2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)

2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET) | 978-1-6654-0138-8/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICSE-SEET52601.2021.00011

Assessing the Students' Understanding and their Mistakes in Code Review Checklists

–An Experience Report of 1,791 Code Review Checklist Questions from 394 Students–

Chun Yong Chong
School of IT, Monash University, Malaysia.
chong.chunyong@monash.edu

Patanamon Thongtanunam
The University of Melbourne, Australia.
patanamon.t@unimelb.edu.au

Chakkrit Tantithamthavorn
Monash University, Australia.
chakkrit@monash.edu

***Abstract*—Code review is a widely-used practice in software development companies to identify defects. Hence, code review has been included in many software engineering curricula at universities worldwide. However, teaching code review is still a challenging task because the code review effectiveness depends on the code reading and analytical skills of a reviewer. While several studies have investigated the code reading techniques that students should use to find defects during code review, little has focused on a learning activity that involves analytical skills. Indeed, developing a code review checklist should stimulate students to develop their analytical skills to anticipate potential issues (i.e., software defects). Yet, it is unclear whether students can anticipate potential issues given their limited experience in software development (programming, testing, etc.). We perform a qualitative analysis to investigate whether students are capable of creating code review checklists, and if the checklists can be used to guide reviewers to find defects. In addition, we identify common mistakes that students make when developing a code review checklist. Our results show that while there are some misconceptions among students about the purpose of code review, students are able to anticipate potential defects and create a relatively good code review checklist. Hence, our results lead us to conclude that developing a code review checklist can be a part of the learning activities for code review in order to scaffold students' skills.**

***Index Terms*—Software Engineering Education, Assessment Methods for Software Quality Assurance, Checklist-based Code Review**

I. Introduction

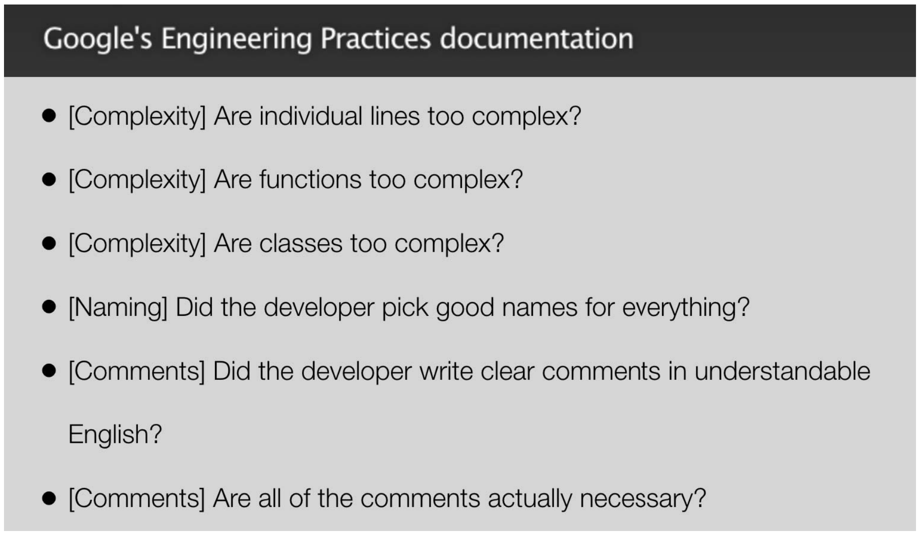
Code review has been widely known to be an effective software quality assurance technique to detect defects through a manual examination of source code written by different developers, other than the code authors themselves [2], [9], [34], [36]. While the goals of automated software testing are much more focused on the functional correctness of software systems, the goals of code reviews include broader defect types such as the reliability of software architecture, the maintainability of source code, and the volatility of code changes [1], [9]. Shull *et al.* argued that code reviews can uncover more than half of software defects which could be extra-deterministic behaviour and such defects cannot be detected by software testing [30]. Several studies found that code reviews have an impact on software quality [33], [35]. Moreover, prior work also showed the quality of other software artifacts such as requirements, design documents, as well as test cases [6], [8], [11]. Code reviews have been widely used in both open source and proprietary organisations (e.g., Google, Microsoft, Facebook, and OpenStack) [25], [28], [29]. In addition, code review has also been used as an assessment method for recruiting software engineers in a software company.[1]

Since code review has become an important and mandatory Software Engineering (SE) practice in the software industry, it is imperative for SE students to be equipped with code review skills prior to employment. However, when teaching code reviews to students, one of the main challenges is modelling students' critical software analysis and reading skills, which are the important skillsets for conducting code reviews. To perform effective code reviews, students should (1) anticipate the common issues or defects in software development, (2) prioritize the importance and severity of the anticipated issues, and (3) analyse software artefacts to identify potential issues (i.e., software defects) [8]. Yet, the common teaching and learning approaches are focused on the last step, i.e., analysing software artefacts by asking students to examine a given piece of source code that is embedded with defects [20], [37]. However, such a code review activity requires a lot of technical knowledge and support to help students to achieve the targeted goal (i.e. maximise the number of defects found during a code review), which can be a cognitive overload for students especially those with less experience in programming [21].

Prior studies show that checklist-based code review is an effective method of learning code reviews for inexperienced students, since the checklist can guide students to focus on the important issues or defects when performing code reviews [6], [27]. Checklist-based code review is considered to be more systematic than *ad-hoc* reviewing (i.e., students identify defects from given software artefacts without checklists and proper instructions) [1]. Typically, a checklist is prepared for a specific type of software artifact (code, UML, UI mockup, etc.), where the checklist may contain multiple questions related to the artifact to be examined. Hence, "a question" or "checklist questions" to refer the item(s) in the checklist. Such a systematic code review method is more suitable for less experienced students [21]. Checklist-based code review is one

[1]https://vampwillow.wordpress.com/2015/03/05/hacking-thoughtworks-recruitment-revisited-graduate-code-reviews/

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 05,2023 at 17:26:01 UTC from IEEE Xplore. Restrictions apply.



Google's Engineering Practices documentation

- [Complexity] Are individual lines too complex?
- [Complexity] Are functions too complex?
- [Complexity] Are classes too complex?
- [Naming] Did the developer pick good names for everything?
- [Comments] Did the developer write clear comments in understandable English?
- [Comments] Are all of the comments actually necessary?

Fig. 1. An example of code review checklists from Google's Engineering Practices documentation

of the structured code review techniques that was formalised by Fagan [1], [9]. A *checklist* is a set of questions related to the potential issues or defects that might occur in code reviews. Figure 1 shows an example of code review checklists from Google's Engineering Practices documentation.[2]

Although checklist-based code review is an effective method to train students' reading skills [27], providing checklists to students may not allow students to develop their analytical skills to anticipate potential issues or defects since they might directly use the provided checklist without understanding and appreciating the key motivations behind checklist-based code review. In addition, the current code review practice in the industry setting generally does not provide code checklists. Instead, reviewers should depend on their own experience and domain knowledge to anticipate issues (i.e., having a checklist in mind) when performing code reviews [28]. Therefore, having students to develop their own checklists should stimulate students to develop the critical and analytical skills to be able to anticipate potential issues or defects in software artefacts that the students will review. Yet, it is unclear whether students can anticipate potential issues (i.e., software defects) given their limited programming experience and what are the mistake that students might have when anticipating issues for code reviews.

In this paper, we aim to investigate whether students can anticipate potential issues before conducting a code review. Hence, we create an assignment to ask students to develop a code review checklist based on a given requirements specification. Then, we examine the student's checklists with respect to two research questions: (1) To what degree can the students checklists be used to identify defects during code review? and (2) What are the common mistakes in students' checklists?. To answer the research questions, we conduct a manual categorization of the questions in the students' checklist into the common defects. We also perform an open coding to identify the common mistakes that the students make when developing a code review checklist. We perform a study with 394 students over the 2018 and 2019 cohorts at Monash University across two campuses (i.e., Australia and Malaysia). The results of this study will help educators improve the teaching strategies to stimulate students in developing their analytical skills for code reviews.

The structure of the paper is as follow. Section II provides a background of the code review practice and the importance of code review. Section III discusses the related work. Section IV describes our research methodology. Section V presents the results. Section VI discusses a broader implication of the results and the limitations of our study. Finally, Section VII draws a conclusion.

II. Background

In this section, we provide a background of the code review practice and software defects that are commonly found during code reviews.

A. Code Review

Code review (or software inspection) is one of the most important practices for software quality assurance which is the core foundation of the Software Engineering discipline. Based on Linus's law *"Given enough eyeballs, all defects are shallow"* [24], the main goal of code review is to manually examine source code to identify software defects, i.e., an error, flaw or fault that causes a software system to behave incorrectly or produce an unexpected result. Several studies have shown that code review provides substantial benefits to improve the quality of a software system [10], [30], [34]. For example, Shull *et al.* [30] argued that code reviews can uncover more than half of software defects which is more than the

[2]https://google.github.io/eng-practices/review/reviewer/looking-for.html

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 05,2023 at 17:26:01 UTC from IEEE Xplore. Restrictions apply.

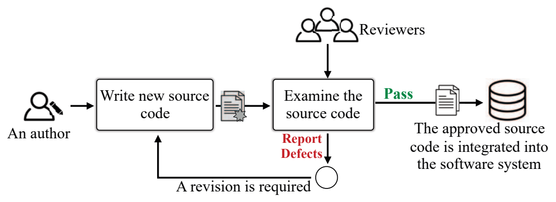


Fig. 2. An overview of the code review process.

number of defects found from software testing. Fagan [10] also found that code review at IBM can uncover 90% of software defects over the lifecycle of a product. A more detail of software defects is described in Section II-B. In addition, prior studies also showed that code review provides additional benefits to software engineering practices, such as improving team cohesion and communication [32], easing the maintainability of the software system [20], [22], and knowledge sharing [35].

Figure 2 shows an overview of the code review process. In general, when an author (i.e., a software developer) writes new pieces of source code for a software system, reviewers (i.e., software developers other than the author) will carefully examine the source code and identify potential defects or problems that new source code might unintentionally introduce to the existing codebase. If defects are indeed found, the author will revise the source code to address the defects or potential problems reported by the reviewers. Once the reviewers agree that the new source code is of sufficiently high quality and free of observable defects, the new source code will be integrated into the software system.

To effectively identify defects, code review requires the analytical and reading skills of reviewers [1], [32]. Adhoc and checklist-based code reviews are the common methods that reviewers use when examining source code. Below is a brief description of the ad-hoc and checklist-based code review techniques.

- **Ad-hoc Code Review** is a lightweight method that is widely used in modern software organisations. The code review of this method is based on a general viewpoint of reviewers who use their personal experience and expertise to identify defects. The strength of this method is that it gives the reviewers freedom to read the code and identify various defects [18]. On the other hand, the weakness in this method is that code review can be ineffective if it is performed by inexperience reviewers.
- **Checklist-based Code Review** is a more systematic method of code review. The method is proposed by Fagan [8], where a reviewer prepares a checklist (i.e., a list of questions or predefined issues or defects that need to be checked) beforehand. Then, the checklist is used to guide the reviewer to identify defects when performing code review. A strength of this method is that a checklist will help reviewers to concentrate their focus on important issues or defects [1]. A weakness of this method is that inexperienced reviewers may overlook the defects that are not directed by the checklist.

B. *Software Defects in Code Reviews*

Prior work attempted to categorise software defects that are commonly found in source code during code reviews [20]. They argued that defect classification can be useful when creating company coding standards and code review checklists. Moreover, recent studies also used the taxonomy proposed by Mantyla *et al.* [20] to evaluate the recent practices of code reviews in open source projects [3], [34]. Table I summarises software defects in code review, which are briefly described as follow. **Documentation defects** refer to the issues on the textual information that eases the understanding of source code (e.g., element naming and comments) or that is used to document the run-time problems (e.g., exception messages). The concern also includes programming language matters related to documentation (e.g., the scope of a variable and method). **Visual representation defects** refer to the concern on the readability of source code. The concern mainly focuses on how should the code elements be organised (e.g., removing unnecessary blank spaces) to ease developers when reading and comprehending the source code. **Structure defects** refer to the concern on the maintainability and evolability of the inspected software. Unlike the documentation and visual representation, structure defect types also includes compilation and run-time issues. **Resource defects** refer to the defect of incorrect manipulation of data and other resources. The concern also includes memory allocation and release for variables. **Check defects** refer to the defect types related to incorrect or missing validation of variables or values. This concern also includes the validation check of user inputs. **Interface defects** refer to the concern on the use of APIs. **Logic defects** refer to the concern on the correctness of logic in the system. **Large defects** refers to defect types on incomplete features and user interfaces.

III. Related Work

A. *Teaching Code Reviews for Software Engineering Education*

In software industry, code review has become a mandatory step in the Software Engineering (SE) workflow of many software organisations, e.g., Google [28], Microsoft [25]. Hence, it is imperative for SE students to be equipped with code review skills prior to their employment. Code review activities have been integrated into the software engineering and computer science syllabus in many universities [16], [32], [38]. For example, Wang *et al.* [38] and Hundhausen *et al.* [16] integrated code review activities into the Computer Science subject called "pedagogical code review" where students assess and provide feedback for a software program of other students. Portugal *et al.* [23] showed that code inspection can be used to validate the functional and non-functional requirements elicited from stakeholders when introduced at the university-level.

From a pedagogy's point-of-view, Hilburn *et al.* [13] suggested that code review is an active learning technique in software engineering education because it is a team activity;

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 05,2023 at 17:26:01 UTC from IEEE Xplore. Restrictions apply.

TABLE I
A TAXONOMY OF DEFECTS IN CODE REVIEW [20]

Type	Description
Documentation Defects	
Naming	Defects on the software element names.
Comment	Defects on a code comment.
Debug Info	Defects on the information that will be used for debugging.
Element Type	Defects on the type of variables.
Immutable	Defect when not declaring variable to be immutable when it should have been.
Visibility	Defects on the visbility of variables.
Other	Other defects relating to textual information.
Visual Representation Defects	
Bracket Usage	Defects on incorrect or missing necessary brackets.
Indentation	Defects on incorrect indentation.
Blank Line Usage	Defects on the use of incorrect or unnecessary blank lines.
Long Line	Defects on a long code statement which cause the readability.
Space Usage	Defects on unnecessary blank spaces.
Grouping	Defects on the cohesion of code elements.
Structure Defects	
Long Sub-routine	Defects on a lengthy function or procedure.
Dead Code	Defects on code statements that do not serve any meaningful purpose.
Duplication	Defects on duplicate code statements.
Semantic Duplication	Defects on code redundancy.
Complex Code	Defects on source code that are difficult to comprehend.
Consistency	Defects on the ways of implementation should be in a similar fashion.
New Functionality	Defects on a piece of source code that does not have a single purpose.
Use Standard Method	Defects whether the standardized approach should be used.
Other	Other defects related to refactoring, organizing, and improving source code.
Resource Defects	
Variable Initialization	Defects on the variables that are uninitialized or incorrectly initialized.
Memory Management	Defects on how program use and manage the memory.
Data & Resource Manipulation	Other defects on the resource management.
Check Defects	
Check Function	Defects on the returned value of a function.
Check Variable	Defects about whether the program validate the variable values.
Check User Input	Defects on the validity of user input.
Interface Defects	
Parameter	Defects on incorrect or missing parameters when calling another function or library.
Function Call	Defects on incorrect interaction between a function and another part of the system or class library.
Logic Defects	
Compare	Defects on logic for a comparison.
Compute	Defects on incorrect logic when the system runs.
Wrong Location	Defects on the wrong location of the operation although it is correct.
Performance	Defects on the efficiency of the algorithm.
Other	Other defects related to the correctness or existence of logic.
Large Defects	
GUI	Defects about the completeness and correctness of user interfaces.
Completeness	Defects about whether the implementation is complete.

it requires technical knowledge to participate; and it involves measurement and use of data to analyse and draw conclusions. Kemerer and Paulk [17] found that when code review activities are included in the learning activities, the quality of student submissions tend to be improved. Hundhausen *et al.* [16] also reported that practicing code review helps students improve the quality of their code and stimulates meaningful discussion and critiques.

1) Challenges in Teaching Code Review: Teaching effective code review is challenging from an educational perspective since code review requires a programming experience, code reading skills, and analytical skills. Prior works mainly focuses on investigating the code reading techniques that help students effectively find defects [21], [27], [32]. For example, McMeekin *et al.* [21] found that ad-hoc code review requires higher cognitive load and it is only suitable for students with a higher education level or have experience in programming. On the other hand, checklist-based code review is suitable for less experienced students as it can be used to guide code reading of the students. Rong *et al.* [27] found that a checklist helps students in terms of a guidance for reading code, i.e. allow students to focus on the important tasks at hand, within a stipulated amount of time. Several code review checklists are developed in order to support code reading during the code

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 05,2023 at 17:26:01 UTC from IEEE Xplore. Restrictions apply.

reviews exercise [7], [15].

In addition to the code reading skills, students should learn how to develop a code review checklist. In other words, providing a checklist to students may hinder the students from learning to analyse and anticipate potential issues or defects that might arise in code review. Moreover, using a generic checklist may lead students to overlook the context-specific problems [21]. Hence, developing a code review checklist should be a learning activity that helps students develop necessary analytical skills to anticipate problems in the context of software development. Furthermore, the industrial practices recommend that reviewers should create and use their own checklists for code reviews [19]. Moreover, it is generally acknowledged that the quality of the checklist will predetermine the quality of the code review process [6]. However, Rong *et al.* [27] argued that developing a high quality checklist is a difficult task for students because students typically have limited domain knowledge (e.g., programming, design, or testing skills). Yet, it is still unclear whether students can develop a code review checklist (i.e., a list of questions that guide reviewers to look for defects) given their limited programming experience. The findings of this work are of importance to help educators to provide early guidance to students if mistakes or misconceptions were found in the preliminary phase of code review process.

IV. Research Methodology

In this section, we describe the goal of this work, the research questions, and the design of our study.

A. Goal & Research Questions

The goal of this work is to investigate whether students can develop a code review checklist (i.e., a list of questions that guide reviewers to look for defects). To achieve this goal, we formulate the following research questions:

- **RQ1: To what degree can the students checklists be used to identify defects during code review?**
  Motivation: This RQ aims to assess whether students can apply analytical skills to create checklists as a guideline to identify software defects during code reviews. The findings will shed some light on the feasibility of this learning activity to educate students to perform effective code reviews.
  Measurements: We examine (1) the number of questions in the students' checklists that can be used to guide reviewers to uncover software defects and (2) the types of software defects that can be identified.
- **RQ2: What are the common mistakes in students' checklists?**
  Motivation: This RQ aims to investigate common mistake among students when developing a code review checklist. The findings could help educators to be aware of and prepare themselves to address the potential mistakes or misunderstanding that students might have when performing code reviews.
  Measurements: We examine (1) the number of questions in the students' checklists that have poor quality and those questions that should not be used to identify software defects and (2) the characteristics of the common mistakes in the students checklists.

B. Designing a Checklist-based Code Review Assignment

To conduct the study, we designed an assignment for developing a code review checklist based on the requirements specification of a given software system. Below, we describe the code review assignment and the teaching and workshop activities that help students to achieve the objective of the code review assignment.

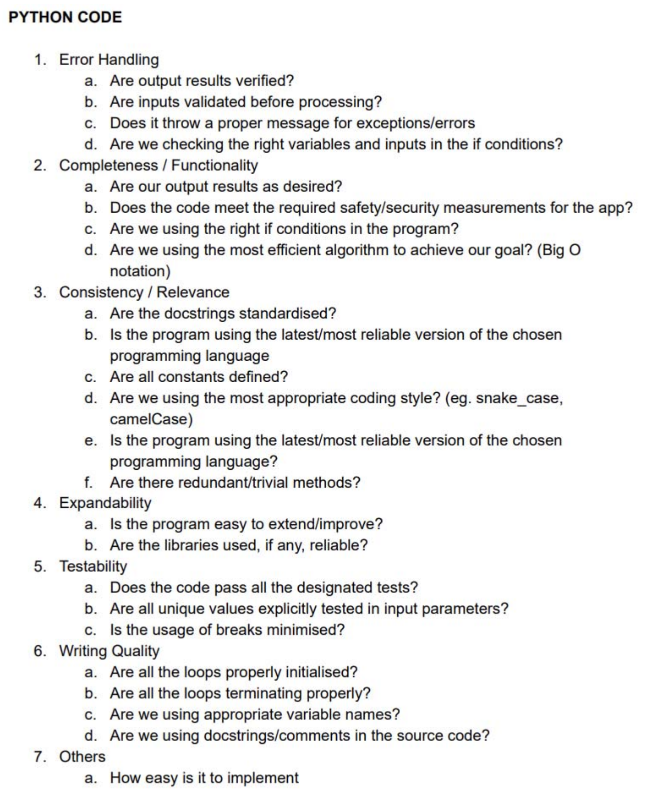
1) The Code Review Assignment: The objective of the code review assignment is to assess students' understanding of checklist-based code review activities (i.e., how do students develop code review checklists). Generally speaking, the students are required to conduct a formal code inspection meeting in a group of 4-5 students in order to mimic formal code review practices on a set of given software artifacts. The structure of the assignment is as follows:

1) Week 2: Students are required to create and submit code review checklists (individually) for a given software artifact. Noted that at this stage, the students only have access to the requirements specification, without the actual artifact that needs to be reviewed. The requirements specification and relevant software artifacts are available online [5]. We do not set a limit on the number of checklist questions that the students can prepare. This will allow the students to design their checklists based on the requirements (i.e. programming languages, expected behaviour of the system, etc.). An example of student's submissions is shown on Figure 3.
2) Week 3: After submitting their checklists, they will be given an actual artifacts to be reviewed. Students need to individually identify potential quality issues (i.e., defects) by applying their own checklists on the provided software artifact. Each student will then prepare a list of issues that they identified, based on their constructed checklist.
3) Week 4: Students are required to conduct a formal inspection with their group members in one of the workshop sessions to discuss and report software defects that they discovered. Students will take turn and discuss the issues that they have discovered, and in return justify if their constructed checklists are useful in identifying potential software defects.

2) Teaching Activities: During the lecture, we highlight that due to the costliness of manual code review, reviewers are required to focus on the most important tasks at hand. Thus, code review checklist plays a very important role to facilitate the review process.

The subject is structured around semi-flipped learning environment, where students are expected to go through pre-class reading notes before attending the weekly lectures. The pre-class reading notes provide detail discussion on the week's topic, with an estimated reading time of 30-45 minutes. The

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 05,2023 at 17:26:01 UTC from IEEE Xplore. Restrictions apply.


PYTHON CODE

1. Error Handling
    a. Are output results verified?
    b. Are inputs validated before processing?
    c. Does it throw a proper message for exceptions/errors
    d. Are we checking the right variables and inputs in the if conditions?
2. Completeness / Functionality
    a. Are our output results as desired?
    b. Does the code meet the required safety/security measurements for the app?
    c. Are we using the right if conditions in the program?
    d. Are we using the most efficient algorithm to achieve our goal? (Big O notation)
3. Consistency / Relevance
    a. Are the docstrings standardised?
    b. Is the program using the latest/most reliable version of the chosen programming language
    c. Are all constants defined?
    d. Are we using the most appropriate coding style? (eg. snake_case, camelCase)
    e. Is the program using the latest/most reliable version of the chosen programming language?
    f. Are there redundant/trivial methods?
4. Expandability
    a. Is the program easy to extend/improve?
    b. Are the libraries used, if any, reliable?
5. Testability
    a. Does the code pass all the designated tests?
    b. Are all unique values explicitly tested in input parameters?
    c. Is the usage of breaks minimised?
6. Writing Quality
    a. Are all the loops properly initialised?
    b. Are all the loops terminating properly?
    c. Are we using appropriate variable names?
    d. Are we using docstrings/comments in the source code?
7. Others
    a. How easy is it to implement

Fig. 3. An example of a code review checklist submitted by students.

weekly lectures (60 minutes per week, 12 weeks in total) will emphasis on the core deliverable of the week and less time are spent on discussing the definition and terminologies since students are expected to go through the reading notes before attending the lectures. Apart from that, there are 2-hour weekly workshops where the students will conduct hands-on exercise on the weekly topics.

In Week 2 of the workshop, students will start working on the Code Review Assignment. They will be doing individual research and preparation to come out with their own sets of checklists that are relevant to the artifact to be reviewed, which in this case is a piece of code written in Python. In Week 4 of the workshop, students will bring along their constructed checklist items, the list of defects that they have discovered using their own checklist, and conduct the face-to-face code review meeting together with their teammates.

C. Data Analysis

We manually analyze the quality of students' checklists. To do so, we first manually classify the questions in the students' checklists based on the types of functional defects shown in Table I that we obtained from Week 2. For the questions that cannot be classified into the functional defects, we perform an open coding to define non-functional defects and common mistakes. We noted that the study protocol has been rigorously reviewed and approved by the Monash University Human Research Ethics Committee (MUHREC Project ID: 21958).

Below, we describe the details of the two phases of our data analysis and the context of the studied cohorts.

1) Data Analysis-1: Manual Classification of Defect Types: A good review checklist should have questions that guide reviewers to look for specific defects during code review activities. Hence, we analyze the type of defects that the questions in the students' checklist aim to identify. To evaluate the students' checklists, we use the taxonomy of functional defects found during code reviews of Mantyla *et al.* [20] (see Table I). The classification is performed by a Research Associate (RA) with solid experience in developing checklists and performing code reviews and teaching our software quality and testing subject in the past 3 years. To ensure the validity of the results, the first author examined the classification results once the first half of the questions of the students' checklists are classified. After the validation, the RA revisited the classified questions to ensure a consistent understanding, then continue classifying the remaining questions in the checklists. Finally, the first author examined all the results. The checklist questions can be classified as undefined if those questions are not relevant to any predefined defects. These undefined questions will be further analyzed in the next step.

2) Data Analysis-2: Open Coding for the Undefined Category: The defect types proposed by Mantyla *et al.* [20] are mainly for functional defects. However, other non-functional defects can be included in the checklists [23]. Hence, we manually analyze the questions in the undefined category of the Data Analysis 1 to see whether they can be used to guide reviewers to look for potential quality issues. We use an open coding approach similar to prior studies [14], [26]. The coding was performed by all the authors of this paper during online meeting sessions. All the coders have experience with code review research and and have strong experience in teaching software engineering subjects. The coders identify the key concerns of each question in the checklist. The key concerns are then noted down when all the coders reach a consensus. In addition to identifying key concerns, the coders also determine whether or not the checklist questions are appropriate, i.e., cannot be used to guide reviewers to identify software defects. If the checklist question is not appropriate, the coder identify the type as a mistake. All the coders manually identify the key concerns and mistakes for each of the checklist questions in the undefined category until reaching a *saturation* state, i.e., no new defect types or mistakes are discovered after coding 50 consecutive checklist questions. Once the list of newly discovered defect types reaches the saturation state, the first author continues to identify the key defect types and the mistakes of the remaining checklist questions. Finally, the second author revisited these questions to validate the results. If a disagreement occurs, all the coders discuss until a consensus is reached.

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 05,2023 at 17:26:01 UTC from IEEE Xplore. Restrictions apply.

D. Studied Cohorts and Context

The participated students are Software Engineering students who enrolled the Software Quality and Testing subject at Monash University in 2018 and 2019. The Software Quality and Testing subject is offered as a second-year core subject at an undergraduate level as part of a 4-year Bachelor degree in Software Engineering. The program's curricula and learning outcomes are aligned with the recommendation from national and international bodies such as ACM and IEEE. The subject is offering for two campuses at different countries (i.e., the main campus is in Australia and the branch campus is in Malaysia). Students of both campuses will undertake the same subject structure, assignments, exams, teaching, and learning materials, ensuring the subject is delivered at the same quality for both campuses. For the main campus, there are 128 enrolled students for 2018 and 159 enrolled students for 2019. For the branch campus, there are 40 enrolled students for 2018 and 67 enrolled students for 2019. Other characteristics of the participated students are described below.

English Language Proficiency: Monash University is an English-speaking institution and all participants have basic competence in English to take part in the study.

Programming Experience: The prerequisite of this subject is either algorithms and programming fundamentals in Python or programming fundamentals in Java, indicating that students have enough programming experience to perform code review.

Code Review Knowledge: Before working on the assignment, the students were given a one-hour lecture of the code review process, including the importance and main goals of conducting code reviews. An example of generic code review checklists [7], [15] is also presented during the lecture hour.

V. Analysis Results

We received a total of 1,791 questions from 394 students across the two studied cohorts in 2018 and 2019. Below, we present the analysis results with regarding to our research questions.

RQ1: To what degree can the students checklists be used to identify defects during code review?

We find that 76% of the checklist questions can be used to identify defects. More specifically, 70% of the checklist questions that are produced by students are clear and can be used to guide reviewers to identify the common defects listed in Table I. In addition, we find that another 6% of the checklist questions aim to identify non-functional defects. A relatively large proportion of appropriate checklist questions suggests that students can anticipate potential problems and produce a good quality of code review checklist.

Figure 4 shows a distribution of the checklist questions across the defect types. Note that the proportion shown in the figure is the proportion of appropriate checklist questions. The result shows that documentation (26%) and structure (22%) defects are the majority defect types that the questions of students checklist aim to identify. The ratio between functional defects (i.e., resource, check, interface, logic, and large defects) and other defects in the student checklists is 35:65, which is similar to the ratio of actual defects found during code review conducted by practitioners (25:75) [3], [20]. This result suggests that student checklists should guide reviewers to uncover defects with a similar ratio of defects as the literature.

TABLE II
THE DISTRIBUTION OF APPROPRIATE QUESTIONS ACROSS DEFECT TYPES.

Defect Type	2018	2019
Documentation	29%	23%
Visual Representation	6%	12%
Structure	24%	20%
Resource	5%	8%
Check	11%	11%
Interface	1%	5%
Logic	15%	13%
Large Defects	1%	0%
Additional	9%	8%

In addition to the common defects listed in Table I, we find that student checklists also include questions that can identify design, requirement, test code, and dependency defects. Based on the recent studies, these defects can be uncovered in the code review process [12], [23], [31]. These additional types of defects account for 8% of the appropriate questions in the students' checklists (see Figure 4). *Design defects* refer to the defects about the consistency between design and code artifacts. *Requirement defects* refer to the defects about the clarity and completeness of code artifacts regarding the requirements specification. *Test code defects* refer to the defects about the quality of the test cases such as test coverage, completeness of the test case documentations (test conditions, test environment, test inputs, expected outcomes, etc.), traceability to requirements, and reusability of test code. *Dependency defects* refers to defects about the usage of external libraries, modules, or APIs as dependencies. This result suggests that in addition to the common defects, students also can anticipate defects in other aspects.

Since we analyze the students' checklists from the 2018 and 2019 cohorts, it is possible that the results may be different between the two cohorts. Table II shows a proportion of appropriate checklist questions in each defect type for each cohort. We perform the Pearson's chi-squared test ($\alpha = 0.05$), i.e., a statistical test for independence of categorical variables, to examine whether the distributions of the checklist questions for the 2018 and 2019 cohorts are statistically different. The test result shows that the distributions of the checklist questions for the two cohorts are not statistically different with p-value of 0.6306. This suggests that the checklists produced by the students in these two cohorts are not statistically different.

RQ2: What are the common mistakes in students' checklists?

From 425 inappropriate checklist questions (26% of all checklist questions), we can identify four common mistakes, i.e, (1) unclear, (2) irrelevant, (3) testing, (4) static analysis. Figure 5 shows that *unclear* is the most common mistake (52%) and the other three mistakes account for a similar

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 05,2023 at 17:26:01 UTC from IEEE Xplore. Restrictions apply.

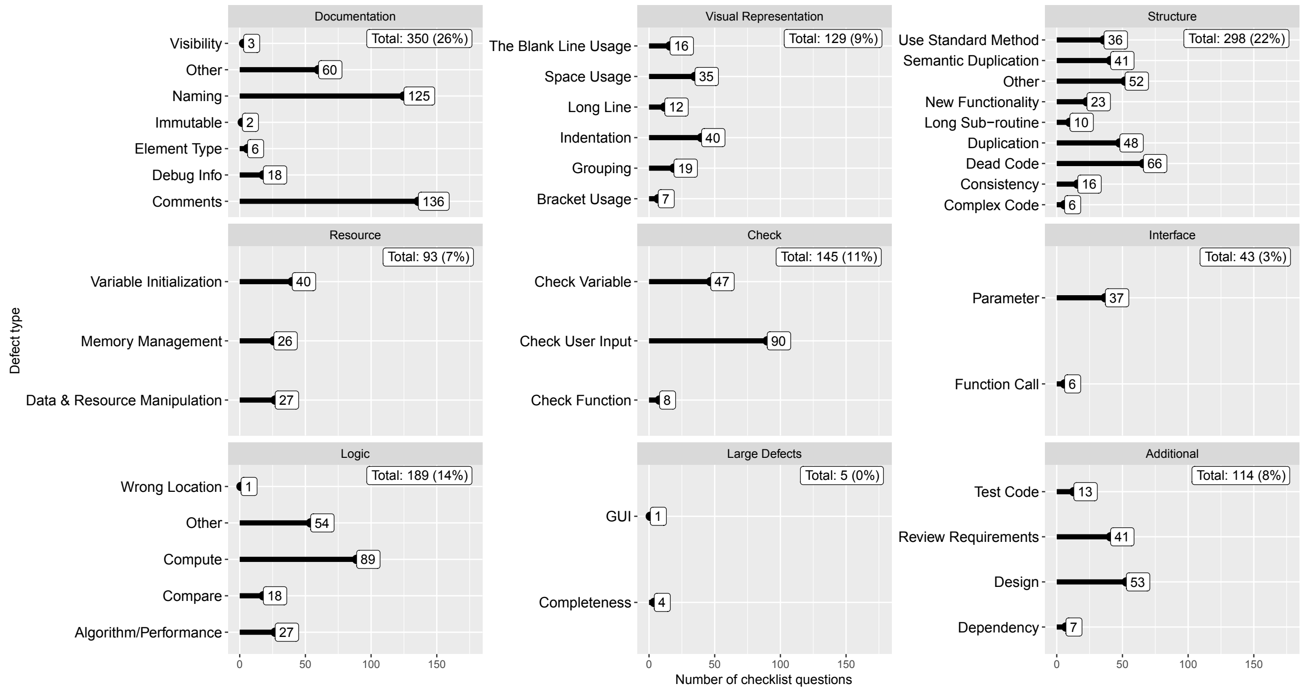


Fig. 4. The number of questions in the checklists with respect to the Mantyla's [20] taxonomy of defect types and the additional categories.

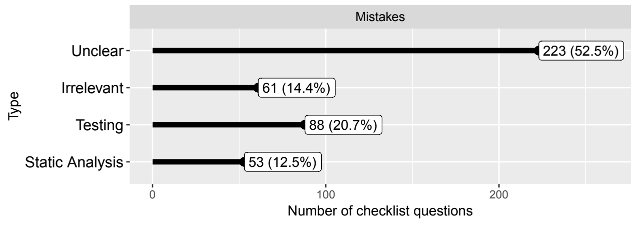


Fig. 5. The number of inappropriate questions in the students' checklists.

proportion (12.5-20.7%). Below, we provide a description, an example and our rationale for each mistake type.

Unclear checklist questions refer to the checklist questions that are too generic or ambiguous to guide reviewers to find a defect. Brykczynski [4] also discussed that a general checklist is inappropriate for code review because it will hinder the reviewer to pay attention on a particular area of the code, which could result in discovering insignificant defects. For example, "*Is the code maintainable?*" is considered as unclear checklist questions. This is because maintainability is a broad aspect and cannot be directly quantified. Hence, such kind of questions are impractical for reviewers to identify defects. To assess the code maintainability, the checklist questions should focus on the defects in the documentation, structure, visual representation aspects listed in TableI.

Irrelevant checklist questions refer to the checklist questions that are irrelevant from the context, e.g., the given requirements specification or programming languages. For example, "*Are all shared variables minimised and handling concurrency of multiple threads and is not leading to a potential deadlock situation?*" is considered irrelevant to the given context. This is because the the use of multiple threads was not described in the requirements specification and the assignment details. We suspect that such irrelevant questions were derived from a generic checklist that is available online.

Testing-dependent checklist questions refer to the checklist questions that are dependent to the testing results. For example,"*Does the code pass unit, integration, and system tests?*" is considered as an inappropriate question. This is because code review should focus on defects based on the source code. Code review should be a complimentary software assurance activity that can discover defects other than the defects detected by testing. Hence, code review should not be dependent to the testing results.

Static Analysis-dependent checklist questions refer to the checklist questions that are dependent to the results of the static analysis tool that check for code styling or syntax, for example, "*Does Pylint give a reasonable score (at least 7/10)?*". This kind of checklist questions are considered as inappropriate because code review should focus on more in-depth defects. Moreover, such errors should be removed by the developers prior to the code review process to minimize the code review effort.

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 05,2023 at 17:26:01 UTC from IEEE Xplore. Restrictions apply.

VI. Lessons Learned

Our analysis results show that given a basic knowledge of algorithms and programming, 76% of the checklist questions are appropriate which can be used to guide reviewers to find defects. The results suggest that students tend to be able to anticipate potential defects and create a good code review checklist. Nevertheless, students might make some mistakes when developing a code review checklist. The first two common mistakes (i.e., unclear and irrelevant questions) are in part due to the misconception about the quality specification of the code review checklist. Thus, we recommend educators cope this issue in the future by making a clear specification of the checklist. The other two common mistakes (i.e., testing-dependent and static-analysis-dependent) are in part due to the misconception about the goal of code review process. Thus, we recommend educators emphasize the key role of code review in software engineering process and a clear distinction of software quality assurance activities.

In addition, developing a code review checklist should be included as one of the learning activities for code review. This can be considered as a scaffolding activity. Students can demonstrate their understanding about code reviews through the development of a code checklist. Then, educators can give a guidance or correct the misconception that students might have before reading code and identifying defects during a code review. For example, we observed that the checklist questions are more focused on documentation, while the defects found in the industrial reviews are structure and large defects [20]. Then, before conducting a code review, educators should support students by emphasizing more on the structural defects in order to stay in line with the industrial practices.

Limitation. Our findings shed the light on the possibility of including code review checklist as an additional learning activity for code review assignment. However, there are some limitations with the research methodology and the consideration to apply the research results in general. The first limitation is the diversity of participants. Although the data is collected from a large group of participants (394 students) from two years of cohorts (2018 and 2019) and two campuses, all the participants are from one university. The conclusion of this study may not be generalized to the students of other universities. Nevertheless, our prerequisite is common, i.e., students have passed algorithms and programming fundamentals. Hence, it is feasible to revisit this work in a different university. The second limitation is the programming background of students. The participants are second-year undergraduate students with an algorithm and programming background. Hence, the conclusion may only be generalized to undergraduate students with limited programming background. Students with more software engineering experience and stronger programming background may generate a different code review checklist.

VII. Conclusion

In this work, we investigate whether students can anticipate potential issues when performing code review. To do so, we examine the code review checklists (a list of questions that guide reviewers to look for defects) that are developed by students. From the 1,791 checklist questions of 394 students across the two cohorts at Monash University, we find that 76% of the questions are appropriate to be used to guide reviewers during code review. More specifically, 70% of the checklist questions can be used to guide reviewers to identify defects that are commonly found during the code review. We also identified four common mistakes of the checklist questions that educators should be aware of when asking students to develop a code review checklist. Our results shed the light on an additional learning activity of code reviews, i.e., developing a code review checklist, which should stimulate students in developing their analytical skills for code reviews.

Acknowledgement

C. Tantithamthavorn was partially supported by the Australian Research Council's Discovery Early Career Researcher Award (DECRA) funding scheme (DE200100941). P. Thongtanunam was partially supported by the Australian Research Council's Discovery Early Career Researcher Award (DECRA) funding scheme (DE210101091). The authors would like to thank Najam Nazar and Jia Chee Pang for their inputs and comments.

References

[1] A. Aurum, H. Petersson, and C. Wohlin, "State-of-the-art: software inspections after 25 years," *Software Testing, Verification and Reliability*, vol. 12, no. 3, pp. 133–154, 2002.

[2] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 2013 international conference on software engineering*. IEEE Press, 2013, pp. 712–721.

[3] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern Code Reviews in Open-Source Projects: Which Problems Do They Fix?" in *Proceedings of the 11th International Working Conference on Mining Software Repositories (MSR)*, 2014, pp. 202–211.

[4] B. Brykczynski, "A survey of software inspection checklists," *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 1, p. 82, 1999.

[5] C. Y. Chong, P. Thongtanunam, and C. Tantithamthavorn, "Dataset - Assessing the Students' Understanding and their Mistakes in Code Review Checklists -An Experience Report of 1,791 Code Review Checklists Questions from 394 Students," Jan. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4431059

[6] A. Dunsmore, M. Roper, and M. Wood, "The development and evaluation of three diverse techniques for object-oriented code inspection," *IEEE transactions on software engineering*, vol. 29, no. 8, pp. 677–686, 2003.

[7] ——, "Practical code inspection techniques for object-oriented systems: an experimental comparison," *IEEE software*, vol. 20, no. 4, pp. 21–29, 2003.

[8] M. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976.

[9] ——, "Design and code inspections to reduce errors in program development," in *Software pioneers*. Springer, 2002, pp. 575–607.

[10] M. E. Fagan, "Advances in software inspections," in *Pioneers and Their Contributions to Software Engineering*. Springer, 2001, pp. 335–360.

[11] T. Gilb, D. Graham, and S. Finzi, *Software inspection*. Addison-Wesley Longman Publishing Co., Inc., 1993.

[12] L. He, J. C. Carver, and R. Vaughn, "Using inspections to teach requirements validation," *CrossTalk: The Journal of Defense Software Engineering*, vol. 21, no. 1, 2008.

[13] T. B. Hilburn, M. Towhidnejad, and S. Salamah, "Read before you write," in *2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)*. IEEE, 2011, pp. 371–380.

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 05,2023 at 17:26:01 UTC from IEEE Xplore. Restrictions apply.

[14] T. Hirao, S. McIntosh, A. Ihara, and K. Matsumoto, "The review linkage graph for code review analytics: a recovery approach and empirical study," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 578–589.

[15] W. S. Humphrey, *A Discipline for Software Engineering*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[16] C. Hundhausen, A. Agrawal, D. Fairbrother, and M. Trevisan, "Integrating pedagogical code reviews into a cs 1 course: an empirical study," in *ACM SIGCSE Bulletin*, vol. 41, no. 1. ACM, 2009, pp. 291–295.

[17] C. F. Kemerer and M. C. Paulk, "The impact of design and code reviews on software quality: An empirical study based on psp data," *IEEE transactions on software engineering*, vol. 35, no. 4, pp. 534–550, 2009.

[18] O. Laitenberger and J.-M. DeBaud, "An encompassing life cycle centric survey of software inspection," *Journal of systems and software*, vol. 50, no. 1, pp. 5–31, 2000.

[19] L. MacLeod, M. Greiler, M.-A. Storey, C. Bird, and J. Czerwonka, "Code Reviewing in the Trenches," *IEEE Software*, vol. 35, pp. 34–42, 2018.

[20] M. V. Mäntylä and C. Lassenius, "What types of defects are really discovered in code reviews?" *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 430–448, 2008.

[21] D. A. McMeekin, B. R. von Konsky, E. Chang, and D. J. Cooper, "Evaluating software inspection cognition levels using bloom's taxonomy," in *2009 22nd Conference on Software Engineering Education and Training*. IEEE, 2009, pp. 232–239.

[22] R. Morales, S. McIntosh, and F. Khomh, "Do Code Review Practices Impact Design Quality? A Case Study of the Qt, VTK, and ITK Projects," in *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SAnER)*, 2015, pp. 171–180.

[23] R. L. Q. Portugal, P. Engiel, J. Pivatelli, and J. C. S. do Prado Leite, "Facing the challenges of teaching requirements engineering," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2016, pp. 461–470.

[24] E. Raymond, "The cathedral and the bazaar," *Knowledge, Technology & Policy*, vol. 12, no. 3, pp. 23–49, 1999.

[25] P. C. Rigby and C. Bird, "Convergent Contemporary Software Peer Review Practices," in *Proceedings of the 9th joint meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2013, pp. 202–212.

[26] P. C. Rigby and M.-A. Storey, "Understanding Broadcast Based Peer Review on Open Source Software Projects," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 541–550.

[27] G. Rong, J. Li, M. Xie, and T. Zheng, "The effect of checklist in code review for inexperienced students: An empirical study," in *2012 IEEE 25th Conference on Software Engineering Education and Training*. IEEE, 2012, pp. 120–124.

[28] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: a case study at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 2018, pp. 181–190.

[29] J. Shimagaki, Y. Kamei, S. McIntosh, A. E. Hassan, and N. Ubayashi, "A Study of the Quality-Impacting Practices of Modern Code Review at Sony Mobile," in *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE)*, 2016, pp. 212–221.

[30] F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, "What We Have Learned About Fighting Defects," in *Proceedings of the 8th International Software Metrics Symposium (METRICS)*, 2002, pp. 249–258.

[31] D. Spadini, M. Aniche, M.-A. Storey, M. Bruntink, and A. Bacchelli, "When testing meets code review: Why and how developers review tests," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 677–687.

[32] S. Sripada, Y. R. Reddy, and A. Sureka, "In support of peer code review and inspection in an undergraduate software engineering course," in *2015 IEEE 28th Conference on Software Engineering Education and Training*. IEEE, 2015, pp. 3–6.

[33] P. Thongtanunam and A. E. Hassan, "Review dynamics and their impact on software quality," in *IEEE Transaction on Software Engineering (TSE)*, 2020, p. to appear.

[34] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Investigating code review practices in defective files: An empirical study of the qt system," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, 2015, pp. 168–179.

[35] ——, "Revisiting code ownership and its relationship with software quality in the scope of modern code review," in *Proceedings of the 38th international conference on software engineering*. ACM, 2016, pp. 1039–1050.

[36] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto, "Who Should Review My Code? a file location-based code-reviewer recommendation approach for modern code review," in *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 141–150.

[37] Y. Wang, H. Li, Y. Feng, Y. Jiang, and Y. Liu, "Assessment of programming language learning based on peer code review model: Implementation and experience report," *Computers & Education*, vol. 59, no. 2, pp. 412–422, 2012.

[38] Y. Wang, L. Yijun, M. Collins, and P. Liu, "Process improvement of peer code review and behavior analysis of its participants," in *ACM SIGCSE Bulletin*, vol. 40, no. 1. ACM, 2008, pp. 107–111.

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 05,2023 at 17:26:01 UTC from IEEE Xplore. Restrictions apply.