Characterizing Verification of Bug Fixes in Two Open Source IDEs

Rodrigo Souza and Christina Chavez
Software Engineering Labs
Department of Computer Science - IM
Federal University of Bahia (UFBA), Brazil
{*rodrigo,flach*}*@dcc.ufba.br*

***Abstract*—Data from bug repositories have been used to enable inquiries about software product and process quality. Unfortunately, such repositories often contain inaccurate, inconsistent, or missing data, which can originate misleading results. In this paper, we investigate the reported status of bug reports from the projects Eclipse and Netbeans, in special the VERIFIED status, corresponding to the verification that a bug fix is appropriate. We show that the VERIFIED status is used in a variety of distinct ways, not always consistent with the traditional definition of software verification. Future research should take into account the multiple contexts in which the VERIFIED status is used when deriving conclusions based on the data from bug reports.**

***Keywords*-mining software repositories; bug tracking systems; software verification; empirical study.**

I. Introduction

Bug repositories (or bug tracking systems) have for a long time been used in software projects to support coordination among stakeholders. Such systems record discussion and progress of software evolution activities, such as corrective, perfective, and ... changes [**?**]. Hence, bug repositories are an opportunity to researchers who intend to investigate issues related to the quality of the product and of the process of software development team.

The information contained in bug repositories has been used in order to predict fault-proneness, to unveil developers' roles from data, to characterize transfer of work in software projects, and so on. Also, it has been used to investigate beliefs in software engineering related to software quality, such as the impact of developer turnover and of collective ownership on the occurrence of bugs.

However, mining bug repositories has its own risks. Previous research has identified problems of missing data (e.g., rationale, traceability links between reported bug fixes and source code changes) and inaccurate data (e.g., misclassification of bugs) [1], [2], [3], [4].

In previous research [cite report], we tried to assess the impact of independent verification of bug fixes on software quality, by mining data from bug repositories. We relied on reported verifications tasks, as recorded in bug reports, and interpreted the recorded data according to the documentation for the specific bug tracking system used. As the partial results suggested that verification has no impact on software quality, we questioned the accuracy of the data about verification of bug fixes, and thus decided to investigate how verification is actually performed and reported on specific software projects.

In particular, we are interested in answer the following exploratory research questions for a limited set of projects:

- **How** is the verification performed: are there performed ad hoc tests, automated tests, code inspection?
- **Who** performs the verification: is there a dedicated team for QA?
- **When** is the verification performed: is it performed just after the fix, or is there a verification phase?

II. Background

Bug tracking systems allow users and developers of a software project to manage a list of bugs for the project. Usually, users and developers can report bugs, along with information such as steps to reproduce the bug, its severity and the operating system used. Developers choose bugs to fix and can report on the progress of the bug fixing activities, ask for clarification, discuss causes for the bug etc.

In this research, we focus on Bugzilla, an open source bug tracking system used by notable software projects such as Eclipse, Mozilla, Linux Kernel, Open Office, Apache, and companies such as NASA and Facebook[1]. The general concepts from Bugzilla should apply to most other bug tracking systems.

One important feature of a bug that is recorded on bug tracking systems is its status. The status records the progress of the bug fixing activity, and, as such, provides data to software engineering studies. Figure 1 shows each status that can be recorded in Bugzilla, along with typical transitions between status values, i.e., the workflow.

In the simplest cases, a bug is created and receive the status `UNCONFIRMED` (if it was created by a regular user) or `NEW` (if it was created by a developer). Next, it is ASSIGNED to a developer, and then it is RESOLVED, possibly by fixing it with a patch on the source code. The solution is then VERIFIED by someone in the quality assurance team. If it passes the quality requirements, it is CLOSED when the next release of the software is released.

[1]Complete list available at http://www.bugzilla.org/installation-list/.

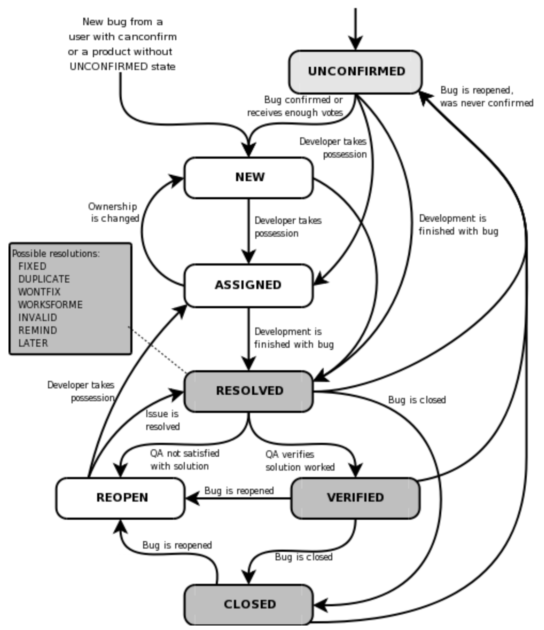
New bug from a user with canconfirm or a product without UNCONFIRMED state
UNCONFIRMED
Bug confirmed or receives enough votes
Bug is reopened, was never confirmed
Developer takes possession
NEW
Ownership is changed
Developer takes possession
Development is finished with bug
Possible resolutions:
FIXED
DUPLICATE
WONTFIX
WORKSFORME
INVALID
REMIND
LATER
ASSIGNED
Development is finished with bug
Developer takes possession
RESOLVED
Bug is closed
Issue is resolved
QA not satisfied with solution
QA verifies solution worked
REOPEN
Bug is reopened
VERIFIED
Bug is reopened
Bug is closed
CLOSED

Figure 1. Workflow for Bugzilla. Source: http://www.bugzilla.org/docs/2.18/html/lifecycle.html.

If it does not pass the quality requirements or if the solution only partially fixes the problem, the bug is REOPENED.

TODO: Roles: reporter, fixer, verifier, reopener, closer.

TODO: bugs with status VERIFIED are used to write the release notes / change log.

In order to understand the bug fixing process of a software team, however, it should be possible to know what activities are performed upon change of status. What do developers do before marking a bug as VERIFIED? Bugzilla documentation states that, when a bug is VERIFIED, it means that "QA [quality assurance team] has looked at the bug and the resolution and agrees that the appropriate resolution has been taken"[2]. Again, the definition is broad, although it assumes the existence of a QA team.

The developer documentation for both Eclipse and NetBeans are unclear about how the verification of a bug fix is performed. So, what forms of software verification are they used in the verification process of these projects?

Software verification techniques are classified in static and dynamic [cite Sommervile]. Static techniques include source code inspection, automated static analysis, and formal verification. Dynamic techniques, or testing, involve executing the software system under certain conditions and comparing its actual behavior with the intended behavior. Testing can be done in an improvised way (ad hoc testing), or can be compiled in a list of test cases, which can then be automated (e.g., as unit tests).

While mining bug repositories, one cannot assume that the VERIFIED status comprises all forms of software verification. Also, one cannot rely on the information provided by the bug tracking system or the developer documentation for a project, since it can be too generic to be useful. Such information can only be assessed by taking a closer look on actual bug reports.

III. Method

In order to answer the research questions—when and how bug fixes are verified, and who verifies them—, a three-part method was used:

1) **Data extraction**: we have obtained publicly available raw data from the Bugzilla repositories of two popular integrated development environments, Eclipse and NetBeans.
2) **Data sampling**: for each project, two representative subprojects were chosen for analysis.
3) **Data analysis**: for each research question, a distinct analysis was required. The analyses will be further described.

A. Data Extraction

In order to perform the desired analyses, we needed access to the data recorded by Bugzilla for a specific project, including status changes and comments. We have found such data for two projects—Eclipse and NetBeans—from the domain of integrated development environments. The data were made available as part of the Mining Software Repositories 2011 Challenge[3] in the form of MySQL database dumps. The files contain all data from the respective databases, except for developer profiles, for privacy reasons.

Eclipse development began in late 1998 with IBM[4]. It was licensed as open source in November, 2001. The available data set contains 316,911 bug reports for its 155 subprojects, from October, 2001 to June, 2010.

NetBeans[5] started as a student project in 1996. It was bought by Sun Microsystems in October, 1999, and it was open sourced in March, 2000. The data set contains 185,578 bug reports for its 39 subprojects, from June, 1998 to June, 2010.

B. Data Sampling

Four subprojects were chosen for further analysis: Eclipse/Platform, Eclipse/EMF, Netbeans/Platform, and Netbeans/VersionControl. The Platform subprojects are the main subprojects for the respective IDEs, so they are both important and representative of each projects' philosophy.

The other two subprojects were chosen at random, restricted to subprojects in which the proportion of verified bugs was greater than the proportion observed in the

[2]https://landfill.bugzilla.org/bugzilla-3.6-branch/page.cgi?id=fields.html

[3]http://www.msrconf.org/msr-challenge.html

[4]http://www.ibm.com/developerworks/rational/library/nov05/cernosek/

[5]http://netbeans.org/about/history.html

respective Platform subprojects. This is to avoid selecting projects in which bugs are seldom marked as VERIFIED. The proportion of VERIFIED bugs per project is as such: Eclipse/Platform: 16.0%; Eclipse/EMF: 48.4%; NetBeans/Platform: 21.4%; NetBeans/VersionControl: 29.7%.

C. Analysis: How Are Bugs Verified?

In order to discover the verification techniques used by the subprojects, we have looked at the comments written by developers when they mark a bug as VERIFIED (meaning that the fix was accepted) or REOPENED (meaning that the fix was rejected). First, an informal analysis was performed, by reading a sample of such messages, looking for textual patterns that indicate that a particular technique was used. Then, based on the informal analysis and on previous knowledge, we have built a regular expression for each technique, that should match messages in which there is evidence that that particular technique was used. The regular expressions were refined by looking at the results matched by them and modifying them to avoid false positives. Finally, we used the regular expressions to find relevant messages.

In this paper, we just list some substrings that the regular expression matches. The actual regular expressions are available at http://....

- automated testing: testcase, test case, unit test, automated test, test result, run the test, test run;
- source code inspection: inspecting, inspection, look at the code, looking at the source;
- ad hoc testing: verified by ...ing;
- automated static analysis: findbugs, checkstyle, static code, static analys.
- formal testing: formal test, format verif

The classification script is smart enough to ignore negative assertions, such as “did not run the test”.

The regular expression for ad hoc testing is the most restrictive of them, because ad hoc testing was found to be reported in many distinct ways (when it is reported). We expect to find fewer examples of ad hoc testing than it should be found.

D. Analysis: When Are Bugs Verified?

In order to determine if there is a well-defined verification phase for the subprojects, we have selected all reported verifications (i.e., status changes to VERIFIED) over the lifetime of each subproject. Then, we have plotted, for each day in the interval, the accumulated number of verifications reported since the beginning. The curve is monotonically increasing, with steeper climbs representing periods of intense verification activity.

Also, we have obtained the release dates for multiple versions of Eclipse and NetBeans. The information was obtained in the respective websites. In cases in which older information was not available, archived versions of the web pages were accessed via the website www.archive.org.

If a subproject presents a well-defined verification phase, it is expected that the verification activity is more intense a few days before a release. Such pattern can be identified by visual inspection of the graph, by looking for steeper climbs in the verification curve preceding the release dates.

E. Analysis: Who Verifies Bugs?

In order to determine whether there is a quality assurance (QA) team, we have counted how many times each developer has marked a bug as FIXED or VERIFIED. We considered that a developer is part of a QA team if s/he verified at least 10 times more than s/he fixed bugs. Also, we have computed the proportion of verifications for the subproject that was performed by the discovered QA team. It is expected that, if the discovered set of developer is actually a QA team, they should be responsible for the majority of the verifications.

IV. Results and Discussion

Each of the three analysis was performed on Eclipse/Platform, Eclipse/EMF, NetBeans/Platform, and NetBeans/VersionControl. The results are presented.

A. How Are Bugs Verified?

Most verification messages do not provide any clue about the verification technique used. Many messages just state that the bug fix was verified, optionally informing the build used to verify the fix.

In Eclipse/Platform, though, a significant number of verification messages include the technique used, e.g., “verified by code inspection”, or “verified with test case”. Based on such observations, we have built a search term for inspections and another one for unit tests. Then, we have classified the technique described in each verification message using the search term.

In Eclipse/Platform, there are 13,483 verifications, each one with a message. By the search criteria, 494 were identified as code inspections, 118 were identified as tests and 74 were identified as ad hoc.

Evidence of both code inspection and automated testing was found in all projects, although the evidence was weaker.

We also noted that, in Eclipse/Platform, often the fixer requests someone else to verify the fix, by asking “please verify, [developer name]”. If the bug is reopened, then the fixer and the verifier exchange roles. That is considered good practice.

It is difficult to identify ad hoc cases, because they are so pervasive that they are not reported, and also they are reported in many different ways.

Conclusion: it’s hard to determine the technique used by just looking at the data. Eclipse/Platform documents it better, sometimes.

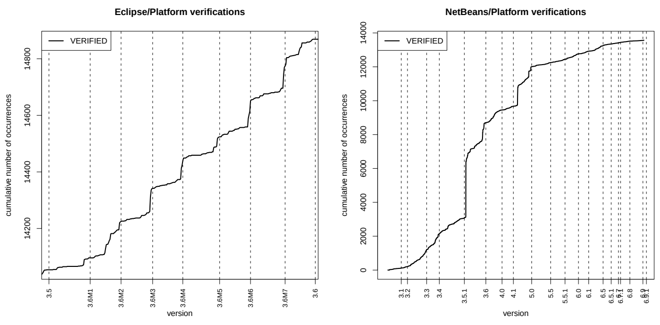

Figure 2. Number of verifications over time.

B. When Are Bugs Verified?

In order to answer whether there is a verification phase for the projects analyzed, we plotted the cumulative number of verifications along the lifetime of each project, as shown in Figure XXX, in red. The vertical lines represent release dates for each version. As a basis for comparison, the number of bug fixes is also plotted, in black.

We found evidence of a verification phase for Eclipse/Platform. In this project, verifications appear to be performed in bursts, with periods of intense activity separated by about 6 to 7 weeks of low or no activity. By taking a closer look in the period between the releases 3.5 and 3.6 (Figure XXX), it is clear that the periods of intense verification activity occur just before the milestones leading to 3.6.

No such pattern was found for Netbeans' subprojects.

The first thing that comes to attention are "cliff walls", points in time when a large amount of bugs are verified, represented by steep, almost vertical portions of the plotted line. They appear in Eclipse/EMF and Netbeans/Platform. In Netbeans/Platform, all verifications in the cliff wall preceding version 5.5 share the same message: "Verification of old issues". For Eclipse/EMF, the message is "Move to verified as per bug 206558", which refers to the task of changing bugs marked as FIXED to VERIFIED.

Cliff walls represent a change in the development process, and the mass verification is needed in order to leverage useful features from Bugzilla (e.g., looking for verified bugs in order to aid the writing of release notes).

Because of cliff walls, extra care must be taken before doing any analysis that relies on the VERIFIED status. In such situations, the VERIFIED status is applied blindly, with no software verification being actually performed. Because cliff walls contain a large amount of bugs, they are likely to bias the results of analyses.

C. Who Verifies Bugs?

Table I presents, for each subproject, the number of developers attributed to a QA team, and the proportion of bug verifications they account for.

Project	**QA team size**	**% of verifications by QA team**
Eclipse/Platform	4 (2.4%)	1.1%
Eclipse/EMF	0 (0.0%)	0.0%
NetBeans/Platform	5 (21.7%)	93.3%
NetBeans/VersionControl	5 (20.8%)	93.2%

Table I
DISCOVERED QA TEAM FOR ALL FOUR SUBPROJECTS.

In NetBeans it is possible to infer the existence of a QA team, composed by approximately 20% of the developers, that performs about 90% of all verifications.

In the Eclipse subprojects, there is no evidence of a dedicated QA team. In Eclipse/EMF no developer focus specifically on verification. In Eclipse/Platform, 2.4% of the developers focus on verification tasks, contributing only to 1.1% of the verifications.

V. CONCLUSION

The VERIFIED status is used in a variety of distinct ways, not always consistent with the traditional definition of software verification ...

Summary of the findings:

- It is difficult to detect the verification technique used in each reported verification.
- Beware of cliff walls!
- Know your project.

With the knowledge obtained from this exploratory research, we can improve and extend our previous work on the impact of independent verification on software quality. Examples of refined questions:

- Does independent verification improve software quality when it is performed by developers from the QA team?
- Does independent verification improve software quality when it is performed as an separate phase of the software release cycle?
- Does independent verification improve software quality when a particular verification technique (e.g., inspections, unit testing) is used?

ACKNOWLEDGMENT

The first author is supported by FAPESB under grant BOL0119/2010.

REFERENCES

[1] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 298–308. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070530

[2] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, “Fair and balanced?: bias in bug-fix datasets,” in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC/FSE ’09. New York, NY, USA: ACM, 2009, pp. 121–130. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595716

[3] T. H. D. Nguyen, B. Adams, and A. E. Hassan, “A case study of bias in bug-fix datasets,” in *WCRE*, G. Antoniol, M. Pinzger, and E. J. Chikofsky, Eds. IEEE Computer Society, 2010, pp. 259–268.

[4] K. Ayari, P. Meshkinfam, G. Antoniol, and M. Di Penta, “Threats on building models from cvs and bugzilla repositories: the mozilla case study,” in *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, ser. CASCON ’07. New York, NY, USA: ACM, 2007, pp. 215–228. [Online]. Available: http://doi.acm.org/10.1145/1321211.1321234