Information and Software Technology 65 (2015) 114–128

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Understanding the triaging and fixing processes of long lived bugs

Ripon K. Saha [*], Sarfraz Khurshid, Dewayne E. Perry

Center for Advanced Research in Software Engineering (ARiSE), Department of Electrical and Computer Engineering, The University of Texas at Austin, USA

ARTICLE INFO

Article history:
Received 24 June 2014
Received in revised form 24 February 2015
Accepted 1 March 2015
Available online 11 March 2015

Keywords:
Bug tracking
Bug triaging
Bug survival time

ABSTRACT

Context: Bug fixing is an integral part of software development and maintenance. A large number of bugs often indicate poor software quality, since buggy behavior not only causes failures that may be costly but also has a detrimental effect on the user's overall experience with the software product. The impact of *long lived* bugs can be even more critical since experiencing the same bug version after version can be particularly frustrating for user. While there are many studies that investigate factors affecting bug fixing time for entire bug repositories, to the best of our knowledge, none of these studies investigates the extent and reasons of long lived bugs.
Objective: In this paper, we investigate the triaging and fixing processes of long lived bugs so that we can identify the reasons for delay and improve the overall bug fixing process.
Methodology: We mine the bug repositories of popular open source projects, and analyze long lived bugs from five different perspectives: their proportion, severity, assignment, reasons, as well as the nature of fixes.
Results: Our study on seven open-source projects shows that there are a considerable number of long lived bugs in each system and over 90% of them adversely affect the user's experience. The reasons for these long lived bugs are diverse including long assignment time, not understanding their importance in advance, etc. However, many bug-fixes were delayed without any specific reasons. Furthermore, 40% of long lived bugs need only small fixes.
Conclusion: Our overall results suggest that a significant number of long lived bugs may be minimized through careful triaging and prioritization if developers could predict their severity, change effort, and change impact in advance. We believe our results will help both developers and researchers better to understand factors behind delays, improve the overall bug fixing process, and investigate analytical approaches for prioritizing bugs based on bug severity as well as expected bug fixing effort.

1. Introduction

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

To ensure high software quality for each release, developers/managers triage bugs carefully and schedule the bug fixing tasks based on their severity and priority. Despite such a rigorous process, there are still many bugs that live for a long time. We believe the impact of these *long lived* bugs (for our study, bugs that are not fixed within one year after they are reported) is even more critical since the users may experience the same failures version after version. Therefore, it is important to understand the extent and reasons of these long lived bugs so that we can improve software quality.

A number of previous studies have investigated the overall factors affecting bug fix time. Giger et al. [8] empirically investigated the relationships between bug report attributes and the time to fix. Zhang et al. [31] predicted overall bug fix time in commercial projects. Canfora et al. [6] used survival analysis to determine the relationship between the risk of not fixing a bug within a given time frame and specific code constructs changed when fixing the bug. Zhang et al. [30] examined factors affecting bug fixing time along three dimensions: bug reports, source code involved in the fix, and code changes that are required to fix the bug.

While these studies are useful in understanding the overall factors related to bug fix time, we know of no study that has specifically investigated long lived bugs to understand why they take such a long time to be fixed and how important they are. We point out

* Corresponding author.
E-mail addresses: ripon@utexas.edu (R.K. Saha), khurshid@ece.utexas.edu (S. Khurshid), perry@ece.utexas.edu (D.E. Perry).

http://dx.doi.org/10.1016/j.infsof.2015.03.002
0950-5849/© 2015 Elsevier B.V. All rights reserved.

that analyzing entire bug datasets using various machine learning or data mining techniques (as done in previous work) is not sufficient in understanding long lived bugs due to the imbalanced dataset.[1] Imbalanced datasets are a major problem in most data mining applications since machine learning algorithms can be biased towards the majority class due to over-prevalence [12]. We expect (and our results also support) that the proportion of long-lived bugs would be lot less than 50% of the total bugs, thus resulting an imbalanced dataset. Therefore, if we automatically analyze all the bug reports using a standard data mining technique, it is highly likely that the main factors behind long lived bugs will get lost. In this paper, we conduct an exploratory study focused solely on long lived bugs to understand their extent and reasons with respect to following research questions:

1. **What proportion of the bugs are long lived?** The answer to this question is important since if there are few long lived bugs, there may be little reason to worry.
2. **How important are long lived bugs in terms of severity?** It is important to understand how crucial these bugs were from the perspective of both developers and users. If they are minor or trivial bugs, their impact would be less on overall software quality.
3. **Where is most of the time spent in the bug fixing process?** The answer to this question is important to identify the time consuming phases so that developers as well as researchers can work on improving the process involving this phase.
4. **What are common reasons for long lived bugs?** To improve the bug fixing process, first we need to understand the underlying reasons for delays. Delineating the common reasons for long lived bugs will help researchers deal with the problem more systematically.
5. **What is the nature of long lived bug fixes?** The answer to this question will help us in better understanding the bug fixing process, estimating change efforts, and so on, which will be useful in exploring potential approaches for improving overall bug fixing processes.

We study seven open source projects: JDT, CDT, PDE, and Platform from the Eclipse product family, written in Java,[2] and the Linux Kernel, WineHQ, and GDB, written in C. Our key observations are summarized below:

1. Despite advances in software development and maintenance processes, there are a significant number of bugs in each project that survive for more than one year.
2. More than 90% of long lived bugs affect users' normal working experiences and thus are important to fix. Moreover, there are duplicate bug reports for these long lived bugs, which indicate the users' demand for fixing them.
3. The average bug assignment time of these bugs was more than one year despite the availability of a number of automatic bug assignment tools that could have been used. The bug fix time after the assignment was another year on average.
4. The reasons for long lived bugs are diverse. While problem complexity, reproducibility, and not understanding the importance of some of the bugs in advance are the common reasons, we observed there are many bug-fixes that got delayed without any specific reason.
5. Unlike previous studies [30], we found that a bug surviving for a year or more does not necessarily mean that it requires a large fix. We found that 40% of long-lived bug fixes involved only a few changes in only one file.

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

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

This paper extends our previous "long lived bugs" paper presented at CSMR-WCRE 2014 [22] in three directions. First, we perform the same set of experiments on three additional popular projects: the Linux Kernel, WineHQ, and GDB, which are not only written in different programming language but are also from different domains than our previous subject systems. Second, we provide more details for our previous results. Finally, our new results show that our previous findings hold even for software projects from different domains and written in different languages. Thus this paper reports more generalizable results. We believe these findings will play an important role in developing new approaches for bug triaging as well as improving the overall bug fixing process.

2. Background

In this section, we provide the necessary background for our study that includes a brief description of bug tracking systems and a typical bug life cycle.

2.1. Bug tracking system

Generally project stakeholders maintain a bug database for tracking all the bugs associated with their projects. There are several online bug tracking systems available such as Bugzilla, JIRA, and Mantis. These systems enable developers/managers to manage bug databases for their projects. Different repositories may have different data structures and follow different life cycles of bugs. The dataset we created and used in our work was extracted from Bugzilla, a popular online bug tracking system. Therefore, the rest of the discussion in this paper regarding the bug tracking system is limited to Bugzilla.

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

Developers in a particular project can define their own severity level. According to Eclipse Bugzilla documentation, the severity level can be one of the following values, which represent the degree of potential harm.[3]

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

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

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

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

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

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

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

In addition to providing severity level, reporters also specify the software version, the platform and operating system where they encountered the bug so that developers can productively attempt to reproduce it. Bug reporters also can attach files to the bug report such as screen shots, and failing test cases. Once a bug is posted, all

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