Does the Role Matter? An Investigation of the Code Quality of Casual Contributors in GitHub

Yao Lu*, Xinjun Mao*, Zude Li†, Yang Zhang*, Tao Wang* and Gang Yin*
*College of Computer
National University of Defense Technology, Changsha, China
Email: {luyao08, xjmao, yangzhang15, taowang2005, gangyin}@nudt.edu.cn
†School of Information Science and Engineering
Central South University, Changsha, China
Email: zli@csu.edu.cn

***Abstract*—For popular Open Source Software (OSS) projects there are always a large number of worldwide developers who have been glued to making code contributions, while most of these developers play the role of casual contributors due to their very limited code commits (for fixing defects and enhancing features, casually). The frequent turnover of such group of casual developers and the wide variations among their coding experiences challenge the project management on code and quality.**

This paper describes a case study which aims to estimate the quality of code made by casual contributors in 21 popular GitHub projects. The results of this case study show that: (1) casual contributors introduced greater quantity and severity of Code Quality Issues (CQIs) than main contributors; (2) developers who contribute in different projects as main and casual contributors didn't perform statistically differently in terms of code quality; (3) casual contributors who have few project stars introduced more CQIs than those who have many. Furthermore, the paper lists the CQI categories which are most frequently introduced by casual contributors in the investigated projects. These findings provide valuable insights into code quality in the OSS context, and can guide OSS developers in improving the quality of the code contributions.

I. Introduction

One essential distinction for open-source software (OSS) projects from proprietary software projects is that: the internal quality of code plays a more critical role for the project success [1]. On one hand, such code quality (e.g., readability and maintainability) will largely affect the quantity and the quality of the code contributions made later by the developers in the OSS context [2]. On the other hand, open-source code should be 'rigorously modular, self contained and self explanatory', in order to support development at worldwide sites, which forms a general 'criterion' for internal code quality control [3]. Besides, another reason for obtaining high quality code from an open source project is the fact that the next step may be the maintenance of the open product to address vertical marketing requirements [4]. Previous study [5] found that the distribution of code contributions in a OSS project approximately obeys the *Pareto Principle*: 30% of code contributions comes from 70% of the contributors. Typically, these 70% contributors casually submit a very limited number of defect fixes, feature enhancements, etc., and thus we call them *casual contributors*. From the perspective of total quality management for OSS products, it is required to effectively ensure and control the quality of the code made by casual contributors equivalently with that for main contributors, and this might have been more challenging since the quantity and variance of casual contributors is usually considerably larger than that of main contributors. We, therefore, conducted a study to investigate the code quality of casual contributors in OSS communities.

Static code analysis is an important way to assess and maintain software quality, and has become an integral part of the modern software process [6]. For instance, some tools are supported as GitHub Integrations for pull-request workflow to check the quality of submitted patches. Combining with the feature provided by version control system (*e.g.*, the *git blame* command in Git), an issue reported by the static tool can be tracked to its author, which enables us to evaluate the code quality of a developer.

In this study, we use SonarQube—a popular and powerful static analysis tool to analyze the revision history of 21 sampled GitHub projects. We propose a taxonomy for main and casual contributors and evaluate their code quality using the static analysis results. Below are the most noteworthy contributions of this paper:

- We propose a method to estimate developers' code quality using the metric "*Code Quality Issue Density (CQID)*".
- We find that casual contributors tend to introduce greater quantity and severity of CQIs than main contributors.
- We find that developers who contribute in different projects as main and casual roles didn't perform statistically differently in terms of code quality, and casual contributors who have few project stars introduce more CQIs than those who have many.
- We also find that casual contributors tend to introduce CQIs on coding convention, error-handling, CWE (Common Weekness Enumeration) and brain-overload, which can decline readability, reliability, efficiency and testability of the software.

The findings can give insights into the characteristics of code quality in the OSS context, and can also guide internal and external developers in managing code quality.

The rest of the paper is organized as follows. Section II discusses related work. Section III describes the goal and

1530-1362/16 $31.00 © 2016 IEEE
DOI 10.1109/APSEC.2016.44

corresponding research questions of the study. Section IV introduces the methodology of the case study. Section V gives the findings and Section VI analyzes the corresponding implications. Section VII identifies the threats to validity of our study. Finally, Section VIII concludes the paper and discusses the future work.

II. Background & Related Work

A. Code Quality Measurement

Code quality measurement has been discussed in the literature for decades and considerable kinds of metrics have been proposed. Some commonly-accepted ones measure code quality from different views, *e.g.*, complexity metrics (*McCabe's Cyclomatic Complexity*, *Halstead Complexity*, etc.), object-oriented metric sets (*CK* metric set [7]) and code smells (*Duplicated Code*, *Feature Envy*, etc). Some metrics focus on a specific characteristic of software quality: *Maintainability Index* for maintainability, *Test Coverage* for testability. The following part reviews related literature not focusing on a specific part or characteristic, but measuring code quality by a single metric.

Dixon et al. used a single metric fault-proneness to evaluate code quality [8]. A code quality score is determined by the probability that a source file is fault-prone, and the value is scaled to run from 0 to 10. Goues and Weimer [9] used a set of seven metrics (including code churn, author rank, code clones, etc.) as code quality metrics. Incorporating these metrics, they proposed two new specification miners and compared them to previous approaches. Their miner learns more specification and has a lower false positive rate. In order to understand structural quality, Stamelos et al. [4] conducted a case study on 100 applications written for Linux using a measurement tool. They mapped four quality criteria to several metrics, and calculated a final score for each component. They found that the quality of code produced by open source is lower than that which is expected by an industrial standard, and the average component size of an application is negatively related to the user satisfaction for the application. Wong-Mozqueda et al. [10] adopted a set of seven well known metrics as quality indicators and mined the relationships between code quality and test coverage. By performing a correlation analysis on three popular GitHub projects, they found that all of the response variables had modest but significant relationship with line coverage and a stronger relationship with branch coverage.

B. Static Analysis

Static analysis tools look for violations of reasonable or recommended code practice [11], and has become an integral part of the modern software developer's toolbox for assessing and maintaining software quality [6]. To precisely attribute the introduction and elimination of these violations to individual developers, Avgustinov et al. [6] proposed an approach for tracking static analysis violations over the revision history. They performed an experimental study on several large open-source projects, which provided evidence that these fingerprints are well-defined and capture the individual developers' coding habits. Nagappan et al. [12] conducted a case study on the early prediction of pre-release defect density based on the issues found using static analysis tools. They found that static analysis issue density can be used to predict pre-release defect density at significant levels, and can also be used to discriminate between components of high and low quality. Nagappan et al [13] investigated the use of automated inspection for a industrial software system at Nortel Networks. They proposed a defect classification scheme for enumerating the types of defects that can be identified by static tools, and demonstrated that automated code inspection faults can be used as efficient predictors of failures.

C. Human Factor on Software Quality

Previous studies [14–17] have shown that human factors play a significant role in the quality of software components. Nagappan et al. [16] conducted a case study on Windows Vista and provided evidence that the metrics on organizational structure are related to failure-proneness. Bird et al. [18] examined the relationship between ownership measures and software failures in Windows Vista and Windows 7. They found that measures of ownership have a relationship with both pre-release faults and post-release failures: high levels of ownership are associated with less defects. They also found that a developer tends to introduce defects more easily as a minor contributor than as a major contributor. Boh et al. [19] found that project specific expertise has a much larger impact on the time required to perform development tasks than high levels of diverse experience in unrelated projects. Mockus et al. [20] found that changes made by developers who are more familiar with the code are less likely to induce bugs.

III. Research Questions

The main goal of this study is to evaluate the internal quality of code made by casual contributors in OSS projects. More specifically, we structure our goal around several research questions.

Intuitively, because of lack of understanding of the project and diverse array of programming experience among casual contributors, they would contribute lower quality code, compared with the main contributors. Our first question is to verify this hypothesis:

RQ1: Is the quality of the code made by casual contributors lower than that of core contributors?

In OSS communities, prolific developers have social incentives to contribute multiple projects [21]. Since a person's energy is limited and his (her) interest is focused on a few projects, (s)he may contribute different projects as different roles. If the hypothesis in *RQ1* holds, will they contribute lower quality code when playing a casual contributor role? Hence our second research question is thus:

RQ2: When developers contribute multiple projects as different roles (main/casual contributors), do they perform differently in terms of the code quality?

On the basis of the former two research questions, we want to further understand what kinds of *Code Quality Issues*

(CQIs) casual contributors tend to introduce. The answers to such questions can guide core developers in managing code quality from external contributors, and can also guide developers in contributing higher quality code, thus improving the possibility of acceptance of their submitted pull requests. ***RQ3: What categories of CQIs do casual contributors tend to introduce? Which aspects of software quality do these CQIs tend to affect?***

IV. Methodology

A. Data-set and Preprocessing

1) Project selection: In this paper, we select software projects from GitHub platform, which is the most popular code hosting site built on Git version control system [22]. The projects are chosen according to the following principles:

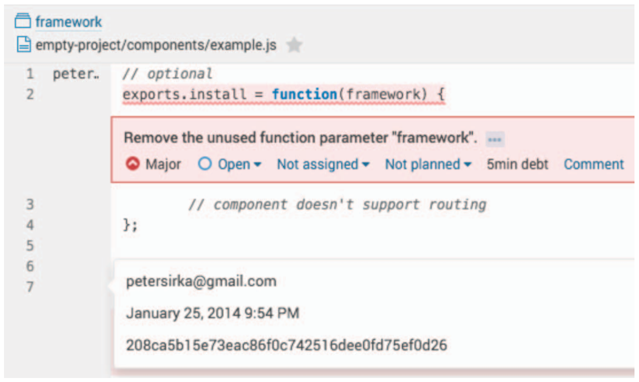
- The languages are Java, JavaScript and Python.
- The project is not forked.
- The star number of the project is in the top 10 of the language in GitHub.
- The number of commits is in the range of 100–20000.
- The project can be correctly analyzed by SonarQube.

As a result, 21 projects are sampled, including 6 for Python, 7 for Java and 8 for JavaScript projects. We cloned the whole git repository of each project, and extracted the corresponding data of commits, participants and stars before March 1st 2016 using GitHub API[1]. The list of sampled projects and codes on data acquisition and processing in this study can be accessed at the GitHub page[2].

2) Code quality analysis: There are a variety of static analysis tools, while most of them focus on a specific field or language. For instance, CheckStyle mainly checks the coding style, while FindBugs aims to help developers find potential defects; PMD only analyzes Java source files while Oink is for C++ projects. In this paper, we select SonarQube[3] platform– a powerful tool to perform static analysis, which supports more than 20 code languages and covers 7 axes of code quality: *architecture & design, duplications, unit tests, complexity, potential bugs, coding rules* and *comments*. It is a web-based application and provides a powerful plug-in mechanism to support users to add new languages, rules and integrations. Besides, some superb features (*e.g.*, TimeMachine, Technical Debt, Quality Issue Tracking, etc.) make users manage code quality more efficiently.

After installing the Git plugin, SonarQube can automatically detect the introduced commit of the CQI using *git blame* command and display the relevant information on the source code view (as shown in Figure 1). These plugin data can be accessed through web service API, and SonarQube also stores the git email of the author of the CQI in the MySQL database, which facilitates us to get the CQIs introduced by a contributor.

[1]https://developer.github.com/v3/
[2]https://github.com/roadfar/CasualContributor
[3]http://www.sonarqube.org/



framework
empty-project/components/example.js

```
peter.  // optional
        exports.install = function(framework) {
```

Remove the unused function parameter "framework".
Major Open Not assigned Not planned 5min debt Comment

```
            // component doesn't support routing
        };
```

petersirka@gmail.com
January 25, 2014 9:54 PM
208ca5b15e73eac86f0c742516dee0fd75ef0d26

Fig. 1. Screenshot of SonarQube Issue Page

In order to get the CQIs introduced by all contributors in a project, we need to scan its whole revision history. We first wrote a simple algorithm to analyze every revision for a project, and found that for the projects whose commit number is more than 5000, the scanning time can be more than one day on a Quad Core i7 processor, 16GB memory machine. So we optimized the algorithm, only scanning the commits that need to be scanned. We sorted all the commits of a project by the commit time: C_1, C_2, ..., C_n, and defined the commits need to be analyzed: NC_1, NC_2, ..., NC_k ($k \leq n$). NC_i is defined in a recursive way below:

$$NC_i = \begin{cases} C_i & \text{i = 1} \\ \bigcap_{j=1}^{k} f(C_j) \cap f(NC_{i-1}) = \emptyset, & \\ \bigcap_{j=1}^{k} f(C_j) \cap f(NC_{i-1}) \cap f(NC_i) \neq \emptyset & \text{i > 1} \end{cases}$$

$f(C_i)$ is the set of changed files of C_i, and $C_j, C_{j+1}, \ldots, C_k$ is the set of commits between NC_{i-1} and NC_i. An example is shown in Figure 2: if C_1 is a commit need to be analyzed,

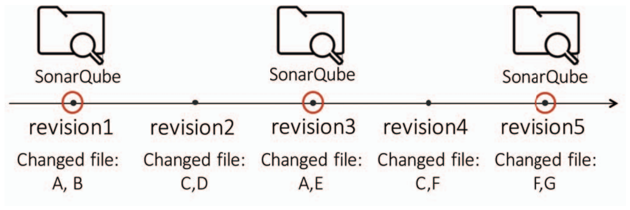


Fig. 2. Optimization of the Scanning Algorithm

C_3 and C_1, C_5 and C_4 have common changed files, therefore C_3 and C_5 are the commits need to be analyzed.

B. Definitions of Core Terms

Unlike some previous literatures, we define the *contributors* as those who have made technical contributions to a project:

- ***Contributor*** — a developer in the OSS context who has made at least one commit to the project, not including the commits whose pull requests are rejected.

Some literatures have used similar or opposite definitions of the term *Casual Contributor*. Zhou and Mockus [23] defined

Long Term Contributor to be a contributor who stays with the project for at least three years and who has productivity exceeding 10th percentile among the participants with a tenure exceeding three years. To measure the ownership of a component, Bird et al. [18] examined the distribution of the of ownership, and defined *Minor Contributor* whose contribution percentage to a component is below 5%.

In this study, we refer to some definitions in [18], and define casual contributors as those whose contribution percentage is below 5%. We adopted the number of commits to a project to evaluate a developer's contribution. Since the number of commits of different projects are different, we calculated a unified threshold on the whole data set and the value is 5. Therefore, the core terms used throughout this paper are defined below:

- ***Casual Contributor*** — a contributor in the OSS context whose commit number to a project is below 5.
- ***Main Contributor*** — a contributor in the OSS context whose commit number to a project is at or above 5.

As a result, there are 1596 casual contributors and 367 main contributors in our data-set, and about 81 percent of the developers are casual contributors.

C. Developers' Code Quality Measurement

Traditionally, software quality has been decomposed into internal and external quality attributes [1]. External quality attributes are often reflected at runtime stage, *e.g.*, functionality, usability, correctness, etc., which can be perceived by users. An important and commonly-used measure for external quality is *defect* [24]. Correspondingly, internal quality attributes are often reflected at development and maintenance stage, *e.g.*, maintainability, readability, security, etc, which are more concerned by developers. For poor internal quality attributes, there are some commonly-accepted patterns, which can decrease its sub-characteristics. For example, over-complexity of methods and less comments affect maintainability and readability; unused parameters and duplications may cause security and reliability problems. Both external and internal quality are critical in a software project [1]. In this paper, we focus on a static view of the software, considering its internal quality from the point of view of developer, since we think it can reflect developers' code quality.

1) Code Quality Issues: While running an analysis, SonarQube raises an issue when a piece of code breaks a coding rule, and stores it in the MySQL database. The set of coding rules is defined through the quality profile associated with the project, and developers can also manually create rules. To distinguish issues in SonarQube from the issues in issue-tracking systems, we define issues analyzed by SonarQube as *Code Quality Issues (CQIs)*. Being similar with defects, each CQI in SonarQube has one of five severities:

- ***BLOCKER*** — Bug with a high probability to impact the behavior of the application in production, *e.g.*, memory leak, unclosed JDBC connection, etc. The code must be immediately fixed.
- ***CRITICAL*** — Either a bug with a low probability to impact the behavior of the application in production or an issue which represents a security flaw, *e.g.*, empty catch block, SQL injection, etc. The code must be immediately reviewed.
- ***MAJOR*** — Quality flaw which can highly impact the developer productivity, *e.g.*, uncovered piece of code, duplicated blocks, unused parameters, etc.
- ***MINOR*** — Quality flaw which can slightly impact the developer productivity, *e.g.*, lines should not be too long, "switch" statements should have at least 3 cases, etc.
- ***INFO*** — Neither a bug nor a quality flaw, just a finding.

2) CQID metric for developers' code quality: We use the *Code Quality Issue Density (CQID)* — number of introduced CQIs per changed code line to assess a developer's code quality in a project, without considering difference among CQIs' severities. We calculated the number of changed code lines of a user using the *git log* command, counting the added and changed lines [25].

V. Findings

A. RQ1: Code Quality of Casual Contributors

To examine the code quality between main and casual contributors, we first compared the average CQID of the two groups at different severities. As shown in Figure 3, except for the *info* severity at which both main and casual contributors have introduced few CQIs, casual contributors introduce more CQIs than main contributors averagely. Especially for higher severity CQIs such as *blocker* and *critical*, the average of CQID introduced by casual contributors are over three times more than that of main contributors. This preliminary result motivated us to make an in-depth analysis.

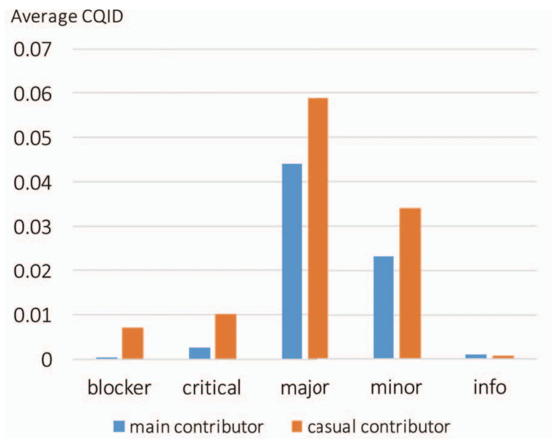


Fig. 3. Average CQID introduced by main and casual contributors

Since the distributions of introduced CQID of main and casual contributors do not follow normal distribution, we use the non parametric *Wilcoxon-Mann-Whitney (WMW) test* to examine the significance of difference between the two groups. In this study, we use *SPSS* to do statistical analysis, and the significance level is set at $\alpha = 0.05$. The *WMW test* shows that the p value is extremely low (<0.001). So we reject the hypothesis that the introduced CQID between the two groups are identical, and reach a finding that:

Finding 1: In OSS communities, casual contributors tend to introduce more CQIs than main contributors. Especially, casual contributors tend to introduce more high-severity CQIs than main contributors.

B. RQ2: Code Quality of a Developer as Different Roles

In our data set, we observed that there are a small group of developers who have contributed multiple projects, and some of them have contributed different projects as different roles. This motivated us to examine whether they perform differently in terms of their code quality when acting main and casual contributor roles. We picked out 29 developers who satisfy our condition. Note that if a developer contributes more than two projects as main and casual contributor roles, we made multiple pairs respectively. For instance, *artem-zinnatullin* submitted 25 commits to *RxJava* project as a main contributor, while having 3 and 4 commits to *retrofit* and *okhttp* as a casual contributor. So *RxJava-retrofit* and *RxJava-okhttp* are the two pairs formed in the example. Consequently, 37 pairs are formed. Since the sample size is small, we use the *paired-sample t-test* to examine the significance of difference of CQID between the pairs. The results are shown in TABLE I. We can see that there is little correlation between the introduced CQID when developers act different roles contributing different projects, and the results for paired samples test show the difference is not significant (p = 0.168).

TABLE I
RESULTS FOR PAIRED-SAMPLE T TEST

	Paired Samples Statistics			
	N	**mean**	**Std.Error Mean**	
main	37	.042	.011	
casual	37	.110	.048	
	Paired Samples Correlations			
	Correlation		Sig.	
main & casual	.034		.842	
	Paired Samples Test			
	Paired Differences		**t**	**Sig.**
	Mean	**Std. Error Mean**		
main - casual	-.069	.049	-1.405	.168

Further, we observed the star numbers of these contributors' projects are relatively high, which represent high popularity of their projects. Combined with *Finding 1*, the result for *Finding 2* raised us a question that whether the CQIs introduced by casual contributors are mainly attributed to the developers who have less project stars. Hence, we made an extensive analysis on the difference of project stars between the high CQID and low CQID group among casual contributors. We sorted the data of CQID introduced by casual contributors and divided it into four quarters. *Independent-sample t-test* was used to examine the significance of difference of star number between the first quarter and the last quarter. The results show that the mean value for high CQID and low CQID group are 123.59 and 339.87, respectively, and the p values for *f-test* (<0.001) and *t-test* (0.049) are both below 0.05. Therefore, we can make a reference that:

Finding 2: In the OSS context, developers don't perform differently in terms of code quality when acting main and casual contributor roles, and developers having fewer project stars tend to introduce more CQIs than the ones having more.

C. RQ3: Categories of Casual Contributors' CQIs

A CQI raises when a component breaks a rule, and conversely a CQI corresponds to a rule. By default, SonarQube sets a characteristic of software quality and one or more tags for each rule. Therefore, a CQI corresponds to a characteristic and multiple tags. In order to understand what CQIs casual contributors frequently introduce, we firstly classified the CQIs using the default taxonomy. If a CQI has multiple tags, it is classified to each category respectively. Figure 4 shows the top

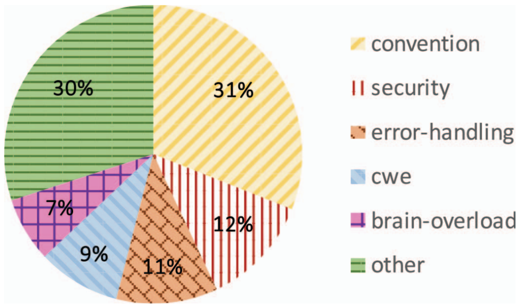


Fig. 4. Classification of CQIs

5 categories of CQIs, and the interpretation for each category is listed below:

- ***Convention*** — coding convention, typically formatting, naming, whitespace, etc.
- ***Security*** — relates to the security of an application.
- ***Error-handing*** — improper handle on exception.
- ***CWE*** — Common Weakness Enumeration, CWE is a community-developed dictionary of software weakness types, which provides a unified, measurable set of software weaknesses related to architecture and design[4].
- ***Brain-overload*** — there is too much to keep in programmers' head at one time.

In our data set, we find that most of the CQIs tagged with *error-handling* are tagged with *security* as well: 93% CQIs on *security* are tagged with *error-handling*, and 99% CQIs on *error-handling* are tagged with *security*. This suggests that a big part of the CQIs in the two categories are the same ones, calculated by our method. Therefore, we can summarize from

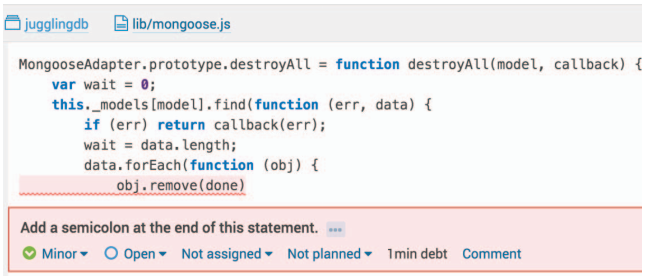
[4]http://cwe.mitre.org/

TABLE II
MOST EASILY BROKEN RULES BY CASUAL CONTRIBUTORS

Category	Name	Broken times	Percentage[1]	Severity	Language
Convention	Each statement should end with a semicolon	73122	39.6%	minor	JavaScript
	Statements should be on separate lines	31928	6.2%	minor	Java
	Method names should comply with a naming convention	21373	23.4%	minor	Python
Error-handling	Generic exceptions should never be thrown	87515	16.9%	major	Java
	Exception handlers should preserve the original exception	31876	6.2%	critical	Java
	Throwable and Error should not be caught	12359	2.4%	blocker	Java
CWE	Fields in a "Serializable" class should either be transient or serializable	4872	1.0%	critical	Java
	Class variable fields should not have public accessibility	4536	0.9%	major	Java
	Dead stores should be removed	2630	0.5%	major	Java
Brain-overload	Functions should not be too complex	16661	9.0%	major	JavaScript
	Methods should not be too complex	13092	7.1%	major	Java
	Functions should not be too complex	7158	7.8%	major	Python

[1] The percentage value is the ratio between the broken times and the total number of CQIs of corresponding language

Figure 4 that over 50% CQIs introduced by casual contributors are on coding convention and exception handling. Further, we list the top three rules of the four categories which are the most easily broken by casual contributors in TABLE II. Generally, coding convention are less strict rules to obey, compared with other higher severity rules, such as rules on *exception-handling*. OSS communities usually stipulate their own coding convention to unify the code style. To examine whether the CQIs on *convention* are in conformity to the projects' coding style, we randomly chose 50 CQIs and found that 44 are not. An example on adding semicolons at the end of statements in JavaScript is shown in Figure 5.



```
MongooseAdapter.prototype.destroyAll = function destroyAll(model, callback) {
    var wait = 0;
    this._models[model].find(function (err, data) {
        if (err) return callback(err);
        wait = data.length;
        data.forEach(function (obj) {
            obj.remove(done)
```

Add a semicolon at the end of this statement.

Fig. 5. An example of CQI on Coding Convention

The characteristics are evaluated by the SQALE[5] (Software Quality Assessment based on Lifecycle Expectations) methodology. In SQALE Quality Model, the priority of characteristics are: Testability > Reliability > Changeability > Efficiency > Security > Maintainability > Portability > Reusability, which means that an app should be testable first, then should be reliable, then changeable, etc. Figure 6 shows the top-4 characteristics influenced by casual contributors. We can see that casual contributors mainly introduce quality issues on software's readability, security, reliability, efficiency and testability, and the number of issues increases as the

[5]http://www.sqale.org/

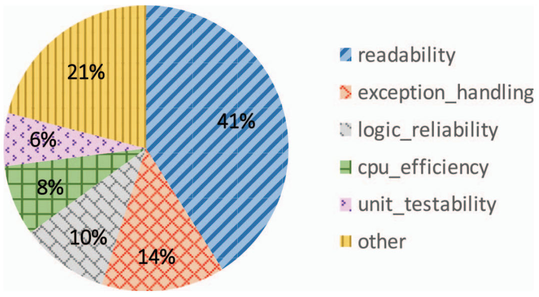


Fig. 6. Influenced characteristics

priority of characteristics drop. Consequently, we can make a summary that:

Finding 3: In the OSS context, casual contributors tend to introduce CQIs on convention, security, error-handling, CWE and brain-overload, which can decline testability, efficiency, reliability and readability of the software.

VI. IMPLICATIONS

Based on the findings above, this section discusses the implications of managing code quality from the view of core developers and contributing high-quality code from the view of casual developers.

A. *Managing Code Quality*

Based on *Finding 1* and *Finding 2*, we can infer that core developers should be more aware of the pull requests from newcomers, especially for those who have fewer project stars and less experience contributing other projects. Besides checking whether functional requirements are satisfied, *e.g.* a defect-fixing or a feature enhancement, they should be aware of the influence of changed code on quality as well. At this point, core developers can adopt the *Continuous Inspection*[6]

[6]http://www.sonarsource.com/products/features/continuous-inspection/

process to automatically analyze the quality of the submitted patches. *Continuous Inspection* is a paradigm that provides continuous code quality management–incorporating shorter feedback loops to ensure rapid resolution of quality issues. GitHub users can deploy *Continuous Inspection* tools into pull-request workflow, which are integrated into the *Continuous Integration* process, and perform quality analysis on the merged code after the passing unit testing. If the severities of CQIs of the pull request are high, e.g. *critical* or *blocker*, the pull request will be rejected automatically. This process helps finding quality problems early when fixing them is still cheap and easy, and can also educate developers instantly when code is still fresh in their mind.

B. Contributing High-quality Code

Newcomers are critical for OSS communities to retain sustainable. However, newcomers often face difficulties and obstacles when onboarding to a project [26]. Demonstrated skill level is an essential factor for the acceptance by communities, thus influencing newcomers' continued permanence [27]. Among the casual contributors in the study, though not affirmed, we believe the quality of developers' patches influence their chances becoming a long-term contributor. Therefore, newcomers should be more aware of the quality of their initial submissions. We can see from *Finding 3* that the most issues introduced by casual contributors are on coding convention. Generally, OSS projects would publish their coding conventions on the contribution page. Some are stipulated by the communities, *e.g., Ruby on Rails*, while some adopt the official style guide, *e.g., httpie*. So newcomers should read and follow the coding conventions presented in contributing page before submitting. Another implication for contributors is to keep a global awareness on structure quality when making changes to a piece of code. For example, we found some developers forget deleting unused parameters or leading to increase of cyclomatic complexity after performing changes to methods. These issues cannot be recognized by compiling or unit testing, but can reduce the code quality. Therefore, contributors especially for newcomers who are not familiar with the project, should keep an eye on a wider scope of code piece when performing changes.

VII. Threats to Validity

This section discusses the threats to validity that may have influenced the study. The three subsections present threats to internal, external and construct validity.

A. Internal Validity

The validity of results in this study is built on the validity of the tool we used—SonarQube. The concepts and taxonomy we used are all default values set by SonarSource. The characteristic model are based on the SQALE methodology, which is a public methodology to support the evaluation of a software application's source code in the most objective, accurate, reproducible and automated possible way [28]. A common issue of static analysis tools is the false positives [29], and we also find some cases in our study. For instance, we found that SonarQube analyzed *naming conventions* for the files auto-generated by the development tools as well, e.g., R file is a resource file generated by Android Development Toolkit, and the naming style of variables (e.g., "m_text") is inconsistent with Java convention (e.g., "mText"). As a result, SonarQube reported CQIs on *Convention* at almost every line of code in the R file, which are also involved in our data set.

B. External Validity

To ensure the size of the data-set, we sampled the projects that are the most popular projects with a large number of stars. For the projects with less stars, further verification need to be made. Besides, the study only concerned the Python, Java and JavaScript projects, other popular language such as C++ and Ruby are not covered. Another risk to external validity is the data used when analyzing ***RQ2***. We observed that all the developers contributing different projects as different roles have a large number of project stars and followers, which reflect their social status and technical level. So the conclusion may not hold for the developers with lower social status and technical experience.

C. Construct Validity

Referencing the commonly-used code quality metric: defect density, we measure developers' code quality by the density of CQIs, without considering the severity of CQIs. However, as [30] mentioned, not all quality issues are equally important in a given context. In our own experience, many *info* and *minor* level CQIs can be ignored, as the case might be. So as a supplement, we compared the CQID between main and casual contributors at different severity levels, and the result enhanced our finding. Besides, we believe that this threat can be alleviated in the big data context.

VIII. Conclusion and Future Work

The level of internal quality that a software product has today will determine the level of its cost liability tomorrow. In this study, we investigated the quality of code made by casual contributors in 21 popular OSS projects within GitHub, and obtained some valuable findings. For example, our study reveals that casual contributors, especially for those who have less project stars introduce more CQIs than main contributors, and tend to introduce more high-severity CQIs. We also find that when developers contribute different projects as different roles, they don't perform differently in terms of code quality. Categories and rules are presented to identify what kinds of CQIs casual contributors frequently introduce. Finally, some advices are given to help core developers manage code quality and casual developers contribute high-quality code. We believe that our findings and implications provide valuable guidelines for quality management in the OSS context.

In terms of future work, we will scale up this study in two dimensions. First, duplicate the case study on more projects written by other languages, *e.g.*, C++ and Ruby. Second, make a survey to deeply understand casual contributors' psychology and behavior. Besides, we will investigate the influence of *Continuous Inspection* on the software process.

Acknowledgment

We gratefully acknowledge the financial support from Natural Science Foundation of China under Grant No. 61472430, 61502512, 61532004 and 61379051. We thank our postgraduate students Xiang Hou and Yiang Gan for their help with the experiments, and also want to thank SonarSource for their kind help on the tool use.

References

[1] P. Tonella and S. L. Abebe, "Code quality from the programmer's perspective," *Acat*, 2008.

[2] N. B. Kshetri, P. B. Palvia, and R. B. Singh, "Improving open source software maintenance," *Journal of Computer Information Systems*, vol. 50, no. 50, pp. 81–90, 2010.

[3] T. Bollinger, R. Nelson, K. M. Self, and S. J. Turnbull, "Open-source methods: Peering through the clutter," *IEEE Software*, vol. 16, no. 4, pp. 8–11, 1999.

[4] S. Ioannis, A. Lefteris, O. Apostolos, and G. L. Bleris, "Code quality analysis in open source software development." *Information Systems Journal*, vol. 12, no. 1, pp. 43–60, 2002.

[5] G. Gousios, E. Kalliamvakou, and D. Spinellis, "Measuring developer contribution from software repository data." in *Proc. the Int'l Working Conf. on Mining Software Repositories*, 2009, pp. 129–132.

[6] P. Avgustinov, A. I. Baars, A. S. Henriksen, and G. Lavender, "Tracking static analysis violations over time to capture developer characteristics," in *Proc. the IEEE Int'l Conf. on Software Engineering*, 2015, pp. 437–447.

[7] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. on Software Engineering*, vol. 20, no. 6, pp. 476 – 493, 1994.

[8] B. M. Dixon, "An objective measure of code quality," 2010.

[9] C. L. Goues and W. Weimer, "Measuring code quality to improve specification mining," *IEEE Trans. on Software Engineering*, vol. 38, no. 1, pp. 175–190, 2012.

[10] J. A. Wong-Mozqueda, R. Haines, and C. Jay, "Is code quality related to test coverage?" in *Int'l Workshop on Sustainable Software Systems Engineering*, 2015.

[11] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE Software*, vol. 25, no. 25, pp. 22–29, 2008.

[12] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Int'l Conf. on Software Engineering, 2005.*, 2005, pp. 580 – 586.

[13] N. Nagappan, L. Williams, J. Hudepohl, W. Snipes, and M. Vouk, "Preliminary results on using static analysis tools for software inspection," *Proc. Fifteenth IEEE Int'l Symp. on Reliability Engineering*, pp. 429–439, 2004.

[14] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu, "Putting it all together: using socio-technical networks to predict failures," in *Proc. 20th Int'l Symp. on Software Reliability Engineering,*, 2009, pp. 109–119.

[15] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley, "Identification of coordination requirements: implications for the design of collaboration and awareness tools," in *Proc. of the 20th Anniversary Conf. on Computer supported cooperative work.* ACM, 2006, pp. 353–362.

[16] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality: an empirical case study," in *Proc. of the 30th Int'l Conf. on Software engineering*. ACM, 2008, pp. 521–530.

[17] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?" in *Proc. of the 16th ACM SIGSOFT Int'l Symp. on Foundations of software engineering*. ACM, 2008, pp. 2–12.

[18] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: examining the effects of ownership on software quality," in *Proc. of the 19th ACM SIGSOFT Symp. and the 13th European Conf. on Foundations of software engineering*. ACM, 2011, pp. 4–14.

[19] W. F. Boh and J. A. Espinosa, "Learning from experience in software development: A multilevel analysis." *Management Science*, vol. 53, no. 8, pp. 1315–1331, 2007.

[20] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[21] B. Vasilescu, K. Blincoe, X. Qi, C. Casalnuovo, D. Damian, P. Devanbu, and V. Filkov, "The sky is not the limit: multitasking across github projects," in *Proc. the 38th Int'l Conf. on Software Engineering*, ACM, Ed., 2016, pp. 994–1005.

[22] Y. Yu, H. Wang, G. Yin, and T. Wang, "Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment?" *Information and Software Technology*, vol. 74, no. C, pp. 204–218, 2016.

[23] M. Zhou and A. Mockus, "Who will stay in the floss community? modeling participant's initial behavior," *Proc. IEEE Trans. on Software Engineering*, vol. 41, no. 1, pp. 82–99, 2015.

[24] H. Zhang and M. A. Babar, "Systematic reviews in software engineering: An empirical investigation," *Information and Software Technology*, vol. 55, no. 7, pp. 1341–1354, 2013.

[25] Y. Lu, X. Mao, and Z. Li, "Assessing software maintainability based on class diagram design: A preliminary case study," *Lecture Notes on Software Engineering*, vol. 4, no. 1, pp. 53–58, 2016.

[26] B. Dagenais, H. Ossher, R. K. E. Bellamy, and M. P. Robillard, "Moving into a new software project landscape," in *IEEE Int'l Conf. on Software Engineering*, 2010, pp. 275–284.

[27] A. Schilling, S. Laumer, and T. Weitzel, "Who will remain? an evaluation of actual person-job and person-team fit to predict developer retention in floss projects," in *Hawaii Int'l Conf. on System Sciences*, 2012, pp. 3446–3455.

[28] J. L. Letouzey, "The sqale method definition document," in *3rd Int'l Workshop on Managing Technical Debt (MTD)*, 2012, pp. 31 – 36.

[29] H. Rocha, M. T. Valente, H. Maques-Neto, and G. Murphy, "An empirical study on recommendations of similar bugs," in *Int'l Conf. on Software Analysis, Evolution and Reengineering*, 2016.

[30] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE Trans. on Software Engineering*, vol. 32, no. 4, pp. 240–253, 2006.