Quantitative analysis of performance in key parameter of code review - Defects Individuation.

Dorealda Dalipaj[1]

University Rey Juan Carlos, Madrid. dorealda.dalipaj@urjc.es

Abstract **-** Finding and removing defects close to their point of injection remains the main motivation and a key parameter to code review. However, different studies argued that as many software programs rely on bug reports to correct software errors in maintenance activities, developers spend much time to identify these bug reports, the time taken to carry out the process of code review is long and the performance of code review in finding errors was less than expected.

The aim of this study is to bring empirical evidence on the time spent by developers to identify the bugs reports, the time spent by them to carry out the reviewing process and to bring evidence by quantifying the performance of the very key parameter of code review: individuation of bugs.

With the abundance of data and having a diverse set of projects to observe, our case study is focused on the code review process of the cloud computing software, Open Stack. Our focus is to bring answers to what extent developers are actually individuating and fixing bugs while they are applying changes in the code finding thus if there is any code review providing more value than the others.

The identification of the bugs report time, review time and the number of bugs fixed during this process, are the most fundamental parameters for characterization of the performance of the code review process and the most important metrics having a positive increasing relation with the benefits of code review.

1 Introduction

Code review, sometimes referred as peer review, employed both in industrial and open source contexts, is an activity in which people, other than the author of a software deliverable, examine it for defects and improvement opportunities. Code review is characterized *as a systematic approach to examine a product in detail, using a predefined sequence of steps to determine if the product is fit for its intended use* [8].

There have been different ways of performing defect detenction since its beginning up to nowadays. The formal review or inspection according to Fanagan [9] approach required the conduction of an inspection meeting for actually

finding of defects. Different controlled experiments showed that there were no significant differeneces in the total number of defects found when comparing meeting-based with meeting-less-based inspections [10], [11]. Other studies [12] were carried out and proved that more defects were identified with meeting-less-based approaches. As a result a wide range of mechanisms and techniques of code review were developed. From static analysis [15] [16] [17], which examines the code in the absence of input data and without running the code and is tool based, to modern code review [18] [19] [20], which aligning with the distributed nature of many projects is asynchronous and frequently supporting geographically distributed reviewers. Because of their many uses and benefits, code reviews are a standard part of the modern software engineering workflow.

Although it is generally accepted that quality in software remains a challenge due to defects presence. A major quality issue with software is that defects are a byproduct of the complex development process and the ability to develop defect free software remains a big challenge for the software community. It is possible to improve the quality of software product by injecting fewer defects or by identifying and removing defects injected. It is also generally accepted that performance of software review is affected by several factors of the defect detection process. Code review performance is associated with the effort spent to carry out the process and the number of defects found.

Most empirical studies try to assess the impact of specific process settings on performance. Sources of process variability range from structure (how steps of the inspection are organized), inspection inputs (reviewer ability and product quality), techniques applied to defect identification that define how each step is carried out), context and tool support [13]. A controlled experiment by Johnson and Tjahjono [10] showed that *total defects identified, effort spent in the process, false positive defects, duplicates* are fundamental variables to analyse when controlling the performance of code review.

2 Discussion

Although code review is used in software engineering primarily for finding defects, several studies argue that they often do not find functionality defects.

Over the past two years, a common tool for code review, Code Flow, has achieved wide-spread adoption at Microsoft. The functionality of CodeFlow is similar to other review tools such Mondrian [18] (adopted at Google), Phabricator [19] (adopted at Facebook) or open-source Gerrit [20]. Two studies where conducted at Microsoft, with Code Flow as case study, on code review process.

The first study ([2]) took place with professional developers, testers, and managers at Microsoft. The results show that, although the top motivation driving code reviews is finding defects, the practice and the actual outcomes are less about finding errors than expected: Defect related comments comprise a small proportion, only 14%, and mainly cover small logical low-level issues. The second study conducted at Microsoft [3] stated that code review do not find

bugs. They found that only about 15% of the comments provided by reviewers indicate a possible defect, much less functionallity issues that should block a code submission.

Another empirical study on the effectiveness of security code review [1], conducted an experiment on 30 developers. They conducted manual code review of a small web application. The web application supplied to the developers had seven known vulnerabilities. Their findings concluded that none of the subjects found all confirmed vulnerabilities (they were able to find only 5 out of 7) and that reports of false vulnerabilities were significantly correlated with reports of valid vulnerabilities.

A different experiment [4] argued that in large scale software programs where bug tracking systems are used,due to the excessive number of duplicate bug reports, developers spend much time to identify these bug reports.

Keeping in mind the above discussion we did these questions:

RQ 1. What amount of time do developers need to identify bug reports?

RQ 2. What is the actual amount of time that developers take in carrying out the review process?

RQ 3. Are all the code review tools performing with a low number of defects detection?

With the abundance of data coming from the engineering systems and having a diverse set of projects to observe [6] [7], we ask if there is any code review providing more value than the others?

To provide answer for the above question we perform a large empirical study on the actual 213 active projects of Open Stack. For the purpose of our study we analyse them devided by the 9 projects that are the core of Open Stack (see 3), and group the rest of them as Other Projects category. We choose Open Stack as it is a large project that have adopted code reviews on a large scale and have a reasonable traceability between commits, review and defects reports. It uses Launchpad, a bugtracking system for tracking the issue reports, and Gerrit, a lightweight code review tool. Additionally, being an open source cloud computing software, it is backed by a global collaboration of developers. It has other flavors worthy of additional benefits which influences the outcome, and which can bring a different picture from the one in [1], [2], [3] and [4].

In the remainder of this paper, we first describe the necessary background notions for our work (section 3). Next, we describe the case study setup (section 4), then present the results of the research questions (section 5). After discussion and threats to validity (section 6 and section 7), we discuss future work (section 8) and we finish with conclusions (section 9).

3 Background

This section provides background information about the bugtracking and code review environments of Open Stack and the tools for obtaining data from their repositories.

Openstack is a free and open source set of software tools for building and managing cloud computing platforms.

OpenStack is made up of many different moving parts. Because of its open nature, anyone can add additional components to OpenStack to help it to meet their needs. This is whay actually in Open Stack there are 213 active projects. But the OpenStack community has collaboratively identified 9 key components that are a part of the "core" of OpenStack, which are distributed as a part of any OpenStack system and officially maintained by the OpenStack community: Nova, Swift, Cinder, Neutron, Horizon, Keystone, Glance, Ceilometer, and Heat. Thus in this study we analyse the 9 core components of Open Stack and categorise the rest as Other Projects.

Open Stack uses Launchpad, an issue tracking system, which is a repository used to enable users and developers to report defects and feature requests. It allows such a reported issue to be triaged and (if deemed important) assigned to team members, to discuss the issue with any interested team member and to track the history of all work on the issue. During these issue discussions, team members can ask questions, share their opinions and help other team members. Open Stack uses a dedicated reviewing enivornment, Gerrit, to review patches and bug fixes. It supports with lightweight processes for reviewing code changes, i.e., to decide whether a developers change is safe to integrate into the official Version Control System. During such lightweight code review, assigned reviewers make comments on a code change or ask questions that can lead to a discussion of the change and/or different revisions of the code change, before a final decision is made about the code change. If accepted, the most recent revision of the code change can enter the version control system, otherwise the change is abandoned and the developer will move on to something else.

In this paper, we are interested in quantifying the time that developers need to go through the code review, by measuring the time to merge a change, and to measure how many bugs (and possibly of what type) where fixed in the code changes that successfully passed in the VCS.

To obtain the issue reports and code review data of these ecosystems, we used the data set provided by Gonzalez-Barahona et al. [21]. They developed the MetricsGrimoire tool suite to mine the repositories of OpenStack, then store the corresponding data into a relational database. We make use of their issue report and code review data sets [22] to perform our study.

4 Case Study Setup

This section explains the methodology used to address our research questions. We discuss the selection of the case study system, how we identified and the extraction of bug reports, and the results that we obtained.

4.1 Selection of Case Study System

The aim of our study, to begin with, is to measure the time that the developers need to identify the bug reports in the bugtracking system, the time they need to carry out the review, and provide quantitative evidence on these two metrics and on the number of bugs detected during the process.

Our case study system choice falls on Open Stack, because for achieving our aims, we required projects with a substantial number of commits linked to issue reports and code review. And this is readily done in Open Stack.

Furthermore, thanks to MetricsGrimoire tool, we can mine the repositories of Launchpad and Gerrit systematically updated.

What we need to do is to identify the issue reports classified as bugs, link them the respective review and then extract the patterns we need for carring out our results.

4.2 Identifying classified Bug Reports

In Launchpad, besides bugs reports, the developers can work with *specifications* (approved design specifications for additions and changes to the project teams code repositories) and *blueprints* (lightweight feature specifications).

Identify which of the reports have been classified as describing bugs is not a trivial task. Tickets usually are commented. Reviewers do discuss about having found a bug in a certain report or not. But we saw that analysing comments of a ticket is not the most efficient way for extracting its classification, because we wil not recover 100% of the tickets.

Manually analysing the tickets and how the Launchpad work flow treats them, we came accross a pattern of the reports states with regards to confirming new bugs:

1. a ticket, stating a possible bug, is opened in launchpad and its status is New,
2. if the problem stated in the ticket is reproduced than the bug is confirmed as genuine and the ticket status passes from New to Confirmed,
3. only when a bug is confirmed, the status then changes from Confirmed to In Progress and an issue will be opened in gerrit.

Thus, we automaticlly analysed the Launchpad repository searching for tickets that have a match with this pattern. These are the tickets that have been classified as bug reports. Once we identify we extract them in a new repository for further inspection.

Our results showed that 57720 tickets out of 88421 have been classified as bugs (you can review this results in a python notebook [1] that we make avail-

[1] github.com/ddalipaj/CR_Defect_Individuation_Rate/blob/master/finding_bugs.ipynb

abe online). Hence 65.3% of the total tickets in Launchpad are bugs, and an issue for fixing has been opened for them in Gerrit. At this point we were able to quantify the time that developers spend on identifying bug reports as the distance between the moment when the ticket is first inserted in Launchpad up to the moment it is confirmed as a genuine bug.

The percentages of reported issues (tickets) classified as bugs in OpenStack, grouped by projects and last the percentage on the total number of tickets.

Project	Total Number Tickets	Bug Classified Tickets	Percentage of Bug Tickets
Nova	13018	8073	63%
Swift	1681	974	58%
Cinder	4064	2691	66%
Horizon	5349	3526	66%
Keystone	4101	2517	61%
Glance	2895	1826	63%
Ceilometer	1880	1302	69%
Heat	3125	2218	71%
Other Projects	53245	34593	65%
Total OS	88421	57720	65%

Fig. 1. The percentages of reported bugs in OS - From July, 2010 - January, 2016.

4.3 Linking the Issue Reports to the Reviews

The next step is to link the tickets that we have already extracted with their respective issue in the code review system. To detect the links between ticket and reviews, we first referred to the name of the branch on which a code change had been made, since some of them follow the naming convention "bug/989868" with "989868" a ticket identifier. After the extraction, we manually analysed a random number of issues and their respective tickets. We discovered that some issues were matched to a ticket like a related artifact, while indeed the issue was reviewing the ticket and merging not in the master branch of the project under which the defect was originated, but in another branch. For quatifying the time that developers do need to carry out the review process, we must take into consideration only the merges into the master branch of the project, the merges in the versions of the same project for the preservation of compatibility are not elements of this metric. Thus this selection was clearly erroneus. We tried the other way around approach. Instead of linking reviews to tickets, we linked tickets to reviews using the information that we find in the comments of the tickets. Whenever a review recieves a proposal for a fix, or a merge for a fix, it is reported in the comments of the respective ticket. Precisely, a merge comment looks like the following:

Reviewed: https://review.openstack.org/100018
Committed: https://git.openstack.org/cgit/openstack/nova/commit
/?id=be58dd8432a8d12484f5553d79a02e720e2c0435
Submitter: Jenkins
Branch: master ...

Where the first line, clearly, provide us with the link to the issue in Gerrit.

The first problem that arises in analysing the comments is that, some of them for some ticket, are a summary of some commit history. Therefore, in these cases, we find more than a match with the pattern we are looking for within the body of the comment, while the commit itself is not a merge in the master branch of the project that orginated the ticket, consequently not the correct result.

However there is a fixed format of the comments that report a merge, which is the one in the example above. The information related to the review is stated at the very begining of the comment and manually analysing the tickets in Launchpad we have seen that they are found in the first 6 rows.

Thus the first step is trunking of the comments, so that we extract just the first 6 lines from every one of them. Doing this assures the identifying of the right review. We can view the steps of this process in a python notebook [2] that is available online. The table below shows the number of tickets and issues that we were able to link with its counterpart.

Number of distinct issues merged	**Number of distinct merged linked**	**Number of distinct fixes merged**	**Number of distinct fixes linked**
44799	37080	211207	42200
Percentage of distinct tickets linked to its review	**82,8%**	**Percentage of distinct reviews linked to the tickets**	**20%**
Number of all tickets merged	**Number of all tickets linked**	**Number of all fixes merged**	**Number of all fixes linked**
70587	63739	211207	63739
Percentage of all tickets linked to its review	**90,2%**	**Percentage of all reviews linked to the tickets**	**30,2%**

Fig. 2. The percentages of tickets and issues linked with its counterparts in OS - From July, 2010 - January, 2016.

At this point we are able to carry out the time to review in OpenStack as the distance between the first patch is uploaded to Gerrit up to when a fix change is merged to the code base.

[2] https://github.com/ddalipaj/Analysis_Tickets_Issues/blob/master/master_merge.ipynb

5 Case Study Preliminary Results

RQ 1. What amount of time do developers need to identify bug reports?

We computed the time to identify bug reports as discussed in 4.2. Then, we calculated the median effect size across all Open Stack projects in order to globally rank the metrics from most extreme effect size, and last the quantiles. The results are shown in the table below:

Median time for classifying a bug report:		1.96 hours	< 1 days	
Quantiles:				
0.25	273.0 seconds	4.6 minutes		
0.50	7059.5 seconds	117.7 minutes	1.96 hours	
0.75	257626.0 seconds	4293.8 minutes	71.56 hours	2.9 days

Fig. 3. The median time to classify a bug report accross all projects in Open Stack - From July, 2010 - January, 2016.

RQ 2. What is the actual amount of time that developers take in carrying out the review process?

We computed the time to carry out the review process as discussed in 4.3. Again, we calculated the median accross all Open Stack projects. The results are shown in the table below:

Median time for closing a review:	52.17 hours	< 3 days	
Quantiles:			
0.25	29587.3 seconds	8.21 hours	< 1 days
0.50	187816.0 seconds	52.17 hours	< 3 days
0.75	769489.5 seconds	213.75 hours	< 9 days

Fig. 4. The median time to classify a bug report accross all projects in Open Stack - From July, 2010 - January, 2016.

RQ 3. Are all the code review tools performing with a low number of defects detection?

This is a preliminary step in the phd. The third research query is the topic of the next future, thus work in progress. We can state that currently we are working with the elements of the review process that we dispose, that are the comments and the commit analysing both the human discussion and the changes in the code.

6 Threats to Validity

Threats to internal validity concern confounding factors that might influence the results. There are likely unknown factors that impact defect-detection that

we have not measured yet. Due to the elaborate filtering that we performed in order to link two repositories (bug repository, and code review), the heuristics used to find the links between the repositories are not 100% accurate, however we used the state-of-the-practice linking algorithms at our disposal. Recent features in Gerrit show that clean traceability between version control and review repositories is now within reach of each project, hence the available data for future of this study will only grow in volume.

7 Future Work

Our immediatly future work is to quantify the rate at which bugs are discovered during the code review process. In our study we focus on the time for identifying bug reports, for carrying out the review process and the number of defects identified during review. But there are several other metrics that influences and characterize the performance of code review process that we would like to investigate.

8 Conclusion

In this paper we empririically studied the impact of the time for identifying the bug reports, the time to carry out the review process and are conducting a study on quantifying the number of bugs fixed during a code review. From the preliminary results that we bring into evidence and the future results that we hope to have, we believe that our study will open up a variety of research opportunities to continue investigating the impact of collaborative characteristics on performance assurance.

9 Aknowledgement

This study is funded under SENECA EID project, funded under Marie-Skodowska Curie Actions.

The preliminary results of this study can be found in three python notebook available online:
1. https://github.com/ddalipaj/CR_Defects_Individuation_Rate/blob/master/finding_bugs.ipynb
2. https://github.com/ddalipaj/Analysis_Tickets_Issues/blob/master/master_merge.ipynb
3. https://github.com/ddalipaj/Reviewing_Time_Gerrit/blob/master/reviewing_time_Gerrit.ipynb

References

1. Edmundson, A., Holtkamp, B., Rivera, E., Finifter, M., Mettler, A., Wagner, D. (2013). An empirical study on the effectiveness of security code review. In Engineering Secure Software and Systems (pp. 197-212). Springer Berlin Heidelberg.
2. Bacchelli Alberto, and Christian Bird. "Expectations, outcomes, and challenges of modern code review." Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, 2013.
3. Czerwonka Jacek, Michaela Greiler and Jack Tilford. "Code Reviews Do Not Find Bugs."
4. Tao Zhang; Byungjeong Lee "A Bug Rule Based Technique with Feedback for Classifying Bug Reports", Computer and Information Technology (CIT), 2011 IEEE 11th International Conference on, On page(s): 336 - 343
5. IaaS cloud computing of Rackspace Cloud and NASA: https://it.wikipedia.org/wiki/OpenStack and https://www.openstack.org
6. Bug Tracking for OpenStack. https://it.wikipedia.org/wiki/Launchpad
7. Gerrit Code Review for OpenStack. //review.openstack.org/Documentation/intro-quick.html
8. D. L. Parnas and M. Lawford. Inspection's role in software quality assurance. In Software, IEEE, vol. 20, 2003.
9. M. E. Fagan. Design and Code inspections to reduce errors in program development. In IBM Systems Journal 15 pp. 182-211, 1976.
10. P. M. Johnson, and D. Tjahjono. Does Every Inspection Really Need a Meeting? In Empirical Software Engineering, vol. 3, no. 1, pp. 9-35, 1998.
11. P. McCarthy, A. Porter, H. Siy et al. An experiment to assess cost-benefits of inspection meetings and their alternatives: a pilot study. In Proceedings of the 3rd International Symposium on Software Metrics: From Measurement to Empirical Results, 1996.
12. A. Porter, H. Siy, C. A. Toman et al. An experiment to assess the cost-benefits of code inspections in large scale software development. In SIGSOFT Softw. Eng. Notes, vol. 20, no. 4, pp. 92-103, 1995.
13. D. E. Perry, A. Porter, M. W. Wade. Reducing inspection interval in large-scale software development. In Software Engineering, I EEE Transactions on, vol. 28, no. 7, pp. 695-705, 2002.
14. Broy M (2002) Ayewah, N., Hovemeyer, D., Morgenthaler, J. D., Penix, J., Pugh, W. (2008). Using static analysis to find bugs. Software, IEEE, 25(5), 22-29.
15. W. R. Bush, J. D. Pincus, D. J. Sielaff. A static analyzer for finding dynamic programming errors, Softw. Pract. Exper. , vol. 30, no. 7, pp. 775802, 2000.
16. Hallem, D. Park, and D. Engler, Uprooting software defects at the source, Queue, vol. 1, no. 8, pp. 6471, 2003.
17. B. Chess and J. West, Secure Programming with Static Analysis, 1st ed. Addison-Wesley Professional, Jul. 2007.
18. N. Kennedy. How google does web-based code reviews with mondrian. http://www.test.org/doe/, Dec. 2006.
19. A. Tsotsis. Meet phabricator, the witty code review tool built inside facebook. http://techcrunch.com/2011/08/07/oh-what-noble-scribe-hath- penned-these-words/, Aug. 2006.
20. Gerrit code review - https://www.gerritcodereview.com/

21. J. M. Gonzalez-Barahona, G. Robles, and D. Izquierdo-Cortazar. The metrics-grimoire database collection. In 12th Working Conference on Mining Software Repositories (MSR) , pages 478481, May 2015
22. http://activity.openstack.org/dash/browser/data/db/