From intention to code review - the valuable metrics. The case study of a cloud infrastructure.

Dorealda Dalipaj
Universidad Rey Juan Carlos
LibreSoft
Madrid, Spain
dorealda.dalipaj@urjc.es

Jesus M. Gonzales Barahona
Universidad Rey Juan Carlos
GSyC/LibreSoft
Madrid, Spain
jgb@gsyc.es

***Abstract*—The purpose of code review is to find and fix the source code defects that can cause vulnerabilities and thus give rise to an exploit. The code review process encloses a collection of metrics which can be used to improve the software development process. Benefits from those metrics varies from measuring the progress of a development team to investigating into software development policies and guidelines.**
From this excellent source of metrics we highlight *bug triaging* and *time to review*. These metrics do not implicate subjective relations. They are only material facts. As such, they can be recorded to trace the outcome of the code review process.
We have analysed those two metrics along the line of the development process of a large cloud operating system, OpenStack. Finally we related our results with those of previous studies and draw different conclusions.
We believe that the monitoring and management of both these metrics can produce direct benefit on the effectiveness of the code review process, by being able to identify which development practices provide more value than others.

***Keywords*-code, review, process, metrics, software, engineering.**

I. Introduction

Software development process is characterized by a sequence of different activities, with the intent of better planning and management. Some of the activities depend on technical factors, while others are merely organizational and administering issue dependent.
In one of the phases of software development, at some point, the implementation of features starts. Consequently, when during the testing phase errors are found, developers proceed to triage the bugs. Finally, depending on the result of this activity, a review of that chunk of code containing that bug may start.
This process ends when the code with the bug fixed runs into the code base of the project, and whoever might be waiting takes dependency on the new code.
In this paper we take into consideration and analyse exactly the span of time that extends from finding the errors up to when the new code, with the errors fixed, runs into the code base. We will call this span the time from **I**ntention to **C**ode **R**eview and will refer to it as **ICR** in the remaining of the paper.
The ICR process encloses a collection of metrics which can be used to improve the software development process. These software metrics can be grouped in two distinct classes: relative and absolute. While relative metrics are a representation of an attribute which can not be directly measured and rely on the context where the metric was derived, absolute metrics are numerical values which describe a trait of the code that do not involve subjective context but are only material fact.
With former code inspection, many of this metrics could not be measured, but today, with the modern systems, we can trace the information needed to analyse them.
The final purpose of this study is to bring into focus two absolute metrics from ICR process: *time spent for bug triaging* (**BTT**) and *time spend to review the code* (**CRT**).
We are going to quantify the duration of the ICR process and measure both BTT and CRT metrics of the projects we selected as our case study. These measures will conceptually help us to proportion the size that BTT and CRT have on ICR, consequently the relations between the metrics and the process.
Our proposal is that the two metrics, BTT and CRT, are very important for different reasons. First, through them we can better characterise the ICR process. Second and most important, if supervising these two metrics, we believe that the code review process can significantly improve.
With the abundance of data and having a diverse set of projects to observe, our selected case study is OpenStack, the cloud computing infrastructure. We are performing a large empirical study on its more than 200 active projects.
Actually considered the future of cloud, our choice is OpenStack because it is a large project that has adopted code reviews on a large scale. Using modern software systems, it has a reasonable traceability of information of its software development process.
OpenStack uses Launchpad, a bugtracking system for tracking the issue reports, and Gerrit, a lightweight code review tool. Additionally, being an open source project, it is backed by a global collaboration of developers.
Besides OpenStack, code review is applied on a varied array of software projects. These projects have highly different settings, incentive structures and schemes, time pressures, and cultures.
In an effort to characterize and understand these differences,

previous studies ([1], [2], [3]) have examined open and non open source projects like Android OS, Chromium OS, Bing, Office, MS SQL, and projects internal to AMD.
They analysed how time to carry out the review process varies in these different projects. Some conclusions found that the practice of code review is slowing down the code integration activity. They found that in some cases not only the bug triaging is taking too long, but also that the median time for code review is days long, if not weeks.
In line with the previous studies, we compared our findings on CRT with the results obtained for this metric from these studies. Looking at the results, our main assumption is that if some of the code review metrics in different projects have become similar as the projects have normally advanced, then such metrics may be indicative of practices that may represent commonly successful and efficient methods of review.
As such, these can be prescriptive to other projects taking into consideration to add code review to their development process.

In the remainder of this paper, we first describe some previous work on the topic (section II) and the background of the case study (section III). Next we detail the methodology (section IV). After describing the analysis (Section V) and displaying results (section VI), we discuss the conclusion and future work (section VII).

II. Previous Work

The purpose of code review is to find and fix the source code defects that can cause vulnerabilities and thus give rise to an exploit. 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"* [4].
The formal review or inspection according to Fagan's approach [5], introduced 40 years ago, required the conduction of an inspection meeting for in fact finding defects. There have been different ways of performing defect detection during these 40 years, from meeting based to meetingless based inspections.
Several experiments proved that no significant differences were found in the outcome between meeting and meetingless based inspections of the review process ([6], [7]). Another experiment [8] proved that time for conducting the review process significantly improved with meetingless based approaches.
As a result, a considerate number of mechanisms and techniques were developed to code review, which results nowadays in the modern and ligh-weight code review process. From the tool based static analysis ([9], [10], [11]) which examines the code in the absence of input data and without running the code, to the widely adopted modern code review ([12], [13], [14]) which, aligning with the distributed nature of many projects, is asynchronous and frequently supporting geographically distributed reviewers.
Today, as a consequence of the many benefits, code review is a standard process of the modern software engineering workflow. It is generally accepted that the performance of code review is associated with the effort spent to carry it out and is influenced by a range of factors, some of which are external to the technical aspects of the process [15].
In the practical guide to code review, Wiegers denotes that *in formal inspection process the similarities outweigh their differences* [16]. Furthermore, in their study on convergent practices to modern code review, Rigby and Bird compared parameters of review, such as review interval and the number of comments in review discussions. They also found that *while there were some minor divergences in contemporary practice, the findings converged* [1], bringing them to conclude that even in modern code review the *similarities outweigh their differences*.

III. Case Study Background

This section provides background information about OpenStack and the tools used for obtaining data from its repositories.
OpenStack is an open source set of software tools for building and managing cloud computing platforms. Its community has collaboratively identified 9 key components as the core part of OpenStack. Additional components are add to OpenStack from partners with the purpose of helping them to meet their needs. This is why, actually, in OpenStack there are more than 200 active projects.
OpenStack uses Launchpad as its issue tracking system. It is a repository that enables users and developers to report *defects* and *feature* requests.
Once a *defect* (or issue) is reported, developers move to triage it. If it is deemed important, the issue is assigned to team members. Launchpad records the discussions that happen with any interested team member and 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.
Features are also registered in Launchpad. First features are splitted in a list of working items called *specifications* and then implemented. But Launchpad is missing some functions that would make iterating on specifications design and approval usable, like ability to discuss or iterate on several revisions of a specification, or record multiple approvals. Thus some projects decided to experiment with using a specific git repository (**-specs*) to propose, discuss, iterate and track approvals on specifications using Gerrit instead of Launchpad. Those projects still ultimately use Launchpad once the specification is approved, to track the implementation status of the approved feature.
OpenStack uses a dedicated reviewing environment, Gerrit, to implement features, review patches and bug fixes. It supports lightweight processes for reviewing code changes, i.e., to decide whether a developers change is safe to integrate into the official Version Control System (VCS). During this process, 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 VCS, otherwise the change is abandoned and the developer will move on to something else [17].
To obtain the issue reports and code review data of these ecosystems, we used the MetricsGrimoire toolset [18] provided by Bitergia to mine the repositories of OpenStack, then store the corresponding data into a relational database.

IV. Methodology

To investigate the metrics that we proposed, we collected all the data about the code review process, from the intention, which corresponds to the moment when a report describing a possible bug is opened, up to the moment the fix, for that bug, runs in the code base of the project.
In order to achieve this, we first extracted the code review process information from Launchpad and Gerrit.
We then preprocessed the data, identified the attributes for measuring the metrics, and finally performed our analysis. The operations of data extraction and processing were automatic.
In order to ensure the quality of the data collection, after every applied heuristic, we manually analysed the selection picking up a number of random elements. Finally, we related the results of our analysis with that of the previous studies and draw our results.

A. Data Extraction and Preprocessing

In our previous work ([19], [20]), we have described the extraction process and the resulting dataset for both Launchpad[1] and Gerrit[2] repositories, using the Metrics Grimoire [18] toolset. The data is also available for other researchers to use.
In the remainder of this section, we discuss what implies the bug triaging and review activity for the project, and briefly describe how we extracted all the information necessary to analyse each metric.

1) Bug Triaging: This metric is a measure of the time that developers need to identify if an issue, that has been reported, is a bug. How do developers identify which ticket [3] is describing a bug? This is not straightforward in OpenStack. Analysing the Launchpad work flow, we came across a pattern in the evolution of a ticket's states. While going through the bug triaging activity, developers use different states for a ticket, in different stages, which allows the developers to indicate that a given ticket is describing a bug. When a ticket, stating a possible bug, is opened in Launchpad, its status is set to *New*. If the problem described in the ticket is reproduced, the bug is confirmed as genuine and the ticket status changes from *New* to *Confirmed*. Only when a bug is confirmed, the status then changes from *Confirmed* to *In Progress* the moment when an issue is opened for review in Gerrit.

[1]activity.openstack.org/dash/browser/data/db/tickets.mysql.7z
[2]activity.openstack.org/dash/browser/data/db/reviews.mysql.7z
[3]in Launchpad, the report of an issue is called a ticket.

Thus, we analysed the Launchpad repository searching for tickets that match with this pattern. These are the tickets that have been classified as bug reports.
Once identified, we extracted them in a new repository for further inspection. Our results showed that, begining from 2010 up to January 2017, in Launchpad, 83.254 tickets out of 123.997 have been classified as bugs. Hence approximately 67% of the total tickets in Launchpad have been reproduced as genuine bugs, and an issue for fixing them has been respectively opened in Gerrit.
At this point we are able to quantify the time that developers spend during the bug triaging activity as the distance in time between the moment when a ticket is first opened in Launchpad up to the moment it is confirmed as a genuine bug.

2) Review Time: The next step is to link the tickets (bugs) that we have already extracted from the issue tracking system with their respective review the Gerrit code review system. Traceability of this linkage is not a trivial task in OpenStack. To detect the links between ticket and reviews, we referred to the information provided in the comments of the tickets. When a proposal for a fix, or a merge for a fix, is received, it's information is reported in the comments of the respective issue.
But, the first problem that arises from the comments is that in some cases they are a summary of the commit history. In those cases we find more than a matching review with the issue we are analysing. But not all these commits are merged in the master branch of the project that originated the ticket. Hence we obtain the incorrect matching bug-review.
However, manually analysing a number of random tickets, we noted that the comments reporting a merge do have a structured format. In this structure, the information related to the review is stated at the head of the comment. More specifically, the 6th attribute of this structure provides the branch where the current fix was merged.
Thus the first step, towards the match bug-review, is trunking the comments. We extracted the comment's head, exactly the first 6 lines from every one of them. By doing this, we are able to match each bug with the review process that has provided a fix for it into the master branch of the project.
At this point we are able to quantify the time to review in OpenStack as the distance in time from when an issue for fixing the bug is started in Gerrit up to when a fix is merged to the code base (master branch of the project).

V. Analysis

In this section we display the results that were obtained from the analysis of OpenStack project for both metrics: bug triaging time and review time. Our purpose usefulness was discussed in section I.

The researchers community is continuously gaining interest in those two metrics, analysing them in different experiments (for example [1], [2], [3], [19], [21] etc). But while there are studies that have quantified time to review [1], [2], at the best of our knowledge still no study has done the same for bug triaging.

Despite this, more and more attention is directed at understanding the impact that bug triaging has on the overall review process and to what factors influence this activity.
On one hand the industries put a lot of effort in individuating and fixing defects injected by development. But on the other hand they need the review to be short, because a long lasting process usually is not that effective.

A long review may generate process stalls and affects whoever might be waiting to take a dependency on the new code. Additionally, the longer the review time is, the more complicated is for the author to move back to the change and integrate the feedback of the reviewers without potentially introducing new defects.

This is why, in this context, we underline the importance of maintaining the process time as controlled as possible. And the two most important elements influencing this time are precisely bug triaging and review time. By supervising these metrics the review process time can be as short or long as according to the case.

In order to show our assumption we first analysed the ICR process, then extracted the two metrics of BTT and CRT, with the purpose to better understand the inlfuence that these metrics have on ICR.

The data we are analysing from OpenStack, comprises all the historical span from 2010 to January 2017. As we mentioned in section IV, in this overall history, 83.254 tickets were classified as bugs. For our analysis, we are considering only the fixed bugs, which change has been merged into the code base, thus excluding the reviews still in progress, abandoned and duplicated issues.

Therefore, the total number of tickets we have analysed is 59.209, which is approximately 71% of the total number of identified bugs.

VI. Results

We computed the three measures from Intention to Code Review (ICR), Bug Triaging Time (BTT) and Code Review Time (CRT) as the following:

ICR - the time from when the ticket is opened in the issue tracking system, to when the fix for the bug is merged into the code base;
BTT - the time from the moment when the ticket is opened in Launchpad up to the moment it is *Confirmed* as a genuine bug.
CRT - the time from when the first patch is uploaded in Gerrit up to when the change is merged to the code base.

Afterwards, we calculated the median effect size across all Open Stack projects history in order to globally rank the metrics from most extreme effect size.
Additionally, we calculate and expose the resuts for the ICR, BTT and CRT per quantiles.

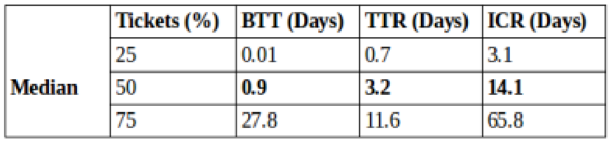
We found that the median time for:

a. bug triaging in OpenStack (Launchpad) is 0.9 days,

b. code review in OpenStack (Gerrit) is 2.1 days,

c. intention to code review in OpenStack is 14.1 days.

The results of the quantiles for BTT, TTR and ICR are shown in the table 1, then additionally the same results are shown in graphical form 2 to visually let us understand better what portion of ICR occupy BTT and TTR. The x-axis represents the percentage of tickets, while the y-axis represents the number of days. The results in both Fig. 1 and Fig. 2 are measured in days:



	Tickets (%)	**BTT (Days)**	**TTR (Days)**	**ICR (Days)**
Median	25	0.01	0.7	3.1
	50	**0.9**	**3.2**	**14.1**
	75	27.8	11.6	65.8

Fig. 1. The quantiles for BTT, TTR and ICR measured in days.

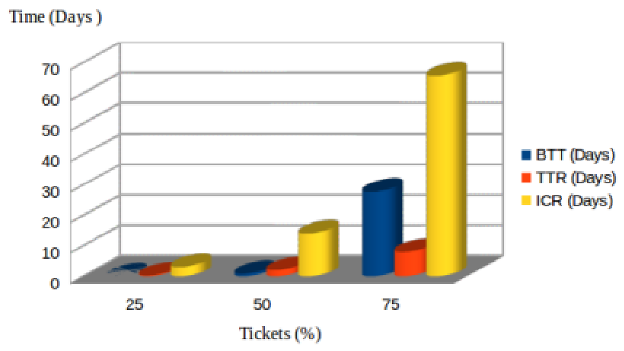
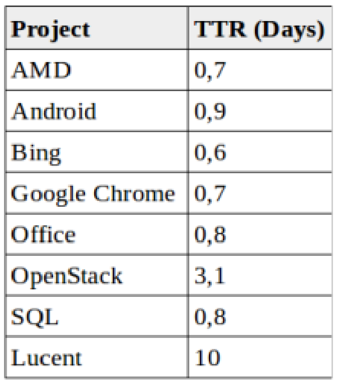


Fig. 2. Chart of the quantiles for BTT, TTR and ICR measured in days.

VII. Conclusions and Future Work

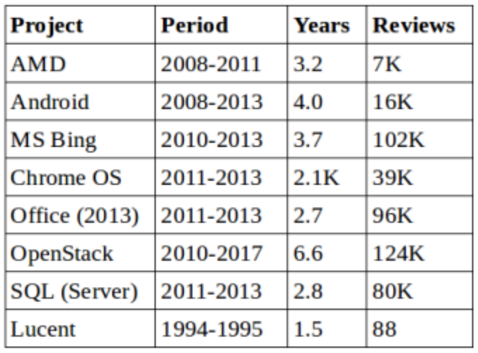
As previously mentioned, related the results obtained from our analysis with that of the previous studies [1], [22], in an effort to characterize and understand the differences. These studies have analysed the review process for some of the AMD, Microsoft, and Google-led projects. The results from these studies along with our analysis of OpenStack are shown in the table below (Fig. 3):



Project	TTR (Days)
AMD	0,7
Android	0,9
Bing	0,6
Google Chrome	0,7
Office	0,8
OpenStack	3,1
SQL	0,8
Lucent	10

Fig. 3. Time To Review (TTR) in different projects.

Additionally in the table below, Fig. 4 we add more context to the results by displaying characteristics of the projects analysed: the time period examined in years and the number of reviews.



Project	Period	Years	Reviews
AMD	2008-2011	3.2	7K
Android	2008-2013	4.0	16K
MS Bing	2010-2013	3.7	102K
Chrome OS	2011-2013	2.1K	39K
Office (2013)	2011-2013	2.7	96K
OpenStack	2010-2017	6.6	124K
SQL (Server)	2011-2013	2.8	80K
Lucent	1994-1995	1.5	88

Fig. 4. Time To Review (TTR) in different projects.

The TTR metric has been studied in a number of experiments over the last 40 years ([5], [3], [23]). Our contribution here is to add the large cloud computing project, such as OpenStack, to the existing diverse set of projects analysed on this parameter.
The results displayed both in Fig. 3 and 4, show that despite differences among projects, the parameter of TTR have similar values. Our main assumption is that if some of the code review metrics in different projects have become similar as the projects have normally advanced, then such metrics may be indicative of practices that may represent commonly successful and efficient methods of review.
The only measure we have, at the best of our knowledge, for the BTT metric are some indications provided by [24] and [25], two experiments respectively conducted with the support data from Microsoft and Bugzilla. They found that *bug triage takes up a large amount of developers resources and it goes from about 24 hours to many lasting days, if not weeks*. Our contribution here is to bring some numbers on how long this process takes in a large cloud computing project, such as OpenStack.

The final purpose of this paper is to highlight, from the ICR process, the two metrics of BTT and TTR, and recognize them as key metrics to supervise during the review process. You can't control what you can't measure. And by controlling these metrics, not only the effectiveness of code review can be quantified and the progress of a development team can be measured, but also policies and guidelines, which can improve the development process, may arise. As such, these metrics can be prescriptive to other projects taking into consideration to add code review to their development process.
Additionally, in a continuous integration and deployment environment, such as OpenStack, the role that the two activities of bug triaging and code review have on the overall process of ICR (Fig. 1 and 2) becomes of vital importance. Not only the quality of the code under development must be assured, but also the time waiting to take possesion of the new code must be relatively as short as possible. Thus controlling ICR, and the parameters that affect it, including here BTT and TTR, becomes crucial to the success of a project itself.

Taking into consideration the above discussion, our future work is to continue analysing (along with ICR, BTT and TTR) more phases from the software development process. We highlight the fact that also the feature implementation follows the same workflow in Gerrit.
Thus, our work will be aimed at characterizing various phases highlighting those parameters that influence the performance of the process, and define what factors influence the paramenters itself. The purpose of our work is raising specific metrics as key performance indicators (KPIs). They will be KPIs that when controled are able to improve the software development process either by discovering software development policies or guidelines, and by setting configurations of a development team.

ACKNOWLEDGMENT

The authors would like to thank SENECA, EID project under Marie-Skodowska Curie Actions, for funding one of the authors, and Bitergia company for the Metrics Grimoire toolset, which realised the extraction of the dataset used in this paper.

REFERENCES

[1] Peter C. Rigby and Christian Bird. 2013. *Convergent contemporary software peer review practices.* In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE2013). ACM, New York, NY, USA, 202-212.
[2] Amiangshu Bosu and Jeffrey Carver. 2013. *Impact of Peer Code Review on Peer Impression Formation: A Survey.* Proceedings of the 7th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2013. Baltimore, MD, USA, 133-142.
[3] A. Porter, H. Siy, A. Mockus, and L. Votta. *Understanding the sources of variation in software inspections.* ACM Transactions Software Engineering Methodology, 7(1):4 1-79, 1.
[4] D. L. Parnas and M. Lawford. *Inspections role in software quality assurance.* In Software, IEEE, vol. 20, 2003.
[5] M. E. Fagan. *Design and Code inspections to reduce errors in program development.* In IBM Systems Journal 15 pp. 182-211, 1976.
[6] 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.

[7] 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.

[8] 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.

[9] 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. 775-802, 2000.

[10] Hallem, D. Park, and D. Engler. *Uprooting software defects at the source.* Queue, vol. 1, no. 8, pp. 64-71, 2003.

[11] B. Chess and J. West. *Secure Programming with Static Analysis.* 1st ed. Addison-Wesley Professional, Jul. 2007.

[12] N. Kennedy. *How google does web-based code reviews with mondrian.* http://www.test.org/doe/, Dec. 2006.

[13] 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.

[14] Gerrit code review - https://www.gerritcodereview.com/.

[15] Baysal, O., Kononenko, O., Holmes, R., Godfrey, M. W. (2015). *Investigating technical and non-technical factors influencing modern code review.* Empirical Software Engineering, 1-28.

[16] K. E. Wiegers. *Peer Reviews in Software: A Practical Guide. Addison-Wesley Information Technology Series.* Addison-Wesley, 2001.

[17] http://docs.openstack.org/infra/manual/developers.html

[18] J. M. Gonzalez-Barahona, G. Robles, and D. Izquierdo-Cortazar. *The metricsgrimoire database collection.* In 12th Working Conference on Mining Software Repositories (MSR), pages 478-481, May 2015.

[19] Dorealda Dalipaj. *A quantitative analysis of performance in the key parameter in code review - Individuation of defects.* Proceedings of the Doctoral Consortium at the 12th International Conference on Open Source Systems (OSS) 2016, Gothenburg, Sweden, 11-24.

[20] Dorealda Dalipaj, Jesus M. Gonzalez-Barahona, Daniel Izquierdo-Cortazar. *Software engineering artifact in software development process Linkage between issues and code review processes.* Proceedings of the 15th International Conference on New Trends in Intelligent Software Methodology Tools and Techniques (SoMeT) 2016, Larnaca, Cyprus, 115-122.

[21] P. C. Rigby, D. M. German, and M.A. Storey. *Open source software peer review practices: A case study of the apache server.* In ICSE: Proceedings of the 30th international conference on Software Engineering, pages 541-550, 2008.

[22] P. C. Rigby. Understanding Open Source Software Peer Review: Review Processes, Parameters and Statistical Models, and Underlying Behaviours and Mechanisms. http://hdl.handle.net/1828/3258. University of Victoria, Canada, Dissertation, 2011.

[23] P. C. Rigby, D. M. German, and M. A. Storey. *Open source software peer review practices: A case study of the apache server.* In ICSE: Proceedings of the 30th international conference on Software Engineering, pag. 541550, 2008.

[24] Czerwonka Jacek, Michaela Greiler and Jack Tilford. *Code Reviews Do Not Find Bugs. How the Current Code Review Best Practice Slows Us Down.* Proceedings of the 2015 International Conference on Software Engineering. IEEE Publisher, 2015.

[25] 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, pp 336-343.