2,376 bug reports, out of 3,075 in the original dataset. Table 8.1 summarizes the properties of our dataset.

8.2.3 Evaluation Metrics

To evaluate the effectiveness of ranking, we calculate the same three metrics: Recall at Top N, Mean Average Precision, and Mean Reciprocal Rank, as used in previous chapters. In addition, we also define the following metric:

Result Quality (RQ): MAP and MRR each give a particular measure of the overall result quality. However, their usefulness in practice is limited especially for bug localization, because they consider the complete ranked list of files, even though developers are unlikely to examine more than the top few. For example, if a faulty file is ranked at the 100th position in result R_1 and at the 200th position in another result R_2, R_1 would have better MAP and MRR values, but the improvement would be of no practical use to developers. Therefore, we define the following alternate result quality metric (RQ) to choose one

result over another.

$$RQ = \frac{w_i * Recall@1 + w_j * Recall@3 + w_k * Recall@5}{w_i + w_j + w_k} \tag{8.1}$$

In our study, we set w_i, w_j, and w_k to 3, 2, and 1 respectively, to put the most weight on Recall at Top 1 and the least weight on Recall at Top 5.

8.3 Study Results

This section presents our results with respect to the research questions we defined in Section 8.2.1.

8.3.1 Approximate accuracy upper bound

Before we start our exploration into different techniques, we first investigate the approximate theoretical upper bound on how much text analysis techniques can increase the accuracy of bug localization. Indeed, if the terms required to accurately localize a bug are not present in the bug report, no natural language technique will effectively help to improve the accuracy.

Methodology

In our previous study [127], we observed that bug reports frequently share some terms with the corresponding source code files that were eventually fixed. However, the existence of some common terms between a bug report and the corresponding faulty files does not necessarily mean that we can localize the bug in the Top 5 files of the ranked list. The reason is that those terms could also be present in non-faulty files. Therefore, we investigate whether bug reports have enough discriminating terms to localize the faulty files in the Top 5 files. One approach would be to construct queries from all possible subsets of terms found in a bug report and then see if there is any such query that can localize the bug in the Top 5 files. However, such an approach would have an exponential time complexity and thus

be impractical, since a bug report may contain hundreds of terms. Therefore, we design a more approximate approach that has linear time complexity.

First, we extract all the words from the summary and description fields of the bug report and from these words create a set T_r. We further augment T_r with the result of splitting any compound words in T_r, according to CamelCase. T_r then forms the query. Then, we perform unstructured retrieval using BLUiR for this query. BLUiR ranks all source code files, and thus this result gives the rank of all faulty files. Let us assume that the rank of the top faulty file is k_{top}.

Next, we iteratively consider each term and observe its impact on the results. For each term t, we first remove it from T_r, then construct a query from the remainder of T_r, and then perform an unstructured retrieval using BLUiR. Let the rank of the first faulty file in the result be k. If $k_{top} > k$, i.e., the rank of the top faulty file has improved, t is likely not important, or even detrimental, for the query and we permanently remove it from T_r. We also update k_{top} with the new top value k. On the other hand, if the rank of the top faulty file deteriorates due to the removal of t, we add t back into T_r. In this way, after iteration through all terms in T_r, we have removed all possible noise from the query, thus getting the approximate maximum accuracy for the given bug report text.

From the results, for a given bug report i, we can observe both the approximate minimum number of terms (m_i), as the number of terms that result from this process, and the approximate maximum accuracy (k_{top_i}). For a subject system having n bug reports, we get $M = \{m_1, m_2, ..., m_n\}$ and $K_{top} = \{k_{top_1}, k_{top_2}, ..., k_{top_n}\}$. Then we compute the minimum, maximum, mean, and median of M to understand the number of key terms in all bug reports of a project and compute the recall at Top 1, Top 3, and Top 5 using K_{top}. to understand the approximate maximum accuracy.

Please note that sophisticated IR approaches such as Latent semantic indexing (LSI) and Latent Dirichlet Allocation (LDA) could retrieve relevant documents even when there is no common terms between a query and documents. Furthermore, we have not considered

synonyms and different forms of the same word for this experiment. The reason is that we wanted to perform a simple experiment for estimating an approximate upper bound to understand the remaining space for improvement in IR-based bug localization. There is no impact of this metric on the other findings of the study.

Results

Table 8.2 presents an overview of our results. We observe that, for the considered software projects, bug reports have enough discriminating terms to localize on average 70% of the bugs in the Top 1 file, 80% in the Top 3 files, and 82% in the Top 5 files. As these are quite high numbers, we conjecture that our approximate approach gives a result that is near the upper bound.

We also observe that, in all projects, on average 3 terms in queries are effective in discriminating the faulty files from other files. Specifically, for 75% of the bug reports, only 2 to 5 terms were enough to rank the faulty files in the Top 5 files. Similar results have been obtained in the NLP domain. For example, Kumaran and Allan [68], in a study on how to effectively use long queries for natural English documents, observed that the best sub-queries never contain more than 10 terms, with most having 6 or fewer. The challenge, though, in both contexts, is that even though only a few terms are required, we do not know a priori which terms those are. In this study, we investigate whether different NLP techniques help us weight each term in a query in such a way that some key terms get more weight than others, enabling these terms to play a more important role in the ranking of faulty files.

8.3.2 Part-Of-Speech Based Term Weighting

Different parts-of-speech carry different information. Researchers have found that considering only nouns in the query improves other kinds of traceability recovery [26]. We investigate whether the same is true for bug localization.

Table 8.2: Approximate Maximum Accuracy

Project	Min # of key words				Top 1	Top 3	Top 5
	Min	Med	Mean	Max			
SWT	1	3	3.2	32	74(81.3%)	86(94.5%)	88(96.7%)
PDE	1	3	3.3	8	136(69.7%)	167(85.6%)	172(88.2%)
JDT	1	3	3.3	8	323(66.2%)	370(75.8%)	385(78.9%)
Platform	1	3	3.5	20	1,121(70.0%)	1,269(79.2%)	1,307(81.6%)

Methodology

To investigate whether part-of-speech (POS) based term weighting improves bug localization, we first have to determine the POS of each word. For this, we have used the POS tagger from the Stanford NLP toolkit [149]. The Stanford POS tagger takes into account both preceding and following words via a dependency network representation. Furthermore, it leverages the broad use of lexical features, including jointly conditioning on multiple consecutive words, and fine-grained modeling of unknown word features.

Once we identify the POS of each term, we explore the possibility of discarding terms of some kinds of POS or appropriate term weighting based on POS. Since a previous study [26] on other traceability link discovery problems reported that taking into account only *nouns* is sufficient to increase accuracy, we begin our exploration with nouns, and gradually add *verbs* and *adjectives* in our experiment.

First, we take only nouns to construct queries, and run BLUiR to get the accuracy for each subject system. Then we add verbs, to see if doing so improves the result; if so, then verbs are also important. To investigate the relative importance of nouns and verbs, we systematically assign them different weights, w_n and w_v respectively, while constructing queries. More specifically, we set w_n to each multiple of 0.1 from 0.0 to 1.0, inclusive, and w_v to $1 - w_n$. This process gives 10 representations of queries, $(w_n, w_v) = \{(1.0, 0.0), (0.1, 0.9), (0.2, 0.8), ...(0.0, 1.0)\}$. When $w_n = 0$ the nouns are ignored, and when $w_v = 0$ verbs are ignored. We discard (1.0, 0.0) since we already have the results for noun only queries. Then we run BLUiR and get the accuracy for each combination,

to understand the importance of nouns and verbs. For adding adjectives, we fix the best combination of weights for nouns and verbs, according to the Result Quality (RQ) metric defined in Section 8.2.3, and experiment with different weights for adjectives, ranging from 0.0 to 1.0 in 0.1 increments. This process gives us another 10 representations of queries per project,—e.g., for SWT we get

$$(w_n, w_v, w_a) = \{(0.7, 0.3, 0.1), (0.7, 0.3, 0.2), ..(0.7, 0.3, 1.0)\}$$

Note that the sum of the weights for nouns, verbs and adjectives is greater than 1. Then, we run BLUiR and get the accuracy for each query representation. We then select the weight combinations with the best accuracy.

Results

Table 8.3 presents the accuracies for three cases: 1) when we just run BLUiR as is, i.e., using all the terms in the bug reports, ii) when we use only nouns, and iii) the combinations for which we got the best accuracy. Our results show that indeed for many software projects, using only nouns increased accuracy by up to 68% for Recall at Top 1 compared to when we used all terms. However, for some bug reports the accuracy decreased. When we added verbs, accuracy increased as compared to using nouns only, for all software projects. However, there remain bug reports where we lose accuracy as compared to when we use all terms. When we manually investigated those cases, we found that adjectives were also sometimes important. As a result, we found that adding adjectives with a small weight increases the accuracy even further.

Although across the different software projects the weights found in the best combinations are not the same, they are similar, with nouns being by far the most important, and adjectives being the least important. Our final results show that the combination of nouns, verbs, and adjectives with appropriate term weights increases the accuracy considerably, as compared to using all the terms of a bug report as a query.

Table 8.3: Effect of POS (Improvements over All Terms)

Project	w_n,w_v, w_a	**Top 1**	**Top 3**	**Top 5**	**MAP**	**MRR**
	All Terms	37	65	75	0.513	0.582
	(1.0,0.0,0.0)	43	65	70	0.545	611
SWT	Improvement	+16.2%	0.0%	-6.7%	+6.2%	+5.1%
	(0.7,0.3,0.2)	46	64	70	0.555	0.626
	Improvement	+24.3%	-1.5%	-6.7%	+8.1%	+7.7%
	All Terms	25	77	95	0.247	0.297
	(1.0,0.0,0.0)	42	76	94	0.281	0.343
PDE	Improvement	+68.0%	-1.3%	-1.1%	+13.7%	+15.3%
	(0.7,0.3,0.1)	43	81	98	0.294	0.356
	Improvement	+72.0%	+5.2%	+3.2%	+19.0%	+19.6%
	All Terms	97	167	221	0.256	0.310
	(1.0,0.0,0.0)	102	175	206	0.256	0.314
JDT	Improvement	+5.2%	+4.8%	-6.8%	-0.2%	+1.3%
	(0.6,0.4,0.3)	126	195	233	0.295	0.358
	Improvement	+29.9%	+16.8%	+5.4%	+15.1%	+15.4%
	All Terms	517	775	889	0.342	0.430
	(1.0,0.0,0.0)	537	796	894	0.353	0.442
Platform	Improvement	+3.9%	+2.7%	+0.6%	+3.2%	+2.8%
	(0.6,0.4,0.2)	568	838	957	0.371	0.464
	Improvement	+9.9%	+8.1%	+7.6%	+8.6%	+8.2%

8.3.3 Structure Based Term Weighting

Since we observe considerably improved results by weighting terms based on POS, we also wanted to investigate whether various kinds of text such as the bug summary, the bug description, the log, quoted terms, etc. should be assigned different weights. In this study, we investigate the relative importance of all terms, especially those in the bug summary and the bug description. Please note that we do not use any NLP techniques for this experiment.

Methodology

To investigate the relative importance of summary and description terms, we first extract both the bug summary and bug description terms from a given bug report, and, analogous to what is done in Section 8.3.2, systematically assign them different weights, w_s and w_d respectively to construct queries. More specifically, we set w_s to each multiple of 0.1 from 0.0 to 1.0, inclusive, and w_d to $1 - w_s$. This process gives 11 representations of queries, $(w_s, w_d) = \{(1.0, 0.0), (0.1, 0.9), (0.2, 0.8), ...(0.0, 1.0)\}$. When $w_s = 0$ the summary terms are ignored, when $w_d = 0$ the description terms are ignored, and when $w_s = w_d = 0.5$ all terms are weighted equally, which is basically traditional unstructured retrieval. Then we run BLUiR and get the accuracy for each query representation to understand the importance of both kinds of terms.

Results

Table 8.4 presents the accuracies for the combination where we got the best accuracy. We use the Result Quality (RQ) metric defined in Section 8.2.3 to choose the best accuracy. We also calculate the improvement in accuracy of the best weighting over BLUiR.

From the results, we observe that terms from bug summaries are more important than terms from bug descriptions, but we cannot completely ignore the bug descriptions either. Therefore, an appropriate term weighting of the bug summary and the bug description increases bug localization accuracy for most bug reports. From the results, we observe that

Table 8.4: Effect of Summary and Description Terms

Project	(w_s,w_d)	Top 1	Top 3	Top 5	MAP	MRR
	(0.8,0.2)	45	66	72	0.557	0.623
SWT	Improvement	+21.6%	+1.5%	-4.0%	+8.5%	+7.2%
	(0.6,0.4)	27	77	94	0.246	0.294
PDE	Improvement	+8.0%	0.0%	-1.0%	-0.4%	-1.0%
	(0.8,0.2)	111	181	231	0.281	0.340
JDT	Improvement	+14.4%	+8.4%	+4.5%	+9.8%	+9.5%
	(0.7,0.3)	538	803	926	0.354	0.446
Platform	Improvement	+4.1%	+3.6%	+4.2%	+3.7%	+3.9%

different combinations of summary and description weights give better results for different systems. However, the results are consistent in that summary terms are always more important than the description terms when terms need different weighting. There is only one exception, PDE, where both types of terms are equally important. Furthermore, we also observe that in each project Recall at Top 1 file shows the most improvement, which is very beneficial in the context of bug localization.

We have performed another set of experiments by assigning varying weights to a few other kinds of text in bug reports, such as quoted text and log text. However, we have only observed minor differences in the results.

8.3.4 Synonyms

Since bug reporters and developers may not always use the same terms in a bug report and corresponding source code files, respectively, normalizing such terms in source code and bug reports, or expanding bug reports with synonyms could improve the accuracy of bug localization. Query expansion using synonyms has indeed been found to improve the accuracy in the area of code search [79]. In this work, we investigate whether synonyms are really useful to improve the accuracy of bug localization.

Methodology

We use WordNet [79], a useful tool for computational linguistics and natural language processing, to extract synonyms of a noun, verb, or adjective. However, extracting the appropriate set of synonyms of a word is not straightforward, since a single word may have many different meanings, and thus many synonym sets. Therefore, WordNet provides synonyms in the form of various clusters, each representing a similar meaning. WordNet also has an API that can tell us which synonyms are most frequently used for the given word. To limit the number of query variants to consider, we take only the most frequently used synonym set. Previous work has argued that the most popular synonyms in WordNet may not be the best for a software development context [144]. We will consider alternate sources of information in future work.

Once we have a synonym set for a given word, we can expand an occurrence of that word in a query by all the synonyms in the set. However, we just do not simply add all the synonyms into the query, as doing so can completely change the balance of terms within the query. To address this issue, Indri provides the `#syn` operator for specifying synonyms. For example, for the term `incorrect`, with synonym `wrong`, we may formulate a query as `#syn(incorrect wrong)`. Indri also allows assigning a weight to each synonym, e.g., `#wsyn(1.0 incorrect 0.5 wrong)`, i.e., only half as much weight to "wrong" as to "incorrect". Since WordNet requires the POS of the word in order to return synonyms, we have used only nouns, verbs, and adjectives, and assigned the different parts of speech the best weight combination from the experiment in Section 8.3.2.

Results

First, we expanded queries with synonyms using `#syn`, which basically assigns the same weight to all the synonyms. We found that in all cases the accuracy decreased considerably. Then we experimented with `#wsyn`, i.e., by assigning different weights to the original word and its synonyms. We found the best results when the synonyms received half of the weight

Table 8.5: Effect of Synonyms (Improvements over the Best POS-Based Results)

Project	Top 1	Top 3	Top 5	MAP	MRR
SWT	45(-2.2%)	66(+3.1)%	71(+1.4%)	0.553(-0.3%)	0.628(+0.3%)
PDE	42(-2.3%)	80(-1.2%)	97(-1.0%)	0.289(-1.8%)	0.350(-1.5%)
JDT	126(0.0%)	196(+0.5%)	235(+0.9%)	0.294(0.0%)	0.356(0.0%)
Platform	561(-1.2%)	824(-1.7%)	951(-0.6%)	0.370(-0.3%)	0.467(+0.5%)

of the original word. Table 8.5 presents the accuracy for that combination. Since we expanded queries having only nouns, verbs, and adjectives, we compute the improvements over the best POS-based accuracies. From the results in Table 8.5, we observe that even after reducing the weight of synonyms, we do not perceive any significant benefits from word synonyms, unlike the case of code search [79]. Although the use of synonyms improved the accuracy in some cases for SWT and JDT, it decreased the accuracy for PDE and Platform. When we investigated the query-wise results in a more detail, we found that synonyms actually affected the results for many queries. However, the total gain in terms of result improvement is not greater than the total loss in most subject systems. We suspect that since bug reports are long, synonyms may introduce much noise. In code search, developers use only a few precise terms, and thus synonyms can play a better role.

8.3.5 Related Forms of Words

As for synonyms, bug reporters and developers also may use the same word but in different forms (e.g., a different POS or tense) in the bug report and in the corresponding source files. Most importantly, we have observed that such differences are not always eliminated by a stemmer. For example, for the word `went`, most stemmers (e.g., Porter or Krovetz) return `went` itself, not `go`. Some stemmers are also conservative. For example, for a word like `decorations`, a stemmer may return `decoration`, rather than the base form `decorate`. Therefore, we investigate whether eliminating such differences improves accuracy.

Table 8.6: Effect of Related Forms (Improvements over the Best POS-Based Results)

Project	Top 1	Top 3	Top 5	MAP	MRR
SWT	46(0%)	66(+3.1%)	74(+5.7)%	0.567(+2.1%)	0.638(+1.9%)
PDE	43(0%)	84(+3.7%)	98(0.0%)	0.295(0.1%)	0.356(+0.1%)
JDT	126(0%)	194(-0.5%)	237(+1.7%)	0.293(-0.5%)	0.359(-0.3%)
Platform	571(+0.5%)	845(0.8%)	950(-0.7%)	0.368(-0.9%)	0.460(-0.9%)

Methodology

For each word in the query, in order to capture the differences due to related forms of the word, we extract all possible forms of the word due to the variation in in different parts of speech. For this, we use Java API WordNet Searching (JAWS)[45] that provides this functionality. Then, we use these word forms as synonyms to construct a query, and experiment with both unweighted (`#syn`) and weighted (`#wsyn`) versions, as done in Section 8.3.4. Again, we build on the POS-based weighting strategy.

Results

Table 8.6 presents the best accuracies when we assign half the weight of the original word to the related words, and the improvements over the best POS-based results. From the results, we observe that the improvement in accuracy due to the use of related words is small. However, the improvement is more consistent than the improvement we got from the use of synonyms. We see that for almost all subject systems the accuracy increased in terms of Recall at Top 1, Top 3, and Top 5. We believe that the reason is that related forms of words are less likely to introduce noise than synonyms. Therefore, the probability that accuracy will decrease is small. There is then a benefit when reporters and developers actually use different forms of the same word.

[45] `http://lyle.smu.edu/~tspell/jaws/`

8.3.6 Filtering Possible Irrelevant Files

The previous sections have considered improvements to the IR bug localization process by preprocessing queries. In this section, we consider the effect of post-processing the results. Specifically, we investigate the possibility that words found in class and package names can help filter irrelevant results out of the resulting ranked list.

Methodology

We follow a fairly conservative methodology to filter out possibly irrelevant files from the ranked list, since a wrong guess of an irrelevant file (i.e., removal of a relevant file) would hurt the accuracy. After extensive manual investigation into the results of many experiments, we have developed some heuristics to address this issue, depending on the ranking of the files concerned, and class name and package name similarity with the given bug report. Below, we first describe how we compute class name similarity, and then describe the heuristics and process we use to remove possibly irrelevant files.

Class Name Similarity. To compute the class name similarity between a bug report and a class name, we first extract all the terms from the bug report (both summary and description), tokenize them to simplify compound words, and form a set of terms T_r. Furthermore, for each term t in T_r, we stem all the terms and add all the related forms (different POS forms of the same word). In this way, we construct a term set for a given bug report. Second, we tokenize the class name to extract all its tokens based on a CamelCase heuristic, and stem the resulting tokens. Then, we calculate a token similarity score S_t as the portion of the resulting tokens that are present in T_r. For example, if one of the words in a bug report is `format` and we are considering the class name `CodeFormatter`, the token similarity score, S_t is 0.5 (1 out of 2). We compute another similarity score, S_e which is 1 if a class name is exactly present in the bug report, and 0 otherwise. Finally, we average S_t and S_e to compute the total similarity score, which is between 0 and 1. When no class token is present in the bug report, the total similarity score is 0, and when the class name is

present in the bug report as-is (e.g. `CodeFormatter` in the bug report in its entirety), the total similarity score is 1.

Removal Process. For a given bug report, we first get the ranked list of source code files returned by BLUiR. Then, for each file, we consider whether we should keep it or remove it from the ranked list, using the following heuristics.

H1: First, we look at the class similarity score. If a class name gets a similarity score that is 0.25 or higher, we keep the file. Otherwise, we may discard it, depending on H2 and H3. The threshold of 0.25 is based on our experiments.

H2: If any package tokens appear in the bug report, we keep the file. Otherwise we remove the file, unless it is the highest ranked file.

H3: We always keep the highest ranked file, but may change its rank. For this file, if the above two heuristics do not hold, we consider the relationship of the file's original similarity score as compared to the query, and that of the second and third ranked files. If the similarity score of the first file is well ahead of that of the second and third files, we keep the file in the first position. More specifically, let d_1 be the difference between the similarity scores of the first and second files and d_2 be the difference in the similarity scores of the second and third files. Then, we consider that the similarity score of first file is well ahead of that of the second one when $d_1 > 1.25 \times d_2$. If the latter condition does not hold, we move the first ranked file to the second position.

Again, note that we set the thresholds of 0.25 in H1 and 1.25 in H2 based on many experiments and manual investigations. We use the same thresholds for all subject systems.

Results

To understand whether filtering out of irrelevant files indeed increases the accuracy, we first take the best POS-based results that we got in Section 8.3.2. Then, we go through the ranked list for each bug report and remove all possibly irrelevant files (or rerank the top file) based on the heuristics described above. Table 8.7 shows the accuracies for the new ranked lists

Table 8.7: Effect of Filtering Possibly Irrelevant Files (Improvements over Best POS Based Results)

Project	Top 1	Top 3	Top 5	MAP	MRR
SWT	48(+4.3%)	69(+7.8%)	73(+4.3%)	0.538(-3.0%)	0.643(+2.7%)
PDE	47(+9.3%)	82(+1.2%)	99(+1.0%)	0.289(-1.9%)	0.354(-0.5%)
JDT	139(+10.3%)	226(+15.9%)	259(+11.2%)	0.311(+5.6%)	0.381(+6.5%)
Platform	592(+4.2%)	895(+6.8%)	1025(+7.1%)	0.376(+1.3%)	0.478(+3.0%)

of files. From the results, we observe that, for all cases, our approach increased accuracies in terms of Recall at Top 1, Top 3, and Top 5 files. Therefore, the result is very consistent, especially considering that we have used the same set of heuristics and thresholds for all projects. The magnitude of improvement is also not small, considering that it is calculated over the best POS-based results. Therefore, we believe that guessing possible irrelevant files based on class and package names and removing them may be an effective means to increase accuracy.

It should be noted that for some cases MAP decreases due to the removal of files. We believe this is expected and does not hurt the practical usability of a bug localization tool, as we discussed in Section 8.2.3. For example, if a bug report involves three faulty files, which are ranked as 3rd, 10th, and 50th, and if our algorithm mistakenly removes the file in the 50th position, the MAP value for that ranking would decrease. However, this decrease does not hurt the practical usability since developers probably never go to the 50th position to investigate a file. On the other hand, if we mistakenly remove a file from the Top 5, the impact would be high. Our overall accuracy improvements in Recall at Top 1, Top 3, and Top 5 show that our algorithm is beneficial more often than it makes such mistakes.

8.4 An Application: BLUiR+

In the previous section, we have explored whether various techniques based on NLP can potentially increase the accuracy of IR-based bug localization. We have observed that POS-based term weighting, summary and description based term weighting, and removing possibly irrelevant files can considerably improve the accuracy of state-of-the-art, BLUiR. On the other hand, taking into account synonyms did not increase accuracy, while unifying related forms of words only increased accuracy slightly.

So far, for each of these techniques, we have only observed its individual effect or its effect combined with POS-based weighting. However, the improvements from the individual techniques could overlap with each other, in which case the total improvement would not be the sum of the individual improvements when the techniques are applied together. Alternatively, several techniques could synergize and provide even greater improvements when combined. To understand the joint effect of these techniques, and their potential for improving the state-of-the-art, we have combined the findings from POS-based term weighting, summary and description based term weighting, and filtering of possibly irrelevant files into BLUiR. We call the resulting tool BLUiR+.

8.4.1 Methodology

To combine the techniques, we first need to construct a query with an appropriate term weighting. Our experiments suggested five possible term weights: summary weight (w_s), description weight (w_d), noun weight (w_n), verb weight (w_v), and adjective weight (w_a). Since our experiments with these weights reported in Tables 8.3 and 8.4 were exploratory, we presented the results in terms of the best weight combination for each software project. To incorporate our findings into a generic tool, however, we need to select a common weight set for all subject systems. Following a simple strategy, for each weight (w), we just take the average of the best values obtained for the four subject systems as shown in Tables 8.3 and 8.4. In this way, we set w_s, w_d, w_n, w_v, w_a to 0.7, 0.3, 0.6, 0.4, 0.2, respectively.

To construct queries from bug reports, we first remove all the terms whose POS is other than noun, verb, or adjective. Then, for each remaining term, we assign two kinds of weights: i) one from w_s and w_d, and ii) one from w_n, w_v, and w_a. To get an aggregated weight for each term, we simply sum the two weights. We leave the design of a more sophisticated strategy for choosing and combining weights to a future work.

BLUiR supports two kinds of retrieval: i) traditional or unstructured, and ii) structured. We can extend either approach with the new weights.

8.4.2 Results

Table 8.8 presents the results for unstructured retrieval. The results show that BLUiR+ improves the accuracy of BLUiR in terms of all metrics except Recall at Top 5 for SWT. The overall accuracy improvements are 23%, 18%, and 14% for Recall at Top 1, Top 3, and Top 5, as compared to the BLUiR results for the same metric. In SWT, Recall at Top 5 files decreases. When we investigated, we found that the bug reports of SWT are already very good and have the highest accuracies among the four subject systems. Therefore, BLUiR+ does not improve the accuracy for Recall at Top 5. However, it does improve all the other metrics for SWT.

Table 8.9 presents the results for structured retrieval. The results again show that BLUiR+ improves the accuracy of structured retrieval of BLUiR for all subject systems except SWT. As we discussed for unstructured retrieval, SWT bug reports are already very good, and thus get little benefit from BLUiR+. The overall accuracy improvements with BLUiR+ are 7%, 10%, and 9% for Recall at Top 1, Top 3, and Top 5 of corresponding BLUiR metric respectively. These accuracy improvements are less than the corresponding improvements for unstructured retrieval. However, structured retrieval has been found to already be highly effective as a core of IR-based bug localization [127]. Therefore, 9% improvement for Recall at Top 5 is a good improvement. Finally, while the Recall at Top 1 with BLUiR+ remains far below the theoretical maximum (Table 8.2), being around 50% of

Table 8.8: Improvement in BLUiR+ over BLUiR for Unstructured Retrieval

Project	Tool	Top 1	Top 3	Top 5	MAP	MRR
SWT	BLUiR	37	65	75	0.513	0.582
	BLUiR+	46	68	72	0.526	0.633
	Improvement	+24.3%	+4.6%	-4.0%	+2.5%	+8.9%
PDE	BLUiR	25	77	95	0.247	0.297
	BLUiR+	45	84	96	0.289	0.356
	Improvement	+80.0%	+9.1%	+1.1%	+16.9%	+19.6%
JDT	BLUiR	97	167	221	0.256	0.310
	BLUiR+	144	230	263	0.315	0.386
	Improvement	+48.5%	+37.7%	+19.0%	23.0%	24.4%
Platform	BLUiR	517	775	889	0.342	0.429
	BLUiR+	595	899	1025	0.376	0.480
	Improvement	+15.1%	+16.0%	+15.3%	+10.0%	+11.7%
	Overall Impr.	+22.8%	+18.2%	+13.8%	+12.9%	+14.8%

the theoretical maximum except in the case of SWT which reaches 74%, the Recall at Top 5 with BLUiR+ reaches, for example, 81% in the case of Platform, which has the largest number of bug reports.

8.5 Threats to Validity

We now discuss the validity and generalizability of our findings.

Construct Validity. We use two artifacts of a software repository: the source code and the bug reports, which are generally well understood. Our evaluation uses a public dataset of bug reports and source code shared by Zhou at al. [173], enabling fair comparison and reproducible findings. The metrics used for evaluation are standard in IR, and are straightforward to compute.

Internal Validity. To compute the upper bound accuracy of IR-based bug localization, we defined an efficient but approximate algorithm to find the most effective sub queries. Therefore, the upper bound reported in our study may not be the actual upper bound. We have used this metric only to understand the remaining space for improvement for IR-based bug localization. There is no impact of this metric on the other findings of the

Table 8.9: Improvement in BLUiR+ over BLUiR for Structured Retrieval

Project	Tool	Top 1	Top 3	Top 5	MAP	MRR
SWT	BLUiR	53	68	76	0.595	0.687
	BLUiR+	55	70	75	0.581	0.699
	Improvement	+3.8%	+2.9%	-1.3%	-2.4	+1.6%
PDE	BLUiR	53	85	100	0.312	0.388
	BLUiR+	61	89	104	0.337	0.417
	Improvement	+15.1%	+4.7%	+4.0%	+7.9%	+7.6%
JDT	BLUiR	142	210	254	0.320	0.400
	BLUiR+	165	252	284	0.351	0.438
	Improvement	+16.2%	+20.0%	11.8%	+9.5%	+9.5%
Platform	BLUiR	612	878	974	0.362	0.454
	BLUiR+	641	953	1064	0.400	0.514
	Improvement	+4.7%	+8.5%	+9.2%	+10.5%	+13.2%
	Overall Impr.	+7.2%	+9.9%	+8.8%	+9.6%	+11.5%

study.

Currently, no available tool provides 100% accurate POS information. Therefore, there may be some misclassifications in the POS-based experiments. To minimize this threat, we have used the POS tagger from the Stanford NLP toolkit, which is one of the best POS taggers currently available.

We have used the WordNet database to determine synonyms and different forms of parts-of-speech for a word. WordNet, however, is derived from natural language sources, and thus it may not contain all words that are specific to a software development context. If a word is not available in WordNet, we do not get its synonyms or related word information. Furthermore, to remove the noise from synonyms we have considered only the most frequently used synonym set. This strategy may again miss relevant information.

Another threat to internal validity is the potential faults in our implementations as well as in the used libraries and frameworks. To reduce this threat, we used mature libraries and frameworks that have been widely used in various software engineering and information retrieval applications.

External Validity. We have used only four subject systems in our experiment, and

all of them are open source projects. Although, they are widely used projects, our findings may not be generalizable to other open source projects or closed source projects. This risk of insufficient generalization could be mitigated by expanding the benchmark to include more subject systems (both open source and closed source). We will explore this in our future work.

8.6 Related Work

The literature on applications of information retrieval in software engineering is enormous. IR-based bug localization started from a very closely related field of research called feature or concept location [78, 93, 111], which is also one of the most common applications of IR to software engineering. We focus only on recent related work on IR-based bug localization and its improvement.

8.6.1 IR-based Bug Localization

Researchers have proposed a number of retrieval techniques and empirically evaluated their accuracy for bug localization.

Lukins et al. [83] use the Latent Dirichlet Allocation (LDA), a generative statistical model widely used for topic modeling, for bug localization. Rao et al. [115] compare a number of techniques such as the Unigram Model (UM), the Vector Space Model (VSM), the Latent Semantic Analysis Model (LSA), the Latent Dirichlet Allocation (LDA), the Cluster Based Document Model (CBDM), and various combinations to investigate their relative performance for bug localization. Based on their evaluation, they concluded that sophisticated models such as LSA, LDA, or CBDM are not necessarily better than simpler models such as UM or VSM. We focus mainly on variants involving NLP-based techniques.

8.6.2 Adapting IR Models for Bug Localization

Nguyen et al. [100] proposed a topic model called BugScout for bug localization, which basically extends the LDA model. Instead of using LDA directly, BugScout infers a topic model for the bug report and for each of the source files, and then correlates bug reports and buggy files via shared topics. The rationale behind this design is that in addition to the topics specific to the report, the contents of a bug report must also describe the occurrence of the bug(s). Based on an empirical evaluation on seven large-scale systems, they reported that their approach can localize 45% of bug reports in the Top 10 files.

Zhou et al. [173] proposed BugLocator, which extends the traditional VSM by combining a sophisticated TF.IDF formulation, and a modeling heuristic for file length so that the retrieval model does not favor small files during the ranking. Based on an experiment with more than 3400 bug reports from four projects, they showed that the revised VSM outperforms traditional VSM.

In our previous work [127], we further improved VSM by incorporating program structure into the retrieval model. Our key insight was that the importance of various kinds of terms in source code, such as class names, method names or variable names, are not the same. For example, developers may use more meaningful terms in a class name than in a simple variable name. We leverage this observation in the structured retrieval model of BLUiR. Based on an empirical evaluation with Zhou et al.'s dataset [173], we showed that BLUiR outperforms previous IR models for bug localization.

Recently Wang et al. [153] have proposed an approach to compose 15 vector space models, each with a different TF.IDF weighting scheme. Then, they defined the search space of possible compositions and adapted a genetic algorithm to heuristically find a near optimal solution. Using Zhou et al.'s dataset [173], they showed that this approach improved on the state-of-the-art by 8%. In this work, we have also investigated the opportunities to improve the core of IR-based bug localization, based on NLP, and found that an appropriate term weighting and improved the core state-of-the-art by 9%.

8.6.3 Integrating repository information

Sisman and Kak [138] showed how defect histories and modification histories of a software project can be effectively used to improve the accuracy of bug localization. To this end, first they estimate a prior probability distribution for defect proneness associated with the files in a given version of the project. Then, these prior probabilities are used in an IR framework to determine the posterior probability of a file being the cause of a bug. By incorporating a temporal decay into the estimation of the prior probabilities, they showed that the improvements in MAP can be as large as 80%.

Zhou et al. [173] and Davies et al. [28] showed how to utilize the similar bug fix information to enhance the accuracy of bug localization. Finally, Wang and Lo [151] incorporated version history, similar report, and program structure together, and got results that improve on those of the best previously available tools. In this paper, we have focused on improving the core accuracy, and thus did not incorporate any additional sources of information that are not directly available in software repositories. However, in the future we are interested in investigating the joint effect of such additional information and our improved core retrieval.

8.6.4 Query reformulation

Researchers have also investigated the possibility of query reformulation to increase the accuracy of IR techniques, although mostly in the context of code search. Haiduc et al. [45] proposed a machine-learning based recommender called Refoqus, which is trained with a sample of queries and relevant results. Then, for a given query, Refoqus automatically recommends a reformulation strategy that improves the query's performance. Hill et al. [49] proposed a NL-based results view, named Conquer, for searching source code for maintenance, which automatically extracts natural language phrases from source code identifiers and shows the results in a hierarchy. Lu et al. [79] proposed an approach that extends a query with synonyms generated from WordNet. The evaluation of most of these approaches has

been focused on concept location and thus involves human written short queries. In this work, we solely focus on IR-based bug localization, which involves long queries, and investigate various NLP-based term weighting techniques to improve the accuracy.

8.7 Summary

Information retrieval (IR) based bug localization shows a great promise to help narrow down the search space for finding bugs in large systems. This can be especially beneficial for developers who are not very familiar with the code base. Thus, IR-based bug retrieval has received a lot of attention in recent years. Although the accuracy of IR-based bug localization techniques has improved considerably, by adapting the IR model specifically for bug localization, recent improvements are mostly based on the integration of new sources of information, such as the set of files fixed for similar or recent bugs. Such distilled information may, however, not be readily available for many real world projects. Thus, in this work we have investigated the possibility of processing the bug report and the resulting ranked list of files using various NLP techniques.

Our results show that bug summary and bug description based term weighting and POS-based term weighting individually improve the results considerably. On the other hand, expanding queries using synonyms, which showed promising results for code search, is not very effective for bug localization. We also observe that class names and package names may provide useful insight into the ranked list of files and may be effectively used to filter out some possibly irrelevant files, thus increasing the accuracy of the results. We have incorporated our findings into an existing state-of-the-art bug localization tool, BLUiR, resulting in the tool BLUiR+. We found that BLUiR+ improves the accuracy of Recall at Top 1, and Top 3, and Top 5 by 23%, 19%, and 14%, respectively, over BLUiR for unstructured retrieval, and 7%, 10%, and 9% for structured retrieval. We believe our results can give researchers more insight about the textual perspective on bug reports for bug localization.

Chapter 9

Conclusion

Bug fixing is a fundamental and critical activity in the development and maintenance of software since buggy behavior may not only cause costly failures but can also affect user's overall experiences with the software product. However, detecting, localizing, and fixing bugs are difficult and expensive tasks, especially for large software systems. Therefore, many bugs remain unfixed for inordinate period of time. In this dissertation, we have built a deeper understanding of current bug fixing processes via mining software repositories, and proposed new techniques to help developers perform two important steps of bug fixing, bug detection and localization, efficiently.

To assist with fast bug detection, we focused on improving regression test prioritization techniques so that regression bugs are exposed early in the testing phased. We have introduced a new approach, REPiR, to address the problem of regression test prioritization by reducing it to a standard IR problem, and developed a prototype, REPiR, that realizes our concept. REPiR does not require any dynamic profiling or static program analysis. We rigorously evaluated REPiR using a dataset consisting of 24 version-pairs from eight projects with both real and seeded regression faults, and compared it with 10 existing RTP strategies. The results show that REPiR is more efficient and outperforms the existing strategies for the majority of the studied subjects. We also show that REPiR can be made oblivious to the

underlying programming language for test-class prioritization, seldom losing accuracy. We believe that this alternative approach to RTP represents a promising and largely unexplored new territory for investigation, providing an opportunity to gain new traction on this old and entrenched problem. Moreover, further gains might be achieved by investigating such IR techniques in conjunction with traditional static and dynamic program analysis, integrating the two disparate approaches, each exploiting complementary and independent forms of evidence regarding RTP.

To assist with bug localization, we focused on improving the IR-based bug localization techniques that identifies the source code files that need to be fixed for a given bug report. We introduced a new approached, BLUiR, which leverages the structured retrieval technique for bug localization. Our key insight is that structured information retrieval based on code constructs, such as class and method names, enables more accurate bug localization. We evaluated BLUiR on four open source Java projects with approximately 3,400 bugs. Our results show that BLUiR achieved better accuracy than the state-of-the-art tool, BugLocator.

Another key limitation of the existing IR-based bug localization studies is that they focus on software written in object-oriented languages, primarily Java. However, much of the most critical and widely used software, such as operating systems, compilers, and programming language runtime environments, is written in C. In this dissertation, we have created a large dataset for C programs that has more than 7,500 bug reports from five popular open source projects. Then, compared the results of IR-based bug localization on C and Java software. Therefore, our results give a richer perspective on the effectiveness of bug localization than that provided by previous studies. The main technical challenge in applying IR-based bug localization to real C projects is to cope with the use of C preprocessor directives and macros. We have shown that this issue can be addressed in a lightweight way using existing technology. Another main lesson of our work is that even though C developers use English words substantially less often in their method and identifier names than

Java developers, IR-based bug localization can still be effective on C code, comparably to Java code. On the other hand, our considered C projects benefit less than our considered Java projects from taking into account information from program structure. This suggests that a greater understanding may be needed of the properties of bug reports and source code to make IR-based bug localization more effective in practice.

Finally, we investigated the possibility of processing the bug report and the resulting ranked list of files using various NLP techniques. Our results show that bug summary and bug description based term weighting and POS-based term weighting individually improve the results considerably. On the other hand, expanding queries using synonyms, which showed promising results for code search, is not very effective for bug localization. We also observe that class names and package names may provide useful insight into the ranked list of files and may be effectively used to filter out some possibly irrelevant files, thus increasing the accuracy of the results. We have incorporated our findings into BLUiR, resulting in the tool BLUiR+. We found that BLUiR+ improves the accuracy of Recall at Top 1, and Top 3, and Top 5 by 23%, 19%, and 14%, respectively, over BLUiR for unstructured retrieval, and 7%, 10%, and 9% for structured retrieval. We believe our results will give researchers and developers more insight about the IR-based detection and localization, and will advance the current practice in these areas.

Bibliography

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. Van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82(11):1780–1792, Nov. 2009.

[2] P. Anbalagan and M. Vouk. On predicting the time taken to correct bug reports in open source projects. In *Proceeding of the International Conference on Software Maintenance*, pages 523–526, 2009.

[3] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *CASCON*, page 23. ACM, 2008.

[4] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceeding of the International Conference on Software Engineering*, pages 361–370. ACM, 2006.

[5] M. J. Arafeen and H. Do. Test case prioritization using requirements-based clustering. In *ICST*, pages 312–321, 2013.

[6] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE*, pages 1–10, 2011.

[7] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The missing links: Bugs and bug-fix commits. In *Proceedings of the Eighteenth ACM SIGSOFT*

International Symposium on Foundations of Software Engineering, FSE '10, pages 97–106. ACM, 2010.

[8] T. Ball. On the limit of control flow analysis for regression test selection. In *ISSTA*, pages 134–142, 1998.

[9] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. A two-step technique for extract class refactoring. In *ASE*, pages 151–154, 2010.

[10] B. Beizer. *Software Testing Techniques (2nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[11] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceeding of the International Symposium on the Foundations of Software Engineering*, pages 308–318. ACM, 2008.

[12] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Duplicate bug reports considered harmful...really? In *Proceeding of the International Conference on Software Maintenance*, pages 337–345, 2008.

[13] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos. Graph-based analysis and prediction for software evolution. In *ICSE*, pages 419–429. IEEE Press, 2012.

[14] P. Bhattacharya and I. Neamtiu. Bug-fix time prediction models: can we do better? In *Proceeding of the Working Conference on Mining Software Repositories*, pages 207–210. ACM, 2011.

[15] D. Binkley, H. Feild, D. Lawrie, and M. Pighin. Increasing diversity: Natural language measures for software fault prediction. *JSS*, 82(11):1793–1803, 2009.

[16] D. Binkley and D. Lawrie. Applications of information retrieval to software development. *ESE (P. Laplante, ed.)*, 2010.

[17] D. Binkley and D. Lawrie. Applications of information retrieval to software maintenance and evolution. *ESE (P. Laplante, ed.)*, 2010.

[18] D. Binkley, D. Lawrie, and C. Uehlinger. Vocabulary normalization improves ir-based concept location. In *ICSM*, pages 588–591, 2012.

[19] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *ESEC/FSE*, pages 121–130. ACM, 2009.

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

[21] G. Bortis and A. van der Hoek. Porchlight: A tag-based approach to bug triaging. In *ICSE*, pages 342–351. IEEE, 2013.

[22] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th international conference on the World Wide Web (WWW)*, pages 107–117, 1998.

[23] F. P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, Apr. 1987.

[24] G. Canfora, M. Ceccarelli, L. Cerulo, and M. Di Penta. How long does a bug survive? an empirical study. In *Proceeding of the International Conference on Software Maintenance*, pages 191–200. IEEE Computer Society, 2011.

[25] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *Metrics*. IEEE Computer Society, 2005.

[26] G. Capobianco, A. De Lucia, R. Oliveto, A. Panichella, and S. Panichella. On the role of the nouns in IR-based traceability recovery. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 148–157. IEEE, 2009.

[27] T.-H. Chen, M. Nagappan, E. Shihab, and A. E. Hassan. An empirical study of dormant bugs. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 82–91. ACM, 2014.

[28] S. Davies, M. Roper, and M. Wood. Using bug report similarity to enhance bug localisation. In *Proceedings of the 2012 19th Working Conference on Reverse Engineering*, WCRE '12, pages 125–134, Washington, DC, USA, 2012. IEEE Computer Society.

[29] S. Deerwester, S. T. Dumais, G. W. Furn, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *JASIS*, 41(6):391–407, 1990.

[30] F. Diaz. Regularizing query-based retrieval scores. *Information Retrieval*, 10(6):531–562, 2007.

[31] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol. Can better identifier splitting techniques help feature location? In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ICPC '11, pages 11–20, Washington, DC, USA, 2011. IEEE Computer Society.

[32] H. Do and G. Rothermel. A controlled experiment assessing test case prioritization techniques via mutation faults. In *ICSM*, pages 411–420, 2005.

[33] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *ISSRE*, pages 113–124, 2004.

[34] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *ISSTA*, pages 102–112, 2000.

[35] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *ICSE*, pages 329–338, 2001.

[36] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *TSE*, 28(2):159–182, 2002.

[37] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 71–80, Washington, DC, USA, 2009. IEEE Computer Society.

[38] H. Fang, T. Tao, and C. Zhai. A formal study of information retrieval heuristics. In *Proc. of the ACM SIGIR conference*, pages 49–56, 2004.

[39] W. Frakes. A case study of a reusable component collection in the information retrieval domain. *JSS*, 72(2), 2004.

[40] G. Gay, S. Haiduc, A. Marcus, and T. Menzies. On the use of relevance feedback in ir-based concept location. In *Proceedings of the IEEE International Conference on Software Maintenance, 2009*, pages 351–360, 2009.

[41] E. Giger, M. Pinzger, and H. Gall. Predicting the fix time of bugs. In *Proceeding of the International Workshop on Recommendation Systems for Software Engineering*, pages 52–56. ACM, 2010.

[42] M. Gligoric, S. Negara, O. Legunsen, and D. Marinov. An empirical evaluation and comparison of manual and automated test selection. In *ASE*, pages 361–372, 2014.

[43] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 495–504. IEEE, 2010.

[44] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Not my bug! and other reasons for software bug report reassignments. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, pages 395–404. ACM, 2011.

[45] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 842–851. IEEE Press, 2013.

[46] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 392–401. IEEE Press, 2013.

[47] E. Hill, S. Rao, and A. Kak. On the use of stemming for concern location and bug localization in java. In *SCAM*, pages 184–193, 2012.

[48] E. Hill, S. Rao, and A. Kak. On the use of stemming for concern location and bug localization in java. In *Proceedings of the12th IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '12, 2012.

[49] E. Hill, M. Roldan-Vega, J. A. Fails, and G. Mallet. Nl-based query refinement and contextualized code search results: A user study. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 34–43. IEEE, 2014.

[50] T. Hofmann. Probabilistic latent semantic indexing. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 50–57. ACM, 1999.

[51] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *ASE*, pages 34–43. ACM, 2007.

[52] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, Dec. 2004.

[53] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through*

Build, Test, and Deployment Automation. Addison-Wesley Professional, 1st edition, 2010.

[54] D. Huo, T. Ding, C. McMillan, and M. Gethers. An empirical study of the effects of expert knowledge on bug reports. In *ICSME*, pages 1–10. IEEE, 2014.

[55] N. Japkowicz. Learning from imbalanced data sets: A comparison of various strategies. In *Proceeding of the AAAi Workshop on Learning from Imbalanced Data Sets*, pages 10–15. AAAi, 2000.

[56] D. Jeffrey and N. Gupta. Test case prioritization using relevant slices. In *COMPSAC*, pages 411–420, 2006.

[57] J. Jeon, W. B. Croft, and J. H. Lee. Finding similar questions in large question and answer archives. In *Proc. of the 14th ACM conference on Information and knowledge management*, pages 84–90, 2005.

[58] B. Jiang and W. Chan. Bypassing code coverage approximation limitations via effective input-based randomized test case prioritization. In *COMPSAC*, pages 190–199. IEEE Computer Society, 2013.

[59] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse. Adaptive random test case prioritization. In *ASE*, pages 233–244. IEEE, 2009.

[60] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. In *ICSM*, pages 92–101, 2001.

[61] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.

[62] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *ICSE*, pages 119–129, 2002.

[63] S. Kim, E. J. Whitehead, Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transaction on Software Engineering*, 34(2):181–196, Mar. 2008.

[64] S. Kim, H. Zhang, R. Wu, and L. Gong. Dealing with noise in defect prediction. In *ICSE*, pages 481–490. IEEE, 2011.

[65] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead. Automatic identification of bug-introducing changes. In *Proceeding of the Automated Software Engineering*, pages 81–90. IEEE, 2006.

[66] P. S. Kochhar, Y. Tian, and D. Lo. Potential biases in bug localization: Do they matter? In *ASE*, pages 803–814. ACM, 2014.

[67] P. S. Kochhar, Y. Tian, and D. Lo. Potential biases in bug localization: Do they matter? In *ASE*, Västerås, Sweden, 2014.

[68] G. Kumaran and J. Allan. Effective and efficient user interaction for long queries. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 11–18. ACM, 2008.

[69] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals. Predicting the severity of a reported bug. In *Proceeding of the Working Conference on Mining Software Repositories*, pages 1–10, 2010.

[70] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck. Comparing mining algorithms for predicting the severity of a reported bug. In *CSMR*, pages 249–258. IEEE, 2011.

[71] A. Lamkanfi, J. Pérez, and S. Demeyer. The eclipse and mozilla defect tracking

dataset: a genuine dataset for mining bug information. In *Proceeding of the Working Conference on Mining Software Repositories*, pages 203–206. IEEE Press, 2013.

[72] D. Lawrie, H. Feild, and D. Binkley. Leveraged quality assessment using information retrieval techniques. In *ICPC*, pages 149–158, 2006.

[73] M. Lease, J. Allan, and W. B. Croft. Regression Rank: Learning to Meet the Opportunity of Descriptive Queries. In *Proceedings of the European Conference on Information Retrieval*, pages 90–101, 2009.

[74] H. K. N. Leung and L. White. Insights into regression testing. In *ICSM*, pages 60–69, 1989.

[75] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *TSE*, 33(4):225–237, 2007.

[76] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 15–26, New York, NY, USA, 2005. ACM.

[77] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *ASE*, pages 234–243, 2007.

[78] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. 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*, ASE '07, pages 234–243, 2007.

[79] M. Lu, X. Sun, S. Wang, D. Lo, and Y. Duan. Query expansion via WordNet for effective code search. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 545–549. IEEE, 2015.

[80] Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi. Extended comprehensive study of association measures for fault localization. *Journal of Software: Evolution and Process*, (26):172–219, 2014.

[81] A. D. Lucia, R. Oliveto, and P. Sgueglia. Incremental approach and user feedbacks: a silver bullet for traceability recovery. In *ICSM*, pages 299–309, 2006.

[82] S. Lukins, N. Kraft, and L. Etzkorn. Bug localization using latent dirichlet allocation. In *WCRE*, pages 155–164, 2010.

[83] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent dirichlet allocation. *Information and Software Technology*, 52(9):972 – 990, 2010.

[84] Z. Ma and J. Zhao. Test case prioritization based on analysis of program structure. In *APSEC*, pages 471–478, 2008.

[85] D. MacKenzie, P. Eggert, and R. Stallman. *Comparing and Merging Files With Gnu Diff and Patch*. Network Theory Ltd, Jan. 2003.

[86] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[87] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[88] A. Marcus and J. Maletic. Identification of high-level concept clones in source code. In *ASE*, pages 107–114, 2001.

[89] A. Marcus and J. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE*, pages 125–135, 2003.

[90] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic. Using data fusion and web mining to support feature location in software. In *ICPC*, pages 14–23, 2010.

[91] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, pages 214–223, Washington, DC, USA, 2004. IEEE Computer Society.

[92] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *WCRE*, pages 214–223, Delft, The Netherlands, 2004.

[93] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, pages 214–223, 2004.

[94] L. Marks, Y. Zou, and A. E. Hassan. Studying the fix-time for bugs in large open source projects. In *Proceeding of the Promise*, pages 11:1–11:8. ACM, 2011.

[95] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel. A static approach to prioritizing junit test cases. *TSE*, 38(6):1258–1275, 2012.

[96] T. Menzies and A. Marcus. Automated severity assessment of software defect reports. In *ICSM*, pages 346–355. IEEE, 2008.

[97] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen. On the use of stack traces to improve text retrieval-based bug localization. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 151–160. IEEE, 2014.

[98] A. Murgia, J. Pérez, S. Demeyer, C. De Roover, C. Scholliers, and V. Jonckers. Predicting bug-fixing time using bug change history. *BENEVOL 2013*, page 20, 2013.

[99] J. Natt och Dag, V. Gervasi, S. Brinkkemper, and B. Regnell. Speeding up requirements management in a product software company: linking customer wishes

to product requirements through linguistic engineering. In *Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International*, pages 283–294, Sept 2004.

[100] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 263–272, 2011.

[101] T. H. Nguyen, B. Adams, and A. E. Hassan. A case study of bias in bug-fix datasets. In *WCRE*, pages 259–268. IEEE, 2010.

[102] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proceeding of the International Conference on Software Engineering*, pages 315–324, 2010.

[103] Y. Padioleau. Parsing C/C++ code without pre-processing. In *International Conference on Compiler Construction (CC'09)*, pages 109–125, York, UK, Mar. 2009.

[104] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys 2008*, pages 247–260, Glasgow, Scotland, Mar. 2008.

[105] L. D. Panjer. Predicting eclipse bug lifetimes. In *Proceeding of the Working Conference on Mining Software Repositories*, pages 29–32. IEEE Computer Society, 2007.

[106] J. Park, M. Kim, B. Ray, and D.-H. Bae. An empirical study of supplementary bug fixes. In *Proceeding of the Working Conference on Mining Software Repositories*, pages 40–49, 2012.

[107] D. E. Perry and C. S. Stieg. Software faults in evolving a large, real-time system: A case study. In *Proceeding of the ESEC*, pages 48–67, 1993.

[108] J. M. Ponte and W. B. Croft. A language modeling approach to information retrieval. In *Proceedings of the 21st annual ACM SIGIR conference*, pages 275–281, 1998.

[109] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Combining probabilistic ranking and latent semantic indexing for feature identification. In *ICPC*, pages 137–148, Athens, Greece, 2006.

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

[111] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Softw. Eng.*, 33(6):420–432, 2007.

[112] D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, ICPC '07, pages 37–48, 2007.

[113] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu. Sample size vs. bias in defect prediction. In *FSE*, pages 147–157. ACM, 2013.

[114] S. Rao and A. Kak. Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In *MSR*, pages 43–52, 2011.

[115] S. Rao and A. Kak. 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*, MSR'11, pages 43–52, 2011.

[116] S. Robertson, H. Zaragoza, and M. Taylor. Simple bm25 extension to multiple weighted fields. In *Proc. of the 13th ACM conference on Information and knowledge management (CIKM)*, pages 42–49, 2004.

[117] S. E. Robertson, S. Walker, and M. Beaulieu. Experimentation as a way of life: Okapi at trec. *Information Processing & Management*, 36(1):95–108, 2000.

[118] J. ROCCHIO. Relevance feedback in information retrieval. *SMART Retrieval System: Experimens in Automatic Document Processing*, 1971.

[119] R. Rosenthal and R. L. Rosnow. *Essentials of behavioral research: Methods and data analysis*, volume 2. McGraw-Hill New York, 1991.

[120] G. Rothermel, M. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *ICSM*, pages 34–43, 1998.

[121] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *TOSEM*, 6(2):173–210, 1997.

[122] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: an empirical study. In *ICSM*, pages 179–188, 1999.

[123] R. K. Saha, S. Khurshid, and D. E. Perry. An empirical study of long lived bugs. In *Proceeding of the IEEE CSMR-18/WCRE-21 Software Evolution Week*, pages 144–153. IEEE, 2014.

[124] R. K. Saha, S. Khurshid, and D. E. Perry. Understanding the triaging and fixing processes of long lived bugs. *Information and Software Technology*, 65:114–128, 2015.

[125] R. K. Saha, J. Lawall, S. Khurshid, and D. E. Perry. On the effectiveness of information retrieval based bug localization for c programs. In *ICSME*, pages 161–170. IEEE, 2014.

[126] R. K. Saha, J. Lawall, S. Khurshid, and D. E. Perry. Are these bugs really normal?

In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 258–268. IEEE Press, 2015.

[127] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *Proceeding of the Automated Software Engineering*, pages 345–355, 2013.

[128] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry. An information retrieval approach for regression test prioritization based on program changes. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 268–279. IEEE Press, 2015.

[129] G. Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[130] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523, 1988.

[131] D. Schuler and A. Zeller. Javalanche: efficient mutation testing for java. In *FSE*, pages 297–298, 2009.

[132] N. Shahmehri, M. Kamkar, and P. Fritzson. Semi-automatic bug localization in software maintenance. In *Proceedings of the International Conference on Software Maintenance*, pages 30–36, 1990.

[133] E. Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, USA, 1983.

[134] E. Shihab, A. Ihara, Y. Kamei, W. Ibrahim, M. Ohira, B. Adams, A. Hassan, and K.-i. Matsumoto. Studying re-opened bugs in open source software. *Empirical Software Engineering*, 18(5):1005–1042, 2013.

[135] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In *Proceeding of the Working Conference on Mining Software Repositories*, pages 2–11, 2013.

[136] A. Singhal. Modern information retrieval: A brief overview. *IEEE Data Engineering Bulletin*, 24(4):35–43, 2001.

[137] A. Singhal, C. Buckley, and M. Mitra. Pivoted document length normalization. In *Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 21–29. ACM, 1996.

[138] B. Sisman and A. Kak. Incorporating version histories in information retrieval based bug localization. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 50–59, 2012.

[139] H. Srikanth, L. Williams, and J. Osborne. System test case prioritization of new and regression test cases. In *ISESE*, pages 64–73, 2005.

[140] T. Strohman, D. Metzler, H. Turtle, and W. B. Croft. Indri: A language model-based search engine for complex queries. In *Proceedings of the International Conference on Intelligent Analysis*, pages 2–6, 2005.

[141] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang. Towards more accurate retrieval of duplicate bug reports. In *ASE*, pages 253–262, 2011.

[142] S. Thomas, H. Hemmati, A. Hassan, and D. Blostein. Static test case prioritization using topic models. *ESE*, 19(1):182–212, 2014.

[143] F. Thung, D. Lo, L. Jiang, Lucia, F. Rahman, and P. Devanbu. When would this bug get reported? In *Proceeding of the International Conference on Software Maintenance*, pages 420–429, 2012.

[144] Y. Tian, D. Lo, and J. L. Lawall. Automated construction of a software-specific word similarity database. In *CSMR-WCRE*, pages 44–53, 2014.

[145] Y. Tian, D. Lo, and C. Sun. Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In *WCRE*, pages 215–224. IEEE, 2012.

[146] Y. Tian, D. Lo, and C. Sun. Drone: Predicting priority of reported bugs by multi-factor analysis. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 200–209. IEEE, 2013.

[147] TIOBE Software. TIOBE programming community index, May 2014. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[148] P. Tonella, P. Avesani, and A. Susi. Using the case-based ranking methodology for test case prioritization. In *ICSM*, pages 123–133, 2006.

[149] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 173–180. Association for Computational Linguistics, 2003.

[150] S. Wang and D. Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *ICPC*, Hyderabad, India, June 2014.

[151] S. Wang and D. Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *22nd International Conference on Program Comprehension (ICPC)*, pages 53–63, 2014.

[152] S. Wang, D. Lo, and J. Lawall. Compositional vector space models for improved bug localization. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 171–180. IEEE, 2014.

[153] S. Wang, D. Lo, and J. Lawall. Compositional vector space models for improved bug localization. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 171–180, Sept 2014.

[154] S. Wang, D. Lo, Z. Xing, and L. Jiang. Concern localization using information retrieval: An empirical study on Linux kernel. In *WCRE*, pages 92–96, Limerick, Ireland, Oct. 2011.

[155] X. Wei and W. B. Croft. Lda-based document models for ad-hoc retrieval. In *ICRDIRl*, pages 178–185, Seattle, WA, 2006.

[156] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Proceeding of the Working Conference on Mining Software Repositories*, pages 1–8, 2007.

[157] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 181–190. IEEE, 2014.

[158] W. Wong, J. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *ISSRE*, pages 230–238, 1997.

[159] H. K. Wright, M. Kim, and D. E. Perry. Validity concerns in software engineering research. In *Proc. of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 411–414. ACM, 2010.

[160] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 15–25. ACM, 2011.

[161] X. Yi and J. Allan. A comparative study of utilizing topic models for information retrieval. In *Proceedings of the 31st European Conference on Information Retrieval (ECIR)*, pages 29–41. Springer-Verlag, 2009.

[162] S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *ISSTA*, pages 201–212, 2009.

[163] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, Feb. 2002.

[164] C. Zhai. Notes on the lemur tfidf model (unpublished work). Technical report, Carnegie Mellon University, 2001.

[165] F. Zhang, F. Khomh, Y. Zou, and A. E. Hassan. An empirical study on factors impacting bug fixing time. In *Proceeding of the International Conference on Software Maintenance*, pages 225–234. IEEE Computer Society, 2012.

[166] H. Zhang, L. Gong, and S. Versteeg. Predicting bug-fixing time: an empirical study of commercial software projects. In *Proceeding of the International Conference on Software Engineering*, pages 1042–1051. IEEE Press, 2013.

[167] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *ICSE*, pages 192–201, 2013.

[168] L. Zhang, M. Kim, and S. Khurshid. Localizing failure-inducing program edits based on spectrum information. In *ICSM*, pages 23–32, 2011.

[169] L. Zhang, M. Kim, and S. Khurshid. Faulttracer: a spectrum-based approach to localizing failure-inducing program edits. *Journal of Software: Evolution and Process*, 25(12):1357–1383, 2013.

[170] L. Zhang, L. Zhang, and S. Khurshid. Injecting mechanical faults to localize developer faults for evolving software. In *OOPSLA*, pages 765–784, 2013.

[171] L. Zhang, J. Zhou, D. Hao, L. Zhang, and H. Mei. Prioritizing JUnit test cases in absence of coverage information. In *ICSM*, pages 19–28, 2009.

[172] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, AADEBUG'05, pages 33–42, New York, NY, USA, 2005. ACM.

[173] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In *Proceeding of the International Conference on Software Engineering*, pages 14–24, 2012.