An Automated Approach to Assessing Code Quality via Bug Reports

Yuexiao Teng*
East China University of Science and Technology, Shanghai, China

*Corresponding author. E-mail: roger.yxt@aliyun.com
Manuscript submitted January 1, 2020; accepted January 31, 2020.
doi: 10.17706/jsw.15.1.23-29

Abstract: Mining Software Repositories (MSR) can serve various purposes such as analyzing code coverage, predicting code changes. One of software repositories is bug repositories. However, bug repositories, i.e. bug reports, has not been made full use of due to that many researchers or practitioners lay little emphasis on mining bug reports. It is because they may not be aware of its value or practical meaning. In order to take advantage of bug reports, in the paper, an original approach has been put forward to assessing code quality or at least estimating the status of software by mining bug reports, which is automated. The results of experiments show the effectiveness of this novel approach.

To author's best knowledge, it was the first time to make such full use of mining bug reports in an automated way.

Key words: Data mining, mining software Repositories (MSR), bug reports, code quality.

1. Introduction

Software tools (e.g., source code repositories, bug tracking systems) are useful in developing, testing and maintaining software systems. The logs or reports of these tools are called software repositories. Typically, source control repositories, bug repositories, archived communications, deployment logs, and code repositories are examples of software repositories that are commonly available for most software projects [1]. They contain rich and special information e.g. code changes, bug fixes, memory dump and therefore they are fairly worth being studied.

However, currently researchers and engineers do not lay enough emphasis on mining bug reports. A bug report contains many fields, such as product, component, severity, priority, fixer, operating system (OS), platform, etc., which provide important information for the bug triaging and fixing process [2]. Thus, mining bug reports could be useful in software engineering, software testing, process management and software quality. In this paper, I have proposed an automated approach to evaluating code quality by mining bug reports. Since computer science or engineering research are usually conducted on open-source software e.g. E. Kouroshfar *et al*. [3], [4] carried out research on HBase and E. D. S. Maldonado *et al*. [5]-[15] did case studies on Tomcat, I carefully selected four open-source software for experimental purposes. I firstly exported some bug reports from the open-source software e.g. HBase, Tomcat. The next step was that I leveraged Java analysis techniques to mine these reports. Based on the experimental results, several metrics are measured in the experiments so as to assess the code quality. The contributions of the paper are:

1) This work evaluates the code quality from a practical perspective. To author's knowledge, this is the first time to evaluate the code quality via bug reports in an automated way.

2) Case studies on open source software projects e.g. Tomcat7, Crimson, HBase demonstrate that the code quality measure evaluated by my proposed approach is accurate.

Paper Organization: the rest of the paper is structured as below. Section 2 summarizes the related work. Section 3 explains the automated approach and the algorithm. Section 4 describes the experiments and presents the analysis. Section 5 discusses the threats to validity. Section 6 draws a conclusion and introduces the future work.

2. Related Work

Mining software repositories is one of popular research topics in terms of software engineering as well as data mining and it has been studied for the last decades. Yet, currently there are not sufficient research efforts made in mining bug reports in evaluating code quality.

Yu *et al*. [5] studied open-source software systems to investigate whether the number of bug reports is correlated with the software quality. They concluded that making use of accumulated defect reports as a metric to representing the quality of a software branch. However, they only focused on the relationship between the numbers of defects reports and their practical meanings and their mining method was manually on the Bugzilla site.

Ahmed Lamkanfi *et al*. [6] investigated the reassignments of bug reports, aiming to address inaccurate information in bug reports. Therefore, the authors have proposed data-mining techniques to make early predictions of which particular reported bugs are assigned to the incorrect component. They conducted case studies on open-source software Eclipse and Mozilla. The results have proved the effectiveness of the proposed approaches.

Sarah Rastkar *et al*. [7] empirically did case studies on how to summarize bug reports in an automated way. They found that that existing conversation-based extractive summary generators can produce summaries for reports that are better than a random classifier. They also found that an extractive summary generator trained on bug reports produces the best results. Their experimental results show that the automatic summaries could help developers save time.

Boyuan Chen *et al*. [8] have made research efforts to design a novel and automated code coverage measures via execution logs called LogCoCo, Log-based Code Coverage, yielding high accuracy. Their approach was automated and the results of LogCoCo can be used to evaluate test case suits, show the practical value of the proposed approach.

3. The Automated Approach Via Bug Reports

In this section, I would first present an overview of the approach and then I would describe the algorithm and approach in details.

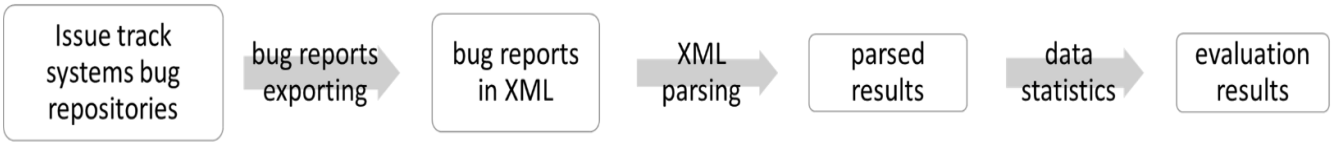

Fig. 1. An overview of the approach

Overall, I undertook empirical studies as my research method.

As illustrated in Fig. 1, there are three major phases from bug exporting to data statistics. I used Java as programming language and DOM4j, a flexible and popular XML framework intended for Java supporting DOM,

JAXP as well as SAX, was applied to parse XML. Then, I carried out data statistics on parsed results.

In the following part, I would introduce my approach in detail to evaluate code quality via bug reports. There are two popular bug tracking systems e.g. Jira [9] which does incomplete bug statistics function manually, Bugzilla [10] which does not support bug statistics function. I undertook empirical studies on these two kinds of bug reports. In addition, there are likewise two kinds of XML bug reports. One is that only one XML file containing the all bugs information e.g. bug ID, creation timestamp and the other is that each bug has its own XML file, which means that there are possibly a huge number of XML files. For the two kinds of bug reports, I can use two different Java methods to handle.

First and foremost, bug reports were exported in XML from Jira or Bugzilla, which was an important step to prepare data for further mining. Then, according to bug report formats one-file or multiple-file XMLs, a Java-based algorithm was applied as generally shown in Fig. 2. Path parameters are respectively passed to the mining functions. The algorithm is iterative and it has nested while loops to parse the XML nodes e.g. bug_status, RESLOVED. Please note that exported Bugzilla bug reports are usually in one-XML-file format.

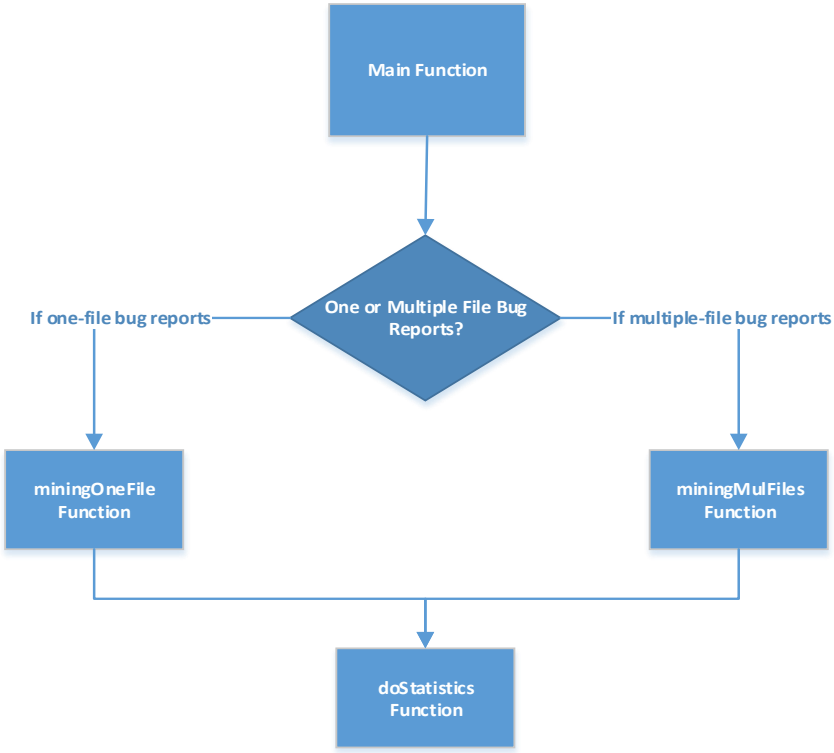

Fig. 2. The general process of the algorithm.

As for one-XML-file case, the idea behind this approach was traversing the bug report file, i.e. the root node <bugzilla>, in XML containing all bug Reports stored in child nodes <bug> with extracting bug ID. Per bug report node, the first step was parsing <bug> node to find the < creation_ts>, < bug_status> and <assigned_to> nodes. Secondly, I processed these nodes. If the value of a < bug_status> node is RESOLVED, this means the corresponding bug has been resolved and thus I should set the bug resolved tag to true, being stored in a HashMap(a hash table based built-in data structure implemented in Java programming language); vice versa. If the value of a <assigned_to> node is not unassigned or empty, this means the corresponding bug has been assigned and thus I should set the bug assign tag to true, being stored in a HashMap; vice versa. Then, if a bug has not been solved, I processed (e.g., converting type), calculated the unsolved time (unsolved time = current

– created) and stored the results in a HashMap. Finally, I carried out the mathematics statistics:

Bug Resolved Rate = The Sum of Resolved Bugs / Total Number of Bugs;

Bug Assigned Rate = The Sum of Assigned Bugs / Total Number of Bugs;

Maximum Bug Unresolved Time; Minimum Bug Unresolved Time; Average Bug Unresolved Time.

As for multiple-XML-file case, the idea of this approach was traversing the bug report directory containing all bug report xml files. Per bug report xml, the first step was parsing <item> node to find the < assignee >, <created> and <resolved> nodes. Secondly, I processed these nodes. If a <resolved> node does not exist, this means the corresponding bug has not been resolved and thus I should set the bug resolved tag to false, being stored in a HashMap; vice versa. If the value of a < assignee> node is Unassigned, this means the corresponding bug has not been assigned and thus I should set the bug assign tag to false, being stored in a HashMap; vice versa. Then, if a bug has not been solved, I processed (e.g., converting type), calculated the unsolved time (unsolved time = current – created) and stored the results in a HashMap. Finally, I executed the same statistics function as one-XML-file case.

4. Experimental Results and Analysis

In this section, firstly experimental results of the mining programme are presented and then I would analyse these results alongside with the implications.

I ran the Java mining programme on four open-source software e.g. HBase(Jira), Tomcat7(Bugzilla), Crimson(Bugzilla), Httpd-1.3(Bugzilla), in order to study bug reports for the assessment of code quality. My Java programme includes 2 parsing methods miningOneFile, specialising in mining one-file bug reports, as well as miningMulFiles, specialising in mining multiple-file bug reports.

4.1. Experimental Findings

In the experiments, code quality, is evaluated by measuring quantitative metrics. The quantitative indicators of programme execution results are presented in the Table I, respectively.

Table 1: The Statistics of Bug Reports Per Open-source Software

Quantitative Metrics	Bug Resolved Rate	Bug Assigned Rate	Maximum Bug Unresolved Time(in Day)	Minimum Bug Unresolved Time(in Day)	Average Bug Unresolved Time(in Day)
HBase	85%	73%	2728	0	525
Tomcat7	98%	96%	3434	2	40
Crimson	6%	8%	6505	3623	5734
HTTPD-1.3	44%	40%	6461	3336	3409

4.2. Results Analysis

Since bugs are normally related to function, efficiency or maintenance defects and code quality or software quality are measured in these metrics according to ISO/IEC-9126 quality characteristics, i.e. Functionality, Maintainability, Efficiency, Portability [13], the results of mining bug reports can be useful in evaluating code quality. Note that using ISO/IEC-9126 standard to evaluate code quality is common e.g. Yiannis Kanellopoulos et al. has proposed a methodology to evaluate source code quality based on the standard [14]. Regular bugs are classified and prioritised based on different standards or definitions by scholars or organizations. For example, Microsoft, one of the largest and top software companies in the world, has its own standards to define software bugs e.g. rating software's bugs on a three point scale [15]. Both Bugzilla [1] and Jira [10] defines bug severity as Blocker, Critical, Major, Minor and Trivial.

In the following, I would discuss five quantitative metrics measuring code quality based on the statistics of bug reports.

1) Bug Resolved Rate (BRR) means that the number of resolved bugs takes up the total number of reported bugs. This metric can be used to assess the robustness and availability of code, which is an aspect of code quality.
2) Bug Assigned Rate (BAR) means that the number of assigned bugs takes up the total number of reported bugs. This metric can be used to assess how many resources are spent on and how much attention are paid to bug fixes, showing the efforts to keep a certain level of code quality. Thus, this can indirectly evaluate code quality.
3) Maximum Bug Unresolved Time (MBUT) means that the maximum value of how long a bug has not been resolved since it was reported on bug tracking systems.
4) Minimum Bug Unresolved Time (NBUT) means that the minimum value of how long a bug has not been resolved since it was reported on bug tracking systems.
5) Average Bug Unresolved Time (ABUT) means that the average value of how long a bug has not been resolved since it was reported on bug tracking systems.

These above three metrics, all related to bug unresolved time, can be used to measure the time delay between unresolved bug creation time and the current time in mathematics, which shows the unsoundness of code quality in because this shows how long software systems can not work properly in terms of remaining unresolved bugs. The bug unresolved time can be one of factors in a number of factors which contribute to software rot. As described by Andrew Hunt and David Thomas [16], while software development is immune from almost all physical laws, entropy hits us hard. Entropy is a term from physics that refers to the amount of "disorder" in a system. Unfortunately, the laws of thermodynamics guarantee that the entropy in the universe tends toward a maximum. When disorder increases in software, programmers call it "software rot."

Based on the experimental results per open-source software, the following analysis was undertaken case by case.

In the HBase case, BRR is high and BAR is acceptable, achieving 73%. The remaining three Bug Unresolved Time indicators tell us that the average unresolved bugs have been existing for around one and half years, which is not short. The code quality in HBase could be evaluated to middle.

In the Tomcat7 case, BRR is almost 100% and BAR is 96%. The remaining three Bug Unresolved Time indicators tell that the unresolved bugs have not been existing for a long in code base. Specially, ABUT has achieved at 40, which is petty short. There are some significant development efforts made to maintain Tomcat7 and therefore the code quality could be assessed to high quality.

In the Crimson case, BRR and BAR are very low. The remaining three Bug Unresolved Time indicators demonstrate that the unresolved bugs have been still in code repositories for an extremely long time(a number of years). Therefore, this suggests that no resources have been allocated on this probably obsolete project and thus the code quality in Crimson could be assessed to low.

In the HTTPD-1.3 case, BRR is middle, at around 44% and BAR is 40%. The remaining three bug unresolved time indicators tell us that compared with Crimson, the MBUT and NBUT are almost equal. Furthermore, ABUT is quite long, at 3409. All the evaluating metrics jointly show that the code quality could be evaluated to low and the reason could be as same as Crimson's.

The above analysis results can be justified by Table 2, presenting the information, which is collected from the official websites easily searched by Google or Baidu, about the latest status of the open-source software I have investigated at the moment.

Implications

Code review aims to examine your work products for defects and improvement opportunities [11].

Naturally the results of code review can be helpful in evaluating code quality. Yet, reviewing all source codes might be laborious and inefficient. Compared with code review, automated mining bug reports is quicker and still effective, providing a measure of evaluating code quality as well as software quality [12] or at least an estimate of the status of software.

Table 2. The Current Status of Open-source Software

Open-source software	Status
HBase	In development
Tomcat7	Currently used and one of two stable versions(7 and 9)
Crimson	Retired since 2010
HTTPD-1.3	No new releases since 2010 and should not be used

Through the proposed quantitative metrics e.g. bug resolved rate, the results of bug reports statistics programmes could be one of measures in assessing code quality by researchers or practitioners. They might take actions e.g. based on the evaluation results, enhancing code quality in software projects.

5. Threats to Validity

In the section, the threats to validity will be discussed.

5.1. Internal Validity

In this paper, I have proposed a new automated approach to evaluating code quality through readily available bug reports. The format of exported bug reports are all in XML. If any other formats are provided, one can refer to the proposed approach presented in the paper to develop his or her programme to adapt to new formats.

5.2. External Validity

I have undertaken case studies on Jira and Bugzilla and the approach has been proved to be effective. To ensure the approach generic, I investigated both Jira and Bugzilla, two well-loved bug tracking systems among world-wild developers. My findings in the experiments might not be generalizable to other bug tracking systems e.g. Mantis, Redmine and Bugtracker which have not been studied sufficiently yet.

6. Conclusions and Future Work

In the paper, I have proposed an automated approach in evaluating code quality by mining bug reports. It is because that few research efforts have been made to study mining bug reports and analysing the results effectively and deeply. Another motivation is that bug reports do contain wealthy software engineering information worthy of being mined for study. The experimental findings and the results analysis clearly demonstrate that the proposed approach is of effectiveness in assessing code quality or at least an estimate of software status. Additionally, this work will facilitate the study of other alternative software repositories e.g. configuration files.

In the future, I intend to support my approach in other formats e.g. CSV and Feed, which will make the approach more adaptable in other scenarios. Furthermore, performance tuning e.g. bug reports files pre-processing will be a future task because of the current modest running programme performance. This further work might deliver better user experience.

References

[1] Wiegers, K. E. (2002). Seven truths about peer reviews.

[2] 14:00-17:00, ISO/IEC 9126: 1991. ISO. Retrieved from: http://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/01/67/16722.html
[3] Yu, L., Ramaswamy, S., & Nair, A. (2013). Using bug reports as a software quality measure.
[4] Kouroshfar, E., Studying the effect of co-change dispersion on software quality. (2013). *Proceedings of the 2013 35th International Conference on Software Engineering*.
[5] Rastkar, S., Murphy, G. C., & Murray, G. (2014). Automatic summarization of bug reports. *IEEE Transactions on Software Engineering*, *40(4)*, 366–380.
[6] Chen, B., Song, J., Xu, P., Hu, X., & Jiang, Z. M. (2018). An automated approach to estimating code coverage measures via execution logs. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, New York, NY, USA.
[7] The road ahead for Mining Software Repositories - IEEE Conference Publication. Retrieved from: https://ieeexplore.ieee.org/abstract/document/4659248
[8] An empirical study of bug report ... preview & related info | Mendeley. Retrieved from: https://www.mendeley.com/catalogue/empirical-study-bug-report-field-reassignment/
[9] Home: Bugzilla: bugzilla.org. Retrieved from: https://www.bugzilla.org/
[10] Software Testing and Continuous Quality Improvement. (2005). Auerbach is an imprint of CRC Press LLC, Boca Raton, Florida, Software Testing, Verification and Reliability - Wiley Online Library.
[11] Jira | Issue & Project Tracking Software | Atlassian. Retrieved from: https://www.atlassian.com/software/jira#
[12] Kanellopoulos, Y., *et al*. (2010). Code quality evaluation methodology using the ISO/IEC 9126 standard.
[13] Microsoft to rate bug severity. *Network Security*, (2001).
[14] Hunt, A., & Thomas, D. (1999). The pragmatic programmer: From journeyman to master.
[15] Maldonado, E. D. S., Abdalkareem, R., Shihab, E., & Serebrenik, A. (2017). An empirical study on the removal of self-admitted technical debt. *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution*.
[16] Lamkanfi, A., & Demeyer, S. (2013). Predicting reassignments of bug reports - An exploratory investigation. *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, Washington, DC, USA.

Yuexiao Teng received his Computer Science B.S. degree in 2009 and Computer Application Technology M.S. degree in 2012 both from East China University of Science and Technology, Shanghai, China. He has several years working experience in information technology industry. Currently, his research interests are software engineering, software testing and data mining etc.