

# Characterizing Verification of Bug Fixes in Two Open Source IDEs

Rodrigo Souza and Christina Chavez  
Software Engineering Labs  
Department of Computer Science - IM  
Universidade Federal da 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 how well data from bug repositories support the discovery of details about the software verification process in two open source projects, Eclipse and NetBeans. We have been able to identify quality assurance teams in NetBeans and to detect a well-defined verification phase in Eclipse. A major challenge, however, was to identify the verification techniques used in the projects. Moreover, we found cases in which a large batch of bug fixes is simultaneously reported to be verified, although no software verification was actually done. Such mass verifications, if not acknowledged, threatens analyses that rely on information about software verification reported on bug repositories. Therefore, we recommend that the exploratory analyses presented in this paper precede inferences based on reported verifications.

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

## I. INTRODUCTION

Bug repositories have for a long time been used in software projects to support coordination among stakeholders. They record discussion and progress of software evolution activities, such as bug fixing and software verification. Hence, bug repositories are an opportunity for researchers who intend to investigate issues related to the quality of both the product and the process of a software development team.

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), inaccurate data (e.g., misclassification of bugs) [1], and biased data [2].

In previous research [3], 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 the projects that were analyzed, Eclipse and NetBeans.

Hence, in this paper, we investigate the following exploratory research questions regarding the software verification process in Eclipse and NetBeans:

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

The next section contains some background information about bug tracking systems and software verification. In Section III, the data and methods used to investigate the research questions are presented. Then, in Section IV, the results are exposed and discussed. Finally, Section V presents some concluding remarks.

## II. BACKGROUND

Bug tracking systems allow users and developers of a software project to manage a list of bugs for the project, along with information such as steps to reproduce the bug and the operating system used. Developers choose bugs to fix and 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 software projects such as Eclipse, Mozilla, Linux Kernel, NetBeans, Apache, and companies such as NASA and Facebook<sup>1</sup>. 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. Figure 1 shows each status that can be recorded in Bugzilla, along with typical transitions between status values, i.e., the workflow.

In simple cases, a bug is created and receive the status UNCONFIRMED (when created by a regular user) or NEW (when 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 is adequate, or otherwise it is REOPENED. When a version of the software is released, all VERIFIED bugs are CLOSED.

<sup>1</sup>Complete list available at <http://www.bugzilla.org/installation-list/>.

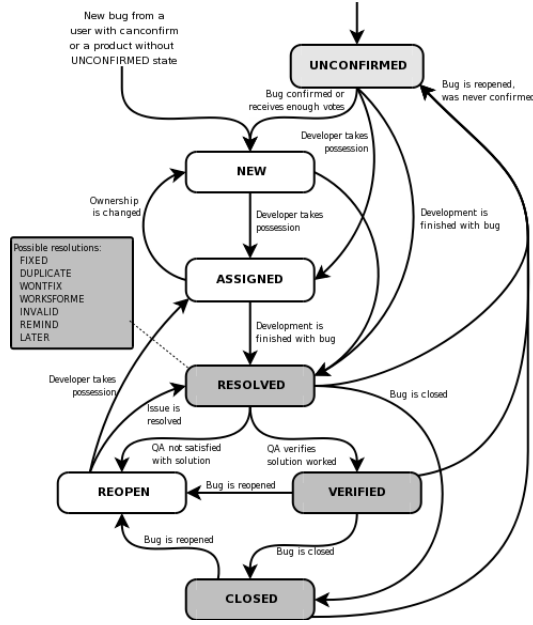


Figure 1. Workflow for Bugzilla. Source: <http://www.bugzilla.org/docs/2.18/html/lifecycle.html>.

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”<sup>2</sup>. It does not specify how developers should look at the resolution (e.g., by looking at the code, or by running the patched software).

Software verification techniques are classified in static and dynamic [4]. 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 it can be structured as a list of test cases, leading to *automated testing*.

### 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 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, as will be further described.

The experimental package is available at <https://sites.google.com/site/rodrigogs2/msr2012>

<sup>2</sup><https://landfill.bugzilla.org/bugzilla-3.6-branch/page.cgi?id=fields.html>

#### 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 was made available as part of the Mining Software Repositories 2011 Challenge<sup>3</sup> in the form of MySQL database dumps. The files contain all data from the respective databases, except for developer profiles, omitted for privacy reasons.

Eclipse development began in late 1998 with IBM<sup>4</sup>. 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<sup>5</sup> started as a student project in 1996. It was then bought by Sun Microsystems in October, 1999, and 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 respective Platform subprojects. The reason is to avoid selecting projects in which bugs are seldom marked as VERIFIED. The following proportions of VERIFIED bugs per project were observed: Eclipse/Platform: 16.0%; Eclipse/EMF: 48.4%; NetBeans/Platform: 21.4%; NetBeans/VersionControl: 29.7%.

#### C. 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 first day available in the data. The curve is monotonically increasing, with steeper ascents representing periods of intense verification activity.

Also, we have obtained the release dates for multiple versions of Eclipse and NetBeans. The information was obtained from 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](http://www.archive.org).

If a subproject presents a well-defined verification phase, it is expected that the verification activity is more intense a

<sup>3</sup><http://2011.msrfconf.org/msr-challenge.html>

<sup>4</sup><http://www.ibm.com/developerworks/rational/library/nov05/cernosek/>

<sup>5</sup><http://netbeans.org/about/history.html>

few days before a release. Such pattern can be identified by visual inspection of the graph, by looking for steeper ascents in the verification curve preceding the release dates.

#### D. Analysis: Who Verifies Bugs?

In order to determine whether there is a team dedicated to quality assurance (QA), 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 (i.e., one order of magnitude) more than s/he fixed bugs. Also, we have computed the proportion of verifications that was performed by the discovered QA team. It is expected that, if the discovered set of developers is actually a QA team, they should be responsible for the majority of the verifications.

#### E. 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 comments, looking for textual patterns that indicate that a particular technique was used. Then, based on the informal analysis and on previous knowledge, a regular expression was built for each technique, to match comments in which there is evidence that a particular technique was used. The regular expressions were refined by looking at the results matched by them and modifying them to avoid false positives.

The complete regular expressions are available in the experimental package. For the sake of simplicity, the following list describes only some substrings matched by the regular expressions for each technique:

- 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 running, verified by [word ending in “ing”];
- automated static analysis: findbugs, checkstyle, static code, static analys;
- formal testing: formal test, formal verif.

During the informal analysis, it was observed that ad hoc testing is reported in many distinct ways that are difficult to capture by means of regular expressions. Hence, its regular expression is expected to match only a few comments.

## 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 next.

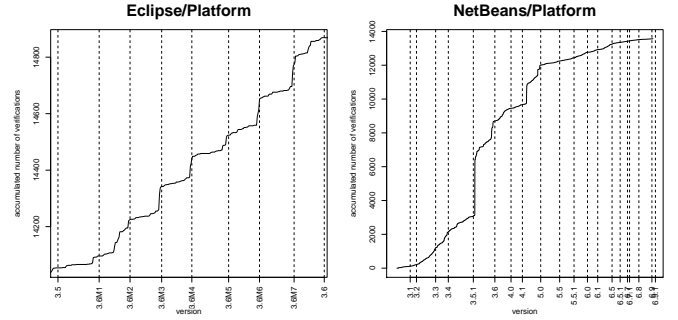


Figure 2. Number of verifications over time.

#### A. When Are Bugs Verified?

Figure 2 shows plots of total accumulated number of verifications over time for Eclipse/Platform (left) and NetBeans/Platform (right). Releases are plotted as dashed vertical lines.

For Eclipse/Platform, the graph shows the period between releases 3.5 and 3.6, including milestone releases every 6 or 7 weeks. It is clear from the graph that the verification activity is intensified in a few days preceding a milestone, represented in the graph by steeper ascents before the vertical lines. This pattern is an indicator of a verification phase. No such pattern was found in the other subprojects by analyzing their graphs (not shown here for brevity).

The graph for NetBeans/Platform (Figure 2, right side) shows the entire project history. Although there are steeper ascents, they are different because they do not precede release dates. Also, at a closer look, they represent thousands of verifications performed in a few minutes by the same developer, with the same comment. The same pattern was found in Eclipse/EMF (not shown).

Of course, no developer can verify so many fixes in so little time. The explanation, supported by the comments, is that such mass verifications represent some cleanup of the bug repository, by marking old bugs as `VERIFIED`—with no verification being actually performed. Researchers should take extra care with mass verifications, as they contain a large amount of bugs and, thus, are likely to bias the results of analyses.

In the next two analyses (who and how), mass verifications were discarded. A verification was considered to be part of a mass verification if the developer who performed it also performed at least other 49 verifications in the same day. Although further research is needed to evaluate such criterion, it was able to identify the most obvious mass verification cases.

After applying the criterion to identify mass verifications, 362 (2.4%) verifications were discarded from Eclipse/Platform, 2348 (72.3%) verifications were discarded from Eclipse/EMF, and 5336 (39.3%) verifications were

Table I  
DISCOVERED QA TEAM FOR ALL FOUR SUBPROJECTS.

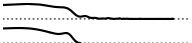

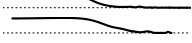
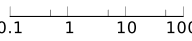
Project	QA team size	% of verifications by QA team	Distribution of ratio (bugs verified / bugs fixed) per developer
<b>Eclipse</b>			
Platform	4 (2.4%)	1.1%	
EMF	0 (0.0%)	0.0%	
<b>NetBeans</b>			
Platform	25 (18.8%)	80.1%	
V.Control	5 (20.8%)	93.2%	

Table II  
VERIFICATION TECHNIQUES FOR ALL FOUR SUBPROJECTS.

Project	Testing	Inspection	Ad Hoc
Eclipse/Platform	250 (1.1%)	511 (2.2%)	67 (0.3%)
Eclipse/EMF	21 (1.6%)	1 (0.1%)	0
NetBeans/Platform	88 (0.8%)	5 (0.0%)	0
NetBeans/VersionControl	4 (0.1%)	1 (0.0%)	0

discarded from NetBeans/Platform. Mass verifications were not identified in NetBeans/VersionControl.

### B. Who Verifies Bugs?

Table I presents, for each subproject, the number of developers attributed to a QA team (i.e., developers who perform verifications at least 10 times more often than contribute bug fixes), and the proportion of bug verifications they account for. Mass verifications and developers who have not contributed with fixes and verifications were discarded from the analysis.

In both NetBeans subprojects it is possible to infer the existence of a QA team, composed by approximately 20% of the developers, performing at least 80% 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.

Although the reported results are based on an arbitrary threshold (10) for the ratio between verifications and fixes, other values (2 and 5) were also used, leading to similar results. The rightmost column of Table I shows the distribution of the ratio, in log scale, where it can be seen that ratios above 10 are uncommon in all projects.

### C. How Are Bugs Verified?

Table II shows, for each subproject, the number of comments associated with a bug being marked as `VERIFIED` or `REOPENED` that refers to a particular verification technique, as matched by the respective regular expressions. Comments associated with mass verifications were discarded.

No references to formal verification were found. Regarding static analysis, only 4 references were found; upon inspection, it was found that only one reference (bug report 15242 for NetBeans/Platform) implies the use of a static analysis tool in the verification process.

In all subprojects, comments suggest the use of automated testing and code inspection in the verification process, the former technique being more frequently referenced (except in Eclipse/Platform, in which code inspection is cited more often). Evidences of ad hoc testing were found only in Eclipse/Platform, probably due to limitations in the regular expressions.

The regular expressions were able to identify the verification technique only in 3.6% of the comments at best (Eclipse/Platform). Such low proportion can be explained partly by limitations of the regular expression method (which can be addressed in future work) and partly by lack of information in the comments themselves.

By sorting through comments, we have found that many of them do not provide any clue about the verification technique used. Most often, developers just state that the bug fix was verified, sometimes informing the build used to verify the fix.

In Eclipse/Platform, comments show that the developer who fixes a bug often asks 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. This behavior illustrates a structured bug fixing/verification process.

In Eclipse/EMF, we found that marking a bug as `VERIFIED` does not mean that the bug fix was verified. Instead, it means that the fix was made available in a build of the software that is published in the subproject’s website<sup>6</sup>.

## V. CONCLUSION

By analyzing four subprojects (two from Eclipse, two from NetBeans), we have found, using only data from bug repositories, subprojects with and without QA teams, with and without a well-defined verification phase. We also have found weaker evidence of the application of automated testing and source code inspection. Also, there were cases in which marking a bug as `VERIFIED` did not imply that any kind of software verification was actually performed.

With the knowledge obtained from this exploratory research, we aim to improve and extend our previous work on the impact of independent verification on software quality. We can investigate, for example, whether verification performed by QA team is more effective than verification performed by other developers.

Researchers should be aware that information about verification techniques may not be common in bug repositories, and that reported verification does not always correspond to actual verification. Some exploration of the data is important to avoid such pitfalls.

## ACKNOWLEDGMENT

This work is supported by FAPESB (grant BOL0119/2010) and CNPq/INES (grant 573964/2008-4).

<sup>6</sup>See [http://wiki.eclipse.org/Modeling\\_PMC\\_Meeting,\\_2007-10-16](http://wiki.eclipse.org/Modeling_PMC_Meeting,_2007-10-16)

## REFERENCES

- [1] J. Aranda and G. Venolia, “The secret life of bugs: Going past the errors and omissions in software repositories,” in *Proc. of the 31st Int. Conf. on Soft. Engineering*, 2009, pp. 298–308.
- [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 *European Soft. Eng. Conf. and Symposium on the Foundations of Soft. Eng.*, ser. ESEC/FSE '09. ACM, 2009.
- [3] R. Souza and C. Chavez, “Impact of the four eyes principle on bug reopening,” Univers. Federal da Bahia, Tech. Rep., 2011.
- [4] I. Sommerville, *Software engineering (5th ed.)*. Addison Wesley Longman Publish. Co., Inc., 1995.