Code quality and mining repositories related to code quality: a systematic review

Jordan McDonald

1. ABSTRACT

VCS (version control systems) are a crucial component of software development, sharing code, raising issues and easy accessibility to code allows remote teams to develop in tandem. Measuring the quality of the code in a repository is an abstract process, as interpretations of what consists as quality code varies from language to language and context. This systematic literature review aims to collate the knowledge regarding this topic and present gaps in the research and summarize the success of other authors along with the metrics used to achieve this. Results suggest that determining code quality is not a predicable process which most of the studies providing only inconclusive outcomes. It was also determined that metrics utilized can be categorized into two sets, high level or low level with the latter focusing on the code level rather than repository statistics. This review hopes to bridge the gaps in each independent paper and suggest ideas from which the research can be improved.

Keywords – Code Quality, Open source, repository

1. BACKGROUND

An Overview of code quality is presented briefly in this section, this shall range from how code quality is defined to identifying select tools utilized to classify code. This information will then be contextualized in terms of repositories and the motivations behind using this technology as a form of storage and version control as an individual or a global software development team. In addition to this it will be briefly surmised how data is obtained from repositories and how these metrics relate to defining the quality of code.

* 1. Code Quality

Code quality in software and how to measure it is often a case of dispute, elements to be considered include how well tested, loosely coupled, efficient and maintainability. Aspects of these points are often very difficult to measure, how would you measure how readable code is without personal opinion interfering is a key example, this is often a subset of good design which has its own abstractions. The technical quality of source code (how well written it is) is an important determinant for software maintainability. When a change is needed in the software, the quality of its source code has an impact on how easy it is: (1) to determine where and how that change can be performed; (2) to implement that change; (3) to avoid unexpected effects of that change; and (4) to validate the changes performed. [2] Measurements may be used to obtain a picture of the quality both of a single component and of an entire program. Typical software metrics are the size of the code (measured in lines of code, number of statements, and so on) and the code complexity (measured through complexity figures such as the cyclomatic complexity). [3]

* 1. Code Quality Tools

There are numerous tools available from which static code can be analyzed, SonarQube is multi language application that provides in depth statistics and visualization. In addition to this SonarQube focuses on potential bugs, complexity measures, duplication and comments which are analyzed against the built in rule sets. Aside from the production level quality of the previous tool there are numerous additional options, some often language specific such as JTest, Clang, Jslint and TOAD.

* 1. Why perform a systematic literature review on this topic?

Research papers on this topic are present with stark degrees of variation, there are multiple papers that tackle the challenge of classifying code in terms of quality which still has merit as a resource for this paper. However there are a reduced number that focus on repositories, which can be classed in two categories – open source or academic, research papers in the context of industry in my search could not be acquired. The goal of my review to fill a gap in the market and draw the results from these sources and present a coherent analysis of the findings in the topic of repository code quality. This should assist in generating more research questions and provide a framework from which prior hypotheses can be compared, in order to provide support or contradict theories.

1. REVIEW METHODS AND CONDUCT

I have devised this system literature review according to the guidelines outlined by Kitchenham and Charters[1]. This paper documents all the steps that have been taken in order to generate the final results, while exploring in depth the intermediary processes which led to my conclusions.

* 1. Research Questions

RQ.1 – What metrics have peer-review literature used to measure software code quality in repositories?

RQ.2 – To what extent are peer reviewed studies successful in using repository data to assess code quality?

* 1. Data sources

The extraction of data related to the topic will be source from digital libraries and databases, to facilitate this process I will browse the websites directly or use portals such as google scholar or the Queens University Belfast library search tools to identify sources of interest. The individual libraries I will be analyzing are listed below.

* <http://ieeexplore.ieee.org>
* <http://dl.acm.org/>
* <http://www.researchgate.net/>
* <http://link.springer.com/>
* <http://flosshub.org/>
  1. Search Strategy and Inclusion Process

Initially a research scope had to be identified, in this case I set a time period restriction of fifteen years (2000-2015). The next step was to designate research questions that would be suitable for the topic, this led to the formulation of keywords extracted from each relevant component of the research questions, synonyms and abbreviations would also be accounted for where appropriate. In addition to this the keyword ‘repository’ was extrapolated to account for popular products such as GitHub and Bit Bucket. At the end of the search phase the results were evaluated and modifications were made to the keywords and Boolean combinations utilized. The final set of keywords and Boolean searches performed are shown below.

Keywords – [GitHub, Bit Bucket, SVN, CVS, repository, code, quality, repositories]

Search Queries – [code AND quality AND repository, repository AND code OR quality, repository AND quality, repository AND bugs]

\*’repository’ has been interchanged with the repository products (not listed above)

* 1. Included/Excluded Studies

Phase one of the exclusion procedure is to A) determine if the paper is written in the English language B) it fits the ten year window previously mentioned and C) the full paper is available for analysis. The collated papers will then be subjected to the second phase which consists of reading the abstract and title for keywords. The final step will then be using my personal opinion to filter the papers down to the required maximum cap (eight papers) via a process of designating each paper as either relevant or irrelevant. This final step will be supplemented by checks such as excluding opinion based papers in order to focus on a quantitative analysis where possible, this phase will be performed on an ad-hoc basis with preference to papers that will contribute to satisfying this literature review.

Phase 2

1. Abstract?
2. Title?

Phase 3

1. Opinion based?
2. Relevant?

Phase 1

1. English
2. Time period
3. Full paper

8

16

11

10

Figure 1 – showing how the papers were filtered

* 1. Data Synthesis

Each paper that passes the filtering process is the subject to a full analysis, selected pieces of information were then stored in Microsoft Excel, which each paper being associated with seven columns, each representing a data category. The goal in this step was to place all the relevant data in a convenient location, this would allow relationships between metrics and sources to be identified much more rapidly. Fields for the Excel form in include:

* Context – Academic, industry or open source.
* Low level metrics – Information drawn from parsing the code base
* High level metrics – Information drawn from the repository logs
* Tool – Whether a tool was utilized
* Languages – Mixed or one particular language
* Outcomes – Conclusions drawn from the paper
* Miscellaneous – Additional details

1. RESULTS
   1. Initial results of peer reviewed literature

The process of selection papers yielded a final count of eight from which to perform a literature review, the years of publication range from 2002 to 2015. Two papers were present in the first half of the original search range, whereas the other six received publication from 2008 to 2014 would could demonstrate a shift in emphasis to exploring repositories and the code within them. This could be particularly due to the popularity surge of Bit Bucket and GitHub, however the data set is too small to provide concreate evidence to support this. Each paper extracted allowed the MS Excel form outlined previously to be fully populated with the data expected in each column. To show the sources of the research papers, refer to figure 2.

|  |  |
| --- | --- |
| **Source** | **Year** |
| IEEE Xplore | 2015 – 2012 - 2015 |
| ACM | 2005 – 2008 - 2014 |
| Research Gate | 2014 |
| Flosshub | 2002 |

Figure 2 – showing data sources and dates

In order to fulfill the requirements of research question one, the types of metrics need to be considered. From this I categorized the data used in each research paper into two categories, high level metrics and low level metrics. Refer to the vectors below for example data that some of the studies used.

High Level = [total commits, total contributors, mean additions, mean deletions, total LOC, total revisions, open bugs in the last six months, commit message parsing, comment quality, unique committers]

Low Level = [number of statements per method, number of methods, cyclomatic complexity, branch counts, readability, path frequency, lack of method cohesion, code clones, number of unconditional jumps]

Low level metrics are those which revolve around analyzing the code level of the repository, from this data such as the amount of functions, while loops and branches in the source code are evaluated. In contrast high level metrics place emphasis on the repository level, using data centered on commits, forks and contributors. These are often collected using tools or API’s, refer to figure 3 below to see the distribution of which repositories use which category of metric.

Figure 3 – shows the rate of metric usage in each paper

Of the eight successful research papers that fit the criteria seven of the studies used multiple programming languages, while the one paper that focused on a singular language, C, also happened to use a tool to perform the code quality analysis, known as Logiscope, the tool focused on low level metrics. In addition there stood variation in the context.

* 1. Results related to both research questions

To answer research question one, I have composed a list of every mentioned metric that is used by the eight papers in order to judge code quality. Each metric has been stated as well as the frequency it has been mentioned in each distinct paper, this will give a broad overview of popular measurements that have been utilized in the studies, refer to figure 4 in the appendix. \*some wording has been modified to allow for a consistent terminology, while filtering out very general terms

* 1. Paper by paper analysis & Contributions to answering research question two

1. Commit Quality in Five High Performance Computing Projects

This papers presents a precise view point towards code quality, honing in on commits and how it relates to quality in a niche HPC project, although non HPC open source projects have been used for comparison. The paper does not directly outline research questions but the main goals they are trying to achieve include designing a case study which compares the two sets of project types with the intention of introducing commit quality measures. For quantifying this goal they used previously defined high level metrics (commit related) and provides variation by utilizing three different repository types, Bit Bucket, SVN and GitHub, however the commit metrics are consistent across each domain. The research results provided no concrete evidence that distinguishes commit quality from each of the two domains, only suggesting that HPC projects need to move onto a more modern VCS.

1. Mining student CVS repositories for performance indicators

The broad goal of the research program was to extract information about student behavior and code from version control repositories, in order to find statistical patterns or predictors of performance. This involved extracting 20 low level metrics as features and then performing analysis on the students final grades, previous grades against the data extracted from CVS. The method is a controlled experiment in an academic setting with over 200 applicants, machine learning algorithms formed the approach taken to classify the feature sets. The study was unable to find any connection between the metrics extracted from CVS and the student’s grades.

1. Code quality analysis in open source software development

This study takes the form of a case study, utilizing a tool (logiscope) in order to determine code quality when applied to 100 Linux open source projects. The student focuses on the language C and the component (method) level with metrics extracted such as number of statements per component and number of comments per component. The tool then classifies the code based on these metrics into five categories that loosely translate from accepted to undefined code. The outcome of the study suggests that code developed in the open source environment is of a lower quality then in the industry and propose suggestions as to why this occurs.

1. Comparing Design and Code Metrics for Software Quality Prediction

The method utilized in this study revolves around machine learning algorithms using WEKA, which a focus on the module (method) level of the code under test. Metric extracted range from the number of parameters in a module, amount of white space and other low level metrics (23 to be exact). The study outcome shows a trend that code only metrics are inferior to when combined with design based metrics, with design focused metrics providing the worst performance. It was also determined that the choice of metrics had a greater impact than the machine learning algorithm implemented.

1. Evaluating the Quality of Open Source Software

The main goal of this research was to extend the development of a code quality tool known as ‘sqo-oss’ however there is an emphasis on discussing the metrics used and the impact on the tool which is of interest in this review. The research extracts its data using a mixed method, low and high level metrics are considered which range from data on commits, to the class level with methods and attributes being placed on scrutiny and measured. The research conclusion support that the paper took and exploratory stance, which the goal being to provide novel ideas, from which the research team hopes to follow up on.

1. A Large Scale Study of Programming Languages and Code Quality in GitHub

The goal of this paper was to determine the effect that a programming language has on open source software quality. The study utilizes a huge code based of 729 projects and 80,000 KLOC, and plans to investigate aspects such as static v dynamic typing and the effect on code quality. The research questions are A) are some languages more defect prone than others and B) which language properties relate to defects and C) which is the relation between language and big category. The study focuses on high level metrics taken from the repository revolving around insertions contributors and commits. The outcomes presented include that the data indicates functional languages are better than procedural languages; it suggests that strong typing is better than weak typing; that static typing is better than dynamic.

1. Measuring Code Quality to Improve Specification Mining

While this topic focuses on the application of specification mining, it provides valuable information on the metrics used for code quality analysis. The method of research is an experiment, via a medium of s code quality tool, which entirely focused on low level metrics. Of the three experiments one is of huge relevance to this review, focusing on a statistical analysis of set code metrics such as code churn in order to predict quality, each point of analysis is considered independently and synthesized. The research concluded the prediction model frequently provided false positives (69%) on certain metrics, which skewed the study’s results. In terms on contribution the study states this greatly exceeds previously developed mining tools, however in the context of this reviews research questions, it reinforces the point that predicting code quality often leads to inconclusive results.

1. GitHub Projects. Quality Analysis of Open-Source Software

The goal of this paper was to classify GitHub projects based on two key metrics, the amount of ‘stargazers’ over time and the survival of issues associated with each project. In addition to this supporting high level metrics were taken into to supplement the main data points, factors including contributors, commits and forks. The research revolves round extracting metrics, from this co-efficient and p-values are defined using a regression analysis model (binomial). The work concluded that it is better for a software project to have focused developers involved in the community rather than having in the team, popular, often followed developers.

1. DISCUSSIONS

In terms of answering research question one I have discovered that there are two discrete categories used by the

Studies, high level and low level metrics. Due to the small data set it is more difficult to draw a correlation on the table of results seen in figure 4, however key performers that apply across multiple papers include LOC (3), cyclomatic complexity (4) and Literal string total (3). Each of these independently don’t indicate code quality, neither does the popularity of the metrics, however in particular cyclomatic complexity and LOC have been prominent references points each paper they are mentioned in. Overall despite the small amount of papers analysed we have identified a set of metrics which satisfies RQ1. Research question two has encountered much more mixed results, 3/8 of the studies provide concreate theoretical outcomes on code quality, however none have a conclusion that directly can identify whether code set A is good OR code set B is bad, the other 5 papers provide inconclusive or no substantial results. Therefore for RQ2 the general consensus of crawling repositories to determine code quality is inconclusive.

The SLR conducted here however contains significant weaknesses, the inability to review a large number of research papers has crippled particular RQ1. This question requires a much larger data set in order to draw substantial conclusions on metrics utilized, however brief, metrics have been identified so despite the limited scope the review was partially successful. A strength of this SLR is the outcomes, from the papers analyzed it has been possible to provide conclusions and contribute to the topic presented in this review. \*see the conclusion section

Aspect of validity need to be considered, external validity has weaknesses in this review. The lack of industrial context in the papers is an aspect of context that is missing, in its place academic, open source and government (NASA) are represented. Construct validity could be criticized, particularly for the metrics, this is a subjective set of data that varied between synonyms (method, module, component) which required translation. In addition to this the research papers chosen often had code quality as a side factor, with another topic taking precedence, it was preferable to have this additional topic to be repository based however not all the papers reflected this which may skew the results. This will have an impact on external validity, since the sources of information are varied, it may be difficult to decipher if the findings will apply to other contexts, particular in industry. It also needs to be considered that all metrics extracted were those explicitly mentioned in the paper, tools might have others hidden under the surface.

1. CONCLUSIONS

This systematic literature review covered eight papers in the topic of ‘code quality and mining repositories related to code quality’ and each have been analyzed and applied to the designated research questions. Research question one was answered by extracting the metrics explicitly mentioned in each paper and frequencies were totaled, this has identified numerous ways code quality can be measured, however the key component to success is the choice and synthesis of these metrics. Research question two examines the current success of predicting code quality using repository mining techniques, in general success has not been consistent with outcomes being entirely inconclusive. In contrast there has been some success which has been detailed in the results section, papers using mixed, high level and low level metrics have merit in achieving this. The review has identified gaps in current studies, particularly there was no paper that focused on an industrial context. The most popular context was open source, which could be a consequence of numerous open source projects on GitHub etc which allow and embrace mining of data. Furthermore only two of the eight studies performed a controlled experiment, which was expected to lead to the clearest results but the outcomes were non-significant.

To combat the threats of validity and gaps in research there is potential to suggest an experiment which may provide an insight between the repository and the quality of the stored code. If possible the experiment should draw repository data from multiple projects (50-100) across all potential contexts – industry, academic and open source. From which a subset of the projects should be randomly chosen to remove potential bias. The experiment should then collect metrics from every repository, these should be both high and low level data points – care should be taken to ensure terminology is consistent and that each metric can be applied to every VCS (version control system). In order to reduce the scope of the experiment it is suggested to focus on a single programming language, this will avoid pitfalls that come with certain languages handling pointers or static v dynamic typing. An alternative to parsing the metrics from the repository individually, an option would be to use a tool (such as SonarQube) which will interpret the code, from this a quality rating can be synthesized. If the selected of metrics is chosen manually, careful thought has to be placed into which data points to select. A strategy I would suggest it to use all potential metrics and assign a weight to those which have greater perceived significance such as a cyclomatic complexity measure.

In summary, the field of researching code quality in repositories has many perils and as of yet has no reliable classifier from which quality van be predicted. The experiment suggested previously offers a method of closing gaps in the research, as well as the results of the SLR which has identifies areas that need to be given greater analysis.

1. REREFENCES
2. B. Kitchenham, S. Charters. Guidelines for Performing Systematic Literature Reviews in Software Engineering, Technical Report EBSE-2007-01, School of Computer Science and Mathematics, Keele University, 2007.
3. Robert Baggen, et al. Standardized code quality benchmarking for improving software maintainability, Springer Science+Business Media, LLC 2011
4. Ioannis Samoladas et al, Open Source Software Development Should Strive for even greater code maintainability, Communications of the ACM Volume 47 Issue 10, October 2004 Pages 83-87
5. Keir Merle et al, Mining student CVS repositories for performance indicators, MSR '05 Proceedings of the 2005 international workshop on Mining software repositories
6. Loannis Stamelos et al, Code quality analysis in open source software development, Information Systems Journal (Impact Factor: 2.07). 01/2002; 12(1):43-60. DOI: 10.1046/j.1365-2575.2002.00117.x
7. Yue Jiang et al, Comparing Design and Code Metrics for Software Quality Prediction, 05/2008; DOI: 10.1145/1370788.1370793
8. Diomidis Spinellis et al, Evaluating the Quality of Open Source Software, Electronic Notes in Theoretical Computer Science 233 (2009) 5–28
9. Baishakhi Ray et al, A Large Scale Study of Programming Languages and Code Quality in Github, FSE 2014 Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering
10. Claire Le Goues and Westley Weimer, Measuring Code Quality to Improve Specification Mining, IEEE transactions on software engineering, vol 38, no 1, January/February, 2012 175
11. Oskar Jarczyk1 et al, GitHub Projects. Quality Analysis of Open-Source Software, Volume 8851 of the series Lecture Notes in Computer Science pp 80-94
12. Agrawal, K. Commit Quality in Five High Performance Computing Projects, Software Engineering for High Performance Computing in Science (SE4HPCS), 2015 IEEE/ACM 1st International Workshop, 2015

Table on Next Page…..

1. APPENDIX

|  |  |  |  |
| --- | --- | --- | --- |
| **High Level Metric** | **Frequency** | **Low Level Metric** | **Frequency** |
| Total commits | 4 | Total times a space followed a comma | 1 |
| Number of files | 1 | Number of comments | 3 |
| Total number of changes | 1 | Literal string total | 1 |
| Total transactions | 1 | Operator total | 1 |
| Average changes per file | 1 | Amount of characters in comments | 1 |
| LOC | 3 | Total function definitions | 1 |
| Cyclomatic complexity | 4 | Total while loops | 1 |
| Number of open bugs (6 months) | 1 | Total for loops | 1 |
| Number of closed bugs (6 months) | 1 | Total termination tokens | 1 |
| Number of unique watchers | 1 | Amount of 4 space indents | 1 |
| Rate of developer turnover | 1 | Assertion message violations | 1 |
| Growth in active developers | 1 | Total leaning tabs | 1 |
| Number of contributors | 3 | Number of statements per method (mean) | 1 |
| Commit message parsing | 1 | Total operators per method |  |
| Total bug fixes | 1 | Maximum nest | 2 |
| Total insertions | 1 | Number of paths per method (mean) | 1 |
| Total number of forks | 1 | Number of unconditional jumps | 2 |
| Current issues | 1 | Comment frequency (over statements) | 2 |
| Release count | 1 | Program length | 2 |
| .. | .. | Average statement size | 1 |
| .. | .. | Number of inputs and outputs of a method | 2 |
| .. | .. | Number of lines per method | 1 |
| .. | .. | Number of public methods | 1 |
| .. | .. | Lack of cohesion in methods | 2 |
| .. | .. | Number of statements | 2 |
| .. | .. | Number of protected methods | 1 |
| .. | .. | Coupling between objects | 2 |
| .. | .. | Depth of inheritance tree | 1 |
| .. | .. | Number of child classes | 1 |
| .. | .. | Code churn | 1 |
| .. | .. | Path frequency | 1 |
| .. | .. | Path density | 1 |

Figure 4 – shows frequency of metric used