Are Developers Fixing Their Own Bugs? Tracing Bug-fixing and Bug-seeding Committers

Daniel Izquierdo-Cortazar,
Andrea Capiluppi, Jesus M. Gonzalez-Barahona
Universidad Rey Juan Carlos, University of East London
{dizquierdo, jgb}@gsyc.es, a.capiluppi@uel.ac.uk

November 30, 2012

Abstract

The process of fixing software bugs plays a key role in the maintenance activities of a software project. Ideally, code ownership and responsibility should be enforced among developers working on the same artifacts, so that those introducing buggy code could also contribute to its fix. However, especially in FLOSS projects, this mechanism is not clearly understood: in particular, it is not known whether those contributors fixing a bug are the same *introducing* and *seeding* it in the first place.

This paper aims to study this issue, by analysing the *comm-central* FLOSS project, which hosts part of the Thunderbird, SeaMonkey, Lightning extensions and Sunbird projects from the Mozilla community. The analysis is focused at the level of lines of code and it uses the information stored in the source code management system.

The results of this study show, at first, that in 80% of the cases, the bug-fixing activity involves source code modified by at most two developers. It also emerges that the developers fixing the bug are only responsible for 3.5% of the previous modifications to the lines affected; this implies that the other developers making changes to those lines could have made that fix. We conclude by stating that, in most of the cases the bug fixing process in *comm-central* is not carried out by the same developers than those who *seeded* the buggy code.

1 Introduction

One of the most recognised advantages of the Free/Libre/Open Source Software (FLOSS) development model is its reliance on an open process: anyone is welcome to contribute; the majority of developers can focus on modularised, limited sections within a very large and complex system; and few core developers are generally experts in several areas of the source code, in a well accepted layered model (the "onion model" [Mockus et al., 2002]). These layers have been connected to actual responsibilities; core developers should focus on the main, more important

features, while experimental versions should be implemented and tested by contributors on the development fringes [Goldman and Gabriel, 2004]. Also, the layers of such model have been related to a shift in productivity: a recurring finding within FLOSS empirical research has shown that most of the development work is achieved by a small amount of developers, in a typical Pareto distribution [Koch, 2009].

The combinations of all the findings above have various, and not completely understood, effects. In some cases, a strong *territoriality* will emerge among developers "owning" certain parts of the code, and becoming more and more proficient in those [German, 2004, Robles et al., 2006]. In other cases, the very nature of the FLOSS development implies that contributors join and then leave without necessarily halting the project [Robles and González-Barahona, 2006], but resulting in abandoned code and orphaned lines [Izquierdo-Cortazar et al., 2009].

Finally, certain developers will need to be active in maintenance activities: *corrective* maintenance fixing bugs in various parts of the code, for instance when source code is first introduced by developers with a low knowledge of the project (junior developers); *perfective* maintenance, for instance when new improved features are needed but the original developers have left the project and abandoned their contributions [Adams et al., 2009]; *adaptive* maintenance, for instance when adaptations are needed, but the source code has been contributed in a programming language different from the main one supported by the project, so the current developers do not have enough skills in that language. Although in specific FLOSS communities there is the shared expectation that the original contributor will support his/her modules (especially in highly modular FLOSS projects, as Moodle or Drupal [Capiluppi et al., 2010]), the volatility of contributors and the process of bug-fixing need to be clarified with respect of who introduced a certain bug, and who contributed the code to fix it. Examining and determining the proportion of errors that are fixed by different developers than those who introduced the error could provide a first approach to better understand the bug-fixing process in the specific FLOSS communities being studied.

In order to tackle this problem, the present study analyses the code base contained within the *comm-central* project[1], a Mercurial Software Configuration Management (SCM) repository of Mozilla components (Thunderbird, SeaMonkey, the Lightning extension and Sunbird). Given the number and ID of each fixed bug, this research evaluates which changes have been performed, and by who, in the process of fixing the specific bug. The objective of this research is to evaluate patterns of bug-fixing activities within this FLOSS community, in order to detect, if any, the most recurrent and relevant scenarios among developers fixing bugs and those seeding the problem in the first place.

This paper makes two main contributions:

1. *Identifying bug-fixing and bug-seeding committers*: the detection of those commits that have fixed a bug is crucial to determine the previous changes that took place to *seed* that bug. Using the source code lines that were handled by committers and tracing their history back make possible to know who previously handled those lines. Thus, it is possible to trace the changes in the SCM that made possible the birth of a potential bug. In addition, it has been detected the existence of exceptional large movements of lines in just one commit what may provoke distortions in the results and were left as open research questions.

[1] `http://hg.mozilla.org/comm-central`

2. *Characterization of bug-seeding activity*: once the bug-seeding commits have been detected, it is also interesting to know how many developers have been involved in those commits that later has been raised as a bug. With this approach, we are able to know the number of people that added or modified a piece of source code before it was detected as an issue by the community.

The paper is organized in the following sections: section 2 analyzes the related work and the background for the study; section 3 and 4 introduce the technique used to extract data from the Mercurial SCM based on the *hg diff* tool. Section 5 presents the main results found after using the proposed method *comm-central*, while section 6 raises a set of threats to validity. Finally section 7 concludes the paper with pointers towards further work.

2 Background and Related Work

This section reports on the related work and the existing tool-sets: it is reported here in order to show how this research builds on, compares to or complements existing approaches and results.

This paper uses the *diff* tool to identify changes between revisions: *diff* is provided by several source code management systems, and its basic algorithm has been theoretically and extensively explained ([Ukkonen, 1985, Miller and Myers, 1985, Myers, 1986]). This tool basically collects two revisions of a file (or revisions of the same directory) and it returns the differences found between them. Its main goal is to look for "plain" differences between two files: however, its implementation contains both a way to identify the "actual" differences between two files, and a facility to ignore "apparent" differences (e.g., spaces, indentations, newlines, etc). The GNU implementation of this algorithm is explained in [MacKenzie et al., 2002]: this paper uses the "unified" format of the *diff* algorithm to retrieve all the differences between each two revisions of the source code found at the Mercurial repository of the *comm-central* project. Other researchers used the *diff* tool in their approach when retrieving data from FLOSS repositories (specifically CVS and Subversion [Canfora et al., 2007, Zimmermann et al., 2006]).

Previous studies have made use both of SCM repositories and log messages left by developers, as a way to determine whether an observed activity is a bug-fixing process or not. Focus has been given to how developers should know precisely how this is being carried out (i.e., the process) and by whom (i.e., the responsibilities [Guo et al., 2010]). Some authors [Kim et al., 2008, Sliwerski et al., 2005] have worked at this level; however, it has to become clear that some FLOSS communities are more effective than others in documenting whether a commit is fixing an existing bug, or if it is a more generic maintenance activity. The present study is only based on the Mozilla Community, since within this community, it is relatively simple (compared to other communities) to determine if one of its commits is related to a bug in the Bug Tracking Systems. In this community, and within the SCM recorded activities, most of the commits dealing with bug fixes (or related to an open bug report) are tagged with an initial word "bug" or "Bug". In some rare occasions, these have been detected to be generic features and not real bugs. A cross-validation is performed below, in order to visualise the precision and recall of this approximation, and it is shown that the above mismatch represents a minor number of occurrences.

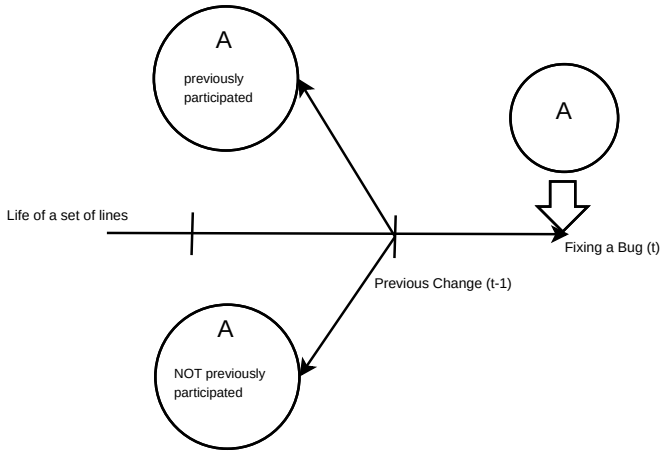

Figure 1: Scenarios of committers and lines changed. Lines that are introduced in a given fix time (t-1) are later (t) detected as being part of a bug-fixing commit. Thus, the set of lines that are being handled in (t) could have been previously introduced by the same developer (A), partially introduced by the same developer or introduced by a different developer.

Similarly to previous studies [Kim et al., 2008, Śliwerski et al., 2005], this research is performed at the granularity level of source lines, which provides a way of handling the ambiguity of working with commits. When considering the committer A who fixes a certain bug, and the lines she modifies, some of these lines could have been introduced fully or partially by the same committer, or introduced by different committers without the participation of A (pictured in Figure 1). Extending these two basic scenarios, we could find further scenarios:

- the same set of lines was modified in a previous commit by the same developer A (only);
- the same set of lines was modified in a previous commit by a different developer B (only);
- the same set of lines was modified by more than one developer (A+B+C+...), including the same developer A fixing the bug;
- the same set of lines was modified by more than one developer (B+C+D+...), but excluding the developer fixing the bug;

In terms of relating the bug-fixing process and its responsibilities, some authors have dealt with the idea of who should be fixing a certain bug [Kagdi et al., 2008, Ma et al., 2009] based on previous changes of the same file, or at least slices of the changes introduced in a file. Another approach used to deal with the same problem has been adopted at the level of the bug tracking system. In a study based on the development of Microsoft Windows Vista and Windows 7, it has been found that the number of reports “opened” by one developer and initially “assigned” to her development team tend to be fixed more quickly than bugs that are assigned to another development team [Guo et al., 2010]. Finally, it has also been reported that specific FLOSS communities try and reinforce a per-contributor sense of responsibility: in highly modular projects (as for instance

Moodle or Drupal), for example, it is a shared expectation within the community that the original contributor will support his/her modules [Capiluppi et al., 2010] and keep them in sync with the evolution of the core system [Hao-Yun Huang and Panchal, 2010]. Finally, other authors have dealt with the idea of looking for bug-fixing patterns in the source code [Pan et al., 2009] analyzing the different revisions provided by a given SCM system, but focusing on the semantics of the source code. In other words, they are aware of several common fix patterns such as ”addition of precondition check” or ”different method call to a class instance”. However, at the level of the source code, and to the best of our knowledge, no studies aiming to determine if developers that fixed the bug are the same than those who introduced the bug have been undertaken.

3 Assumptions and Definitions

3.1 Assumptions – SZZ algorithm

This paper makes use of the SZZ algorithm [Śliwerski et al., 2005], whose main goal is to determine the origin of a bug, by identifying the bug-fixing commits, and by using a *diff* tool. The authors of this algorithm assume that the lines that have been *removed* of *modified* in the bug-fixing commit are the ones where the bug was located. Thus, tracing back the origins of those lines (by means of the *annotate* command in the SCM), the authors could reach the origins of those lines, and admittedly, the origins of the bug. Generally speaking, the first modification or the addition of those *modified* or *removed* lines can be accounted as the origin of that bug. The operationalization of the algorithm used in this paper is slightly different, but based on the same assumption: the lines affected in the process of fixing a bug are the same one that originated or *seeded* that bug.

3.2 Definitions

This study is based on (and could be extended to other) projects which make use of a distributed SCM system called *Mercurial*. For each of the analyzed projects, the log provided by each of the named SCMs was analysed. For this purpose, the definitions used in this empirical case study are the following:

- *Commit* (or revision): change to the source code submitted to the SCM system. This updates the current version of the tree directory with a new set of changes. Such changes are generally summarized in a *patch* which is a set of lines with specific information about the affected files, but also about the affected lines.

- *Committer*: person who has rights to commit to a specific SCM repository, hence allowed to make changes. The Mercurial case presents some peculiarities: the developers working as maintainers and uploading changes to the main branch of the repository are not registered by the Mercurial SCM. Thus, all of the changes are initially considered as uploaded by the original author[2]. Thus, through this paper, the concept of developer, committer or author

[2] For more information regarding this issue, the Mercurial website offers a set of third part extensions where this issue could be solved: `http://mercurial.selenic.com/wiki/UsingExtensions`

will be considered as synonyms. Nevertheless, depending on the SCM, those concepts are slightly different.

- *Bug-fixing commit*: this is a special type of commit where issues reported by other developers have been fixed. In the comm-central repository this is generally reported in the title of a commit by referencing a “bug” or a “Bug”.

- *Line*: this is the basic piece of information of this study and they are generally handled by committers. These lines could be *added* - new line, *modified* - modification of some part of that line and *removed* - there is a deletion of that line.

- *Bug seeding-commit*: given a commit, and the output of the *hg diff* command, it is possible to obtain a complete picture of the lines that were added, modified or removed, but also about the committer, the date and which files were handled. This is necessary both to track which lines have been changed for fixing a bug, and to track which committers have provided changes to the same set of lines in previous commits. Figure 2 shows how the latter identification has been achieved. In the example file (far right), three sets of lines can be recognised (“set of lines 1”, “set of lines 2” and “set of lines 3”): the first two sets are affected by changes, the third has been unchanged throughout.

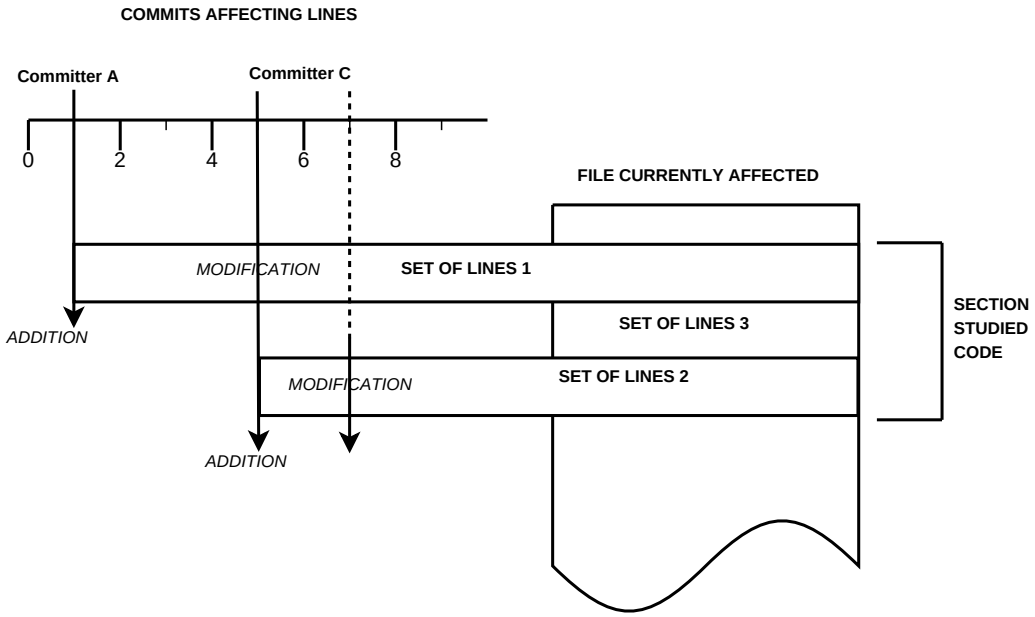

Figure 2: Identifying previous changes and committers

Tracking back the history of each set in the database, we are able to know that “set of lines 1” was added in commit number 1 and then modified in commit number 5. With respect to “set of lines 2”, they were added in commit number 5 and later modified in commit number 7. With respect to the authorship, we know that the “set of lines 1” was added by a developer named A. The modification of “set of lines 1” and the addition of the “set of lines 2” was

done by the same committer, named C and finally, in commit 7 changes were made on the ”set of lines 2” by developer C. In this specific figure, other commits might have happened, but they have not modified or removed the set of lines we are interested in. Specifically, commit number 2, 3, 4 and 6 took place, but none of them modified the studied sets of lines.

4 Empirical Approach and Operationalization

As the main goal of this research, this paper aims to identify and characterise the bug-fixing and bug-seeding activities in FLOSS communities. From a managerial perspective, the bug-seeding activity could be useful to clarify how and when the buggy source code has been introduced into the repository, how developers deal with this, and which effort needs to be applied and by who. In addition, specific sub-questions were formulated to achieve the main goal of the paper:

1. How are the bugs in *comm-central* recorded and referred to by developers? What is the accuracy and consistency of recording such bug-fixing information?

   Rationale: from the maintenance point of view, it is necessary to study how the community records which issues have being fixed. The empirical approach used in this paper is based on the information provided by the log message left by the developers when fixing a bug. This information depends on the analysed community (i.e., Mozilla), and it could be recorded differently in other communities.

2. How can one define the *bug-fixing* and *bug-seeding* activities when tracking the same set of lines?

   Rationale: this question is related to the detection of bug-seeding commits that later were classified as “buggy” by the community. And more specifically, how they are detected by means of the differences found between each pair of revisions in the source code.

3. Are there specific events in the activity log that could impede the correct tracking of such set of lines? How to avoid that such events interfere with the tracking of a given set of lines?

   Rationale: Some events in the community could force to move huge quantities of source code to another repository (e.g., in case of migrations), refactoring (e.g., when changing loads of methods names), license requirements (e.g., when migrating to another license) or others. These factors can cause large peaks to be visualised in the evolutionary trends, that could artificially skew the results.

4. Are there recurring patterns of bug-fixing among the developers of the *comm-central* community?

   Rationale: this questions aims to study the behaviour of developers when fixing bugs and try to look for specific patterns of bug-seeding activity. It is still not well understood how bugs are being introduced in the source code and if those developers that usually introduce issues are the same ones in charge of fixing them. Another interesting question is the one related to how many people are usually introducing changes to the same piece of source code that later is found to be “buggy”.

4.1 Understanding the *diff* output

Past research studies have focused on source code lines in two ways, either by using the source code management system (SCM) hosting the project, or by first downloading the source code from the repository, and then using the *diff* tool provided by the operating systems. In the first case, it is necessary to download the SCM and later use the diff tool provided within, but most researchers avoid that mostly due to the bottleneck represented by the network. In the latter, one has to download the source code for all the revisions of all the files contained in a software system. Using a distributed SCM such as Git or Mercurial (instead of a traditional SCM, as CVS or SVN), the bottleneck of the network is removed and the corresponding analysis becomes much faster. As documented in section 6, this approach still holds some limitations, that have to be addressed in the threats to validity.

A diff is a summary of the changes undertaken between two files, and stored in a SCM system. The diff command compares the files line by line and summarizes the differences in a specific format. Below, the partial output of a unified diff format between two commits (12 and 13) in the *comm-central* repository is shown. This example is not specific from the source code since this is a special file to build the project, however it is simple enough to be easily understood.

```
diff -r f1...1d -r 0b...f7 suite/build.mk
--- a/suite/build.mk Fri Jul 25 11:32:27 2008
+++ b/suite/build.mk Fri Jul 25 11:51:57 2008
@@ -43,6 +43,10 @@

 TIERS += app

+ifdef MOZ_COMPOSER
+tier_app_dirs += editor/ui
+endif
+
 ifdef MOZ_CALENDAR
 MOZ_EXTENSIONS += webdav
 endif
```

The *hg diff* command, by default, shows the diff between two revisions using the unified format: the diff format starts with two-lines header where the original file name is preceded by $---$ and the new file is preceded by $+++$. After this, there are one or more change hunks (usually named as *chunks*) which contain information related to the differences in the file. Those lines which were added starts with a $+$ character, those removed starts with a $-$ character and those which were neither added, nor removed starts with a space character " ". Finally, if a line is modified, this is represented as added and removed, so this changes will appear adjacent to one another. Thus, if a set of adjacent lines are modified, the old revision of the lines will show several lines

beginning with $-$, adjacent to the new revision of the lines, and beginning with $+$. In the previous example, four lines have been added in a file called "build.mk". The values between "@@" represents the position of those lines in that file before and after the change). For more information it is recommended to read the reference [MacKenzie et al., 2002].

4.2 Retrieving Information from *diff* Files

A freely available tool has been used to retrieve information from consists of several steps that are specified in the following list:

1. *Downloading the SCM*: the BlameMe tool is specifically designed to work with Git or Mercurial repositories. These are distributed SCM and provide all of the change history locally. This is an advantage if compared to other centralized SCMs such as CVS or SVN since there is an actual and huge difference in terms of time (avoiding the bottleneck represented by the network access).

2. *Collecting Commits*: as seen above, the very *hg* command provides a special command to check differences between two revisions: *hg diff*. This has been used to interact between the program and their Mercurial repository.

3. *Parsing the revisions*: the tool is launched using the previously downloaded repository and storing all the differences in a MySQL database. For this purpose, each of the lines is stored together with its reference to its file and the position in that file (specifically, there is a list per file, and each node is the position in that file for a given line). If a new set of lines are detected to be added or removed, those are directly added in the specified position (explained in section 4.1.

4.3 Case Study

The proposed method has been applied to describe the bug-fixing process at the level of source lines using the *comm-central* project and its Mercurial repository. As mentioned, this repository contains the source code of Thunderbird, SeaMonkey, Lightning extension and Sunbird [3].

The use of the Mercurial repository (after the migration from CVS) started on the 22th of July, 2008 and it has been studied till the 20th of July, 2010 (i.e., two years of source code history). Considering the whole life of the project until the start of this study, 5,982 commits were studied and the differences between revisions have been stored in a MySQL database. In this database, we have stored information of 4,973,038 changes to the source code regarding added, modified and removed lines.

The case studies presented in this paper are based on the differences between two revisions of the source code, and specifically focused on the bug-fixing commits. The commits studied are 2,969 out of an overall 5,982 commits; the total amount of lines considered are 2,912,866.

[3] However, as addressed in `https://developer.mozilla.org/en/comm-central`, this only includes a subset of the code required to build those projects.

5 Results

This section provides the results of the empirical study performed on the comm-central repository, in three parts: first, the study of how to properly detect bug-fixing commits is reported, detailing on the precision and recall in such process. Second, the issue of dealing with large commits is presented and addressed. Third, the approach of detecting bug-fixing and bug-seeding committers is clustered in several scenarios, and finally the results on each scenarios are proposed.

5.1 Identifying bug-fixing commits

This first part of the research aims to validate the log messages provided by the *comm-central* community, and to understand the consistency and reliability of their records with regards to bug-fixing activities. To achieve this purpose, we developed an empirical approach and then checked how many false positives and false negatives we obtained from applying it. The approach used is as follows:

1. Given 3,000 bug-fixing commits, and a confidence level of 95%, the random sample was sized in 100 commits. From each of those, the log message was retrieved and the log message inspected.

2. A simple heuristic, based on the observation of the log message and used by another paper analyzing the Mozilla community [Kim et al., 2008] was used. This heuristic consists of the selection, as commits fixing an issue, those fitting the following regular expression: $"(b|B)ug.*"$.. This regular expression will filter all of the commit messages which start with the key word "Bug" or "bug".

3. The log message of those random selection of commits was manually inspected to evaluate whether they refer to real bugs, either checking the underlying source code or by parsing the relative Bug Tracking System.

In order to evaluate the precision and recall of such approximation, the constituent parts are as follows:

```
(TP) True positive:  78
(FP) False positive:  7
(TN) True negatives:  6
(FN) False negatives:  9
Total commits:  100
```

Therefore we evaluated:

- Positive predictive value: $TP/(TP + FP) = 78/(78 + 7) = 91,7\%$
- Negative predictive value: $TN/(FN + TN) = 40\%$
- Sensitivity = $TP/(TP + FN) = 78/78 + 9 = 89,65\%$

- Specificity = $TN/(FP + TN) = 6/7 + 6 = 46,15\%$

Since the *Precision* actually coincides with the positive predictive, and the *Recall* coincides with the sensitivity, we conclude that $precision = 91,7\%$ and $recall = 89,65\%$. One further aspect to notice is that out of 100 random commits, 85 have the word "Bug" or "bug" in their title, and 76 out of 100 are actually containing code that deals with a bug. The implications of this initial finding are discussed later.

5.2 Dealing with very large commits

As reported in previous studies, software systems, and most noticeably FLOSS systems, display at times high (and isolated) peaks of activity. In some specific cases, it has been possible to detect a very large amount of source lines (e.g., more of 80% of the overall system) being moved within FLOSS projects [Canfora et al., 2007, Fernández-Ramil et al., 2009, MacLean et al., 2010, Hindle et al., 2008]. This means that in some changes, one can detect huge changes reaching million of lines. From a maintenance or evolutionary point of view, this is hardly accountable as a maintenance activity. However, this problem has not been taken into account by [Kim et al., 2008], whose analysis is one of the pillars for this study.

Also in the study of the *comm-central* repository, it has been found that a small number of commits (no more than 10% of the total set) handles several thousands (in some cases hundreds of thousands) of lines in just one commit. Apart from exceptional cases where developers indeed modified a vast amount of source lines, the peaks could also be caused by automatic bots, changes in the licenses, or by accidental removal and addition of source code. As an example of such distortions, figure 3 shows the number of aggregated number of removed lines[4]. The figure depicts a situation of common removal of lines, but in some specific commits, we can see how suddenly a large set of lines is removed (for example, close to id 723 or 4,200).

In order to deal with such distortions, the commits fully or partially affected by those changes were removed from the sample: given an overall number of 2,912,866 lines and 2,969 commits detected in the bug-fixing process, the sample was therefore reduced to 731,941 lines and 1,747 commits. In summary, the four largest commits (IDs 0; 1,002; 817; 5,213 and 5,383[5]), and the lines affected, were removed from the sample.

5.3 Identifying *bug-fixing* and *bug-seeding* committers

In order to detect the bug-fixing committers, and the developers dealing in the past with the same section of code (as per the scenarios in Figure 1), this paper uses the same assumption formulated in [Kim et al., 2008]: in a bug-fixing commit, one has to consider only the "set of lines" removed or modified in that commit (see Figure 2), instead of the whole file, or set of files, committed in the transaction.

[4]This figure only shows those commits where at least one line was removed.

[5]It should be noticed that the commits listed here are real commits, while the aforementioned, 723 or 4,200 are ids and they do not correspond to real revision numbers in the SCM.

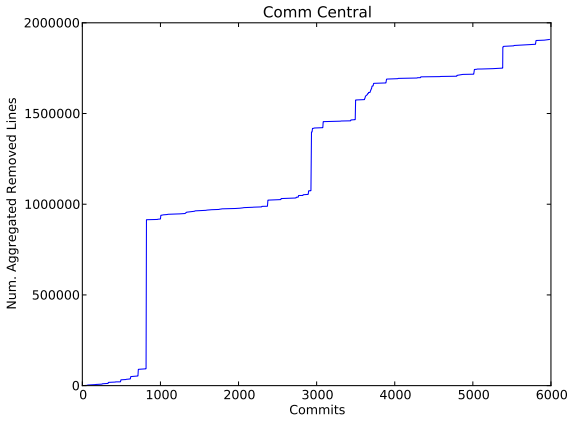

Figure 3: Aggregated number of removed lines detected in bug-fixing commits

This algorithm is named as the "*SZZ* algorithm" and fully detailed in [Śliwerski et al., 2005]: considering the set of lines modified in a bug-fixing commit, the algorithm focuses on the previous commit in time (i.e., "one step back") where all the lines in such set were modified: in this way, it is possible to obtain the latest commit where each line was previously modified (Figure 2), and correlating it with their actual committer (Figure 1). The assumption of the algorithm, also used in this paper, is that the bug was actually introduced in that previous commit.

Using this approach, the total number of developers dealing with either bug-fixing commits or bug-seeding commits were evaluated. Overall, 450 different committers have committed once to the Mercurial repository: of those, 287 are authors of at least one bug-fixing commit, and 383 are authors of at least one bug-seeding commit. This seems to negate that specific developers are dedicated to fix bugs: in addition, it is worth to mention that the Mozilla community has identified the Thunderbird project as "core" project, in which senior developers will peer review the commits made by others. This may distort the dataset used in this paper and open another set of questions, for instance linking those policies with the outcomes of the project.

In order to visualise at first the summary of results, Figure 4 shows the density chart of the bug-seeding developers: since most of the values are located to the left-side of the chart, only 1 or 2 developers are involved in 80% of the cases (1,392 out of 1,747 commits overall). More specifically, 1,035 bug-fixing commits (60% of the overall sample) involve just one developer previously seeding the lines, but only 7% of the total seeded lines (50k out of 747k lines).

Based on this initial set of results, the two scenarios shown in figure 1 were further divided into three more scenarios: one previous developer (covering a 60% of the sample), two previous developers (covering an additional 20% of the sample) and the rest of them (covering the rest of the 20% of the sample). This provides a final list of six scenarios:

S1 – bug-fixing and bug-seeding commits made by committer A;

S2 – bug-fixing commit made by A, bug-seeding commit made by B;

S3 – bug-fixing commit made by A, bug-seeding commit made by A and B only;

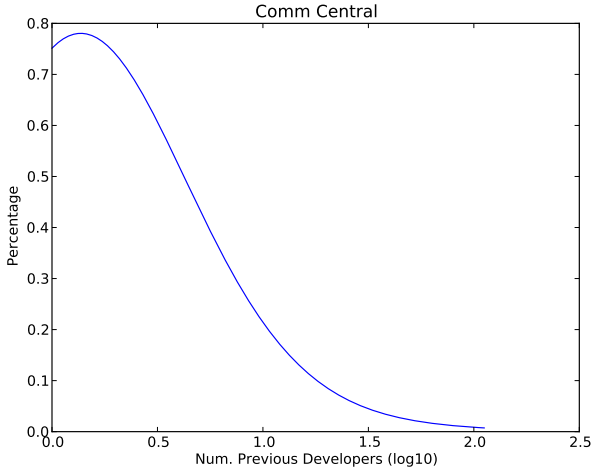

Figure 4: Density chart of number of committers involved in changes at (t-1) to lines bug-fixed at t

	Overall	Same Comm.	Diff. Comm.
Commits	1,035	62	973
Lines	50,078	973	49,348

Table 1: One previous committer

S4 – bug-fixing commit made by A, bug-seeding commit made by B and C only;

S5 – bug-fixing commit made by A, bug-seeding commit made by A, B and others;

S6 – bug-fixing commit made by A, bug-seeding commit made by B, C and others;

5.4 Analysis of Scenarios

Scenarios S1 and S2 Table 1 focuses the analysis on the bug-seeding commits by at most one committer, which correspond for some 60% of all the bug-fixing commits. The summary in the table distinguishes whether the author of the bug-fixing commit is the same committer (*Same Comm.* column, e.g., scenario S1) or a different one (*Diff. Dev.* column, e.g., scenario S2) who seeded those buggy lines. Results show that, in terms of developers involved, the bug-fixing process is performed by different committers from those seedind the bug: for only 6% of these commits (62 out of 1,035) the bug-fixer is the same and the only one involved in the bug-seeding activity (e.g., scenario S1). In all the other cases, a different committer A is involved in the fixing of lines that were seeded by B (e.g., scenario S2).

written down in the method part

	Overall	S3 (A+B)	S4 (B+C)
Commits	357	43	314
Lines	15,581	7,052 (6,834 + 218)	8,529

Table 2: Commits: two previous committers

Scenarios S3 and S4 When considering a maximum of two bug-seeding committers, it was found that only 357 commits comply with the requirements of Scenarios S3 and S4. Table 2 shows the results differentiated for S3 and S4: 43 out of 357 commits were seeded fully or partially by the same committer who finally fixed the bug. In terms of lines handled, 6,834 lines were co-changed with another committer and submitted by the same committer A, while 218 were co-changed with A but committed by another committer B. These results provide another point of view of the community: generally speaking, it seems that most of the commits where two people have previously participated were mostly handled by people different from those who fixed the bug. However, in 43 commits, the same committer was found to participate in the changes. This raises another question related to the quantity of source code handled by other committers than the one who fixed the bug. In that case, we realized that just a 3% of that source code (218 lines) was really handled by someone different: this shows similar results to the S1 and S2 scenarios, where just one committer was found.

As visible in the same table, most of the bug-seeding commits are by other two developers (B+C), but only half of the source code is handled in the process.

Scenarios S5 and S6 The last two scenarios comprise the commits with up to 10 previous committers handling the source code. Table 3 shows the number of committers found for each commit. For instance, for the first row, the values show that there are 27 commits where the same committer fixes and seeds the bug together with others (Scenario 5), while there are 128 commits where that committer did not participate at all (and different people seeded those lines). Albeit more committers could be possible, the threshold of 10 committers reaches 98% of the total sample of commits analyzed (1,717 out of 1,747 commits). Figures 5 (left and right) show the absolute and relative number of commits for the values presented in table 3. In Figure 5 left, the x-axis are divided by the number of previous developers involved in the set of lines that in the current commit were modified or removed. The y-axis represents the number of absolute commits detected. We can see how figure 5 (left) shows that most of the commits were previously handled by people totally different from the ones who were later dealing with the bug-fixing commit. Figure 5 (right) adds extra information in order to check the relative percentages of such values, and to conclude that, in all of the cases, more than a 60% of the total bug-seeding commits had a different committer than the one who made the bug-fixing commit. Using relative numbers, in eight out of ten combinations, the second set of data (commits fixed by A, but not seeded by A) is the most general.

Num. Previous Committers	S5 (A+B+...)	S6 (B+C+...)
3	27 (773 + 3,011)	128 (29,246)
4	11 (85 + 441)	59 (5,010)
5	9 (148 + 696)	24 (1,840)
6	3 (9 + 253)	21 (2,126)
7	3 (30 + 13,089)	12 (2,844)
8	3 (79 + 5,207)	4 (141)
9	5 (30 + 1,328)	8 (3,575)
10	2 (11 + 183)	6 (1,422)

Table 3: Rest of the cases: from 3 to 10 previous committers

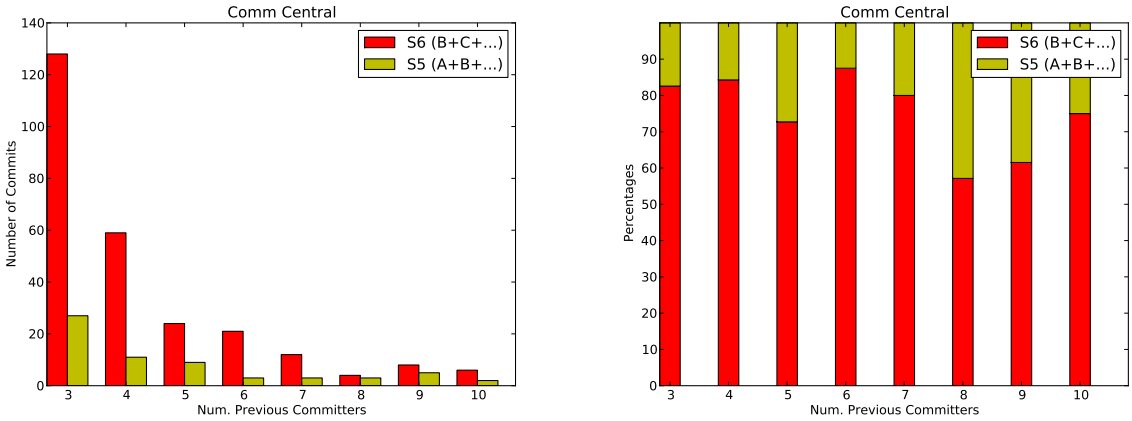

Figure 5: Scenarios and their relevance: S5 refers to those commits where the same committer who is fixing the issue, previously participated. S6 refers to those commits where that committer did not previously participate.

5.5 Finer granularity – Lines affected

In order to study the results at a finer granularity, figure 6 uses the lines to complement the above results. Depending on the number of bug-seeding committers, this figure shows the number of lines seeded in the various scenarios: for each number of previous committers detected (x-axis), the number of seeded lines by the same committer who fixed that bug is shown.

The notation “Same Commmit and Same Committers”, represents the relative number of lines that were also previously handled by the same committer who fixed the issue (Scenarios S1, S3 and S5 - A also previously participated). With the notation ”Same Commit and Diff Committers“ the figure shows the Scenarios S1, S3 and S5, but discarding the lines previously modified by the bug-fixing committer. Finally, the notation ”Rest of them“, is the aggregation of the rest of Scenarios (S2, S4 and S6), where the committer who fixed the bug did not previously participate at all.

As a results from this figure, it can be seen how for all of the cases (except in two previous

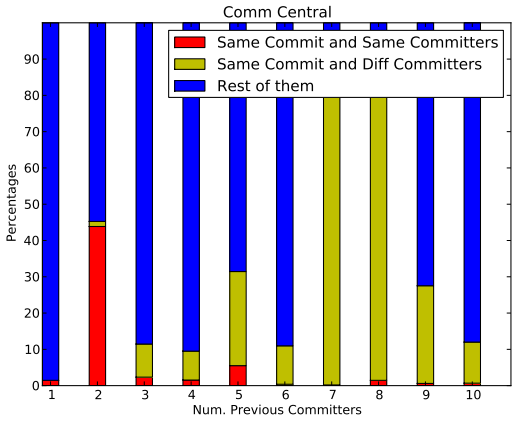

Figure 6: Scenarios and their relevance – Lines affected

developers), the committer who has fixed a bug has not participated at all in seeding that bug.

5.6 Discussion

The analysis of the Mozilla community, and of the *comm-central* project, has shown few interesting insights: given the specificity of this community, and the process that was put in place at the maintenance level, generalising such findings could be problematic. Nonetheless, these observations provide an initial set of results to characterise the bug-fixing and bug-seeding activities in the Mozilla community, to be used as a baseline to be compared against other FLOSS projects.

Overall bug-fixing activity: it has been found that some 50% of all the commits are detected as fixing bugs: considering that the precision of detecting bug-fixing commits proved substantially high, this is an impressive amount. Nonetheless, this value is largely dependent on the policy applied by the Mozilla community when submitting changes to the source code: this policy alone could lead to an overrating or underrating of the results.

Bug-fixing and territoriality: the main result found, that bug-seeding committers are rarely also bug-fixing committers, somehow conflicts with what is found in the FLOSS literature: strong "territorial" developers, and specific responsibilities of the developers over their source code have been observed from previous works [German, 2004], [Capiluppi et al., 2010]. However, it seems that the concepts of "source code territoriality" [6] and "bug-fixing territoriality" are based on different assumptions: for the vast majority, bugs are fixed by other developers than the ones who introduced or seeded such bugs.

Bug-fixing and individual roles: regarding the bug-seeding activity, it was found that each piece of source code modified in a bug-fixing commit is previously modified mostly by one developer. From a system perspective, this reflects the result that many frequency distributions in software are power-law (e.g., many changes are handled only once, and by one developer, while few changes are handled more often and by several developers); from a managerial perspective, this result shows

[6] Pieces of source code (i.e.: methods or files) managed by only one developer.

that developers usually fix bugs that were introduced by other developers. This could either reflect the presence of specific bug-fixing developers, or a more shared activity of bug-fixing, where newcomers tend to fix bugs left open by other developers [Ye et al., 2004]. Furthermore, focusing on Scenarios 3 or 4, there is a 80% of probability that a selected bug-fixing commit was introduced by at most two developers. This again shows that in most of the cases, pieces of source code are by definition a valuable piece of knowledge. Some authors have dealt with the idea of *concepts* when developing software, and it seems that working at the level of methods or functions is the best way to understand previous changes made by others. A possibility here is to match pieces of source code and methods to check this hypothesis.

Bug-seeding and movement of code: at the granularity of commits, Scenarios 5 and 6 have shown that bug-seeding commits were handled by several (even dozens of) committers. A possible explanation could be related to the observed huge movements of lines in bug-fixing commits: it could be found that for a given commit, several people previously participated in such a bug-seeding commit.

6 Threats to Validity

Generally speaking, any empirical study like this is bound to many threats to their validity. It has been claimed that studying FLOSS projects from an empirical point of view could raise several threats that should be considered [Fernandez-Ramil et al., 2008]. Among them, we can find those related to the data extraction, the granularity of the study or how mature is the selected projects.

Construct Validity At first, the heuristic used to obtain bugs from the SCM log messages is far from be perfect. As seen in [Kim et al., 2008], the selection of bugs (even for those projects studied in the Mozilla community) are based in a corpus and some other semantic data which improve the data obtained. Also, as addressed by [Chen et al., 2004], analyzing the SCM logs could be error prone. However, after manually checking 100 commits with the heuristic used, the percentages of error were very low. This is due to the selection of a project from the Mozilla community which generally shows good practices by precisely pointing to the bug tracking system for almost each change in the source code.

Second, most of the work is based on the analysis by the *diff* tool provided by the Mercurial SCM. Although this is a reliable tool, we have detected some limitations in the use of *diff* to retrieve the authorship and other related data. As addressed by [Canfora et al., 2007] and [Zimmermann et al., 2006], we could obtain wrong indications in the number of actual changes in the source code after a commit. One of the main reasons for those changes could be some movements of data from one directory to another, or some merges from different branches. In order to deal with them, most of the big spikes, as aforementioned, were removed.

Finally, it is worth to mention that the large additions of lines are an issue which has not been resolved in this paper. Future revisions following a commit affecting thousands of lines may lead to the wrong conclusions, by showing that most of the work was done by just one committer, although this could be just a distortion of few commits.

Internal Validity The tools and script used could present some minor bugs that may affect the results. However, they have followed a validation process what makes the results reliable enough. After the initial development and after several tests, a final manual study of several commits was carried out and in all of the cases the comparison between the information in the database and the SCM matched in a 100% of the cases. However, the tools used could raise some errors in the future that could not have been taken into account yet .

External Validity The selected project is not large enough to represent the overall number of FLOSS projects. However, we present a first initial step to describe the bug-fixing process based on the Mercurial SCM. As further work, the authors want to extend the analysis at least to the whole Mozilla community.

7 Conclusion and Further Work

This paper has presented an empirical analysis of the *comm-central* FLOSS community, in order to detect whether the bug-fixing activity among developers could be modeled into patterns and recurring scenarios. This community was selected for the consistency and reliability of their messages into the SCM repository, in particular the messages dealing with the bug-fixing activities. With these characteristics, this community and their data can be leveraged to shed important hints on how FLOSS developers proceed to the very needed corrective maintenance, and more importantly, whether the bug-fixing committers are the same who contributed to introduce and seed the bug in the first place.

As a first result, we could confirm the reliability and consistency in referencing the bug-fixing commits within the *comm-central* community, with a precision larger than 90%: this produces very accurate results in terms of tracking the actual bug-fixing committers, and the lines that were modified in the process. It also forms a basis of good practices that will be leveraged in future works when studying the larger Mozilla community (an order of magnitude larger in terms of activity and committers).

Secondly, we proposed a method to define and track both the bug-fixing and the bug-seeding committers: given the set of lines affected by the bug-fixing commit, the set of previous revisions was studied in order to detect which committers were actually "seeding" such bug without contributing to its removal or fixing.

Thirdly, we proposed an approach to avoid the distortion of spurious data: it was observed that the *comm-central* community produces high peaks of activity [MacLean et al., 2010, Canfora et al., 2007, Fernandez-Ramil et al., 2008]. This problem was been raised by [Kim et al., 2008]: what we did to tackle the issue was to remove the five largest commits, which alone were responsible of over 2M lines modified, added or removed. We proposed that researchers should remove at least three main cases: 1- Initial import of commits, 2- Huge removal followed by addition of lines of code, 3- Huge addition followed by removal of lines. In all of those cases, the results could be directly influenced.

Furthermore, we proposed to use the *diff* provided by the SCM as a way to let us know authorship at the granularity of a line: other works such as [Canfora et al., 2007, Zimmermann et al., 2006,

Kim et al., 2008] have used another different approach to deal with the idea of following the life of a line. Several difficulties emerge when trying to track the whole lifecycle of these lines, but not at the level of going a step back in their history. Thus, using this tool could be a faster and more effective way of determining the authorship of each line.

Finally, with respect to the results, it was shown how the corrective maintenance is being carried out by people on the *comm-central* community. We have detected that less than 5% of the bug-fixing commits were handled by who first introduced the changes or "seeded" the bug. With respect to these results, in most of the cases the committers involved in the bug-fixing process are not the same as those initially seeding the bug. These results are vastly different and unexpected if compared with corporate software development, where developers "opening" a bug are most likely to also be responsible for its fixing and closure.

As further work, the authors would like to address two open questions (related to the GQM approach) that could be easily answered using the same dataset. First of all, the central idea of ths paper is related to the fixing process and if the committers are fixing their own bugs. However, we have not studied if those committers are aware that in some cases they have been introducing errors in the source code, or at least the seed of a future bug. Checking how many of them have been working in a given time-window after the detection of a bug could provide another insight of the bug-fixing process.

Another similar idea is related to the seed of the bug. We have seen how given a commit fixing a bug we could trace when the involved source code was previously added or modified and, thus, who was the "bug-seeder". However we do not fully understand the causes. For instance, we could trace if that developer modified a piece of source where she usually does not work, if the commit modified a file that was lately several times modified, if a committer submitted a change in a programming language not usual to her or some other possibilities.

Acknowledgment

The authors would like to thank Prof Cornelia Boldyreff for the extensive comments on the paper. In addition, this work has been partially funded by the European Commission, under the ALERT project (ICT-258098).

References

[Adams et al., 2009] Adams, P. J., Capiluppi, A., and Boldyreff, C. (2009). Coordination and productivity issues in free software: The role of brooks' law. In *ICSM*, pages 319–328. IEEE.

[Canfora et al., 2007] Canfora, G., Cerulo, L., and Penta, M. D. (2007). Identifying changed source code lines from version repositories. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 14, Washington, DC, USA. IEEE Computer Society.

[Capiluppi et al., 2010] Capiluppi, A., Baravalle, A., and Heap, N. (2010). Open standards and e-learning: the role of open source software. In *Proc. of the 6th International Conference on Open Source Systems (OSS 2010)*, Notre Dame, IN, USA.

[Chen et al., 2004] Chen, K., Schach, S. R., Yu, L., Offutt, J., and Heller, G. Z. (2004). Open-source change logs. *Empirical Software Engineering*, 9:197–210.

[Fernández-Ramil et al., 2009] Fernández-Ramil, J., Izquierdo-Cortazar, D., and Mens, T. (2009). What does it take to develop a million lines of open source code? In *OSS*, pages 170–184.

[Fernandez-Ramil et al., 2008] Fernandez-Ramil, J., Lozano, A., Wermelinger, M., and Capiluppi, A. (2008). Empirical studies of open source evolution. In Mens, T. and Demeyer, S., editors, *Software Evolution: State-of-the-art and research advances*, chapter 11, pages 263–288. Springer Verlag.

[German, 2004] German, D. M. (2004). Using software trails to reconstruct the evolution of software: Research articles. *J. Softw. Maint. Evol.*, 16(6):367–384.

[Goldman and Gabriel, 2004] Goldman, R. and Gabriel, R. (2004). *Innovation Happens Elsewhere: How and Why a Company Should Participate in Open Source*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Guo et al., 2010] Guo, P. J., Zimmermann, T., Nagappan, N., and Murphy, B. (2010). Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *ICSE (1)*, pages 495–504.

[Hao-Yun Huang and Panchal, 2010] Hao-Yun Huang, Q. L. and Panchal, J. H. (2010). Analysis of the structure and evolution of an open-source community. In *Proceedings of the ASME 2010 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference IDETC/CIE 2010*.

[Hindle et al., 2008] Hindle, A., German, D. M., and Holt, R. (2008). What do large commits tell us?: a taxonomical study of large commits. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 99–108, New York, NY, USA. ACM.

[Izquierdo-Cortazar et al., 2009] Izquierdo-Cortazar, D., Robles, G., Ortega, F., and Gonzalez-Barahona, J. M. (2009). Using software archaeology to measure knowledge loss in software projects due to developer turnover. *Hawaii International Conference on System Sciences*, 0:1–10.

[Kagdi et al., 2008] Kagdi, H. H., Hammad, M., and Maletic, J. I. (2008). Who can help me with this source code change? In *ICSM*, pages 157–166.

[Kim et al., 2008] Kim, S., E. James Whitehead, J., and Zhang, Y. (2008). Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196.

[Koch, 2009] Koch, S. (2009). Exploring the effects of sourceforge.net coordination and communication tools on the efficiency of open source projects using data envelopment analysis. *Empirical Softw. Engg.*, 14(4):397–417.

[Ma et al., 2009] Ma, D., Schuler, D., Zimmermann, T., and Sillito, J. (2009). Expert recommendation with usage expertise. In *ICSM*, pages 535–538.

[MacKenzie et al., 2002] MacKenzie, D., Eggert, P., and Stallman, R. (2002). *Comparing and Merging Files with GNU diff and patch*. Network Theory Ltd.

[MacLean et al., 2010] MacLean, A. C., Pratt, L. J., Krein, J. L., and Knutson, C. D. (2010). Trends that affect temporal analysis using sourceforge data. In *Proceedings of the 5th International Workshop on Public Data about Software Development (WoPDaSD '10)*, page 6.

[Miller and Myers, 1985] Miller, W. and Myers, E. W. (1985). A file comparison program. *Software - Practice and Experience*, 15(11):1025–1040.

[Mockus et al., 2002] Mockus, A., Fielding, R. T., and Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346.

[Myers, 1986] Myers, E. W. (1986). An o(nd) difference algorithm and its variations. *Algorithmica*, 1:251–266.

[Pan et al., 2009] Pan, K., Kim, S., and Whitehead, Jr., E. J. (2009). Toward an understanding of bug fix patterns. *Empirical Softw. Engg.*, 14(3):286–315.

[Robles and González-Barahona, 2006] Robles, G. and González-Barahona, J. M. (2006). Contributor turnover in libre software projects. In Damiani, E., Fitzgerald, B., Scacchi, W., Scotto, M., and Succi, G., editors, *OSS*, volume 203 of *IFIP*, pages 273–286. Springer.

[Robles et al., 2006] Robles, G., González-Barahona, J. M., and Guervós, J. J. M. (2006). Beyond source code: The importance of other artifacts in software development (a case study). *Journal of Systems and Software*, 79(9):1233–1248.

[Sliwerski et al., 2005] Sliwerski, J., Zimmermann, T., and Zeller, A. (2005). When do changes induce fixes? In *MSR*.

[Śliwerski et al., 2005] Śliwerski, J., Zimmermann, T., and Zeller, A. (2005). When do changes induce fixes? In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA. ACM.

[Ukkonen, 1985] Ukkonen, E. (1985). Algorithms for approximate string matching. *Inf. Control*, 64(1-3):100–118.

[Ye et al., 2004] Ye, Y., Nakakoji, K., Yamamoto, Y., and Kishida, K. (2004). The co-evolution of systems and communities in Free and Open Source software development. In Koch, S., editor, *Free/Open Source Software Development*, pages 59–82. Idea Group Publishing, Hershey, Pennsylvania, USA.

[Zimmermann et al., 2006] Zimmermann, T., Kim, S., Zeller, A., and Whitehead, Jr., E. J. (2006). Mining version archives for co-changed lines. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 72–75, New York, NY, USA. ACM.