STUDYING THE LIVES OF SOFTWARE BUGS

submitted to the Department of Computer and Information Sciences,
University of Strathclyde Glasgow
for the degree of Doctor of Philosophy

by
Steven Davies
2014

DECLARATION

This thesis is the result of the author's original research. It has been composed by the author and has not been previously submitted for examination which has led to the award of a degree.

Signed: _____ Date: _____

ABSTRACT

For as long as people have made software, they have made mistakes in that software. Software bugs are widespread, and the maintenance required to fix them has a major impact on the cost of software and how developers' time is spent. Reducing this maintenance time would lower the cost of software and allow for developers to spend more time on new features, improving the software for end-users.

Bugs are hugely diverse and have a complex life cycle. This makes them difficult to study, and research is often carried out on synthetic bugs or toy programs. However, a better understanding of the bug life cycle would greatly aid in developing tools to reduce the time spent on maintenance. This thesis will study the life cycle of bugs, and develop such an understanding. Overall, this thesis examines over 3000 real bugs, from real projects, concentrating on three of the most important points in the life cycle: origin, reporting and fix.

Firstly, two existing techniques are compared for discovering the origin of a bug. A number of improvements are evaluated, and the most effective approach is found to be combining the techniques. Furthermore, the behaviour of developers is found to have a major impact on the accuracy of the techniques.

Secondly, a large number of bugs are analysed to determine what information is provided when users report bugs. For most bugs, much important information is missing, or inaccurate. Most importantly, there appears to be a considerable gap between what users provide and what developers actually want.

Finally, an evaluation is carried out on a number of novel alterations to techniques used to determine the location of bug fixes. Compared to existing techniques, these alterations successfully increase the number of bugs which can be usefully localised, aiding developers in removing the bugs.

PUBLICATIONS

Portions of this thesis have previously been published in the following papers:

- Steven Davies, "Diagnosing the Causes of Bugs from Bug Reports", Doctoral Symposium, European Conference on Object-Oriented Programming, 2011 (Reviewed, but not formally published)
- Steven Davies, Marc Roper and Murray Wood, "A preliminary evaluation of text-based and dependency-based techniques for determining the origins of bugs", in Proceedings of the Working Conference on Reverse Engineering, 2011
- Steven Davies, Marc Roper and Murray Wood, "Using bug report similarity to enhance bug localisation", in Proceedings of the Working Conference on Reverse Engineering, 2012
- Steven Davies and Marc Roper, "Bug localisation through diverse sources of information", in Proceedings of the International Workshop on Program Debugging, 2013
- Steven Davies, Marc Roper and Murray Wood, "Comparing text-based and dependence-based approaches for determining the origins of bugs", in Journal of Software: Evolution and Process **26** (2014), no. 1

ACKNOWLEDGEMENTS

First and foremost, I'd like to acknowledge the support of my supervisor Marc Roper for his constant efforts and encouragement throughout the development of this thesis, and for convincing me to undertake a PhD in the first place. I would also like to thank my second supervisor Murray Wood for all his help and support over the course of my PhD. The support staff in the department have always been helpful over my time in the department, and I extend my gratitude to them. I would particularly like to acknowledge all the anonymous reviewers whose feedback guided the papers the thesis is based on, and the other researchers whose conversations and questions about my work have helped to shape it. Lastly, I would especially like to thank all those developers and users who contributed to the open-source software that this thesis relies on, both as subject matter and to assist with analysis.

This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/P505747/1].

This thesis would not have been possible if it hadn't been for all the other people reminding me there are other things in life than research. All the students in the department, for all the pizza and drinks we shared. The GIST, teaching me to make my research understandable. PGRS, giving me a welcome place to vent to other researchers. My friends, keeping me sane and taking my mind off things. Felicity, whose love has done so much more for me than this small acknowledgement could ever express. My sister and parents, for everything.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

For many developers, it can often feel like they spend all their time fixing bugs in existing software. This can be hugely frustrating when they would rather be working on new challenges, and providing users with new features and new software. A 2002 survey by the National Institute of Standards & Technology [Nat02] estimated that the time spent on debugging and correcting errors in the US is equivalent to 300,000 full-time computer programmers and software engineers; around a quarter of the total number employed in the US. Other estimates vary, but for some projects the amount of time spent on maintenance can reach up to 50% [SLVA97, LVD06], and even as high as 75% [Gui83]. This has real economic cost; the NIST survey estimated the annual costs of such errors for the US economy to be up to $59.5 billion, borne by both developers and users. Reducing the amount of time that is required to be spent on maintenance would allow this time to be used instead for new features, providing more value for the software end-users (not to mention improving the sanity of developers).

Innumerable researchers have sought to find ways to reduce the burden of software bugs, but bugs are complex, and involve many interactions with a variety of people. Bugs can be introduced in one version and discovered immediately, or lay dormant for many releases. They can arise from a simple small change, or only through a complex interaction of changes. They can affect thousands of users, or only occur for one specific setup. The information reported about them can be incomplete or inaccurate. Sometimes what appears to be one bug can have multiple causes, while something that has very different effects has only one root cause. Bug fixes can be small and targeted, or require widespread changes to the code, and often multiple attempts have to be made to fix a bug before it's finally squashed.

The complexity described here can often make it difficult to develop tools and techniques to assist with bug-fixing. Often, research may be carried out on synthetic bugs, bugs that have been introduced by the researchers themselves, and on small or simple systems. While this research has many benefits, it is not always representative of bugs in the real world. To properly understand real-world bugs, we have to understand their life cycle: how and when are they created, reported and fixed? In this thesis I intend to study the life cycle of bugs, detailing a number of studies into real bugs, drawn from real projects. In particular, the thesis will focus on three critical points spanning the entire life of a bug:

Origin: How do we identify when bugs were introduced?
Discovery: How do users report bugs and what information do they provide?
Fix: How can we use information in the bug report and in the history of the project to help developers find and fix bugs?

These studies will help to develop a fuller knowledge about the life cycle of real bugs as they exist in the wild. Through this, this thesis ultimately aims to further existing techniques for assisting developers, to reduce the time and money spent on fixing bugs.

1.1 Contributions

This thesis makes the following contributions:

- A comparison of techniques used for recovering the original source of bugs, through manual simulation on 166 bugs from 2 open-source[1] projects
- An analysis of the quantity and quality of information provided by users when reporting bugs, based on a study of 1600 bugs from 4 projects
- An extensible method for enhancing existing techniques for automated bug localisation using several additional novel sources of information:
  - Similar bugs to the bug being investigated
  - Coverage information from the project's test suite
  - Stack traces contained in the bug report
- Two evaluations of this technique, on 1586 bugs across 7 open-source projects
- The identification of a number of real-world bug examples and their features, and the effects these features can have on bug-related tools

1.2 Structure

As Chapter 2 will show, there are in essence two main ways to reduce the time and money spent on bug-fixing and maintenance: either reduce the amount of bugs which are introduced into software in the first place, or reduce the amount of time it takes developers to find and fix bugs after they have been reported. Both of these can be aided by examining some of the most pertinent points in the bug life cycle: when they are created, when they are discovered and when they are fixed. This thesis will examine each of these points in turn.

First, Chapter 3 will compare two techniques for uncovering the initial causes of bugs from the changes made to fix them. Next, Chapter 4 will examine the quality and quantity of information provided by reporters when reporting bugs, by evaluating 1600 bugs across 4 projects. Thirdly, Chapter 5 will then look at techniques used to help developers localise a bug from a bug report, and evaluate a number of novel techniques to enhance existing approaches.

Chapter 6 will then conclude the thesis by examining further improvements that could be made to reduce the burden of bugs on software development.

[1] In very general terms, open-source projects are those in which the source code is available for anyone to access and modify, as opposed to proprietary projects where the source is not available to external parties

2 BUG LIFE CYCLES

Despite the large costs associated with bugs in the software industry, and their long history, the nature of bugs is still complex and often poorly understood. How bugs are created, how users deal with them and how they can be prevented or fixed are large and nebulous questions. Additionally, not only are the life cycles of bugs complex and difficult to examine, but the entire concept of a 'bug' has been defined to mean many different things throughout the industry and the research literature.

This chapter will explain why there is value in studying the life cycle of bugs, and how this knowledge could be used to reduce the costs of bugs. First, it will introduce the background on what bugs are in the context of this thesis, and how they are related to common modern software development processes. It will also detail some of the general techniques that will be used throughout the thesis to recover information about bugs and to examine their life cycles. Finally, it will place this work in context by examining some of the existing research on bugs and their life cycle. As befits the size and scope of the problems caused by bugs, a vast amount of research has been carried out on the topic. This chapter will not attempt to examine all of this material, but will instead focus on work of historical significance or with particular relation to the thesis. Relevant research will be explored in more depth in each of Chapters 3 to 5.

2.1 What is a bug?

The term bug is used somewhat ambiguously throughout the software industry and this is also true of related terms such as defect, failure and fault. The Institute of Electrical and Electronics Engineers (IEEE) define *defect* as "a problem which, if not corrected, could cause an application to either fail or to produce incorrect results" [ISO10]. A *fault* then is the "manifestation of an error in software", and a *failure* is a system not performing a required function. From these definitions, the relationship is fairly clear. Failures are caused by defects – which could be in the software, configuration or external environment – or by faults. However, defects and faults do not necessarily have to lead to failures: the conditions required for them to cause a failure may never occur in practice, for example. Unfortunately, these definitions are not universally followed; the standard also states that *bug* is a synonym for fault, but that defect can also be used as a synonym for either fault or failure.

In research papers and in industry, the terms are often used inconsistently, and depend on the context. In a tool such as FindBugs [Fin] for example, the term bug is used to mean a potential defect in code. The tool highlights these to developers with the intention that these defects are

fixed before the software is released (either to customers or to acceptance testing), and some research follows this convention. By the earlier definition, these defects then have not caused any failures, because they have not been released. A tool like Bugzilla [Buga], however, is most frequently (although not exclusively) used to track software problems that have been reported by users in released software. Most commonly these problems take the form of failures. Much research then correspondingly follows this convention, using the term bug to mean post-release defect, an error discovered after software has been released.

Throughout this thesis, unless explicitly stated otherwise, the term bug will be used to refer to an error that has occurred in software after some form of release, and which has been reported in a system such as Bugzilla. This definition is broad: essentially a bug can be defined as anywhere where the behaviour of a system differs from users' or developers' expectations. This is not necessarily the definition given by the IEEE, or in some existing research, but it is the definition commonly intended by developers and users. A bug could therefore be a failure as defined above, but also incorrect output, or even features that need to be developed. This is different from a bug as defined by a tool such as FindBugs, where a bug is essentially a violation of some common development pattern such as a variable that might not always be initialised or a method failing to close resources it has opened[1]. While the bugs that are the focus of this thesis can vary in severity or priority, the overriding thing they have in common is that they have had an effect on, and therefore some cost for, users.

2.2 Bug tracking systems

In most modern development processes, particularly in open-source projects, bugs are recorded in a bug tracking system (BTS). Many such systems exist, but some of the most common BTSs include systems such as Bugzilla or JIRA [Jir]. These are systems which allow bug reporters (who may be end users, support staff, testers or developers themselves) to report details of bugs that they have experienced while using a particular application. Figure 2.1 shows an example of a bug from the BTS of Eclipse [Ecl], a large and widely-used open-source project. It contains a textual descriptions of the bug, details about the system setup and attachments to help explain and reproduce the bug, such as sample code or screenshots. All of this information for a single bug forms a *bug report*. As discussed, this thesis will use the term bug to mean anything which has been reported in a bug report.

BTSs then allow bugs to be assigned to a particular developer, whose job it will be to investigate and fix the bug (or resolve the bug in some other way, such as closing it as invalid). Obviously, the information provided in the bug report is useful for the developers to fix the bug. However, the bug report alone is not always sufficient to locate the cause of a bug. Often, information which is required by the developer may be missing from the report, or the information which is provided may be inaccurate, unstructured or difficult to understand. Additionally, users and developers will sometimes use different terminology, which can lead to misunderstandings or confusion. To alleviate these problems, the BTS also allows for dialogue between the developer and reporter, as well as any other interested users, so that the developer can clarify any issues or request additional information. An example of such dialogue is also shown in

[1] It is of course possible that a violation like this is the *cause* of a bug reported by a user

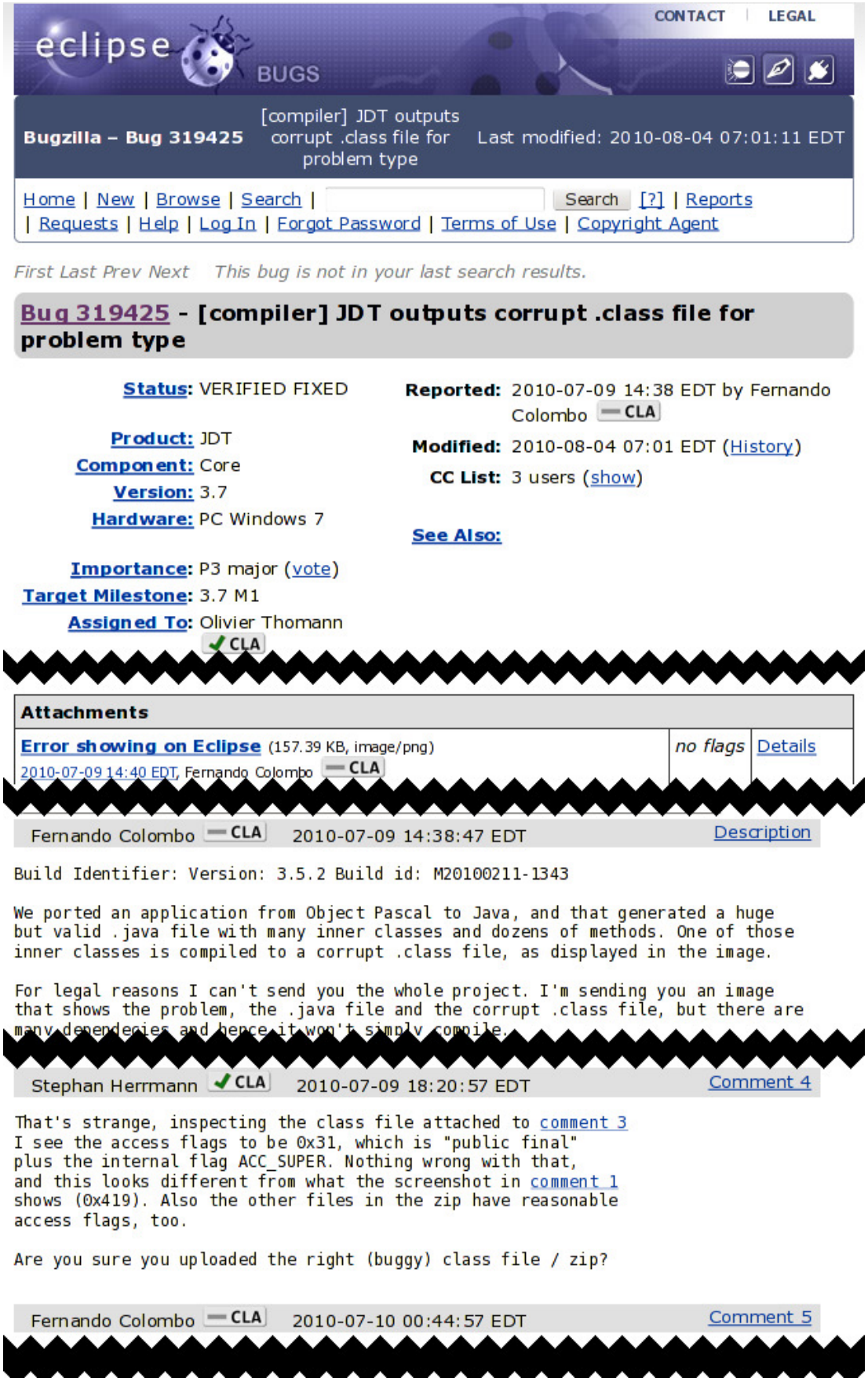


Figure 2.1: Eclipse Bug 319425 (for readability, some sections have been omitted)

Figure 2.1. However, this communication obviously takes time; Breu et al. [BPSZ10] found that a considerable amount of time was spent by developers prompting users for more information, due to it being missing or inaccurate.

Once created, bug reports typically move through a number of stages, known as the bug report life cycle. The exact order and names are dependent on the BTS, and can be configured by the developers. However, the default Bugzilla phases [Bugb], shown in Figure 2.2, are commonly used. Initially, a bug report by a user will be *Unconfirmed*. It will then be up to a developer to confirm whether the bug does indeed exist, at which point the status will be changed to *Confirmed*. The bug will then be assigned to a particular developer, whose job it will be to investigate and fix the bug (or resolve the bug in some other way). Once the developer has been assigned, the status will be changed to *In Progress*. When the developer has finished with the bug, the status will be changed to *Resolved*. In some cases, another developer or quality assurance engineer may then be required to verify that the bug has indeed been handled correctly, in which case the status will be changed to *Verified*.

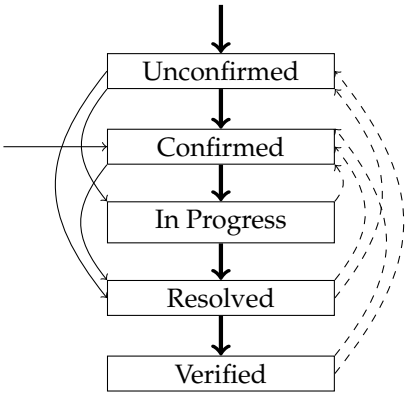


Figure 2.2: Life cycle of a bug report

When they set the status to *Resolved*, the developer will also record the resolution of the bug, depending on the outcome. For example, the bug may be marked as *Fixed* if the developer thinks they have fixed the bug. There are also a number of other resolutions open to a developer, such as *Duplicate*, *Won't Fix*, *Works For Me* or *Invalid*, depending on the developer's reason for not fixing the bug.

The description above is the ideal life cycle of a bug report, shown in thick lines on Figure 2.2, but the reality is generally more complex than that. Koponen [Kop06] studied the life cycle of defects in two open-source projects, Apache [Apa] and Mozilla [Moz], and found evidence of significant numbers of bug reports which transitioned directly from *Unconfirmed* to *Resolved* for example. There are other points where it is possible to skip stages, shown as thinner lines on Figure 2.2, and Koponen even found evidence of bug transitions which Bugzilla theoretically does not allow at all.

It is also possible for bugs to move backwards in the life cycle. Most commonly, this happens if developers temporarily stop working on a bug, but a bug may even be reopened if it is has been found to have been resolved incorrectly. These transitions are shown in dashed lines on Figure 2.2. Again, these are not simply theoretical transitions: Wang and Zhang [WZ12]

found that around 2% of bugs in a particular sub-project of Eclipse were reopened[2] in 2009, and Zimmermann et al. [ZNGM12] also described the problem of reopened bugs in Microsoft Windows, but for confidentiality reasons did not report the actual rate.

It is worth noting that all of the phases described in this section form merely the life cycle of the bug report. The life cycle of the bug itself can be even more complex, and includes at least one more major point: the point at which the bug is first introduced, which must necessarily come before the bug is reported.

The availability of open BTSs has often been cited as an important factor in improving the quality of open-source software [Ray00, MFH02]. This is because they allow absolutely anyone to provide reports of issues and to help debug them, and not just developers and testers of the system. However, there are also a number of problems with such systems, not least the sheer number of bug reports that are received by popular projects. Anvik et al. [AHM05] demonstrated that the large number of bug reports received by major projects took up a considerable proportion of developer time, and that many bug reports were not productive, as they were closed with some resolution other than *Fixed*.

As stated earlier, the information provided in the bug report is useful for the developers to fix the bug, but it is not always sufficient. A survey of developers by Bettenburg et al. [BJS+08] found that the information which developers wanted was often missing from the report, and this has been echoed in various other surveys and studies [HW07, BPZ08, GZNM10]. Another survey by Koru and Tian [KT04] asked developers how often they thought bug reports were mostly complete, and around half answered either 'sometimes' or 'rarely'. When information is provided, it may not actually be accurate, which developers consider to be one of the biggest problems they face [BJS+08, JPZ08]. Breu et al. [BPSZ10] found that a significant proportion of questions asked by developers on bug reports were related to missing or inaccurate information, while Ko and Chilana [KC10] found that the majority of bug reports by normal users of BTSs contributed no useful information.

As well as affecting developers' ability to fix bugs, the quality of bug reports also has an impact on researchers, particularly when attempting to examine the history of a bug. An in-depth study of bugs at Microsoft [AV09], along with a more general survey of employees, found that the history of bugs could not accurately be recreated through electronic sources alone: fields in the bug report were incorrectly recorded, links between the bug report and other data such as source code were missing, and many people other than just the reporter and assigned developer played a part in resolving the bugs. One other problem is that some of the most important information about the bug is stored in free-text fields, rather than in a structured manner. This can make it difficult to process. However, tools exist which can partially extract some of the information: InfoZilla [BPZK08], for example, extracts structured information such as stack traces, lists and patches from the unstructured bug report description.

Clearly, the amount and quality of information provided by users when reporting a bug can have a major influence on the time taken, and therefore the costs paid, to fix the bug. Understanding this important point in the bug life cycle could help to reduce these costs. Unfortunately, while there is much research showing that developers feel bug reports are often incomplete, there has been little research examining bug reports in detail, and enumerating

[2]Moved from *Resolved* or *Verified* to one of the earlier states

which pieces of information are present. Chapter 4 will examine in more detail the information stored in BTSs when bugs are reported, and discuss how it influences the rest of a bug's life cycle.

2.3 Revision control systems

A BTS is useful for showing some of the life cycle of a bug, but, as discussed, the life cycle of a bug extends beyond that of the bug report. Part of the history of a bug is the history of changes within the code: the changes which introduced the bug and the changes which fixed the bug. Thankfully, these changes can often be recovered and examined. This is because, in addition to BTSs, most projects also utilise revision control systems (RCSs) in their development process, which contain this history. RCSs are tools such as CVS [CVS], Subversion [SVN] or Git [Git] which are used to allow multiple developers to work on a project concurrently, while helping to minimise conflicts that occur if changes are made to one part of a system by multiple developers simultaneously. Typically[3], developers check out a copy of the source code from a central repository, make changes on their local machine, and when they have finished a piece of work commit that code back to the repository. If the central version has changed in the meantime, they can update their version, and will have to merge in any changes that conflict with their changes before they can commit.

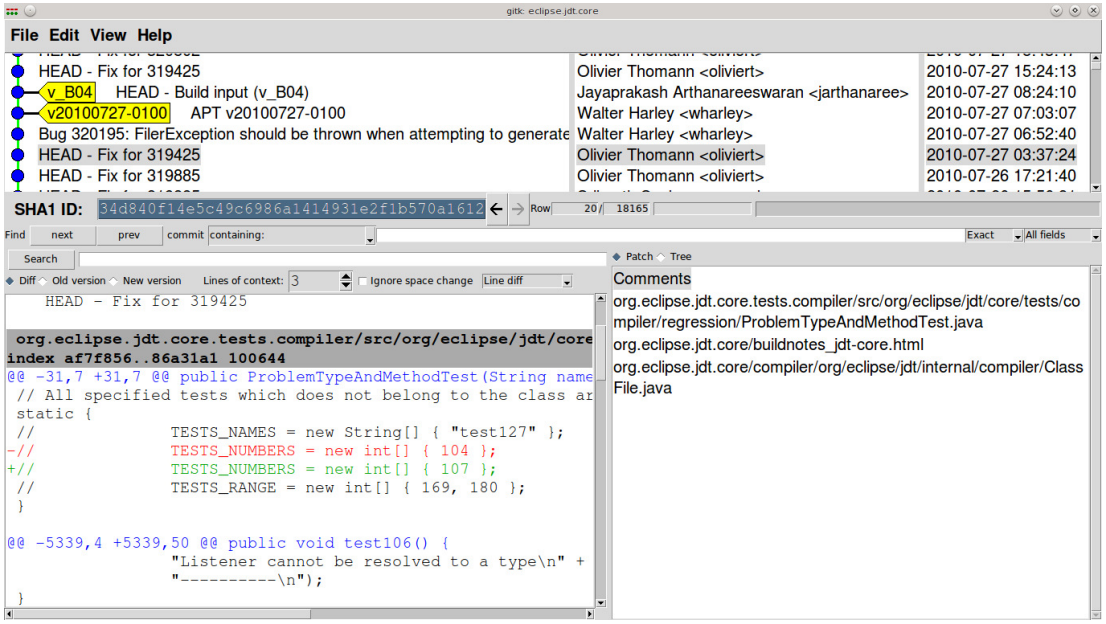


Figure 2.3: Eclipse RCS history

RCSs are also used to store all the past versions of the project. Figure 2.3 shows part of the history of a subproject of Eclipse. Each row in the top-pane is a *commit*[4], an atomic change made by a developer, and includes a comment describing the reason behind the change. The bottom-right of the screen lists the files that were changed in the commit. The bottom-left shows

[3]Terminology varies between systems, and some have a distributed model instead of a central repository, but the principal remains similar

[4]Other RCSs use the similar terms *revision* or *version*. For the most part these can be considered interchangeable, and each will be used throughout this thesis where appropriate.

some details about the commit and repeats the comment, before listing the individual changes made to each file. The use of an RCS allows the organisation to roll back to a previous version of the project if some new changes cause errors, but it also allows them to view the history of particular parts of the project.

The commit message described above is an important feature of most RCSs as it allows developers to describe the rationale behind a change. Some projects even require this: the Linux kernel, for example, requires developers to submit a complete description and justification of the patch [Lin]. This allows other developers to understand the reason something has changed, perhaps a long time after the change has been made. This best practice is not always followed, however; from personal experience it is not uncommon for developers to use poor commit comments or none at all.

Additionally, it is also usually considered best practice for each commit to form a single logical change. The logical change may affect many places in the source code, such as updating a method and all places in the code that call it, as long as all the changes are related. Developers may be expected to avoid, for example, fixing more than one bug in a commit, or making unrelated changes when fixing bugs. Operations like these can make the changes more difficult for other developers to understand. Again however, this advice is not always followed. Jung et al. [JOY09] found that around 8% of changes marked as being related to a bug fix in a small sample of Eclipse bug fixes were in fact not related to the fix. Similarly, Nguyen et al. [NNN13] examined 8 open-source projects and found that 11%–39% of commits which fixed bugs also contained other non-fix changes.

2.4 Mining software repositories

One of the most important advantages of consistent use of BTSs and RCSs is that the information is stored permanently, long after bugs have been fixed. For the developers of the project, this can be useful for reporting purposes and for determining the reasons why particular changes have been made. By combining information from the two systems, this also allows researchers to examine the history of a project and many important stages in the life cycle of an individual bug. Researchers can investigate when the bug was reported, when developers started trying to fix it, view the changes that fixed the bug (and any previous failed attempts to fix it), or trace back to examine the changes that introduced the bug in the first place, and examine how other changes have an effect on the bug.

BTSs and RCSs, together with other collections of data related to software development such as mailing lists, documentation or chat logs, are often collectively known as *software repositories*. An entire field of research has arisen into mining these software repositories in order to discover facts about the software development process and the evolution of software over time. This can be seen, for example, as far back as the laws of software evolution defined by Lehman [Leh80], based on multiple studies of large industrial systems, and more recently in the research presented at specialised conferences such as the Working Conference on Mining Software Repositories [MSR]. The availability of open BTS and RCS repositories for open-source projects, has led to an increase in this sort of research, and comprehensive surveys of research in the area have been published by Kagdi et al. [KCM07] and Hemmati et al. [HNB+13]. The various

repository mining techniques detailed in these surveys allow researchers to combine information from several types of software repositories, and to examine the history and evolution of a project. There are a variety of purposes for such techniques, such as examining trends in software development or modelling developer communication, but there are three main areas which are relevant to this thesis: predicting bug locations and discovering bug origins, bug localisation, and general examination of bug life cycles.

2.5 Predicting bug locations and discovering bug origins

There have been many studies carried out to examine and categorise the causes, effects and fixes of bugs. One of the first of these was by Endres [End75], who analysed the errors discovered during a release of the DOS/VS operating system, finding in particular a massive diversity of bugs in terms of the types of errors and causes. Since then, other studies have been carried out with similar aims [BP84, SC92], and these have also yielded a large number of insights into the large variety of causes and fixes of bugs. While the categorisations and methods used on these studies varied, the intention was always the same: by learning from the changes which introduced bugs, prevent future bugs from ever being introduced in the first place.

Preventing bugs from being introduced is one of the most effective ways to reduce the cost spent on fixing bugs. Indeed, studies have shown that the earlier a problem is fixed in the software life cycle, the cheaper it is to fix, sometimes by multiple orders of magnitude [BP88, BB01]. There are a variety of tools and techniques available aimed at preventing bugs from being introduced into production software. Some of these are development processes, such as software inspections or automated testing. Commercial tools which provide warnings of likely bug locations have been produced, including tools such as FindBugs, Lint [Joh77] and PMD [PMD]. These tools mainly work through static analysis: examining the source code of a project rather than its behaviour at run-time. They are based on searching for code that matches common patterns of bugs. Often the use of the tool has to be tailored for each development environment and project, but they have been found to be cost-effective [APM+07, JCS07].

Similarly, much research has been done on using various metrics to predict faults. Metrics are numerical measurements of particular aspects of interest of source code. The most basic of these are simple statistics, such as the number of lines of code in a method or the number of branches, but many others have also been defined, such as those given by Halstead [Hal77] and McCabe [McC76]. These less straightforward metrics attempt, although not always successfully, to capture the idea of how 'complex' a piece of code is. Other metrics are used to track features of the development process, such as when a method was last changed, or the number of times a method has been changed. A number of surveys have been carried out summarising the use of metrics to predict software faults [CD09, HBB+12, RHTŽ13]. These have found that while metrics can sometimes be useful predictors of software faults, there is often little consistency in which metrics are actually useful, and that this can be heavily dependent on what programs are being analysed, and what is being done with the values of the metrics.

Many of these tools are based simply on common patterns or on static attributes of the source code, and do not consider information about the actual changes that have been made to the code. However, Hall et al. [HBB+12] found that predictors which did take information from

RCSs into account, in conjunction with other information, were generally more effective than those that did not. Shivaji et al. [SJAK09] built a bug predictor, one aspect of which was based on identifying the changes which had introduced bugs in the past. The most impressive aspect of the predictor was its very high precision: for most projects, if the predictor stated there was a bug it was almost always right. This high precision came with a cost of lower recall – many bugs would go unnoticed by the predictor – and this trade-off of precision and recall is very common among many similar tools. Nagappan et al. [NBZ06] also used similar information on changes which introduced bugs, mining the history of projects to build defect predictors based on metric values. Unlike most other work, they concentrated solely on post-release defects, not pre-release. They found that while they could build effective predictors, the most effective set of metrics was different for each project, implying that there is a value in producing such predictors based on project history rather than building generalised tools. Similarly, Williams and Hollingsworth [WH05] built a tool which could check whether the return values of functions were tested before they were used. They then used the checker on multiple versions of a software project, looking for instances where such bugs were fixed, and used the information gained to improve their tool. Through this, they demonstrated how mined information could be useful in general for improving static analysis tools.

What many of these studies have shown is that the performance of tools can be improved by examining previous bugs, and, in particular, by looking at the types of changes which have been responsible for introducing each bug. Examining this point in a bug's life cycle could be incredibly useful for the development of further research tools. Discovering which changes introduced a bug is not always straightforward however, and can involve considerable manual effort, on the part of either the developers or the researchers. As such, the number of examples of bug-introducing changes available to researchers is low. To supplement this, in recent times many researchers have sought techniques to automatically identify the individual code changes that were made to introduce a bug.

The first such approach was developed by Śliwerski et al. [ŚZZ05b]. They used the RCS history of a project, specifically from the *cvs diff* and *cvs annotate* commands, to identify when lines changed in a bug fix had last been altered, assuming that this change was responsible for introducing the bug. Various researchers have since proposed variations on this technique which can make the process more accurate [KZPW06, WS08, JOY09]. An alternative approach, which is based on comparing changes in a program's dependences was proposed by Sinha et al. [SSR10]. These techniques also bear some resemblance to the concept of delta debugging [ZH02], where failing test cases from a later version of code are used repeatedly on earlier versions in order to identify the last version where the test passed.

The identification of the origin of a bug has been put to use in a number of practical tools. HATARI [ŚZZ05a] is a prototype tool, built on the techniques described above, which warns developers about lines of code which are particularly risky to change. These are lines where many changes in the past have led to bugs being introduced. The developer is shown a visual warning of each line's riskiness, and can also view the history of the line, although unfortunately the effectiveness of the tool was not assessed. Similarly, FixCache [KZWZ07] warns developers when a program element has been responsible for causing bugs in the recent past. When a bug is fixed in a system, the origin of the bug is discovered, and the elements involved are stored in a cache. The elements in the cache are highlighted to developers as being risky to change.

They found that for 46%–72% of the faults they examined, the elements involved would have been held in the cache at the time, making the cache an effective list of program elements to prioritise in tests and inspections.

A number of other works also try to predict whether a change that a developer has made is likely to introduce a bug, based on common patterns. Aversano et al. [ACD07] treated the source code as if it was natural language and extracted the text which had changed in each version. They also recorded whether each change introduced a bug, and then trained a variety of machine learning classifiers. Later changes could then be analysed to determine whether the techniques thought they were likely to introduce a bug or not, and warn the developer. Kim et al. [KWZ08] also treated the source code as text, but also added in various software metrics. Both approaches could predict bugs with reasonable accuracy, but still had a lot of room for improvement. Ferzund et al. [FAW09] also used various code metrics, and found that metrics based on the change being made or on the history of the related program element were more successful at predicting bugs than those metrics simply based on the individual element.

These tools illustrate the usefulness of examining the origin of bugs in order to prevent the introduction of future bugs. A discussion and comparison of techniques for identifying bug origins is contained in Chapter 3.

2.6 Bug localisation

Despite years of study and effort, we still cannot produce software which is entirely free of bugs. Regardless of the effectiveness of the bug prediction and prevention techniques described previously, it is not a controversial statement to say we never will be able to produce such error-free software. Unforeseen changes to requirements, and to the environment, coupled with the often unpredictable nature of users and the increasing size and complexity of software mean we can never anticipate every possible problem or prevent every possible bug. However, prevention of bugs is just one way to reduce the cost of maintenance; another way is to reduce the time taken to correct bugs.

The process of fixing a bug report, as detailed previously, has multiple stages. Even once a developer has been assigned to a bug and begun to work on resolving it, there are a number of tasks. One of the tasks that dominates the overall time taken [BNRR08] to fix bugs is bug localisation: identifying where in the code a fix has to be made. This has to happen before the developer can even begin to approach determining what the fix should be or how to apply it.

This act of bug localisation is part of the wider task of feature location, also known as concept assignment, first defined by Biggerstaff et al. [BMW93]. In this context, features or concepts are aspects of the program expressed in human terms e.g. an online shop might have features such as browsing a particular department, buying an item etc. Feature location is the act of assigning such features to particular elements of the source code. There has been a large variety of research conducted on both feature location and bug localisation, and detailed surveys of each have been carried out by Dit et al. [DRGP11] and Wong and Debroy [WD09] respectively. However, while there has been a lot of research, there has not been a corresponding level

of adoption in industrial use, and the surveys suggest that even research evaluations with human subjects are relatively rare. The reasons for the low level of adoption are not clear, and there appears to be a lack of empirical research investigating them. However, many of these techniques require developers to invest a lot of effort up-front in order to locate the cause of a single bug. Anecdotal evidence suggests that this effort may put developers off taking up techniques before they have seen evidence that they will work. Additionally, many of the techniques currently still have fairly low levels of accuracy, meaning evidence that would convince developers to test out a tool is scarce.

By improving the accuracy of these techniques, or by lowering the level of effort required of the developer, we could hasten the adoption of such tools, lowering costs but also providing feedback for researchers to further improve the techniques. One particular class of bug localisation techniques does indeed require much lower effort from a developer. These are techniques based on some form of textual analysis of the software – essentially treating the software as plain text and using techniques from information retrieval (IR). One of the first of these techniques was introduced by Marcus et al. [MSRM04] who used the IR technique of Latent Semantic Indexing (LSI). Here, a corpus is first built of the software, and developers then formulate queries which are evaluated against that corpus, resulting in (hopefully) relevant areas of source code being returned to the developer. While the approach can successfully locate relevant methods for a number of bugs, the performance is far from ideal for actual use by developers, and it also relies on the developer to generate custom queries for each bug. This obviously does not meet our aim of reducing the developer effort. However, similar techniques have been developed where the query is automatically generated from the text of the bug report. For example, Lukins et al. [LKE08] used a manual, but well-defined, process for extracting the query from the bug report, while Nguyen et al. [NNAK+11] used all the text extracted from the bug report, with only some automatic post-processing. These techniques can still be effective even without a developer to formulate a query, and so can be used to reduce the investment of effort required from a developer. Other researchers have also used similar techniques [PMDS05, PGM+06, PMD06, LKE10]. Many of these enhancements have improved the accuracy of earlier techniques, but are still not up to the standard that could be considered necessary.

To improve the accuracy of the techniques, other researchers have attempted to augment them by combining these textual approaches with information from other sources, such as similar bug reports [CEBE08, ZZL12], mailing lists [CEBE08], documentation [ČM03, CEBE08] or RCS comments [CCW+01, ČM03, CC06a]. In these cases, the researchers found that the combination of information from multiple sources was more effective for bug localisation than any individual source on its own.

While there is still a lot of research to be done, the many small improvements that have been made by various researchers suggests that there is still good potential for such bug localisation techniques. Additionally, the potential costs that could be saved by vastly reducing the time taken to fix bugs suggests that such tools are a useful area of research, and a fuller description of some techniques and approaches to improve them is contained in Chapter 5.

2.7 Other studies of bug life cycles

There have been other attempts to study the longer life cycle of bugs, in particular the length of time between the introduction of the bug and its eventual fix. Chou et al. [CYC+01] examined warnings found by twelve automated bug checkers across versions of Linux, although they did not examine user-reported bugs. For all checkers they found the median difference between the introduction and the fix to be around 1.25 years. However, this often varied by the type of warning: those related to potential deadlocks could last significantly longer. The techniques described in Section 2.5 have also been used on two open-source projects, PostgreSQL [Pos] and ArgoUML, to determine that the median time between the introduction of a bug and its eventual fix was around 200 days [KW06]. An investigation of an industrial project using similar techniques found that 52% of bugs were reported in the next release after they were introduced [Ram08].

The origin of a bug, the reporting of the bug, and its eventual removal are obviously three of the most important points in a bug's life cycle, and by studying them we can hope to improve techniques for both bug localisation and bug prediction. There are other important points in a bug life cycle however, and other work in mining software repositories has sought to examine and influence these. D'Ambros et al. [DLP07] used information mined from a BTS to visualise the changes in status that bug reports go through. The intention was to aid developers both in monitoring the overall health of their project and also in identifying the critical components of the system, namely those affected by many bugs. Knab et al. [KFGP09] built a similar tool, showing how often bugs transitioned between the various bug states in the system. This tool could be used by managers to identify weaknesses in the software development process of the organisation.

Anvik et al. [AHM06] used information mined from the source code repositories of Eclipse and Firefox [Fir] to assist with bug triage, the act of deciding whether a bug report should be investigated and by which developer. They achieved precision of over 50%, although with very low recall. They later improved this work, and generalised it to make other recommendations, such as the project component or a list of interested developers who should be notified when the bug is updated [AM11]. Weiß et al. [WPZZ07] used textual features extracted from the titles and descriptions of bugs to produce a predictor that could predict the effort required to fix a new bug, finding the average difference between the predicted time and the actual time to be under four hours. Sandusky et al. [SGR04] studied the relationships between bugs, forming them into *bug report networks* based on formal relationships such as one bug depending on another, or informal relationships such as commenters indicating that bugs looked related. They found that most bugs in their sample were part of one or more bug report networks. Shihab et al. [SIK+12] mined a variety of features to predict the likelihood of a bug being reopened, with reasonable success. They suggested that this information could be used to suggest to developers when a bug might be likely to be reopened, intending that the developers should examine such bugs carefully before confirming them as closed. They also found that the information which was most useful in making predictions varied across different projects, again highlighting the value of building tools which learn from a project's history, rather than attempting to make generic techniques.

Like the act of bug localisation, all of these tools help to reduce the amount of time developers spend on fixing bugs, as well as reducing the delay between a bug being reported and being fixed, which is often a source of frustration for users. They indicate that there is a clear benefit to understanding the life cycle of bugs in order to reduce the costs of bugs on future development.

2.8 Conclusions

This chapter has introduced the life cycle of bugs, and the development process that surrounds them. It has also illustrated the value of studying the life cycle in detail, with the eventual aim of reducing the time developers are required to spend on maintenance, freeing them up for activities that provide more value to the project. From the overall bug life cycle, the chapter has also identified three of the most pertinent points: when the bug is introduced; when the bug is reported; and when the bug is fixed. Existing studies into these points show that better understanding of them has real potential to help alleviate the costs of bugs, and I therefore intend this thesis to study these three points in detail. This information, and the relationships between these stages, can then be used to reduce the time and money spent on bug maintenance, both through learning to prevent bugs in the first place and in speeding up the process of fixing bugs once they have been introduced. Chapter 3 will evaluate techniques for determining the origin of bugs, and determine what can be learned from these origins. Next, Chapter 4 will examine the information provided by users when reporting bugs, and how this relates to techniques for determining bug origins and bug localisation. Finally, Chapter 5 will detail a technique for augmenting existing bug localisation techniques, to improve their performance.

3 DETERMINING THE CAUSES OF BUGS

As Chapter 2 discussed, there have been many tools and techniques developed which attempt to prevent bugs from being introduced into software. This is not surprising, given the suggestion that software developers can spend nearly half their time fixing bugs [LVD06], and the well-documented impact of bugs on software development cost and effort. Often these techniques work by inferring common patterns from code which contains bugs and then finding similar code. However, accurately identifying bug *origins* – the point in the history of the code that a bug was introduced – is particularly challenging. This is because, despite it being the first point in the bug life cycle, it is obviously not the point at which the bug is discovered. Factors such as the length of time between introducing and reporting a bug, and the potentially substantial changes that may have been made to the code in the meantime, complicate attempts to trace back through history to discover the origin of the bug.

Being able to automatically identify which changes actually introduced a bug[1] could be useful in improving the accuracy of any of these bug prevention tools or enabling the introduction of more advanced techniques altogether. Software developers and managers would also stand to benefit from identifying when bugs were introduced, and could use such data to pinpoint potential weaknesses in their processes. Furthermore, identifying such changes also opens possibilities for researchers to more closely examine the nature of changes that can introduce bugs.

In the past several years, two approaches have been proposed that attempt to automatically identify the origin of a bug, based on the changes made to fix it: the text approach and the dependence approach. Both start from the version of code containing the bug fix and progressively examine preceding versions until they find the one that introduced the code responsible for the bug. The approaches differ primarily in how they examine the changes between versions: the text approach [ŚZZ05b] uses only the changes in the text itself, while the dependence approach [SSR10] uses changes in the relationships between control and data in the code. Unfortunately, as yet no implementations of either approach are readily or openly available, which presents a significant challenge to assessing their effectiveness or even their viability. Furthermore, the effort required to create an implementation in both cases is non-trivial, and the dependence approach in particular poses a number of implementation challenges, as discussed in later sections.

As there are no readily available tools, this chapter instead presents the results of an investigation into the accuracy of both approaches and provides insights into their respective strengths

[1] It is also possible for bugs to have existed in a piece of code since it was created. This can still be seen as a change, the creation of code being a change from the previous state where there was no code.

and weaknesses. The aim is to determine the potential value of implementing such systems, as well as the potential problems that would need to be overcome.

3.1 Background and related work

Some of the related work was already covered briefly in Chapter 2, but this section will examine that, and further work, in more detail.

3.1.1 Identifying bug fixes

As discussed previously, two components of modern development are crucial to help identify the origin of a bug: RCSs and BTSs. While these systems are not usually integrated, techniques exist to combine their information [ŚZZ05b]. Unfortunately, but not surprisingly, the introduction of a bug is not generally recorded – developers are hardly likely to record commit comments like "Added new login system. Introduced bug 147, where user accounts can be accidentally deleted through incorrect passwords."

What is often recorded instead is the bug fix, when a developer removes the bug from the system. When fixing bugs, developers will often include the bug ID in their commit comment [ŚZZ05b, BBA+09]. Bugs can therefore often be linked to the commits that fix them by extracting the comments from the RCS and searching for bug IDs. Approaches to identify bug origins have to start from this bug fix and work backwards to identify when that bug was introduced.

Unfortunately, the relationship between bugs and commits is not a simple one, as many studies have shown [ŚZZ05b, BNG+09, JOY09, BPSZ10, NNN13]:

- Bug fixes may be relatively simple and may involve just a line or two in the system and the corresponding commit may involve just one file.
- Alternatively, bug fixes may involve a wider set of far more substantial changes. Consequently, a commit may be made up of one or more changes to one or more files.
- A commit may also involve changes that are unrelated to the bug in question: they may be associated with a different bug, or with no bug at all (for example, a refactoring or enhancement that has been made to the code opportunistically, in addition to the bug the developer was supposed to be fixing).
- A bug report may contain details about more than one bug.
- Developers may make more than one attempt to fix a bug, in which case there will be multiple commits associated with one bug report.
- Perhaps most commonly, a commit may fix a bug, but make no mention of the bug ID at all.

Given this complex relationship, it is obvious that these approaches for linking bugs to their associated fix will not be perfect and there is some evidence that these links are not entirely accurate [BBA+09, NNN13]. However, many other researchers, dating back to Fischer et al. [FPG03], have used this assumption as the basis of their work. This technique has been

developed further by also looking at keywords [ŚZZ05b], linguistic analysis [MCMT10], check-in times [WZKC11] and patches [CKEL11]. However, these techniques are more complex and time-consuming, and it is not clear from the existing research whether the links these techniques discover are more accurate than the simpler technique, and they have not been applied in this work. While there may be some missed or inaccurate links, these should not impact hugely on the overall results of the analysis. Nevertheless, this remains a threat to validity, and future work may be required to investigate whether these additional techniques affect the results.

3.1.2 Text approach

The text approach can be illustrated with an example from Eclipse, Bug 63216. This bug concerned a single file, `NewProgressViewer.java`. As explained previously, an RCS will store every version of a file, allowing us to examine changes in the file over time. Each time a developer commits a change to the file, a new version is stored , identified by increasing the version number[2]. Figure 3.1 shows a small section of code from each of four versions of the file `NewProgressViewer.java`: 1.7, 1.8, 1.38 and 1.39, omitting the intervening versions of the file.

```
light = getColor(BACKGROUND);
if (carbon)
  dark = new Color(disp,230,230,230);
else
  dark = new Color(disp,245,245,245);
taskCol = new Color(disp,120,120,120);
```

(a) v1.7

```
int shift = carbon ?  -25 :  -10;
light = getColor(BACKGROUND);
dark = new Color(disp,
   light.red() + shift,
   light.green() + shift,
   light.blue() + shift);
taskCol = getColor(FOREGROUND);
```

(b) v1.8

```
1.17 int shift = isCarbon ? -25 : -10;
1.7  light = getColor(BACKGROUND);
1.8  dark = new Color(disp,
1.8     light.red() + shift,
1.8     light.green() + shift,
1.8     light.blue() + shift);
1.24 textCol = getColor(FOREGROUND);
```

(c) v1.38 (Version numbers on left)

```
int shift = isCarbon ? -25 : -10;
light = getColor(BACKGROUND);
dark = new Color(disp,
   max(0, light.red() + shift),
   max(0, light.green() + shift),
   max(0, light.blue() + shift));
textCol = getColor(FOREGROUND);
```

(d) v1.39

Figure 3.1: Eclipse Bug 63216 – `NewProgressViewer.java`. Names and formatting altered for clarity.

Figure 3.1 shows the four versions in the order they were committed to the repository, but as described previously, the process of identifying when a bug was introduced must necessarily work backwards, starting from when the bug is fixed. In version 1.39 the developers fixed a bug where, in certain scenarios, negative values were passed to the `Color` constructor causing an `IllegalArgumentException`. By comparing the fixed version to the previous version, version 1.38, we can see that the error was removed by updating three of the lines to wrap parameters with calls to `Math.max(0, ...)` and so avoid the negative numbers. Using the text approach first proposed by Śliwerski et al. [ŚZZ05b], to determine when this bug was introduced the *cvs diff* command is run on version 1.39. This command identifies all the lines that were added, removed or changed in that commit. Lines which are removed from the earlier

[2]Version numbers are formed of integers separated by periods, not decimal numbers e.g. 1.9 is followed by 1.10, 1.11..., not 2.0

version are highlighted in red, lines which are added in the later version are shown in green, and lines which have changed are shown in yellow, with the individual parts of the lines which differ highlighted in orange.

The next step is to run the *cvs annotate* command on the previous version of code, 1.38. This command displays the last version to change each line of code. These are shown on the left-hand side of Figure 3.1 (c). For each line altered in the fix the version it was previously altered in is considered a possible origin of the bug. In this example each of the updated lines was last changed in version 1.8, which the text approach reports as the origin of the bug.

This can be checked by comparing version 1.8 to version 1.7, again using the *cvs diff* command. This shows that this was indeed when the bug was introduced, as previously these parameters were set to specific numbers. It is only the change to use values derived from variables as the arguments to the constructor which introduced the possibility of negative numbers.

Various proposed improvements to the original approach are detailed in Section 3.1.4. Most pertinent to this study is that changes in formatting, whitespace and comments can be ignored, as they are unlikely to have been involved in causing or fixing a bug [KZPW06].

Unfortunately, the text approach is not suitable for all bugs. Figure 3.2 shows a bug involving a `NullPointerException` in the file `DecorationBuilder.java`. The bug was fixed by surrounding existing code with an `if`-statement that checks whether the variable is null. As these added lines did not exist in the previous version of code, *cvs annotate* cannot be used and the text approach cannot therefore identify any origin for this type of bug. A potentially more serious flaw is that there is no guarantee that a bug was actually introduced at the same location as it was fixed.

```
  void applyResult(DResult result){
1:  Descriptor [] d =
      result.getDescriptors();

3:  for(int i=0; i<desc.length; i++){
4:    if(d[i] != null)
5:      desc[i] = d[i];
    }

  }
```

(a) v1.6

```
  void applyResult(DResult result){
1:  Descriptor [] d =
      result.getDescriptors();
2:  if(d != null){
3:    for(int i=0; i<desc.length; i++){
4:      if(d[i] != null)
5:        desc[i] = d[i];
      }
    }
  }
```

(b) v1.7

Figure 3.2: Eclipse Bug 66653 – `DecorationBuilder.java`. Names and formatting altered for clarity.

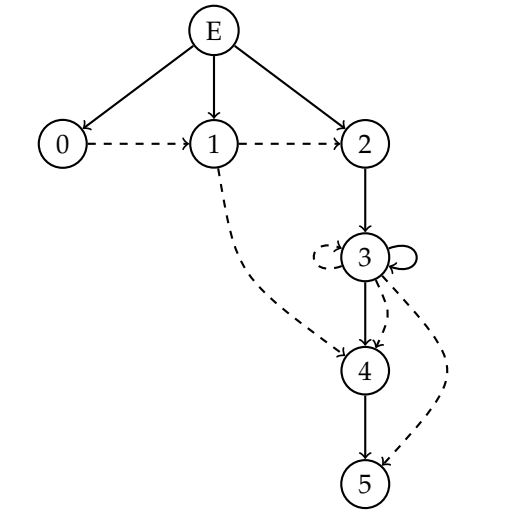
3.1.3 Dependence approach

The dependence approach, introduced by Sinha et al. [SSR10], attempts to address some of the text approach's shortcomings by examining changes in the behaviour of the code rather than simply the text, using a program dependence graph (PDG). The idea of using a PDG within software engineering was first proposed by Ottenstein and Ottenstein [OO84]. In this simplest definition of a dependence graph, each statement is represented by a node and an additional unique *entry* node is added to identify the entry point of the program. Directed edges connect the nodes to identify the control dependences and the data dependences. A control dependence

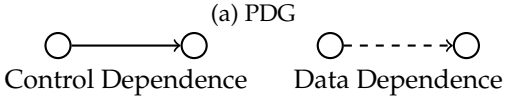
from node X to node Y (shown as $X \rightarrow Y$) exists if the execution of Y is conditional on the outcome of the execution of the predicate at X. A data dependence from node X to node Y (shown as $X \dashrightarrow Y$) exists if Y references a variable which is defined or updated in X.

The original definition of a PDG was made for a monolithic program, with no procedures. For the purposes of discovering bug origins in object-oriented languages, this is clearly unsuitable. However, a PDG can instead be constructed for each method, considering only the dependences within that method. This is the technique used in the original dependence approach, and therefore in this chapter, known as the intra-procedural approach.

An example of a simple method dependence graph for version 1.7 of the code in Figure 3.2 is shown in Figure 3.3. There are two points worth mentioning in relation to this illustration: firstly, an additional node (labelled *0*) has been introduced to represent the `result` parameter; and secondly the method contains references to an instance variable `desc`, but as this is external to the method there are no data dependences associated with it.



(a) PDG



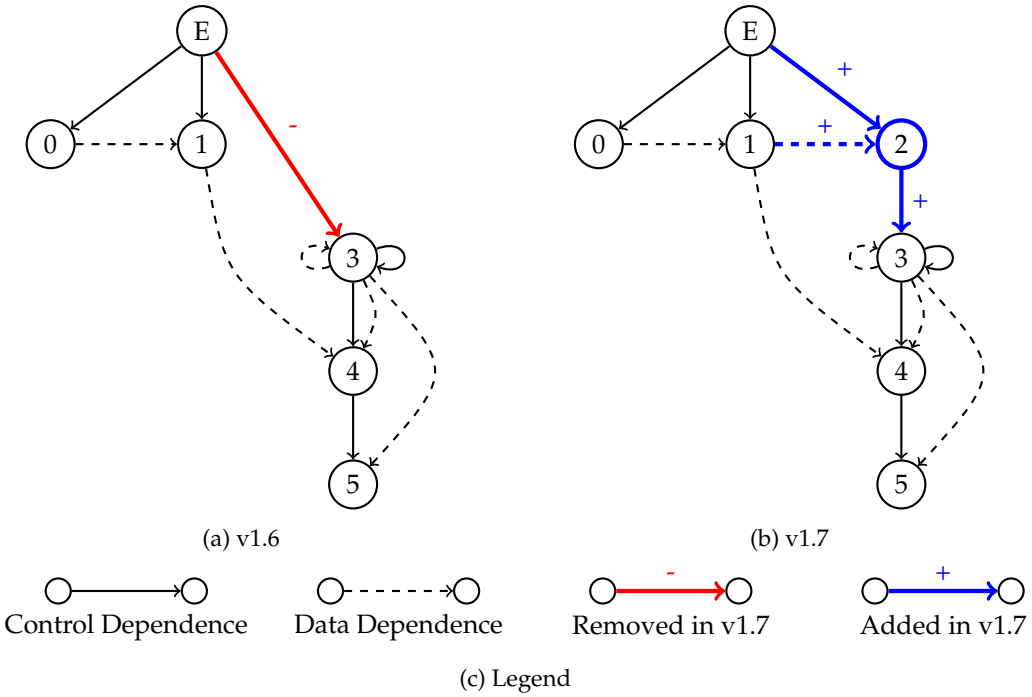
(b) Legend

Figure 3.3: PDG for `DecorationBuilder.applyResult()` v1.7 (See Figure 3.2)

One issue with the intra-procedural approach is that it disregards dependences on features such as instance variables or other methods. It is possible to instead construct inter-procedural PDGs, including dependences from outwith the method, but this adds a large amount of complexity. While it was not carried out here, the advantages and disadvantages of doing so are explored later in this analysis.

To determine the origin of the bug in Figure 3.2 the dependence approach compares the PDG for the fixed version to the PDG for the previous version. The approach first identifies any removed dependences. Added dependences are only examined if no dependences were removed. Figure 3.4 shows the PDGs for both versions of the code. As shown, the control dependence of line 3 on method entry has been removed and the line now has a control dependence on the new `if`-statement. The approach therefore builds the PDG for each preceding version until

it finds when the removed control dependence was added. In this case the dependence, and the bug, was introduced in version 1.5 when the method was created. If there are multiple dependences removed in the fix the dependence approach returns the most recent version in which one of these dependences was introduced.



(a) v1.6

(b) v1.7

(c) Legend

Figure 3.4: PDGs for `DecorationBuilder.applyResult()` (See Figure 3.2)

Figure 3.4 also shows that a new line 2 has been introduced, with two new corresponding control dependences and a new data dependence from line 1 to line 2. As discussed, the approach prioritises removed dependences but if no dependences had been removed the approach would have built PDGs for preceding versions until it found the most recent one that altered either the source or target line of any of the new dependences, i.e. method entry, line 1 or line 3.

The dependence approach returns only a single version, the most recent version involved. The original rationale for this was that if multiple dependences were involved in a fix, each of which was added at a separate time, then the bug would not have actually exhibited itself until the last of these was added. This is in contrast to the text approach, where multiple versions can be returned, potentially a different one for each line in the fix. The rationale for this is that each line contributed to the bug initially. Both of these could be considered valid interpretations of the origin of the bug, but it should be noted that these assumptions are not actually inherent to the approaches in question. Either approach could be modified to return a single or multiple versions.

This work uses the decision of Sinha et al. [SSR10], taking the last change involved to have been the one that introduced the bug. Note however that it is possible that what is reported as a single bug report may in fact manifest itself multiple times in code. Each of these manifestations can be considered a separate bug, which could have a separate origin. Therefore, even for a single bug report there can still be multiple versions in which a bug was introduced.

The dependence approach is not appropriate for all bugs. Using the approach described here the bug fix in Figure 3.1 would not result in any change to the method's dependences. The approach could not therefore identify the origin of the bug, but as seen the text approach managed to do so successfully.

3.1.4 Additional related work

Several improvements have been suggested to the approaches. Williams and Spacco [WS08] built on the text approach using a Java-syntax aware differencing tool, DiffJ [Dif]. This tool reports changes in terms of operations to Java elements, such as renaming or reordering. As well as allowing changes that have no semantic effect to be ignored, this also means that some changes, such as parameter reordering, can be disregarded as they are unlikely to fix a bug. Their approach also developed the concept of line mapping [KZPW06] further. Jung et al. [JOY09] detailed common patterns that can identify individual changes within a fixing commit which were not involved in the fix and proposed a tool for automatically detecting such patterns. These patterns included simple syntactically detectable patterns such as the addition or removal of unnecessary brackets and semicolons, or the renaming of methods. However, they also included those requiring further semantic analysis such as the replacement of constant values with variables of the same value or the addition and removal of temporary variables. The approaches also bear some resemblance to the concept of delta debugging [ZH02], where failing test cases from a later version of code are used repeatedly on earlier versions in order to identify the last version where the test passed. This was extended further with iterative delta debugging [Art09], in which fixes for other bugs are backported to older versions to lessen the chances of the test cases failing for unrelated reasons.

3.2 Evaluation

Ideally, researchers would like to use one or both of the proposed bug localisation approaches to identify the origins of a number of bugs, and learn from these origins. However, it is not clear at all whether either of the approaches is suitable for this aim. As stated previously, limited evaluation has so far been performed on the approaches, and it is difficult to know therefore if the results returned by them are accurate. The original text approach [ŚZZ05b] did not validate whether the commits identified by the approaches actually caused the bugs in question. Subsequent work on other refinements has focused on evaluating the difference between the results returned by the original approach and those of the refined approach. In particular, there has been no evaluation of false negatives: commits which introduced bugs but which are not identified by the approaches. It is important to address this lack of evaluation, and assess the accuracy of the approaches, before attempting to draw any conclusions from the origins they identify.

Identifying whether a commit caused a bug is time-consuming and subjective, as there is no one definition of *bug* and there may well be multiple causes that could be said to introduce a bug. Additionally, implementations of the approaches are not readily available, and are challenging to complete. There is a notable paucity of reliable and well-known implementations of tools to

support the creation and analysis of Java PDGs. Therefore, this section will evaluate a manual *simulation* of the approaches. This allows their expected effectiveness to be assessed, and the most appropriate approach chosen, prior to any future implementation, something which is likely to be vastly more time-consuming.

3.2.1 Subjects

The systems under examination are Eclipse, an integrated development environment (IDE) and development platform, and Rachota [Rac], a time tracking application. These were selected as they vary considerably in size, maturity and usage, although both are open-source, written in Java and use CVS. Bug and commit data for Eclipse was obtained from *Promise* [BMO07], and contains a collection of fixes, each of which is a commit linked to a bug as described in Section 3.1.1. The set contains every fix which could be linked to a bug and which occurred from six months before the 3.0 release of Eclipse until six months afterwards. However, bugs may have been introduced in earlier releases and all previous versions were included in the evaluation. A series of scripts was used to select a random sample of 100 bugs, out of 4136 in total. These were linked to 301 separate commits (from a total of 10402). While this sample size may be considered small, the time required for evaluation prohibited the study of a larger sample.

The data for Rachota was obtained from FLOSSMetrics [Flo]. By a manual examination, all issues raised before 3rd September 2008 and fixed before 15th March 2011 were separated into bugs and enhancements, as the Rachota BTS does not record this information. Then 242 fixing commits were identified for each of the 66 bugs in Rachota. A number of these commits only modified plain text files, i.e. files that did not contain actual source code. While the text approach can be used on these files, the dependence approach cannot. Therefore, only the 130 commits involving Java files will be considered in this evaluation.

Further evaluation was carried out on a sample of jEdit bugs provided by Dit et al. [DRGP11]. jEdit is a text editor for programmers maintained by the free software community, which has been used as a case study in numerous research studies. jEdit was used here for a number of reasons:

- To ensure that a third case, carried out with additional researchers[3], found the same kind of issues that arose with Eclipse and Rachota
- To provide deeper qualitative insights into the application of the two techniques
- To lay groundwork for developing a standardised process for documenting bugs and the outcome of the application of the techniques, which could be used in future studies

Like Eclipse and Rachota, jEdit also uses Bugzilla as its BTS, but unlike those projects it uses Subversion as its RCS. The approaches are applicable to any RCS, but there is one slight difference. In CVS, each file is committed individually, and a version number is maintained for each file individually. In Subversion, all modified files are committed together, and a revision number is used which identifies the state of the entire project, not just a single file.

[3] Additional researchers were the supervisors of this thesis. All work on Eclipse and Rachota was done by the author alone.

All three researchers applied both approaches to eight selected jEdit bugs and then worked through each in detail to identify key decisions that needed to be made to ensure consistency of application, particularly with respect to the dependence approach. Some of the issues that arose are discussed in Sections 3.3 and 3.4, with some jEdit examples being used as detailed examples throughout the chapter.

3.2.2 Research procedure – a worked example

This section describes the research procedure used to simulate the text and dependence based approaches. To demonstrate both of these a detailed example of the analysis of jEdit Bug 1965114 will be used.

The bug was related to the Subversion revision that fixed the bug through the use of the bug ID in the commit comment, as provided by the original sample [DRGP11]. Additional scripts were written to parse the provided dataset into an HTML page with links to jEdit's Bugzilla and ViewVC [Vie] servers to aid analysis. An example of the output from these scripts for Bug 1965114 is shown in Figure 3.5.

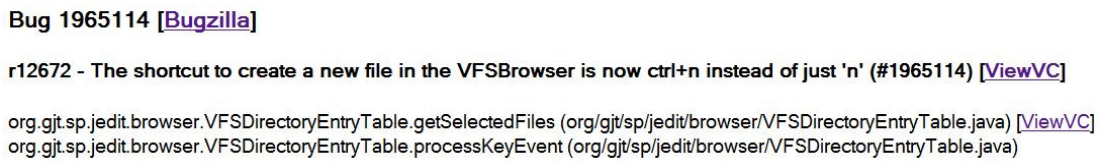


Figure 3.5: Bug summary relating Bugzilla description to Subversion revision

The *Bugzilla* hyperlink in Figure 3.5 opens the bug description shown in Figure 3.6 and the *ViewVC* hyperlink opens the revision description shown in Figure 3.7. The bug is described as: "Pressing 'n' with the file system browser dockable open, and the file list focused, opens a new file." The problem is that typing an `n` character in a filename opens a new file, making it impossible to open files with the letter `n` in their filename using the keyboard. The shortcut to open a new file is supposed to be `Ctrl+n`, not just `n`.

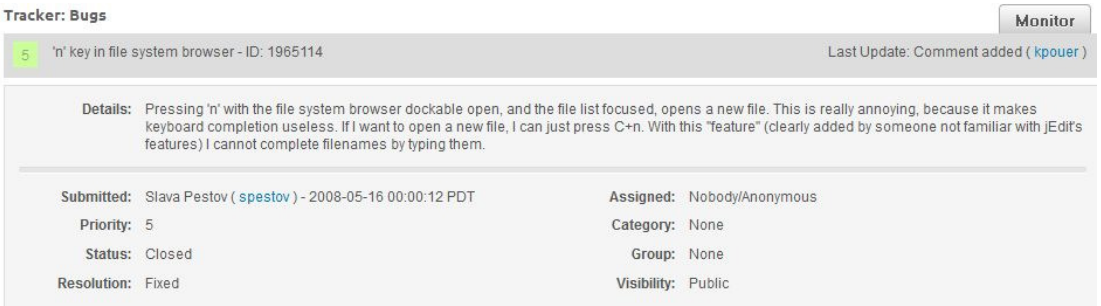


Figure 3.6: Bugzilla: description for Bug 1965114

The Subversion revision description (Figure 3.7) identifies the jEdit commit that 'fixed' the bug (12672) and all the files that were modified during the fix. Often there will be multiple source code (`.java`) files modified for one fix. In this example the text file `CHANGES.txt` (the jEdit release history) has been updated to include a textual description of the bug fix and the single file `VFSDirectoryEntryTable.java` has been modified.

Figure 3.7: ViewVC: Subversion revision description for 'fix' to Bug 1965114

There are now three stages of analysis to be performed using the RCS: identifying the jEdit revision where the bug was introduced; simulating the text approach; and simulating the dependence approach.

3.2.2.1 Origin identification

The first stage of the analysis is to identify the true origin[4] of the bug. This can then be compared to the origins discovered by each of the text and dependence approaches in order to assess their performance. To begin identifying the origin of the bug the BTS entry is read to see if there is any information that could aid understanding of the nature of the bug and its potential source. For Bug 1965114 this helps since it indicates that the origin code is likely to be concerned with file shortcut options, which helps to focus the examination of the modified code.

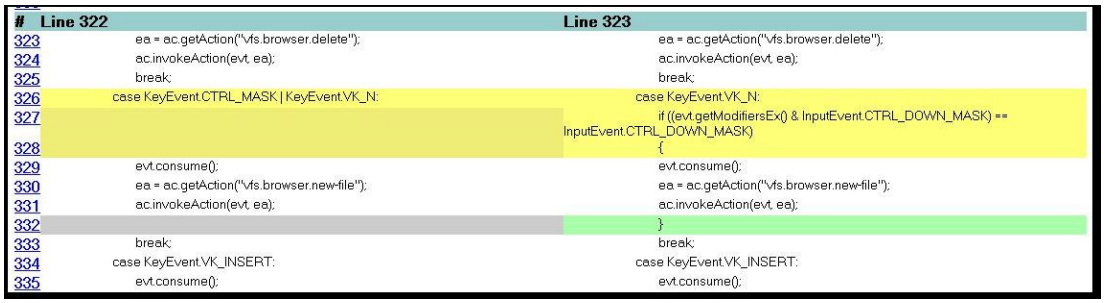


Figure 3.8: ViewVC: jEdit Bug 1965114 side-by-side comparison of revisions 12671 and 12672

Next, the revision that fixed the bug, 12672, is examined. First, the `CHANGES.txt` file was checked in case it contained helpful information; no further information was gained in this case. Second, the source file involved was examined using the annotated view provided by ViewVC. This provides a side-by-side *svn diff* comparison against the preceding version of this file highlighting each line added, removed or changed. Figure 3.8 shows a snapshot of the annotated view for revision 12672 compared to its predecessor 12671. The lines highlighted in yellow (326–328) have changed between revisions and those in green (332) were added (any

[4]Here, and for the remainder of the chapter, identifying the origin means identifying the version and file in which the bug was introduced, but not necessarily the exact line or method

removed lines would be shown in red). Line 327 appears to be an added conditional to fix this bug.

In revision 12672 of `VFSDirectoryEntryTable.java` there were many other changes: ten different lines were changed in total, distributed throughout the file from line 128 to line 624. All of these changes must be considered as potential contributions to the bug fix. In this example these other changes appear to be concerned with opportunistic cleaning of the code – replacing a concrete `LinkedList` by the `List` interface and removing many unnecessary brackets. Such incidental changes can have unfortunate consequences for the two approaches.

Having determined that the fix is likely to be the changes at lines 326–332 in revision 12672 and that the erroneous code is therefore on line 326 of revision 12671, the task now is to identify the origin of this bug – in what revision was the bug introduced? By examining the Subversion annotated view of revision 12671 (shown in Figure 3.9) the last revisions to change the context around line 326 are highlighted.

Figure 3.9: ViewVC: annotated revision view – the numbers in column 3 are hyperlinks to the last revisions that modified the lines in column 1

Opening the most recent revision 10326 associated with line 325 shows that there was only a change of layout in this revision, with no changes to the code. Opening revision 7998 reveals that all the code associated with this case branch was added in this revision, as shown in Figure 3.10. Revision 7998 is therefore deemed to be the origin of Bug 1965114.

3.2.2.2 Text approach

The second stage is to now simulate the text approach. This means going back to revision 12672 which was identified as containing the bug fix. The evaluation was done as if each version of the code had been run through a preprocessor to standardise whitespace and brace formatting, and remove comments. This meant changes in whitespace, formatting and comments were disregarded. Adding or removing comment markers around lines of codes was therefore treated as removing or adding those lines respectively. Changes to import statements were also ignored as these were either accompanied by other changes or were unused and so unrelated to the bug. Otherwise, the text approach assumes that all other changes made in the revision that fixes the bug are part of the bug fix. Furthermore, as mentioned earlier, the text approach cannot deal with fixes that involve adding lines of code and so the focus is only on removed and changed lines.

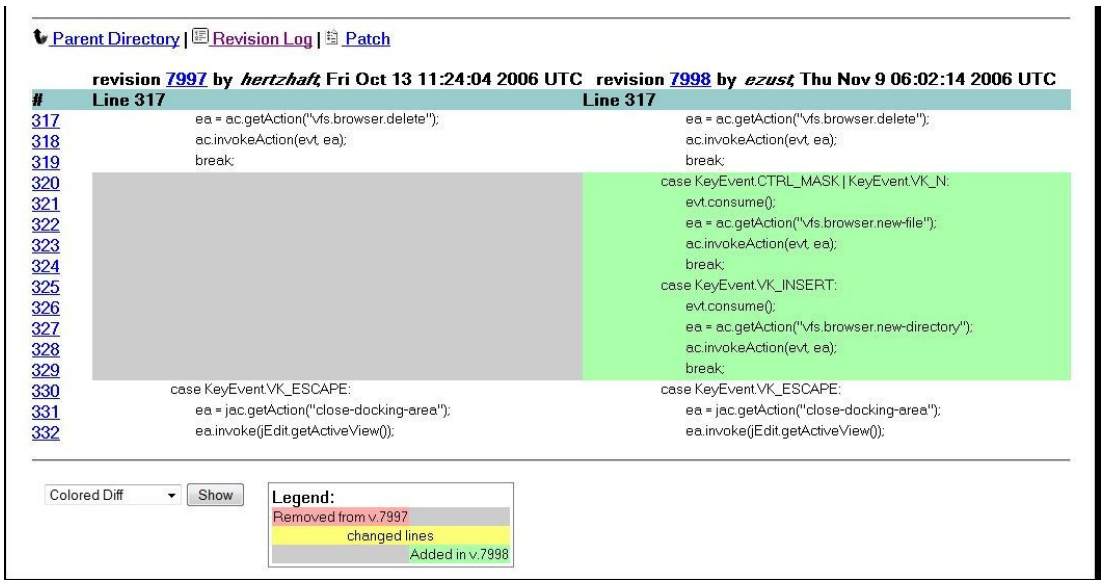


Figure 3.10: ViewVC: code containing Bug 1965114 was added in revision 7998

The text approach works by taking each line that has been changed or deleted in the revision that fixes the bug (subject to the exclusions just mentioned) and tracking back through the revision history to identify the revisions that previously altered these lines. The set of revisions identified is considered the potential origin of the bug in the text approach.

Revision 12672 included ten separate line changes and one line addition. The ten line changes are therefore all traced back through the revision history as potential origins of Bug 1965114. This is similar to the process described above to identify the origin of the bug. The problem is that the text approach cannot distinguish amongst any of the ten changes, as was done above where an understanding of code semantics was used to focus on the changes made around line 326 as being the actual bug fix. Tracking back for each of the ten changes leads to a set of possible bug origins, potentially one origin for each change. In this case the set of revisions identified is 4640, 4648, 5179, 5217, 7170, 7998, 9596, 10275 and 11009. This is a clear example of one of the limitations of the text approach. Notice that the set *does* include the origin, 7998, as a result of finding the revision that introduced the change at line 326.

3.2.2.3 Dependence approach

The third stage is to simulate the dependence approach. The first step is to map methods between versions based on their name and signature. Although it did not occur in this case, this allows the dependence approach to handle methods which are relocated within the class (not true of the text approach).

Next, each method of the bug fix version is compared to the preceding version and any removed or added dependences are noted. In the case of removed dependences, each preceding version of an updated method is manually compared until the most recent version to add one of the dependences is found. If no dependences are removed then the same process is performed, searching for the most recent version to have altered either the source or target of any added data/control dependences. In contrast to the text approach, rather than identifying the full

set of potential bug origins, the dependence approach identifies only the most recent revision associated with a removed/added dependence and assumes that revision is the single bug origin.

The dependence approach therefore starts with the *svn diff* between revision 12672 and its predecessor 12671. Of the ten changes and one addition, eight are concerned with the removal of unnecessary brackets and one is the addition of a closing brace matching the new conditional at line 326. This leaves only two changes that could affect the data or control dependences. Only dependences on other lines within the same method were considered; dependences on other methods and on any fields were ignored.

The first change is line 128 which was altered from:

```
LinkedList<VFSFile> returnValue = new LinkedList<VFSFile>();
```

to:

```
java.util.List<VFSFile> returnValue = new LinkedList<VFSFile>();
```

The dependences within this method are shown in Figure 3.11 (b) and the relevant sections of code in Figure 3.12. As shown, line 128 has a control dependence onto method entry and lines 132 and 134 have data dependences on line 128.

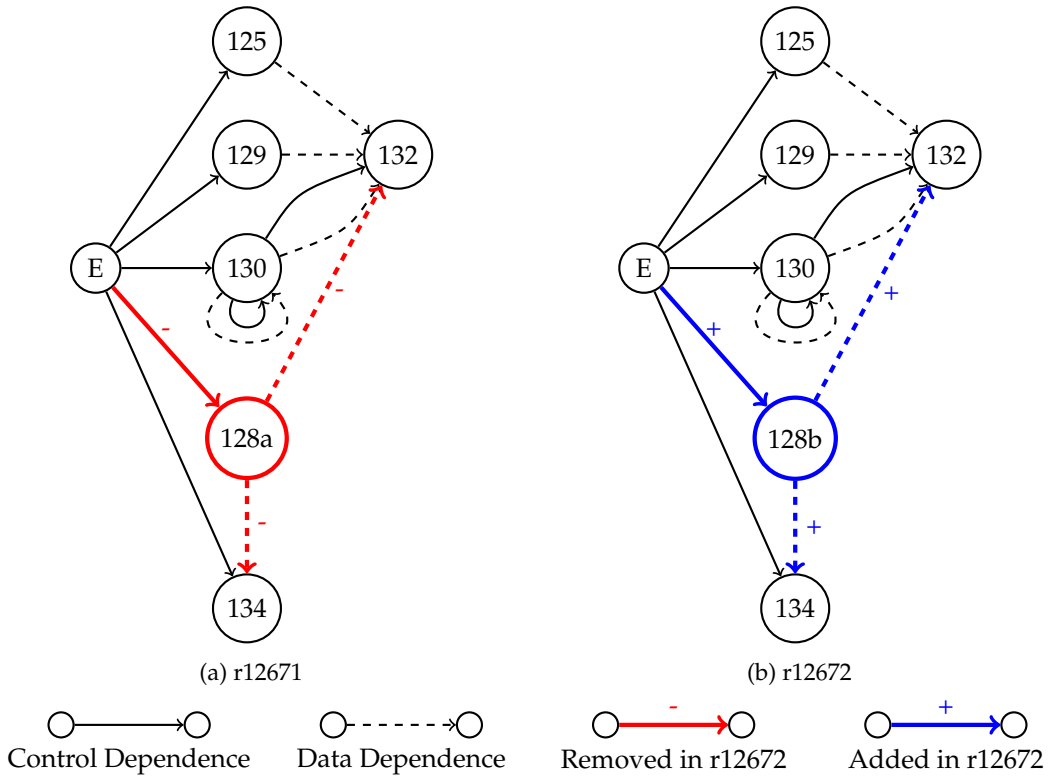
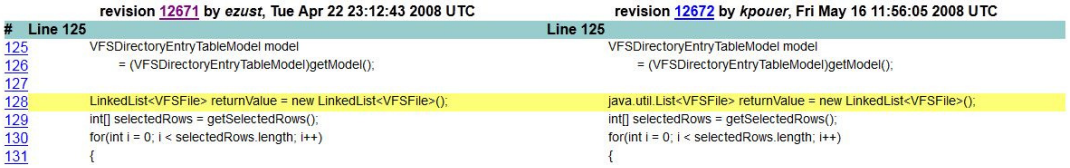


Figure 3.11: PDGs for `VFSDirectoryEntryTable.getSelectedFiles()`

When comparing two methods, the first step of the dependence approach is to examine the graph for each version and determine which nodes in the first graph correspond to which nodes in the second graph, if any. This mapping is based on the similarity of the two lines and the lines surrounding them. Generally this was straightforward to do but where there were ambiguities

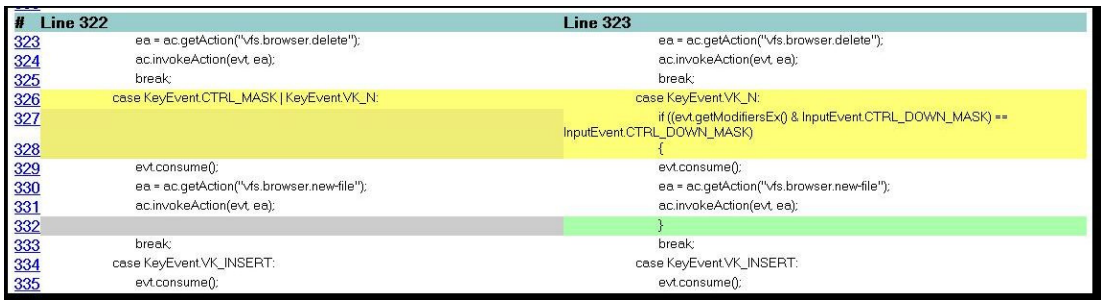


#	revision 12671 by *ezust*, Tue Apr 22 23:12:43 2008 UTC Line 125	revision 12672 by *kpouer*, Fri May 16 11:56:05 2008 UTC Line 125
125	VFSDirectoryEntryTableModel model	VFSDirectoryEntryTableModel model
126	= (VFSDirectoryEntryTableModel)getModel();	= (VFSDirectoryEntryTableModel)getModel();
127		
128	LinkedList<VFSFile> returnValue = new LinkedList<VFSFile>();	java.util.List<VFSFile> returnValue = new LinkedList<VFSFile>();
129	int[] selectedRows = getSelectedRows();	int[] selectedRows = getSelectedRows();
130	for(int i = 0; i < selectedRows.length; i++)	for(int i = 0; i < selectedRows.length; i++)
131	{	{

Figure 3.12: ViewVC: difference between revisions 12671 and 12672 in jEdit (corresponding PDGs shown in Figure 3.11

these were recorded and re-examined at the end in order to ensure that similar cases were treated in the same manner. Specifically, if changes were made to an object's type, or a method was added to or removed from a chain of method calls, then the two nodes were considered to not map to one another. Changes that were made to the condition of an `if`-statement were regarded as the nodes mapping to one another. These decisions are somewhat arbitrary, but are based on suggestions in the original description [SSR10], and ensure consistency in the evaluation.

As such, as this is a change of type, the two nodes representing line 128 are not considered to match. Therefore both the control and data dependences have been removed (and new ones added), as shown by the highlighted edges in Figure 3.11. The dependence approach then searches the previous revisions to determine where any of those dependences were originally added. This process is performed in a similar manner to the text approach described above, using ViewVC to track back through the jEdit revisions. In this case, the generic parameter `VFSFile` was introduced in revision 9596, which would be seen as the introduction of the dependence, although prior to that the original data dependence was introduced in the baseline revision of jEdit, 4631. Had the two nodes representing line 128 instead been considered to match one another, then the two graphs would have been seen as identical: no dependences would have been seen as added or removed. This change would therefore not have led to any origin. The effects of these decisions are explored in more detail in Section 3.4.

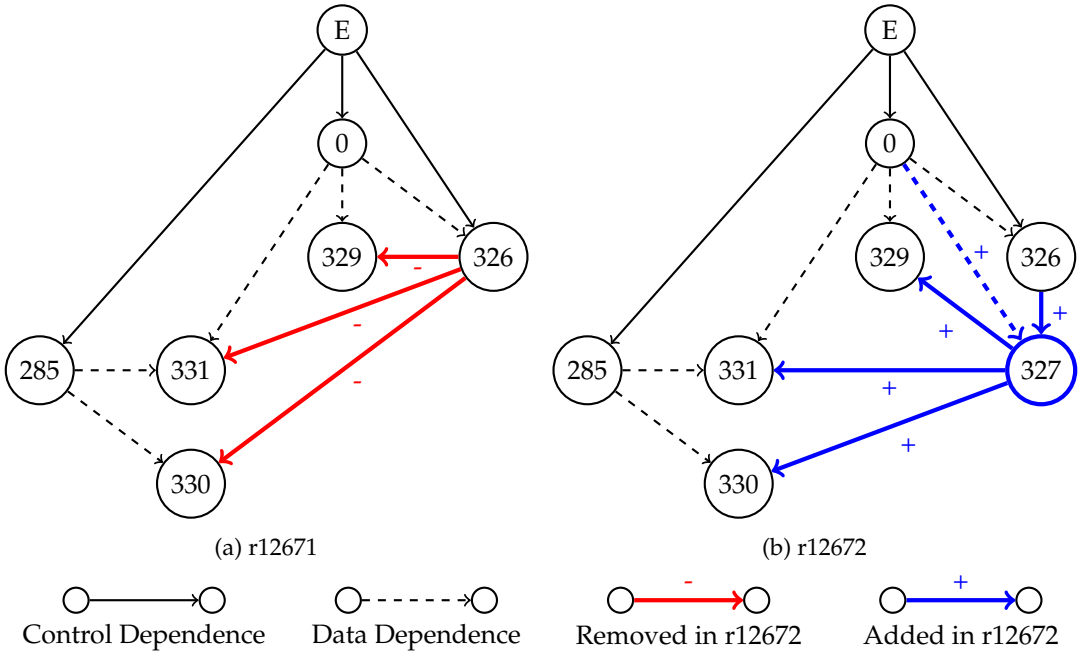


#	Line 322	Line 323
323	ea = ac.getAction("vfs.browser.delete");	ea = ac.getAction("vfs.browser.delete");
324	ac.invokeAction(evt, ea);	ac.invokeAction(evt, ea);
325	break;	break;
326	case KeyEvent.CTRL_MASK | KeyEvent.VK_N:	case KeyEvent.VK_N:
327		if ((evt.getModifiersEx() & InputEvent.CTRL_DOWN_MASK) == InputEvent.CTRL_DOWN_MASK)
328		{
329	evt.consume();	evt.consume();
330	ea = ac.getAction("vfs.browser.new-file");	ea = ac.getAction("vfs.browser.new-file");
331	ac.invokeAction(evt, ea);	ac.invokeAction(evt, ea);
332		}
333	break;	break;
334	case KeyEvent.VK_INSERT:	case KeyEvent.VK_INSERT:
335	evt.consume();	evt.consume();

Figure 3.8 (Repeated): ViewVC: jEdit Bug 1965114 side-by-side comparison of revisions 12671 and 12672

The only other change is the code associated with the bug fix shown in Figure 3.8 (repeated above). Here the control dependences of the three assignments (starting `evt.consume();`) onto line 326 are removed and replaced with control dependences onto the new `if`-statement, as shown in Figure 3.13. In addition a new control dependence from the `if`-statement to the `case`-statement is introduced. As stated above, changes within conditionals are regarded as lines mapping to each other in the PDG, so the change on line 326 can be disregarded. Therefore, the

focus here is on the removed control dependences and the revision(s) in which they originated. Since this is the code associated with the bug fix, tracking back to identify when these were introduced shows that they were all added in revision 7998.



(a) r12671

(b) r12672

(c) Legend

Figure 3.13: PDGs for `VFSDirectoryEntryTable.processKeyEvent()` corresponding to code in Figure 3.8 (Many unchanged nodes and dependences have been omitted)

As mentioned above, the dependence approach only identifies one revision: the most recent based on removed dependences (or added dependences if none are removed). Here the dependence approach will return revision 9596. Note though that if the two versions of line 128 had been considered to be the same node, by ignoring type changes, then the approach would have returned the actual origin, 7998. This example therefore highlights the major impact that subtle changes in the definition of PDG differences have on the dependence approach, particularly since it only returns a single revision. This is explored further in Section 3.4.

3.3 Results and analysis

Before looking at the results of the Eclipse and Rachota evaluation, it is useful to highlight the breadth of issues raised by just the small sample of jEdit bugs.

3.3.1 Qualitative analysis – jEdit

A small, qualitative analysis was carried out using eight jEdit bugs. This was used by the author and the two additional researchers to cross-check and develop an agreed understanding of the application of the text and dependence approaches and also to develop a standard process for summarising the bugs, their origin and the outcome of applying the techniques. Such a process could be used in future evaluations or replications.

This section summarises the eight jEdit bugs. One of these bugs (1965114) was used as a detailed example in Section 3.2.2. The eight bugs discussed were not randomly selected. Instead, these were specifically selected to highlight some of the subtleties that arose in the application of the approaches, particularly the dependence approach, across all three case studies. All three researchers independently applied both approaches to the selected jEdit bugs.

For each bug this section: shows the jEdit bug identifier; gives a brief description of the bug, from the bug report; states the identified origin; lists the revision(s) identified by the text approach; lists the revision identified by the dependence approach; and highlights any interesting observations. These observations and some of the other issues that arose throughout this small analysis are discussed in more detail in Section 3.4.

Bug 1965114: "Shortcut to create a new file in `VFSBroswer` is `n` rather than `Ctrl+n`"
Origin: Revision 7998, which is when the shortcut code was originally added
Text approach: 9 revisions including 7998
Dependence approach: Revision 9596
Notes: Text approach identifies many revisions due to multiple opportunistic changes in bug fix file. There are subtle questions raised for dependence approach: do type and generic type parameter changes cause PDG changes?

Bug 1584436: "Off-by-one error, local variable `i` should have been decremented"
Origin: Revision 5216, which updated one use of `i` but not the others
Text approach: Revision 5173
Dependence approach: Revision 5173
Notes: Bug was caused by change in context which therefore misleads the text approach. There are no changes to data or control dependences in the actual fix, but changes elsewhere mislead the dependence approach.

Bug 1834620: "Lexer problem - only seems to recognise one character after $"
Origin: Revision 4826, when the original functionality was added
Text approach: Nothing
Dependence approach: Revision 4826
Notes: Text approach can't handle this as the bug fix only contains added code. For the dependence approach, data and control changes both lead to origin.

Bug 1541372: "Caret positioning on text folding."
Origin: Revision 3791, when functionality was introduced
Text approach: 4 revisions including 3791
Dependence approach: Revision 4567
Notes: Intervening changes between origin and fix cause difficulties for both approaches. Fix includes changes to multiple files, one of which is relevant but the other is unrelated.

Bug 1600401: "Long lines cause editor to hang when `enter` pressed on them"
Origin: Revision 8103, where the original condition was added without a length check

Text approach: Revision 8107
Dependence approach: Nothing
Notes: Intervening change causes difficulty for text approach. This bug raises a question about dependence approach: if adding conjunct to `if`-statement causes a control dependence change then it finds the wrong revision; if it only causes a data dependence change then it finds the correct revision.

Bug 1571629: "Issue with filtering files in Windows, looks like was a problem with Windows from original"
Origin: Revision 6808. The problem with Windows OS has existed since functionality was added.
Text approach: Revision 6808
Dependence approach: Revision 6808
Notes: Good example of both approaches performing correctly, as all changes lead to origin.

Bug 1548902: "Formatting empty paragraph throws exception, size isn't checked"
Origin: Revision 5196, when the ability to move caret position was introduced
Text approach: Many revisions including 5196
Dependence approach: Revision 5324
Notes: Opportunistic changes such as changing `StringBuffer` to `StringBuilder` mean that the text approach gathers many revisions. The result for the dependence approach depends on PDG subtleties: if type changes are included the approach gets the wrong revision, but otherwise it would have been correct.

Bug 1542026: "Manipulating menus using keyboard – characters going into text file."
Origin: Revision 5514, when original coded was added
Text approach: Nothing
Dependence approach: Revision 5514
Notes: Dependence approach works as intended and text approach fails because there is only added code.

3.3.1.1 Summary

The jEdit study, though small, clearly demonstrates consistent findings with the main quantitative studies on Eclipse and Rachota which follow this section:

- There are examples where each approach clearly works as intended.
- Both approaches are impacted by intervening changes between the origin and the bug.
- The dependence approach is sensitive to the definition of program (and system) dependence graph used – it is not clear, however, that one specific definition will consistently provide the best results.

- Most bug fixes had many *opportunistic* modifications as well as the bug fix, this means that the text approach often returns many revisions (including the correct one), while the dependence approach is more likely to be misled because it only returns the most recent revision.
- Bug fixes which involve changing surrounding context rather than the problematic lines can mislead the text approach; the dependence approach has more potential to identify the origin in these cases.

3.3.2 Quantitative results – Eclipse and Rachota

Table 3.14 shows a summary of where the origin of the bug lies for each of the bug fix commits in Eclipse and Rachota. The majority of commits fixed a single instance of a bug where the origin and the fix lay within the same file, shown as *Single*. Some commits contained fixes for more than one instance of the same bug, shown as *Multiple*. These bugs may have all originated at the same time or may have been introduced in different commits, but in all cases the origin and the fix lay within the same file. A number of commits contained fixes for bugs for which the origin actually lay in a commit made to a separate file and these are classed as *Elsewhere*. Finally some commits which claimed to be bug fixes were not actually fixes, but were instead consequences of the bug fix or unrelated changes, and these were classed as *Related* or *Unrelated*. Examples explaining the distinction between categories in greater detail will be given later in this section. For 20 commits the origin could not be determined, as the changes made by developers were complex and it was not possible to fully understand them or to identify if they were related to the bug in question. These commits were classed as *Unclear*. Note that these 20 files relate to just 3 bugs. All files for these bugs have been omitted from the remainder of the evaluation.

	Eclipse	Rachota
Single	161	88
Multiple	19	21
Related	57	6
Elsewhere	6	1
Unrelated	38	14
Unclear	20	0
Total	301	130

Table 3.14: Count of commits by classification of origin

Only for bugs classed as *Single* or *Multiple* could the two approaches potentially return the correct answer. Identifying the origin of *Elsewhere* bugs would require an analysis over the whole system, not just on one individual file. Commits classed as *Related* and *Unrelated* claim to be bug fixes but there is actually no bug in the file; for these bugs there is no origin and therefore any answer returned by either of the two approaches would be an error.

For each approach the predicted origins were compared to the manually identified origins. The predictions that the approach made correctly were recorded as true positives (TP). False positives (FP), versions that the approach predicted but which were not correct, and false negatives (FN), origins which were manually determined but the approach did not predict, were also recorded.

Table 3.15 shows the results obtained by the text approach and dependence approach. In order to help compare the performance of each the commonly used measures of precision (P) and recall (R) are also shown, along with their harmonic mean F_1-Score (F). As can be seen, for Eclipse the text approach identifies more correct versions than the dependence approach. However, the text approach also generates a much larger number of false positives, as it can return multiple origins for each commit while the dependence approach only returns a single origin. As is to be expected from the lower number of false positives, the precision of the dependence approach is higher, at 44% compared to 29%, although the recall is only 40% compared to 48% for the text approach.

		TP	FP	FN	P	R	F
Eclipse	Text	91	220	100	0.29	0.48	0.36
	Dependence	77	98	114	0.44	0.40	0.42
Rachota	Text	89	39	38	0.70	0.70	0.70
	Dependence	85	23	42	0.79	0.67	0.72

Table 3.15: Overall results

A similar pattern is seen for Rachota but the proportion of false positives compared to true positives is vastly reduced compared to Eclipse. This may be because of the relative simplicity of changes in Rachota. The bugs, and their fixes, often seemed simpler than those in Eclipse and there were often fewer versions between the origin of the bug and the fix. Unsurprisingly, given the smaller number of false positives, the precision and recall for Rachota are higher than for Eclipse.

As detailed in Section 2.5, a growing number of practical tools have been built on top of these techniques, such as HATARI [ŚZZ05a] and FixCache [KZWZ07]. These attempt to use the information about the types of changes which have previously introduced bugs to warn developers when they make similar changes that they may be introducing bugs. However, as discussed previously, the original approaches did not validate whether the commits they identified were actually the causes of the bugs in question, and indeed the results in Table 3.15 suggest they may very well not be. Future work should be carried out to identify whether the inaccuracies identified in the two approaches affect the performance of the techniques and tools built upon them.

3.3.3 Analysis of Eclipse and Rachota

The approaches successfully found origins for a variety of different types of bug. Of the 68 bugs in Eclipse for which at least one origin was successfully identified (out of a total of 100 bugs), around a third resulted in exceptions that either crashed the application, displayed an error to the user or appeared in logs. However, bugs in other areas were also successfully identified:

UI: Bug 63753 – "Checkbox being ignored"
Tests: Bug 74229 – "Failing automated tests"
Code Reviews: Bug 57670 – "Wrong subclass of `InputStream` was being used"
Performance: Bug 64531 – "Find/Replace operation using 100% CPU"

There was a similar diversity of bugs identified in Rachota, although there was a greater proportion of user interface bugs and incorrect output rather than exceptions. This is most likely down to the nature of the applications and of their users. As an IDE, Eclipse is more likely to be used by more technical users, and it often prominently displays exceptions to the users where other applications may hide them. Overall, the approaches did not seem to be more or less effective for any particular type of bug, but there were a number of factors that impacted on their performance. This section will highlight some of these with a number of illustrative examples of bugs.

3.3.3.1 Graph matching, multiple bugs, age of origin and large fixes

Eclipse Bug 49561 – "Commit should only lock parent's folder"

Some bugs demonstrated a wide range of issues.

Although the description only mentions commit operations, the actual bug involved a number of different operations locking the entire Eclipse workspace, preventing other operations from accessing it, when it was only actually necessary to lock individual folders. One of the changes made to fix the bug was version 1.156 of `CompilationUnit.java`, in which a number of lines were changed from:

```
runOperation(operation, monitor)
```

to:

```
operation.runOperation(monitor)
```

This was done as the method `JavaElement.runOperation` had been moved, and during the move changed, to `JavaModelOperation.runOperation`. Each of the calls was used as part of a different operation and therefore each represents a separate instance of the bug. Some of these bugs were introduced in version 1.1 of the file, while others were introduced in version 1.98.

Graph matching The text approach correctly identified both origins of this bug, as these lines had not been materially altered since the bug was introduced. However, as in jEdit Bug 196511, the graph matching technique used has a major impact on what the dependence approach returns. If the line `runOperation(operation, monitor)` is considered to be the same node as `operation.runOperation(monitor)`, then the dependence approach would not return anything at all. However, as the evaluation considered those two lines to be different, the dependence approach returned the most recent version to introduce one of the relevant dependences, version 1.140. The technique used to match nodes between different graphs has a major impact on the conclusions reached by the dependence approach. However, as later examples will show, there is not one definitive strategy which will always give the right answer.

Multiple bugs Even if the dependence approach had returned *a* correct answer, it always only returns a single answer, so could not have hoped to identify both origins. This bug is an example of a number of bug reports in the Eclipse sample where one bug report actually

represents multiple real bugs. Since these were often introduced at different times, the dependence approach fared worse at identifying these, as shown by the row for *Multiple* bugs in Table 3.16.

	Text			Dependence		
	TP	FP	FN	TP	FP	FN
Single	79	75	82	73	41	88
Multiple	12	5	18	4	4	26
Related	0	88	0	0	29	0
Elsewhere	0	10	0	0	4	0
Unrelated	0	42	0	0	20	0

Table 3.16: Results by origin (Eclipse)

Age of origin Quite a large number of versions passed between the bug being introduced and being fixed. While for this bug, and some others, the text approach was still successful, this was not generally true – the more versions passed between the introduction and the fix the less likely it was the origin could successfully be found, as will be shown in Section 3.4.4.

Large fixes `CompilationUnit.java` was only one of the files involved in this fix. In total, there were 13 files whose commit comments indicated they were related to this bug. However, the knock-on effects of moving the `runOperation()` method described above were responsible for most of these. In 3 of the files, there was no actual incidence of a bug, although the code changed was clearly involved in the fix. These commits were classed as *Related*. In an additional 7 files code that was not involved in the bug at all had to be altered and these commits were classed as *Unrelated*. For both these types of change, since there was never any bug in the file, any result returned by an approach would be wrong. The ideal outcome in fact would be for the approaches to return nothing. This did not happen however: these changes resulted in 19 false positives for the text approach and 7 for the dependence approach. This was often true of other bugs; as can be seen in Table 3.16 these types of commits are responsible for a significant number of false positives.

3.3.3.2 Unrelated changes

Eclipse Bug 49891 – "Problems launching a program, when using linked resources in CLASSPATH, which are not set"

This bug is an example of a fairly straightforward fix for which the origin can be quite easily identified, but which can still result in issues for both approaches. In version 1.37 of `RuntimeClasspathEntry.java` the line:

```
return res.getLocation().toOSString();
```

was altered to check first whether the value of `res.getLocation()` was null.

This possibility had existed since version 1.1, the origin of the bug, but version 1.8 of the file had changed the line from the previous:

```
return new String[] {res.getLocation().toOSString()};
```

to its current form.

Both approaches therefore returned version 1.8 of the file as the incorrect origin. Unrelated changes often tripped up both approaches, although in general the dependence approach appeared to handle them slightly better. In particular, because it explicitly mapped methods between versions it could handle code being relocated within a file where the text approach could not.

It is difficult to see here how either approach could account for this problem, however. The change in version 1.8 is indeed a semantic change and cannot simply be disregarded, as it is entirely possible that the same scenario could be the fix for another bug under different circumstances.

3.3.3.3 Bugs caused by changes to other files

Eclipse Bug 54538 – "Bundle-SymbolicName value has changed"

Not all bugs were caused by a change made earlier in the same file. This bug was raised as the class `BundleManifest.java` was not correctly parsing the value of a `String`. It was fixed in version 1.6 of the file. However, the bug was originally caused by a change that was made elsewhere in the system to change the format of the `String`, not in the `BundleManifest.java` itself. At that time, a number of classes which parsed this value were updated, but this class was incorrectly omitted, and this bug then raised at a later date when the problem was discovered. As the origin lies in another class, this commit was therefore classed as *Elsewhere*, and neither the text approach nor the dependence approach could correctly identify the origin.

3.3.3.4 Coincidental correctness

Eclipse Bug 65354 – "[Perspectives] Perspective bar incorrectly updated when opening new perspective"

The fix for this bug was in version 1.11 of `PerspectiveBarManager.java`. Here, two lines were updated from:

```
if (coolBar != null) LayoutUtil.resize(coolBar);
```

to:

```
if (getControl() != null) LayoutUtil.resize(getControl());
```

However, as neither `coolBar` or `getControl()` are local variables, their dependences are not included in either graph and the intra-procedural dependence approach would therefore not be able to determine their origin (although it is possible an inter-procedural approach could have).

An additional change was also made before those lines to remove a call to `updateCoolBar()` (that method was also removed). However, the `updateCoolBar()` method did not actually do anything: its entire contents had been commented out in an earlier version. The removed control dependence on the call however causes the dependence approach to return the correct origin for the bug. In effect, the approach is only correct due to the developer making an unnecessary change.

As this bug shows, developer behaviour has a major effect on the quality of the two approaches. The approaches may be more effective for projects which have a strict policy on making one commit per logical change and less so for projects where developers tend to commit a large variety of changes all in one go. It also raises the question: even if a way could be found to ignore changes which are incidental or unrelated to the main fix, should they be, or are these still useful clues to help discover the origin? Here for instance, it was only due to the unrelated change that the dependence approach worked. After all, developers usually only change code if it is at least tangentially related to the code they need to fix; they are less likely to make changes to code which is totally disconnected from that involved in the bug.

It is also useful to note that the intended use of the approaches will have an effect on how useful they are perceived to be. If a developer for example selects a bug report and wants to know what caused it, issues like these will trip up the approaches. If on the other hand the developer identifies an individual change known to be the bug fix and wants to know the same thing, the approaches may be more useful. For many bugs, although not this one, the approaches could often identify the correct version if not for the presence of other changes in the file.

3.3.3.5 Number of changes

Eclipse Bug 55640 – "[FastView] Screen cheese after Fastview resize"

Firstly, this bug highlights a major problem with trying to interpret developer changes through bug and commit data. The commit comment for this fix, version 1.56 of `ViewPane.java` was "fix for bug 55640: views with custom titles were wasting space". The developer is fairly unequivocally claiming to have fixed a bug with the correct ID and the code is clearly in the same area as that described by the bug. However, the description of the commit and the description of the bug do not entirely match up and it is not clear whether they are definitely referring to the same thing. It is indeed possible that the developer used the wrong bug number when committing the fix or that the bug report was not a true description of what the problem actually was. Alternatively, this could simply be a problem of understanding caused by the analysis not being performed by a developer of the system.

Regardless, it was fairly straightforward to identify the origin of the bug fixed by the developer as version 1.33 of the file. To fix the bug, the developer deleted 32 lines of code[5] and modified another 2, spread throughout the class. However, these lines of code were last changed in a variety of different versions: 1 in version 1.22, 27 in version 1.33, 1 in version 1.34, 1 in version 1.38, 1 in version 1.48 and 3 in version 1.53. This results in the text approach identifying the correct answer, but with an additional 5 false positives. The dependence approach identifies the most recent of these changes, version 1.53, as the source and so gets the incorrect answer. The interesting point here is that the source of the bug was *not* the most recent change to the code (nor was it the oldest), but it was the version in which the majority of lines had last been changed.

[5] Actually, the developer 'commented out' the code, prepending each line with comment characters, so that the compiler ignores the line. As discussed earlier, this has the same effect on the program as if the code had been deleted, and so is treated identically.

3.3.3.6 Non-local dependences

Eclipse Bug 63753 – "Team -> Tag as Version w/ "Move tag if it already exists" option does not work"

Again this bug had a fairly straightforward fix, for which the text approach could correctly identify the origin. A single line:

```
if (confirmDialog.getReturnCode() == IDialogConstants.OK_ID) {
```

was changed to:

```
if (confirmDialog.getReturnCode() == IDialogConstants.YES_ID) {
```

Like previous bugs, if these two lines are interpreted as *not* mapping to one another between dependence graphs then the dependence approach would also return the correct answer. If however, as was assumed here, these lines do map, then the dependences *within* the method have not changed at all. The only change that has been made is a dependence on a static variable. This highlights additional possibilities for altering the dependence approach. Even if a full inter-procedural analysis is not feasible, extending the dependence analysis to take into account constants or field declarations may still improve the recall of the approaches. It is not clear however how many bugs would be affected this way and there may be bugs where doing so would cause the wrong version to be returned.

3.3.3.7 Added dependences

Eclipse Bug 50549 – "[Dialogs] Pref Page Workbench/Keys/Advanced does not use dialog font"

It is not at all clear whether the prioritisation of removed dependences over added dependences in the dependence approach is necessarily the correct technique. In the fix for this bug, version 1.66 of `KeysPreferencePage.java`, there were 2 lines removed, with corresponding control and data dependences removed. The most recent of these was introduced in version 1.63, which is what the dependence approach returned as the incorrect answer. However, there was also 1 line added. The corresponding control dependence, onto method entry, would have led the approach to return version 1.60, the actual origin of the bug.

3.3.4 Summary

A number of key points can be taken from the examples given here and earlier:

- The approaches generate a large number of incorrect results, in the form of both false negatives and false positives.
- For many bugs only one of the approaches gains the correct answer, or indeed any answer at all.
- The decisions taken by the dependence approach on graph matching, depth of analysis and priority of dependences have the potential to hugely affect its performance.
- Large fixes, and the number of versions between a bug and a fix, can reduce the effectiveness of the approaches.

- Unrelated changes have an impact on the results returned, but this can sometimes be to the approaches' benefit.
- Taking the most recent version as that which introduced the bug may not be the most effective technique.

3.4 Potential improvements

It is obvious from the sample, and from the examples highlighted above, that there are a number of areas in which the approaches could be improved. It is worth considering first why the approaches, or variations on them, should work at all. After all, it is not hard to construct theoretical examples of bugs for which neither approach would ever be successful, but these bugs do not appear to be frequent in the sample that was examined. Almost every bug fix involves either a textual change or a correction of an aberrant dependence. While the dependence approach makes use of a PDG, which examines only dependences within a method, it is also possible to construct a system dependence graph (SDG), incorporating all the dependences in a system. Using a theoretical extension of the dependence approach, with a richer variation of an SDG, even bugs which are fixed in other classes or by changes to inheritance hierarchies etc. could be seen as dependence changes. If the approaches were to consider *every* change, whether textual, added dependence or removed dependence and then to consider *every* previous version which affected these lines or dependences, it would seem very likely that the full set would include the bug origin. To do so would of course have huge costs, both in the large number of false positives and in the prohibitive time that would be required to run such a technique.

In a sense however, the approaches, and the decisions taken in their implementation, can be seen as attempts to prune this set of potential origins, in order to reduce these costs. Some of these decisions are:

- Choosing to examine only textual changes or only dependence changes
- Ignoring changes to whitespace or formatting
- Only returning the most recent version
- Ignoring added dependences if dependences have been removed

It is not clear however if these are the most appropriate ways to improve the approaches. This section will present a number of other possible improvements or changes that could be made.

3.4.1 False negatives

The idealised description of the approaches given previously is just that: an ideal. In reality, there are a large number of false negatives returned by each approach. In a very small minority of cases these occurred when a bug had multiple origins of which the approaches only identified some, but for 70 commits in Eclipse classed as *Single* or *Multiple* the text approach returned nothing at all. As stated earlier, this was usually due to fixes which only involved lines being

added. For the dependence approach the equivalent figure was 58 commits, where the fix usually contained no changed dependences.

The original authors proposed that in the cases where the dependence approach found no altered dependences it would fall back to using the text approach [SSR10]. However, this could equally be applied in the other direction, with the text approach given first preference. In fact this may well be preferable due to the extra computational effort required for the dependence approach. The original paper reported the dependence approach to take around 7.2 times as long as the text approach on average, in the worst case taking over 12 hours to analyse 129 fixing commits where the text approach took around 1 hour.

A technique of using both strategies may well be viable. The most common result was for the two approaches to return the same outcome. However, in a significant number of cases one approach identified the correct version while the other did not, as already discussed. Obviously, however, it is also possible for one approach to return the incorrect answer where the other returned nothing.

Table 3.17 shows the effect of returning the result of the second approach if the first approach returns nothing. For both Eclipse and Rachota the change to the dependence approach is the same: a slight increase in both true and false positives, with a corresponding rise in recall but drop in precision. The change for the text approach is more pronounced however, increasing both precision and recall. The difference in F_1-Score between the approaches has now reduced and in fact for Rachota applying the text approach first gives the highest value. Given the potential time saving and the similarity in effectiveness, using the text approach first could benefit some applications.

		TP	FP	FN	P	R	F
Eclipse	Text	91	220	100	0.29	0.48	0.36
	Text, Dependence	116	231	75	0.33	0.61	0.43
	Dependence	77	98	114	0.44	0.40	0.42
	Dependence, Text	96	127	95	0.43	0.50	0.46
Rachota	Text	89	39	38	0.70	0.70	0.70
	Text, Dependence	109	43	18	0.72	0.86	0.78
	Dependence	85	23	42	0.79	0.67	0.72
	Dependence, Text	95	34	32	0.74	0.75	0.74

Table 3.17: Combining approaches

There are perhaps more effective ways of combining the two approaches and of taking further evidence into account. As the earlier examples should have illustrated, there are often a number of decisions that need to be taken in each approach; there is not one correct way for every bug. Sometimes the most recent change is the one that introduced the bug, sometimes it is the first. Sometimes dependences are removed to fix the bugs, sometimes they are added. Sometimes type changes cause the bug, other times they are irrelevant. One potential way to approach handling this would be to aggregate this information and combine it in a probabilistic manner.

As a simplistic example, a fix in version 1.7 involving 10 lines of code, 2 of which were changed in version 1.5 and 8 in version 1.4, could imply that the bug was 80% likely to have been introduced in version 1.4. However, the fact that version 1.5 was more recent could suggest that

the bug is 66% likely to have originated in version 1.5. Multiplying these probabilities would say that the cause was version 1.4 with 65% probability. Obviously the examples given here are arbitrary, but the exact values could be determined empirically and additional evidence could also be taken into account.

In a way, some of the current decisions taken by the approaches can be seen as a blunt form of this: choosing to favour the most recent change is saying that there is a 100% probability it was in that version and 0% probability it was in any preceding version.

3.4.2 Inter-procedural analysis

Even after combining the two approaches, there are still 62 commits for which no result would be returned. In some cases this is desirable – where the file did not actually have a bug in it – but often this is due to the dependence approach not being capable of detecting the dependences that have been changed. However, there are a number of additions that could be made to it in order to increase the number of dependences it considers.

Starting from the basic intra-procedural approach used in the evaluation, a simple step would be to take into consideration dependences on fields, static values or constants. Slightly more complex would be to add in dependences on other methods within the same class, then more complex still to include other classes. Finally, other information could be added in until there is a full SDG as suggested earlier: synchronization, exception handling, inheritance hierarchies etc. Each of these layers adds on more information and brings the system closer to that described at the beginning of this section, both in terms of being more likely to find the origin and of having more false positives and a longer runtime. However, it may be that there is a level between the basic one used here and the full SDG which proves to be the most balanced of solutions.

Another way to increase the number of potential results discovered by the dependence approach is to consider *all* the dependences, rather than stopping at the one which had the most recent change. This would in effect make it more similar to the text approach, with similar downsides.

Another possible investigation is into the graph matching techniques used. The original source does not fully explain the technique used [SSR10], but it is suggested as being similar to that used in JDiff [AOH06]. In that technique, graphs are broken down into smaller subgraphs called *hammocks* and nodes are matched by their structural properties or textual equality – nodes which are textually similar but not exactly equal are not considered to match. However, there are many other graph or tree differencing techniques. In ChangeDistiller [FWPG07] for example, the labels of the nodes are split into n-grams[6]. The similarity between the sets of all the n-grams for each pair of nodes is then measured. Any two nodes where the similarity is above a certain threshold are considered to be the 'same' node. Whether a node in one version is matched to a node in a subsequent version or not has a major effect on how these techniques work, as two nodes which don't match can result in a lot of removed dependences. Using a technique which gives a more accurate decision on whether a line has been updated or instead entirely removed and a new one added could give fewer false positives. Conversely however, using a *less* accurate technique may actually increase the number of dependences altered, which

[6]Sequences of n consecutive characters from the label

if multiple dependences are taken into account may give the approach more chance of finding the correct answer, but at the cost of more false positives.

3.4.3 False positives

A significant proportion of the responses for the text approach were false positives. While most occurred when the approach could not identify the correct answer, a total of 41 false positives occurred where the approach identified the correct version along with one or more false positives, shown by the highlighted cells in Table 3.18. For example, there were 6 instances of commits which caused 1 true positive and 2 false positives, and 98 commits which caused 0 of each; nothing at all was returned for these commits. In addition, the bold cells show the large number of cases where more than one false positive occurred, for a total of 164 false positives.

FP	0	1	2	3	4	5	6
TP							
0	98	44	**25**	**12**	**7**	**3**	**1**
1	59	11	**6**	**4**	**0**	**1**	**0**
2	4	1	**0**	**0**	**0**	**0**	**0**

Table 3.18: Commits classified by number of true and false positives for text approach (Eclipse)

One technique to reduce the number of these responses would be to return a single version, similar to the dependence approach. Doing so would reduce the maximum possible number of false positives per commit to one and would hopefully also result in discarding the false positive in favour of the true positive. Two ways to do so are to return the most recent version in which anything changed or to return the version in which the majority of the lines last changed.

Table 3.19 shows the effect of such a change. As shown the false positives have decreased significantly. Unfortunately, the true positives have also decreased, as the approach no longer returns some of the correct versions it previously would have. This is especially true for bugs classed as *Multiple*, where it is no longer possible to correctly identify all of the origins. However, selecting the version in which most lines were last changed does significantly increase the precision at the cost of a smaller decrease in recall. Selecting the most recent version is similar but for a lesser benefit and larger downside.

	TP	FP	FN	P	R	F
Text Approach (All versions)	91	220	100	0.29	0.48	0.36
Majority of lines	75	103	116	0.42	0.39	0.41
Most recent	66	112	125	0.37	0.35	0.36

Table 3.19: Returning single version for text approach (Eclipse)

3.4.4 Unrelated changes

In total there were 34 fixes containing unrelated changes, of which 20 fixed multiple different bugs. An automated tool could potentially ignore these commits by checking the commit

message for multiple bug IDs. Unfortunately, 13 of these commits were for a bug that was classed as *Unclear* and the remaining sample did not allow conclusions to be drawn about such a change.

One situation where this could be improved is where multiple commits are made to fix a bug, often because the first attempt was incorrect. One improvement would be to ignore changes made in versions that were linked to the bug other than the latest. This is in effect similar to the suggested improvement to remove all versions after the bug was raised as possible origins [KZPW06].

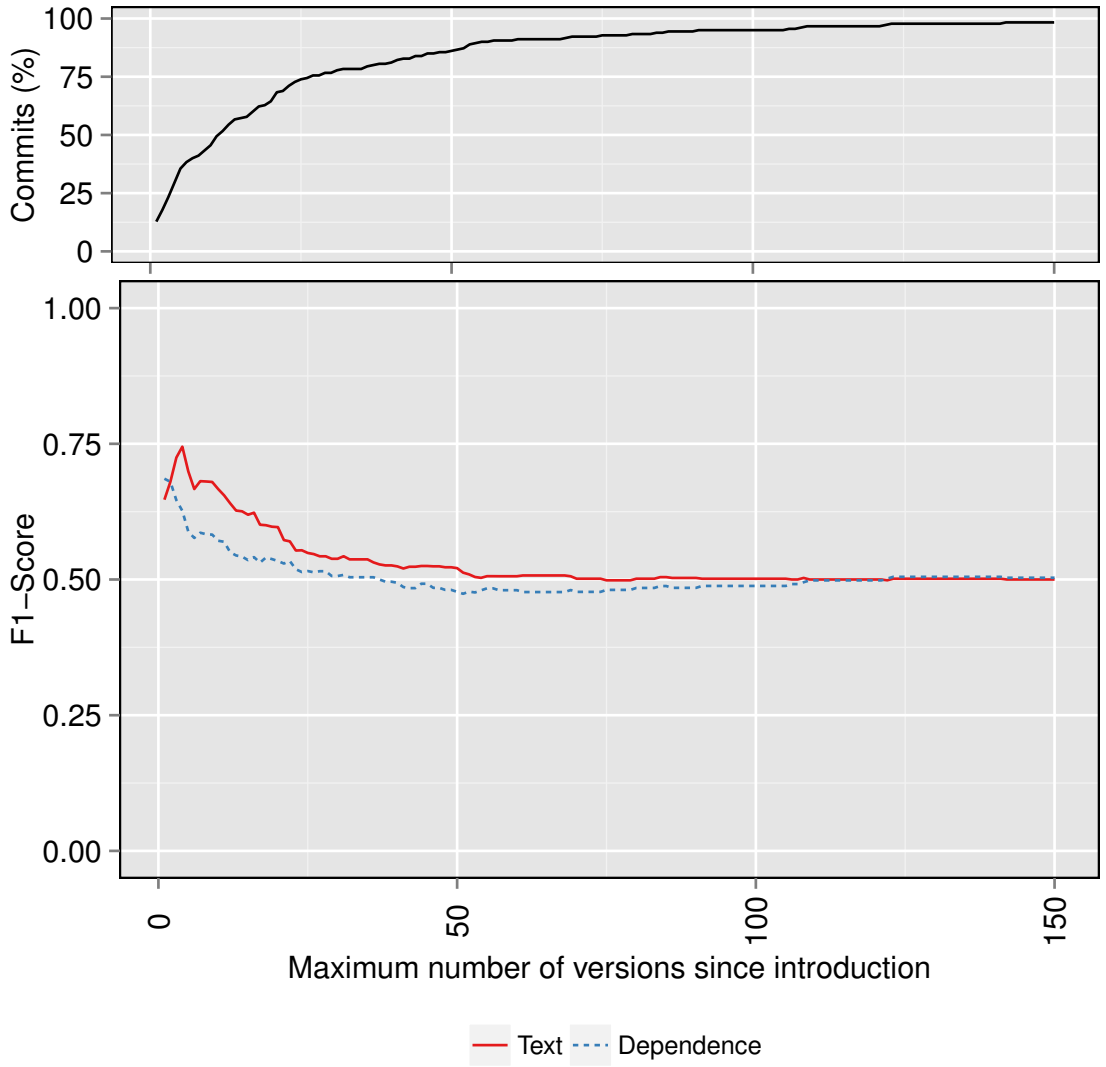


Figure 3.20: Top: Percentage of fixing commits within given number of revisions after origin; Bottom: F_1-Score when considering only those commits

Possibly due to the increased chance of unrelated changes, the approaches tended to get less effective as the time between the bug being introduced and being fixed increased. Figure 3.20 shows the effects on F_1-Score if fixes more than a given number of versions after the origin were to be ignored. Larger values have been omitted from the chart as the values remain largely stable, but the maximum difference between a bug being introduced and being found in Eclipse

was 267 versions. Note that these figures only include bugs for which an origin actually existed so will not correspond with those given earlier in Table 3.15.

Figure 3.20 also illustrates that bugs were often fixed shortly after being introduced. For Eclipse, 12.8% of bugs were fixed 1 version after the bug was introduced with 50% of bugs being fixed within 12 versions.

Another, more complex, way of removing false positives would be to ignore changes to code that had no semantic effects. A number of common patterns were identified by Jung et al. [JOY09], from simple renames or deletions of lines with no effect, to deep semantic analysis. Some of these are already carried out, whilst others could perhaps be easily achieved by using abstract syntax trees rather than straight textual differencing, but detection of other patterns would be challenging. While Jung et al. only considered ignoring these changes in the fix itself, it would also be possible to apply such techniques to changes in intervening versions.

3.4.5 Large commits

Eclipse Bug 49561 showed a large number of false positives being returned when there were many files involved in the bug fix. This appeared to be common: often bug fixes which involved many files had a large number of unrelated changes, leading to an increase in false positives. One proposal is to ignore fixes that change a large number of files [KZPW06]. The number of bugs of each size in Eclipse is shown in Table 3.21.

	Number of bugs
1	55
2	16
3	7
4	4
5	5
6	1
7	3
10	2
13	1
21	1
22	1
24	1

Table 3.21: Number of files per bug (Eclipse)

Figure 3.22 illustrates the F_1-Score if bugs with more than the given number of commits were to be ignored. Similar trends are present for precision and recall. As seen here, the vast majority of bugs were small and the scores are much better for smaller commits. For the few larger bugs the scores decrease. The same was not necessarily true of Rachota; there the largest bugs were smaller and the scores stayed largely constant.

Given the results for Eclipse, ignoring bugs with more than a certain number of commits may increase effectiveness without reducing applicability significantly. This might be appropriate for some use cases but further study would be needed to determine a threshold and it is likely this may vary by project. Furthermore, larger bugs are often more complex, and it is precisely complex bugs that developers would find these approaches more desirable for. It is possible

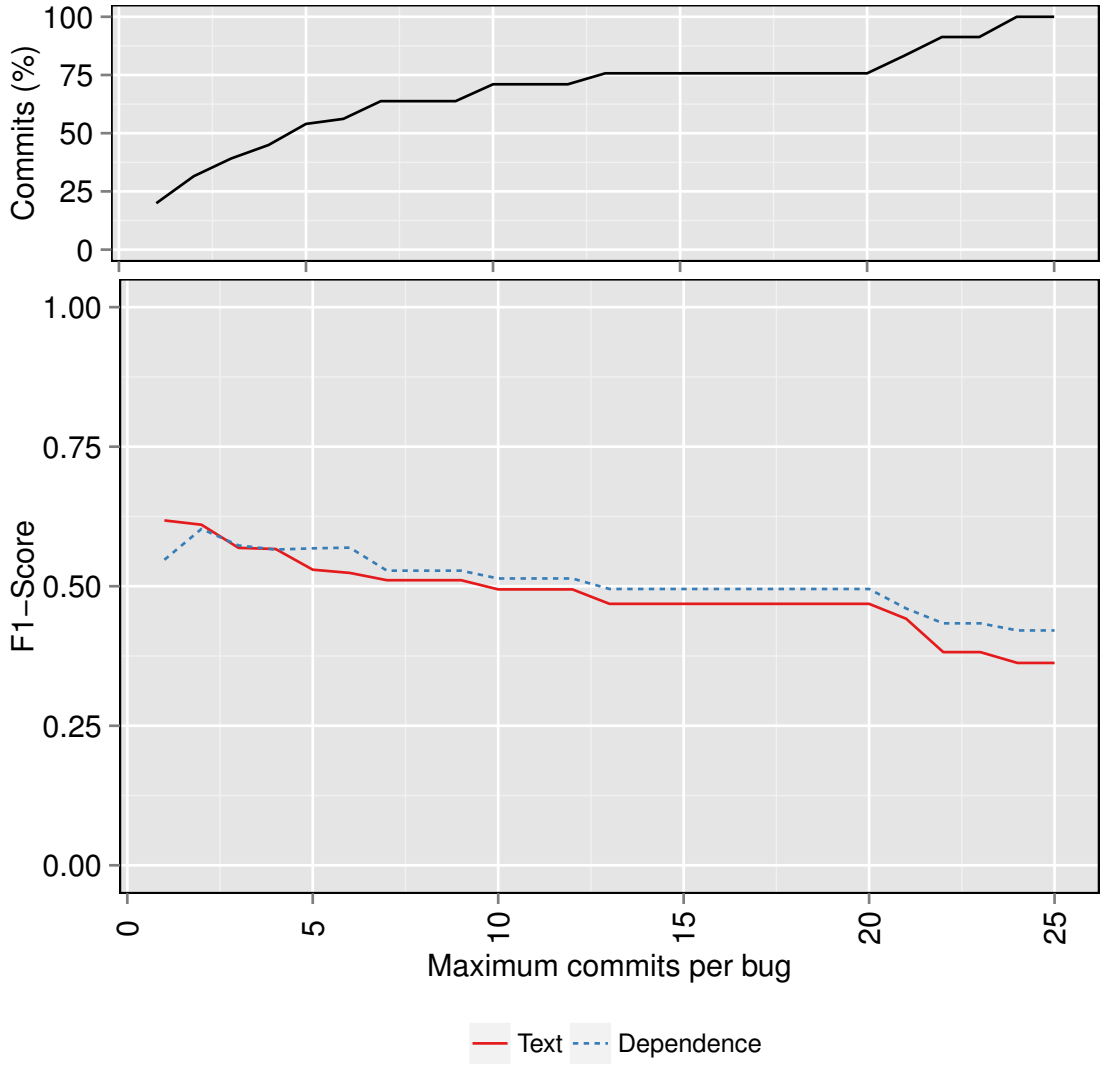


Figure 3.22: Top: Percentage of fixing commits where bug involved fewer than given number of commits; Bottom: F_1-Score when considering only those commits

that ignoring these bugs may improve the scores for the approaches, while rendering them less useful in practice.

3.5 Threats to validity

There are are a number of threats to the reliability and generalisability of the findings from this evaluation. These include the manual application of both the text and dependence approaches, challenges in interpreting the dependence approach, researcher consistency, the accurate determination of bug origins, and potential limitations caused by the particular software systems and bugs selected for analysis.

Both the text and dependence approaches were manually applied as described in Section 3.2.2. As well as having the potential for human error in their application there were also occasional difficulties in determining precisely how they would perform when automated, particularly with regard to the dependence approach. As discussed earlier, there are different possible interpretations of data and control dependence, including issues associated with intra- and inter-procedural dependences, type changes and changes within conditional statements. Furthermore, the handling of constructs such as fields, `synchronized` blocks and `try-catch` blocks were not discussed in the original work. To maintain consistency, scenarios involving such constructs were noted as encountered and then revisited as a group at the end of analysis.

The analysis of Eclipse and Rachota was carried out by one researcher. To try to ensure clear and consistent application of the research procedure, an additional two researchers also independently applied the procedure to eight jEdit bugs as detailed in Section 3.3.1. The researchers then met and performed a detailed group walkthrough of the analysis of each of these bugs to clarify all the issues raised and to ensure consistency of application. The correct identification of the origin of each bug is also a potential threat, especially since none of the researchers were developers of the software systems that were being studied. This threat could not be avoided altogether, but was lessened by noting related and similar commits and again revisiting them at the end of analysis. Also, as described in Section 3.2.2, where possible, commit logs and bug report comments were used to support identification of bug origins.

For Eclipse and Rachota the analysis only ever finds a bug origin in the file where the fix was applied. However, as discussed, the origin to a bug sometimes lay elsewhere. This limitation is partially due to the RCS, CVS, used in these studies which considers each file in a commit as a separate transaction. This threat could partially be addressed by further studying projects that use other RCSs which assign revision numbers to the entire project, such as with the use of Subversion in the jEdit study, or to use techniques that reconstruct project versions [ZWDZ05].

Particular threats to the generalisability of the results are associated with the systems studied, the relatively small number of bugs analysed and the particular bugs that were selected for analysis. The manual analysis of bug origins and application of both approaches is quite time-consuming – about 30–60 minutes per bug on average. The only criteria used for selecting bugs to be analysed was that they could be linked via a bug identifier to a fixing commit, and so these may not be representative of all bugs in the three systems [BBA+09]. Similarly, the three

software systems studied Eclipse, Rachota and jEdit, and their bugs, may not be representative of all software systems.

In considering the above threats it is important to note that the aim of this study was to gain a deeper insight into how the text and dependence approaches may actually perform when applied to real software systems. The goal has not been to investigate whether one approach is 'better' than the other but to identify their apparent strengths and weaknesses, the key issues that arise with their use in practice and to determine the extent which bug origins might be automatically identified. Some confidence in the findings can be derived from the fact that same key issues repeatedly arose across the wide variety of bugs that were studied.

3.6 Conclusions

The aim of the study described in this chapter was to investigate the practical application of two approaches for discovering the origins of bugs, the first point in the bug life cycle. The study was performed by manually simulating both of the approaches on sets of bugs from three case studies: Eclipse, Rachota and jEdit. It investigated the origins of 174 bugs and analysed the potential of the text and dependence approaches to correctly identify these origins. The quantitative evaluation based on Eclipse and Rachota found that both approaches performed somewhat as intended, identifying the bug origin with a precision in the range 29%–79% and a recall in the range 40%–70%. The evidence suggests that the text approach is more likely to identify the correct origin than the dependence approach but at the cost of reduced precision, due to it identifying *all* possible origins while the dependence approach seeks to identify a *single* origin.

Other key lessons derived include:

- The accuracy of using both approaches together was at least as good as, and mostly better than, using either on its own.
- Both approaches are likely to provide inaccurate results when other 'opportunistic' changes are made coincident to the fix. The text approach can accommodate this, at a cost to precision, because, as currently specified, it returns all possible origins.
- Unrelated changes made between the origin and the fix often caused incorrect results – this is a particularly difficult issue for both approaches.
- Both approaches became less accurate as more versions passed between the bug origin and the fix, though, in the systems studied, most bugs were fixed soon after their origin.
- Most bugs required changes to only a few files; bugs which required many changes could often not be resolved, particularly those where the fixes involved multiple files.
- Bug fixes that modify the context of the origin rather than the same lines as the origin are more likely to mislead the text approach than the dependence approach.
- Overall, the approaches generate a considerable number of both false positives and false negatives, and techniques built on the approaches may be inaccurate due to this.

There are a number of threats to these findings mainly concerned with the manual simulation of the approaches by one individual applied to only three case studies and a involving a relatively small number of bugs. However, the fact that the key findings listed above were repeatedly discovered across different bugs and the different systems, supported by a documented, repeatable research procedure should provide confidence in these results. The level of precision and recall discovered for the systems studied would seem reasonable for some applications, especially if it is sufficient to return a relatively small set of files which is likely to contain the bug origin.

The origin of a bug is an important point in its life cycle, and identifying it is useful to develop tools preventing the introduction of bugs. It is inconceivable, however, that it will ever be possible to prevent the introduction of bugs completely – there will always be bugs which arise and need to be fixed. As well as the bug origin, there are additional points in the life cycle of a bug that have an effect on how long it takes them to be fixed, and on the costs of doing so. This chapter has also shown the complexities of studying the life cycles of bugs, and how it can be complicated by influences such as inaccurate information provided by users or developers, unrelated changes being made to code, and the length of the bug life cycle. These sort of issues will be repeated in studies of other parts of the bug life cycle later in this thesis. In particular, the work in this chapter has shown that techniques examining bugs can be influenced by the type of bug and the quality of information available about it. This information is provided when the bug is reported, which will be the focus of the next chapter.

4 INFORMATION IN BUG REPORTS

The techniques in Chapter 3 attempt to identify the earliest point in the bug life cycle. Unfortunately for developers however, this is not actually the point at which a bug is discovered. Instead, a bug, as defined in this thesis, is not found until after the code has been released, and the problem is reported by a user of the system. Developers cannot start fixing bugs until this has happened and they know about the bugs. How a bug is found, and how it is reported, can have a big impact on the likelihood of the bug actually being fixed.

Figure 4.1 shows Mozilla Bug 218037. This bug report is detailed, and contains clear information about what happened and what the user expected to happen. It also contains an extensive example to assist developers to reproduce the bug, and information about the user's environment when they encountered the bug. As long as this information is accurate, this sort of bug report should be reasonably straightforward for a developer to fix[1].

Unfortunately, not all bug reports are quite so well-written as Mozilla Bug 218037. A stark contrast can be seen by examining the bug report for Apache Bug 25091, shown in Figure 4.2. This bug report is very minimal and poorly-written. Indeed, it is not clear whether this is a bug or a user error. Either way, the way the bug report is written takes valuable time away from the developers, although the developer here does still make an effort to help the user, despite the lack of information.

This chapter will investigate how users report bugs in systems. In particular, it will look at what information developers want in bug reports compared to what information users actually provide, how and when users provide the information, how this affects the outcome of the bug. It will also examine in detail the suitability of each piece of information for use in automatic bug localisation, the topic of Chapter 5. In order to do so, it will examine 1600 bugs in 4 open-source projects, examining the quantity and quality of information provided by users when they are reporting bugs.

4.1 What information do developers want to help them fix bugs?

One of the overall aims of examining the information provided in bug reports is to identify ways in which the bug fixing process can be improved. One potential first step in doing so is to examine what information developers themselves want when fixing bugs. Bettenburg et al. [BJS+08] surveyed 156 developers of Apache, Eclipse and Mozilla about what sections of a bug

[1] Although this particular bug had already been reported, and so was marked as a duplicate bug

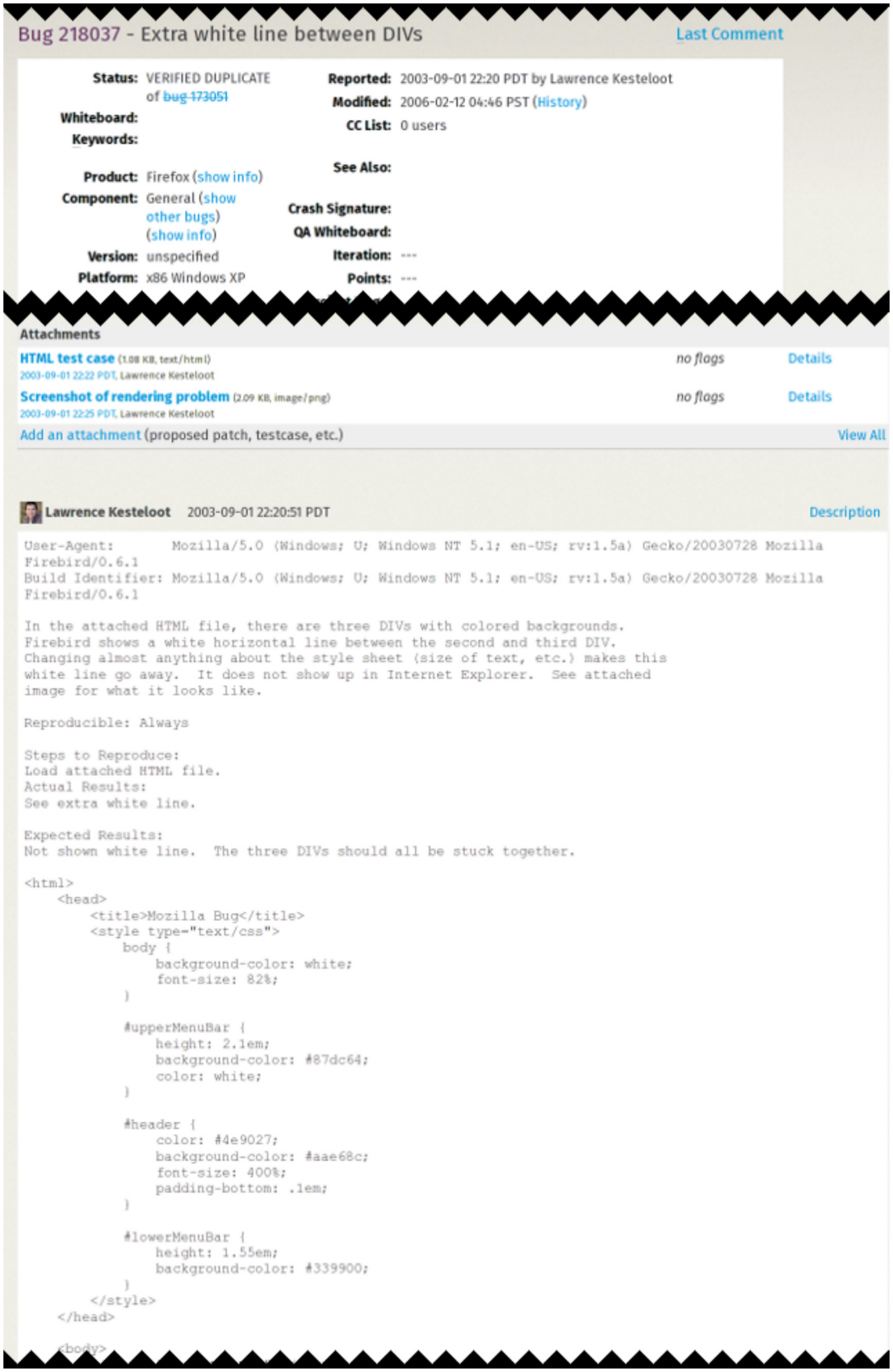


Figure 4.1: Mozilla Bug 218037 (Cropped)

Figure 4.2: Apache Bug 25091 (Cropped)

report they found most useful when fixing bugs. In order, the most important features were reported as:

1. ***Steps to reproduce:*** A clear set of instructions that the developer can use to reproduce the bug on their own machine
2. ***Stack traces:*** A stack trace produced by the application, most often when the bug is reporting a crash in the application
3. ***Test cases:*** One or more test cases that the developer can use to determine when they have fixed the bug
4. ***Observed behaviour:*** What the user saw happen in the application as a result of the bug
5. ***Screenshots:*** A screenshot of the application while the bug is occurring
6. ***Expected behaviour:*** What the user expected to happen, usually contrasted with *Observed behaviour*
7. ***Code examples:*** An example of some code which can cause the bug
8. ***Summary:*** A short (usually one-sentence) summary of the bug
9. ***Version:*** What version of the application the user was using at the time of the error
10. ***Error reports:*** An error report produced by the application as the bug occurred

Similar results were found in a survey of Microsoft developers which was carried out to determine which features most influenced whether or not a bug would eventually be fixed [GZNM10].

Unfortunately, the information developers want is not always provided by users, and various research has reported that *Screenshots* and *Stack traces* [BPZ08], patches (which are proposed solutions to the bug) [HW07] and *Code examples* [BJS+08] have all been found to be relatively uncommon in particular projects. This behaviour does not go unnoticed by developers; one survey suggested only around half feel that bug reports are nearly always complete [KT04]. Additionally, Breu et al. [BPSZ10] looked at the questions asked during the resolution of bug reports, and found a significant proportion of these were related to missing or inaccurate information. An in-depth study of bugs at Microsoft, along with a more general survey of employees, found that the history of bugs could not accurately be recreated through electronic sources alone [AV09]. The majority of bug reports were missing information, and a significant proportion contained inaccurate information. Other researchers have found similar results: Ko and Chilana [KC10] found that the majority of bug reports by normal users of BTSs contributed no useful information. This lack of quality is not necessarily surprising, as similar results have been found for requirements documentation [OHK07, SRC08], which are similar to bug reports in that they are written in natural language and not necessarily by developers.

Some studies have investigated which attributes can be used to help predict the fix time of the bug [HW07], or the developers who should be responsible for fixing it [ČM04]. There have also been attempts at determining how relevant *Stack traces* are to fixing bugs [SBP10], and to whether *Error reports* can be used to identify the causes of bugs [YMX+10]. However, no study has attempted to show how commonly these various features occur in bug reports.

4.2 Investigation

To investigate how often the features desired by developers were actually provided in bug reports, I examined 1600 bugs across 4 projects and detailed how often users provide each part of a bug report. The study also examined various issues surrounding the relationships between different aspects of a bug report, the variance between projects and the potential for automatic extraction of data from bug reports.

4.2.1 Subjects

The four projects involved in the investigation were:

Eclipse: An open-source IDE and application framework, written in Java
Firefox: An open-source browser, written mainly in C++
Apache HTTP: An open-source HTTP server, written in C
Facebook [Fac] **Application Programming Interface (API):** Proprietary APIs for a social network, available in several languages

A number of scripts were developed to extract a random subset of 400 bugs from each of the repositories. These projects represent a variety of different uses and languages. In addition they attract user populations of differing size and technical experience. Whilst three of the projects are open-source and one is closed-source, each of the projects in question makes use of an open bug repository. Anyone can report and comment on bugs about the project, and take part in conversations about the bugs with developers.

It should be noted that the Facebook BTS is not intended for reporting issues found in the Facebook website or various client applications, with which the vast majority of users are aware. It is instead intended specifically for issues to be reported by the much smaller number of developers who create applications that use Facebook's API. The BTS is not promoted to normal Facebook users, only accessible through a site intended for use by developers. Nevertheless, during the evaluation it was found that a noticeable number of bugs reported were by normal Facebook users for issues they are having with their account or the website. These bugs are generally marked quickly as invalid, with the users redirected to a more appropriate source of help. All of these bugs are still included in this analysis, along with all bugs declared invalid for other reasons. The relationship between the information included and whether bugs are considered valid or not will be explored in Section 4.6.

4.2.2 Features

As discussed previously, while the exact implementation varies between systems, bug reports usually consist of a number of fields. Some of these fields, such as the platform or version, can only take a limited number of values. These can be seen in Figure 4.3 for example. Information can be extracted from these fields in a relatively straightforward manner using automated techniques. However, there are a number of other pieces of information which are desirable in a bug report that are not contained within these fields, such as the *Expected behaviour* or *Steps to reproduce*. Unfortunately, in general BTSs have no specific support for any of these features, and

they are usually provided (if indeed they are provided at all) as part of the title, description or comments, all of which are unstructured plain text, or as generic attachments. As such, identifying how often they are provided is not straightforward; there are no simple techniques for automatically extracting them.

Figure 4.3: Fields from Mozilla Bug 218037 (Detail from Figure 4.1)

For each bug in the sample, the basic structured information available was recorded. The unstructured information for each bug report was then manually examined to identify whether any of the following features, based on those listed in Section 4.1, were present in the bug report, and how they were reported. While amongst some projects it was common to explicitly label some features, e.g. "Expected results: ..." (as shown in Figure 4.4), the investigation did not rely on these, although this is investigated further in Section 4.7. A description of what the reporter expected to happen was sufficient, and similar procedures were followed for other features. Since this obviously leads to the evaluation being somewhat subjective, a number of judgement calls had to be made. The most common and important decisions are detailed alongside the relevant feature below, but this is not a complete list.

Figure 4.4: Description from Mozilla Bug 218037 (Detail from Figure 4.1)

The features examined were:

***Observed behaviour* (Obs):** In general, a statement which simply says some particular functionality of the program 'does not work' or something similar was not sufficient to consider *Observed behaviour* as present. Users had to provide at least a minimal amount of detail about what happened or what they saw.

***Expected behaviour* (Exp):** Similarly, while a statement such as 'I received the following error' is sufficient for *Observed behaviour*, it was not sufficient for *Expected behaviour*. The user had to make clear that they expected not to receive any error (or indeed, that they were expecting an error but that the error they received was incorrect). However, requests for new functionality or patch descriptions were often made in an imperative form and this was considered as sufficient for *Expected behaviour*.

***Steps to reproduce* (Rep):** Enough detail had to be provided to allow the bug to be reproduced on another machine, although this did not have to be explicitly given as a numbered sequence of instructions.

***Error reports* (Err):** *Error reports* included stack traces, and any time when a quoted error message was provided, as well as more detailed logs or Java core dumps. Apache logs with error or critical notices were considered *Error reports* as were other levels of logs

if they clearly contained an error message. *Error reports* were counted even if the text appeared only in a screenshot. This would clearly assist an actual developer in locating the cause of a bug, although it would of course be difficult for an automated system to extract the same value from the message.

***Stack traces* (Sta):** *Stack traces* did not have to be from an exception. For example, backtraces obtained through gdb [GDB] were sufficient. The output from strace [Str], truss [Tru] and similar tools were not considered to be *Stack traces*, as these only contained the system calls being made, not the application code itself. In many ways, *Stack traces* is a more specific form of *Error reports*, which is itself a more specific form of *Observed behaviour*.

***Screenshots* (Scr):** Only *Screenshots* of the application in question were counted, not of other applications. Screencasts and other videos of the application were considered to be *Screenshots*. Mockups and screenshots of proposed changes were not counted as *Screenshots*.

***Code examples* (Cod):** *Code examples* were not required to be either complete or minimal. In many cases, users provided links to webpages or Facebook applications which reproduced the bug in questions. These were considered *Code examples*. Apache configuration directives and files were considered *Code examples*. Many parts of the Facebook API could be exercised by simple one-line RESTful URLs. These were also considered *Code examples*.

***Test cases* (Tes):** *Test cases* had to be self-contained and automatic. They did not necessarily have to fit into the existing test suite. No manual action should be needed after setup, but some interpretation of results was considered acceptable e.g. a test that prints 'Success' or 'Failure'.

***Build information* (Bui):** This field records not just the release version of code being used, but also records if the user was using a 'between-release' version of the code, as well as what options were set when they built the software. *Build information* is specified differently for each project:

Apache: Options passed to `./configure`, the script used to build the Apache software.

Eclipse: A milestone or RC version number, or build date.

Firefox: Most commonly a user-agent like string, or a full version number.

Facebook: No build information was given. Unlike the other projects, Facebook was not installed on users' machines. Instead, only one version of it was ever running at a time, on the Facebook servers, and users had no knowledge of the settings used to build the system.

***Application Code* (App):** This was used to record any time when particular lines or methods of the project were identified as being the source or solution of a problem, including patches. Note that this is code from the source of the applicable project, as opposed to *Code examples* which is code written by users that is a client of the project code.

The assessment did not take into consideration whether the information for given features was accurate, nor determine its quality. For example, if a bug report contained instructions to reproduce an issue, then this was marked as containing *Steps to reproduce*, even if a developer pointed out that these steps were not sufficient or correct.

4.2.3 Location of features

For each feature, if it was present in the bug report then its presence was recorded as one of the following values, depending on where in the report it was found:

D: In the description
T: In the title or summary
A: In an attachment
L: In an external resource linked to in the description
C: In a later comment
AC: In an attachment added in a comment
LC: In an external resource linked to in a comment

Only the first of the categories above to match was used, e.g. if the build information was found in the description then this was recorded as **D**, whether or not the information was also present in the title or comments. This roughly corresponds to what is most useful for developers: all the information being given in the bug description. Information which is given in attachments, external resources or later comments requires more work from developers to utilise successfully.

L and **LC** represent any information which is provided in the bug report but which is stored outside the BTS, such as user forums, image hosting sites, or Talkback IDs (a separate system used by Firefox for reporting crash information). Links to such information could often break at some point after the bug has been created, leaving the information unavailable.

4.3 What information do users provide?

Figure 4.5 shows the number of bugs that contain each feature at some point in the bug report. As shown, *Observed behaviour* is found in the vast majority of reports, with *Expected behaviour* in more than half and *Steps to reproduce* in slightly less. Barring some highly unusual bugs, it *should* be possible to provide all these fields in any bug report, so the fact that they are not provided more often could be seen as disappointing. This is especially true for *Steps to reproduce*, as developers consider this the single most important feature needed when handling a bug report [BJS+08].

The other features are found in far fewer reports, with *Screenshots*, *Stack traces* and *Test cases* all being found in less than 10% of bug reports. This is in line with results found by other researchers [BJS+08, BPZ08]. These features are not necessarily appropriate for all bug reports however. For example, we do not know how many bugs without *Stack traces* did not generate a stack trace, and how many did generate a stack trace but the user chose not to report it. The low numbers of these features suggest however that bug fixing tools or techniques that rely on the presence of the features are not likely to be widely applicable.

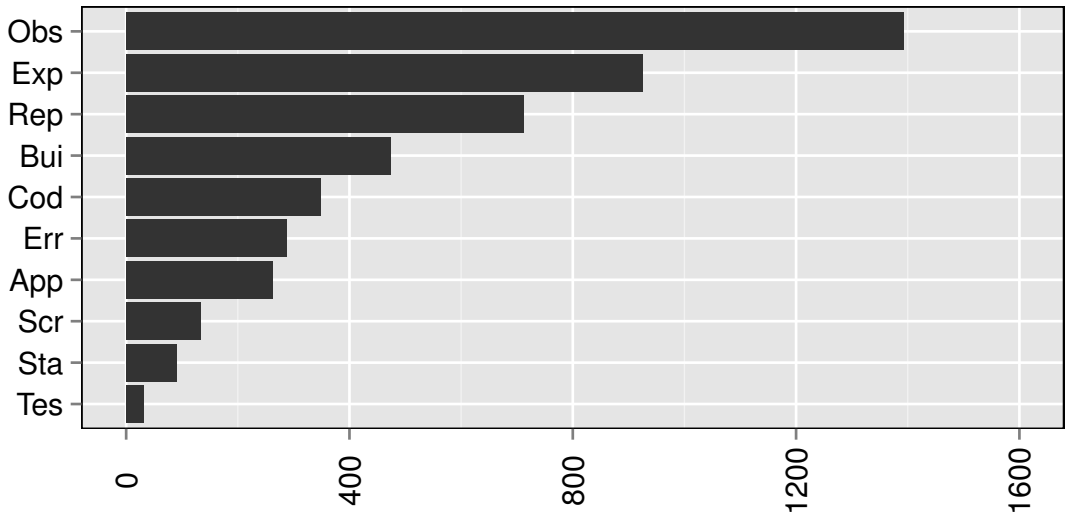


Figure 4.5: Number of bugs in which each feature was present

4.3.1 Differences between projects

As with many other measures in software engineering, it seems likely that the information provided in bug reports will vary between projects. Figure 4.6 shows the number of bug reports with each feature by project. It is clear from this that for some features, they are more common for some projects than for others.

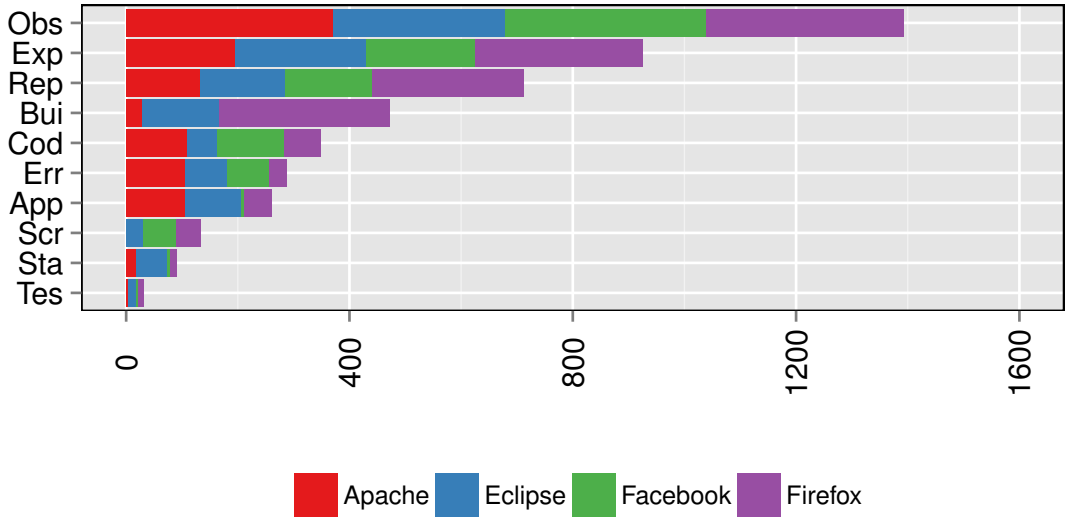


Figure 4.6: Number of bugs in which each feature was present, by project

The differences between projects can be confirmed by examining the tables in Table 4.7, which show for each feature the number of bugs in each project where the feature is present, and the mean value over all projects. If there was no difference between projects, we would expect the values for each project to be close to the mean value i.e. the distribution for each project would be the same. This does not appear to be the case. χ^2-tests for independence can be carried out to test this, checking whether the presence or non-presence of each feature is dependent on

	Provided	Not Provided
Apache	371	29
Eclipse	308	92
Facebook	360	40
Firefox	354	46
Mean	348.25	51.75
p-value	<0.01	

(a) Observed behaviour

	Provided	Not Provided
Apache	196	204
Eclipse	235	165
Facebook	194	206
Firefox	300	100
Mean	231.25	168.75
p-value	<0.01	

(b) Expected behaviour

	Provided	Not Provided
Apache	133	267
Eclipse	152	248
Facebook	155	245
Firefox	272	128
Mean	178	222
p-value	<0.01	

(c) Steps to reproduce

	Provided	Not Provided
Apache	29	371
Eclipse	138	262
Facebook	0	400
Firefox	306	94
Mean	118.25	281.75
p-value	<0.01	

(d) Build information

	Provided	Not Provided
Apache	110	290
Eclipse	54	346
Facebook	119	281
Firefox	66	334
Mean	87.25	312.75
p-value	<0.01	

(e) Code examples

	Provided	Not Provided
Apache	106	294
Eclipse	76	324
Facebook	74	326
Firefox	31	369
Mean	71.75	328.25
p-value	<0.01	

(f) Error reports

	Provided	Not Provided
Apache	106	294
Eclipse	99	301
Facebook	7	393
Firefox	50	350
Mean	65.5	334.5
p-value	<0.01	

(g) App Code

	Provided	Not Provided
Apache	1	399
Eclipse	29	371
Facebook	60	340
Firefox	44	356
Mean	33.5	366.5
p-value	<0.01	

(h) Screenshots

	Provided	Not Provided
Apache	18	382
Eclipse	56	344
Facebook	5	395
Firefox	11	389
Mean	22.5	377.5
p-value	<0.01	

(i) Stack trace

	Provided	Not Provided
Apache	3	397
Eclipse	15	385
Facebook	3	397
Firefox	10	390
Mean	7.75	392.25
p-value	<0.01	

(j) Test Cases

Table 4.7: Occurrences of each feature by project

the project. For each feature, p-values for χ^2-tests on these contingency tables are all less than 0.01. This suggests that there is indeed a significant difference between the mean values and the value for each project, showing that there is a relationship between the project and the presence of each feature.

Firefox is responsible for a significant proportion of the bugs with *Build information*, *Expected behaviour* and *Steps to reproduce*. When we consider the users of the four projects, this is actually somewhat surprising. Apache, Eclipse and Facebook API are all mainly used by people who are developers, just not necessarily of the software in question (although there will be some overlap). Firefox bug reporters, however, are more likely to be ordinary users, who could be expected to be less knowledgeable about the bug reporting process, and the requirements for a good bug report.

One possible reason for that may be that Firefox has an optional interface that specifically prompts its users for this information (along with *Observed behaviour*) in a separate field for each, while the other projects simply expect the reporter to include them in the description (although they may have documentation which tells reporters that all of them are expected). This suggests that such an interface could boost the numbers of bug reports which contain these fields if a similar interface was adopted by other projects.

Unsurprisingly given the nature of the application, Apache bugs contained virtually no *Screenshots* and very few *Test cases* or *Stack traces*. *Build information* is also less common, probably since the only information applicable was the arguments passed to `./configure`, which is only appropriate for users who have built the system themselves, as opposed to downloading a pre-built system. *Error reports*, and in particular error logs, are common however.

As no build identifiers are available to users, Facebook bugs contain no *Build information*. Also, likely due to the fact that the source code is not available to users, very few bug reports contain information about where in the code the bug is likely to be, and automated *Test cases* and *Stack traces* are also rare. Perhaps as a consequence of this, *Screenshots* and *Code examples*, often in the form of links to Facebook applications, are more common.

There are perhaps surprisingly few *Code examples* provided in Eclipse, given that it is after all an IDE. However *Stack traces* are more common than in other projects. This is likely due to the fact that they are more prominent within the application. Most errors within Eclipse will produce stack traces either in a dialog to users, in the error log, or in the Java core dump that may be produced by a crash. This is in contrast to Apache or Firefox, where the user must usually go to some manual effort, for example using gdb, to produce a stack trace for an error.

These differences between projects are important for multiple reasons. First, it highlights how some projects attract more bug reports with a particular feature than others, and it may be useful for the other projects to examine the reasons for this, in order to improve their own bug reporting process. Secondly, in the context of extracting information for automated bug localisation, it is clear that the use of a particular feature may be more or less suitable for a particular project – techniques which utilise *Build information* may be suitable for Firefox but next to useless for Apache. There is unlikely to be one solution equally applicable to all projects.

4.4 How do users provide information?

While the majority of information about a bug is contained in the bug report itself, users would also sometimes provide information as attachments, or even as external links to information held elsewhere. For example, in some Firefox bugs, users would provide sample code by linking to a particular website. However, these websites would often be quite complex, consisting of an entire working site which may contain many irrelevant factors for developers, rather than a minimal working example that would be sufficient for them to reproduce the bug. In addition, the content found at external sites is not fixed, and may have changed since the bug was reported, or may in fact no longer exist at all. In Facebook Bug 3427 for example, even the *Steps to reproduce* was provided as an external link, but this link was now no longer active. This has implications for both developers and researchers wishing to make use of the information.

Figure 4.8 shows how the information required is provided: in the main body of the report, as an attachment to the report, or in an external location that is then linked to in the report. As is to be expected, most information is directly provided within the bug report itself, but for some features the pattern is different. *Application Code* and *Test cases* are more likely to be attached to a bug report than provided in the text. This is due to the fact that these are normally patches provided by the developers. Any automated bug localisation system wishing to use these features would have to be capable of handling attachments as well as the text of the bug report.

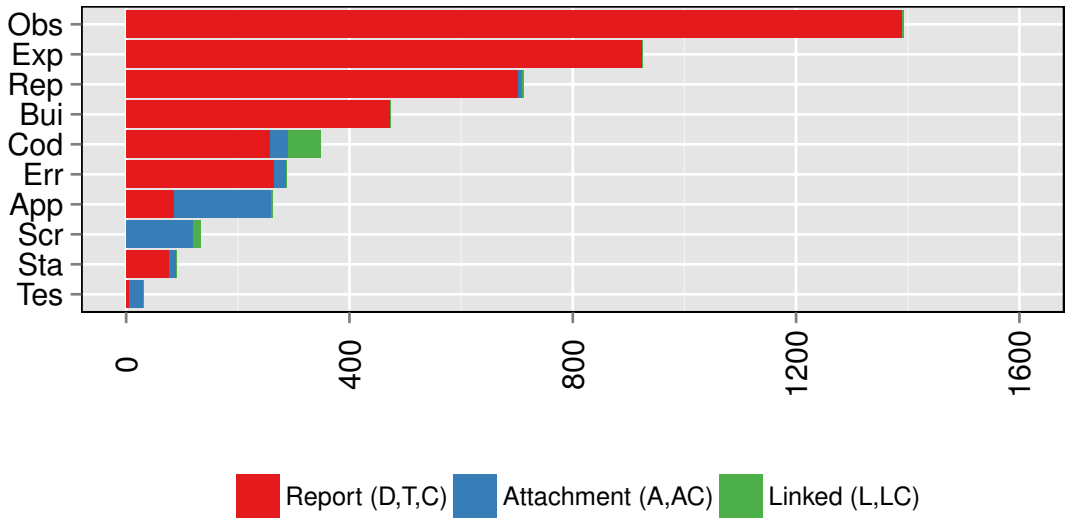


Figure 4.8: Number of bug reports where feature is provided, by location of provision

Unsurprisingly, since only text is allowed, *Screenshots* were never provided in the bug report itself. More surprising though is the proportion that were actually provided through an external site, such as a specialised image hosting site. The reasons for this are not clear, given Bugzilla's ability to provide the same functionality. As discussed earlier, the use of link to external data was most common for *Code examples*, where in particular Firefox or Facebook bug reports would contain links to external websites or Facebook applications.

There are also a reasonable number of bugs which provide either *Code examples*, *Application Code*, *Error reports* or *Stack traces* within the main body of the report. This has important implications when parsing the contents of bug reports automatically. Although written in natural language, the bug description and comments cannot simply be treated as unstructured text. Instead, it will often contain a mixture of structured and unstructured content, and any system, either for bug localisation or any other purpose, which wishes to handle these features will have to be able to extract them from the description body – they cannot rely on the information being provided as an attachment.

4.5 When do users provide information?

Information from users can either be provided when the bug is initially reported, or in the comments section of the bug at a later time. For a developer, it would of course be most useful if all the information was included when the bug is first reported. This is because having to ask for comments, and waiting on responses, has a large effect on how long it takes for the bug to be fixed. Figure 4.9 shows when the various information was provided: when the bug is first reported, or at a later time.

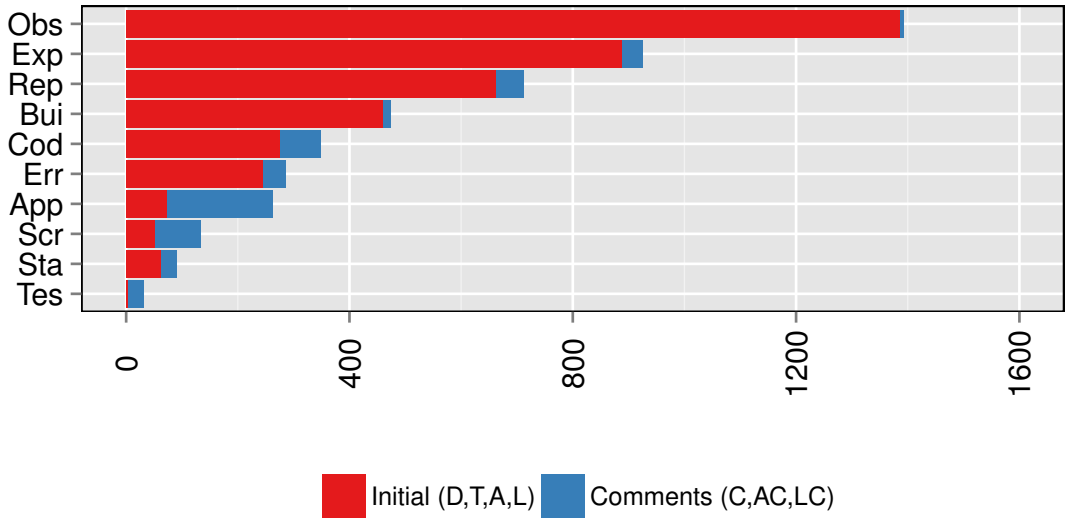


Figure 4.9: Number of bug reports where feature is provided, by time of provision

Including the information when a bug is first reported is by far the most common occurrence. Overall, only 12% of the total amount of information is first provided in comments. This could be considered an encouraging result, suggesting that the original information provided in bugs is mostly sufficient. However, what is not measured is whether the information provided for each feature is *complete*. For example, a user may have provided some *Steps to reproduce*, but these may by incomplete or unclear, and a developer is still required to make additional follow-up queries. As in previous sections, no attempt has been made to verify whether the information provided in the description is complete or accurate.

There are some fields, however, where it is more common for information to not be provided until the comments, rather than in the initial description. In particular, *Application Code* and

Test cases are far more likely to have been added later. This is largely due to the fact that these are usually added by developers working to fix the issue, rather than by the original reporter. Often in fact the code provided may be the exact patch that was applied to fix or test the issue. Any automated system which wished to assist developers in fixing bugs could therefore not rely on this being available, since by this point the developers have largely solved the issue, and the system would be redundant.

Screenshots are often also provided in a later comment, but not for this reason. Often these are provided as an attachment to the very first comment of the bug report, made by the reporter. The reasons for this are unclear, as Bugzilla allows the user to attach a file while describing the issues. This is also true to a lesser extent of other attachments such as *Error reports* or *Code examples*. In consequence, any automated system which examined only the description may well be missing valuable information.

For almost all features, there are some bugs in which the information is not provided initially. For some, such as *Stack traces* and *Error reports*, this may be because users have to be specifically prompted for such information, and are not aware of either how to retrieve it or of its importance. Unfortunately, the investigation has not captured the number of times developers ask for such information but do not receive it. This is particularly interesting for the fields mentioned because these two features are often highly accurate indicators of where in the project source code a bug is likely to be.

4.6 How does the information provided affect the outcome of the bug?

Sadly, not all bugs end up getting fixed by the developers. As Section 2.2 outlines, bugs can also end up being closed as invalid, duplicate or for a number of other reasons. Furthermore, the sample in this chapter also included bugs that were currently being worked upon. There are two fields in Bugzilla which detail the overall outcome of the bug, resolution and status. In the default Bugzilla life cycle [Bugb] (as shown in Figure 2.2, repeated below) these can take on a number of values. However, this life cycle can be customised for each project, and so in the sampled bugs a larger variety of values were present. To simplify things, all the observed values were mapped to a smaller number of possibles outcomes. Table 4.10 shows these outcomes, and the values for resolutions and status to which they correspond.

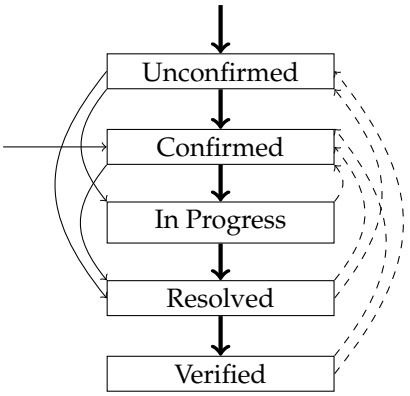


Figure 2.2 (Repeated): Life cycle of a bug report

Outcome	Status	Resolution
Fixed	NA	FIXED
Duplicate	NA	DUPLICATE
Incomplete	NEEDINFO	INCOMPLETE, NEEDS_REPRO, WORKSFORME, EXPIRED
Invalid	NA	INVALID, NOT_ECLIPSE, WONTFIX, BY_DESIGN
In Progress	ASSIGNED, REOPENED	LATER, REMIND
New	NEW, UNCONFIRMED	NA

Table 4.10: Bug outcomes

For information, the number of bugs for each bug outcome are shown in Table 4.11. It is interesting to note here the differences between some of the projects. Facebook in particular has a high number of *Incomplete* and *Invalid* bugs, which may reflect the behaviour stated earlier: non-technical users of Facebook using the BTS to report problems with the website, not the API. The high number of *Invalid* bugs for Apache is more surprising however, and the reasons are not clear.

	Fixed	Duplicate	Incomplete	Invalid	In Progress	New
Apache	121	51	23	143	13	49
Eclipse	228	39	21	59	12	41
Facebook	89	43	122	100	23	23
Firefox	56	110	115	64	3	52

Table 4.11: Bug outcomes by project

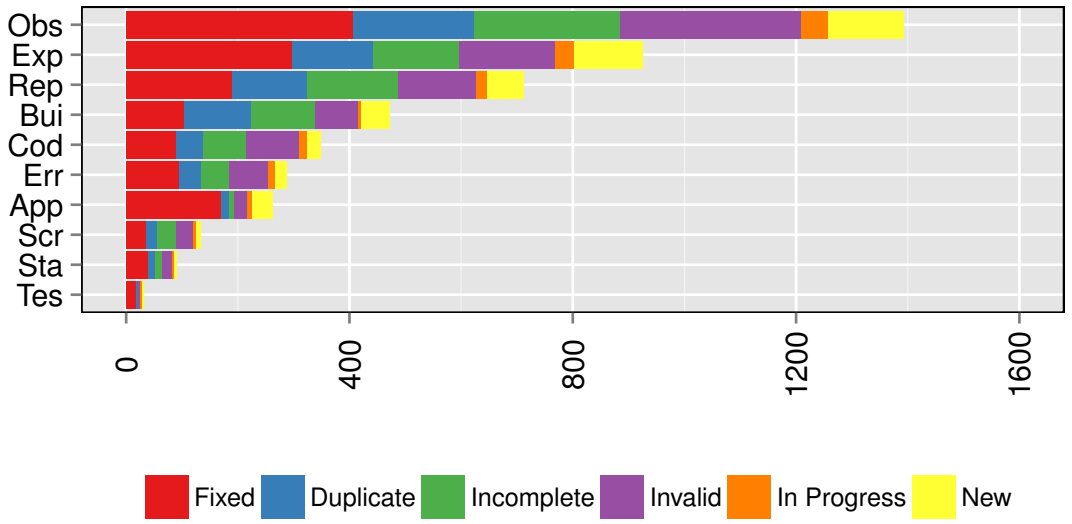


Figure 4.12: Number of bug reports where feature is provided, by outcome of bug

Figure 4.12 shows for each feature the distribution of eventual outcomes of the bugs. At first glance, there does not appear to be any strong relationship between individual features and whether or not the bug is eventually fixed. This can be confirmed by examining the correlations between the presence of each feature and the bug outcome, the significant values of which (p<0.01) are shown in Table 4.13. The only exception to this of any note is the *Application Code*, which has a correlation coefficient of around 0.33 with the outcome of *Fixed*, but this is still

not especially strong. In addition since, as discussed in Section 4.5, *Application Code* is more likely to be provided later on, and often by the developer fixing the bug, this is hardly a useful result. It is also very surprising to note that there is in fact a very weak *negative* correlation between providing *Steps to reproduce*, *Build information*, or *Observed behaviour* and the bug being fixed.

	Fixed	Duplicate	Incomplete	Invalid	In Progress	New
Obs	**-0.09**	0.03	**0.08**	0.02	0.05	-0.05
Exp	0.03	0.02	-0.03	**-0.12**	0.03	**0.11**
Rep	**-0.08**	**0.09**	**0.13**	**-0.07**	-0.01	-0.03
Bui	**-0.12**	**0.18**	**0.11**	**-0.1**	-0.06	0.01
Cod	-0.06	-0.02	0.06	0.05	0.04	-0.06
Err	0.02	-0.01	0	0.01	0.04	-0.05
App	**0.33**	**-0.13**	**-0.16**	**-0.15**	0	0.06
Scr	-0.02	-0.01	**0.07**	0	0.02	-0.04
Sta	**0.07**	-0.01	-0.01	-0.02	0	-0.04
Tes	**0.09**	-0.03	-0.05	-0.04	0.05	0

Table 4.13: Correlation between features and bug outcomes

There also does not appear to be any evidence for a relationship between the overall amount of information provided and the outcome of the bug. This can be seen by examining both Figure 4.14 and Figure 4.15. The range of values seen for *Fixed* bugs does not appear noticeably different to that of other bugs. It does appear that bugs which provide fewer features are more likely to be marked as *Invalid*, but again this result should not be surprising.

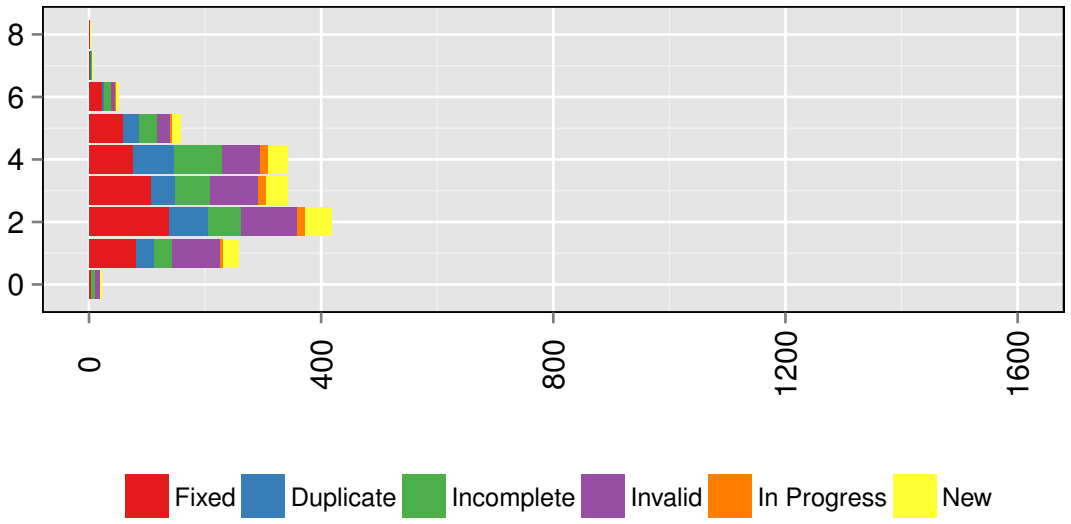


Figure 4.14: Number of features for each bug outcome

4.7 How could this information be extracted?

The sample of bugs used in this chapter is of a reasonable size, but a much larger number of bugs from a wider range of projects would have to be analysed in order to even approach results which can be generalised. Given the time-consuming nature of the work, this would be

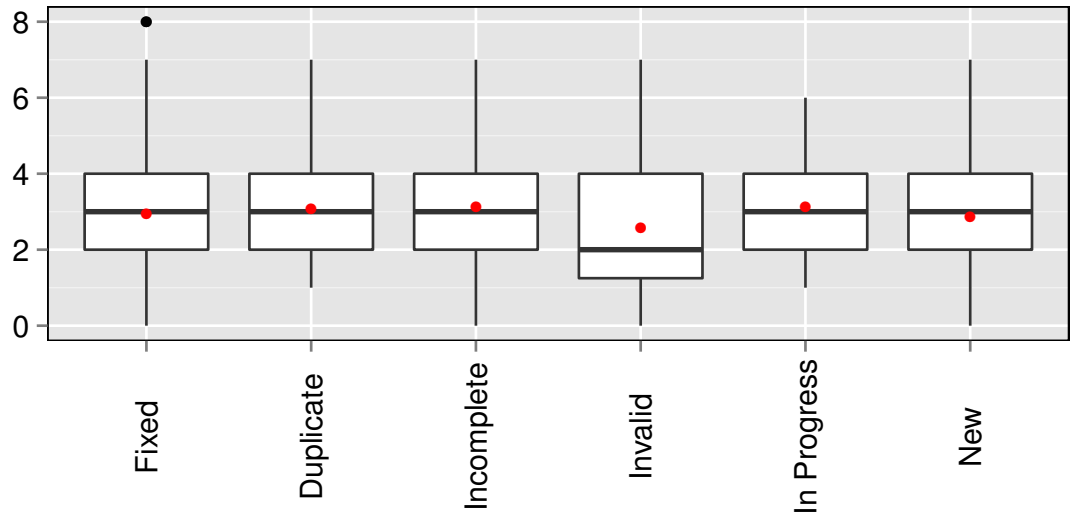


Figure 4.15: Range of number of features for each bug outcome

tedious to perform manually, and it had been initially hoped during the study that much of the work could be automated.

For example, *Screenshots* could be identified by examining the type of attachments or linked files. This would not be entirely accurate however; Firefox Bug 372163 contains a screenshot of how the application looks after the user's proposed patch has been applied, so is essentially a mockup. For Firefox Bug 321832, screenshots were provided, but they were embedded within a word-processing document. This may not present much of an issue for a developer, but would vastly overcomplicate automated image processing.

Previous techniques have also been developed to identify *Stack traces* [BPZK08] but these are not entirely accurate. Similar approaches may be possible for other structured features, such as *Code examples* or *Application Code*, but these are likely to be very challenging. As was shown earlier, these structured pieces of text are usually mixed liberally in with unstructured information.

The three features which are most commonly provided however – *Observed behaviour*, *Expected behaviour* and *Steps to reproduce* – are even more difficult to obtain automatically. Extracting this information automatically could be particularly useful for several reasons. Most simply, it could be used to identify bugs where a particular feature has not been included. It could also be used to discover bugs which share some features which are similar and some that are different e.g. two reports of the same bug by different users, where the *Observed behaviour* might be similar but the users have different ideas of the *Expected behaviour*. This difference may prevent common techniques for detecting duplicate bug reports from being successful. Extraction of these techniques could also be useful for bug localisation techniques – it may be that some of these features are more than useful than others, but at present it may not be possible to separate them. These are unstructured features, and may or may not be explicitly marked by the user providing information. One proposal is to search for instances where the user does explicitly say they are providing a particular feature To test this, the bug descriptions were processed by

a Python script which searched for the following phrases, followed by either a colon, hyphen or new line, ignoring differences in case, spacing and minor spelling variations:

- *Observed behaviour*
  - "observed behaviour"
  - "observed result"
  - "observed response"
  - "actual behaviour"
  - "actual result"
  - "actual response"
- *Expected behaviour*
  - "expected behaviour"
  - "expected result"
  - "expected response"
  - "desired behaviour"
  - "desired result"
  - "desired response"
- *Steps to reproduce*
  - "steps"
  - "reproduce"
  - "str"

Figure 4.16 shows the results of attempting to automatically identify which bugs contain which of these three features, compared to the manual results. As the large number of false negatives indicates, very few of the features could be correctly identified automatically. While these may not be the most sophisticated of expressions, and more complex approaches may give better results, these do give an indication of how few bugs actually contain a regular structure indicating the features desired by developers.

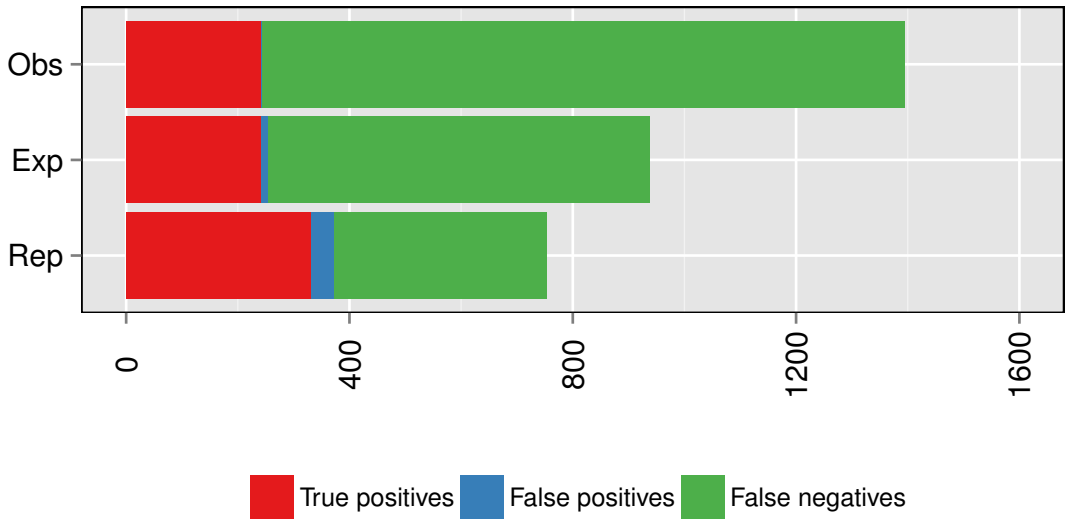


Figure 4.16: Automatically extracted features

Figure 4.17 shows the breakdown of the positive results, those that the did have some form of marker for each feature. As shown, the vast majority of the features that actually can be automatically identified come from Firefox, due to its simplified interface that specifically prompts users for these features. The unfortunate side effect is that Firefox is also responsible for most of the false positives, although there are relatively few of these. For example, in Firefox Bug 472170, what's stated as *Expected behaviour* is actually *Observed behaviour*.

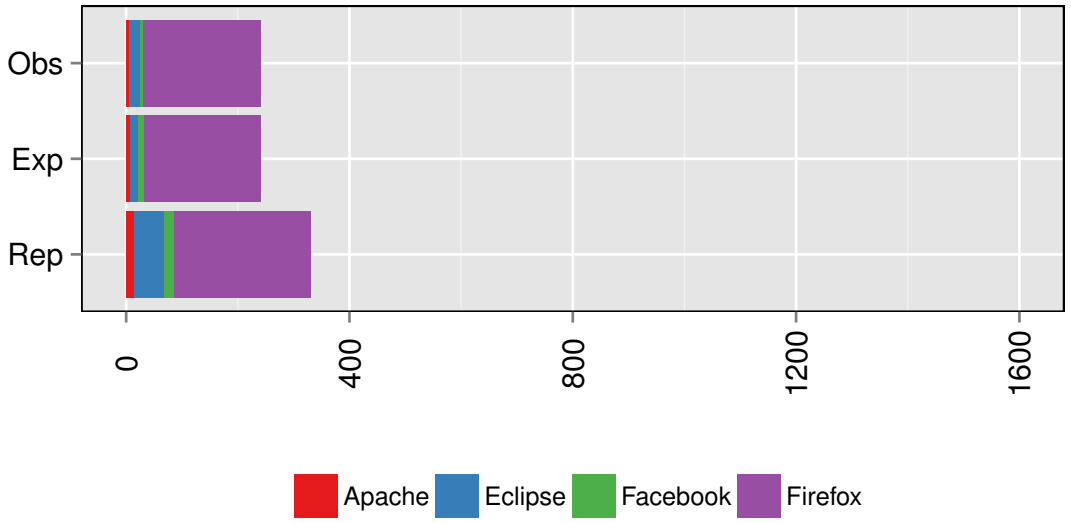


Figure 4.17: Automatically extracted features by project (true positives only)

4.8 How could this information be used by automated tools?

One aim of this chapter is to identify which features could be used to assist automated tools for fixings bugs. Each of the features could to some extent be processed by an automatic tool, although it is likely that each will need to be handled in a different manner. It is unlikely for example that much useful data could be extracted from screenshots by an automated tool. The features are also likely to provide differing levels of usefulness and accuracy in determining possible locations and causes for the bug.

Another aspect that should be considered before developing any automated tools is how often the information is actually provided, which was detailed in earlier sections. Before spending time developing systems which attempt to assist developers with understanding some aspect of bug reports, it is worthwhile to ask how often those aspects are provided. After all, there may be little point developing systems which can recommend the location of a bug given a screenshot if *Screenshots* are rarely provided by users, for example. Additionally BTSs could contain inaccurate information.

As shown, few of the features most desired by developers could be easily separated from the remainder of the plain text descriptions of bugs. Those that could, such as *Stack traces* were not provided particularly often. This section will assess the potential of each feature to assist automated bug fixing tools.

Steps to reproduce, Expected behaviour, Observed behaviour These features are all provided relatively often, and usually within the main bug description. Unfortunately, as Section 4.7 showed, the majority of the time these fields cannot be automatically separated from one another, or from the remainder of the bug description. As such, the most appropriate way to use these features is likely to mean keeping the description as a whole, and using IR techniques to relate the bug to the source code (as detailed in Section 2.6) or to find similar bugs.

Stack traces, Error reports These features may be simpler to extract from bug reports than many other features, although they are not entirely straightforward. However, they are not present in a large number of bugs. While they could therefore be an appropriate source of additional information to use in automated tools, they are not applicable for many bugs, and may be of most use only when used in conjunction with other sources of information.

Screenshots A combination of the difficulty of extracting information from *Screenshots* along with their relative scarcity mean it would be quite challenging to produce any significant bug fixing tools utilising *Screenshots*, with little applicability.

Code examples *Code examples* are not common. In addition, the language of the code is often not the same as the language of the application itself. For example, Eclipse is written in Java, and while the majority of its usage is also for Java programs, plugins exist for a myriad of different languages, and sample code could be in any of them. Sample code in Apache normally takes the form of sample configuration directives, in an Apache-specific language. Firefox code usually takes the form of HTML, but could also include JavaScript, and Facebook code can be simply a URL, but can also include any language which can get or post HTTP requests. Like *Screenshots* then, it may be difficult to take advantage of information from *Code examples* for automated bug fixing, and it is likely to be applicable only to a few bugs, since very few bugs provide the appropriate information.

Application Code *Application Code* was not particularly common in the given bug reports. Additionally, as the provision of such code usually indicates that the reporter has already performed some investigation of the bug on their own, utilising this information for bug fixing may not provide any significant benefit to the developers.

Test cases Much research has already been done on using failing test cases to determine bug locations. This work shows however that it is very rare for such tests to be provided by the user. Furthermore, many of the tests will not be provided in the same form as any existing test suite. Developers will therefore usually still be required to construct new test cases themselves in order to utilise such techniques.

Version, Build information The limited amount of information in these fields mean they are unlikely to provide much assistance with bug fixing on their own, but they could be used as part of a larger system.

4.9 Conclusions

This chapter has examined 1600 bugs across 4 projects and detailed how often users provide each part of a bug report. From this information, it is clear that in many cases, bug reports are not complete, and often do not provide all the information that developers could use to fix bugs. It also indicates that there would be a number of difficulties in extracting this information from the bug reports via automated means.

The quality of bug reports is important for developers. Even entirely invalid bug reports still take up some of a developer's time, in order to resolve them as such. Valid reports which are missing some information may be considered even worse, as developers are required to spend time doing their own investigation or repeatedly asking users to supply further information. It is also clear that there is a mismatch between the information that developers would want to appear in a bug report, and that they consider important for fixing bugs, and the information that actually appears.

Furthermore, the quality of bug reports has an important impact on research on other points of the bug life cycle. Many BTSs do not differentiate between different types of issues, in particular bugs and requests for new functionality. Even when they do differentiate, the issue type is not always provided accurately by the reporter. However, techniques such as those in Chapter 3 for discovering the origins of bugs are mostly appropriate only for bugs, and not for the introduction of new functionality. As Chapter 5 will show as well, the availability (or lack thereof) of particular types of information mean that this information cannot be relied upon to help with discovering the locations of bugs.

5 Determining where to fix bugs

After a bug has been created, and subsequently reported, the (hopefully) final point in its life cycle is its removal, or fix. The process of fixing a bug has several stages. First, developers have to read and understand the bug report they have been given, which may not always be easy. Often they will attempt to recreate the bug themselves, in order to check the bug report's accuracy and to get a better understanding of the operation of the system. Next, they have to identify where in the code of the project the bug is likely to lie. This can often be a complex process, involving customer communication, system knowledge, analysis tools, support from other developers, and occasionally a bit of inspired guesswork. All of this has to happen before the developer can even begin to think about *how* to fix the bug.

A survey of software development engineers at Microsoft [BNRR08] showed that this process – identifying the location of a bug – was one of the tasks that most dominated the time taken in the bug-fixing process. This problem can be especially acute for developers who are inexperienced, either with the system under maintenance or simply in general, or those working on large complex systems. One way to reduce the amount of time spent on this part of debugging is to give developers suggestions for where in the code the bug is likely to lie, while they are viewing the bug report. This chapter will describe existing techniques for automating some of the process, with a view to doing just that. Furthermore, it will describe novel approaches to improving and combining these techniques. The improvements will be evaluated in two separate studies over seven open-source projects.

5.1 Related work

The act of identifying where in the source of a project a particular bug is located is usually referred to as bug localisation. Bug localisation is part of the wider concept of feature location, which was first defined by Biggerstaff et al. [BMW93] (although they referred to it as concept assignment). They defined feature location as the act of discovering concepts of a program in human-related terms, and linking them to the parts of the program responsible for implementing those concepts.

Since then, many researchers have proposed various approaches to solving (or partially solving) the problem. This review will not attempt to explore all of these approaches; instead it will focus only on those that are specifically related to bug localisation, significant for historical reasons, or required to aid the reader's understanding. Detailed surveys on feature location in

general [DRGP11] and bug localisation in particular [WD09] have been carried out by other researchers and provide a more complete picture of the area.

5.1.1 Bug localisation

In a comprehensive survey, Dit et al. [DRGP11] defined three main approaches for feature location:

- Dynamic techniques which use runtime information about the system, such as execution traces
- Static techniques which analyse the structure of source code, e.g. graphs of control and data dependences
- Textual techniques using IR approaches, treating the source code as a textual corpus

These categories are broad and contain considerable diversity. In addition there are other techniques which use more than one approach or which also draw on other sources of information such as mining the project's revision history.

5.1.1.1 Dynamic techniques

Dynamic feature location techniques use information about the system as it is executing in order to determine which parts of code have been exercised. This approach to feature location was first presented as software reconnaissance [WS95, WCF96]. Here, the program is first instrumented to record which parts are executed while it is running. Developers then execute several scenarios (either manually or automatically) which exercise the system both with and without the feature that they are interested in locating. For example, to investigate a form validation feature, the developer might record the execution trace for a scenario where they provided valid values to the form, and a trace for a scenario where they entered invalid values. The software reconnaissance tool can then be used to isolate which parts of the system were only executed while using the feature in question.

Since this initial proposal, the approach has been refined by other researchers, some of whom have focused specifically on bug localisation rather than just feature location. When applied to bug localisation, one of the scenarios in question is usually designed to recreate the bug that is being investigated, while the other scenarios are intended to be 'close' without triggering the bug. For example, Reps et al. [RBDL97] described the successful use of the technique to find Year 2000-related bugs by executing systems with the clock set to a date after the year 2000, and comparing these traces to runs with the clock set to before the year 2000. Liu et al. [LYF+05] used a technique where a full suite of tests is executed against a code base, and the values of predicates within the program measured. Predicates which are true disproportionately more or less often in failing tests than passing tests are marked as suspicious, and to be investigated by the programmer as a possible bug location. They found that this could be successfully used to identify the location of a bug while requiring the developer to examine less of the program than other techniques. Ren and Ryder [RR07] performed a case study on some Eclipse bugs where the causes for failing tests could be successfully identified by ranking methods according to

their suspiciousness, based on the methods' position in the call graph of the failing tests and the changes which had recently been applied.

A similar approach is focused specifically on regression bugs: features which worked in a previous version of the system but which no longer work correctly. Harrold et al. [HRS+00] ran the same tests on two versions of the program, a buggy version and a non-buggy version. They found that by comparing traces from the two runs, they could partially identify the location of the change that caused the bug.

Similar techniques have been put forward into several practical tools. Tarantula [JHS02, JBH07] presents developers with a visualisation of the source code, where individual lines are coloured based on their suspiciousness, determined from examining the traces of several test cases. Based on this work, Apollo [ADTP10] runs an execution multiple times, modifying the input automatically to take different paths through the program, and uses this to automatically identify and locate failures in PHP web applications. A similar tool was introduced by Dallmeier et al. [DLZ05], which attempts to identify sequences of methods which are faulty, rather than single lines in isolations. Finally, Pinpoint [CKF+02] records real transactions that take place on an internet application. It detects transactions as either failed or successful, and uses clustering of the execution traces to highlight components that it believes to be at fault for the failed transactions.

While they have been shown to be somewhat useful, dynamic techniques do have several downsides. The quality of results can be dependent on the quality of traces provided [ED05], particularly if the traces which don't exhibit the bug are not sufficiently 'close' to those which do. This can require developers to spend considerable amounts of times in preparing not only scenarios or automated tests that correctly replicate the bug (which would of course be desirable whether or not they were using automated bug localisation techniques) but also scenarios and tests which do not replicate the bug but which are still somewhat similar. The techniques also impose performance overheads on the system in order to collect the data [EWSG09] that may be unacceptable in production environments. As such, there is little evidence to suggest that they are much-used in any industrial context [CZvD+09].

5.1.1.2 Static techniques

In contrast to dynamic techniques, static techniques do not need access to a running system in order to perform feature location. Instead, they analyse the source code of the project and identify relationships between program components. Chen and Rajlich [CR00] proposed a technique, later implemented in the RIPPLES tool [CR01], in which an abstract system dependence graph was built of the project. Here, nodes in the graph represent functions and global variables, while edges are either calls from one function to another, or data flow from a function to a variable. This was intended as an interactive tool; developers select a starting node of the graph, usually the program entry point, and then choose further nodes to explore, based on the graph given by the tool. The tool supports the developer by maintaining a search graph, keeping track of which nodes have already been visited and deemed relevant or irrelevant to the feature.

Robillard and Murphy [RM02] developed a similar idea of Concern Graphs, in which a feature is represented as a graph of program elements, and implemented a tool called FEAT that used a richer set of information to construct the graphs. Small case studies showed that it could be useful to help developers find the code covering most of a wide-ranging concern over a large code base, whilst eliminating much of the irrelevant code. Similarly, Trifu [Tri08] used graphs constructed by dataflow analysis.

What all these approaches have in common is that they require the developer to identify a starting location, either the starting point of the application or elements known or suspected to be related to the feature, and then to pick which nodes to explore and in which order. Robillard [Rob08] modified this approach to present developers with a list of recommended elements to explore, by analysing the nodes of the graph, but it is still the developer that ultimately has to actively drive the search. Additionally, the static analysis approaches have the downside of presenting developers with a large number of irrelevant elements along with the elements related to the feature of interest. This is particularly true for cross-cutting concerns such as logging, which are often spread across large areas of the code.

5.1.1.3 Textual techniques

The third main approach to bug localisation involves treating the source code of the project as if it was natural language. At its most basic, this practice can be seen in the widespread use of grep [Gre] or other similar search tools, where developers attempt to find concepts simply by searching for related terms in the code. Petrenko et al. [PRV08] described a technique where developers used grep, combined with ontology fragments – an organisation of part of the information in the system and the relationships between concepts – which they recorded as they explored the system. They found that these ontology fragments allowed the developers to identify more useful queries that they could then grep for, as evaluated against a small sample of real-world bugs.

In their more advanced forms, these approaches tend to borrow heavily from IR techniques. Mostly, they build an index of the source code of a project, in which each program element (at the desired level of granularity) is represented as a document, and the contents of that document consist of a number of terms. In the basic Vector Space Model (VSM), a document-term matrix is constructed, which simply records which terms are used in which documents. Developers then issue queries, in natural language, and by matching the terms in the query to the terms in the matrix, the most relevant documents should be selected and presented to the developer. Where the more advanced textual approaches to bug localisation mostly differ is in the information sources on which they draw, and the various weighting schemes that are applied to the matrix and query.

Marcus et al. [MSRM04] first introduced the IR technique of LSI to feature location. This attempts to alleviate problems due to polysemy – where one term can have multiple meanings – and synonymy – where multiple terms mean the same thing. This can be a particular problem in bug localisation; users and developers will often not use the same language when talking about the system [KMC06]. They found that their approach was simpler to use than the dynamic techniques they compared it to, while still being more effective than grep-based

approaches. This was later incorporated into a prototype tool for developers [PMDS05, PMD06], and developed further for bug localisation by Poshyvanyk et al. [PGM+06], although that approach used a combination of both textual and dynamic information. A similar IR technique of Latent Dirichlet Allocation (LDA) has also been applied to bug localisation [LKE08, LKE10, NNAK+11]. LDA is a technique in which each document is represented as a mix of topics, and each topic is a mix of terms. It is used to 'discover' common topics which are hidden across the corpus of source code, and in many ways these topics can be seen to represent individual concepts. There are a large number of other different IR techniques that can be applied to the feature location problem. While all of them have been shown to be somewhat effective, comparisons by Rao and Kak [RK11] and Wang et al. [WLXJ11] have shown that simpler techniques like the basic VSM described above could actually be more effective than the more complex techniques such as LSI or LDA. Thomas et al. [TNBH13] also found that the parameters used to tune the techniques had a large impact on how effective they were, and that the best results could be obtained by combining several different techniques.

Other approaches to textual-based bug localisation have tried to include other information than just the source code when constructing documents. Cleary et al. [CEBE08] incorporated information from bug reports, mailing lists and other documentation to develop a richer set of relationships between terms, although they found little gain over simpler techniques. Zhou et al. [ZZL12] also used bug report information, but only evaluated this at the source file level. Canfora and Cerulo [CC06a] built an index of each project in which each line of code is represented as a document. Each document consists of the text of any bug reports or commit messages that have been related to that line. They reported relatively encouraging results, but observed large discrepancies between different systems, especially regarding the maturity of projects. This was similar to earlier work [CC05] carried out at the file level of granularity, but slightly improved in performance, with the penalty of a higher cost in time and space requirements. CVSSearch [CCW+01] builds a similar index but utilises only CVS comments, while Hipikat [ČM03] infers links from source code to commit messages, bugs and documentation.

These textual approaches usually return to a developer a very large number of results, sometimes the entire system, ordered by the probability that the document is related to the developer's query. All of these techniques performed favourably when compared to grep-based techniques but, particularly on larger projects, the first result they returned to developers that was actually relevant to their query would often lie well outwith what a developer would realistically examine.

Most of the tools here require the developer to formulate a query before carrying out a search (in practice, this would often be an iterative process, with queries being refined and tested). This is similar to the static and dynamic techniques, in which developers have to be actively involved in the bug localisation. Unlike those techniques however, it is possible to make more automated approaches to bug localisation, in which the queries are automatically generated from the bug report. Torchiano and Ricca [TR10] gave preliminary results of such a tool, in which a basic document-term matrix was constructed from the source and the text of a change request (similar to, although more general than, a bug report) was used as the query. They reported very high recall for their approach, although quite poor precision. DebugAdvisor [AJL+09] allowed developers to search for likely bug locations using *fat queries*, which could

contain not only textual descriptions of the bug, but also stack traces or error logs. The different parts of the queries were converted into typed documents, and these were then matched to source code files using standard IR techniques. In a similar way to Hipikat [ČM03], the tool then used information about the relationships between bugs, documents and people to recommend to developers a list of related bugs, other developers who may be knowledgeable about the problem, and code files and functions. A preliminary internal evaluation by developers at Microsoft was carried out, in which a total of 628 queries were submitted. Developers could voluntarily rate the recommendations they were given, and of the 208 responses that were rated, 78% were rated by developers as useful. Additional anecdotal evidence also suggested that developers found that DebugAdvisor was a useful tool in helping them to locate and fix bugs.

5.1.1.4 Hybrid techniques

As mentioned, the different techniques all have different strengths and weaknesses. As such, researchers have obviously sought to combine some of the techniques, to (hopefully) get the advantages of each. Zhao et al. [ZZL+06] used IR techniques in order to create an initial list of functions of interest. This list is then extended through static analysis to incorporate other functions which have strong relationships with the initial functions, and to rank them. They found that this static analysis could be used to improve both the precision and recall over IR-only techniques. Similar approaches have used other relationships between classes to appropriately weight the list returned by the IR technique [ASGA12].

Poshyvanyk et al. [PGM+06] combined the IR approach of LSI with a dynamic technique where various scenarios are executed on the running system. Both these approaches produce a probability for each function of it being related to the feature in question. These are then combined using an affine transformation, where the weight awarded to each component can be varied depending on the confidence in each technique. They found this combination of approaches to produce better results than either technique had on its own. This approach was further developed as PROMESIR [PGM+07].

SITIR [LMPR07] is a similar approach. Here, an execution trace of the desired feature is first used to provide a potential list of functions. Then this list is ordered by the results of an LSI query. This cuts down on the large number of results that a developer can be presented with, while still hopefully placing the most relevant results first. Their approach was found to out-perform the two basic approaches that it was based on, confirming the value of a hybrid approach, but this was only evaluated on a very small set of bugs.

Dora [HP07] uses an initial developer query, possibly extracted from a bug report, to identify a set of possible related methods using similar textual methods to those described earlier. From these, the developer selects one or more they suspect to be related to the feature, and Dora builds a graph of methods related to these seeds by caller or callee relationships. The additional nodes being added to the graph are filtered to identify which of the called or calling methods are actually relevant to the feature, again by comparing the text of the methods to the initial user query and hiding those methods which are not relevant. Similar to some of the static

approaches presented earlier, this tool can be fairly effective, but again it relies on the developer to actively drive the search for relevant code elements.

CERBERUS [EAAG08] combines all three approaches to feature location. It uses the combined dynamic and textual approach of PROMESIR [PGM+07] to generate an initial list of relevant elements, and then uses static analysis to extend this set of elements. In an evaluation on the open-source JavaScript implementation Rhino, the authors found that CERBERUS was more effective at locating features than a number of other techniques which used only one type of analysis.

Gethers et al. [GDKP12] also combined LSI with execution traces and relationships between elements. Unlike some of the earlier techniques however, they determined the relationships between elements by analysing the history of RCS commits in the project to identify *evolutionary couplings* – elements which are frequently changed together. Their approach was also adaptable, and could operate even if some of these information sources were not available. They found that, while the level of accuracy reached was still far from what would be needed by developers, the combination of all three sources of information was a significant improvement over previous techniques for bug localisation.

5.1.2 General bug predictors

The approaches presented for bug localisation are all intended to help a developer find the location of one, specific, bug. However, there is a long history of research and tools aimed at more general bug-finding. These tools generally provide developers with a list of code locations at which they think there might be bugs. Many of these approaches use metric values to identify which components in a system are likely to be error-prone, dating at least to the work of Selby and Porter [SP89], which used a vast range of metrics such as size, previous faults, previous changes and complexity measures to classify files as either fault-prone or not. Countless other researchers have also examined the relationship between various metrics and bugs, and detailed surveys have been conducted elsewhere [CD09, Kit10]. Of particular relevance to this work, Zimmermann et al. [ZPZ07] showed there was a correlation between some complexity metrics and post-release defects, although a similar correlation was also found simply for total number of lines of code. Kim et al. [KWZ08] showed that measuring the change in metric values could be an effective predictor in whether a change was likely to introduce a bug or not. Similarly, Couto et al. [CSV+12] found that examining the behaviour of metrics over time in a project could be a predictor of later defects being discovered. Hassan [Has09] did similarly, but utilised information about the type and frequency of past changes to a component.

Using metrics to identify potential bug locations is one form of static analysis. There are other static analysis tools which can be used to inspect code and to issue warnings about areas that are likely to contain bugs. These are tools such as lint [Joh77] or FindBugs [Fin], which warn when code matches certain patterns, such as calling a method on a null variable, or a runtime cast which will never succeed. While these can be bugs, they are not usually the type of bugs that would be reported by users, although they may be the cause of such bugs. Indeed several researchers have found evidence of a weak correlation between warnings issued by static analysis tools and bug reports later reported by users [GE09, CMSV11] .

5.1.3 Extracting information from bug reports

In addition to bug localisation and prediction, there are many other uses for extracting data from bug reports. These are often based around trying to reduce the amount of time spent on particular stages in the bug report life cycle, such as the time taken to prioritise and assign bugs, or to check for duplicate bugs. Several researchers have used natural language techniques to discover duplicate bug reports [RAN07, JW08, SLW+10, AM11]. Podgurski et al. [PLF+03] built a system which would cluster bugs, represented by failed tests, that had the same cause. This was based on an analysis of the execution profile of the tests. Wang et al. [WZX+08] produced a system which utilised both similarity in textual description and execution profiles.

Rastkar et al. [RMM10] produced automatic summaries of bug reports, in order to reduce the time taken for a developer to understand the bug report. Antoniol et al. [AAD+08] used the description of a bug to classify whether bug reports were actual bugs or were requests for enhancements. Sureka and Indukuri [SI10] used the title to classify a bug by severity.

Machine-learning processes which can assist in *triaging* bugs, recommending developers who would be well-suited to fix a new bug report, have been extensively studied [ČM04, AHM06, CC06b].

Guo et al. [GZNM10] built a model which would predict whether or not bugs would eventually be fixed or not, based on factors of the bug report and the reputation and relationships of the people related to the bug. Hooimeijer and Weimer [HW07] built a model which would predict how long it took to fix a bug, based on features of the bug report available when, or soon after, it was created. Weiß et al. [WPZZ07] produced a similar model based on the similarity of a bug report to existing fixed reports.

5.1.4 Conclusions

One unfortunate fact is that very few of the bug localisation techniques presented here have seen widespread use in practice, as evidenced by the low number of professional evaluations in the systematic survey of feature location techniques [DRGP11]. This is hardly a surprising fact given the long periods of time usually found between research and industrial use [OGKW08]. Additionally, a user experiment into bug localization techniques [PO11] showed that most work on the problem made several assumptions about the way developers use bug localisation tools that were not always true. They did however find that the techniques could have some beneficial effects, but that the current evaluation techniques used were not always sufficient to determine the quality of a technique.

There are of course many reasons for the low adoption of tools by developers, such as lack of availability as 'finished' tools, usability issues, and reluctance to commit to new tools that have large initial costs [AP08, NGS+10]. For bug localisation, this can make developers unwilling to try techniques that require them to do a lot of investigative work for a particular bug, until they can be shown evidence that such a technique is likely to work. In an attempt to reduce this problem, this chapter focuses on techniques which do not require a large amount of upfront work for an individual bug. Such techniques could be used for example to present a developer

with a list of potential bug locations as they are first reading the initial bug report, through integration with a BTS.

For the most part, static and dynamic techniques are not suitable for this kind of bug localisation. Textual approaches are, however; the description of the bug report can be used as a query over the corpus formed by the project source code. However, textual approaches miss out on a large number of clues that could also be used, which include lessons from static and dynamic analysis as well as more general bug-finding tools. As some of the hybrid techniques also showed, combining information from multiple sources was often more effective than one approach alone. The remainder of this chapter will examine novel ways to enhance textual bug localisation techniques with additional information from a number of sources, and consider how the results of such an approach could be integrated with a BTS and into the development process.

5.2 Preliminary study: bugs with similar descriptions

As discussed previously, developer adoption of automated bug localisation tools has been relatively low; most developers rely on their own or colleagues' knowledge, basic reading techniques or grep-like tools [SLVA97]. Anecdotal evidence from developers suggests they are often unwilling to try new tools until they have been shown to be reliably useful. Reducing the effort required for each bug would make developers more likely to experiment with unfamiliar tools, increasing both adoption and feedback to improve the techniques. One way to address this is to use as much information as possible from the report of a bug itself, rather than relying on the developer. As Chapter 4 showed however, much of the information that is useful for developers is not easily obtainable as a discrete part of a bug report. Instead, the information all forms part of the general description of the bug, and it is difficult to extract individual components, such as the expected behaviour, steps to reproduce etc.

Several techniques [CC06a, PGM+06, LKE10], based on IR methods, have been proposed which perform bug localisation by using the source code as a textual corpus and a bug description as a query. While they have shown promise, they still often place the first method relevant to a given bug well outwith those a developer would be likely to examine. One way to improve these techniques is to draw on additional sources of information related to the source code.

Existing research has shown that similarities in the language used can be utilised to detect duplicate bug reports: multiple bug reports referring to the same issue. It is this success with duplicate bugs which prompted this research: if duplicate bugs reports, which by definition are fixed in the same source location, can be detected through the use of similar language, can bugs which are in the same location but *not* duplicates be detected in the same way? It seems possible therefore that bug reports which are textually similar could correspond to similar locations in the source code. As an example, the following three bugs were all raised in the jEdit project:

Bug 1659666: "perl.xml patch for quote-like operators"
Bug 1760646: "perl colorization"
Bug 1807549: "Perl escape still not working correctly"

As the summaries state, each bug is related to Perl, and the bug descriptions make clear that these bugs are all related to parsing the Perl syntax. Although these bugs are *not* duplicates, each of them required a change to the same method to fix: `org.gjt.sp.jedit.syntax.TokenMarker.markTokens(LineContext, TokenHandler, Segment)`. That method makes no mention of Perl, and it is not clear that a textual approach would have successfully identified it. While this example is trivial, this preliminary study seeks to examine if this pattern is widespread, and if this fact can be exploited to improve bug localisation techniques.

It appears unlikely that any such technique would be applicable for a large number of bugs, but this is not a major issue. By identifying and combining multiple small improvements, as detailed in later sections, a more substantial overall improvement should be achieved. As discussed previously, other research [TNBH13] has shown that combining information from multiple classifiers can result in an improvement over any single classifier.

5.2.1 Techniques & implementation

As stated, this chapter aims to investigate techniques that will recommend bug locations to developers while requiring minimal input, and explore this through the development of a prototype system. When a developer views a bug report, the system should retrieve a list of methods deemed relevant to the bug and present them to the developer. This section will explain three approaches to this problem which will then be evaluated and compared.

`Source`: An existing approach, explained earlier, which relies on measuring the similarity between the description of the bug report and the source code.

`Bug`: The new approach proposed in this preliminary study, which measures the similarity between the text used in the bug report and the text of other already fixed bug reports, recommending the methods that were involved in the previous bug fix.

`Comb`: An approach which combines the results given by `Source` and `Bug`.

5.2.1.1 Bug description approach (`Bug`)

In the bug description approach, the similarity between bug reports is used to predict which methods are related to a new bug report. For each method in the system in question, a binary classifier is created which can answer a simple yes/no question: Is this bug related to this method? These classifiers are based on a model of the bug report, which takes the form of a document-term matrix, an example of which is given in Table 5.1. In this instance, the description of each bug is a document, each represented by a row in the matrix. Each of the terms used in any bug report is represented by a column of the matrix. In its simplest form, each cell in the matrix is either a 1 or a 0 depending on whether the given term is contained inside the given bug report.

When a new bug is reported, each classifier is called in turn with the contents of the bug report. If the classifier returns positively for any method, then that method is presented to the user as potentially being related to the bug report. When a bug is fixed, the methods involved in the fix are used to update the classifiers.

	attribute	specify	crash	hang	⋯
Bug 1	0	1	0	1	⋯
Bug 2	0	1	1	1	⋯
Bug 3	0	0	0	1	⋯
Bug 4	1	0	1	1	⋯
⋮	⋮	⋮	⋮	⋮	⋱

Table 5.1: Example document-term matrix

Words in the bug report are split at any non-letter characters, and at any change from lower- to upper-case. This allows multi-word identifiers using common coding conventions to be split into constituent words, e.g. the identifier name `dateParser` becomes *date* and *parser*. The words are then filtered by removing stop-words. These are commonly used English words which will appear in many bug reports, and are not useful for discriminating between bug reports. Additionally, rather than being a simple binary indicator, each element in the matrix contains a Term Frequency-Inverse Document Frequency (TF-IDF) value [SB88], which weights the importance of terms positively based on how commonly they appear within the document, and negatively based on how common they are in the corpus of bug reports as a whole. Stemming, the reduction of words to their root, was not found to improve results and so is not applied.

Each classifier is a C4.5 decision tree, implemented in Java using the Weka [Wek] data mining library. Each decision node in the tree is based upon an individual word, and whether the word is present or not present in the bug report. Each leaf node is one of two classes: whether the method is relevant or not relevant to the bug. Note that any type of classifier could be used, but decision trees result in models which are easier to interpret and quicker to train, aiding analysis and experimentation for this preliminary study.

It may be obvious that for any given method almost all bugs in the system will *not* be relevant. Training a classifier with such a large number of negative examples and very few positive examples can have an effect on its performance, as it becomes heavily biased and less likely to label any new bugs as being related to a method. As such, it was decided to artificially restrict the number of negative examples used when training the classifiers. Whenever a bug is known to be related to a method, it will always be used to update the classifier. Otherwise, there is a 5% chance that the bug will be included. The 5% threshold was chosen after initial experimentation: including every example simply results in the classifier returning a negative prediction for every bug, and after trying a number of thresholds 5% appeared to give a good overall performance. This random undersampling is a commonly used and effective technique for handling unbalanced data sets [HKN02], although further experimentation would be needed to accurately determine the best possible sampling technique and threshold. Additionally, using fewer subjects also greatly reduces the time taken to train the classifiers.

5.2.1.2 Source code approach (`Source`)

In order to compare `Bug` to existing techniques, an implementation was also required of those techniques, which identify methods by their textual similarity to the bug report. Unfortunately no suitable implementations were readily available. Therefore, a basic implementation was

created in Python, using the Gensim [Gen] framework. As this required each document in the corpus to be located in a separate file, a simple parser was implemented in Java using Eclipse's Abstract Syntax Tree (AST) functionality. A copy of the project being examined was obtained and recursively scanned. First, each file was searched for class declarations. When class declarations were found they were scanned for constructor declarations, method declarations or nested class declarations which were then scanned recursively. The contents of each method or constructor, including comments, were written out to disk, one file per method. Anything else in the class, e.g. fields or static initialisers, was ignored.

From these files a basic document-term matrix was created, with one document for each method in the project, weighted by TF-IDF value. This is similar to the matrix presented in Table 5.1, but with each row being the contents of one method, not one bug report. While other research [MSRM04, CEBE08, LKE10] has recently shown that LSI or LDA could be more suitable for bug localisation, these techniques are sensitive to changes in parameter settings and to the query used. In this work, as the query is automatically extracted and not provided by a developer, initial experiments suggested that LDA and LSI did not appear to be any more successful than the simpler techniques, although further experiments would be needed to ascertain this. Regardless, it is thought that those techniques would be equally amenable to the proposals in this chapter.

Stop-word removal was again applied as was identifier splitting, using the same conventions as for the bug description approach. In addition stemming was applied, using Gensim's implementation of the Porter stemming algorithm, as it appeared to be more effective on the code base than it had been on the bug description. This decision was taken based on the improved performance it demonstrated in preliminary testing, and may be seen as counter-intuitive: other research suggests that stemming should in fact be less suitable for source code than the natural language bug reports [HRK12]. Investigating this was out-of-scope for the work, and it is possible this is project- or use-specific and would not generalise.

When a new bug report is raised, the description is used as a query on the models, and the methods returned to the user in descending order of similarity to the bug report. Unlike the bug description approach, the models do not have to be updated when bugs are fixed. Ideally, they should be updated when methods are changed or added, but this was not implemented for this preliminary study.

5.2.1.3 Combined approach (Comb)

As stated earlier the aim of this work is to enhance existing techniques rather than solely to introduce a new technique. To do so, the results of Bug are used to improve those of Source. First Bug is used to partition the methods in the system as either relevant or not relevant. Then each partition is sorted individually in the order returned by Source. All the methods deemed relevant are then returned to the user in order, followed by those methods deemed irrelevant. In this way, methods which are incorrectly omitted by Bug will still be presented to the user, although they will be located further down the list than Source, depending on how many false positives were returned. However, methods which are predicted correctly will be promoted from their position in Source.

5.2.2 Evaluation

To determine how the proposed technique could assist developers, and how it compares to existing techniques, I carried out an evaluation on 372 bugs across 4 real-world open-source projects.

5.2.2.1 Subjects

Four projects were used to evaluate the techniques, ArgoUML, JabRef [Jab], jEdit and muCommander [Muc], from a dataset provided by Dit et al. [DRGP11][1]. These are all small to medium projects written in Java, covering a variety of uses: ArgoUML is a UML editor, JabRef is a reference and citation manager, jEdit is a text editor primarily intended for use by programmers and muCommander is a file manager. Each sample contains the descriptions of a number of bugs and a list of which methods were updated in the fix of each.

In addition, a set of additional training bugs was also obtained for each project, used solely for training the classifiers and not for evaluation purposes. This was done automatically through examination of all Subversion comments since the start of the project to identify bug numbers, using the same technique outlined in Chapter 3. As there, and unlike other work, the method used to parse the commit comments for this evaluation is very liberal: any string of numbers is considered a potential link between a bug and a commit. The method does not search for keywords such as "Bug" or "Fixed", as these are often project-specific, and even within projects developers often follow different conventions or may be inconsistent. There is no check that the developer was actually referring to fixing that specific bug. However, the next step is to download the bug description for each number found; any numbers which are not actually bug IDs will therefore be disregarded at this step.

The code changes in each commit are also parsed using the parser described earlier, and the methods included in the fix recorded. This process is similar to that followed by Dit et al. [DRGP11], but it is not entirely identical, and their sample has been manually verified. As these bugs are only used for training, the criticisms of the technique raised earlier that the links may not be entirely accurate do not cause any issues with the validity of the evaluation, although it may of course adversely affect the performance of the classifiers.

Table 5.2 gives details of the projects involved, and the versions and bugs used in the original dataset. All the evaluated bugs were raised and fixed between the two releases specified; the additional training bugs consist of all those raised since the start of the project to the final revision specified, unless already in the evaluation set. The corpus required for SOURCE was built by parsing the source code from the start version.

BUG is only suitable for methods which have been involved in more than one bug, as it requires some evidence to train the classifier. Table 5.2 therefore also shows the number of applicable bugs: evaluated bugs for which BUG can be applied, i.e. those for which at least one method was involved in another bug fix. In addition it shows the distribution of bugs within methods. As shown, the number of methods with any bugs at all is 10%–80% of all the methods in the various projects. Within that, only 1%–24% are related to more than one bug. However,

[1] Due to differences in its composition, the fifth project in the dataset, Eclipse, was not suitable for this evaluation

	ArgoUML	JabRef	jEdit	muCommander
Start version	0.20	2.0	4.2	0.8.0
Start revision	9699	1456	5111	1596
End version	0.22	2.6	4.3	0.8.5
End revision	10969	3265	16702	3578
Evaluated bugs	91	39	150	92
Applicable bugs	43	23	96	57
Training bugs	266	34	314	117
Methods	10806	2946	5263	3916
Methods with 0 bugs	8169	2655	2784	767
Methods with 1 bug	1956	247	2168	2226
Methods with ⩾2 bugs	681	44	311	923

Table 5.2: Evaluated projects

these methods are related to a large number of bugs, accounting for 59% of all the evaluated bugs.

5.2.2.2 Source code approach

To evaluate Source, a model is built of each project using the technique described earlier. In turn the description of each bug report is then used as a query on the model, returning all methods in descending similarity.

In general, we are interested in measuring the position of the top-ranked method which is actually relevant to the bug, known as the first relevant method (FRM). From this point it is assumed that the developer will be able to more accurately identify the remainder of the methods required to fix the bugs through other impact analysis techniques, such as examining methods related to the method found. While this assumption may not be entirely accurate [BKEL11], this is one of the most common way of evaluating bug localisation approaches [PGM+06, LKE10].

	ArgoUML	JabRef	jEdit	muCommander
1	1 (1%)	0 (0%)	2 (1%)	3 (3%)
⩽10	9 (10%)	5 (13%)	23 (15%)	13 (14%)
⩽100	29 (32%)	16 (41%)	67 (45%)	32 (35%)
⩽1000	63 (69%)	27 (69%)	93 (62%)	53 (58%)

Table 5.3: Bugs for which the FRM lies in the top-x results (Source)

Table 5.3 shows a summary of the results for the four projects. The results are relatively disappointing: even in the best case the FRM is in the top-10 results for only 50 bugs, and most results lie outwith the top-100, well beyond what a developer would realistically examine. In addition, for 106 of the bugs none of the methods returned are valid. For these bugs, while the approach is applicable, all of the methods which had to be fixed were first created after the version of the project for which the model was built. As such, no relevant methods could be returned for these bugs. For evaluation purposes, these methods were treated as if the FRM was placed at the last possible position, as a developer would have to look at every single method in order to discover that none are relevant. This is a shortcoming of this preliminary study, as the model for Source is only built once, from the initial version of the project. To remedy this, the

model should be updated whenever a change is made to the source, but, for this preliminary study, this was prohibited by the time required.

5.2.2.3 Bug description approach

To evaluate BUG, a classifier, initially empty, was trained for each method. In the order in which they were raised, the description of each bug was retrieved, the bug was classified as either relevant or not relevant for that method and the result recorded. Then the bug description was used to update the classifier, using the actual result i.e. not that predicted by the classifier. As well as those bugs from the original dataset, the additional training bugs were also included when updating the classifier, although the results of the prediction were not recorded and are not used in the analysis. This whole process was repeated for each method. This mimics the way the technique would be used in real life. Since this technique does not define an ordering amongst the methods classified as relevant, a conservative estimate was used: all methods which were not relevant but which were predicted as being relevant were assumed to be returned first, followed by those which were actually relevant. This in effect gives the worst possible position that the FRM could be presented at, giving a lower bound on the performance of the technique.

It is interesting to examine the number of methods returned as being relevant to each bug. The range is shown in Table 5.4. As shown, for all projects the approach predicts only a fraction of the methods in each project as being relevant. However, this is still almost always more than the number of methods that could be expected to be involved in any one bug. The number of methods returned has important implications when combining the two approaches.

	ArgoUML	JabRef	jEdit	muCommander
Minimum	3	14	4	0
Median	63	36	19	82
Maximum	302	46	70	183
Total number of methods	10806	2946	5263	3916

Table 5.4: Number of methods predicted as relevant (BUG)

Table 5.5 shows the positions of the FRMs when using bug description similarity. As expected, the results are worse than SOURCE. As stated however, this approach is not necessarily intended to be used alone, and one issue is that there is no order defined amongst the methods deemed relevant. By combining the two approaches we can in effect give an order to those methods.

	ArgoUML	JabRef	jEdit	muCommander
1	0 (0%)	0 (0%)	0 (0%)	0 (0%)
$\leqslant$10	1 (1%)	0 (0%)	5 (3%)	0 (0%)
$\leqslant$100	9 (10%)	12 (31%)	25 (17%)	7 (8%)
$\leqslant$1000	21 (23%)	12 (31%)	25 (17%)	19 (21%)

Table 5.5: Bugs for which the FRM lies in the top-x results (BUG)

5.2.2.4 Combined approach

Comb was then implemented as described in Section 5.2.1.3. Figure 5.6 shows the position of the FRM for each of the bugs in each project, sorted in ascending order. The dashed line shows the position given by Source, while the solid line gives the positions returned by Comb. For JabRef and jEdit there is a noticeable improvement, but for the other two projects the approaches are fairly similar. The flat tail at the end of each graph illustrates those bugs for which no relevant method could be returned, as previously explained.

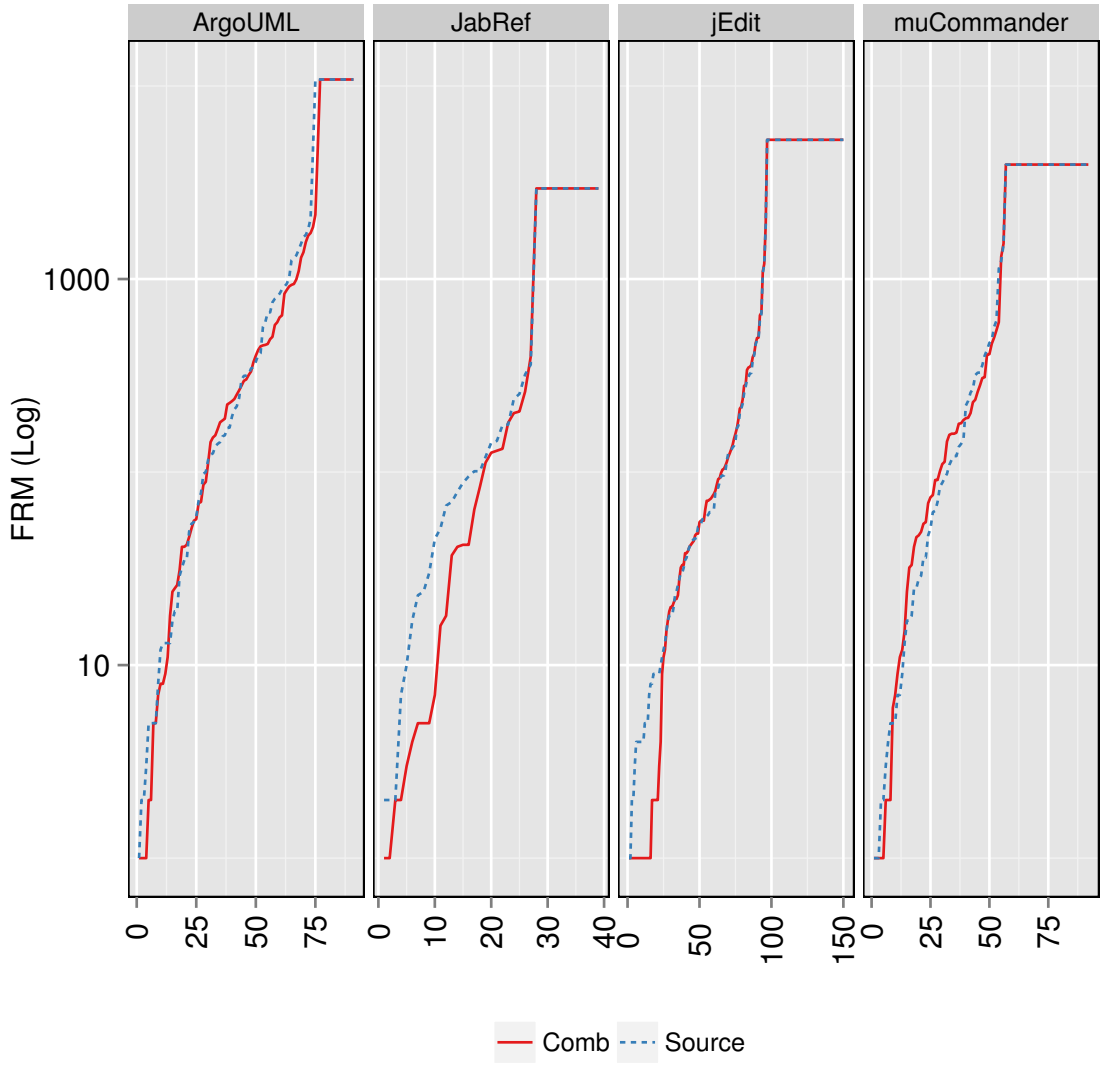


Figure 5.6: Position of FRM using Source and Comb. Each point on x-axis is a single bug, but position does not correspond to bug ID. Each approach is sorted individually: bug 1 for Source is not necessarily the same bug as bug 1 for Comb.

Table 5.7 shows the results for Comb in more detail. Compared to Source results seem much improved: for 27 bugs the FRM is in the first position compared to 6 for Source and 57 are in the top-10, compared to 50. The number of top-100 results has fallen from 144 to 140, but this difference is less important as these bugs already lie outwith the range of methods a developer would be likely to examine.

	ArgoUML	JabRef	jEdit	muCommander
1	4 (4%)	2 (5%)	16 (11%)	5 (5%)
⩽10	12 (13%)	10 (26%)	24 (16%)	11 (12%)
⩽100	29 (32%)	18 (46%)	65 (43%)	28 (30%)
⩽1000	67 (74%)	27 (69%)	93 (62%)	54 (59%)

Table 5.7: Bugs for which the FRM lies in the top-x results (COMB)

For our purposes, we are more interested in achieving improvements towards the top of the list of returned methods. From a user's perspective it is more useful for an approach to improve the rank of a relevant method from 25 to 2 than from 400 to 200 for example. In these circumstances it is common to examine the change in mean reciprocal rank as well as the position of the FRMs. The reciprocal rank for each bug is simply the inverse of the position of the FRM, and the Mean Reciprocal Rank (MRR) is the mean of the ranks for all bugs in a project. The MRR falls between 0 and 1, with 1 indicating that for every bug the FRM was at position 1. In order to evaluate the difference in MRR between the two approaches, the following hypotheses were tested, where MRR_S is the MRR of SOURCE and MRR_C is the MRR of COMB:

$$H_0 : MRR_S - MRR_C >= 0$$
$$H_1 : MRR_S - MRR_C < 0$$

To evaluate this difference, a paired t-test[2] was carried out on each project. Table 5.8 gives the value for MRR_S and MRR_C and the p-value for the test on each project. For all projects the MRR is improved between SOURCE and COMB, which is significant at the p<0.05 level for 2 of the projects.

	ArgoUML	JabRef	jEdit	muCommander
MRR_S	0.0461	0.0538	0.0532	0.0682
MRR_C	0.0718	0.1187	0.1371	0.0825
p-value	0.0932	0.0440	0.0001	0.2889

Table 5.8: Change in reciprocal ranks of FRM, measured by paired t-test(COMB)

5.2.3 Analysis

To illustrate the effect of the technique, it is worthwhile to examine in detail a number of examples.

5.2.3.1 ArgoUML Bug 4364

This bug has the summary "Dragging association onto diagram creates n-ary diamond" while the description contains a short list of steps to reproduce and a stack trace. Fixing the bug required changes to 4 methods, of which the first returned by SOURCE was `org.argouml.uml.`

[2]It should be noted that the differences in reciprocal ranks are not normally distributed. However, as the differences have finite variance t-tests can still be appropriately used as long as the sample size is large enough, often suggested to be more than 30 [OJ82].

`diagram.static_structure.ClassDiagramGraphModel.canAddNode(Object)` at position 672. As ArgoUML contains 10806 methods, this is actually a relatively good ranking but it is still far outwith what a developer would likely examine.

The classifier generated by BUG for this method at the time of this bug is shown in Figure 5.9. Diamonds represent decisions taken on whether or not the given word is present in the bug report, and rectangles contain the final decision, as well as the number of bug reports that terminated at that node. As shown, it has a number of branches, as there were multiple bugs associated with this method. This bug was predicted as relevant to the method due to the lack of the word 'panel' along with the presence of the word 'access' in the description. In fact BUG predicted 113 methods as being relevant of which this was the only one to actually be so. Due to the similarity between the bug report and the method however, when those 113 methods were sorted according to SOURCE, this method was then ranked at position 8, a marked improvement.

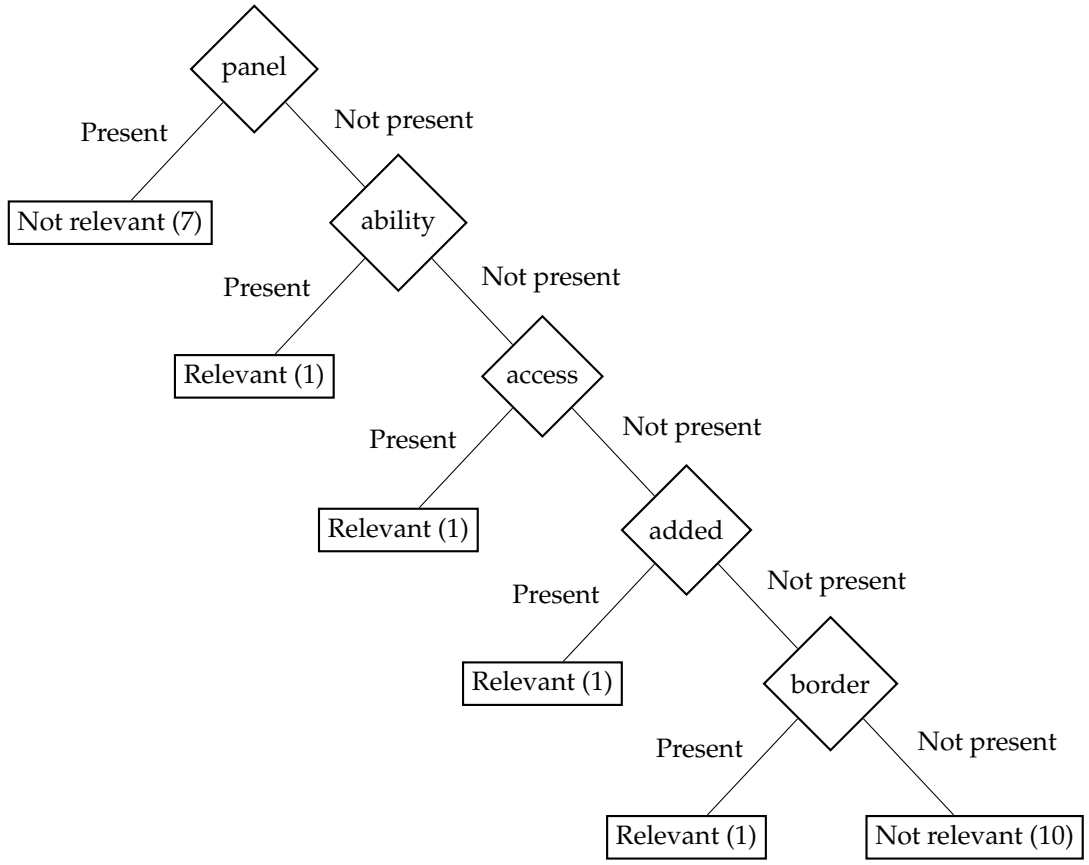


Figure 5.9: Classifier for ArgoUML method `org.argouml.uml.diagram.static_structure.ClassDiagramGraphModel.canAddNode(Object)`

Of the 4 bugs associated with this method the classifier correctly predicted 2. However, it also generated 15 false positives, having a detrimental effect on other bugs.

5.2.3.2 jEdit Bug 1747300

This bug has the summary "Enable customization of folding presentations", while the description elaborates on this only partially. This bug required changes to 17 methods to fix across a number of classes. The first of these to be returned by SOURCE was `org.gjt.sp.jedit.options.GutterOptionPane._save()` at position 59. As the classifier did not predict that this method was relevant for this bug, COMB actually decreased the rank of this method to 69. However, the position of another method `org.gjt.sp.jedit.EditPane.propertiesChanged()` was improved from 63 to 3, and this was returned as the FRM.

As shown in Figure 5.10, the classifier for the method is much simpler than that for ArgoUML Bug 4364. In addition, it actually bases it prediction on the *absence* of a word. In fact, many classifiers were very simple, and not all appeared to be making decisions 'intelligently'. Indeed, some classifiers actually always predicted that all bugs were relevant to a method, with no regards for the bug content. This behaviour will be revisited in Section 5.3.4.

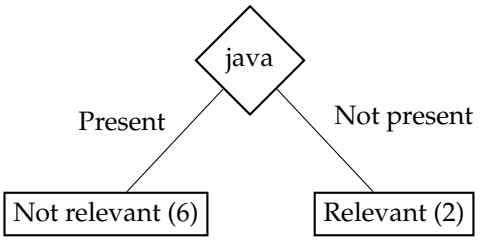


Figure 5.10: Classifier for jEdit method `org.gjt.sp.jedit.options.GutterOptionPane._save()`

5.2.3.3 JabRef Bug 1540646

COMB was not always effective however. For JabRef Bug 1540646, the FRM returned by SOURCE was `net.sf.jabref.TablePrefsTab.TablePrefsTab(JabRefPreferences,JabRefFrame)`, at position 88. For this bug, BUG predicted 39 methods, none of which were relevant, causing the FRM of COMB to fall to position 126. However, the amount by which this bug worsened is much smaller than the amount by which the other examples were improved.

5.2.3.4 Changes in position

Overall, there were more bugs for which the position of the FRM deteriorated using COMB than bugs where it improved. In general though, where COMB improved a bug's ranking it did so by a large amount, whereas for those where it worsened, it was normally by a small amount. Table 5.11 shows the range of change amongst bugs where the position of the FRM improved and worsened.

Figure 5.12 shows the positions returned by SOURCE as solid lines, and the positions returned for the same bugs by COMB as crosses. As shown, for a few bugs the position is vastly improved by COMB, illustrated by the crosses towards the bottom of the chart. For the remaining bugs the position is close to that of SOURCE approach, and has not been made substantially worse. This is

	ArgoUML	JabRef	jEdit	muCommander
Improved bugs	16	11	23	17
Minimum improvement	3	1	1	2
Median improvement	176	65	29	64
Maximum improvement	10737	302	167	1041
Deteriorated bugs	57	16	71	38
Minimum deterioration	5	5	4	6
Median deterioration	49	31	17	62
Maximum deterioration	284	39	48	153

Table 5.11: Changes in position for improved and deteriorated bugs (COMB)

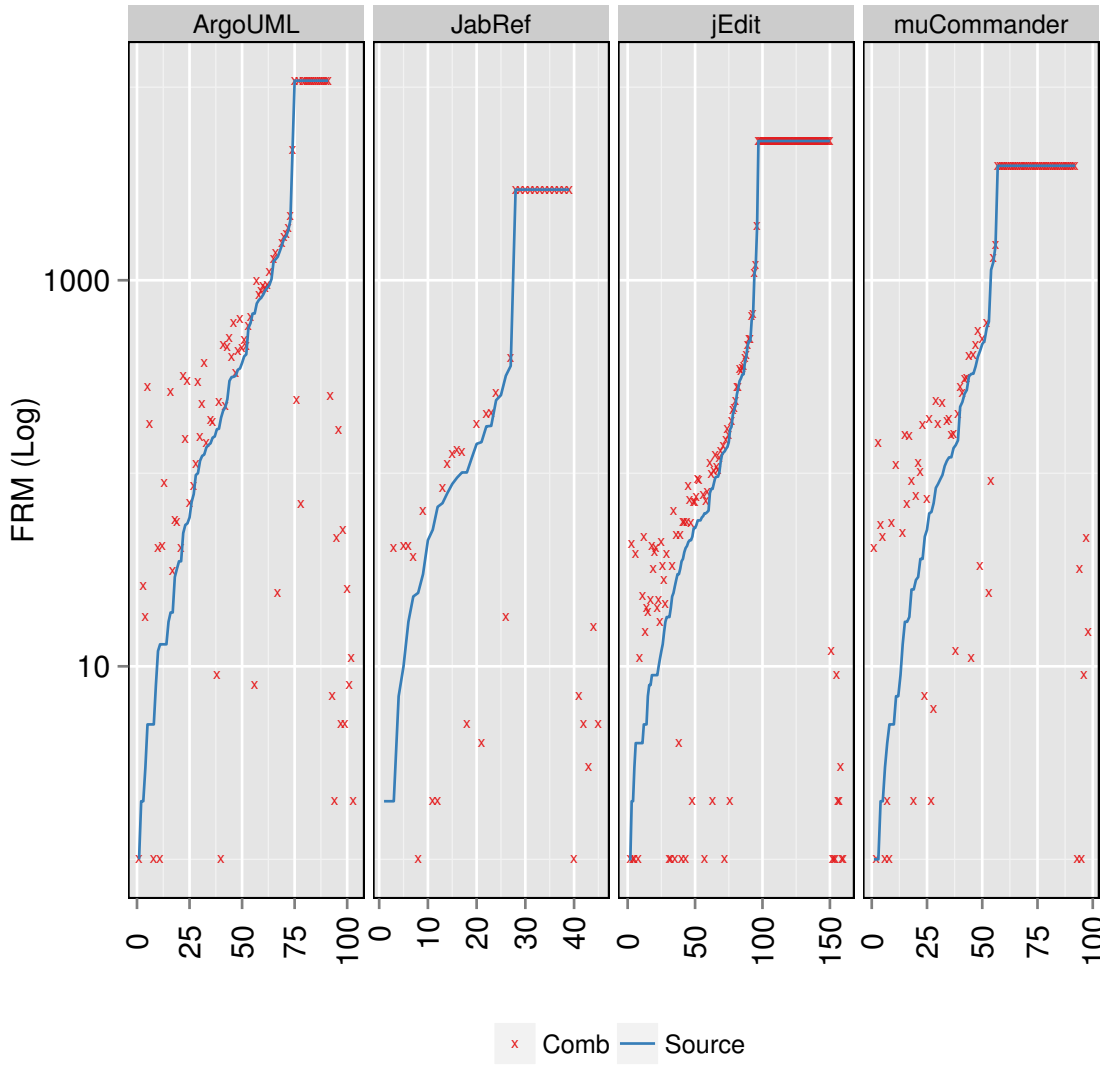


Figure 5.12: Change in position of FRM using SOURCE and COMB. Each point on x-axis is a single bug, but position does not correspond to bug ID. It is however consistent: bug 1 for SOURCE is the same bug as bug 1 for COMB.

because for most bugs BUG returns few results, and so therefore has minimal negative impact on the results of SOURCE.

5.2.4 Threats to validity

A number of threats to validity exist in this study. First, the methods identified as being relevant to a bug may not be accurate. The fact that a commit comment contained a bug number, and that the commit altered a particular method does not necessarily indicate the method is related to the bug. The bug number may be inaccurate, or the developer may have made an unrelated change at the same time. However, this technique has been shown to be mostly accurate, and the methods and bugs for the study have been drawn from a standard dataset available to all researchers to validate. Additionally, these bugs were only used in the training of the models, not in the evaluation.

In addition, the FRM may not be the best measure of how a developer would use a bug localisation tool. They may not examine the list of methods in the order presented, and there may be multiple methods relevant to a bug report. Nevertheless, this measure is widely-used and allows for comparison with other techniques.

The systems studied cover a variety of use cases and sizes but are all written in Java and all open-source, and the bugs selected may not be a valid sample of the project [BBA+09]. Further studies would be needed to examine if these results are generalisable to other systems.

The small number of projects could also be considered a threat to validity, and ideally replication with further projects would be carried out. This is not straightforward however; for many projects the process is problematic and fairly time-consuming. The fact that the systems had to be patched to obtain coverage information may also be a threat to validity, as the system being tested is no longer identical to that created by developers. However, this threat is minor: the changes made were small, and to the build system of the project rather than the source code.

Finally, there may be errors in the implementation. This has been lessened by releasing the code for scrutiny.

5.2.5 Conclusions

This section has proposed a technique for utilising the similarity of bug reports to enhance bug localisation techniques for determining methods relevant to a bug. While it appears that, when used alone, the technique is not particularly effective, it has been shown to improve other techniques when used in conjunction with them. In particular, when evaluated on 372 bugs, the combined technique is shown to have an improvement on all 4 projects, statistically significant for 2 of them. The number of bugs where the first relevant method presented to developers is the first result increased from 6 to 27, and those where it is in the top-10 from 50 to 57, showing that the approach is a viable method for enhancing bug localisation techniques. However, there are still many improvements that would need to be made before it is suitable for use by actual developers in a real-world scenario.

5.3 Other sources of information

As the preliminary study in the previous section has shown, combining textual approaches with information from previous bug reports appears to be a useful technique for improving bug localisation. This leads to the question: are there further additional sources of information which could also enhance bug localisation? This seems feasible; as well as previous bug reports there are a number of other sources of information which have not as yet been exploited by other research. These also may prove to be useful enhancements but like the information from bug reports, each of these sources is likely to be inapplicable for large numbers of bugs and may provide only weak evidence for bug localisation. The aim is to identify multiple such sources, such that each can provide small improvements over basic techniques, but together can make a more substantial improvement.

This section will investigate improvements to bug localisation techniques by combining information from the following sources:

- Method test coverage
- Textual similarity between bug reports and method contents
- Similarity between bug reports
- Previous bug counts for methods
- Stack traces

In addition to assessing these sources, this section will also investigate how the sources can be effectively combined, as the method used in the previous section was simplistic and cannot easily scale to integrating information from more than two sources.

5.3.1 Test coverage

Test coverage is a measure indicating the extent to which particular units of code are exercised by a project's test suite. Coverage can be measured at a number of different levels – such as the percentage of statements, branches or methods executed – but the basic concept is that test coverage can find areas of the system which may require more testing, or establish whether the testing on a project is 'good enough'. In theory, untested areas of code could potentially contain any number of problems while sections of code which are fully tested should contain few bugs. Of course, this scenario is just an ideal – bugs are still commonly found in areas which have been tested – and this leads to a simple question: are there more bugs reported in the areas with low test coverage?

Surprisingly, it appears that little research has been done in this area. Studying two large industrial projects, Mockus et al. [MNDT09] found that the probability of finding a bug in a file after release was inversely related to the level of test coverage in the file. Similarly, Berner et al. [BWK07] found an increase in the number of defects detected when they began using code coverage to identify untested functionality on a large industrial project, although both studies also identified a number of other possible factors. However, Piwowarski et al. [POC93] stated that anecdotal evidence at IBM suggested this was not always true: some users found that areas with high test coverage in fact had a higher incidence of bugs. While this might at first seem counter-intuitive, there may be a number of logical explanations: testers may concentrate their

efforts on the more complex, and potentially buggier, areas of code at the expense of those they perceive as 'simple', or possibly areas of code which change most frequently are most likely to be tested thoroughly but also the most likely to contain bugs. Whatever the explanation, it is difficult to draw conclusions in either direction from the existing research, and there is also a lack of tools or datasets to allow replication.

5.3.1.1 Investigation

To investigate the potential that test coverage has for bug localisation and answer the question posed earlier – 'are there more bugs reported in the areas with low test coverage?' – an investigation was carried out on three open-source projects: Ant [Ant], JMeter [Jme] and JodaTime [Jod]. These are all relatively popular Java projects, with existing automated test-suites and build systems, suitable for measuring test coverage. A system was then developed to find all the bug fixes for each project, determine the current version at the point the bug was reported, and run the full test suite at that point of time, recording the test coverage levels.

To simplify tool development and improve the efficiency of analysis, the complete revision history of each project was obtained and converted into a Git repository. Finding all bug fixes in each project was done automatically by parsing all commit comments and considering any string of consecutive digits as a bug number and therefore possibly referring to a fix, as described in earlier sections. As before, it is possible that some of the identified fixes will be false positives, but this risk was mitigated by discarding those potential fixes for which no corresponding bug report could be found. From the bug report the tool also identified the current revision of the repository at the time of the report. Also discarded from the evaluation were any bug reports for which no methods were changed (as described later in this section).

Table 5.13 shows the number of fixes and bugs for each project. Note that, as discussed in previous chapters, developers do not always successfully fix a bug first time. Sometimes, multiple attempts must be made to fix a bug, causing the number of fixes shown to be higher than the number of bugs. Table 5.13 also shows a summary of the time interval between a bug report and the corresponding fix (in terms of minimum, maximum and inter-quartile ranges). It is interesting to note that for each project there were a small number of extremely long-lived bugs and a much larger number of shorter bugs.

	Ant	JMeter	JodaTime
Fixes	1285	371	62
Bugs	991	164	59
Min	2 minutes	1 minute	4 hours
Q1	2 days	1 hour	3 days
Q2	3 weeks	2 days	3 weeks
Q3	1 year	3 weeks	12 weeks
Max	9 years	6 years	5 years

Table 5.13: Project details and interquartile ranges of bug lifetimes

In order to establish the level of test coverage and try to determine if there exists a difference between the coverage level of buggy methods and that for non-buggy methods, the tool checked

out the project at the time the bug was reported and ran the corresponding automated tests. This was far from a smooth task: poor documentation, missing libraries and older versions of required tools, made it necessary to patch the build system of each project (and the patch required had to be altered over time, to match changes in the project). Indeed, a number of early bugs from Ant were excluded as the required version of code could no longer be built successfully. As none of the projects already collected coverage information, patches were also used to gather this information by modifying the build system and using Emma [Emm] to instrument the project code after it was built, and then collect coverage information as the tests were then run. Coverage information was collected as the percentage of basic blocks – a sequence of Java bytecode instructions containing no jumps – executed within a method.

The coverage of existing tests can only be used to aid in bug localisation if projects actually have existing tests, and these tests cover a reasonable proportion of the codebase. For each project, the top row of Figure 5.14 shows the number of tests executed for each bug report. Coverage information is shown in the middle row of Figure 5.14. For all projects, coverage is consistently above 50% (with a few blips, explained in the next paragraph). It should therefore be possible to use this information to assist in bug localisation.

As mentioned previously, running the tests at each revision was not straight-forward, and there were occasional problems executing tests which could not be worked around. The occasional dips in the number of tests in the top row of Figure 5.14 were caused by revisions which could not be successfully built and tested. There is also a permanent reduction in JodaTime caused by a restructuring of the project to remove large sections of the codebase into entirely separate projects (these sections were not part of the project core and neither the coverage information nor bugs were included in this analysis, but unfortunately the tests could not be prevented from running).

The bottom row of Figure 5.14 shows the number of tests that failed or caused errors in each revision. Note that some of these are legitimate, caused by developers checking in broken revisions. Unfortunately however some are caused by changes made by the patches. While efforts were made to reduce the number of failing tests as much as possible, many of the broken tests could not be fixed without significant effort, or worse, without altering the behaviour of the code. Overall around 2% of tests failed. The majority of tests succeeded and these failures should not have a significant adverse effect on the bug localisation process. Note that if it was not possible to run the tests, coverage information from the last successful test run was used (this was deemed to be preferable to giving no answer at all).

Finally, to determine the difference in coverage between methods with bugs and those without, it was also necessary to identify the methods containing bugs. Using the same approach as the preliminary study, every bug-fixing revision was checked out and compared to the previous version and any changed methods were assumed to be part of the fix and therefore related to the bug. Of course, this may not always be true as developers sometimes make unrelated changes at the same time as bug fixes, but manual validation of the fixes would be needed to eliminate these. The time required for this validation would have greatly reduced the number of bugs it was feasible to examine. Not every bug fix made changes to a method: some fixes were to class-level attributes or to non-Java files, and in some cases the methods required to fix the bug were created from scratch. As mentioned previously, only bugs in which at least

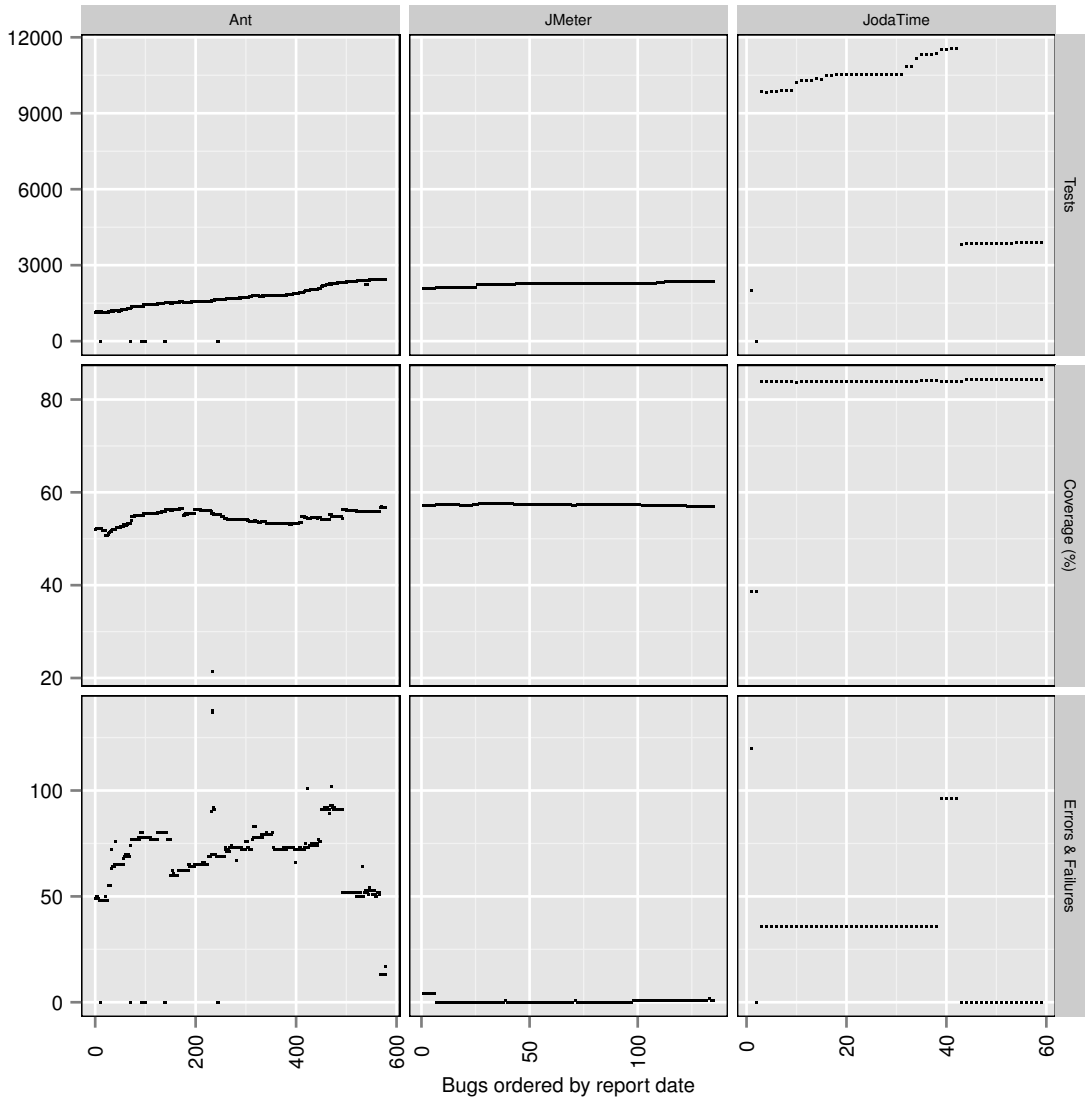


Figure 5.14: Top: Tests run for each bug; Middle: Coverage level for each test run; Bottom: Errors and failures in each test run

one method which existed at the time of the report was updated were considered as part of the evaluation.

The number of methods involved with each bug fix are shown in Table 5.15. From this it can be seen that the majority of bug fixes involved changes to a relatively small number of methods. In Ant for example, 349 bug fixes involved changing just a single method. A further 338 required changes to between two to five methods. After this point the number of fixes which involve many methods drops off, although there are still a reasonable number.

Number of methods	1	2-5	6-10	11+
Number of bugs				
Ant	349	338	135	169
JMeter	35	57	34	38
JodaTime	5	20	15	19

Table 5.15: Number of methods related to each bug

5.3.1.2 Analysis

For test coverage information to be a viable information source for bug localisation, we need to examine the difference in coverage levels between methods which are related to a particular bug and those which are not. Figure 5.16 shows this difference ordered by ascending levels of coverage of related methods. As may be seen, the coverage level of related methods varies widely, and is neither consistently above or below that of unrelated methods. This largely mirrors the results found by other researchers, limited as they were [POC93, BWK07, MNDT09].

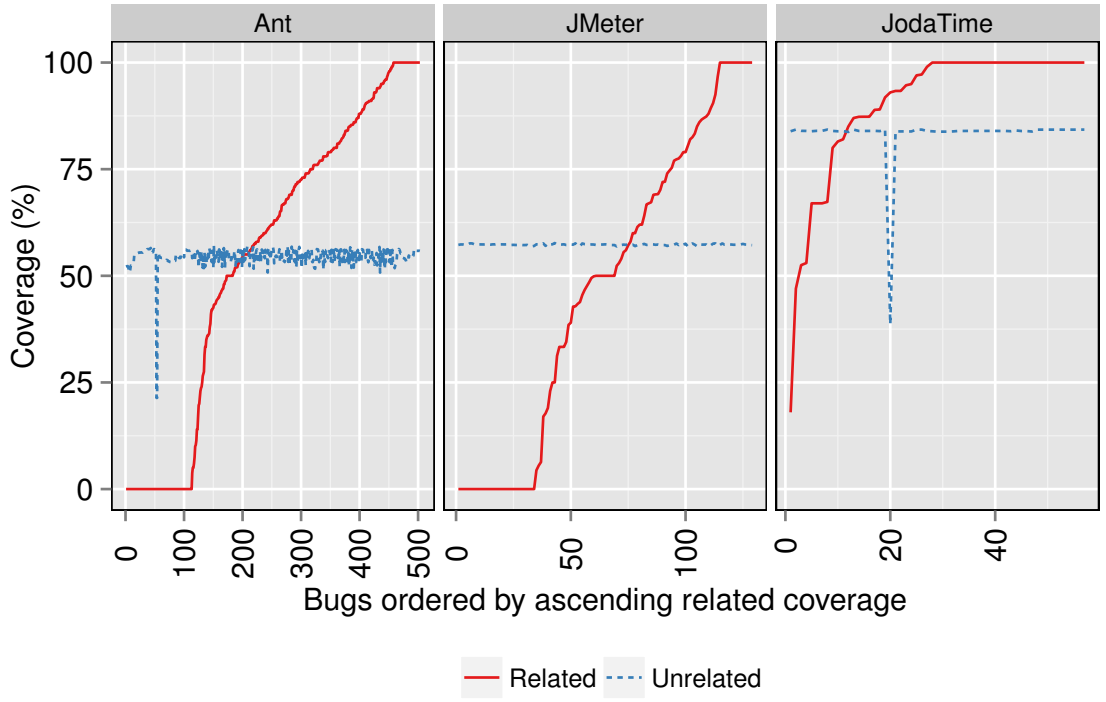


Figure 5.16: Coverage for methods related and unrelated to a bug

Table 5.17 summarises this information and shows the mean levels of coverage in related and unrelated methods for each project. For JMeter the coverage level amongst methods related to bugs is indeed lower than those unrelated to the bug, by 10.1%. However, the opposite is true for JodaTime (coverage is 7.1% *higher*) and Ant (although the difference here is minimal). The reason behind these differences is unclear however. To test whether the differences in coverage level are significant, a paired Wilcoxon rank-sum test was carried out for each project, where each pair is the related and unrelated coverage for a single bug. This test requires no assumptions about the distribution of the coverage levels or on the size of each distribution, and since, as discussed previously, there are valid arguments for coverage being higher or lower in related methods, no assumptions were made about the direction of change. The results of the tests are also shown in Table 5.17, and show that the difference is significant at the 0.05 level for both JodaTime and JMeter.

	Ant	JMeter	JodaTime
Related (%)	54.80	47.16	90.32
Unrelated (%)	54.47	57.30	83.23
p-value	0.74	0.00	0.00

Table 5.17: Coverage levels for projects

While the difference between the coverage in related and unrelated methods is not large, it is still present, and it could therefore be possible to utilise this information for bug localisation. The fact that for JodaTime the coverage is higher in related methods, while the opposite is true for JMeter, is not actually important for the purposes of bug localisation. The models trained to combine the varying sources of information, explained later in Section 5.4, are project-specific, and should be able to account for this difference. Similarly, even if the difference in coverage for Ant is small and not significant, the models should learn to disregard this information.

While it is not necessary that differences in coverage levels be consistent across projects, ideally they should remain consistent within a single project across time. Figure 5.18 shows the cumulative running mean coverage in related and unrelated methods over time. For the two projects with significant differences in coverage levels, after some initial fluctuation these differences seem to stabilise. It is also interesting to note that earlier periods in Ant's history demonstrate more accentuated differences in related/unrelated method coverage level than later revisions.

In order to actually utilise this coverage information, the coverage level is used as a probability. The difference in direction is accounted for by the combination technique described later, which can weight the probability positively for some projects and negatively for others. Unlike the information gained from the bug report and the source code presented in earlier sections, this prediction is independent of the bug report under consideration – two bugs reported at the same time will result in the same set of predicted relevant methods.

5.3.2 Textual information

As described in Section 5.2, one important source of information comes from comparing the source code of the project to the text of the bug report. This evaluation will utilise this information in a similar manner to before.

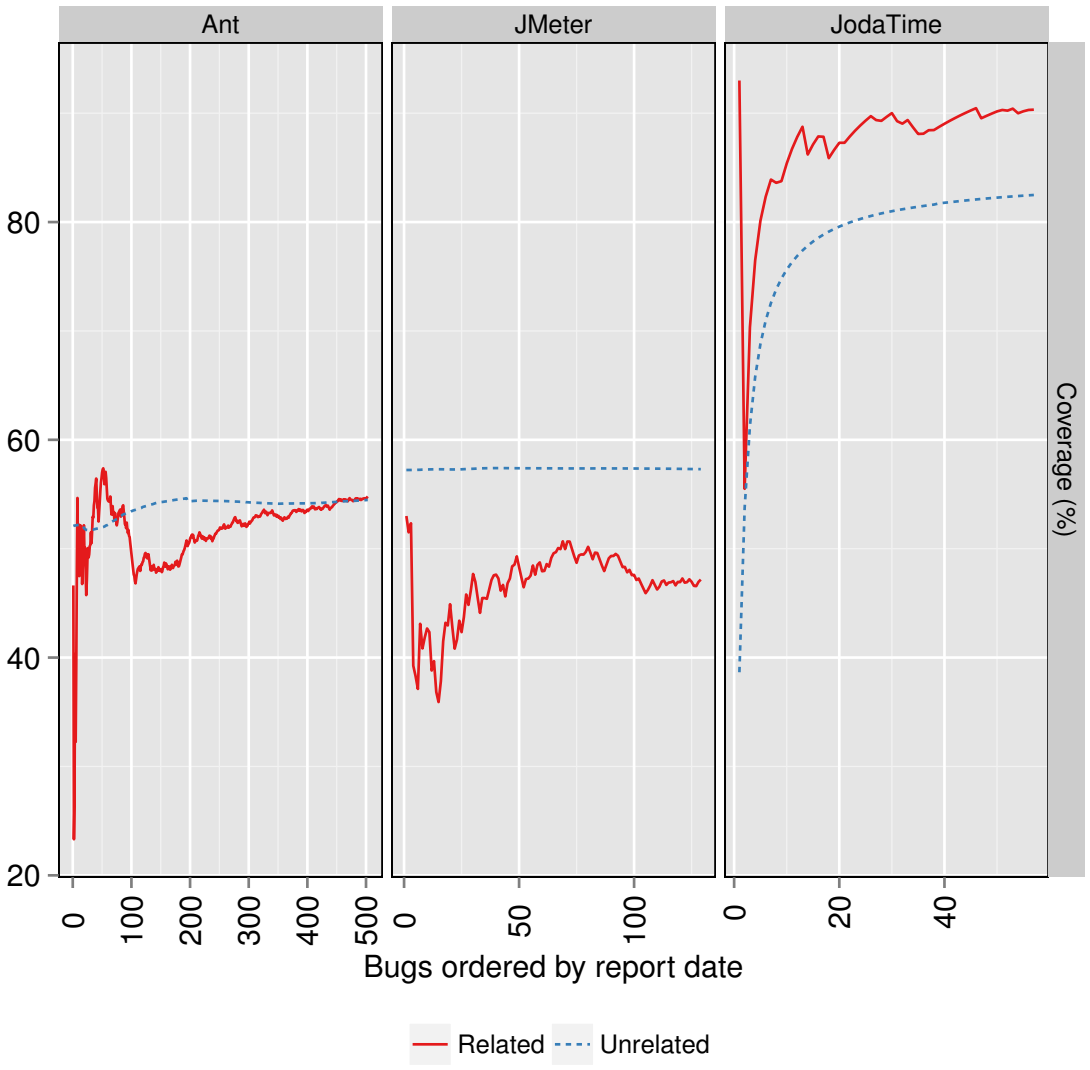


Figure 5.18: Cumulative running mean coverage in methods related and unrelated to a bug

5.3.3 Similarity of bug reports

Again, Section 5.2 showed that information about previous bug reports could be useful for bug localisation. In that section however, this information was combined with that from the source code approach by partitioning the methods into two groups: those the approach predicted as relevant and those that it did not. Each group was then sorted individually by the similarity of the method to the bug report as determined by the source code approach and the ranked list of methods presented to the developer. While this approach was shown to improve the results returned by the source code approach alone, partitioning does not scale to integrating information from more than two sources. Consequently, for this evaluation, the bug similarity component was altered slightly to output for every method a probability of the method being related to the bug in question. How this probability is combined with the other sources of information will be explained in Section 5.4.

5.3.4 Previous bugs

In the preliminary study detailed in Section 5.2, most classifiers were simple, with some predicting all bugs as relevant to a particular method, regardless of the description. For those classifiers, the only reason they worked was because methods which had had bugs in the past appeared more likely to have other bugs. This should not be surprising: previous research has shown that bugs tend to recur in the same location [FO00] and it is well-established that the distribution of bugs within systems is not uniform [CMM+11]. However, this fact had not been utilised by previous textual bug localisation techniques.

To test if the benefits of Comb were due to this effect, a new approach was built and tested on the data used in the preliminary study, the previous bug approach (Prev). Rather than partitioning the results with classifiers, the methods were simply partitioned into those that had previously had bugs and those that had not. As before, each partition was then sorted by the similarity of the methods to the bug report.

Table 5.19 shows that there are slightly fewer bugs in the first positions using Prev than Comb, shown in Table 5.7 (repeated below), but otherwise performance is similar. Figure 5.20 also shows that Prev is more varied than Comb. As soon as one bug is found in a method, Prev will predict that every bug from then on is related to that method. It therefore predicts far more methods as being related to a bug than Comb. Because of this, it can have a larger detrimental effect on the position of bugs it gets incorrect, while making it more likely to improve other bugs.

	ArgoUML	JabRef	jEdit	muCommander
1	3 (3%)	1 (3%)	11 (7%)	5 (5%)
⩽10	7 (8%)	12 (31%)	40 (27%)	16 (17%)
⩽100	28 (31%)	20 (51%)	62 (41%)	31 (34%)
⩽1000	51 (56%)	27 (69%)	96 (64%)	56 (61%)

Table 5.19: Bugs for which the FRM lies in the top-x results (Prev)

	ArgoUML	JabRef	jEdit	muCommander
1	4 (4%)	2 (5%)	16 (11%)	5 (5%)
⩽10	12 (13%)	10 (26%)	24 (16%)	11 (12%)
⩽100	29 (32%)	18 (46%)	65 (43%)	28 (30%)
⩽1000	67 (74%)	27 (69%)	93 (62%)	54 (59%)

Table 5.7 (Repeated): Bugs for which the FRM lies in the top-x results (Comb)

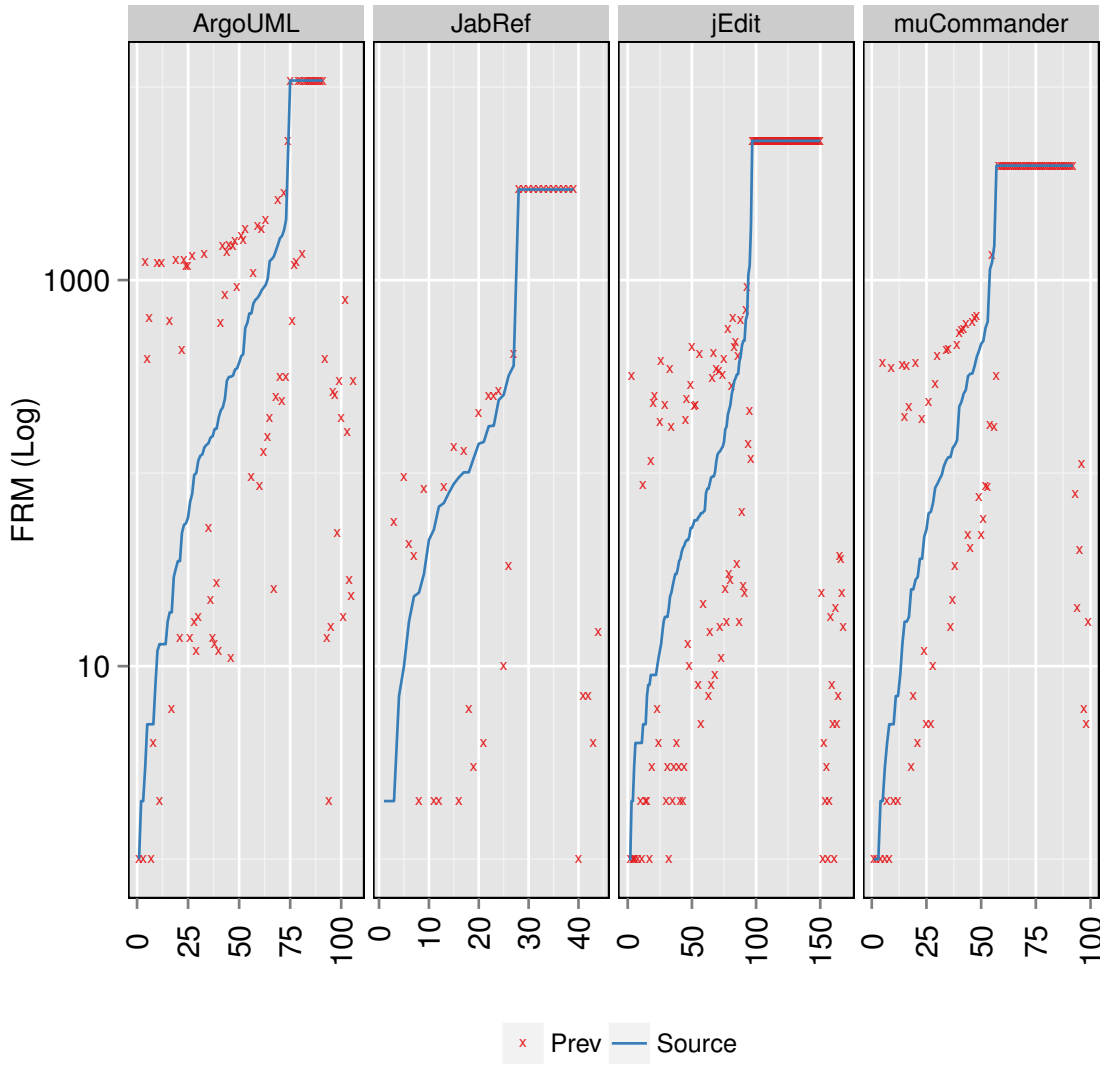


Figure 5.20: Change in position of FRM using Source and Prev. Each point on x-axis is a single bug, but position does not correspond to bug ID. It is however consistent: bug 1 for Source is the same bug as bug 1 for Comb.

Overall, Table 5.21 shows that PREV has a similar positive effect on MRR as COMB, significant for 2 of the projects. As discussed, this information is simple and easy to obtain. Even without the changes proposed in the rest of this chapter, it appears that utilising information about the number of previous bugs in a method may be a cheap and effective way to enhance any bug localisation technique.

	ArgoUML	JabRef	jEdit	muCommander
MRR_S	0.0461	0.0538	0.0532	0.0682
MRR_P	0.0619	0.1185	0.1402	0.0943
p-value	0.1092	0.0280	0.0000	0.0768

Table 5.21: Change in reciprocal ranks of FRM, measured by paired t-test(PREV)

To incorporate this information, one of the components of the combined approach simply rates methods which have been involved in more bugs than others as being more likely to be related to the current bug. Note that unlike the previous two sources of information, but similarly to the coverage information, this source is not specific to the bug being investigated: at any given time, the technique will predict the same set of methods as highly relevant regardless of the actual description of the bug.

5.3.5 Stack traces

When reporting bugs, users will sometimes include a stack trace, usually of an exception that occurred and was displayed to them by the application. Schröter et al. [SBP10] found that when such a stack trace was included in the bug report of a fixed bug, at least one of the methods in the stack trace was changed to fix the bug around 60% of the time. They also found that the higher up the stack trace a method appeared, the more likely it was to be changed. This makes stack traces a useful indicator to a developer of where to start looking for a bug location. It is not perfect however: if it was, then there would be little point in using an automated bug localisation technique in the presence of stack traces, as the developer would already have a more accurate list of possible bug locations.

Where stack traces are present the depth of each method in the stack trace is used as a possible source of information, by taking the inverse of the method's position in the stack trace. Unfortunately however, stack traces are rarely present in bug reports, often appearing in as few as 10% of reports, as shown in Chapter 4 and by other research [BPZ08, SBP10]. As such, they are not suitable as a sole source of information for bug localisation, but only in combination with other sources as described here.

5.4 Combining sources

To combine these five sources of information, each source was configured to output a value between 0 and 1 for each method, indicating the likelihood that the method is related to the bug in question. Essentially, this value is treated as a probability. More precisely these were:

`Cov`: The block-level coverage of the method
`Source`: The cosine similarity between the bug description and the method

`Bug`: The probability output by the bug language classifier as to whether the bug is related to the method, based on past bug reports in the system

`Prev`: The number of previous bugs that the method has been involved in, divided by the total number of bugs in the system

`Sta`: The inverse of the position of the method in any stack trace in the bug report

Whenever a new bug is reported, a linear regression model is trained, using the information about previous bug fixes in the system as training data. The linear regression is implemented using scikit-learn [Sci]. Each input represents the relationship between a method and a fixed bug, and consists of the values for each of the five features at the time that bug was reported with an output of 1 or 0, based on whether the method was related to the bug. After the model has been trained, input data is created for each method in the system based on the *current* values of the five features. This data is then fed to the model. The output of the model is then the probability for each method that it is related to the new bug that has been reported. The methods are then ordered in descending order of this probability to produce the ranked list that would be given to a developer.

Initially, when a new bug is reported, there is very little previous data available to train the model. However, as time passes and more bugs are fixed, more data is available for training. Over time therefore, the models should grow more accurate, and the weights allocated to each feature should become better estimates of how accurate that feature is. This allows distinct models to be produced for each project, which is desirable due to the differences between projects (for instance as seen in Section 5.3.1.2, the relationship between test coverage and bug likelihood varies between projects).

As before, using all of the negative examples – methods which are *not* related to a particular bug fix – results both in unbalanced classes and prohibitively long training times. To deal with these problems, random under-sampling was used, as it is for building the classifiers for `Bug`. Due to the increased number of bugs and methods, the percentage chance of including a negative example was reduced to 1%. This threshold will be explored further in Section 5.6.4.

5.5 Evaluation

To evaluate the performance of the bug localisation techniques, I ran an investigation on the same bugs and projects presented in Section 5.3.1.1 using a similar process – the bug reports are considered in time order and in the context of the revision for which they were originally reported. This means that the techniques that rely on historical data of some sort, such as number of previous bugs and bug report similarity, initially have very little information to work on, but it also simulates the way that the technique would be applied in practice.

For each bug the position of the FRM was measured – as earlier, this is the first method returned which appears in the list of the methods which were actually changed to fix the bug. Note that while the results in Section 5.3.1.2 suggest that it may not be appropriate to use coverage information to predict bugs on Ant, the technique for combining the values should be robust enough to handle this, giving coverage information a low weight for Ant if it is not a reliable indicator of bugs.

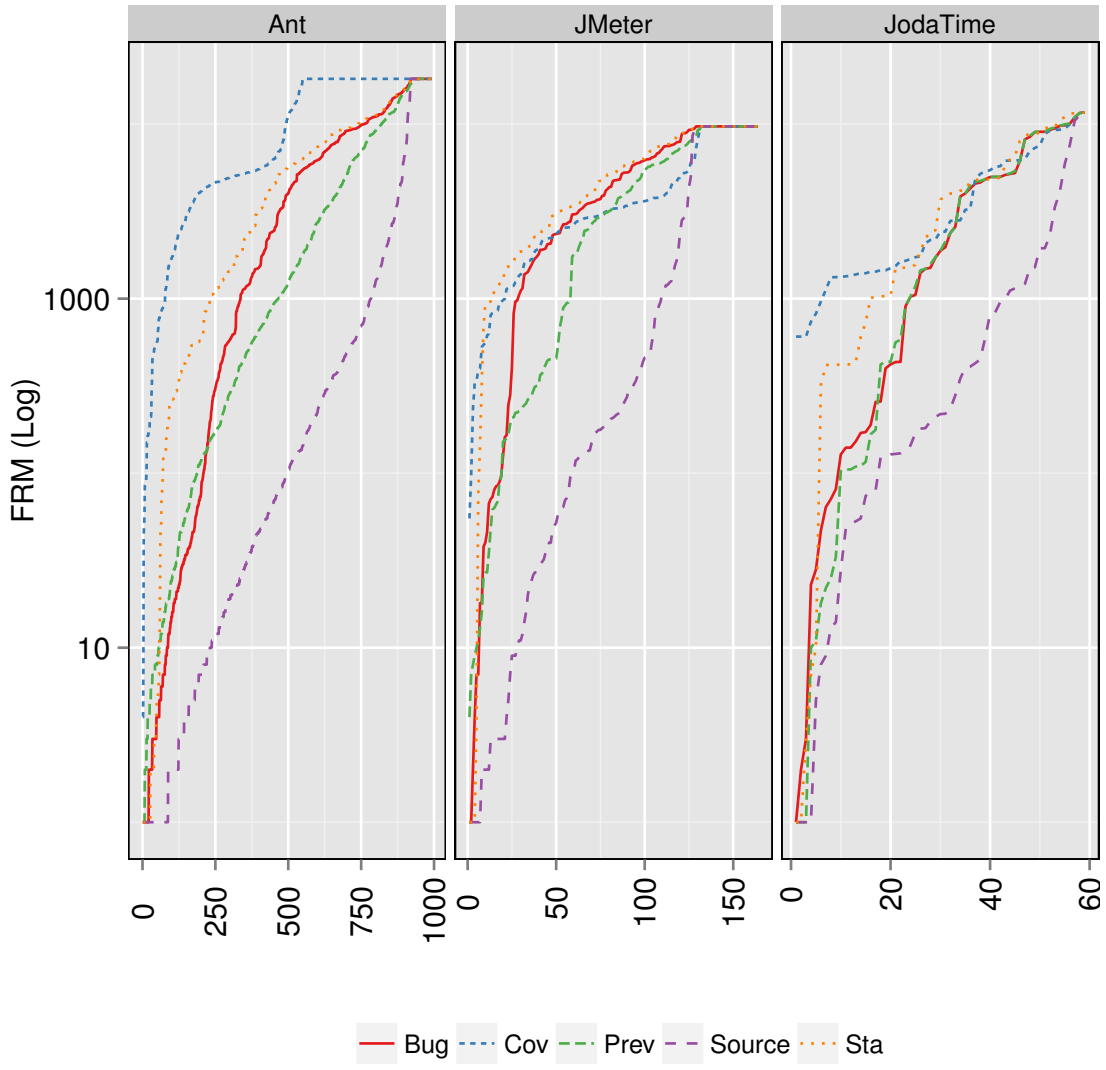


Figure 5.22: Position of FRM for each source of information. Each point on x-axis is a single bug, but position does not correspond to bug ID. Each approach is sorted individually e.g. bug 1 for SOURCE is not necessarily the same bug as bug 1 for BUG.

Before looking at the combined results, it is worth looking at the results given by each individual component. The positions returned by each of the individual components for each bug are shown in Figure 5.22. This chart shows that, as expected, for each project the source code information appears to be a far better predictor of which methods are relevant to a bug than any of the others. Table 5.23 also shows for each of the components how many of the results were returned in the first position and in the top 10 positions, which confirms this result.

	Ant	JMeter	JodaTime
Cov	0 (0%)	0 (0%)	0 (0%)
Source	87 (9%)	7 (4%)	4 (7%)
Bug	21 (2%)	2 (1%)	1 (2%)
Prev	7 (1%)	0 (0%)	3 (5%)
Sta	26 (3%)	4 (2%)	2 (3%)

	Ant	JMeter	JodaTime
Cov	3 (0%)	0 (0%)	0 (0%)
Source	237 (24%)	28 (17%)	7 (12%)
Bug	87 (9%)	6 (4%)	3 (5%)
Prev	59 (6%)	5 (3%)	4 (7%)
Sta	57 (6%)	5 (3%)	5 (8%)

Table 5.23: Number of bugs for which the FRM is the top result (left), and in the top-10 results (right)

While the source code approach appears to be the most effective approach, some of the other approaches still perform well. It is important also to note that the other approaches are not just correctly predicting smaller subsets of the ones identified by the source code approach; instead they are correctly predicting methods for a different set of bugs. For example, there are 28 bugs across all three projects for which stack trace information gives the first relevant method in position 1 but no other approach does. This is also true for information from previous bug reports which gives 22 unique top results, and the count of previous bugs which gives 9 unique top results. This suggests that combining the approaches could be an effective way of getting the best possible performance, if a method can be found to adequately combine the information or determine for any given bug which source of information is most likely to give the best answer.

As before, when comparing bug localisation approaches, the most important thing is how the technique performs towards the top of the list of results. So, to compare the individual approaches we again examine the MRR of each component. The MRR values for each component are shown in Table 5.24 and again show the source code approach to be superior to the others.

	Ant	JMeter	JodaTime
Cov	0.001	0.001	0.001
Source	0.139	0.091	0.082
Bug	0.044	0.021	0.035
Prev	0.024	0.008	0.058
Sta	0.038	0.026	0.044

Table 5.24: MRRs for each source of information

For the rest of the evaluation, the results for the combined approach described in Section 5.4 will be compared against just the source code approach. Out of the five approaches, it is the one that is based on those widely used in existing literature [MSRM04, PGM+06, LKE10, NNAK+11, RK11, WLXJ11], and also the most successful individual component. Figure 5.25 shows the overall performance of those two techniques. From this graph it would appear that

the combined approach performs consistently better for Ant and slightly better for JMeter, but the difference is less clear for JodaTime.

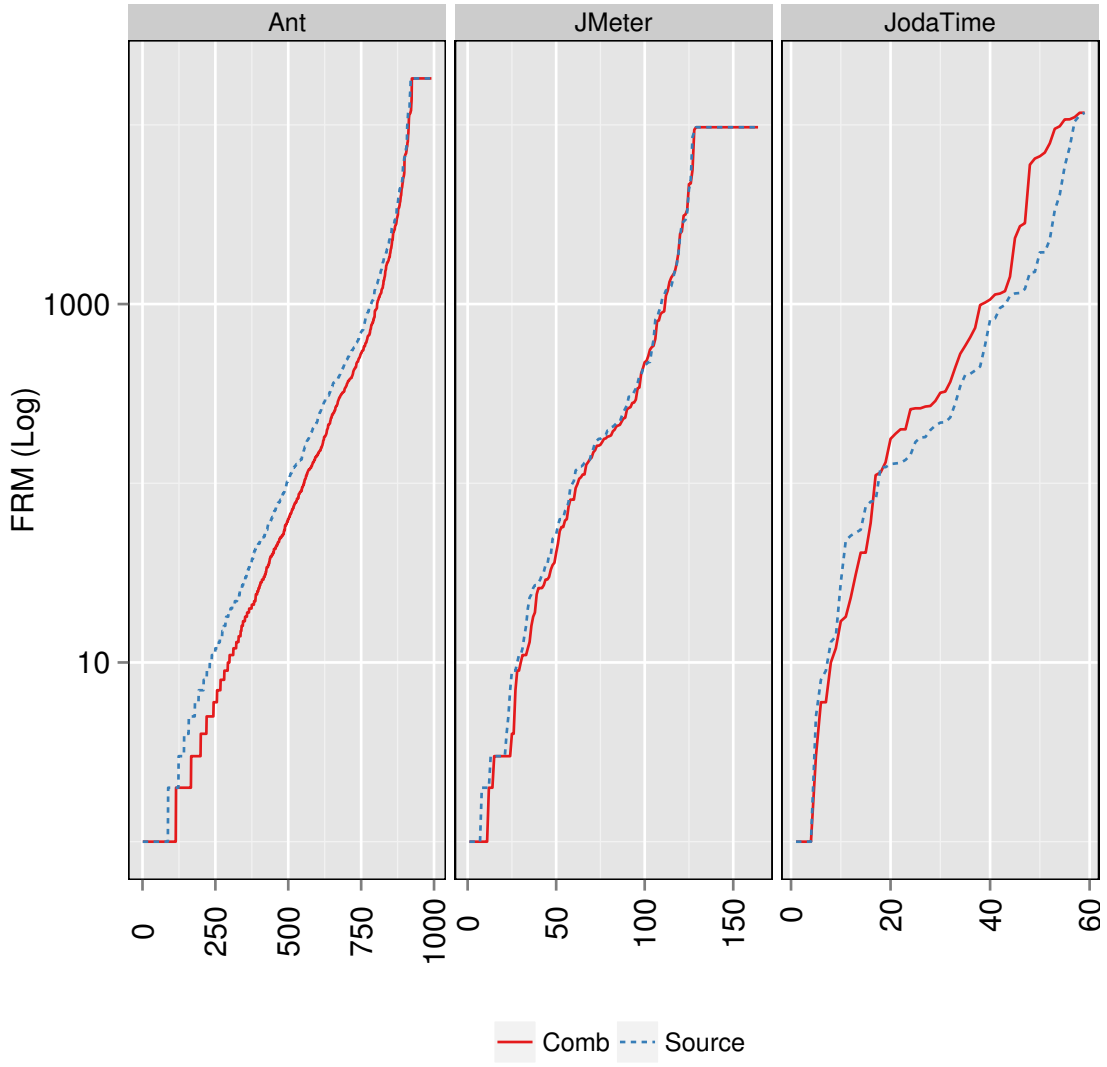


Figure 5.25: Position of FRM using SOURCE (solid) and COMB (dashed). Each point on x-axis is a single bug, but position does not correspond to bug ID. Each approach is sorted individually: bug 1 for SOURCE is not necessarily the same bug as bug 1 for COMB.

Table 5.26 shows the MRR for the results returned by the combined approach. As expected, it shows an improvement for all three projects. As before, *t*-tests were carried out on the change in reciprocal ranks between the two approaches, although these have only shown the change for Ant to be significant.

This difference in approaches is more clearly illustrated in Figure 5.27 which shows the position given by the source code approach on the line, with a cross marking the performance on the same bug for the combined approach. This goes some way towards explaining the changes in MRR values: for Ant and JMeter, when the combined approach returns a worse position than the source code approach, it is usually by a small amount (even allowing for the log scale), but when it improves the performance it is normally by a larger amount. The same is not

	Ant	JMeter	JodaTime
MRR_S	0.139	0.091	0.082
MRR_C	0.180	0.111	0.088
p-value	0.000	0.071	0.440

Table 5.26: Change in reciprocal ranks of FRM, measured by paired t-test(Comb)

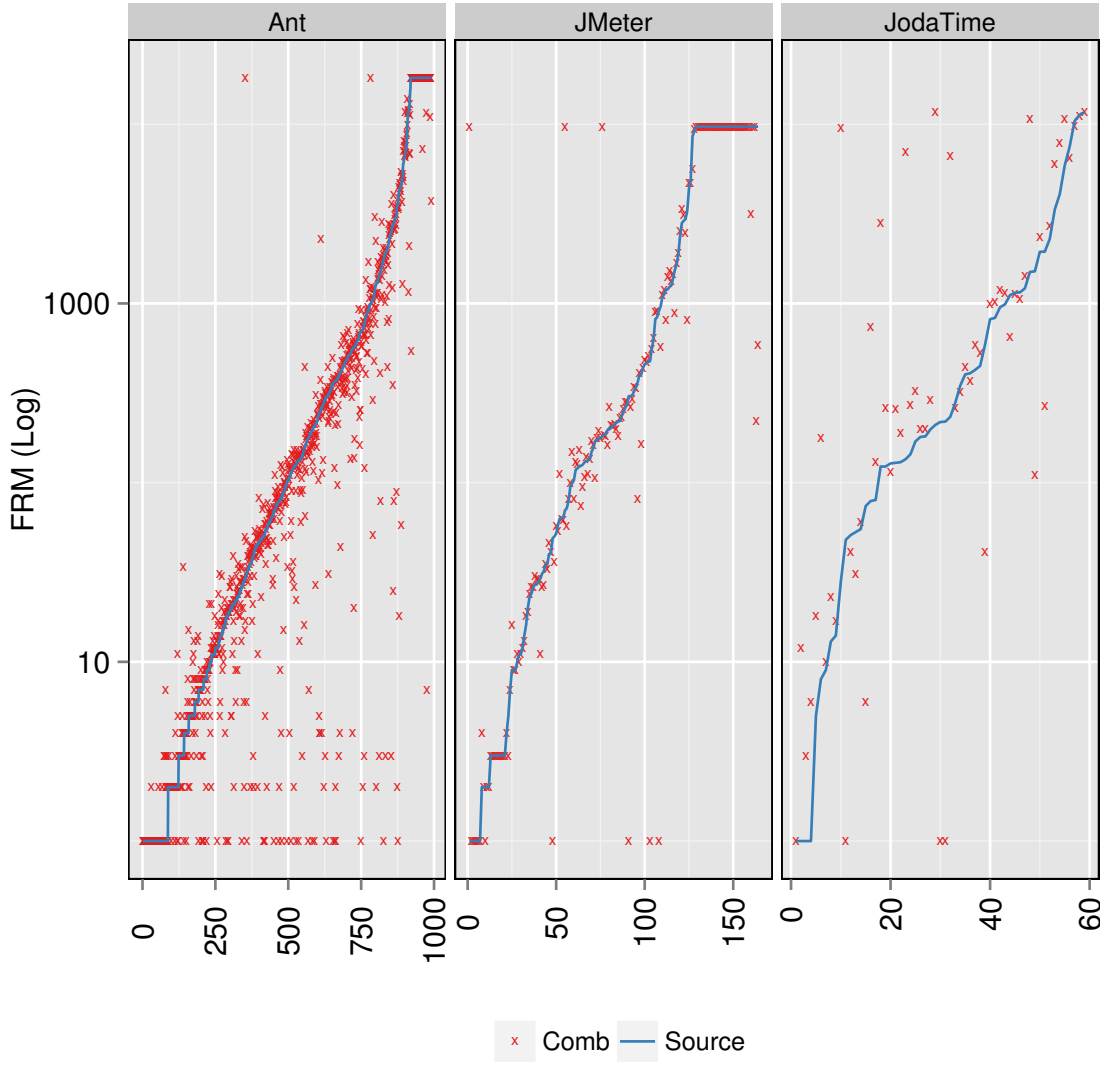


Figure 5.27: Change in position of FRM using Source (lines) and Comb (crosses). Each point on x-axis is a single bug, but position does not correspond to bug ID. It is however consistent: bug 1 for Source is the same bug as bug 1 for Comb.

necessarily true of JodaTime, where for a number of bugs the performance of the approach is worse by a considerable amount.

Table 5.28 also shows a summary of the changes on each project. As previously suggested, for Ant and JMeter, when the combined approach improves the position of the FRM for a bug, it is usually by a higher amount than when the position is made worse. Again, this is not the case for JodaTime.

	Ant	JMeter	JodaTime
Improved bugs	395	42	19
Minimum improvement	1	1	10
Median improvement	52	34	93
Maximum improvement	18158	9480	1677
Deteriorated bugs	328	59	38
Minimum deterioration	1	1	1
Median deterioration	11	12	156
Maximum deterioration	18135	9700	11471

Table 5.28: Changes in position for improved and deteriorated bugs (Comb)

As before however, the overall aim is to improve the number of bugs where the FRM is returned towards the top of the results. For comparison with Table 5.23 (repeated below), Table 5.29 shows the range of results returned by the combined approach. As shown, the number of bugs for which the first relevant method is in the top position has increased considerably, from 98 for the source code approach to 129 for the combined approach, across all projects. Similarly, the number of results in the top-10 has increased from 272 to 336.

	Ant	JMeter	JodaTime
Cov	0 (0%)	0 (0%)	0 (0%)
Source	87 (9%)	7 (4%)	4 (7%)
Bug	21 (2%)	2 (1%)	1 (2%)
Prev	7 (1%)	0 (0%)	3 (5%)
Sta	26 (3%)	4 (2%)	2 (3%)

	Ant	JMeter	JodaTime
Cov	3 (0%)	0 (0%)	0 (0%)
Source	237 (24%)	28 (17%)	7 (12%)
Bug	87 (9%)	6 (4%)	3 (5%)
Prev	59 (6%)	5 (3%)	4 (7%)
Sta	57 (6%)	5 (3%)	5 (8%)

Table 5.23 (Repeated): Number of bugs for which the FRM is the top result (left), and in the top-10 results (right)

	Ant	JMeter	JodaTime
1	114 (12%)	11 (7%)	4 (7%)
⩽10	298 (30%)	30 (18%)	8 (14%)
⩽100	553 (56%)	62 (38%)	16 (27%)
⩽1000	804 (81%)	111 (68%)	38 (64%)

Table 5.29: Bugs for which the FRM lies in the top-x results (Comb)

5.6 Discussion

Section 5.5 showed that the combined approach could be successful at improving the performance of bug localisation techniques. There are a number of elements which are worth examining in more detail however, which will be investigated in this section. It will also examine some possible improvements that may be made to the approach.

5.6.1 Linear regression

A linear regression may not (indeed, probably will not) be the best method for combining the results from the various sources of information, which is explored in Section 5.6.5. However, it was chosen as an initial starting point, which would be easy to experiment with and to illustrate. Table 5.30 shows the weights given to each source of information in the linear regression for the final bug in each project. The direction, positive or negative, of each weight is important. As expected, for Ant and JodaTime, where coverage is higher in related methods, coverage is weighted one way whereas for JMeter is is weight the other direction. The magnitude of the weight gives some indication of how important each feature is. Note however that, as this data may violate some of the standard assumptions for linear regression, we cannot draw exact conclusions from these values. In particular, the values for each feature are *not* independent of one another. Despite the weight afforded to it, we therefore cannot, for example, definitely conclude that PREV is necessarily the 'most important' source of information.

	Ant	JMeter	JodaTime
Cov	-0.01	0.01	-0.01
Source	1.29	1.81	0.74
Bug	0.42	0.13	-0.01
Prev	5.51	1.37	1.55
Sta	0.90	0.98	3.16
Intercept	-0.00	0.00	0.00

Table 5.30: Linear regression coefficients at end of evaluation for each source of information

Nevertheless, the data suggests that some of the sources of information are vastly more indicative than others. Most disappointingly, the use of coverage information has almost no effect on the overall performance that isn't already explained by other variables. This is not surprising for Ant, where the difference in coverage levels had been minimal, but is more surprising for JMeter and JodaTime. Conversely, the value given by the number of previous bugs in a method is given quite a high weight. While, as mentioned, this does not make it the most important source of information, it does somewhat suggest that, simply put, buggy methods are likely to have more bugs in them, which is in line with other findings on the subject.

One interesting thing to note is that for JodaTime, the project for which the combined approach has least benefit, the values for STA and SOURCE seem quite different than from the other projects. This may account for why the combined approach performs worst here; despite the source code approach being the best-performing source of information, it is given comparatively low-weight. From examining the results in JodaTime, it seems that for many early bugs, the source code approach does not perform particularly well, and it is only later when most of its 'successes' occur. This may have led to the regression considering it unreliable, meaning that on the later bugs where it was actually predicting correctly it was not given much weight. Quite how to overcome this sort of problem is still an open question; it may be that the performance of the combined approach simply improves with more data, and that it may not be suitable for projects in their early stages (JodaTime has much less information available than the other projects). It may also be possible to give each of the sources of information an initial weight, based on heuristics and information from other projects, that is then updated as data is produced for a project.

5.6.2 Class-level analysis

While the combined approach is an improvement over existing techniques, in general the absolute performance may still be considered relatively disappointing. However, the fact that a method was not actually changed as part of fixing a bug does not necessarily mean that that method is not useful to a developer in localising the bug. For example, the method may be located in the same class as one that needs to be changed to fix a bug.

In Ant Bug 43342, for example, the only relevant method is in position 13777. However, the first method returned in the list, which itself is not directly relevant to the bug, is in the same class as that method. If a developer were to explore the context surrounding a method, they may then realise that this other method is relevant to the bug.

To investigate, another measure FRM_{CL} (and a corresponding MRR_{CL}) can be used. This is the position of the first method which is in the same class as a method changed to fix a bug. Obviously, for any given bug, this value can never be greater than the FRM. Table 5.31 shows the results of measuring whether or not each method is in the same class as a relevant method. For Ant, 49% of bugs now have a relevant class returned in the top-10 positions, and even for the other two projects, the results are fairly good, at 32% and 44%. This suggests that it may be viable to actually use these techniques in a real setting.

	Ant	JMeter	JodaTime
Cov	5 (1%)	0 (0%)	0 (0%)
Source	172 (17%)	19 (12%)	18 (31%)
Bug	50 (5%)	4 (2%)	3 (5%)
Prev	14 (1%)	2 (1%)	5 (8%)
Sta	51 (5%)	5 (3%)	4 (7%)
Comb	211 (21%)	23 (14%)	15 (25%)

	Ant	JMeter	JodaTime
Cov	6 (1%)	0 (0%)	3 (5%)
Source	431 (43%)	50 (30%)	28 (47%)
Bug	169 (17%)	10 (6%)	11 (19%)
Prev	111 (11%)	7 (4%)	14 (24%)
Sta	66 (7%)	6 (4%)	7 (12%)
Comb	485 (49%)	52 (32%)	26 (44%)

Table 5.31: Number of bugs for which the FRM is the top result (left), and in the top-10 results (right)

As the system is currently implemented, this change has to be performed as a post-processing step during the analysis. It is is possible however that this could lead to some classes being missed. For example, a class which consisted mainly of getters and setters may have no single method which is highly similar to a bug report, but taken as a whole the class may be highly similar. Such a class would not be returned by the current system. However, better results could be obtained by changing the actual initial stages to return information at a class-level: doing so is left as future work.

Table 5.32 shows the MRR_{CL} for the approaches when considering only if the method is in the same class as a relevant method. It is interesting to note that while the level of improvement of the combined approach over the source code approach for Ant and JMeter is similar to the difference between the approaches at method level, for JodaTime the performance of the combined approach has now decreased considerably from the source code approach. Further investigation would be needed to determine why this is the case.

	Ant	JMeter	JodaTime
Source	0.26	0.18	0.37
Comb	0.31	0.20	0.30

Table 5.32: MRR_{CL} for Source and Comb on each project

5.6.3 Changes over time

Figure 5.33 shows for each project the cumulative MRR over time. This is the MRR of all bugs up to and including that bug. This indicates how the performance of the approaches changes over time. It is interesting to note from this figure that in JodaTime the combined approach had actually outperformed the source code approach for most of the time, the gap only narrowing towards the end. This could be explained by the poor performance of the source code approach on early bugs as already explained earlier in this section.

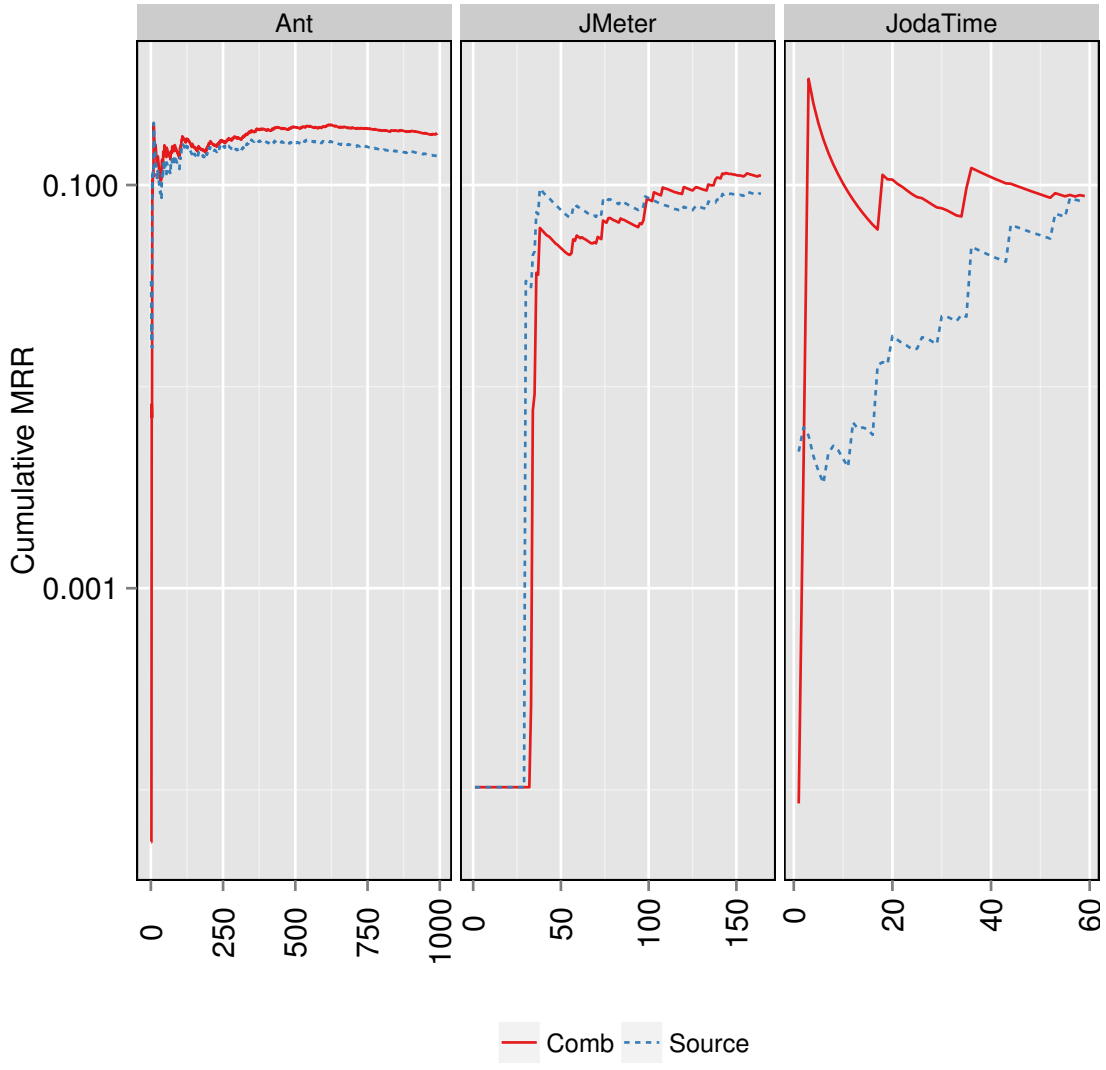


Figure 5.33: Cumulative MRR for Comb and Source. Bugs ordered by report date.

One other interesting fact highlighted by Figure 5.33 is that the combination technique appears to require a certain amount of data before the performance 'settles down'. In Ant, the initial difference between COMB and SOURCE is rather volatile, and for JMeter it is only after around 100 bugs or so that COMB starts to out-perform SOURCE. If more data had been available for JodaTime it is also possible that the performance of the technique could have been better.

The fact that these charts 'level out' suggests that it may not be necessary to use the entire history when training the techniques. It is also possible that the behaviour of systems will change over time. This would be particularly true of coverage information for example. As the level of coverage changes over time, it may be become a more, or less, reliable indicator of whether a method is buggy.

Table 5.34 shows the performance of the combined approach if only the most recent data is used to train the approach. Note that, similarly to when looking at class-level information, this is currently done as a post-processing step. The training works by discarding data from more than a certain number of events ago, but an event can be either a bug report or a bug fix, and does not necessarily correspond to a particular revision of code or unit of time. The total number of events for each project is: Ant (3038); JMeter (586); JodaTime (152). Because these do not relate to a fixed number of bugs, it is hard to draw exact conclusions about how much past information is required, but nevertheless, the results suggest that training on only some of the past events may provide equal performance but require less data and therefore time to produce.

	Ant	JMeter	JodaTime
All	0.18	0.11	0.09
10	0.13	0.06	0.07
50	0.15	0.09	0.08
100	0.16	0.1	0.09
250	0.17	0.11	-
500	0.18	0.11	-
1000	0.18	-	-
1500	0.18	-	-

Table 5.34: MRRs for COMB when considering only the last *n* events on each project

Another point to note is that this alteration only investigated changing the number of events examined during the final combination of the sources of information. This should account for any changes in whether particular sources of information become more or less trustworthy over time. However the components using the count of previous bugs and information from previous bug reports still take into consideration the entire history of the project, not just those events specified. It may be that better results are achieved by limiting the number of bugs considered when training these classifiers also.

5.6.4 Unbalanced classes

As discussed before, using all of the information about which methods are *not* related to a particular bug prohibitively increases the time taken for the evaluation. Furthermore, it has been shown that having such highly unbalanced classes can have a detrimental effect on the

performance of some techniques. To counteract this, the commonly used technique of undersampling the majority class was employed. To test the effect of the negative undersampling, the linear regression was repeated using varying chances of including negative examples. These results are shown in Table 5.35. As shown, 1% is a reasonably appropriate threshold to use, giving the best results on two of the projects. However, for Ant, the best results are actually found using a higher threshold. It appears that the best value may be project-specific. Future work could also determine if there was a better mechanism for reducing the amount of examples used in training.

	Ant	JMeter	JodaTime
0.1%	0.166	0.098	0.065
1%	0.180	0.111	0.088
2%	0.186	0.105	0.082
5%	0.188	0.105	0.083
10%	0.186	0.105	0.079

Table 5.35: MRRs for Comb with varying chances of accepting negative examples

An alternative approach, using the exact same number of positive and negative examples was also considered. However, this gave poorer performance than the approaches above: MRRs of 0.158 for Ant, 0.107 for JMeter and 0.042 for JodaTime.

5.6.5 Different learning techniques

As discussed, a linear regression is likely to not be the best way to combine this information. Some other mechanism of combining the sources of information may be more suitable. To investigate this possibility, the results from the individual components were also combined using a number of other different techniques.

- Random forest
- Naïve Bayes
- Genetic programming
- Borda count
- Simple combination

This is not an exclusive list of possible techniques, but it covers a wide range of techniques. To simplify comparisons, other techniques which are similar to those already mentioned, such as decision trees, ridge regression etc. , were excluded at this time. Similarly, using Support Vector Machines (SVMs) was also not evaluated, since preliminary attempts suggested the time taken would be prohibitive, and the models produced are often hard to interpret.

This section will describe each of these different techniques in turn. It will then compare them to each other, and to the original linear regression technique.

5.6.5.1 Random forest

Random forests [Bre01] are collections of decision trees. Each decision tree consists of a number of nodes, each of which is based on one particular source of information for a bug e.g. is the

similarity between the bug report and source code greater than 0.6? In a random forest, each decision tree is trained against a random subset of the past data, and each node created by choosing from a random subset of features. Whenever a new bug is reported, it is evaluated against *all* the trees in the forest, and the result averaged. Using random forests as opposed to single decision trees has been shown to be more robust to noise in the data, and is therefore quite appropriate for this task.

While there are a number of parameters that can be configured when training random forests, the most important of these is the number of trees in the forest. Other parameters have been left at the suggested defaults given by scikit-learn for the appropriate type of data. Table 5.36 shows the MRRs for forests of varying sizes. As shown, the performance increases as the number of trees increases, up to a point. This is to be expected, as beyond a certain point adding new trees will not improve the performance. Adding more trees also increases the time taken to perform evaluations, and indeed the evaluation of forests with 10000 trees on Ant (the largest project) could not be completed due to the prohibitive time.

	Ant	JMeter	JodaTime
1	0.01	0.01	0.003
10	0.057	0.03	0.002
100	0.078	0.046	0.019
1000	0.085	0.056	0.019
10000	-	0.054	0.019

Table 5.36: MRRs for random forests of varying size

Unfortunately, even the best results here, for forests of 1000 trees, are not as good as the original linear regression approach. In fact, the results are poorer than that given by Source alone, without utilising the other sources of information. To investigate this, we can examine the forests produced. By measuring the proportion of inputs that pass through each node in the tree, it is possible to estimate how important each feature is to the final output. Table 5.37 shows these values for the final bug in each project. As expected, the most important source of information is Source. For Ant and JMeter this feature dominates the final result, with other features in effect only serving to introduce noise that makes it perform worse than Source alone. This is in contrast to the weights given to the linear regression (in Table 5.30, repeated below) where there was more balance between features, which may explain the poorer performance of the random forest technique.

	Ant	JMeter	JodaTime
Cov	0.00	0.01	0.06
Source	0.90	0.94	0.51
Bug	0.03	0.01	0.03
Prev	0.03	0.03	0.32
Sta	0.04	0.01	0.08

Table 5.37: Estimated importance of each feature in final random forest

	Ant	JMeter	JodaTime
Cov	-0.01	0.01	-0.01
Source	1.29	1.81	0.74
Bug	0.42	0.13	-0.01
Prev	5.51	1.37	1.55
Sta	0.90	0.98	3.16
Intercept	-0.00	0.00	0.00

Table 5.30 (Repeated): Linear regression coefficients at end of evaluation for each source of information

5.6.5.2 Naïve Bayes

Naïve Bayes is a machine learning algorithm for classification where the overall output is based on multiplying individual probabilities determined by the values for each feature. This is based on the naïve assumption that the features are conditionally independent of each other. Despite this assumption rarely being true in practice, classifiers created using this algorithm have been shown to have good performance in multiple domains [Zha04]. While there are some variations on the algorithm, differing in their assumption of probability distributions, for this work the most appropriate approach assumed that the probability distribution of each feature was Gaussian, with parameters estimated based on the training data.

Similarly to random forest, the Naïve Bayes approaches fail to out-perform the linear regression approach, or even SOURCE. For completeness, results are included in Table 5.41 at the end of this section. Naïve Bayes models are also not straight-forward to interpret, and so no further detail will be provided here.

5.6.5.3 Genetic programming

Genetic programming [Koz92] is one of a number of evolutionary techniques inspired by nature, in which a number of programs are evolved to solve a problem. A pool of, initially random, programs is created and each is evaluated against a set of test data to determine how fit it is. New programs are then created by mutation and crossover of the programs, with the most fit programs being more likely to be chosen to pass their genes on to the next generation. This process is repeated for a number of generations, and the most fit individual of the final generation selected as the overall solution. Unlike the linear regression shown earlier, this technique can evolve solutions which express non-linear combinations of the features to produce the output.

In order to assess the performance of genetic programming in combining the different sources of information, a prototype implementation was created using the Pyevolve [Pye] framework. While there are many parameters that can be configured for a genetic programming algorithm, this was a preliminary investigation and so most parameters have been set to default values recommended by the framework. The most important factors to consider are the operations and inputs that are available, and the function which is used to evaluate the fitness of an individual program. In this case, programs were composed of the individual feature values, the basic mathematical operators: $+$, $-$, $\times$, $\div$, and small integer constants. As with the other techniques, the output of the program was treated as a probability that a particular bug is

related to a particular method. The fitness function used was the mean absolute error between the predicted value and the expected output: either 1 if the bug was related to the method, or 0 otherwise.

Again, the performance of the genetic programming approach was poor. Table 5.38 helps to highlight the reasons for this, by showing the best program generated for the final bug in each project. Note that the values output by these programs are not used directly as percentages, but instead normalised. These programs are not straight-forward to interpret, but after simplification the program for Ant can be seen to weight SOURCE as the most important feature, as we would expect, followed by BUG. However, both STA and COV have negative impacts on the final result. For JMeter, the generated program is in fact invalid, as it involves dividing by zero. In these circumstances, no methods at all would be predicted as relevant to the given bug. For JodaTime, the program weights BUG and PREV favourably, and STA negatively, with no mention of SOURCE. What these programs indicate is the difficulty involved in configuring genetic programming.

	Program
Ant	((((Bug − 1) ÷ Sta) − (Cov + (Prev × 1))) + ((Sta × 1) − ((Sta − Prev) − (Source + Source))))
JMeter	(((Cov ÷ (Bug − Bug)) × Sta) + ((Sta + (Sta − Bug)) ÷ Sta))
JodaTime	((Prev × 1) + (((Bug − 1) ÷ Sta) − (Bug × (1 − Prev))))

Table 5.38: Final programs generated at end of evaluation

While the performance is poor (final results are shown later in Table 5.41), there is still considerable scope for improvement. This technique is entirely capable of expressing the same relationship as that returned by the linear regression, along with much more complex relationships. However, perhaps even more so than other approaches, this approach is very sensitive to certain settings. In particular, the fitness functions used and the range of operators available can have a major impact on the performance. As such, future research would be needed to fully evaluate whether or not it could still be an appropriate method for combining the various sources of information.

5.6.5.4 Borda count

Borda count [dB81] is a technique originally devised for electoral contests. However, it has also previously been applied for any situation in which the output of multiple components has to be combined, and in particular for combining information for bug localisation [TNBH13]. In it, each voter, in this case the different sources of information, gives a ranked list of the candidates, in this case the methods of the project. The Borda count method gives each candidate a score for each voter based on the number of candidates e.g. in a 5-candidate election, the candidate ranked first receives 5 points, second receives 4 points, and so on. The points for each candidate from all the voters are summed, and the candidate with the highest total is the winner.

For the purposes of bug localisation, the list shown to the developer would be in descending order of total number of points. This is a simple method: unlike the other methods presented here it does not evolve over time, and it ignores the actual values presented by each source of

information, considering only the rank. Nevertheless, it has been shown to be effective for the purposes of bug localisation before, and so is presented here for completeness.

Table 5.39 shows the MRRs achieved on each project by the Borda count method. There are some variations that have been applied to the technique. In the modified version, which was used in the previous work [TNBH13], the number of points given to the first-ranked candidate from each voter is equal to the number of candidates for which that voter gave a score, rather than the total number of candidates. This can however result in voters which do not score all the candidates being somewhat 'penalised', as their votes count for less e.g. stack trace information, for which only very few methods will get given a positive score. Table 5.39 therefore also shows the results of using the original method, in which the number of points awarded is based on the total number of candidates. As expected, for the sources of information used here, this is more effective, but still less so than linear regression.

	Ant	JMeter	JodaTime
Borda count	0.055	0.032	0.066
Borda count (modified)	0.041	0.011	0.024

Table 5.39: MRRs for original and modified Borda count methods

5.6.5.5 Simple combination

Another, somewhat naïve approach is simply to merge the results given by each source of information. The first five methods it returns will be whichever methods are given in first place for each of the basic sources of information, in an arbitrary order. This will be followed by all the methods in second place, then third etc., with duplicate methods only being returned once.

The order used to choose between the different sources of information can obviously have some effect on the overall performance. For this evaluations, three different orders were evaluated, each customised for a specific project. This is done by ranking the sources of information according to how well they performed individually, as was shown in Table 5.24[3](repeated below). The results are shown in Table 5.40. What is interesting is that, for Ant, the version customised to that project was not in fact the best order. In fact, the order based on JMeter (Source, Sta, Bug, Prev, Cov) gave the best, or equal-best, performance on all three projects. However, the difference is slight: while it it possible to tune this approach to a particular project, reasonable performance can be attained without doing so. One other interesting thing to note is that, despite its naïvety, the simple combination method is actually more effective than any of the alternative techniques presented in this section, and on JodaTime even out-performs linear regression.

[3]Obviously, on a real project, this information would not be known at the start of using a bug localisation tool

	Ant	JMeter	JodaTime
Cov	0.001	0.001	0.001
Source	0.139	0.091	0.082
Bug	0.044	0.021	0.035
Prev	0.024	0.008	0.058
Sta	0.038	0.026	0.044

Table 5.24 (Repeated): MRRs for each source of information

	Ant	JMeter	JodaTime
Simple combination (JMeter)	0.142	0.086	0.110
Simple combination (Ant)	0.137	0.082	0.104
Simple combination (JodaTime)	0.134	0.081	0.110

Table 5.40: MRRs for each simple combination

5.6.5.6 Conclusions

The results given by each of the alternatives proposed here are shown in Table 5.41, along with `Comb`, the original linear regression. Contrary to initial expectations, none of them outperform the original approach, except the simple combination approach on JodaTime. This is quite surprising, given the nature of the data. However, some of the approaches given here are sensitive to the tuning of the parameters they require. This is particularly true of genetic programming. Further research would be needed to ascertain whether or not they could better the linear regression approach.

	Ant	JMeter	JodaTime
Comb	0.180	0.111	0.088
Random Forest (1000 trees)	0.085	0.056	0.019
Naïve Bayes (Gaussian)	0.079	0.029	0.076
Borda count	0.055	0.032	0.066
Genetic Programming	0.035	0.035	0.038
Simple combination (JMeter)	0.142	0.086	0.110

Table 5.41: MRRs for different ways of combining information

5.7 Threats to validity

The threats to validity in this study are largely the same as that in the preliminary study, listed in Section 5.2.4, but are repeated in summary form here.

- Links between bugs and relevant methods may not be accurate, but the technique used has been used in other studies and shown to be mostly accurate. Note that unlike the preliminary study, the links used in evaluation were not validated manually – this was to increase the amount of bugs analysed.
- FRM may not be the most accurate measure of how developers would use localisation techniques, but is widely used in other studies and allows for comparison between techniques.

- Results may not generalised to other projects. Replication would be needed on other systems, developed in other languages.
- There may be errors in the implementation, but the code has been released, and is available to scrutinise.

5.8 Conclusions

This chapter makes several contributions of note in relation to the problem of bug localisation: firstly it rebuilds and analyses the history of seven open-source projects – ArgoUML, JabRef, jEdit, muCommander, Ant, JMeter and JodaTime – and establishes the viability of combining information from multiple sources as a method to improve the performance of bug localisation techniques. It also evaluates individually the contribution of some of these sources of information. The test history of three of the projects is recreated, giving valuable insights into testing trends in open-source projects, and suggesting that test coverage may be a possible source of information for bug localisation. In addition, the tools and results required for replication have been made available.

The findings in relation to test coverage demonstrate a marked inconsistency which makes it impossible to state with any certainty that bugs are more or less likely to be found in methods which are untested. However, within individual projects there are indeed differences: for JMeter the coverage level amongst methods related to bugs is indeed lower than those unrelated to the bug, by 10.1%, but the opposite is true for JodaTime where the coverage of methods related to bugs is 7.1% higher. For Ant there was no discernible difference. This suggests that is indeed possible to use test coverage information as a source of bug localisation information for some projects, if used in conjunction with other sources of information.

For the sources of information considered, the approach which performs a language-based comparison of the text of the bug description with the vocabulary of each method in the system produced the best results – placing a method relevant to a bug fix in the top ten results for 24% of all bugs for Ant. Combining the various sources together raises this figure to 30% for Ant – encouraging results but probably not accurate enough to be immediately useful, and the improvement is much smaller for other projects. This chapter has also shown however that developing techniques which seek to identify relevant classes may also be worthwhile – for Ant, the combined approach found the relevant class for nearly half of all bugs. Future development could see the approach improved further, to really help reduce the time and money spent on bug localisation. Some of these potential improvements are outlined in Chapter 6.

6 Conclusions and Future work

This thesis has examined the life cycle of software bugs, covering each of the three most pertinent points in any bug's life: when it was introduced, when it was reported and when it was fixed. The studies presented increase the knowledge available about the life cycle of bugs. This chapter summarises the main findings of the thesis and presents a number of avenues for future work.

6.1 Contributions

This thesis has made a number of concrete contributions covering the three main points in the bug life cycle.

6.1.1 Bug origins

Two existing approaches for recovering the origin of bugs were compared and evaluated on 166 bugs from 2 open-source projects. The main findings were:

- Overall, the approaches were somewhat effective, but generated many false positives and false negatives
- The dependence approach was slightly more effective at identifying the correct origin, with F_1-Scores of 0.42 and 0.72 on the two projects
- Most effective however was to combine both techniques, rather than use them in isolation, raising the F_1-Scores to 0.46 and 0.78
- Slightly more than half of bug fixes involved changing only a single file, while for larger bugs, the origins are often difficult to find; the performance of the approaches could be improved by ignoring commits which changed a large number of files
- The accuracy of the techniques are worsened by the length of time between the origin and fix
- Developers often made unrelated changes at the same time as fixing bugs, and this was particularly detrimental to the performance of the two approaches
- The manually verified origin information for the bugs has been made available to other researchers and can be used to verify the performance of existing or future approaches

6.1.2 Bug reporting

A sample of 1600 bugs from 4 projects was obtained, and an analysis carried out on the quality and quantity of information provided by bug reporters in these bug reports. Overall, the analysis found:

- The majority of bug reports contained three or fewer of the ten pieces of information that developers consider most important
- Less than half of the bugs contained *Steps to reproduce*, while less than a quarter contained each of *Stack traces*, *Screenshots*, *Test cases* or *Code examples*
- Even when reporters do provide *Observed behaviour*, *Expected behaviour* or *Steps to reproduce*, they very rarely identify them as such, making extraction of the information by automated means challenging

6.1.3 Bug localisation

Finally, two studies were carried out on a number of ways to enhance existing techniques for automated bug localisation. These studies analysed 1586 bugs from 7 projects, and found:

- The identification of several novel sources of information for bug localisation which had not previously been exploited: methods involved in similar bug fixes, test coverage information, the number of previous bugs in a method and stack traces within a bug report
- Three of these sources of information – previous similar bugs, 'buggy' methods and stack traces – were shown to be somewhat useful for improving the performance of bug localisation
- Test coverage was not shown to be useful for bug localisation – for some projects, the level of test coverage in methods which are related to a bug is higher than methods which are not, while for other projects the opposite is true, but despite this difference remaining consistent over time within a single project, it is hard to exploit this for bug localisation
- Nevertheless, the addition of new information was found to improve the performance of bug localisation over existing IR-based techniques: in the final system, the number of bugs where the FRM is in the top position increases from 98 to 129, and in the top-10 from 272 to 336

6.1.4 Overall

The data and source code for all the projects has been made available for other researchers to replicate and build upon, and can be found at `http://personal.cis.strath.ac.uk/s.davies/` or in the supplemental material DVD in Appendix A.

Hopefully, through these techniques and the other insights in this work, the cost and time spent on bugs can be reduced, freeing users to concentrate on the work they need the software to achieve, and allowing developers more time to work on providing new features and driving

software forward. However, there is still much that is not known about bugs, and important work to be done to put this knowledge into practice. The remainder of chapter will detail some of the avenues of possible future work that should be undertaken in order to utilise this knowledge to reduce the time and money spent on bugs.

6.2 Applicability and replication

All of the studies in this thesis have been carried out on real-world bugs drawn from real-world projects. In most cases, they have been on datasets much larger than comparable studies, if comparable studies have existed at all. Nevertheless, given the vast numbers of bugs out there, the studies have only been able to cover a small subset of bugs. It is possible that the bugs studied are not representative even of other bugs within the same project. Furthermore, the projects studied were almost all open-source, and mostly written in Java. They may not be representative of all projects, and commercial projects especially may have significantly different characteristics. The studies have been carried out with either manual simulations of existing techniques, or re-implementations, rather than tools that already exist, and errors may have been introduced through this. Whilst efforts have been taken throughout the thesis to account for these problems, the only way to gain more confidence in the results is through replication on other projects, particularly industrial projects.

6.3 Availability of tools

Throughout the development of this thesis, one of the most common frustrations was with research which described or evaluated tools without making such tools available. Related to the point in the previous section, this has the effect of increasing how difficult it is to replicate work, thereby reducing the amount of replication performed. In an attempt to alleviate this, all the pieces of software created during this thesis (along with the data and papers) have been made publicly available. Unfortunately however, these are still research prototypes. Further work would be needed to reduce the work required for other researchers to replicate this research. As well as other researchers, this would potentially also allow tools to be used by developers, in order to actually reduce their bug-fixing burden.

For reporting on bug origins, the number of papers using such techniques shows there is a demand for such software. However, as there are no publicly available implementations, it is hard to compare research results, and large periods of time must be spent in reimplementing. Furthermore, this information could also be useful for developers and managers, but they cannot yet easily obtain it.

The bug localisation techniques developed in Chapter 5, and indeed most other similar techniques, have only been evaluated by comparing them to historical evidence of bug localisation. This evidence is likely to be both incomplete and partially inaccurate. The evaluation also is based on the assumption that developers will examine the list of returned methods in order, but this may not be true. In reality, they are likely to combine the list of methods with their own knowledge and intuition. It is important therefore that these bug localisation techniques are evaluated with actual developers.

To overcome developers' reluctance to utilise new, untested tools, it would be appropriate to integrate these techniques into existing tools. To do so, I propose augmenting BTSs to display a list of potential bug locations alongside the bug report. This could be done either by updating the systems, or through the use of a proxy or similar system. Similar functionality could also be added into IDEs which have integrated bug tracking functions.

Integration with a BTS would assist developers, but the system also needs to be updated whenever new bugs are fixed, in order to learn from the new information. To handle this, the second part of any implementation would be integration into continuous integration (CI) servers, as these are commonly used in many development processes. This would be relatively straightforward, as many CI systems allow the use of plugins which can be executed whenever new commits are made to a project.

The use of both these systems would not only allow for controlled evaluations to be carried out, but would also hopefully allow the software to be released freely, allowing developers to test it out 'in the wild', increasing the amount of evaluation data and feedback. Throughout development, this eventual aim of implementation has been kept in mind, and the system developed in such a way that integration with BTS and CI systems should be straightforward. Additionally, through the use of a straight-forward plug-in structure, new or altered algorithms and tools can be added easily, allowing further experimentation to be carried out.

6.4 Exploiting user knowledge

As noted throughout this thesis and in other studies [AV09, BBA+09], the information contained in electronic repositories is not always accurate. Whilst automated methods to determine bug origins and bug locations are desirable, they will not always be highly accurate, and depending on the context of use it may be possible to incorporate users in the process to improve the accuracy.

Chapter 3 detailed multiple proposed methods for removing false positives when determining bug origins. However, these may not be highly accurate and in some cases they may not be necessary. Depending on the desired use of the approaches, it may be that users could be available to manually remove false positives and to assist the approaches in other ways.

If the approaches are being used to assist a developer with finding the origin of a specific bug then a reasonable first step would be to present the developer with a list of all changes in the fix. From there, the developer could select any changes which are obviously not related to the bug. This could easily reduce the number of false positives quite significantly as a number of irrelevant changes could be ignored. This process could also be carried out throughout the search by presenting the user with a list of proposed origins. Again, the user could select some as being irrelevant and the approaches could then follow the trail back through older versions. As well as potentially being more useful for a developer, such a technique could also lead to a better understanding of what type of changes are responsible for causing or fixing bugs. This information could then be incorporated into the approaches to improve them, learning from the developer's feedback.

A similar approach could be taken for bug localisation. Frankly, it seems unlikely that all of the information sources used by developers to localise bugs can be replaced by automated means. In bug localisation, like with many other software tools, the best results are probably more likely to be found in the combination of knowledge from the developer and the computer. For example, it could be more effective to present a developer with a list of possible bug locations, but to allow them to refine the query used, mark certain words in the bug report as being more important or as irrelevant, or to incorporate the idea of relevance feedback in some way.

6.5 Bug origins

Chapter 3 already detailed a number of possible improvements that could be made to the approaches for determining the origins of bugs. In summary, these were:

Combining information: As results showed, there were a number of times when one approach would return a results while the other did not. Combining the results from the two approaches gave a better result, but it should be possible to expand on this, by combining information in a more nuanced manner. This could take the form of using probabilities given by each approach and combining these, in a similar manner to that done for bug localisation in Chapter 5.

Inter-procedural analysis: A number of bugs were not caused by changes made within the method in which the fix was made. Expanding the dependence analysis to include inter-procedural changes would be necessary to find the origins of these bugs.

Limiting the results returned: The text approach in particular would often return many incorrect results. Limiting it to only return a single result may result in improved performance. However, the best technique to limit the results is still to be determined.

Ignoring unrelated changes: Both approaches, in common with many other techniques, were confused when developers made unrelated changes. Incorporating some way to identify and ignore these unrelated changes could improve the performance of the approaches.

All of these possible improvements could have a significant effect on the performance of the two approaches. However, they would most easily be applied to a working copy of a tool, rather than by amending the manual process.

6.6 Bug reporting

Chapter 4 showed that there are many pieces of information that developers want in bug reports that are often not provided by users. While the main intention of this study was to provide information to help in bug localisation, the results discovered could also be used to improve BTSs. For example, if an automated system could point out to users that they haven't provided particular bits of information, along with information about how this information makes it more likely their bug will be fixed, the quality of bug reports could be improved for everyone. This is in many ways similar to the Cuezilla [BJS+08] tool presented by Bettenburg et al., although that does not appear to have been released for developers to use.

6.7 Bug localisation

The studies in Chapter 5 have yielded a number of insights into the diverse life cycles of bugs. However, for the information to be of any practical benefit to developers, there is still a large amount of future work that could be undertaken. In particular, although the techniques have been shown to improve the results given by IR-based bug localisation techniques, in absolute terms there is still a long way to go before these techniques can be truly useful for developers. For many bugs, the first method shown to developers which is actually relevant to the bug can lie well outwith the range of bugs a developer would examine. This has obvious implications on the adoption of such tools; if developers feel that the results returned by the tools are inaccurate, or even that they slow down the process of bug localisation, they will be reluctant to continue using the tools, even if for some cases they work accurately. There are therefore still a number of future areas of work.

6.7.1 Other sources of information

As mentioned previously, the implementation of the bug localisation tool allows new evaluators to be added relatively straightforwardly. Indeed, there are still many potential sources of information about bugs and methods which could be integrated into the process.

6.7.1.1 Commits which introduced other bugs

As Chapter 3 showed, a lot of useful information can be gained about bugs from identifying their origins. Automated analysis of these origins within a project could then be used to give better bug localisation results. As a simple example, commits which introduced bugs may be more likely to also introduce other bugs, particularly in the context of large-scale refactorings and new features, and therefore it may be worth ranking methods which were affected by such commits more highly. Before any such relationship could be examined however, first a technique for identifying bug origins would have to be implemented, as no publicly accessible ones exist. Secondly, to improve their accuracy, some of the potential improvements outlined in Chapter 3 could be utilised in the implementation. Finally, this tool would have to be integrated into the bug localisation tool.

6.7.1.2 Relationship between when a bug was reported and when it was introduced

Surprisingly little research has been done on determining when bugs are introduced. Kim and Whitehead Jr. [KW06] showed the median time between bugs being introduced and bugs being fixed in two projects was around 200 days, while Chou et al. [CYC+01] examined errors in versions of Linux, finding the median difference between the origin and fix of a bug to be around 1.25 years. In Chapter 3, it was determined that around 12% of bugs were fixed one revision after being introduced, with 50% of bugs being fixed within 12 revisions. As such, and in line with intuition, it seems likely that when a new bug is reported, there is a higher chance of it being located in code which has been recently changed than code which has not been changed recently. If there was a relationship between when bugs were reported and when

they were fixed, this information could be integrated into the bug localisation tool, raising the rankings of methods which have been changed recently before the bug.

6.7.1.3 Relationship between error logs and bug location

Error logs are similar to stack traces in some ways, in that they often contain information about what code was being executed when bugs occurred. No research, however, appears to have been carried out on how effective error logs are at assisting developers in locating bugs. Yuan et al. [YMX+10] did develop a system which combined error logs with static analysis of source code to identify which paths through the code must have been taken to produce the log output. Such information could also potentially prove useful in bug localisation, although like stack traces, the information is not always particularly commonly provided in bug reports.

6.7.1.4 Relationship between bug reports and other indicators of code quality

There has been much research into various metrics as indicators of where bugs are likely to be found [ZPZ07]. Additionally, there are numerous static and dynamic analysis tools intended to locate particular classes of bugs, and to measure the coverage of test suites [YLW06, APM+07]. Intuitively, it would seem that the cause of a bug report is more likely to be found in regions of code:

- That contain abnormal values for various metrics e.g. complexity
- That have not been fully tested
- That contain warnings of possible errors

However, there have been no or few studies on links between any of these and bug reports. If links were to exist it could help focus searches on more likely bug locations.

6.7.2 Other improvements

There is also work that could be done to improve the individual sources of information. In JMeter Bug 53310 for example, none of the sources of information rank any of the relevant methods highly. The bug report for this bug is incredibly simple, just one short sentence. However, that sentence explicitly contains the name of a class in the system, methods of which had to be changed to fix the bug. In particular the textual approach should have been expected to perform well with this bug, but because it searches through the *content* of a method, and does not take the name of the class into account, it performs poorly. On one hand, automated bug localisation for this sort of bug seems pointless; the developer has already been told where the bug lies. If however a developer examined the list of methods presented as possible locations and did *not* see any of the methods in that class, they would lose confidence in the ability of the bug localisation technique to cope with other bugs. Any attempt to introduce automated bug localisation techniques into practice would have to take such situations into account in order to have the support of developers.

6.7.3 Structured classifiers

All evaluation has been performed at the method level of granularity. However, the techniques are also applicable at class-, package- or even line-level, and one possible improvement would be to build a hierarchy of classifiers, so that an incoming bug is first analysed to determine which sub-project it belongs to, then which package within that, then sub-package etc. before finally examining the file. This could both speed up the process, by restricting the number of classifiers that had to be tested for each bug, and also be a useful indicator to the developer of more general possible locations.

6.7.4 Identifying key words

It has been shown in other work that the query used can have a major effect on the performance of any search [LKE10]. Very often there are a large number of irrelevant words in a bug report, and some highly important words, e.g. the terms "perl" and "syntax" in the bugs presented as motivating examples in Section 5.2. There may be ways to identify such key terms, perhaps by relying on the bug summary. This often acts as a very succinct description, and could be used instead of the entire description, or the terms contained within it weighted more highly. Alternatively, techniques such as LSI and LDA could be used here, as these are intended to more accurately draw out the key information of a document.

6.7.5 Evaluating how users use tools

Almost all research on the area of bug localisation evaluates tools assuming that developers will look at the results presented to them in a particular way: starting from the top, stopping at the first relevant result, and then using other techniques to find the remainder of related methods. This allows different techniques to be compared relatively straight-forwardly. However, it may not match how developers would actually use such tools if they came commonly into use [BKEL11]. Developers might for example view the list in some other order, or may have to examine multiple methods before discovering that ones they viewed at first were actually relevant. Once a tool has been properly developed, studies of actual use would be crucial to determine how best to steer research in future.

One such possibility to improve actual use by developers would be for systems to display confidence values along with each result. For example, on some bugs there may be almost no information available to the bug localisation system, or it may simply be unable to confidently predict any particular methods, in which case such a fact could be conveyed to the user. This would result in the developer not being required to spend time investigating results which are not likely to be correct. This would help to alleviate one of the problems presented earlier: developers deciding that bug localisation systems are unreliable on the basis of some initial tests and so choosing not to use them.

6.8 Bug life cycles

All of the work in this thesis has concentrated on three important points in the bug life cycle. These are not however, the only important points. For example, there may be failed attempts to fix bugs, or there may be points where the bug changes, if the code involved changes. Understanding these could give an even better understanding into bugs, and help to reduce their associated costs.

Furthermore, understanding the relationship between events in the life cycle of a bug could be of greater help. For example, how long is there between a bug being introduced and it being discovered? How does this affect its chances of being found and fixed correctly or quickly? Answers to these questions, and others like them, would be of great use to both developers and researchers.

6.9 Closing remarks

As this thesis has shown, the life cycle of bugs is complex. Understanding this life cycle, understanding the important points in it, and understanding the relationship between them is crucial to help prevent and remove them. Through the studies presented here, I have aimed to increase this understanding, and to help reduce the burden of bugs and maintenance on developers, freeing them up to concentrate on the new features that actually drive software forward.

A THESIS DATA

All data and source code for the thesis can be found at `http://personal.cis.strath.ac.uk/s.davies/` or in the supplemental material DVD.

BIBLIOGRAPHY

[AAD+08] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc, *Is it a Bug or an Enhancement? A Text-based Approach to Classify Change Requests*, Proceedings of the Centre for Advanced Studies Conference, 2008.

[ACD07] Lerina Aversano, Luigi Cerulo, and Concettina Del Grosso, *Learning from bug-introducing changes to prevent fault prone code*, Proceedings of the International Workshop on Principles of Software Evolution, 2007.

[ADTP10] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia, *Practical fault localization for dynamic web applications*, Proceedings of the International Conference on Software Engineering, 2010.

[AHM05] John Anvik, Lyndon Hiew, and Gail C. Murphy, *Coping with an open bug repository*, Proceedings of the OOPSLA workshop on Eclipse technology eXchange, October 2005.

[AHM06] _____, *Who should fix this bug?*, Proceedings of the International Conference on Software Engineering, 2006.

[AJL+09] B. Ashok, Joseph Joy, Hongkang Liang, Sriram K. Rajamani, Gopal Srinivasa, and Vipindeep Vangala, *DebugAdvisor : A Recommender System for Debugging*, Proceedings of the European Software Engineering Conference/Symposium on the Foundations of Software Engineering, 2009.

[AM11] John Anvik and Gail C. Murphy, *Reducing the effort of bug report triage*, Transactions on Software Engineering and Methodology **20** (2011), no. 3.

[Ant] *Apache Ant* [online], URL: `http://ant.apache.org`.

[AOH06] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold, *JDiff: A differencing technique and tool for object-oriented programs*, Automated Software Engineering **14** (2006), no. 1.

[AP08] Nathaniel Ayewah and William Pugh, *A report on a survey and study of static analysis users*, Proceedings of the Workshop on Defects in Large Software Systems, 2008.

[Apa] *Apache HTTP* [online], URL: `http://httpd.apache.org`.

[APM+07] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou, *Evaluating static analysis defect warnings on production software*, Proceedings of the Workshop on Program Analysis for Software Tools and Engineering, June 2007.

[Arg] *ArgoUML* [online], URL: `http://argouml.tigris.org`.

[Art09] Cyrille Artho, *Iterative delta debugging*, Lecture Notes In Computer Science **5394** (2009).

[ASGA12] Nasir Ali, Aminata Sabané, Yann-Gaël Guéhéneuc, and Giuliano Antoniol, *Improving Bug Location Using Binary Class Relationships*, Proceedings of the International Working Conference on Source Code Analysis and Manipulation, September 2012.

[AV09] Jorge Aranda and Gina Venolia, *The secret life of bugs: Going past the errors and omissions in software repositories*, Proceedings of the International Conference on Software Engineering, 2009.

[BB01] Barry W. Boehm and Victor R. Basili, *Software Defect Reduction Top 10 List*, Computer **34** (2001), no. 1.

[BBA+09] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu, *Fair and Balanced?: Bias in Bug-Fix Datasets*, Proceedings of the European Software Engineering Conference/Symposium on the Foundations of Software Engineering, 2009.

[BJS+08] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiß, Rahul Premraj, and Thomas Zimmermann, *What makes a good bug report?*, Foundations of Software Engineering (2008).

[BKEL11] Matthew Beard, Nicholas A. Kraft, Letha H. Etzkorn, and Stacy K. Lukins, *Measuring the Accuracy of Information Retrieval Based Bug Localization Techniques*, Proceedings of the Working Conference on Reverse Engineering, 2011.

[BMO07] Gary Boetticher, Tim Menzies, and Thomas Ostrand, *PROMISE Repository of empirical software engineering data* [online], 2007, URL: `http://promisedata.org/?p=17`.

[BMW93] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster, *The concept assignment problem in program understanding*, Proceedings of the International Conference on Software Engineering, 1993.

[BNG+09] Christian Bird, Nachiappan Nagappan, Harald C. Gall, Brendan Murphy, and Premkumar Devanbu, *Putting it all together: using socio-technical networks to predict failures*, Proceedings of the International Symposium on Software Reliability Engineering, 2009.

[BNRR08] Shilpa Bugde, Nachiappan Nagappan, Sriram K. Rajamani, and G. Ramalingam, *Global Software Servicing: Observational Experiences at Microsoft*, Proceedings of the International Conference on Global Software Engineering, August 2008.

[BP84] Victor R. Basili and Barry T. Perricone, *Software errors and complexity: an empirical investigation*, Communications of the ACM **27** (1984), no. 1.

[BP88] Barry W. Boehm and Philip N. Papaccio, *Understanding and Controlling Software Costs*, IEEE Transactions on Software Engineering **14** (1988), no. 10.

[BPSZ10] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann, *Information needs in bug reports: improving cooperation between developers and users*, Proceedings of the Conference on Computer Supported Cooperative Work, 2010.

[BPZ08] Nicolas Bettenburg, Rahul Premraj, and Thomas Zimmermann, *Duplicate bug reports considered harmful . . . really?*, Proceedings of the International Conference on Software Maintenance, 2008.

[BPZK08] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim, *Extracting structural information from bug reports*, Proceedings of the Working Conference on Mining Software Repositories, 2008.

[Bre01] Leo Breiman, *Random Forests*, Machine Learning **45** (2001), no. 1.

[Buga] *Bugzilla* [online], URL: `http://www.bugzilla.org`.

[Bugb] *Life Cycle of a Bug* [online], URL: `http://www.bugzilla.org/docs/4.4/en/html/lifecycle.html`.

[BWK07] Stefan Berner, Roland Weber, and Rudolf K. Keller, *Enhancing Software Testing by Judicious Use of Code Coverage Information*, Proceedings of the International Conference on Software Engineering, May 2007.

[CC05] Gerardo Canfora and Luigi Cerulo, *Impact Analysis by Mining Software and Change Request Repositories*, Proceedings of the International Software Metrics Symposium, no. Metrics, 2005.

[CC06a] _____, *Fine grained indexing of software repositories to support impact analysis*, Proceedings of the International Conference on Software Engineering, 2006.

[CC06b] _____, *Supporting change request assignment in open source development*, Proceedings of the Symposium on Applied Computing, April 2006.

[CCW+01] Annie Chen, Eric Chou, Joshua Wong, Andrew Y. Yao, Qing Zhang, Shao Zhang, and Amir Michail, *CVSSearch: searching through source code using CVS comments*, Proceedings of the International Conference on Software Maintenance, 2001.

[CD09] Cagatay Catal and Banu Diri, *A systematic review of software fault prediction studies*, Expert Systems with Applications **36** (2009), no. 4.

[CEBE08] Brendan Cleary, Chris Exton, Jim Buckley, and Michael English, *An empirical analysis of information retrieval based concept location techniques in software comprehension*, Empirical Software Engineering **14** (2008), no. 1.

[CKEL11] Christopher S. Corley, Nicholas A. Kraft, Letha H. Etzkorn, and Stacy K. Lukins, *Recovering traceability links between source code and fixed bugs via patch analysis*, Proceedings of the International Workshop on Traceability in Emerging Forms of Software Engineering, 2011.

[CKF+02] Mike Y. Chen, Emre Kıcıman, Eugene Fratkin, Armando Fox, and Eric Brewer, *Pinpoint: Problem Determination in Large, Dynamic Internet Services*, Proceedings of the International Conference on Dependable Systems and Networks, 2002.

[ČM03] Davor Čubranić and Gail C. Murphy, *Hipikat: recommending pertinent software development artifacts*, Proceedings of the International Conference on Software Engineering, 2003.

[ČM04] _____, *Automatic bug triage using text categorization*, Proceedings of the International Conference on Software Engineering and Knowledge Engineering, 2004.

[CMM+11] Giulio Concas, Michele Marchesi, Alessandro Murgia, Roberto Tonelli, and Ivana Turnu, *On the Distribution of Bugs in the Eclipse System*, IEEE Transactions on Software Engineering **37** (2011), no. 6.

[CMSV11] Cesar Couto, Jo ao Eduardo Montandon, Christofer Silva, and Marco Tulio Valente, *Static correspondence and correlation between field defects and warnings reported by a bug finding tool*, Software Quality Journal (2011).

[CR00] Kunrong Chen and Václav Rajlich, *Case study of feature location using dependence graph*, Proceedings of the International Workshop on Program Comprehension, 2000.

[CR01] _____, *RIPPLES : Tool for Change in Legacy Software*, Proceedings of the International Conference on Software Maintenance, 2001.

[CSV+12] Cesar Couto, Christofer Silva, Marco Tulio Valente, Roberto Bigonha, and Nicolas Anquetil, *Uncovering Causal Relationships between Software Metrics and Bugs*, Proceedings of the European Conference on Software Maintenance and Reengineering, 2012.

[CVS] *CVS* [online], URL: `http://www.nongnu.org/cvs`.

[CYC+01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler, *An empirical study of operating systems errors*, Proceedings of the Symposium on Operating Systems Principles, vol. 35, October 2001.

[CZvD+09] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke, *A Systematic Survey of Program Comprehension through Dynamic Analysis*, IEEE Transactions on Software Engineering **35** (2009), no. 5.

[dB81] Jean-Charles de Borda, *Mémoire sur les élections au scrutin*, Mémoires de l'Académie Royale des Sciences (1781).

[Dif] *DiffJ* [online], URL: `http://www.incava.org/projects/1042574828`.

[DLP07] Marco D'Ambros, Michele Lanza, and Martin Pinzger, *A Bug's Life Visualizing a Bug Database*, Proceedings of the Working Conference on Software Visualization, 2007.

[DLZ05] Valentin Dallmeier, Christian Lindig, and Andreas Zeller, *Lightweight Defect Localization for Java*, Proceedings of the European Conference on Object-Oriented Programming, 2005.

[DRGP11] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk, *Feature location in source code: a taxonomy and survey*, Journal of Software Maintenance and Evolution (2011).

[EAAG08] M. Eaddy, A.V. Aho, Giuliano Antoniol, and Yann-Gaël Guéhéneuc, *CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis*, Proceedings of the International Conference on Program Comprehension, June 2008.

[Ecl] *Eclipse* [online], URL: `http://www.eclipse.org`.

[ED05] Andrew David Eisenberg and Kris De Volder, *Dynamic feature traces: finding features in unfamiliar code*, Proceedings of the International Conference on Software Maintenance, 2005.

[Emm] *Emma* [online], URL: `http://emma.sourceforge.net`.

[End75] Albert Endres, *An analysis of errors and their causes in system programs*, ACM SIGPLAN Notices **10** (1975), no. 6.

[EWSG09] Dennis Edwards, Norman Wilde, Sharon Simmons, and Eric Golden, *Instrumenting Time-Sensitive Software for Feature Location*, Proceedings of the International Conference on Program Comprehension, 2009.

[Fac] *Facebook* [online], URL: `https://developers.facebook.com`.

[FAW09] Javed Ferzund, Syed Nadeem Ahsan, and Franz Wotawa, *Empirical Evaluation of Hunk Metrics as Bug Predictors*, Proceedings of the International Workshop on Software Measurement, vol. 5891, 2009.

[Fin] *FindBugs* [online], URL: `http://findbugs.sourceforge.net`.

[Fir] *Mozilla Firefox* [online], URL: `http://www.mozilla.org/firefox`.

[Flo] *FLOSSMetrics* [online], URL: `http://flossmetrics.org`.

[FO00] Norman E. Fenton and Niclas Ohlsson, *Quantitative analysis of faults and failures in a complex software system*, IEEE Transactions on Software Engineering **26** (2000), no. 8.

[FPG03] Michael Fischer, Martin Pinzger, and Harald C. Gall, *Populating a Release History Database from version control and bug tracking systems*, Proceedings of the International Conference on Software Maintenance, 2003.

[FWPG07] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald C. Gall, *Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction*, IEEE Transactions on Software Engineering **33** (2007), no. 11.

[GDB] *GDB* [online], URL: `http://www.gnu.org/software/gdb/`.

[GDKP12] Malcom Gethers, Bogdan Dit, Huzefa Kagdi, and Denys Poshyvanyk, *Integrated Impact Analysis for Managing Software Changes*, Proceedings of the International Conference on Software Engineering, 2012.

[GE09] Philip J. Guo and Dawson Engler, *Linux kernel developer responses to static analysis bug reports*, Proceedings of the USENIX Annual Technical Conference, 2009.

[Gen] *Gensim* [online], URL: `http://radimrehurek.com/gensim`.

[Git] *Git* [online], URL: `http://git-scm.com`.

[Gre] *Grep* [online], URL: `http://pubs.opengroup.org/onlinepubs/9699919799/utilities/grep.html`.

[Gui83] Tor Guimaraes, *Managing application program maintenance expenditures*, Communications of the ACM **26** (1983), no. 10.

[GZNM10] Philip J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy, *Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows*, Proceedings of the International Conference on Software Engineering, 2010.

[Hal77] Maurice H. Halstead, *Elements of Software Science*, 1977.

[Has09] Ahmed E. Hassan, *Predicting faults using the complexity of code changes*, Proceedings of the International Conference on Software Engineering, 2009.

[HBB+12] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell, *A Systematic Literature Review on Fault Prediction Performance in Software Engineering*, IEEE Transactions on Software Engineering **38** (2012), no. 6.

[HKN02] Jason Van Hulse, Taghi M Khoshgoftaar, and Amri Napolitano, *Experimental Perspectives on Learning from Imbalanced Data*, Proceedings of the International Conference on Machine Learning, 2002.

[HNB+13] Hadi Hemmati, Sarah Nadi, Olga Baysal, Oleksii Kononenko, Wei Wang, Reid Holmes, and Michael W. Godfrey, *The MSR Cookbook Mining a Decade of Research*, Proceedings of the Working Conference on Mining Software Repositories, 2013.

[HP07] Emily Hill and Lori Pollock, *Exploring the Neighborhood with Dora*, Proceedings of the International Conference on Automated Software Engineering, 2007.

[HRK12] Emily Hill, Shivani Rao, and Avinash Kak, *On the Use of Stemming for Concern Location and Bug Localization in Java*, Proceedings of the International Working Conference on Source Code Analysis and Manipulation, September 2012.

[HRS+00] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi, *An Empirical Investigation of the Relationship Between Spectra Differences and Regression Faults*, Journal of Software Testing, Verification and Reliability **10** (2000), no. 3.

[HW07] Pieter Hooimeijer and Westley Weimer, *Modeling bug report quality*, Proceedings of the International Conference on Automated Software Engineering, 2007.

[ISO10] ISO/IEC/IEEE, *Systems and software engineering - Vocabulary*, Tech. report, 2010.

[Jab] *JabRef* [online], URL: `http://jabref.sourceforge.net`.

[JBH07] James A. Jones, James F. Bowring, and Mary Jean Harrold, *Debugging in Parallel*, Proceedings of the International Symposium on Software Testing and Assurance, 2007.

[JCS07] Ciera Christopher Jaspan, I-chin Chen, and Anoop Sharma, *Understanding the Value of Program Analysis Tools*, Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages and Applications, 2007.

[Jed] *jEdit* [online], URL: `http://jedit.org`.

[JHS02] James A. Jones, Mary Jean Harrold, and John Stasko, *Visualization of Test Information to Assist Fault Localization*, Proceedings of the International Conference on Software Engineering, 2002.

[Jir] *JIRA* [online], URL: `https://www.atlassian.com/software/jira`.

[Jme] *JMeter* [online], URL: `http://jmeter.apache.org`.

[Jod] *JodaTime* [online], URL: `http://joda-time.sourceforge.net`.

[Joh77] Stephen Johnson, *Lint, a C program checker*, Tech. report, Bell Laboratories, 1977.

[JOY09] Yungbum Jung, Hakjoo Oh, and Kwangkeun Yi, *Identifying static analysis techniques for finding non-fix hunks in fix revisions*, Proceedings of the Interntaional Workshop on Data-intensive Software Management and Mining, 2009.

[JPZ08] Sascha Just, Rahul Premraj, and Thomas Zimmermann, *Towards the next generation of bug tracking systems*, Proceedings of the Symposium on Visual Languages and Human-Centric Computing, September 2008.

[JW08] Nicholas Jalbert and Westley Weimer, *Automated duplicate detection for bug tracking systems*, Proceedings of the International Conference on Dependable Systems and Networks, 2008.

[KC10] Andrew J. Ko and Parmit K. Chilana, *How power users help and hinder open bug reporting*, Proceedings of the CHI Conference on Human Factors in Computing Systems, 2010.

[KCM07] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic, *A survey and taxonomy of approaches for mining software repositories in the context of software evolution*, Journal of Software Maintenance and Evolution **19** (2007), no. 2.

[KFGP09] Patrick Knab, Beat Fluri, Harald C. Gall, and Martin Pinzger, *Interactive views for analyzing problem reports*, Proceedings of the International Conference on Software Maintenance, September 2009.

[Kit10] Barbara A. Kitchenham, *What's up with software metrics? – A preliminary mapping study*, Journal of Systems and Software **83** (2010), no. 1.

[KMC06] Andrew J. Ko, Brad A. Myers, and Duen Horng Chau, *A Linguistic Analysis of How People Describe Software Problems*, Proceedings of the Symposium on Visual Languages and Human-Centric Computing, 2006.

[Kop06] Timo Koponen, *Life cycle of Defects in Open Source Software Projects*, International Federation for Information Processing **203** (2006).

[Koz92] John R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, 1992.

[KT04] A. Güneş Koru and Jeff Tian, *Defect handling in medium and large open source projects*, IEEE Software **21** (2004), no. 4.

[KW06] Sunghun Kim and E. James Whitehead Jr., *How long did it take to fix bugs?*, Proceedings of the International Conference on Software Engineering, May 2006.

[KWZ08] Sunghun Kim, E. James Whitehead Jr., and Yi Zhang, *Classifying Software Changes : Clean or Buggy ?*, IEEE Transactions on Software Engineering **34** (2008), no. 2.

[KZPW06] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Whitehead Jr., *Automatic Identification of Bug-Introducing Changes*, Proceedings of the International Conference on Automated Software Engineering, 2006.

[KZWZ07] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller, *Predicting Faults from Cached History*, Proceedings of the International Conference on Software Engineering, May 2007.

[Leh80] Meir M. Lehman, *Programs, life cycles, and laws of software evolution*, Proceedings of the IEEE **68** (1980), no. 9.

[Lin] *How to Get Your Change Into the Linux Kernel* [online], URL: `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/SubmittingPatches?id=8543ae1296f6ec1490c7afab6ae0fe97bf87ebf8`.

[LKE08] Stacy K. Lukins, Nicholas A. Kraft, and Letha H. Etzkorn, *Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation*, Proceedings of the Working Conference on Reverse Engineering, October 2008.

[LKE10] _____, *Bug localization using latent Dirichlet allocation*, Information and Software Technology **52** (2010), no. 9.

[LMPR07] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Václav Rajlich, *Feature location via information retrieval based filtering of a single scenario execution trace*, Proceedings of the International Conference on Automated Software Engineering, 2007.

[LVD06] Thomas D. LaToza, Gina Venolia, and Robert DeLine, *Maintaining Mental Models : A Study of Developer Work Habits*, Proceedings of the International Conference on Software Engineering, 2006.

[LYF+05] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff, *SOBER: Statistical Model-based Bug Localization*, ACM SIGSOFT Software Engineering Notes **30** (2005), no. 5.

[McC76] T.J. McCabe, *A Complexity Measure*, IEEE Transactions on Software Engineering **SE-2** (1976), no. 4.

[MCMT10] Alessandro Murgia, Giulio Concas, Michele Marchesi, and Roberto Tonelli, *A machine learning approach for text categorization of fixing-issue commits on CVS*, Proceedings of the International Symposium on Empirical Software Engineering and Measurement, 2010.

[MFH02] Audris Mockus, Roy T Fielding, and James D Herbsleb, *Two case studies of open source software development: Apache and Mozilla*, Transactions on Software Engineering and Methodology **11** (2002), no. 3.

[MNDT09] Audris Mockus, Nachiappan Nagappan, and Trung T. Dinh-Trong, *Test coverage and post-verification defects: A multiple case study*, Proceedings of the International Symposium on Empirical Software Engineering and Measurement, October 2009.

[Moz] *Mozilla* [online], URL: `http://www.mozilla.org`.

[MSR] *Working Conference on Mining Software Repositories* [online], URL: `http://www.msrconf.org`.

[MSRM04] Andrian Marcus, Andrey Sergeyev, Václav Rajlich, and Jonathan I. Maletic, *An information retrieval approach to concept location in source code*, Proceedings of the Working Conference on Reverse Engineering, 2004.

[Muc] *muCommander* [online], URL: `http://www.mucommander.com`.

[Nat02] National Institute Of Standards & Technology, *The Economic Impacts of Inadequate Infrastructure for Software Testing (Planning Report #02-3)*, Tech. report, 2002.

[NBZ06] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller, *Mining metrics to predict component failures*, Proceedings of the International Conference on Software Engineering, 2006.

[NGS+10] Mangala Gowri Nanda, Monika Gupta, Saurabh Sinha, Satish Chandra, David Schmidt, and Pradeep Balachandran, *Making defect-finding tools work for you*, Proceedings of the International Conference on Software Engineering, 2010.

[NNAK+11] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen, *A topic-based approach for narrowing the search space of buggy files from a bug report*, Proceedings of the International Conference on Automated Software Engineering, November 2011.

[NNN13] Hoan Anh Nguyen, Anh Tuan Nguyen, and Tien N Nguyen, *Filtering Noise in Mixed-Purpose Fixing Commits to Improve Defect Prediction and Localization*, Proceedings of the International Symposium on Software Reliability Engineering, 2013.

[OGKW08] Leon J. Osterweil, Carlo Ghezzi, Jeff Kramer, and Alexander L. Wolf, *Determining the Impact of Software Engineering Research on Practice*, Computer **41** (2008), no. 3.

[OHK07] Olga Ormandjieva, Ishrar Hussain, and Leila Kosseim, *Toward a text classification system for the quality assessment of software requirements written in natural language*, Proceedings of the Symposium on the Foundations of Software Engineering, 2007.

[OJ82] Frank Owen and Ron Jones, *Statistics*, Polytech, 1982.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein, *The program dependence graph in a software development environment*, ACM Sigplan Notices (1984).

[PGM+06] Denys Poshyvanyk, Yann-Gaël Guéhéneuc, Andrian Marcus, Giuliano Antoniol, and Václav Rajlich, *Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification*, Proceedings of the International Conference on Program Comprehension, 2006.

[PGM+07] _____, *Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval*, IEEE Transactions on Software Engineering **33** (2007), no. 6.

[PLF+03] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang, *Automated support for classifying software failure reports*, Proceedings of the International Conference on Software Engineering, May 2003.

[PMD] *PMD* [online], URL: `http://pmd.sourceforge.net`.

[PMD06] Denys Poshyvanyk, Andrian Marcus, and Yubo Dong, *JIRiSS-an eclipse plug-in for source code exploration*, Proceedings of the International Conference on Program Comprehension, 2006.

[PMDS05] Denys Poshyvanyk, Andrian Marcus, Yubo Dong, and Andrey Sergeyev, *IRiSS – A Source Code Exploration Tool*, Proceedings of the International Conference on Software Maintenance, 2005.

[PO11] Chris Parnin and Alessandro Orso, *Are Automated Debugging Techniques Actually Helping Programmers?*, Proceedings of the International Symposium on Software Testing and Assurance, 2011.

[POC93] Paul Piwowarski, Mitsuru Ohba, and Joe Caruso, *Coverage Measurement Experience During Function Test*, Proceedings of the International Conference on Software Engineering, 1993.

[Pos] *PostgreSQL* [online], URL: `http://www.postgresql.org`.

[PRV08] Maksym Petrenko, Václav Rajlich, and Radu Vanciu, *Partial Domain Comprehension in Software Evolution and Maintenance*, Proceedings of the International Conference on Program Comprehension, June 2008.

[Pye] *Pyevolve* [online], URL: `http://pyevolve.sourceforge.net`.

[Rac] *Rachota* [online], URL: `http://rachota.sourceforge.net`.

[Ram08] Rudolf Ramler, *The impact of product development on the lifecycle of defects*, Proceedings of the International Symposium on Software Testing and Assurance, 2008.

[RAN07] Per Runeson, Magnus Alexandersson, and Oskar Nyholm, *Detection of Duplicate Defect Reports Using Natural Language Processing*, Proceedings of the International Conference on Software Engineering, May 2007.

[Ray00] Eric S. Raymond, *The Cathedral and the Bazaar*, September 2000.

[RBDL97] Thomas Reps, Thomas Ball, Manuvir Das, and James Larust, *The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem*, Proceedings of the European Software Engineering Conference/Symposium on the Foundations of Software Engineering, 1997.

[RHTŽ13] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič, *Software fault prediction metrics: A systematic literature review*, Information and Software Technology **55** (2013), no. 8.

[RK11] Shivani Rao and Avinash Kak, *Retrieval from Software Libraries for Bug Localization : A Comparative Study of Generic and Composite Text Models*, Proceedings of the Working Conference on Mining Software Repositories, 2011.

[RM02] Martin P. Robillard and Gail C. Murphy, *Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies*, Proceedings of the International Conference on Software Engineering, 2002.

[RMM10] Sarah Rastkar, Gail C. Murphy, and Gabriel Murray, *Summarizing Software Artifacts : A Case Study of Bug Reports*, Proceedings of the International Conference on Software Engineering, 2010.

[Rob08] Martin P. Robillard, *Topology analysis of software dependencies*, ACM Transactions on Software Engineering and Methodology **17** (2008), no. 4.

[RR07] Xiaoxia Ren and Barbara G. Ryder, *Heuristic ranking of java program edits for fault localization*, Proceedings of the International Symposium on Software Testing and Assurance, 2007.

[SB88] Gerard Salton and Christopher Buckley, *Term Weighting Approaches In Automatic Text Retrieval*, Information Processing & Management **24** (1988), no. 5.

[SBP10] Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj, *Do stack traces help developers fix bugs?*, Proceedings of the Working Conference on Mining Software Repositories, May 2010.

[SC92] Mark Sullivan and Ram Chillarege, *A comparison of software defects in database management systems and operating systems*, FTCS-22. Digest of Papers., 1992.

[Sci] *scikit-learn* [online], URL: `http://scikit-learn.org`.

[SGR04] Robert J. Sandusky, Les Gasser, and Gabriel Ripoche, *Bug Report Networks: Varieties, Strategies, and Impacts in a F/OSS Development Community*, Proceedings of the Working Conference on Mining Software Repositories, 2004.

[SI10] Ashish Sureka and Kishore Varma Indukuri, *Linguistic analysis of bug report titles with respect to the dimension of bug importance*, Proceedings of the ACM Bangalore Conference, January 2010.

[SIK+12] Emad Shihab, Akinori Ihara, Yasutaka Kamei, Walid M. Ibrahim, Masao Ohira, Bram Adams, Ahmed E. Hassan, and Ken-ichi Matsumoto, *Studying re-opened bugs in open source software*, Empirical Software Engineering **18** (2012), no. 5.

[SJAK09] Shivkumar Shivaji, E. James Whitehead Jr., Ram Akella, and Sunghun Kim, *Reducing Features to Improve Bug Prediction*, November 2009.

[SLVA97] Janice Singer, Timothy C. Lethbridge, Norman Vinson, and Nicolas Anquetil, *An examination of software engineering work practices*, Proceedings of the Centre for Advanced Studies Conference, November 1997.

[SLW+10] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo, *A discriminative model approach for accurate duplicate bug report retrieval*, Proceedings of the International Conference on Software Engineering, May 2010.

[SP89] Richard W. Selby and Adam A. Porter, *Software Metric Classification Trees Help Guide the Maintenance of Large-scale Systems Automatic Classification Tree*, Proceedings of the International Conference on Software Maintenance, 1989.

[SRC08] Philipp Schuegerl, Juergen Rilling, and Philippe Charland, *Enriching SE ontologies with bug report quality*, Proceedings of the International Workshop on Semantic Web Enabled Software Engineering, 2008.

[SSR10] Vibha Singhal Sinha, Saurabh Sinha, and Swathi Rao, *BUGINNINGS: identifying the origins of a bug*, Proceedings of the India Software Engineering Conference, 2010.

[Str] *strace* [online], URL: `http://sourceforge.net/projects/strace`.

[SVN] *SVN* [online], URL: `http://subversion.tigris.org`.

[ŚZZ05a] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller, *HATARI: Raising Risk Awareness*, Proceedings of the European Software Engineering Conference/Symposium on the Foundations of Software Engineering, 2005.

[ŚZZ05b] ———, *When do changes induce fixes?*, Proceedings of the Working Conference on Mining Software Repositories, 2005.

[TNBH13] Stephen W. Thomas, Meiyappan Nagappan, Dorothea Blostein, and Ahmed E. Hassan, *The Impact of Classifier Configuration and Classifier Combination on Bug Localization*, IEEE Transactions on Software Engineering **39** (2013), no. 10.

[TR10] Marco Torchiano and Filippo Ricca, *Impact Analysis by means of Unstructured Knowledge in the Context of Bug Repositories*, Proceedings of the International Symposium on Empirical Software Engineering and Measurement, 2010.

[Tri08] Mircea Trifu, *Using Dataflow Information for Concern Identification in Object-Oriented Software Systems*, Proceedings of the European Conference on Software Maintenance and Reengineering, April 2008.

[Tru] *truss* [online], URL: `http://docs.oracle.com/cd/E19082-01/819-2239/truss-1/index.html`.

[Vie] *ViewVC* [online], URL: `http://www.viewvc.org/`.

[WCF96] Norman Wilde, Christopher Casey, and West Florida, *Early Field Experience with the Software Reconnaissance Technique for Program Comprehension*, Proceedings of the International Conference on Software Maintenance, 1996.

[WD09] W. Eric Wong and Vidroha Debroy, *A Survey of Software Fault Localization*, Tech. Report UTDCS-45-09, 2009.

[Wek] *Weka* [online], URL: `www.cs.waikato.ac.nz/ml/weka`.

[WH05] Chadd C. Williams and Jeffrey K. Hollingsworth, *Automatic mining of source code repositories to improve bug finding techniques*, IEEE Transactions on Software Engineering **31** (2005), no. 6.

[WLXJ11] Shaowei Wang, David Lo, Zhenchang Xing, and Lingxiao Jiang, *Concern Localization using Information Retrieval : An Empirical Study on Linux Kernel*, Proceedings of the Working Conference on Reverse Engineering, 2011.

[WPZZ07] Cathrin Weiß, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller, *How Long Will It Take to Fix This Bug?*, Proceedings of the International Conference on Software Engineering, no. 2, May 2007.

[WS95] Norman Wilde and Michael C. Scully, *Software Reconnaissance: Mapping program features to code*, Journal of Software Maintenance: Research and Practice **7** (1995), no. 1.

[WS08] Chadd C. Williams and Jaime Spacco, *SZZ revisited: verifying when changes induce fixes*, Proceedings of the Workshop on Defects in Large Software Systems, 2008.

[WZ12] Jue Wang and Hongyu Zhang, *Predicting defect numbers based on defect state transition models*, Proceedings of the International Symposium on Empirical Software Engineering and Measurement, 2012.

[WZKC11] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and S. C. Cheung, *ReLink : Recovering Links between Bugs and Changes*, Proceedings of the European Software Engineering Conference/Symposium on the Foundations of Software Engineering, 2011.

[WZX+08] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun, *An approach to detecting duplicate bug reports using natural language and execution information*, Proceedings of the International Conference on Software Engineering, May 2008.

[YLW06] Qian Yang, J. Jenny Li, and David Weiss, *A survey of coverage based testing tools*, Proceedings of the International Workshop on Automation of Software Test, 2006.

[YMX+10] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy, *SherLog : Error Diagnosis by Connecting Clues from Run-time Logs*, Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, 2010.

[ZH02] Andreas Zeller and Ralf Hildebrandt, *Simplifying and Isolating Failure-Inducing Input*, IEEE Transactions on Software Engineering **28** (2002), no. 2.

[Zha04] Harry Zhang, *The optimality of naive Bayes*, Proceedings of the International Conference of the Floride Artifical Intelligence Research Society, 2004.

[ZNGM12] Thomas Zimmermann, Nachiappan Nagappan, Philip J. Guo, and Brendan Murphy, *Characterizing and Predicting Which Bugs Get Reopened*, Proceedings of the International Conference on Software Engineering, 2012.

[ZPZ07] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller, *Predicting Defects for Eclipse*, Proceedings of the International Conference on Predictive Models in Software Engineering, May 2007.

[ZWDZ05] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller, *Mining version histories to guide software changes*, IEEE Transactions on Software Engineering **31** (2005), no. 6.

[ZZL+06] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang, *SNIAFL: Towards a Static Noninteractive Approach to Feature Location*, Transactions on Software Engineering and Methodology **15** (2006), no. 2.

[ZZL12] Jian Zhou, Hongyu Zhang, and David Lo, *Where Should the Bugs Be Fixed ? - more accurate information retrieval-based bug localization based on bug reports*, Proceedings of the International Conference on Software Engineering, 2012.