Front. Comput. Sci., 2020, 14(5): 145205
https://doi.org/10.1007/s11704-019-9023-2

RESEARCH ARTICLE

Improving students' programming quality with the continuous inspection process: a social coding perspective

Yao LU[1], Xinjun MAO (✉)[1,2], Tao WANG[1], Gang YIN[1], Zude LI[3]

1 College of Computer, National University of Defense Technology, Changsha 410073, China
2 Key Laboratory of Software Engineering for Complex Systems, Changsha 410073, China
3 School of Information Science and Engineering, Central South University, Changsha 410012, China

Abstract College students majoring in computer science and software engineering need to master skills for high-quality programming. However, rich research has shown that both the teaching and learning of high-quality programming are challenging and deficient in most college education systems. Recently, the continuous inspection paradigm has been widely used by developers on social coding sites (e.g., GitHub) as an important method to ensure the internal quality of massive code contributions. This paper presents a case where continuous inspection is introduced into the classroom setting to improve students' programming quality. In the study, we first designed a specific continuous inspection process for students' collaborative projects and built an execution environment for the process. We then conducted a controlled experiment with 48 students from the same course during two school years to evaluate how the process affects their programming quality. Our results show that continuous inspection can help students in identifying their bad coding habits, mastering a set of good coding rules and significantly reducing the density of code quality issues introduced in the code. Furthermore, we describe the lessons learned during the study and propose ideas to replicate and improve the process and its execution platform.

Keywords continuous inspection, programming quality, SonarQube

Received January 17, 2019; accepted July 10, 2019

E-mail: xjmao@nudt.edu.cn

1 Introduction

College students majoring in computer science (CS) and software engineering (SE) are potential future professional software engineers who need to master skills for high-quality programming [1–3], that is, the ability to write code with high readability, understandability, maintainability, etc. However, researchers [4–6] have recognized that graduates entering the workforce lack high-quality programming skills. Eric Brechner, Director of Developer Excellence at Microsoft, identified *high-quality code that lasts* as one of the 5 aspects of capabilities that graduates need to improve [7]. One important cause underlying this problem is that both the teaching and learning of high-quality programming skills are challenging in most college education systems. On the one hand, students usually cannot sufficiently master programming skills through the "knowledge-inculcation" model of the instructors [3, 8, 9]. The training of programming, a practical skill, requires students' real practices and personal meaning construction [10]. In addition, knowledge background and programming experience differ among individual students, and thus, it is not conducive for personalized learning to conduct unified teaching in the class. On the other hand, when students conduct programming assignments or projects, teachers usually do not have effective programming-quality improvement and assessment methods [2, 11]. Consequently, a gap exists between students' programming skills and the expectations of industry managers or other hiring personnel [6, 12].

The lack of high-quality programming capabilities can put job-hunting students at a disadvantage and can even influence the productivity and quality of newly hired employees. Therefore, it is critical to train students from the beginning to write code that not only works correctly but also meets internal quality standards[1] [13, 14].

Social coding sites, such as GitHub, receive large numbers of contributions from developers around the world [15, 16], and the frequent turnover of external developers and the wide variations among their coding experience challenge the project management of code and its internal quality [17]. To meet this challenge, automated quality assurance methods are increasingly adopted and integrated into the contribution workflow [15, 18]. Figure 1 illustrates a typical process in GitHub that integrates the continuous integration and continuous inspection processes into the pull-based model to ensure the external and internal quality of contributions, respectively. Pull-based development is a paradigm for distributed software development in which developers can perform changes on a copied version of the central repository. The pull-based workflow decouples the development effort from the decision to incorporate the results of the development in the code base [15]. The workflow begins with a developer forking[2] a repository that she wants to contribute to. After making changes to the forked repository, she opens a pull request[3] (PR), expressing her readiness to have the branch[4] containing her changes merged into the main repository. Next, the continuous integration and continuous inspection processes are automatically started. A continuous integration system, such as Jenkins, and a continuous inspection tool, such as SonarQube, are triggered to run the tests and analyze the static code quality of the PR. The results of the two processes are sent to the core developers, who perform manual code review and potentially request more changes. Typically, the core developers will not accept the PR until it has been modified to pass both quality gates [18].

Essentially, continuous inspection is the quality assurance process which emphasizes that code should be continuously inspected by static quality analysis tools and that all stakeholders should participate in the quality assurance process. From the perspective of contributors, the results of continuous inspection identify their bad coding habits. In addition, the continuous alerts and modifications in the process can help them to learn and memorize their quality issue patterns according to the *ebbinghaus forgetting curve*[5] [19]. This motivates us to examine whether such an automated quality assurance method can be used in the classroom setting to address the challenges in the teaching and learning of high-quality programming.

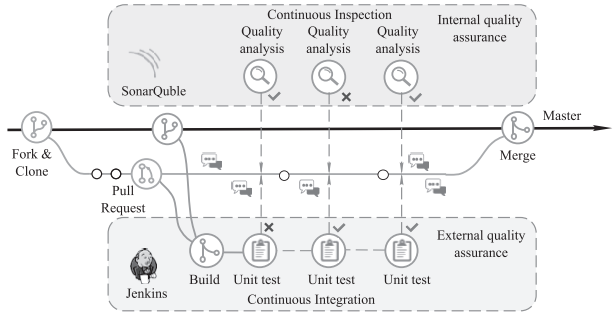


Fig. 1 The overview of automated quality assurance process for internal and external code quality in GitHub

In this study, we designed a specific continuous inspection process for students' collaborative projects to monitor, analyze and improve their programming quality. To support the process, we built an execution environment and integrated it into a popular teaching platform – *TRUSTIE* [20]. Further, with this platform, we conducted a controlled experiment to investigate how the process affects students' programming quality. In the experiment, 48 students working in 11 teams during two school years were required to complete an innovative software project. On the basis of the data, we examined the changes in the students' introduced *code quality issue densities (CQIDs)* after adopting continuous inspection, and investigated the quantities and categories of the coding rules that they learned. In addition, a post-course survey and interviews were used to understand the problems that the students encountered and their perceptions of the process. The main contributions of our research are as follows:

- We present how we designed a practical continuous inspection process for students' collaborative projects in practice.
- We find that continuous inspection can help students to identify their bad coding habits, master a set of good

[1] The external quality characteristics are those parts of a product that affect its users, e.g., correctness, reliability, and usability; while the internal quality characteristics are typically perceived by developers, e.g., readability, maintainability, and understandability of code
[2] Forking a repository allows developers to perform changes to a copy of a repository without affecting the original repository
[3] A pull request records a set of code commits and a description of the changes
[4] Branch is a lightweight mechanism in Git that supports users to concurrently perform changes on different threads
[5] The forgetting curve hypothesizes a decline in memory retention over time, and it shows how information is lost over time when there is no attempt to retain it

coding rules and significantly reduce their introduced CQIDs, thereby improving their programming quality.

- We find that the code quality issues (CQIs)[6] most frequently introduced by students during the two school years are concentrated on a few coding rules, more than half of which are coding conventions.
- We provide recommendations on the process and platform for practitioners to apply continuous inspection in classroom and field settings.

To the best of our knowledge, we present the first case study applying the continuous inspection paradigm in a classroom setting, and we believe that the results can provide actionable guidance for future related work.

2 Background and related work

In this section, we review related work on students' programming quality and introduce the background of the continuous inspection paradigm that we adopted in the study.

2.1 Students' programming quality and skills

Traditionally, software quality has been decomposed into internal and external quality attributes [14,21,22]. The external quality attributes are often reflected at the runtime stage, e.g., functionality, usability, and correctness, which can be perceived by users. An important and commonly used measure for the external quality is *defect* [23]. Correspondingly, the internal quality attributes are often reflected at the development and maintenance stage, e.g., maintainability, readability, and security, with which developers are more concerned. For poor internal quality attributes, there are some well-accepted patterns or indicators [24]. For example, the over-complexity of the methods affects the maintainability and readability of code; and unused parameters and duplications could cause security and reliability problems. In this paper, we focus on a static view of the software, while considering its internal quality of student code.

Multiple studies have shown that both the teaching and learning of high-quality programming are deficient in colleges' education systems. Breuker et al. [2] examined the internal quality of the code written by first- and second-year students based on seven sub-characteristics: size, readability, understandability, structure, complexity, duplicates, and ill-formed statements, which corresponded to 22 metrics. Using a Mann-Whitney test on the measurement results, they found that there is no clear difference between the internal quality of code produced by students in different years of study. Salman [1] investigated the differences in internal code quality between students and professionals in the context of test-driven development. He conducted Mann-Whitney tests on 20 proposed internal quality metrics, e.g., cyclomatic complexity, lines of code, unique operands count, maintenance severity, etc. The statistical tests showed that code quality differs between students and professionals in the cases of *test-last development* and *test-driven development* tasks. He also found that *test-driven development* has an effect on improving the code quality. Carver and Kraft [4] conducted an empirical study in two offerings of a senior-level computer science course to determine how well students are learning testing skills. They investigated student testing skills in two phases: manual creation of the test suite and use of Code Cover to improve the test suite. The test coverage of student code was measured with three metrics: statement coverage, branch coverage, and condition coverage. The results indicated that without a coverage tool, students achieve significantly less than 100% statement, branch or condition coverage. When using Code Coverage, students increase coverage levels.

Pair programming is an important approach in the reviewed studies to train students' programming skills and improve their programming quality. Akinola [25] compared the solo and pair programming performances of 60 student programmers in terms of their effort (comprehension time and coding time), bug occurrence (the number of bugs/errors) and effectiveness (score obtained at the end of the exercise). He divided the students into two equally sized groups and assigned both groups the same programming task. The t-test statistic results showed that pair programming improves performance relative to solo programming in terms of the factors analyzed. To determine the effects of pair programming on students' behavior and performance in a Middle Eastern society, Nawahdah and Taji [5] conducted an experiment targeting two sections of an advanced computer programming course in two semesters (30 and 29 students in both sections in the first and second semester, respectively). They observed that students in the pair-programming section produce better quality code (e.g., fewer code lines, fewer syntactic errors, and more comments, etc.) than the students in the traditional section. Braught et al. [26] conducted a controlled study that directly measured 176 students' acquisition of individual programming skills. They used course scores and survey responses to measure student performance, attitudes, and

[6] CQIs refer to the violations reported by static analysis tools

retention. Using an ANOVA test, they found that pair programming improves the individual programming skills.

Additionally, a few other approaches were proposed to improve students' programming quality. Code review is a method in which the students or tutors manually inspect the quality of code. Generally, code review can be categorized into three categories based on the role of the reviewer: self code review, peer code review and tutor code review, among which peer code review is recognized as the most practical [27, 28]. Hundhausen et al. [29] proposed a tutor-based code review process called pedagogical code review in which student code is reviewed by trained moderators. By analyzing inspection logs and exit surveys, they found that pedagogical code review improves the quality of students' code, stimulates discussions of programming issues, and promotes a sense of community. Chen and Tu [11] conducted an experiment on the programming assignments of a junior-year Windows programming course. During the process, the students were asked to keep removing 12 bad smells (e.g., long method and duplicated code). The statistical results showed that as the program assignments progressed, the average standard deviation value of smell density significantly decreased, and students' responses showed a significant improvement in code quality.

Pair programming or code review, a method to improve students' programming quality through the surveillance and alerts of the peers or tutors, is a relatively subjective and manpower-consuming task, which is in stark contrast to the continuous inspection method in our study. A detailed comparison between the two approaches is discussed in Section 6. Chen's work [11], requiring the students to keep removing code smells, is similar to the continuous inspection process in effect, while the inspection scope of code internal quality is relatively narrow (12 bad smells). In contrast, referencing the social coding methods, we implemented an automated inspection process for the pushed code. The static quality analysis tool we adopted, SonarQube, covers 7 axes of code quality (a detailed introduction to the execution environment is presented in Section 4).

2.2 Continuous code quality assurance

The *DevOps* movement intends to establish a culture and environment where building, testing, and releasing software can occur rapidly, frequently, and more reliably [30]. Automation of software quality assurance is key to the success of *DevOps* [31]. Continuous integration is a widely adopted automated method for assuring the external quality of code. Vasilescu et al. [32] analyzed historical data on process and outcomes in GitHub projects to investigate the effects of continuous integration. They found that continuous integration improves the productivity of project teams who can integrate more outside contributions without an observable diminishment in external code quality (number of bugs per unit time). It has also been introduced in the classroom setting. Bowyer and Hughes [33] conducted a study in which SE undergraduates were given a short intensive experience of test-driven development with continuous integration using an environment that imitated a typical industrial circumstance. The results showed good participation by student pairs and clear understanding of agile processes and configuration management. Heckman and King [34] presented the canary framework, employing GitHub and Jenkins for supporting collaboration, continuous integration, code analysis and automated grading. They conducted a case study in five courses with more than 3,000 students. They found the framework provided benefits of automated grading and supported the teaching philosophy of situated learning.

Continuous integration ensures the external quality of code through automating the building and testing process, that is, it does not concern the internal quality of code. Focusing on the internal quality assurance of code, SonarSource published the *continuous inspection white paper* [35] in 2013, in which the key characteristics and principles are applied in this study. The white paper discusses four types of shortcomings (e.g., lack of process ownership and pushback from development teams) in traditional code quality management such as *punctual audits* (the code review processes are usually performed by external auditors at specified intervals). In light of these shortcomings, the white paper proposes 10 principles of continuous inspection, and we list the core 4 principles: (1) managing software quality must be everyone's concern from the beginning of development; (2) software quality requirements must be objective; (3) stakeholders must be alerted when new quality flaws are injected; and (4) software products must be continuously inspected. The essential goal behind continuous inspection is to find problems early when fixing them is still inexpensive and easy.

As an internal quality assurance paradigm, continuous inspection has been widely adopted in open source software communities and industrial companies [36]. However, to the best of our knowledge, few if any existing studies have introduced this popular method into the educational context or have reported the effects on students' programming quality. This study is therefore designed to understand the effects and challenges of continuous inspection in the classroom setting.

3 The continuous inspection process

We inherit the principles of continuous inspection [35] and design a quality assurance process by integrating continuous inspection into students' collaborative projects. In particular, the process emphasizes that (1) the students' code changes must be continuously inspected by static quality analysis tools; (2) all projects members should participate in the quality assurance process; and (3) stakeholders must be alerted when new CQIs are injected. The process is based on the Git branching collaboration model, which is commonly adopted in industry [37, 38]. Typically, a collaborative development team is composed of more than two students, and includes a team leader role, which is responsible for team management, including task assignment and contribution collection. The process (see Fig. 2) begins with the team leader creating the project repository. Then, each member forks the repository and clones it to the local Git repository. The team leader submits code on the *develop branch*, and the members develop on their own branches. After making changes, e.g., implementing a feature or fixing a defect, the members push their local changes to the remote repository. The quality analysis processes are then automatically triggered, and the members should fix the CQIs reported by the static quality analysis tool (see mark ① in Fig. 2). When the members decide to merge their contributions to the leader's branch, they first pull the latest code on the *develop* branch to resolve the conflicts[7] caused by concurrent changes. Then, they open PRs, expressing their readiness to have the branches containing their changes merged into the leader's *develop* branch. The team leader and members can review the PRs in *TRUSTIE* and discuss whether to merge them. During this process, the leader may require the members to make additional changes, based on the results of the review process, which can lead the members to repeat the pushing and fixing actions and result in updated PRs. When the PRs are accepted and merged to the *develop* branch, the quality analysis processes are triggered, which can again lead to the CQI fixing actions (see mark ② in Fig. 2). After building a new version, the team leader merges the code on *develop* to the *master* branch, which stores stable releases of the software system. In addition, we stipulate the following regulations that the students should follow during the process:

- Each team should fix or close all CQIs of the projects before starting the continuous inspection process.
- Each student is responsible for fixing the CQIs of her modules on her own branch.
- The CQIs should be fixed within 24 hours after being reported by SonarQube.
- All CQIs on the *master* branch should be fixed or closed. If a CQI is marked as "*won't fix*", reasons should be given as comments in SonarQube.
- The descriptions of commits or PRs aiming to fix CQIs should be specified as "Fixing CQIs (...)".

4 Methodology

To evaluate the effectiveness of the continuous inspection process on students' programming quality, we conducted a two-year controlled experiment with the students in our college. We are interested in investigating the improvement in

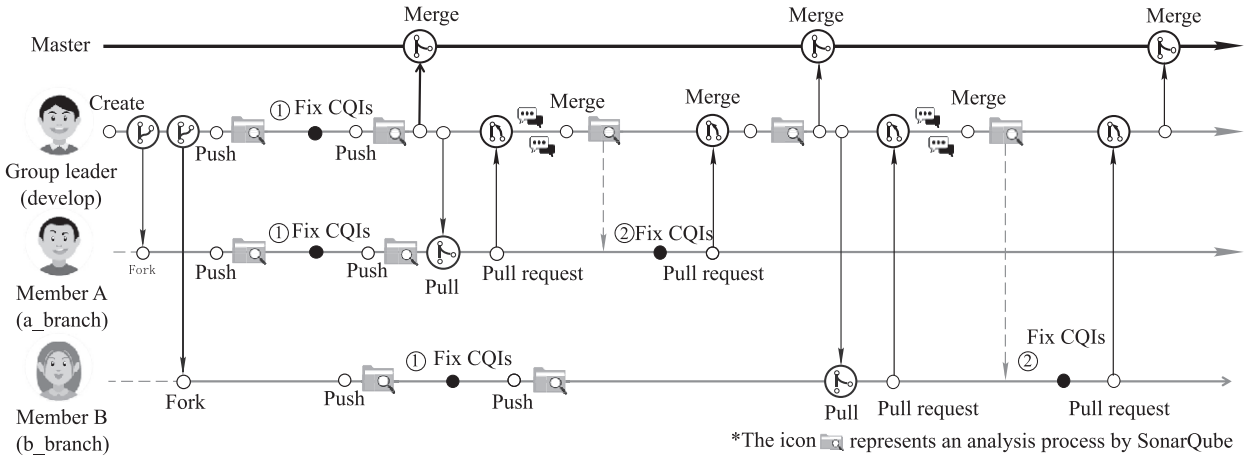


Fig. 2 The continuous inspection process for students' collaborative projects

[7] Conflicts occur when *Git* tries to merge two branches that have changes in common code pieces

students' programming quality and their violated coding rules through the process. In addition, we want to understand the issues they encountered during the experiment. Specifically, we consider the following research questions:

RQ1: To what extent has the students' programming quality improved after adopting the continuous inspection process?

RQ2: What are the quantities and categories of the introduced CQIs through the continuous inspection process?

RQ3: What challenges do the students face when working with the continuous inspection process?

To answer our research questions, we followed a mixed-method approach combining quantitative and qualitative methods [39]. We analyzed the CQI data on the whole revision histories of the students' repositories and obtained the students' perceptions through a post-course survey and semi-structured interviews.

4.1 Setting

4.1.1 The courses

In our college, the undergraduates majoring in SE have to take a series of three mandatory courses, which are spread over the autumn, spring and summer semesters. During the autumn semester, in addition to studying the curriculum knowledge, the students are required to read the code of an open source Android project, and then add some new features. During the spring and summer semesters, the courses are centered on a software-development project: the students have to come up with a novel idea on a software system and collaboratively develop the project using the *iterative development model*. They are required to accomplish four iterations (two iterations in each semester). At the end of each iteration, they submit an executable version of the software and its related documents. During the first iteration in the spring semester, the teacher teaches related theoretical and empirical knowledge in the class combining the students' work at the beginning of each phase (e.g., requirement and design). After the first two iterations of the course project in the spring semester, the students are usually more familiar with the programming language, development environment and pull-based collaborative development process. Therefore, in the summer semester, the course is intended to help students to improve their programming quality through the final two iterations, to achieve higher educational goals. The summer semester lasts only for approximately 20 days, which is much shorter than the spring semester. However, during the summer semester, the students are required to spend all day on this single course. Thus, the actual time they spent on the course is no less than in the previous semester. Moreover, the students' continuous participation during the summer semester provides conditions for us to implement the continuous inspection process. In the summer semester, each team was required to fix all CQIs in the initial analysis process in the first two days of the semester. Note that the teacher did not teach good coding rules or practices in the class throughout the two semesters.

4.1.2 Participants

We conducted the experiment on the students in the same courses during the 2016 and 2017 school years. During the 2016 school year, we had a total number of 22 students in the class, who were grouped into 5 teams. During the 2017 school year, the number of the students increased to 26, forming 6 teams. All the students were male juniors majoring in SE, and all the teams consisted of 4 or 5 students. In addition, we had two classroom teachers and three teaching assistants (TAs). Each teacher and teaching assistant supervised a project team on their progress in development, artifacts and tool use.

4.1.3 Projects

At the beginning of the project phase in the spring semester, each team was given one week to propose innovative ideas and requirements for their projects. In the end, the students produced original software systems covering different domains, e.g., AR-Navigation Android apps, smart library robots, and multi-drone rescue systems. All projects have used Java as the main programming language, and more than 5,000 lines of source code were written by the students in each team.

4.1.4 The continuous inspection platform

In the courses, we leverage *TRUSTIE* [20] *(Trustworthy software tools and integration environment)* to support our course teaching and to execute the quality assurance process. *TRUSTIE* is a popular platform among Chinese universities that enables collaborative learning (e.g., resource sharing, homework assignments, and discussion forums) and collaborative development (e.g., issue tracking, task assignment, version control, and PRs & comments).

The collaborative development environment builds on Git and GitLab. We built the continuous inspection environment by integrating SonarQube into the collaborative development environment. SonarQube is a popular static analysis platform that supports more than 20 code languages and covers 7 axes

of code quality: *architecture & design, duplications, test coverage, complexity, potential bugs, coding standards* and *comments*. It is a web-based application and provides a powerful plugin mechanism to support users in adding new languages, rules and integrations. The quality characteristic model of SonarQube is based on the SQALE (software quality assessment based on lifecycle expectations) methodology, which is a generic quality model to support the evaluation of nonfunctional requirements related to the code quality [40]. We used the popular automation server Jenkins to connect the code repositories in GitLab and SonarQube deployed in *TRUSTIE* (see Fig. 3). By configuring some settings in Jenkins, we implemented an automated process to analyze the latest revision in the code repositories when new code changes were pushed or merged to the GitLab server. After installing the Git plugin, SonarQube can automatically detect the introduced commit of a CQI using the *git blame* command and display the relevant information in the source code view (as shown by the white dialog box in Fig. 4). This feature allows us to easily obtain the CQIs introduced by a student. Note that the version of SonarQube that we used is 5.6, with the Java plugin (SonarJava) version of 4.1.

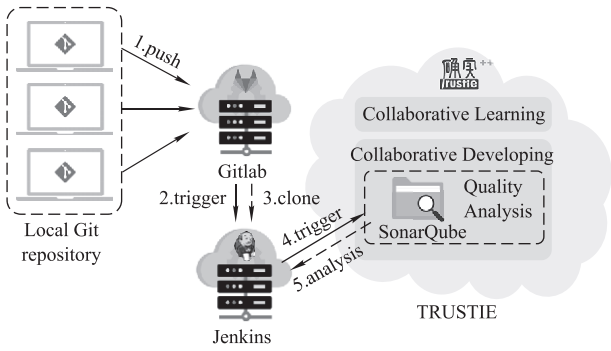


Fig. 3 The quality analysis framework in TRUSTIE

Fig. 4 The screenshot of students' CQIs displayed in SonarQube

4.2 Research methods

To investigate the effects of continuous inspection on the students regarding their programming quality, we designed the controlled experiment from two dimensions. The first dimension examined the programming-quality changes in the student individuals before and after they participated in the continuous inspection process. Second, to examine whether the effects (if any) were caused by the process, we compared student CQID changes between two groups that did adopt and did not adopt continuous inspection. Because the number of the teams in 2016 was odd and relatively small, they all participated in the continuous inspection process. Accordingly, in 2017, we randomly divided the 6 teams into two groups (3 teams in each group): the experimental group, which adopted continuous inspection, and the control group, which did not. The experiment was executed in a controlled setting to eliminate possible effects of potential disturbing factors, for example, the students were supervised by teachers or TAs to submit code patches using their own Git account in the form of a pull request, rather than merging code using "copy and paste"; the status of the collaborative development environment was continuously monitored in case of unexpected service crashes. The students were required to continuously participate in our experiment to avoid absence such as a leave.

4.2.1 Data collection

To answer *RQ1* and *RQ2*, we collected quantitative data that record students' development history and the introduced and fixed CQIs. We also collected students' perceptions regarding *RQ1* and *RQ3* through a survey and interview.

Quantitative data Given that all the projects used Java programming language, we only analyzed the code quality of the Java modules. The students used Git to update the related documents in addition to code changes. Therefore, the commit histories contain document updates, non-Java code changes and Java code changes. Additionally, we observe that the commit history contains some large commits which are used to resubmit the whole project or to clean the content on branches; thus, we call these commits *non-change commits*. Table 1 summarizes the data on the commits of the 11 teams throughout the two semesters.

By qualifying the code quality, CQIs are used to indicate which good coding practices are violated in the static analysis. The good coding practices manifest as coding rules in SonarQube. Accordingly, while running an analysis, SonarQube raises a CQI when a piece of code breaks a coding rule. Note that SonarQube stores CQIs in an incremental way: if new CQIs are introduced, it stores them in the *MySQL* database; otherwise, it updates the statuses of existing CQIs when their statuses are changed (e.g., fixed in code or removed by users in SonarQube). In addition, if a

Table 1 Statistical data of commits of the teams

Team	Project type	Total code lines	Java code lines[1]	Total commits	Java commits	non-change commits	Java authors	Total CQIs
2016 school year								
T1	Android app	5,143	4,094	199	56	20	5	2,434
T2	PC game	80,552	52,399	204	31	17	3	9,542
T3	NAO robot	20,081	1,481	194	71	3	3	5,425
T4	NAO robot	37,886	8,935	195	98	1	5	4,138
T5	Android app	10,830	6,534	237	89	26	4	24,790
2017 school year								
T6[2]	NAO robot	11,005	5,586	111	61	14	5	12,079
T7	Multi-drone system	17,407	9,973	219	128	10	4	17,770
T8	Smart device	9,370	6,949	184	109	10	4	5,433
T9	NAO robot	7,520	2,067	78	27	1	4	936
T10	Multi-drone system	11,448	2,120	107	68	6	4	3,799
T11	Website	9,810	2,480	78	11	4	4	428

[1]The Java code lines contain both open-sourced and changed code lines
[2]In the 2017 school year, T6, T8, T10 are included in the experimental group, and T7, T9, T11 are included in the control group

CQI is closed, we cannot obtain its introduced commit information through the web service API. Therefore, to obtain all the CQIs introduced throughout the development history, we wrote a script to analyze the revisions that had touched *.java* files (excluding the *non-change commits*), extracting the corresponding introduced commit data. Consequently, three data sources formed our data set: the Git repositories, the SonarQube database in *TRUSTIE* and the extracted CQI data. In this study, the SonarQube Java plugin that we used contained 141 rules, which were all applied in the inspection process. The detailed description of the rules can be found at a GitHub project on this work: roadfar/continuous_inspection_in_classroom.

Qualitative data We sent an anonymous online survey to the students after the final lesson of the course. The survey had 10 questions, including single-choice, Likert-scale, and open-ended questions. Specifically, we used four questions to ask the students about their practices before and through the process of continuous inspection and about their attitudes toward the coding rules in future programming practices. We also used five questions (including single-choice and Likert-scale questions) to obtain student perceptions of the introduced CQIs and their attitudes toward the continuous inspection process. Lastly, one open-ended question was used to understand the issues they encountered throughout the process. Consequently, we received 22 answers and 13 answers in 2016 (all the students participated in the continuous inspection process in the 2016 school year; thus all of them participated in our survey) and 2017 school years (the survey was targeted to the experimental group in the 2017 school year), respectively. The average time taken to answer the questionnaire was 114 seconds.

In addition, we conducted interviews with 10 students (4 and 6 in 2016 and 2017, respectively) who worked actively during the continuous inspection process. Our goal in these interviews was to understand, in greater detail, their work practices, their perceptions of the process, and the problems they encountered with the platform and process. The interviews were semi-structured based on 9 guiding questions, and the interviewer could dig deeper with additional questions when appropriate [41]. Each interview was recorded and lasted approximately 15 to 30 minutes.

4.2.2 Analysis

We conducted hypothesis testing to examine the changes in student programming quality before and after they participated in the continuous inspection process. In this study, *programming quality* refers to the ability to build software with high internal code quality, e.g., good coding style, low complexity, and best coding practices. Bad code quality patterns can be identified as CQIs by static analysis tools, and more CQIs in a code file indicate lower internal quality of the code. Therefore, we used the *code quality issue density*, that is, the number of introduced CQIs per changed code line, to measure programming quality without considering the differences in severity among CQIs [17]. We calculated the number of changed code lines of a student using the *git log* command, which counts the added and deleted lines [42]. Note that the changed code lines did not include autogenerated code such as the code in R.java and BuildConfig.java files. We used the *nonparametric Wilcoxon signed-rank test* for paired samples to test for a difference in the mean CQIDs of students in the control and experimental groups. We used the p value to de-

termine statistical significance, with the significance level set to $\alpha = 0.05$.

5 Results

In this section, we report the results of the quantitative and qualitative methods. When quoting survey respondents and interviewees, we refer to them using [YrX] notation and [YiX] notation, respectively, where Y is the school year (16 and 17 correspond to 2016 and 2017, respectively) and rX or iX represents the respondent's or the interviewee's ID.

5.1 RQ1: programming quality improvements

5.1.1 Changes in the introduced CQIDs

We first compared the introduced CQIDs of the students who adopted continuous inspection between the two phases, i.e., the spring semester when they did not adopt continuous inspection and the last quarter of the summer semester when they did adopt continuous inspection. The average introduced CQID of a student in a phase (before and after using continuous inspection) was calculated by the average CQID on the commits (not including non-change commits) during that phase. When calculating the number of CQIs, we did not include the CQIs that raised in autogenerated code files, e.g., the R file in Android[8]. 10 and 16 students in the 2016 and 2017 school years respectively submitted Java code in both phases (the remaining students were responsible for non-Java modules or did not submit Java code in both periods). The average number of changed code lines for these students is 1672, with the minimum, median, and maximum numbers of 269, 806, and 5987, respectively. The introduced CQIDs are shown in Fig. 5: s1-s10 represent the students in 2016 and the remaining represent students in 2017. Among the 16 students in the 2017 school year, s11-s20 represent the students in the experimental group and the remaining represent the students in the control group. The results show that the number of introduced CQIDs significantly decreased for the majority of the students after adopting the continuous inspection process, and the results of the *Wilcoxon signed-rank paired test* for the students in both school years confirmed the significance: p values of 0.037 and 0.004 for the 2016 and 2017 school years, respectively (p<0.05). Furthermore, we conducted the same analysis on the control group who did not adopt continuous inspection in the 2017 school year, and the results did not show significant differences between introduced CQIDs (p value of 0.590). This result is consistent with previous work [2], which shows that there is no clear difference between the internal quality produced in different years without specific training procedures. Therefore, we reject the hypothesis that the introduced CQIDs of students adopting continuous inspection in the two phases are identical and conclude that the number of introduced CQIDs significantly decreased after adopting the continuous inspection process. To further understand the CQID changes of the students who adopt continuous inspection in the process, we divided the summer semester (20 days) into 4 phases (5 days for each phase) and analyzed the changes in average CQIDs throughout the 4 phases. The average CQIDs of individual students in the 4 phases and the corresponding distributions

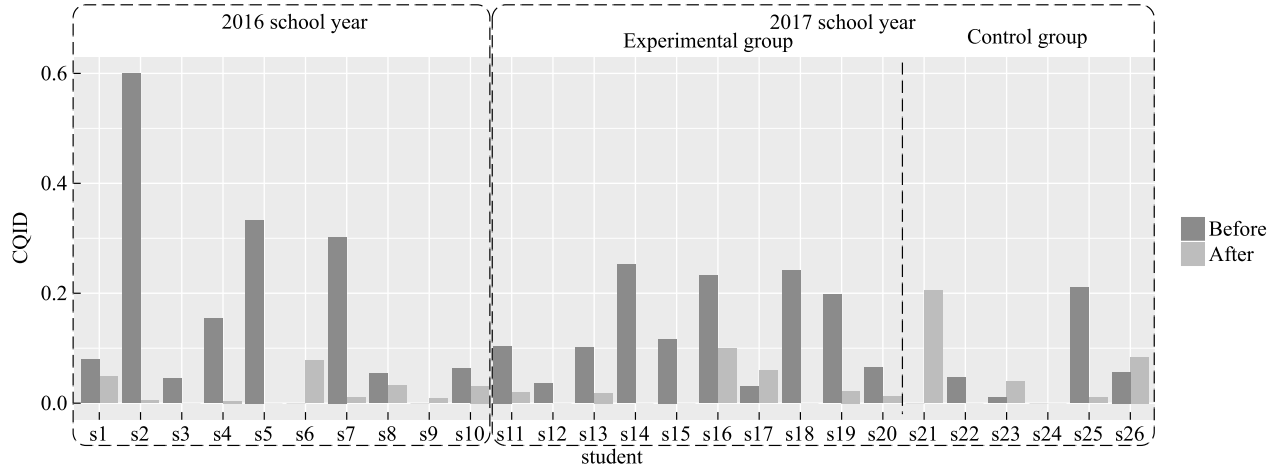


Fig. 5 Students' introduced CQIDs before and after adopting continuous inspection

[8] In the poster version (DOI: 10.1145/3183440.3195054) published in the 40th International Conference on Software Engineering (ICSE 2018), we did not exclude the CQIs raised in autogenerated files, which leads to slight differences in the students' introduced CQIDs, but the results of the hypothesis testing still hold

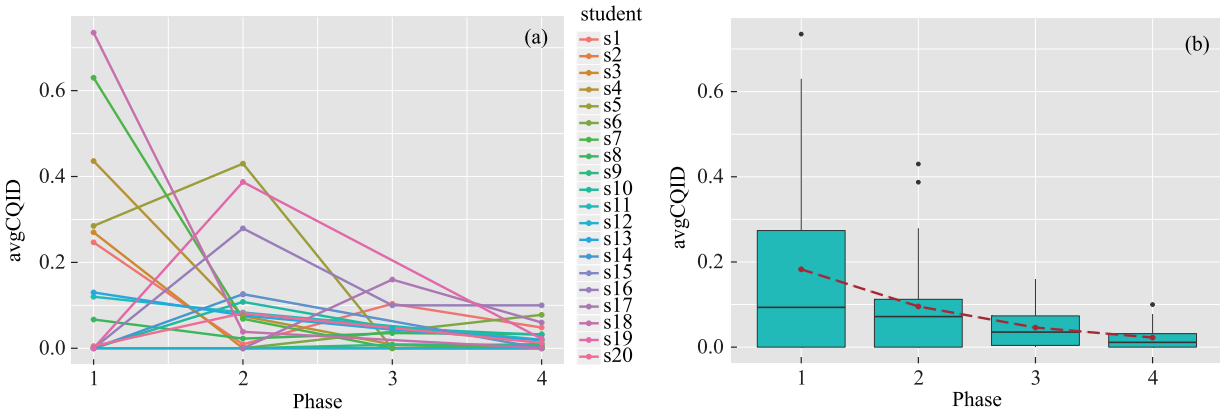


Fig. 6 The changes in the students' average CQIDs in the four phases. (a) Each student's average CQIDs in the four phases; (b) the distribution of students' average CQIDs in the four phases. The red points represent mean values of CQIDs of each phase

are summarized in Fig. 6(a) and Fig. 6(b), respectively. We observe that in the first 5 days, the students' average CQIDs distribute in a relatively wide range. In the second phase, the students' average CQIDs fall into a narrower range, and the third quartile and mean value decrease by approximately half. In the last two phases, the students' average CQIDs generally continue to decrease, and the mean and median value reduce to 0.02 in the last phase.

5.1.2 Perceived impacts of continuous inspection

We provided the students who adopted continuous inspection with a set of 3 questions with a 5-level Likert scale to query about their perceptions of the process; the answers are presented in Fig. 7. The results show that the majority of the students (more than 93%) agreed that the process helped them to recognize bad coding habits and improve their programming quality. In addition, more than 91% of the students agreed either "mostly" or "strongly" that it is necessary to introduce continuous inspection in software development practices, while the remaining respondents felt uncertain. When answering the last open-ended question where they could address anything about continuous inspection, 43.2% of the students expressed positive feelings about the process. For example, "*I have learnt many poor practices that I haven't perceived before, and it does help me to improve programming capability*" [16r4] and "*It (the process) is very useful, and can help a student become an excellent programmer*" [16r6]. Moreover, some students mentioned the significance of SonarQube: "*SonarQube is a cool tool, and I will always use it in the future. I believe it will improve my programming quality*" [16r19]; "*The majority of the reported CQIs make sense. Unexpectedly, however, it does not even allow commenting code, and I have to get used to it gradually*" [17r12].

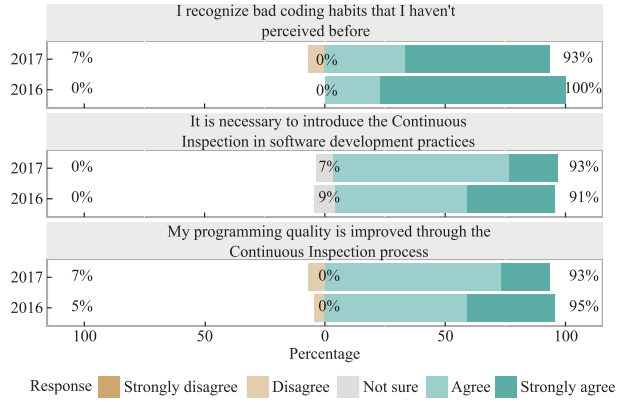


Fig. 7 The students' perceptions of the process

Furthermore, in the interview, some students described their detailed experiences of the process. One student in *Team 1* described his feelings and mentioned the incentive mechanism of the process:

> *"(...) The teachers ask us to fix all CQIs analyzed by SonarQube, which means you should better consciously avoid introducing the issues that are already known. I think it is an important incentive factor to help me build good coding habits."* [16i2].

Some other students mentioned the effects of continuous inspection on software quality, e.g.,

> *"I think continuous inspection is meaningful. At an earlier stage, I write code assuming that the app performs in ideal cases, without considering the exceptional situations, which leads to low software reliability. Continuous inspection can help me find and address such issues."* [17i2].
>
> *"Continuous inspection helps us find and solve code issues at early stages, avoiding running into too much technical debt or even introducing bugs."* [17i7].

Overall, our experimental results show that the students' introduced CQIDs significantly decreased after adopting the

continuous inspection process, while the students' introduced CQIDs in control group did not show significant differences between the two phases. Therefore, we can conclude that the adoption of the continuous inspection process leads to the decrease of the students' introduced CQIDs. Our qualitative results reveal two main mechanisms of continuous inspection which make the results happen: (1) the regulation that every member should fix their introduced CQIs stimulates the students to consciously avoid introducing known CQIs to reduce the workload (the transparency of everyone's contribution quality might be another reason, which was not reported by the students); (2) the 'continuous' reporting and fixing activities help them remember the CQI patterns, thereby building good coding habits.

5.2 RQ2: the quantities and categories of introduced CQIs

5.2.1 All CQIs and rules

We first analyzed the quantities and categories of the CQIs that the students introduced throughout the continuous inspection process. By default, SonarQube sets one or more tags for a coding rule, and one tag covers one or multiple aspects of the 7 axes of code quality. Accordingly, we classified the CQIs with multiple tags into the first category. To understand the students' programming practices, we excluded the CQIs that are raised by *non-change* commits or the CQIs introduced by autogenerated code. Table 2 lists the top 80% of the CQIs introduced by the students during the two school years (82.3% and 81.0% for the two school years, respectively).

We observe that the number of violated rules for the CQIs introduced by the students in both school years approximately follow the *Pareto Principle*: 80% of the CQIs come from 20% of the rules. The high *Gini coefficients*[9] for the number of violated rules of each team (ranging from 0.59 to 0.83) indicate that the introduced CQIs are concentrated in a few coding rules. These rules, however, account for only a small proportion of all Java rules in SonarQube: 14.9% and 17.7% in the 2016 and 2017 school years, respectively. Furthermore, we analyzed the categories of the introduced CQIs. Interestingly, the top 6 categories of the CQIs introduced by the students in the two school years tend to be concentrated in the same categories, even having a similar distribution. In particular, Table 2 shows that the most frequently introduced CQIs concern *coding convention*, which constitutes approximately half of all CQIs. To understand the typical CQIs introduced by the students, we analyzed the categories of the rules violated by the most students (see Table 3). The results indicate the common poor coding habits of the students, e.g., "commenting out" code, reserving unused imports, improperly naming constants.

Table 2 The categories of the introduced CQIs and violated rules

Category	CQIs	CQIs pct.[1]	Rules[2]	Rules pct.[3]
2016 school year				
Convention	20,032	53.8%	15	10.6%
CERT[4]	3,967	10.7%	40	28.4%
CWE[5]	1,925	5.2%	6	4.3%
Misra[6]	1,834	5.5%	4	2.8%
Unused	1,521	4.1%	2	1.4%
Clumsy[7]	1,358	3.6%	16	11.3%
2017 school year				
Convention	22,731	47.7%	16	11.3%
CERT	5,973	12.5%	31	22.0%
Misra	3,137	6.6%	4	2.9%
Clumsy	2,311	4.9%	15	10.6%
Unused	2,306	4.8%	2	1.4%
CWE	2,155	4.5%	6	4.3%

[1]The *CQIs pct.* refers to the percentage of CQIs in a specific category to all CQIs
[2]The *Rules* are the number of violated rules corresponding to the CQIs in a specific category
[3]The *Rules pct.* refers to the percentage of the rules in a specific category to all Java rules
[4]The CERT standard. Most of the CERT rules are good programming practices and not language specific
[5]Common Weakness Enumeration is a formal list of software weakness types
[6]Best practice guidelines for the safe and secure apps
[7]Extra steps are used to accomplish something that could be done more clearly

Table 3 The typical CQIs of the top six categories

Category	Rule example	Students pct.
Convention	Constant names should comply with a naming convention	69.7%
Unused	Useless imports should be removed	75.8%
CERT	Exception handlers should preserve the original exceptions	75.8%
CWE	Class variable fields should not have public accessibility	72.7%
MISRA	Sections of code should not be "commented out"	78.8%
Clumsy	Collection.isEmpty() should be used to test for emptiness	42.4%

5.2.2 The violated rules in the process

We further analyzed the students' practices of violating coding rules in the continuous inspection process. In the summer semester, the median and mean values of students' commits are 18.00 and 23.15, respectively, and the median and mean values of the violated commits for the same rules per student are 2.00 and 2.10, respectively, indicating that more than half

[9] The Gini coefficient is an econometrics measure used to measure income concentration; the highest Gini coefficient for a country is approximately 0.6, signifying that a few people have cornered most of the income

of the rules are violated in no more than two commits by the same students. Based on this preliminary result, we analyzed the violated commits for the same rules of individual students who have participated in the continuous inspection process. The results are presented in Fig. 8.

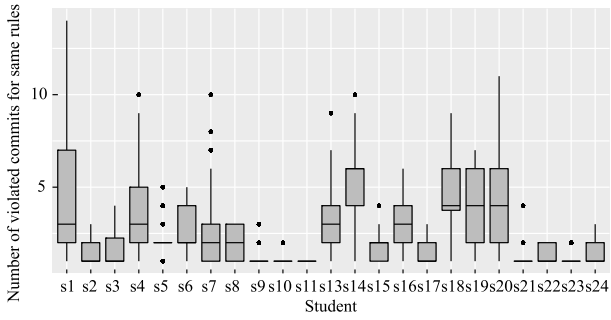


Fig. 8 The distribution of students' violated commits of the same rules in the continuous inspection process

S1-s20 correspond to the students (s1-s20) who are investigated in Section 5.1. s21-s24 are the students who adopted the continuous inspection process but did not submit Java code in both phases. The results show that the median values of the number of violated commits for the same rules of 79.2% of the students (19) are less than three. We then investigated the top ten rules that the students repeatedly introduced (see Table 4). We found that these rules are not always easy for students to follow during the code maintenance phase. For example, it is challenging and sometimes unnecessary for the students to write comments for every public APIs or fields (the first rule in Table 4); it is useful to reserve necessary code pieces in the form of comments for potential future use (the second rule in Table 4), and it is sometimes difficult to maintain the complexity of methods with the goal of implementing additional features (the fourth rule in Table 4).

In sum, the CQIs introduced by the students in the two school years are concentrated in a few of the same coding rules and categories, with approximately half of the rules concerning coding conventions. For more than half of the violated rules, the majority of the students violate the same rules in no more than two commits, and the rules most repeatedly violated by the students are not always easy for students to follow during the code maintenance phase.

5.3 RQ3: the challenges

To find the pain points experienced when implementing the process, we introduced a mandatory open-ended question in the survey and asked the students to state the challenges they faced during their work practices. In addition, we also communicated with the students about this subject during the interviews. We learned that the challenges revolve around three main aspects: *repetitive fixing work*, *tool issues* and *workload*.

Repetitive fixing work The work coupling among the students and the mechanism of the distributed collaboration development model raise issues of *work conflicts*, which occur when Git tries to merge two branches that have changes in common code pieces. For example, both student *A* and student *B* change the tenth line of the same code file on their own branches, and *A* tries to merge *B*'s branch; then, a conflict occurs. Moreover, improper task assignment by the team leader (e.g., the coupling between the tasks that are assigned by the leader is high) and low-quality structure of code (i.e., the coupling between the modules is high) in the classroom setting make this issue common. A few students mentioned this issue in the survey, e.g., "*Conflicts would happen when multiple students submit PRs*" [16r11] and "*Resolving conflicts is a difficult work*" [16r4]. Worse still, CQIs are sometimes raised repeatedly as a result of resolving conflicts, thus necessitating the repetitive work of fixing CQIs. For example,

> "*I have ever met such scenes: I submitted a code patch and then fixed the CQIs reported by SonarQube, but a teammate overrode my code, and same CQIs were reported again. Consequently, when I pulled the latest code to my branch, I had to fix them again.*" [16i1].

The case above describes a common situation: when resolving conflicts, a student usually tends to retain their own code pieces, thereby causing repetitive reports of CQIs.

Tool issues *False positives* frequently occur in static analysis tools [43] and are mentioned by the students in our study as well. For example, "*False positives exist in SonarQube, which should be improved*" [17i9,17i19], "*There are some*

Table 4 Statistical data of commits and violated commits of the same rules in the continuous inspection process

Rule description	Category	Commits	CQIs	Students	Students pct.
Public types, methods and fields (API) should be documented with Javadoc	Convention	113	14,428	23	95.8%
Sections of code should not be "commented out"	Misra	88	4,637	21	87.5%
String literals should not be duplicated	Design	84	616	22	91.7%
Methods should not be too complex	Brain-overload	76	961	22	91.7%
Useless imports should be removed	Unused	76	3,644	19	79.2%

false positives reported by SonarQube, and similar issues are usually reported at many code pieces. In this case, I have to mark all of them as "won't fix" one by one, which spends a lot of time." [16i3]. Another student described a specific case of false positives that he encountered:

> "*The R file is automatically generated by Android SDK, and the naming convention of member variables does not meet Java naming standards, while SonarQube reports issues on all of them. More than that, since the file is updated after each compiling process, the issues marked as "won't fix" would be reported again after the latest changes are pushed to the server.*" [16i2].

The *utility of the CQIs* is also mentioned by respondents in the survey: they think that resolving some CQIs is useless for improving code quality, e.g., "*Some CQIs reported by SonarQube make no difference to code quality*" [16r18], "*Some CQIs are too simple*" [16r9], and "*The ability of SonarQube to find CQIs is relatively complete, while some CQIs are still over sensitive*" [16r19, 17r27]. In this regard, a student suggested rule customization of SonarQube in the interview: "(...) It is best to provide the rule customization feature, because every team has her own coding convention" [17i14]. Additionally, *the difficulties in understanding coding rules* are mentioned by some students, e.g., "*(...) I hope that the platform can add specific explanations and solution examples to the CQIs*" [16r14] and "*SonarQube should provide fixing recommendations for the CQIs*" [17i9].

Workload We asked the students to fix newly reported CQIs in one day. Some interviewees expressed *the difficulties in fixing CQIs in time*. For example,

> "*I didn't have enough time for fixing all CQIs after submitting code patches. In most cases, I had not completed the coding work until night, and new tasks would come in the next day. So, I would fix high-severity CQIs first.*" [16i2].

Another student expressed his concern about the granularity of commits when asked about the strategies adopted to follow our requirements on CQI fixing time:

> "*Well, I think it depends on the amount of code you submitted. If you change less than 100 lines of code and get a few CQIs, it is easy for me to fix them in one day; otherwise, if you have not submitted code changes until having modified hundreds of lines, it will be difficult for me to fix the CQIs in a short span of time.*" [16i4].

6 Lessons and discussion

We now discuss the lessons learned through the two-year experiment and provide recommendations that could help to streamline the experience for practitioners in related work. Moreover, we contrast the continuous inspection paradigm with the well-known pair programming method in terms of improving students' programming quality.

6.1 Recommendations

6.1.1 Recommendations for the process

We present the guidelines for the process based on three aspects: the first one is geared toward students while the others are geared toward teachers.

Minimizing contribution friction. Minimizing contribution friction is recommended for contributors when addressing the granularity of PRs in social coding sites, for which small and isolated contributions are easier for integrators to process and the impact of each change is more easily evaluated [44]. In the classroom setting, especially when introducing continuous inspection, it is even more strongly recommended for students to minimize contribution granularity when submitting contributions. The results for *RQ2* and *RQ3* show that the introduced CQIs are concentrated in a few coding rules, and work conflicts often lead to repetitive fixing work. In this regard, small and isolated contributions can reduce the number of times the same rules are reported as well as conflicts, thus improving the development efficiency.

Teaching coding rules in class. Through the continuous inspection process, the coding rules that the students have mastered are largely determined by what they introduce. The variety of the coding-skill requirements and module types make the coding rules learned by the students diverse with respect to category and limited in quantity. Therefore, teachers could combine the students' autonomous coding practices and classroom teaching. It is recommended that teachers summarize the coding rules that are easily introduced by the students and share them with all the students in the class. Additionally, they can teach and explain the coding rules that the students find difficult in class.

Customizing coding rules. As reported by the students, the utility issue of CQIs sometimes puzzles them. To address this issue, teachers can customize the coding rules based on the teaching requirements: they can choose valuable coding rules or even add new rules, ignoring low-utility ones. Furthermore, they can customize coding rules based on the internal quality measures of students' code proposed by previous studies [2, 45].

6.1.2 Recommendations for the platform

We first provide recommendations on the deployment of the continuous inspection environment.

The deployment of the continuous inspection environment. In our work, we implemented the automated analysis process by using Jenkins to connect the code repositories and SonarQube. The latest version of GitLab community edition (free) has provided features for the continuous integration/deployment workflows. Accordingly, practitioners can use this function to configure SonarQube as the continuous-inspection tool. In addition, the GitLab Ultimate edition (paid) has provided a complete set of solutions to the continuous inspection process, which uses SonarQube as the code analysis tool. Hence, users can easily deploy a continuous inspection environment by using the GitLab Ultimate edition.

Our work also uncovers several aspects for improvement of the continuous inspection platform.

Personalized guidance in learning coding rules. To support the students in learning more coding rules, in addition to the violated rules, it is recommended that the platform provides personalized learning paths for programming quality for the students. In particular, the platform can '1learn" the students' development histories to continuously and automatically recommend the potential coding rules that they have not yet mastered.

CQID rank list. To improve the students' motivation for writing high-quality code, the platform can present a CQID rank list on the project page, displaying the project members' submitted code lines, introduced CQIs and CQIDs, and fixed CQIs. As with the contribution boards of social coding sites, such a transparency mechanism of contribution and quality can stimulate the students to improve their technical skills and manage their reputation [16], as well as facilitate the teachers' evaluation work.

Intelligent recommendation of repair scheme. In our experiment, the students often face difficulty in understanding the coding rules and finding suitable solutions. In this regard, in addition to providing more specific explanations and repair examples, the platform can recommend potential fixing schemes for the reported CQIs. As demonstrated by the results of *RQ2*, similar CQIs can be reported in different code pieces due to cloned code. Therefore, the CQI fixing history could be instructive for fixing new CQIs, especially in the same project context. Accordingly, the platform could recommend potential fixing schemes through mining the fixing histories. Similar methods have been adopted in previous work [46] to recommend bug-fixing schemes.

6.2 Comparison of continuous inspection and existing methods

As a coding practice advocated by extreme programming [47], pair programming requires that teams of two programmers work simultaneously on the same design, algorithm, code, or test. Sitting shoulder to shoulder at one computer, one member of the pair is the "driver", actively creating code and controlling the keyboard and mouse. The other member (the "navigator") constantly reviews the keyed data in order to identify tactical and strategic deficiencies, including erroneous syntax and logic, misspelling, and implementations that do not map to the design [48]. Compared with pair programming, code review requires certain reviewers to periodically walk through the students' programming assignments rather than monitoring the code pieces that the students are typing. The code inspection tool in continuous inspection plays a role similar to the "navigator" in pair programming and the reviewer in code review. Table 5 summarizes their main differences regarding six aspects. Among the three approaches, the inspection frequency of code review is relatively low. In contrast to pair programming and code review, the inspection approach of continuous inspection is objective and professional. The automated inspection process saves additional manpower compared to both code review and pair programming, thus providing more opportunities for students to conduct programming practices, which is critical for students' collaborative programming projects. Although both the inspection approaches of pair programming and code review are manual, the quality inspection criterion of code review [29] (especially the tutor-based code review) seems more professional and consolidated, the quality data are traceable, and the support for distributed development is better. Previous studies have shown that pair programming [5,25,48–50] and code review [27,29,51] are conducive to improving students' programming quality as well as guaranteeing the quality of code. Further evaluation of their differences in performance regarding learning effects and development efficiency is required.

7 Threats to validity

Despite our best efforts, there are several threats to the validity of the results of this study. The three subsections below present the threats to the internal, external and construct validity.

Internal validity The validity of the results of this study is based on the validity of the tool that we use: SonarQube.

Table 5 Main differences among continuous inspection, pair programming, and code review

	Continuous inspection	Pair programming	Code review
Inspection frequency	continuous	realtime	periodical
Inspector	static analysis tool	peer	tutors or students
Inspection objectivity	objective	subjective	subjective
Inspection scope	the whole project	specific code pieces	the whole project
Traceability[1]	traceable	untraceable	traceable
Support for distributed development	supported	unsupported	supported

[1]Traceability refers to whether the code quality data and developer activity data are traceable for a method

The concepts and taxonomy we use are the default values set by SonarSource. The characteristic models 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 way [40]. A common issue of static analysis tools is false positives [43], which occur in our study. To deal with this issue, we took the following measures: (1) we asked the students to mark the false positives as 'won't fix' CQIs and leave the reasons (specified in Section 3), which is a handling method for false positives provided by SonarQube; (2) when conducting quantitative analyses, we excluded the commonly-reported false positives (e.g., the CQIs reported in R.java and BuildConfig.java files) and excluded the CQIs with "won't fix" tags.

The post-course survey and interview are used to qualitatively assess the students' perceptions of the process. Although we made the survey anonymous and asked the students to answer objectively, they might have tended to select positive options in the survey and express positive feelings in the interview [52]. In addition, the question-order effect [53] (e.g., one question could have provided context for the next one) may lead the respondents to a specific answer. In our case, we ordered the questions based on the natural sequence of actions and the difficulty of answering the questions to help respondents recall and understand the context of the questions.

External validity Although the experiment is conducted with junior-level students in both school years, the number of participants in our experiment is limited (48 students in total), which may influence the generalizability of the results. In addition, all the students are male; further evaluation is required to determine whether the results hold for female students.

Construct validity Referencing the widely used external quality metric, *defect density*, we measured the students' programming quality based on the density of the CQIs that she has introduced, without considering the severity of the CQIs. However, not all quality issues are equally important in a given context [54]. Particularly, in our experiment, the introduced CQIs are concentrated in a few coding rules, which means that a single coding rule tends to be violated many times by a student. Thus, the CQID metric may not accurately measure the students' programming quality regarding different coding rules. Last, our finding regarding RQ1 holds only when they are participating in the continuous inspection process. Although we believe that students can build good programming habits through the continuous training, whether the good programming habits stick without the continuous inspection method must be examined in future studies.

8 Conclusion

In this study, we introduced continuous inspection, an internal-quality assurance method widely adopted on social coding sites, into an educational setting. We designed a specific continuous inspection process for students' collaborative projects and then conducted a two-year controlled experiment to investigate how the process influences the students' programming quality. The quantitative and qualitative results indicate that the process can help students identify their poor coding habits, master a set of best coding practices to improve internal code quality, and significantly reduce their introduced CQIDs. Furthermore, we share our experience and offer recommendations to replicate and improve the process and its execution platform. The work, however, is not finished. We are still on our journey of continuous improvement of instructional design and study. On the one hand, we plan to improve the process and platform of continuous inspection. On the other hand, as mentioned in Section 6, we intend to conduct a controlled experiment on the students in the following semesters to investigate the differences in learning effects between continuous inspection and pair programming.

Acknowledgements We gratefully acknowledge the financial support from National Key R&D Program of China (2018YFB1004202) and the National Natural Science Foundation of China (Grant Nos. 61472430, 61502512,

61532004 and 61379051). We also want to thank our students on their active participation in our study.

References

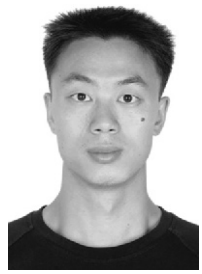
1. Salman I. Students versus professionals as experiment subjects: an investigation on the effectiveness of TDD on code quality. Master's Thesis, University Oulu, 2014
2. Breuker D M, Derriks J, Brunekreef J. Measuring static quality of student code. In: Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education. 2011, 13–17
3. Feldman Y A. Teaching quality object-oriented programming. Technology on Educational Resources in Computing, 2005, 5(1): 1
4. Carver J C, Kraft N A. Evaluating the testing ability of senior-level computer science students. In: Proceedings of IEEE-CS Conference on Software Engineering Education and Training. 2011, 169–178
5. Nawahdah M, Taji D. Investigating students' behavior and code quality when applying pair-programming as a teaching technique in a middle eastern society. In: Proceedings of IEEE Global Engineering Education Conference. 2016, 32–39
6. Radermacher A D. Evaluating the gap between the skills and abilities of senior undergraduate computer science students and the expectations of industry. North Dakota State University, Thesis, 2012
7. Begel A, Simon B. Struggles of new college graduates in their first software development job. ACM SIGCSE Bulletin, 2008, 40(1): 226–230
8. Robins A, Rountree J, Rountree N. Learning and teaching programming: a review and discussion. Computer Science Education, 2003, 13(2): 137–172
9. Higgins C A, Gray G, Symeonidis P, Tsintsifas A. Automated assessment and experiences of teaching programming. Technology on Educational Resources in Computing, 2005, 5(3): 5
10. Piaget J. Psychology and Epistemology: Towards A Theory of Knowledge. Markham: Penguin Books Canada, 1977
11. Chen W K, Tu P Y. Grading code quality of programming assignments based on bad smells. In: Proceedings of the 24th IEEE-CS Conference on Software Engineering Education and Training. 2011, 559
12. Radermacher A, Walia G, Knudson D. Investigating the skill gap between graduating students and industry expectations. In: Proceedings of the 36th International Conference on Software Engineering Companion. 2014, 291–300
13. McConnell S. Code Complete. Pearson Education, 2004
14. ISO. IEC25010: 2011 systems and software engineering–systems and software quality requirements and evaluation (square)–system and software quality models. International Organization for Standardization. 2011, 34–35
15. Gousios G, Pinzger M, Deursen A V. An exploratory study of the pull-based software development model. In: Proceedings of the 36th International Conference on Software Engineering. 2014, 345–355
16. Dabbish L, Stuart C, Tsay J, Herbsleb J. Social coding in GitHub: transparency and collaboration in an open software repository. In: Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work. 2012, 1277–1286
17. Lu Y, Mao X J, Li Z D, Zhang Y, Wang T, Yin G. Does the role matter? an investigation of the code quality of casual contributors in GitHub. In: Proceedings of the 23rd Asia-Pacific Software Engineering Conference. 2016, 49–56
18. Yu Y, Vasilescu B, Wang H M, Filkov V, Devanbu P. Initial and eventual software quality relating to continuous integration in GitHub. 2016, arXiv preprint arXiv:1606.00521
19. Ebbinghaus H. Memory: a contribution to experimental psychology. Annals of Neurosciences, 2013, 20(4): 155
20. Wang H M, Yin G, Li X, Li X. TRUSTIE: A Software Development Platform for Crowdsourcing. Crowdsourcing. Springer Berlin Heidelberg, 2015
21. Wong C P, Xiong Y F, Zhang H Y, Hao D. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: Proceedings of International Conference on Software Maintenance and Evolution. 2014, 181–190
22. Tonella P, Abebe S L. Code quality from the programmer's perspective. In: Proceedings of XII Advanced Computing and Analysis Techniques in Physics Research. 2008
23. Zhang H, Ali B M. Systematic reviews in software engineering: an empirical investigation. Information and Software Technology, 2013, 55(7): 1341–1354
24. Lu Y, Mao X J, Li Z D, Zhang Y, Wang T, Yin G. Internal quality assurance for external contributions in GitHub: an empirical investigation. Journal of Software: Evolution and Process, 2018, 30(4): e1918
25. Akinola O S. An empirical comparative analysis of programming effort, bugs incurrence and code quality between solo and pair programmers. Middle East Technology Scientific Research, 2014, 21(12): 2231–2237
26. Braught G, Wahls T, Eby L M. The case for pair programming in the computer science classroom. ACM Transactions on Computing Education, 2011, 11(1): 2
27. Wang Y Q, Li Y J, Collins M, Liu P J. Process improvement of peer code review and behavior analysis of its participants. In: Proceedings of SigCSE Technical Symposium on Computer Science Education. 2008, 107–111
28. Cunha A D D, Greathead D. Does personality matter?: an analysis of code-review ability. Communications of the ACM, 2007, 50(5): 109–112
29. Hundhausen C, Agrawal A, Fairbrother D, Trevisan M. Integrating pedagogical code reviews into a CS 1 course: an empirical study. ACM SIGCSE Bulletion, 2009, 41(1): 291–295
30. Hüttermann M. DevOps for Developers. Apress, 2012
31. Waller J, Ehmke N C, Hasselbring W. Including performance benchmarks into continuous integration to enable devops. ACM SIGSOFT Software Engineering Notes, 2015, 40(2): 1–4
32. Vasilescu B, Yu Y, Wang H M, Devanbu P, Filkov V. Quality and productivity outcomes relating to continuous integration in GitHub. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering. 2015, 805–816
33. Bowyer J, Hughes J. Assessing undergraduate experience of continuous integration and test-driven development. In: Proceedings of the 28th International Conference on Software Engineering. 2006, 691–694
34. Heckman S, King J. Developing software engineering skills using real tools for automated grading. In: Proceedings of the 49th ACM Techni-

cal Symposium on Computer Science Education. 2018, 794–799

35. Gaudin O, SonarSource. Continuous inspection: a paradigm shift in software quality management. Technical Report, SonarSource S.A., Switzerland, 2013
36. Merson P, Yoder J W, Guerra E M, Aguiar A. Continuous inspection: a pattern for keeping your code healthy and aligned to the architecture. In: Proceedings of the 3rd Asian Conference on Pattern Languages of Programs. 2014
37. Barroca L, Sharp H, Salah D, Taylor K, Gregory Peggy. Bridging the gap between research and agile practice: an evolutionary model. International Journal of System Assurance Engineering and Management, 2018, 9(2): 323–334
38. Krusche S, Berisha M, Bruegge B. Teaching code review management using branch based workflows. In: Proceedings of the 38th International Conference on Software Engineering Companion. 2016, 384–393
39. Jick T D. Mixing qualitative and quantitative methods: triangulation in action. Administrative Science Quarterly, 1979, 24(4): 602–611
40. Letouzey J L. The SQALE method definition document. In: Proceedings of the 3rd International Workshop on Managing Technical Debt. 2012, 31–36
41. Zagalsky A, Feliciano J, Storey M A, Zhao Y Y, Wang W L. The emergence of GitHub as a collaborative platform for education. In: Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work and Social Computing. 2015, 1906–1917
42. Lu Y, Mao X J, Li Z D. Assessing software maintainability based on class diagram design: a preliminary case study. Lecture Notes on Software Engineering, 2016, 4(1): 53–58
43. Chess B, McGraw G. Static analysis for security. IEEE Security & Privacy, 2004, 2(6): 76–79
44. Gousios G, Storey M A, Bacchelli A. Work practices and challenges in pull-based development: the contributor's perspective. In: Proceedings of the 38th International Conference on Software Engineering. 2016, 285–296
45. Mengel S A, Yerramilli V. A case study of the static analysis of the quality of novice student programs. ACM SIGCSE Bulletin, 1999, 31(1): 78–82
46. Kim S H, Pan K, Whitehead E E J. Memories of bug fixes. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2006, 35–45
47. Beck K. Extreme Programming Explained: Embrace Change. Addison-Wesley Professional, 2000
48. Mcdowell C, Werner L, Bullock H, Fernald J. Pair programming improves student retention, confidence, and program quality. Communication of the ACM, 2006, 49(8): 91
49. Mcdowell C, Werner L, Bullock H E, Fernald J. The impact of pair programming on student performance, perception and persistence. In: Proceedings of the 25th International Conference on Software Engineering. 2003, 602–603
50. Mcdowell C, Werner L, Bullock H. The effects of pair-programming on performance in an introductory programming course. ACM SIGCSE Bulletin, 2002, 34(1): 38–42
51. Nagoya F, Liu S Y, Chen Y T. A tool and case study for specification-based program review. In: Proceedings of the 29th Annual International Computer Software and Applications Conference. 2005, 375–380
52. Campbell D T, Stanley J C, Lees Gage N. Experimental and quasi-experimental designs for research. Handbook of Research on Teaching, 1963, 5: 171–246
53. Sigelaman L. Question-order effects on presidential popularity. Public Opinion Quarterly, 1981, 45(2): 199–207
54. Zheng J, Williams L, Nagappan N, Snipes W, Hudepohl J P, Vouk M A. On the value of static analysis for fault detection in software. IEEE Transaction on Software Engineering, 2006, 32(4): 240–253

Yao Lu is a doctoral candidate in the College of Computer, National University of Defense Technology, China. His work interests include open source software engineering, data mining, and crowdsourced learning.

Xinjun Mao is a professor in the College of Computer, National University of Defense Technology, China. He received his PhD degree in computer science from National University of Defense Technology, China in 1998. His research interests include software engineering, multi-agent system, robot system, self-adaptive system, and crowdsourcing.



Tao Wang is an assistant professor in the College of Computer, National University of Defense Technology, China. He received his PhD degree in computer science from National University of Defense Technology, China in 2015. His work interests include open source software engineering, machine learning, data mining, and knowledge discovering in open source software.

Gang Yin is an associate professor in the College of Computer, National University of Defense Technology, China. He received his PhD degree in computer science from National University of Defense Technology, China in 2006. He has published more than 60 research papers in international conferences and journals. His current research interests include distributed computing, information security, and software engineering.

Zude Li is an assistant professor at Central South University. He obtained his PhD degree in 2010 from The University of Western Ontario, Canada. His research interests are in the fields of software architecture, evolution and quality. He is appointed as a software engineering expert in SKANE SOFT.