Software Engineering Artifact in Software Development Process - Linkage between bugs and issues.

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

ABSTRACT

Researchers working with software repositories, often when building performance or quality models, need to recover traceability links between bug reports in issue tracking repositories and reviews in code review systems.

However, too often the information stored in bug tracking repositories is not explicitly tagged or linked to the issues reviewing them. Researchers have to resort to heuristics to tag the data (for example, to identify if an issue is a bug report or a work item).

In this study we promote a *research artifact* in software engineering, a reusable unit of research that can be used to support other research endeavors and has acted as a support material that enabled the creation of the results published in a great number of papers until now - linking bugs and issues of the code review software process. We present two *state-of-the-practice* algorithms on how to link bugs and issues, picking as case study the open source cloud computing project, OpenStack.

OpenStack enforces strict development guidelines and rules on the quality of the data in its issue tracking repository. We empirically compare the outcome of the two approaches, highlighting the most prominent one.

Keywords

Research artifact, software engineering, state-of-the-practice algorithms, open source.

1. INTRODUCTION

Software process data, especially bug reports and code changes, are widely used in software engineering research. The integration of reports and reviews, on one hand, provides valuable information on the history and evolution of a software project [1]. On the other hand, it provide the means for conducting studies that qualitatively and quantitatively analyse fundamental parameters for characterization of quality and performance of the code review and the most important metrics having a positive increasing relation with the benefits of this process.

It is used, e.g., to build models which study the impact of characteristics of issue and review discussions on the defect-proneness of a patch [2], to spot defect-introducing code changes during review, to predict the number and locale of bugs in future software releases, or to investigate technical and non-technical factors influencing modern code review ([3, 4, 5, 6, 7]).

However, to achieve such studies, much of the information stored in software repositories must be processed and cleaned for further analysis. Such processing and cleaning is often done using assumptions and heuristics. A generally accepted assumption is that most issue tracking systems contain just bug reports. Actually, they contain other items such as approved design specifications for additions and changes to the project team's code repositories or light weight feature specifications.

The quality of these assumptions and heuristics represents a threat to the validity of results derived from software repositories. Recent studies, by Bird et al. [8] and Antoniol et al. [9], have casted doubts on the quality of data produced using such heuristics. Bird et al. [8] note that when considering links between bug reports and actual code, there exists bias in the severity level of bugs and the experience of developers fixing bugs, thus a quality model built using only the linked bugs, will be biased towards the behaviour of more experienced developers. Antoniol et al. [9] note that many of the issues in an issue tracking system do not actually represent bug reports, therefore, using such data might lead to incorrect bug counts for the different parts of a software system.

To prevent or minimize such situations, we present two algorithms to a correct way of linking bugs and issues, which take into consideration the assumptions that researchers performing this task usually consider as true. Our approach uses a reverse engineered model of the code review process, and extracts key information from the issue tracking and review systems of OpenStack. The results we bring evidence, do not involve subjective context but are material facts. As such, they can be recorded to trace the efficiency and effectiveness of the two methods.

We have used such approach to support a previous study [10] that brings evidence of the performance of code review by quantifying the time spent by developers to identify the bugs reports, the time spent by them to carry out the reviewing process, as well as to assess the performance of code review - individuation of bugs.

WOODSTOCK '97 El Paso, Texas USA

DOI: 10.475/123_4

Summarizing, this paper provides the following two contributions:

- we present two fully automated approaches for linking issues from bugtracking system to reviews from code review system of Open Stack, bringing out their advantages and/or disadvantages,
- we compare the results the two methods highlighting which is the most outstanding.

In the remainder of this paper, we first describe the necessary background notions for our work (section 2). Next, we describe the methodology (section 3), then present the results and discuss some conclusions (section 4). After future work (section 5), we conclude with aknowledgements (section 6).

2. BACKGROUND

This section provides background information about the bug-tracking and code review environments of OpenStack and the tools for obtaining data from its repositories.

We choose OpenStack because it is a large project that has adopted code reviews on a large scale, and therefore with an abundance of data coming from the engineering systems. OpenStack uses Launchpad, an issue tracking system, which is a repository that enables 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. OpenStack uses a dedicated reviewing environment, Gerrit, to review patches and bug fixes. It supports lightweight processes for reviewing code changes, i.e., to decide whether a developer's 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.

To obtain the issue reports and code review data of these ecosystems, we used the data set provided by González-Barahona et al. [24]. They developed the MetricsGrimoire tool to mine the repositories of OpenStack, then store the corresponding data into a relational database.

3. METHODOLOGY

Our approach uses a reverse engineered model of the code review process, and extracts key information from the issue tracking and code review systems.

In the specific case of OpenStack, the issues contain the identification of the bug it is reviewing. While, on the other hand, the comments of the bugs contain the identification of the issue that is reviewing the ticket. Consequently we extract these information in a new repository for further analysis.

To obtain this information we do the following steps:

1. Extraction of the information from the data sources. OpenStack repositories consists in the data from the issue tracking system (*Launchpad*) and the data from the code review system (*Gerrit*). We extracted these data using the *Metrics Grimoire*[1], a toolset for mining the repositories of OpenStack, provided and mantained by *Bitergia*.
2. Identification of data needed for the linkage. In this process we understand the infrastructure of both the issue tracking system and the code review system of the project we are analyzing. We individuate the elements which carry out the information that permits us to link the reports with the issues from both systems. On the behalf of the code review system we found that the issues indicate the identification of the bug they are reviewing. On the behalf of the bug tracking system we found that the comments of a bug reports the identification of the issue that is reviewing them. Both informations allow us to link the reviews with the bugs and viceversa.
3. Dataset preprocessing. Not all the information provided by the repositories of OpenStack is needed for this analysis. First, we are interested in linking bugs with issues. Thus we can ignore all the tickets in the bug tracking system that are not classified as bug. Second, our final aim is to recover the linkage for the fixed issues. Consequently we can ignore all the issues for which the status is other than 'Fix Released'.
4. Analysis. We use Jupyter Notebook, a computational environment in which we can combine code execution, rich text, mathematics, plots and rich media, along with Pandas, an open source library specialised in providing high-performance data structures and data analysis.

3.1 Linking issues with bugs.

The first approach for linking recovers links from code review system to bug tracking system. To detect the links between issues and bugs, 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" being a bug identifier, which is called the *related artifact*. Applying regular expressions we extracted the bug identification, and then proceded with the linkage. Summarizing the *state-of-practice* algorithm for linking from the code reveiw system to the bug tracking system is:

identify the *related artifact*;

apply regular expresions to extract *bug_id* from *related artifacts*;

link between issues and bugs matching where bug identification is *bug_id*.

The most inconvenient drawback in this approach is that, even though OpenStack project follows a disciplined software development process with careful attention to continuously reporting the information that permits traceability between issues and bugs, in practice, often, developers do not report such related artifacts, resulting in missing links.

After applying the above approach to OpenStack, we identified 72% of the links with issues to the bugs they review.

[1]https://github.com/MetricsGrimoire/

3.2 Linking bugs with issues.

The second approach for linking recovers links from bug tracking system to code review system.

The first problem that arises in this case is how to identify the tickets classified as bugs. In Launchpad, besides bugs reports, the developers can work with *specifications* (approved design specifications for additions and changes to the project code repositories) and *blueprints* (lightweight feature specifications). Identifying 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, analysing the comments of a ticket is not the most efficient way for extracting its classification, not only because we will not identify 100% of the tickets but we risk false positives too.

By manual analysis of the tickets and studying the Launchpad work flow, we came across a pattern in the evolution of a report states, with regards to confirming new bugs:

> a) when a ticket, stating a possible bug, is opened in Launchpad, its status is set to *New*;
>
> b) if the problem described in the ticket is reproduced, the bug is confirmed as genuine and the ticket status changes from *New* to *Confirmed*;
>
> c) 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.

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 analysis.

Next step, we linked tickets to reviews using the information that we find in the comments of the tickets. Whenever a review receives 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/13083
> Committed: https://git.openstack.org/cgit/openstack
> /nova/commit/?id=be58dd8432a8d12484f5553d79a02
> e720e2c0435
> Submitter: Jenkins
> Branch: master ...

We can see that in the first line, clearly, we are provided with the link to the issue in Gerrit.

A problem that arises in analysing the comments is that, for some ticket, they 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 originated the ticket, consequently not the correct result.

However there is a fixed format of the comments that report a merge (which is the one you can see in the example above). In this format, the information related to the review is stated at the very beginning of the comment. Manually analysing the tickets in Launchpad, we have seen that they are found in the first 6 rows of the comment.

Thus the first step is trunking the comments, so that we extract just the first 6 lines from every one of them. Then, we are sure we will identify the right review.

Summarizing the *state-of-practice* algorithm for linking from the the bug tracking system to the code reveiw system is:

> identify *bug reports*;
>
> exctract *issue_id*, the information related to the issue that is reviewing the bug from the comments of the bug;
>
> link between bugs and issues where issue identification is *issue_id*.

After applying the above approach to OpenStack, we identified 90.2% of the links with bugs to the issue reviewing them.

Furthermore, analysing the missing merges from the resulting dataset by picking up arbitrarily tickets, we found that the reason they could not be linked is that some of the gerrit identification, of the issue reviewing the bug, are missing from the Gerrit repository.

For example:

> the bug:
> https://review.openstack.org/#/q/topic:bug/
> 1253497
>
> is fixed in the issue:
> https://review.openstack.org/
> #/c/92547/ ...
>
> but the issue id 92547 is not found in the Gerrit database.

However, we would note that, thanks to the heuristics applied, there are no false positives in the resulting dataset in this case.

3.3 Comparison of Results and Conclusions

In this paper we present two *state-of-the-practice* algorithms regarding what is by now a *research artifacts* in software engineering. We discussed it's reusability and usefulness to support other research endeavors in section 1. We run the two algorithms on a large open software project, such OpenStack and obtained two distinct results (Table 1), showing clearly that one approach outstands the other. We hope these results will be a guideline to other researchers in future helping them to decide how to recover the links at the best of their advantage. While, on one hand, we tested our approach on a single case study, OpenStack, on the other what enfroces our confidence that this case acts as a representative for the category is that code review process follows the same principles and practices everywhere.

4. FUTURE WORK

We would like to continue to explore the power of the information from the code review process. In section 1 we mentioned a study that noted that when considering links between bug reports and actual code, there exists bias in the severity level of bugs and the experience of developers fixing bugs, thus a quality model built using only the linked bugs, will be biased towards the behaviour of more experienced developers.

To prevent and minimize such situations, we would require a perfect dataset where such linkages and tagging of issues was done by professional developers and with great attention

Table 1: The results of the two state-of-the practice algorithms applied on OpenStack (OS) history from July 2010 - January 2016.

Number of tickets in OS	Nr. of tickets fixed in OS	1st Approach (Gerrit-Launchpad)	2nd Approach (Launchpad-Gerrit)
88421	70587	72%	90.2%

to detail. While such a perfect dataset might not exist, we believe that a near-ideal dataset does exist. This dataset is derived from the OpenStack project.

The OpenStack project follows a disciplined software development process with careful attention to continuously maintaining the linkages between issues and code changes through automated tool support. The strong discipline and the automated tool support gives us strong confidence that this project represents a near-ideal case of high quality linkage dataset that is correctly tagged.

Therefore, in our immediate future work we aim to verify the effects of such biases for the OpenStack project, to see if even with a near-ideal dataset, biases do exist - leading us into verifying the conjecture if biases are more likely a symptom of the underlying software development process instead of being due to the used heuristics.

5. ACKNOWLEDGMENTS

We would like to thank the SENECA EID project, which is funding this research under Marie-Skodowska Curie Actions.

6. REFERENCES

[1] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. Proceedings of the International Conference on Software Maintenance (ICSM 03). IEEE Computer Society, 2003.

[2] The Impact of Human Discussions on Just -In-Time Quality Assurance. Parastou Tourani and Bram Adams. SANER 2016 - 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering.

[3] A. Bernstein, J. Ekanayake, and M. Pinzger. Improving defect prediction using temporal features and non linear models. In IWPSE'07, pages 11-18, Dubrovnik, Croatia, September 2007.

[4] 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.

[5] H. Joshi, C. Zhang, S. Ramaswamy, and C. Bayrak. Local and global recency weighting approach to bug prediction. In MSR'07, pages 33-34, Minneapolis, Minnesota, USA, May 2007. IEEE Computer Society.

[6] M.-T. J. Ostrand, F.-E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. IEEE Trans. Softw. Eng., 31(4):340-355, 2005.

[7] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In MSR'05, pages 24-28, Saint Louis, Missouri, USA, May 2005. ACM.

[8] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (FSE 09). Amsterdam, The Netherlands: ACM, 2009, pp. 121-130.

[9] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.G. Guhneuc. Is it a bug or an enhancement?: a text-based approach to classify change requests. Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds (CASCON 08). Ontario, Canada: ACM, 2008, pp. 304-318.

[10] Dalipaj Dorealda. A Quantitative Analysis of Performance in the Key Parameter in Code Review - Individuation of Defects. International Conference on Open Source Systems (OSS) 2016, Doctoral Consortium.

[11] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In WCRE'03, pages 90-99, Victoria, B.C., Canada, November 2003.

[12] D. Cubranic and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In ICSE'03, pages 408-418, Washington, DC, USA, 2003.

[13] A. Schroter, T. Zimmermann, R. Premraj, and A. Zeller. If your bug database could talk... In ICSE'06, pages 18-20, Rio de Janeiro, Brazil, September 2006.

[14] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In PROMISE'07, pages 1-9, Minneapolis, Minnesota, USA, May 2007. IEEE Computer Society.

[15] T. Zimmermann and P. Weissgerber. Preprocessing cvs data for fine-grained analysis. In MSR'04, pages 2-6, Edinburgh, Scotland, UK, May 2004. ACM.

[16] G. A. Liebchen and M. Shepperd. Data sets and data quality in software engineering. In PROMISE'08, pages 39-44, New York, NY, USA, 2008.

[17] N. Bettenburg, S. Just, A. Schroeter, C. Weiss, R. Premraj, and T. Zimmermann. Quality of bug reports in eclipse. In eTX'07, pages 21-25, Montreal, Canada, October 2007.

[18] N. Bettenburg, S. Just, A. Schroeter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? Technical report, Saarland University, Saarbrucken, Germany, September 2007.

[19] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Duplicate bug reports considered harmful... really? In ICSM'08, pages 337-345, October 2008.

[20] P. Hooimeijer and W. Weimer. Modeling bug report quality. In ASE'07, pages 34-43, Atlanta, Georgia, USA, November 2007.

[21] K. Chen, S. R. Schach, L. Yu, J. Outt, and G. Z. Heller. Open-source change logs. Emp. Softw. Eng., 9(3):197-210, 2004.

[22] A. Bachmann and A. Bernstein. Software process data quality and characteristics: a historical view on open and closed source projects. In IWPSE-Evol'09, pages 119-128, Amsterdam, The Netherlands, August 2009.

[23] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In ICSE'09, pages 298-308, May 2009.

[24] 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.