Internal quality assurance for external contributions in GitHub: An empirical investigation

Article *in* Journal of Software: Evolution and Process · November 2017
DOI: 10.1002/smr.1918

6 authors, including:

Yao Lu
National University of Defense Technology
8 PUBLICATIONS **5** CITATIONS
SEE PROFILE

Xinjun Mao
National University of Defense Technology
123 PUBLICATIONS **255** CITATIONS
SEE PROFILE

Yang Zhang
National University of Defense Technology
14 PUBLICATIONS **31** CITATIONS
SEE PROFILE

Tao Wang
National University of Defense Technology
48 PUBLICATIONS **156** CITATIONS
SEE PROFILE

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2017; **00**:1–20
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

Internal Quality Assurance for External Contributions in GitHub: an Empirical Investigation*

Yao Lu[1]*, Xinjun Mao[1], Zude Li[2], Yang Zhang[1], Tao Wang[1] and Gang Yin[1]

[1] *College of Computer, National University of Defense Technology, Changsha, China*
[2] *School of Information Science and Engineering, Central South University, Changsha, China*

SUMMARY

For popular Open Source Software (OSS) projects there is 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 casually fixing defects and enhancing features). The frequent turnover of such a group of developers and the wide variations in their coding experiences challenge the project management on code and quality. This paper aims to investigate the status quo of internal quality assurance for external contributions in social coding sites (*e.g.*, GitHub). We first conducted a case study of 21 popular GitHub projects to estimate the code quality of the casual contributors. The quantitative results show that: (1) the casual contributors introduced greater quantity and severity of Code Quality Issues (CQIs) than the main contributors; (2) the developers who contribute to different projects as main and casual contributors did not perform significantly differently in terms of their code quality; (3) the casual contributors who have few project stars tended to introduce more CQIs than those who have many. Based on these findings, we further conducted a survey of 81 developers on GitHub, to understand the integrators' and contributors' actual practices in terms of internal quality assurance. The qualitative results expose some limitations of present internal quality control for external contributions in GitHub. Finally, in consideration of these limitations, we discuss an alternative quality management paradigm: Continuous Inspection for industrial practices. Copyright © 2017 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: internal quality; GitHub; casual contributor

1. 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 in the project's success [1]. On one hand, such code quality (e.g., readability and maintainability) will largely affect the quantity and quality of the code contributions made later by the developers in the OSS context [2, 3, 4]. 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 [5]. Another reason for obtaining high quality code from an open source project

*This paper is a an enhanced version of paper (Lu et al. "Does the Role Matter? An Investigation of Casual Contributors Code Quality in GitHub", 23rd Asia-Pacific Software Engineering Conference, APSEC 2016). In the APSEC'16 paper, we conducted a quantitative analysis of code quality between casual contributors and main contributors in GitHub. Based on the findings, in this enhanced submission, we conducted an online survey to deeply understand the developers work practices and encountered challenges in terms of ensuring the internal quality of contributions. This research is supported by research grants from Natural Science Foundation of China under Grant No. 61472430, 61502512, 61532004 and 61379051.

*Correspondence to: College of Computer, National University of Defense Technology, Sanyi Road, Kaifu District, Changsha, China.

is the fact that the next step could be the maintenance of an open product to address the vertical marketing requirements [6]. A previous study [7] found that the distribution of code contributions in a OSS project approximately obeys the *Pareto Principle*: 30% of the code contributions comes from 70% of the contributors. Typically, these 70% contributors casually submit a very limited number of defect fixes, feature enhancements, etc.; thus we call them *casual contributors*. From the perspective of total quality management for the OSS products, it is required to effectively ensure and control the quality of the code made by casual contributors in a manner that is equivalently to that for the main contributors, and this process might be more challenging because the quantity and variance of casual contributors is usually considerably greater than that of main contributors. Therefore, we conducted a case study to investigate the code quality of casual contributors in the OSS communities.

Static code analysis is an important way to assess and maintain software internal quality, and has become an integral part of the modern software process [8]. For example, some of the tools are supported as GitHub Integrations for pull-request workflows, to check the quality of the submitted patches. Combined with the features provided by the 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, which is 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 density of introduced quality issues. Based on the quantitative findings, we further conduct a survey to understand integrators' and contributors' work practices and encountered challenges in terms of internal quality assurance. The following are some of the most noteworthy contributions of this paper:

- We propose a method to estimate developers' code quality using the "*Code Quality Issue Density (CQID)*" metric.
- We find that casual contributors tend to introduce a greater quantity of CQIs with greater severity than the main contributors. Nevertheless, when the developers contribute to different projects in main and casual roles, they did not perform significantly differently in terms of the code quality.
- We find that casual contributors tend to introduce CQIs on coding convention, error-handling, CWE (Common Weekness Enumeration) and brain-overload, which can decline the readability, reliability, efficiency and testability of the software.
- We obtain insights into the integrators' and contributors' work practices and encountered issues in terms of internal quality management, and we discuss *Continuous Inspection* as an alternative quality assurance model for social coding sites.

These findings can provide insights into the characteristics of the code quality in the OSS context, and can also guide the internal and external developers in managing code quality. We believe that this work can contribute to theory building by providing empirical evidence about the common practices of quality management in OSS communities.

The remainder of this paper is organized as follows. Section 2 discusses the related work. Section 3 introduces the research questions and methodology of the study. Section 4 gives the quantitative and qualitative findings for the research questions, and Section 5 discusses the implications based on the findings. Section 6 identifies the threats to the validity of our study. Finally, Section 7 concludes the paper and discusses our proposals for the future work.

2. BACKGROUND AND RELATED WORK

2.1. Code Quality Measurement

Code quality measurement has been discussed for a long time and many types of metrics have been proposed. Some commonly accepted ones measure the code quality from different views, *e.g.*, complexity metrics (*McCabe's Cyclomatic Complexity*, *Halstead Complexity*, etc.), object-oriented metric sets (*CK* metric set [9]) and code smells (*Duplicated Code*, *Feature Envy*, etc)[10].

DOI: 10.1002/smr

Some metrics focus on a specific characteristic of software quality: the *Maintainability Index* for maintainability, and the *Test Coverage* for testability. The following section reviews related literature and does not focus on a specific part or characteristic; instead, it focusses on measuring the code quality by a single metric.

Dixon et al. used a single metric fault-proneness to evaluate code quality [11]. 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 [12] 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. To understand the structural quality, Stamelos et al. [6] conducted a case study on 100 applications that were 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 the quality that 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. [13] adopted a set of seven well known metrics as quality indicators and mined the relationship between code quality and test coverage. Performing a correlation analysis on three popular GitHub projects, they found that all of the response variables had modest but significant relationship with the line coverage and a stronger relationship with the branch coverage. To deriving meaningful metric thresholds for the effective use of software metrics, Alves et al. [14] designed a method that determines metric thresholds empirically from measurement data. Their method respects the distributions and scales of source code metrics, and is resilient against outliers in metric values or system size. They applied the method to a benchmark of 100 object-oriented software systems, both proprietary and open-source, to derive thresholds for metrics included in the SIG maintainability model.

In these studies, the internal code quality is measured by a suite of metrics. Hence, the quality of code is determined by the values of the metrics and the corresponding thresholds, *i.e.*, low-quality code is indicated when the metric values are beyond or below certain thresholds. In our study, we refer to the metric: *defect density* that measures the external quality of software, and use *code quality issue density* to measure the internal quality of code. A code quality issue is a violation in code that can decrease the internal quality attributes (such as readability, maintainability and security) of code, and is usually analyzed by static analysis tools. Some of the code quality issues are reported because the metric values are beyond or below certain thresholds, *e.g.*, a method is reported an issue on complexity when its cyclomatic complexity exceeds the threshold value.

2.2. *Static Analysis and Internal Quality*

Static analysis tools look for violations of reasonable or recommended code practice [15], and they have become an integral part of the modern software developer's toolbox for assessing and maintaining software quality [8]. To precisely attribute the introduction and elimination of these violations to individual developers, Avgustinov et al. [8] 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. [16] 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. [17] 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.

Baca et al. [18] conducted a case study to evaluate static code analysis in industry on defect detection capability, deployment and usage of automated code analysis. They found that the tool is capable of detecting memory-related vulnerabilities, but few vulnerabilities of other types. The deployment of the tool played an important role in its success as an early vulnerability detector.

DOI: 10.1002/smr

They also found that the correction of false positives in some cases created new vulnerabilities in previously safe code, and that, in addition, the tool should be integrated with bug reporting systems, and developers should share the responsibility for classifying and reporting warnings. To reduce the fault report rate of static code analysis, Szer [19] proposed an approach and a toolchain for integrating static analysis and runtime verification. They utilized the static analysis results to generate runtime verification specifications, and used runtime verification results to eliminate false positives of the static analysis tools. Yamashita [20] et al. replicated a technique for calculating metric thresholds to determine high-risk files based on code size and complexity using a very large set of open and closed source projects written primarily in Java. They found that the probability of a file having a defect is higher in the very high-risk group with a few exceptions, and the same amount of code in large and complex files was associated with fewer defects than when located in smaller and less complex files. Their findings indicated that risk thresholds for size and complexity metrics have to be used with caution if at all.

In contact to these research efforts focused on the correlation between static code quality and other product measurements, *e.g.*, internal code quality *v.s.* defects [16, 17, 20], our study focuses on the code quality from the perspective of developers. More specifically, we intend to analyze the relationship between developer roles and code quality in the OSS context.

2.3. *Software Quality Assurance*

Previous studies [21, 22, 23, 24] have shown that human factors play a significant role in the quality of software components. Georgios et al. [25, 26] conducted two surveys on a set of integrators and contributors in GitHub to understand their work practices and challenges in pull-based development. They found that the contribution quality is a major concern for both integrators and contributors, and is one of the most frequently reported challenge items. They also found that automated testing, as a way of achieving shared understanding of quality, is a commonly accepted method to ensure the contribution quality. Kupsch et al. [27] discuss the methods and challenges on using software assurance tools. They also present quantitative evidence about the effects that can occur when assurance tools are applied in a simplistic or naive way.

Cavalcanti et al. [28] performed a systematic review on the literatures on studying change requests. They classified the selected 142 studies into two dimensions according to following the topics: challenges and opportunities. They also investigated tools and services for change request management, to understand whether and how they addressed the topics that were identified. Nagappan et al. [23] conducted a case study on Windows Vista and provided evidence that the organizational metrics are related to failure-proneness. Bird et al. [29] 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 fewer 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. [30] found that project specific expertise has a much larger impact on the time required to perform development tasks than high levels of diverse experience on unrelated projects. Mockus et al. [31] found that changes made by developers who are more experienced with a piece of code are less likely to induce failure.

None of these studies focus on the topic of internal quality assurance for external contributions in the OSS context. In our study, we aim to understand the integrators' and contributors' work practices of internal quality assurance, and the encountered challenges in the process.

3. METHODOLOGY

The main goal in this study is to investigate the internal quality assurance in social coding sites (*e.g.*, GitHub). We are interested in evaluating the internal quality of code made by casual contributors in OSS projects, and in understanding the work practices of the contributors and integrators on internal quality assurance in OSS communities. We structured our goal around several research questions.

DOI: 10.1002/smr

In OSS projects, a significant portion of the contributions come from casual contributors, while the lack of understanding of the project and diverse array of programming experience among them challenge the code quality ensurance work. Thus, our first research question was to evaluate the code quality of the casual contributors:

RQ1: How is the code quality of the casual contributors in social coding cites?

We sought to examine whether casual contributors would contribute lower quality of code, compared with that of the main contributors. We were also interested in finding out what quality issues the casual contributors tend to introduce. Moreover, prolific developers in OSS communities have social incentives to contribute multiple projects [32]. Since a person's energy is limited and her interest is focused on a few projects, she might contribute different projects in different roles. Then, will she contribute lower quality code when playing a casual contributor role? Therefore, we refined our first research question as follows:

RQ1.1*: Is the quality of the code made by casual contributors lower than that of the core contributors?*

RQ1.2*: When developers contribute multiple projects as different roles (main/casual contributor), do they perform differently in terms of the code quality?*

RQ1.3*: What types of CQIs do casual contributors tend to introduce? Which aspects of the software quality do these CQIs affect?*

Subsequently, we were interested in understanding how integrators and contributors work on managing internal quality of code in practice, as well as their experiences and encountered issues. This exploration is needed to guide future work in this area and led to our last research question:

RQ2: What are the developers' work practices for ensuring internal quality of the contributions in social coding sites?

RQ2.1*: How do the integrators and contributors perform in order to ensure internal quality of the contributions in social coding sites?*

RQ2.2*: What are the challenges of managing the internal quality of the contributions in social coding sites?*

To answer the two research questions, we analyzed the code quality of the developers in GitHub and conducted a online survey, which are described below.

3.1. Data Analysis

3.1.1. Project selection: In this paper, we select software projects from GitHub platform, which is the most popular code hosting site and is built on the Git version control system [33] [34] [35]. The projects are chosen according to the following principles:

- The language is Java, JavaScript or 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.

We extracted 30 projects (10 for each language) based on the first three principles using GitHub API, and filtered 9 projects according to the last two principles (1 for small number of commits, 3 for large number of commits and 5 for reporting errors during the analyzing process). As a result, 21 projects are sampled, including 6 Python, 7 Java and 8 JavaScript projects (main metric values for the sampled Python projects are shown in TABLE I). We cloned the whole git repository of each project, and extracted the corresponding data of commits, participants, stars and Pull Requests (PRs) before March 1st 2016 using GitHub API. The list of sampled projects and codes on data acquisition and processing in this study can be accessed at the GitHub page[‡].

3.1.2. Code quality analysis: There are a variety of static analysis tools, while most of them focus on a specific field or language. For example, CheckStyle mainly checks the coding style, while

[‡]https://github.com/roadfar/CasualContributor

DOI: 10.1002/smr

Table I. Main Metric Values for the Sampled Python Projects

Language	Project	Stars	Participants	CQIs
Python	jkbrzt/httpie	20828	583	570
	mitsuhiko/flask	18139	1186	2485
	kennethreitz/requests	17260	2151	3409
	rg3/youtube-dl	13960	4908	36333
	scrapy/scrapy	12256	823	29632
	letsencrypt/letsencrypt	11510	1086	22836

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 the SonarQube platform, which is a powerful tool to perform static analysis; it 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 in adding new languages, rules and integrations, and it also provides integrations with mainstream IDEs such as Eclipse, Visual Studio and IntelliJ IDEA through the SonarLint plugins. In addition, some superb features (e.g., TimeMachine, Technical Debt, Quality Issue Tracking, etc.) make the users manage the code quality more efficiently. The quality characteristic model is based on the SQALE (Software Quality Assessment based on Lifecycle Expectations) methodology[§]. SQALE is a quality model to support the evaluation of the non-functional requirements that relate to the code quality. It is a generic method, independent of the language and source code analysis tools, and has been used by many organizations for applications of any type and any size. In addition, this method allows doing the precise management of design debt for agile software development projects [36, 37].

After installing the Git plugin, SonarQube can automatically detect the introduced commit of the CQI using the *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 the web service API, and SonarQube

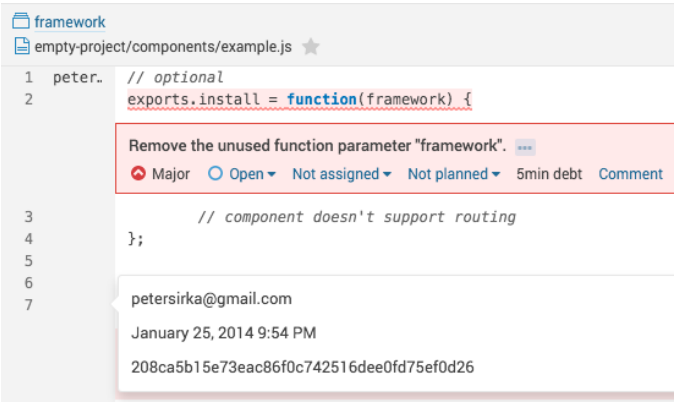

Figure 1. Screenshot of SonarQube Issue Page

also stores the git email of the author of the CQI in the MySQL database, which facilitates us in obtaining the CQIs introduced by a contributor.

To obtain the CQIs introduced by all of the 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 we

[§]http://www.sqale.org

DOI: 10.1002/smr

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. Thus, we optimized the algorithm, while scanning only the commits that need to be scanned. We sorted all the commits of a project by the commit time: $C_1, C_2, \ldots, C_n$, and defined the commits need to be analyzed: NC_1, $NC_2, \ldots, NC_k$ ($k \leq n$). NC_i is defined in a recursive way below:

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

$f(C_i)$ 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, then C_3 and C_1, C_5 and C_4 have common changed files (file A and F, respectively); therefore, C_3 and C_5 are the commits need to be analyzed.

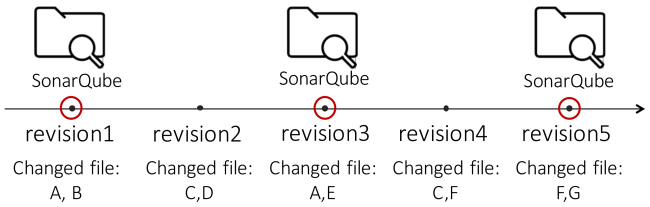

Figure 2. Optimization of the Scanning Algorithm

3.1.3. Definitions of Core Terms: Unlike some of the 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 PRs are rejected.

Some studies in the literatures have used similar or opposite definitions of the term casual contributor. Zhou and Mockus [38] defined *Long Term Contributor* to be a contributor who stays with the project for at least three years and who has productivity that exceeds the 10th percentile among the participants who have a tenure that exceeds three years. To measure the ownership of a component, Bird et al. [29] examined the distribution of ownership, and defined *Minor Contributor* as one whose contribution percentage to a component is below 5%.

In this study, we adopted the number of commits to a project to evaluate a developer's contribution, because the number of commits reflects the frequency of developers' contributing actions, which conveys better the 'casual' concept in the term '*casual contributor*', compared with other contribution metrics such as *changed code lines*. In addition, the numbers of commits and contributors of different projects are different, so that the contributors with large number of commits in the projects that have large number of commits or few contributors may be categorized as casual contributors when using a percentage threshold, which does not fit well with our understandings of casual contributors (contributors who have made very limited code contributions to a project casually). Therefore, we adopted an absolute number of threshold instead of the percentage when defining *casual contributors*. When calculating the threshold value, we refer to the definition of '*minor contributor*' in [29], and categorized casual contributors as those whose commit numbers are below the 5th percentile. We calculated the unified threshold on the whole data set and the value is 5. This means that casual contributors typically contribute around one PR to a project, because the mean number of commits in a PR is 4.47 [39]. 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.

DOI: 10.1002/smr

- ***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 identified as casual contributors.

3.1.4. Developers' Code Quality Measurement: Traditionally, software quality has been decomposed into internal and external quality attributes [1] [40] [41]. The external quality attributes are often reflected at the runtime stage, *e.g.*, functionality, usability, correctness, etc., which can be perceived by users. An important and commonly used measure for the external quality is *defect* [42]. Correspondingly, the 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 the sub-characteristics. For example, the over-complexity of the methods and fewer comments affect the maintainability and readability; and unused parameters and duplications could 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, while considering its internal quality from the point of view of developer, since it can reflect the developers' code quality.

3.1.5. 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 the issues analyzed by SonarQube as *Code Quality Issues (CQIs)*. Similar to defects, each CQI in SonarQube has one of five severities:

- ***BLOCKER*** — A bug with a high probability of impacting the behavior of the application in production, *e.g.*, memory leak, unclosed JDBC connection, etc. The code must be fixed immediately.
- ***CRITICAL*** — Either a bug with a low probability to impact the behavior of the application in production or an issue that represents a security flaw, *e.g.*, empty catch block, SQL injection, etc. The code must be reviewed immediately.
- ***MAJOR*** — A quality flaw that can highly impact the developer productivity, *e.g.*, uncovered piece of code, duplicated blocks, unused parameters, etc.
- ***MINOR*** — A quality flaw that 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.

3.1.6. CQID metric for developers' code quality: We use the Code Quality Issue Density (CQID), which is the number of introduced CQIs per changed code line to assess a developer's code quality in a project, without considering the differences among the CQIs' severities. We calculated the number of changed code lines of a user using the *git log* command, counting the added and changed lines [43].

3.2. Survey Design

3.2.1. Protocol: Based on the quantitative results, to understand the developers' work practices on internal quality management in GitHub, we sent two anonymous online surveys to the integrators[¶] and contributors[‖] of the sampled projects. Both surveys were split into three logical sections: demographic information, multiple choice or Likert-scale questions and open-ended questions. In the demographic section, we asked them about their experiences in software development and

[¶] https://roadfar.typeform.com/to/v0Kb8d
[‖] https://roadfar.typeform.com/to/xeYsd3

DOI: 10.1002/smr

GitHub use. In the second section, we used Likert-scale questions to learn their work practices and attitudes on handling and addressing the code quality of PRs. To further elicit the developers opinions, in all of the questions that had predefined answers but no related open-ended question, we included an optional 'Other' response. Finally, we used open-ended questions to understand the integrators and contributors encountered problems and their perceptions of code quality management. To allow the respondents to understand the term 'internal quality', we illustrated it in the subjects with examples.

3.2.2. Data collection: We extracted the emails of the integrators (*i.e.*, the GitHub users who have closed other developers' PRs) and the contributors who have registered their email address on GitHub. To encourage participation, the surveys were customized on the participants' salutation, and the web address was sent by personal email to all of the participants. As a result, we emailed 161 integrators and 471 contributors, and received 16 answers (10% answer rate) and 65 answers (14% answer rate), respectively.

3.2.3. Participants: The majority of the respondents self-identified as company employees (69%), while 16% are full-time OSS developers, other occupations including student, consultant, founder and unemployed. Most of surveyed integrators have more than 6 years of software development experience (75%) and considerable experience (>4 years) in using GitHub (75%). In terms of the investigated contributors, the software development experience appear to be more diverse, compared with that of the integrators: 62% have more than 6 years of software development experience, while 3 have less than one year. In addition, 81% have been using GitHub for more than 4 years, and all of the contributors have ever acted as casual contributors.

4. RESULTS

4.1. RQ1.1: Code Quality of Casual Contributors

To examine the code quality between the main and casual contributors, we first compared the average CQID value of the two groups per project in each severity. Results show that casual contributors introduce greater CQIDs than main contributors at *blocker*, *critical*, *major*, *minor* and *info* severities in 100.00%, 90.48%, 71.43%, 61.90% and 71.43% of the sampled projects respectively. We then compared the average CQID of the two groups on the whole dataset at different severities. The results are shown in Figure 3, except for the *info* severity at which both the main and

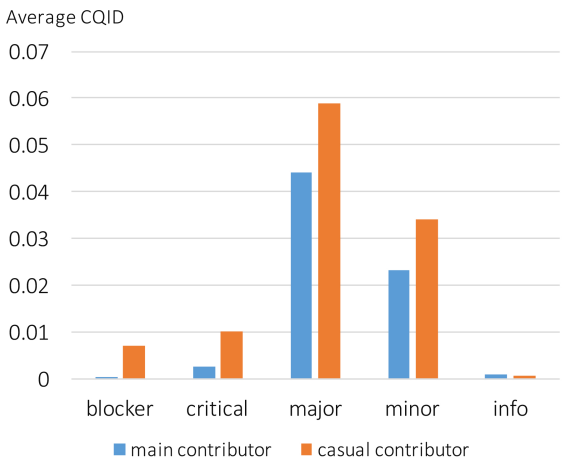

Figure 3. Average CQID introduced by main and casual contributors

casual contributors have introduced few CQIs, the casual contributors introduce more CQIs than

DOI: 10.1002/smr

main contributors on average. Especially for higher severity CQIs such as *blocker* and *critical*, the average of the CQID introduced by the casual contributors is over three times more than that of the main contributors. This preliminary result motivated us to complete an in-depth analysis.

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

Finding 1: In OSS communities, casual contributors tend to introduce more CQIs than main contributors. Especially, the number of high-severity CQIs introduced by casual contributors are three times more than the number from main contributors.

4.2. RQ1.2: Code Quality of a Developer as Different Roles

In our data set, we observed that there is a small group of developers who have contributed multiple projects, and some of them have contributed different projects in different roles. This finding motivated us to examine whether they perform differently in terms of their code quality when acting as main and casual contributor roles. We picked out 29 developers who satisfied our condition. Note that if a developer contributes more than two projects as main and casual contributor roles, we made multiple pairs respectively. For example, *artem-zinnatullin* submitted 25 commits to the *RxJava* project as a main contributor, while having 3 and 4 commits to *retrofit* and *okhttp* as a casual contributor. Thus, *RxJava-retrofit* and *RxJava-okhttp* are the two pairs that are 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 the difference in the CQID between the pairs. The results are shown in TABLE II. We can see that there is small correlation between the introduced CQID when the developers act in different roles when contributing to different projects. The results for the *paired samples test* show that the difference in the mean value is not significant ($p = 0.168$ >0.05), and the differences in the mean value of CQID at all severities are not significant either.

Table II. The results for the 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 Sample Test										
	Mean				Std. Deviation				t	Sig.
main - casual	-.069				.049				-1.405	.168
	blocker		**critical**		**major**		**minor**		**info**	
	t	Sig.	t	Sig.	t	Sig.	t	Sig.	t	Sig.
main - casual	1.436	0.160	-1.000	0.324	-1.101	0.279	-1.092	0.282	0.280	0.781

Further, we observed that the star numbers of these contributors projects are relatively high, which represent the high popularity of their projects. Combined with *Finding 1*, the result for *Finding 2* raised the question as to whether the CQIs introduced by casual contributors are mainly attributed to the developers who have fewer project stars. Hence, we made an extensive analysis on

DOI: 10.1002/smr

the differences in the project stars between the high CQID and low CQID group among the casual contributors. We sorted the data of the CQIDs introduced by casual contributors and divided it into four parts. An *independent sample t-test* was used to examine the significance of the differences in the star number between the first quarter and the last quarter. The results show that the mean value for the high and low CQID group are 123.59 and 339.87, respectively, and the p values for the *f-test* (<0.001) and *t-test* (0.049) are both below 0.05. Therefore, we can make the following statement:

> *Finding 2:* In the OSS context, the developers do not perform differently in terms of the code quality when acting as main and casual contributor roles, and the developers who have fewer project stars tend to introduce more CQIs than those who have more.

4.3. RQ1.3: 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. To understand what CQIs the casual contributors introduce frequently, we first classified the CQIs using the default taxonomy. If a CQI has multiple tags, then it is classified into applicable each category. Figure 4 shows the top 5

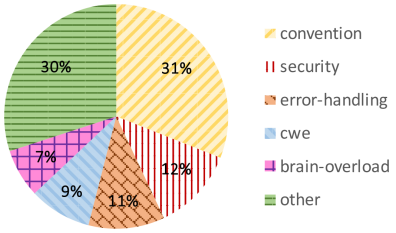

Figure 4. Classification of the CQIs

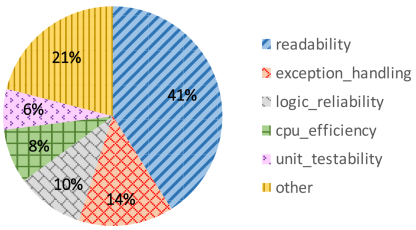

Figure 5. Influenced characteristics

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 that are related to architecture and design .
- ***Brain-overload*** — there is too much to keep in the 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: 99% of the CQIs on *error-handling* are tagged with *security*, and 93% of the CQIs on *security* are tagged with *error-handling*. This finding suggests that a large part of the CQIs in the two categories consists of the same CQIs as calculated by our method. Therefore, we can summarize from Figure 4 that over 50% of the CQIs introduced by casual contributors are on coding conventions and exception handling. Further, for each of the four categories, we list the top three rules that are the most easily broken by casual contributors in TABLE III. Usually, the coding conventions 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 conventions to unify the code style. To examine whether the CQIs on *convention* are in conformity with the projects coding style, we randomly chose 50 CQIs and found that 44 are not. An example on adding semicolons at the ends of statements in JavaScript is shown in Figure 6.

Figure 5 shows the top 4 characteristics that are influenced by casual contributors. The characteristics are evaluated by the SQALE (Software Quality Assessment based on Lifecycle Expectations) methodology. In the SQALE Quality Model, the priorities of the characteristics are the following: Testability > Reliability > Changeability > Efficiency > Security > Maintainability

DOI: 10.1002/smr

Table III. The most easily broken rules by casual contributors

Category	Name	Broken times	Per.[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 be either 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

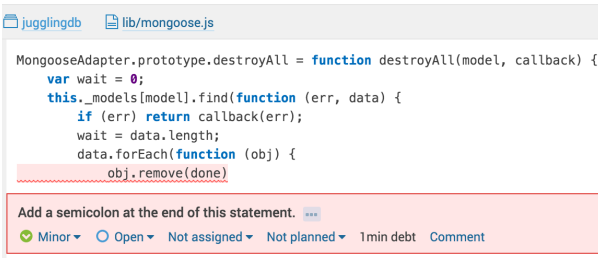

Figure 6. An example of a CQI on Coding Convention

> Portability > Reusability, which means that an app should be testable first and then it should be reliable, then changeable, etc. It can be seen that the casual contributors introduce mainly quality issues on the softwares readability, security, reliability, efficiency and testability, and the number of issues increases as the priority of characteristics drop. Consequently, we can make the following summary:

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

DOI: 10.1002/smr

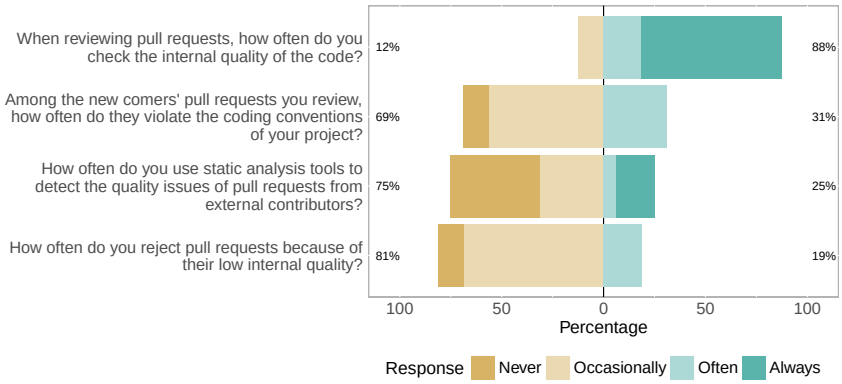

Figure 7. Work practices of integrators on internal quality management

4.4. RQ2.1: Integrators' and Contributors' Work Practices

This section presents the results of our exploratory surveys. When quoting the survey respondents, we refer to the integrators and contributors using [iX] and [cX] notations respectively, where X is the respondents ID in our survey. Codes that result from coding open-ended answers are *underlined* [26].

4.4.1. Integrators' work practices: To ask the integrators about their work practices of managing the internal quality of PRs, we provided them with a set of 4 questions with a 4-level Likert scale. The answers are presented in Figure 7. The results show that, in general, the majority of integrators (88%) would check the internal quality of the code when reviewing the PRs. In most of the cases, however, the process is manual: most of them (75%) occasionally or never rely on static analysis tools. When reviewing the PRs, a considerable portion of the integrators (31%) often encounter coding-convention issues, which is consistent with the results of data analysis in Section 4.3. However, a smaller proportion of them would reject PRs because of low internal quality at the same frequency, from which we can infer that some PRs that have been examined for *coding-convention* issues are still accepted.

In the open-ended question section, we asked the integrators to write down the aspects that they focus on with regard to internal quality when reviewing PRs. The results show that the aspects of internal quality that they value are not the same from one to another. For example, in terms of coding conventions, some of the respondents treat it strictly, *e.g., "[the PRs] must match the coding styles of current project"* [i6], *"Coding style have to be honored by a commit"* [i16]; yet, some integrators tend to ignore coding-convention issues only if they are really bad, *e.g., "Design & correctness are the most important ones. Conventions/formatting I'll mention only if it's really bad"* [i2]. Overall, the most frequently mentioned quality aspects are coding style (54%), documentation (31%) and complexity (31%). Other mentioned aspects such as design (15%), integration with the remainder of the code (15%), white space (8%) and structure (8%) are also about the clarity and understandability of the code.

4.4.2. Contributors' work practices: We provided a set of 5 questions to the contributors to investigate their work practices with regard to ensuring the internal quality of PRs. The answers are shown in Figure 8. We can see that most of the investigated contributors (86%) would always consider the internal quality of their changed code, while there exist a few respondents (28%) have ever neglected the impact of the code and its integration with the remainder of the code. With regard to coding style, a small proportion of the contributors (18%) do not have a habit of reading the conventions that are formulated by communities. Nevertheless, over half of the contributors (52%) are accustomed to using static analysis tools.

DOI: 10.1002/smr

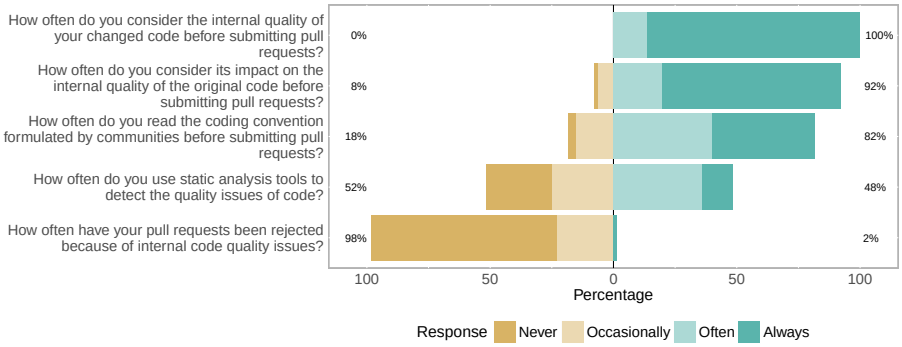

Figure 8. Work practices of contributors on internal quality management

Synthesizing the integrators and contributors work practices on the internal quality assurance, we can reach the following finding:

Finding 4: In the OSS context, most of the integrators and contributors value the internal quality of the code, while they tend to check manually, without using quality analysis tools. Different integrators care for different aspects of the internal quality, while the coding style, readability, clarity and well integration with existing code are the frequently audited characteristics. A small proportion of contributors do not become accustomed to reading the coding conventions, and they tend to neglect the consequent impact on the internal quality of the code.

4.5. RQ2.2: Challenges of Internal Quality Assurance

To find the pain points of the internal quality assurance experienced by integrators and contributors, we explicitly introduced a mandatory open-ended question in both of the surveys, and we asked the respondents to state the challenge that they faced. We concluded the answers from the integrators and contributors perspectives below.

4.5.1. Integrators' pains: We learned that the challenges that the integrators faced mainly revolve around two themes: challenges about the *contribution quality and technical level of the contributors* and challenges about the *tools*. 31% of the respondents mentioned that the internal qualities are sometimes *paid less attention to* by the contributors, compared with the external qualities, *e.g.*, "*People only care about solving their immediate problem(s)*" [i1], "*Contributors are often not invested enough to make code quality fixes*" [i2], "*They don't add tests, let alone adding documentation*" [i3]. Some of the integrators mentioned the issue of the *diversity of contributors' programming skills*. For example, "*It's difficult with so diverse programming skills (...)*" [i3], "*There are constantly newcomers to educate*" [i4]. An integrator mentioned the language issue and complained about the Python and JavaScript contributors: "*Python/JS dev[eloper]s are the worst, they seem to make up the rules on the fly (...) they decide the coding standard is no longer important (...) Programmers are the problem*" [i13].

The *tool* theme also permeates several challenges. Some of the integrators mentioned that current platform support for internal quality assurance is not sufficient and hoped for mechanisms like *inline reporting* and *editing quality issues*. For example, "*Also, I don't want to look like a jerk asking to respect the max line width. I'd love it if we had inline lint reporting on Github, so I can just be like 'please fix lint warnings'. (...) inline lint warnings would be great (e.g. coming from Travis/Circle/X CI). Inline editing of PRs would also be great, so that I can fix tiny nits without bothering the author*" [i2]. A respondent [i12] mentioned the *automation of the maintenance of internal quality*, "*I would prefer to see public source code maintenance (open source and otherwise) move toward becoming "maintainers" through automation and decentralization*".

DOI: 10.1002/smr

4.5.2. Contributors' pains: The challenges that are frequently reported by contributors are around the *management of core teams*, the *complexity essence of internal quality* and *tools*. Some of the respondents complained of the *prejudice* of integrators, *e.g.*, "*Many projects have an unhealthy power structure (eg. there are more contributions than those people able and willing to review and integrate the pull requests)*" [c6], "*Sometimes, the project stewards don't know best and should rather conform to the prevalent public opinion. 'You can always fork and maintain your fork' is a joke*" [c13], "*There is a lot of hostility when submitting code to many small projects with few contributors. I'm often met like this. I believe it's because the project maintainer's aren't used to having other people play with their toys/code*" [c23]. A respondent [c9] mentioned the *contradiction between internal quality and contribution attraction*: "*Quality/style is important but being too strict can also deter contributors. A project has to find a right balance*". A few respondents feel that it is challenging to understand the project comprehensively, *e.g.*, "*understanding the project as a whole*" [c17]. The cognitive dissonance on the internal quality among the developers is mentioned, *e.g.*, "*It varies a lot between projects and people*" [c28].

Similar to the integrators' answers, some of the contributors also mentioned the issue of tool support for internal quality assurance. Quality issues about *automated detection and fixing of quality issues* are expected by some respondents, *e.g.*, "*Comprehensive automatic review of pull requests is not quite there yet (checking code quality and style). Automatic CI is not quite the same, GitHub+CI could really propose fixes to most issues (whitespace, code style) automatically and that would help newcomers*" [c29]. At the same time, there are also some of the challenges mentioned by some respondents, *e.g.*, "*I think it's a hard problem to solve automatically with static analysis, since the hardest problems are clearly communicating mental models between contributors and users, naming things unambiguously, and getting people on the same page. While tools like automatic formatting fixers would take off a lot of the back-and-forth between maintainers and contributors, it'll always be a hard problem. Which is okay, we can build tools to help people do this better, but it'll stay a human problem for quite some time*" [c12].

Synthesizing the integrators' and contributors' answers, we conclude the main challenges on internal quality assurance below:

> *Finding 5:* In the OSS context, the contributors tend to attach more importance to the external quality than to the internal quality, and the diversity of their programming skills challenges the internal quality of the contributions. In addition, understanding the project comprehensively is sometimes challenging for the contributors, and the variety of internal quality requirements among the projects puzzles them. Platform support for internal quality assurance, such as automated quality issue detection, fixing and integration into pull-based workflow, are expected by both integrators and contributors.

5. IMPLICATIONS

Based on the findings above, this section discusses the limitations of internal quality management in modern social coding sites and a possible improvement method: Continuous Inspection.

5.1. Limitations of Internal Quality Management in Social Coding Sites

5.1.1. Limitations of internal quality requirement specification: To ensure the internal quality of the contributions, the projects (especially the popular ones) usually specify the internal quality requirements in a contribution page. Generally, the internal quality requirements are on coding conventions, *e.g.*, Rails lists 11 coding rules, such as whitespace and naming styles, in the contribution page. However, we can see from *Finding 3* that the most frequently introduced CQIs by casual contributors are on conventions, which are still merged into the code base. Based on these findings, we attribute this result to two main limitations of such a specification method for internal quality requirements. The first is **lack of enforceability**. *Finding 5* shows that some of the

DOI: 10.1002/smr

contributors invested less into the internal quality of the code, compared with that of functional aspects. In addition, from the integrators and contributors feedback in *Finding 4*, we can see that a few contributors do not build a habit of reading the contribution page. Without constraints, PRs with convention issues are submitted. Second, as indicated in Finding 5, internal quality requirements vary a large amount among developers and projects. For example, though both were written in Ruby, *gitlabhq* adopts a commonly accepted style guide[**], while *rails* formulates its own style[††]. *Finding 2* shows that the developers do not perform differently in terms of the code quality when contributing multiple projects, which implies that they tend to maintain their coding habits regardless of the projects context. This finding is consistent with previous work [6], which provides evidence that the developer-introduced violations can be used to compute fingerprints. Hence, the lack of uniformity on internal quality requirements would puzzle developers because they must switch coding habits when contributing to multiple projects.

5.1.2. Limitations of manual review on internal quality: Most OSS projects adopt the manual review to audit the internal quality of PRs, *i.e.*, the integrators manually decide whether a PR conforms to the quality requirements of the project. Based on *Finding 4* and *5*, we can see some limitations of such methods for ensuring the internal quality. First, the process **adds burdens to the integrators**. The integrators must check the internal quality of each PR and communicate the issues with the authors. The burdens are heavier on the integrators in popular projects [39]. When encountering CQIs, whether to reject PRs puzzles them: if they reject the PR, they would discuss the issues with the contributor; then, the contributor would have to fix it, and the integrators have to examine it again, which can even deter the contributors. If they accept the PR, then they would fix it by themselves without bothering the contributor. Second, the review process is subjective and manual [25], which can **challenge the reviewing quality**. As evidence, the results in Section 4.4.2 show that a small proportion of the surveyed integrators (12%) do not form a habit of checking the internal quality of the PRs. Nevertheless, the reviewing process is not always reliable for the integrators who are accustomed to auditing the internal quality; in fact, *Finding 4* shows that different integrators tend to pay different amounts of attention to the internal quality and emphasize different aspects of the quality characteristics.

5.2. Continuous Inspection for Internal Quality Assurance

In light of the limitations of the existing method, we seek a paradigm that provides an automated and objective way to assure the internal quality. At this point, *Continuous Inspection* is an alternative method that was proposed by SonarSource in 2013 [44]. *Continuous Inspection* provides continuous code quality management that incorporates shorter feedback loops to ensure the rapid resolution of quality issues. We believe that this method can be an alternative paradigm for tackling the challenges in the prior section because of the principles below [44]: **(1) All of the stakeholders in the development process: stakeholders must be alerted when new quality flaws are injected, and all new issues and existing critical issues must be assigned a clear path and timeline for resolution**, which means that the integrators and contributors should be involved in the quality assurance activities. Under an automated quality analysis process, anyone who introduces quality issues is responsible for fixing them. This principle implicitly requires the developers to comply with predefined quality standards unconditionally; thus, it effectively addresses the challenge *"lack of enforceability"* along with *reducing the "burden of integrators"*. **(2) Software quality requirements must be objective and must be common to all software products, regardless of their specifics**. Compared with manual reviewing, this principle emphasizes that the process should be implemented by tools, in an objective way. There is no doubt that it helps to tackle the *"lack of uniformity"* issue if all of the OSS projects adopt such a method and the corresponding tools in future. In addition, the method is also beneficial to improving the *"reviewing quality"*.

[**]https://github.com/bbatsov/ruby-style-guide/blob/master/README.md#source-code-layout
[††]http://edgeguides.rubyonrails.org/contributing_to_ruby_on_rails.html

DOI: 10.1002/smr

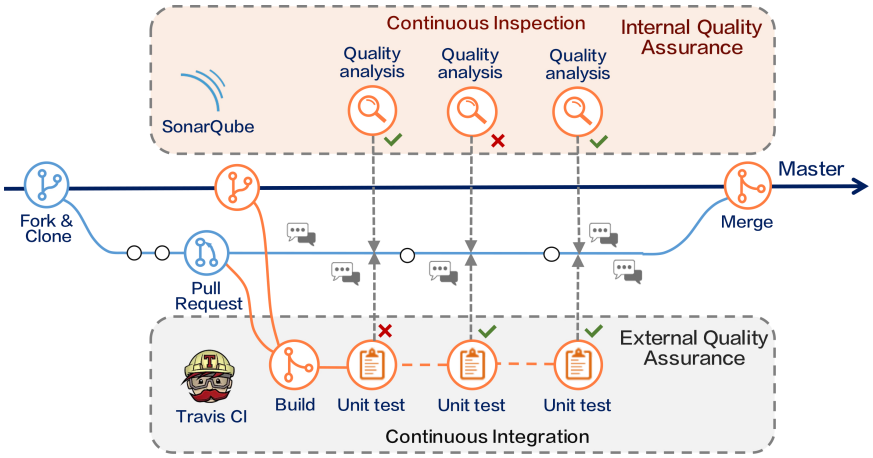

Figure 9. The process of Continuous Inspection integrated into the Continuous Integration process

To support *Continuous Inspection* in the OSS context, SonarSource provides a GitHub integration to automatically analyze the quality of the incoming PRs, which has been adopted by a few projects in GitHub. A typical process is depicted in Figure 9. In the *Continuous Integration* process, the continuous integration tool (*e.g.*, Travis CI) automatically creates a new testing branch by merging the PR authors branch and the main branch for an incoming PR; then, it builds and performs unit testing. Until the PR passes the test suite, it would trigger a quality analysis process. If the severities of the introduced CQIs of the pull request are high, *e.g.*, *critical* or *blocker*, the PR will be rejected automatically. Until the high-severity CQIs are fixed, the PR is delivered to integrators to review. Because it is similar to the *Continuous Inspection* process, which adopts unit testing to ensure the external quality, the *Continuous Inspection* process adopts static analysis to ensure the internal quality. This process helps in finding quality problems early when fixing them is still cheap and easy, and it can also educate developers instantly while the code is still fresh in their minds.

6. THREATS TO VALIDITY

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

6.1. Internal Validity

The validity of the quantitative results in this study is built on the validity of the tool that we used: SonarQube. The concepts and taxonomy that we used are all default values set by SonarSource. The characteristic model is based on the SQALE methodology, which is a public methodology to support the evaluation of a software applications source code in the objective, accurate, reproducible and automated way. The SQALE method results of internal research performed within *inspearit*[‡‡]. It has been validated through the analysis of millions of lines of code of numerous languages. It is now implemented by private, open source and commercial tools and used within large organizations. A common issue of static analysis tools is false positives [45], and we also find some cases in our study. For example, we found that SonarQube analyzed *naming conventions* for the files that were auto-generated by the development tools as well, e.g., the R file is a resource file that is generated by the Android Development Toolkit, and the naming style of the variables (e.g., m_ text) is inconsistent with the 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.

[‡‡]http://www.inspearit.com

DOI: 10.1002/smr

In the survey design part, the question-order effect (*e.g.*, one question could have provided context for the next) could lead the respondents to a specific answer [46]. In our case, we decided to order the questions based on the natural sequence of actions, to help the respondents recall and understand the contexts of the questions asked [26]. In addition, the social desirability bias [47] (*i.e.*, a respondents possible tendency to appear in a positive light, such as by showing that they are fair or rational) could have influenced the answers. To mitigate this issue, we informed the participants that the responses would be anonymous.

6.2. *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 that have fewer stars, further verification must be made. In addition, the study concerned only the Python, Java and JavaScript projects, and other popular languages such as C++ and Ruby are not covered. Another risk to the external validity is the data that is used when analyzing ***RQ2***. We observed that all of the developers who contribute to different projects in different roles have a large number of project stars and followers, which reflects their social status and technical level. Thus, the conclusion might not hold for the developers with lower social status and less technical experience.

In terms of the qualitative part, a main limitation is in the scale of the survey. We attempted to send the surveys to a number of developers, while we found that a portion of the registered email addresses (approximately 30%) were invalid, and the low response rate leads to the limited number of respondents.

6.3. *Construct Validity*

Referencing the commonly used code quality metric, namely, the defect density, we measured the developers code quality by the density of the CQIs without considering the severity of the CQIs. However, as [48] 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, according to the specific case. Thus, as a supplement, we compared the CQID between the main and casual contributors at different severity levels, and the results enhanced our finding. In addition, we believe that this threat can be alleviated in the big data context.

7. CONCLUSIONS AND FUTURE WORK

The level of internal quality that a software product has today tends to affect the level of its cost liability tomorrow. In this study, we first quantitatively investigate the internal quality of the code made by casual contributors in 21 popular OSS projects within GitHub, and we obtained some valuable findings. For example, our study reveals that the casual contributors, especially those who have fewer project stars, introduce three times more high-severity CQIs than the main contributors do. We also find that when the developers contribute to different projects in different roles, they do not perform differently in terms of the code quality. Categories and rules are presented to identify what types of CQIs casual contributors frequently introduce. Based on the results, we conducted a survey on integrators and contributors in GitHub. Qualitative analysis of the survey exposes some limitations of internal quality management in GitHub, *e.g.*, the diversity of the contributors programming skills challenges the internal quality assurance for the integrators and adds external burdens on them; additionally, the contributors sometimes have difficulty in understanding the project comprehensively, and the variety of internal quality requirements among the projects puzzles them. Finally, we discuss *Continuous Inspection* as an alternative method for internal quality assurance in social coding sites. We believe that our findings and implications provide valuable guidelines for quality management in the OSS context.

In terms of future work, based on the implications of this study, we will conduct a case study on GitHub projects to analyze the influence of *Continuous Inspection* on the quality and efficiency of the software and process.

DOI: 10.1002/smr

8. ACKNOWLEDGMENTS

We thank our postgraduate students Xiang Hou and Yiang Gan for their help with the experiments, and we also want to thank SonarSource for their kind help in tool usage.

REFERENCES

1. Tonella P, Abebe SL. Code quality from the programmer's perspective. *PoS* 2008; **001**.
2. Kshetri NB, Palvia PB, Singh RB. Improving open source software maintenance. *Journal of Computer Information Systems* 2010; **50**(50):81–90.
3. Yu Y, Yin G, Wang T, Yang C, Wang H. Determinants of pull-based development in the context of continuous integration. *Science China Information Sciences* 2016; **59**(8):080 104.
4. Zhang Y, Wang H, Yin G, Wang T, Yu Y. Social media in github: the role of @-mention in assisting software development. *Science China Information Sciences* 2017; **60**(3):032 102.
5. Bollinger T, Nelson R, Self KM, Turnbull SJ. Open-source methods: Peering through the clutter. *IEEE Software* 1999; **16**(4):8–11.
6. Ioannis S, Lefteris A, Apostolos O, Bleris GL. Code quality analysis in open source software development. *Information Systems Journal* 2002; **12**(1):43–60.
7. Gousios G, Kalliamvakou E, Spinellis D. Measuring developer contribution from software repository data. *in Proceedings of 2008 International Working Conference on Mining Software Repositories*, 2008; 129–132.
8. Avgustinov P, Baars AI, Henriksen AS, Lavender G. Tracking static analysis violations over time to capture developer characteristics. *IEEE International Conference on Software Engineering*, 2015; 437–447.
9. Chidamber SR, Kemerer CF. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 1994; **20**(6):476 – 493.
10. Van Emden E, Moonen L. Java quality assurance by detecting code smells. *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, IEEE, 2002; 97–106.
11. Dixon M. An objective measure of code quality. *Technical Report*, Energy Group, Beverly, Massachusetts 2008.
12. Goues CL, Weimer W. Measuring code quality to improve specification mining. *IEEE Transactions on Software Engineering* 2012; **38**(1):175–190.
13. Wong-Mozqueda JA, Haines R, Jay C. Is code quality related to test coverage? *in Proceedings of the International Workshop on Sustainable Software Systems Engineering*, 2015.
14. Alves TL, Ypma C, Visser J. Deriving metric thresholds from benchmark data. *IEEE International Conference on Software Maintenance*, 2010; 1–10.
15. Ayewah N, Hovemeyer D, Morgenthaler JD, Penix J, Pugh W. Using static analysis to find bugs. *IEEE Software* 2008; **25**(25):22–29.
16. Nagappan N, Ball T. Static analysis tools as early indicators of pre-release defect density. *in Proceedings of the 27th International Conference on Software Engineering, 2005.*, 2005; 580 – 586.
17. Nagappan N, Williams L, Hudepohl J, Snipes W, Vouk M. Preliminary results on using static analysis tools for software inspection. *the 15th IEEE International Symposium on Reliability Engineering* 2004; :429–439.
18. Baca D, Carlsson B, Kai P, Lundberg L. Improving software security with static automated code analysis in an industry setting. *Software Practice and Experience* 2012; **43**(3):259–279.
19. Sözer H. Integrated static code analysis and runtime verification. *Software Practice and Experience* 2015; **45**(10):1359–1373.
20. Yamashita K, Huang C, Nagappan M, Kamei Y, Mockus A, Hassan AE, Ubayashi N. Thresholds for size and complexity metrics: A case study from the perspective of defect density. *IEEE International Conference on Software Quality, Reliability and Security*, 2016; 191–201.
21. Bird C, Nagappan N, Gall H, Murphy B, Devanbu P. Putting it all together: using socio-technical networks to predict failures. *in Proceedings of the 20th International Symposium on Software Reliability Engineering,*, IEEE, 2009; 109–119.
22. Cataldo M, Wagstrom PA, Herbsleb JD, Carley KM. Identification of coordination requirements: implications for the design of collaboration and awareness tools. *in Proceedings of the 20th anniversary conference on Computer supported cooperative work*, ACM, 2006; 353–362.
23. Nagappan N, Murphy B, Basili V. The influence of organizational structure on software quality: an empirical case study. *in Proceedings of the 30th international conference on Software engineering*, ACM, 2008; 521–530.
24. Pinzger M, Nagappan N, Murphy B. Can developer-module networks predict failures? *in Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, ACM, 2008; 2–12.
25. Gousios G, Zaidman A, Storey MA, Deursen AV. Work practices and challenges in pull-based development: The integrator's perspective. *in Proceedings of the 37th International Conference on Software Engineering*, vol. 1, IEEE Press, 2014; 358–368.
26. Gousios G, Storey MA, Bacchelli A. Work practices and challenges in pull-based development: the contributor's perspective. *in Proceedings of the 38th International Conference Software Engineering*, ACM, 2016; 358–368.
27. Kupsch JA, Heymann E, Miller B, Basupalli V. Bad and good news about using software assurance tools. *Software Practice and Experience* 2017; **47**(1):143–156.
28. Cavalcanti YC, Anselmo P Mota Silveira Neto, Machado IDC, Vale TF, Almeida ES, Meira SRDL. Challenges and opportunities for software change request repositories: a systematic mapping study. *Software Evolution and Process* 2014; **26**(7):620–653.
29. Bird C, Nagappan N, Murphy B, Gall H, Devanbu P. Don't touch my code!: examining the effects of ownership on software quality. *in Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ACM, 2011; 4–14.

DOI: 10.1002/smr

30. Boh WF, Espinosa JA. Learning from experience in software development: A multilevel analysis. *Management Science* 2007; **53**(8):1315–1331.
31. Mockus A, Weiss DM. Predicting risk of software changes. *Bell Labs Technical Journal* 2000; **5**(2):169–180.
32. Vasilescu B, Blincoe K, Qi X, Casalnuovo C, Damian D, Devanbu P, Filkov V. The sky is not the limit: multitasking across github projects. *in Proceedings of the 38th International Conference on Software Engineering*, ACM, 2016; 994–1005.
33. Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D. The promises and perils of mining github. *in Proceedings of the 11th Working Conference on Mining Software Repositories*, ACM, 2014; 92–101.
34. Yu Y, Vasilescu B, Wang H, Filkov V, Devanbu P. Initial and eventual software quality relating to continuous integration in github 2016; .
35. Yu Y, Wang H, Yin G, Wang T. Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? *Information and Software Technology* 2016; **74**(C):204–218.
36. Letouzey JL. The sqale method definition document. *in Proceedings of the 3rd International Workshop on Managing Technical Debt (MTD)*, 2012; 31 – 36.
37. Letouzey JL, Coq T. The sqale analysis model: An analysis model compliant with the representation condition for assessing the quality of software source code. *Advances in System Testing and Validation Lifecycle (VALID), 2010 Second International Conference on*, IEEE, 2010; 43–48.
38. Zhou M, Mockus A. Who will stay in the floss community? modeling participant's initial behavior. *IEEE Transactions on Software Engineering* 2015; **41**(1):82–99.
39. Gousios G, Pinzger M, Deursen AV. An exploratory study of the pull-based software development model. *in Proceedings of the 36th International Conference Software Engineering*, ACM, 2014; 345–355.
40. ISO I. Iec25010: 2011 systems and software engineering–systems and software quality requirements and evaluation (square)–system and software quality models. *International Organization for Standardization* 2011; :34.
41. Wong CP, Xiong Y, Zhang H, Hao D. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. *in Proceedings of the International Conference on Software Maintenance and Evolution*, IEEE, 2014; 181–190.
42. Zhang H, Babar MA. Systematic reviews in software engineering: An empirical investigation. *Information and Software Technology* 2013; **55**(7):1341–1354.
43. Lu Y, Mao X, Li Z. Assessing software maintainability based on class diagram design: A preliminary case study. *Lecture Notes on Software Engineering* 2016; **4**(1):53–58.
44. Gaudin O, SonarSource. Continuous inspection: A paradigm shift in software quality management. *Technical Report*, SonarSource S.A., Switzerland 2013.
45. Rocha H, Valente MT, Maques-Neto H, Murphy G. An empirical study on recommendations of similar bugs. *in Proceedings of the 23rd International Conference on Software Analysis, Evolution and Reengineering*, vol. 1, IEEE, 2016; 46–56.
46. Sigelaman L. Question-order effects on presidential popularity. *Public Opinion Quarterly* 1981; **45**(2):199–207.
47. Furnham A. Response bias, social desirability and dissimulation. *Personality and Individual Differences* 1986; **7**(3):385–400.
48. Zheng J, Williams L, Nagappan N, Snipes W, Hudepohl JP, Vouk MA. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering* 2006; **32**(4):240–253.

DOI: 10.1002/smr