# Can students help themselves? An investigation of students' feedback on the quality of the source code

Raul Andrade*, João Brunet†
*†*Department of Systems and Computation*, *Federal University of Campina Grande*
Campina Grande, Brazil
* joseraul@copin.ufcg.edu.br, †joao.arthur@computacao.ufcg.edu.br

***Abstract*—This Research to Practice Full Paper presents a study on the evaluation of qualitative aspects of students' programs in an introductory programming course. Approaches have been proposed in order to address the quality of the source code, but they typically focus on automated analysis of syntactic aspects which might lead to generic feedback. In this study, we investigate if, by including students as evaluators, we could provide personalized feedback on the quality of source code. To do so, we applied a survey with assignments and their respective source codes answered by students in previous terms. Teachers and students analyze those source codes and gave suggestions to improve them qualitatively and, we found that most students identified code quality aspects with a similarity equal to or greater than 50% in comparison to teachers' and that similarity increases as students progress in the course. We found that students are particularly good at finding and giving feedback on complexity issues. This study may lead to further investigations on addressing source code quality on collaborative learning, and may also support the development of lint-like tools, once it yields detailed information on how students provide feedback regarding source code quality.**

***Index Terms*—learning programming; code quality; personalized feedback; crowdsourcing**

## I. Introduction

The introductory programming subjects typically involve a large number of tasks, which makes it hard to individually follow student's evolution [1]. In the Computer Science course (CS) from Federal University of Campina Grande (UFCG), for example, this subject has around 100 students enrolled by semester, and they usually submit hundreds of solutions per day. For this reason, it is laborious to provide manual feedback for each student submission over time.

To tackle this problem, researchers have been developing approaches to provide automatic feedback to students' source code [2] [3]. In general, the proposals implement the Online Judges idea [4], in which they automatically verify the submitted solution against teachers' pre-defined tests. These studies focus on functional feedback, i.e., if the program is correct according to the tests (if it is doing what it's supposed to do). However, there is also the need to analyze the code quality produced by the students.

Source code quality involve a number of aspects related to readability, testability and maintainability of the code under analysis and it is deeply investigated on large programs. However, these concepts are important even for small programs, such as the ones developed by students in introductory programming subjects. In particular, in the context of this work, we address solution's complexity, size of the source code, variables' nomenclature, among others [5] (illustrated by Figure 2). Although there are research efforts in this regard, some proposed tools, such as Qcheck [6], automate qualitative analysis based on syntactic aspects, like the defined criteria of the python community in pep 8 [7]. The pep 8 is a style guide for Python code comprising the standard library in the main Python distribution. However, the provided feedback is sometimes generic. So, in this work, we investigate students can provide personalized feedback on the quality of the code. Our main question is: **Can students evaluate (formative assessment [8]) qualitatively their pairs' review?**

In this study, we applied a survey with tasks and their respective source codes, developed by students from previous semesters. We analyzed 375 submissions functionally correct, referring to 5 different programming assignments. 75 students and 4 teachers from the introductory programming subject participated, at CS/UFCG. Teachers and students from the introductory programming from CS/UFCG analyzed these submissions and gave suggestions to improve them qualitatively. By calculating Jaccard's similarity coefficient between students and teachers, we found out that the majority of students identified code quality aspects with equal or superior to 50% similarity and this similarity increases as students progress in the subject. We also found out that students are particularly good at finding and giving feedback about complexity issues, like code duplication and refactor code.

Based on ours analysis, the students can elaborate, on a significant level, useful feedback about their pairs' code quality. This study can support the development of code quality evaluation tools. So that, by including the student as an evaluator, it can in a collaborative way, increase the workforce on qualitative evaluation and provide a more detailed feedback. Besides, this study makes the following contributions:

- We verified that a group of students can provide useful feedback about their pairs' code quality; and
- We verified which aspects of code quality the students identify more.

This article is organized as follows: Section II presents the challenges of teaching-learning and evaluating in introductory programming. Section III mentions some related works. Section IV shows an overview of the class in which the study

978-1-5386-1174-6/18/$31.00 ©2018 IEEE

was accomplished, and the problem approached. Section V describes the study design. The study's results, including a debate, are presented in Section VI. Validation threats are considered and discussed in Section VII. Finally, the study's conclusions and instructions for future works are approached in Section VIII.

## II. Background

In CS courses, the subjects related to programming have a fundamental role in the students' academic development. On the other hand, the learning difficulty on these subjects is obviously clear [9]. According to Wilcox [1], among the challenges that influence on learning lies the teacher's struggle to individually follows the student's development. That happens due to the classes, especially the introductory subjects, having a high number of enrolled students. Therefore, it is almost unfeasible for the teacher to effectively provide detailed manual feedback for each student enrolled on the laboratory activities.

According to Schunk and Petermann [10], the feedback provides reflection about the expected and achieved results, being considered an important ally on the process of knowledge construction. In this context, feedback plays an important role in the learning process. Current research works in this field have been proposing tools to provide such feedback in an automated manner. Most tools that provide feedback, implement the Online Judges idea [4], in other words, the codes submitted by the students are automatically corrected using the teachers' pre-defined tests [2] [3]. The focus of the automatized feedback camp and tasks evaluation is the functional correction of the programs, although some tools also incorporate feedback on quality aspects [5].

Typically, research work on providing automated feedback on the quality of the source code focus on tools, like Qcheck [6], which automatize the quantitative analysis based on coding patterns established by the Python community on pep 8 [7]. Also, some IDE's detect quality code issues and other refactoring opportunities, but these can be too complex for inexperienced developers, and they were not developed to support learning activities [5].

According to Hattie and Timperley [11], an effective feedback is personal and task specific. In this context, Glassman et al. [12] proposed a collaborative approach, based on the Crowdsourcing concept, in which students themselves elaborate personalized hints for the resolution and enhancement of tasks solution. The study's goal was to enable a personalized mentorship for classes with a large number of students. The idea came from the principle that students have the capacity to participate on their colleagues' mentoring process on the basis of their learned experiencing on previous problems resolutions.

Collaborative approaches in feedback with the intent to formatively evaluate (self evaluation or in pairs) is a process that has interaction as a fundamental mechanism to knowledge construction. In this process, students share informations, make decisions, discuss, between other actions. This practice has showed itself efficient, especially in online courses with many students [13]. According to Shang et al. [13], relating experience with interaction can aid students in meaning attribution, due to the reflexion of distinctive points of view. The proposals that the students' activities (tasks) be distributed to a set of reviewers (other students) that have to evaluate and provide formative feedback about the activity, can help students become reflective practitioners of their occupation and autonomous on their own learning. In the education, not only it can minimize the scale problem in obtaining feedback on quality code issues, but also benefits the improvement of the student's learning, because he can become more critical and autonomous.

## III. Related work

The focus of the feedback field in the programming assignment has been the functional correction of programs, although some tools also include feedback about quality aspects [5]. Joy et al. [14], for example, offers BOSS, a system to evaluate works. It performs automatized tests for correction and source code quality, verifies plagiarism, and provides feedback. Typically, the tools that provide automatic feedback are based on software engineering metrics, like cyclomatic complexity and LOC, or lint tools integrated [6]. However, in some cases, the feedback can be generic and not very effective for beginning students. In this regard, other studies were developed including students as reviewers.

Hundhausen et al. [15] developed an adaptation of studio-based instruction for computing education called the pedagogical code review (PCR). The goal was to explore instructional methods to refine solutions in an interactive way, using critical review between students and instructors. They developed an online environment that allows the PCRs to occur in an asynchronous way outside the classroom and compared the results between a CS1 course with PCRs online and a CS1 course with PCRs face to face. The study identified that in the course with PCRs face to face the self-efficiency, review quality and peer learning were significantly higher and students were more positive towards PCR.

Wang et al. [16] presents the EduPCRe, an online evaluation system based on the code inspection process used in the software industry of reviewing of pairs. In this approach, students review other students' code and teachers assess and assign scores to students based on their performance, realized review and their continuance on the peer review process. The authors noticed significant improvements in the students' learning in many aspects.

Using the concept of personalized hints, Glassamn et al. [12] propose a collaborative approach, where students from a class of Software engineering elaborate hints themselves to resolution and optimization of solutions. The goal was to enable a personalized mentorship to large classes. The idea comes from the principle that students themselves possess the ability of participating in their colleagues' tutoring process based on their acquired experience throughout previous problems resolutions.

The works mentioned reinforce that including the student as an evaluator can have a positive impact in their performance and learning in the subject. Our proposal's singularity is observing the context of the introductory programming subject from CS/UFCG and approach exclusively quality code issues, through personalized hints elaborated by the students, without the teachers' supervision.

## IV. Overview of introductory programming subject from CS/UFCG

The teachers of UFCG Computer Science introductory programming subject approach practical and theoretical aspects in an intercalated way. They usually present theoretical aspects in the classroom and provide a number of practical assignments to be developed in Python programming language during the laboratory sessions.

The subject is organized into ten sequential units. Each one of them approaches a different topic, as described by Table 1. Each student advances in his own pace, meaning that the same Introductory programming class can have students in different units.

TABLE I
Units of the subject.

| Unit | Description | Unit | Description |
|---|---|---|---|
| 1 | Elementary computer programming concepts. | 6 | Functions. |
| 2 | Writing simple programs. | 7 | Arrays. |
| 3 | Conditions, alternatives, and functions. | 8 | Lists. |
| 4 | Defined loops. | 9 | Sequences of sequences and matrices. |
| 5 | Indefinite loops. | 10 | Maps. |

The adopted teaching methodology is based on three key aspects: (i) flipped classroom, in which students learn in their own environments, watching videos available by their teachers, reading detailed lessons, performing assignments, among other activities. In this particular methodology, the teacher plays a mediating role, and the classroom time is dedicated to discuss students questions; (ii) continued evaluation: students are submitted to a minitest every week in order to verify their performance in the current unit; and (iii) mastery learning [17], where to advance from one unit to another, students need to demonstrate the abilities and competencies required to master the unit. In the UFCG subject, they have to answer correctly two assignments per unit in order to demonstrate proficiency.

## V. Study design

In this study, we investigated students' ability to elaborate hints related to code quality, more precisely if their hints show similar aspects to the teachers' from the subject. We applied supervised survey as research method to verify if students can provide useful feedback to improve their pairs' code quality. If so, what quality issues are identified and how close they are to the ones identified by the teachers. To drive our study, we formulated the following research questions:

- **RQ1**: Can the students of introductory programming from CS/UFCG identify code quality issues?
- **RQ2**: What are the most frequent code quality issues identified by the students of introductory programming from CS/UFCG?
- **RQ3**: Can the students of introductory programming from CS/UFCG give useful feedback to improve the code quality of their pairs'?

The subject of introductory programming from CS/UFCG is divided in 4 different classes. The classes are composed by students in close units. In this way, even with different professors, the contents are approached in the same way for all students. To mitigate the threats, the experiment happened simultaneously in every class with the same duration time. Besides, we conducted a previous training with the volunteer researchers that applied the experiment so that the explanation was similar for all classes.

### A. Participants

The subjects of this study are the following: 4 teachers and 75 from the 114 enrolled students in the introductory programming subject from CS/UFCG during 2017's first semester. We conducted the experiment during class time to increase the attendance. However, in order to verify if the number of participants represents the population we're studying, we calculated the sample size according to Equation (1), where n= calculated sample, N= population, Z= standard normal variable associated to confidence level, p= event's real probability, and e= sampling error.

$$n = \frac{N \cdot Z^2 \cdot p \cdot (1-p)}{Z^2 \cdot p \cdot (1-p) + e^2 \cdot (N-1)} \qquad (1)$$

We verified that our sample represents study population with sampling error (difference between estimated number and real number) close to 7% and with 95% of confidence (probability that the effective sampling error is lower than the admitted sampling error).

As mentioned in Section III, due to the chosen methodology, there are students in different study units in the same subject. Unit 4 presents, historically, more student retention. In this study, it represented a little more than 49% of the participants. In the students' group, we did not identify any student from unit 9.

### B. Survey

We applied a survey with programming assignments and their respective source codes developed by students from previous semesters. All the solutions are functionally correct, but they present problems regarding quality. In this study, we focus on the following: (i) complexity (code duplication, dispensable code and refactor code); (II) spacing (row spacing, character spacing and indentation); (iii) variables (type, absence, excess and identifier) and; (iv) header (absence, incomplete and excessive data). In Figure 1, we present an example of a source code that was used in this experiment.

```
1   # coding: utf-8
2   # xxxx.xxxxxxxx / xxxx / 2014.2
3   # Collatz life

4   number = int(raw_input())
5   cont = 0

6   while True:
7     if number == 1:
8       cont += 1
9       break

10    if number % 2 == 0:
11      number = number/2.0
12      cont += 1
13    else:
14      number = 3 * number + 1
15      cont += 1
16  print cont
```

Fig. 1. Correct code but with quality issues.

The subjects of this study analyzed the source codes and gave hints to improve them qualitatively. We selected assignments and their respective solutions considering the presence of the main code quality aspects addressed in this study (detailed in Table II).

TABLE II
GENERAL ASPECTS OF CODE QUALITY EVALUATED IN CS/UFCG.

| Aspect | What is analyzed? |
|---|---|
| Header | Checks if the solution has a header, if it is in compliance with the question and if there aren't absent information. |
| Complexity | Verifies if the solution is more complex than it has to be, and if it is possible to simplify it. |
| Variable | Verifies if there is lack or excess of variables in the code and aspects related to the nomenclature. |
| Spacing | Verifies the lack or excess of space between lines and characters, and problems related to indentation. |

### C. Metrics

To compare and analyze the hints, we coded the interviews using a technique of qualitative analysis of interview data [18]. First, we transcript reading. Second, we labeling important snippets. Then, we defining what codes are most important. Afterwards we categorize and define which are more relevant and how they are connected. In order to better illustrate this process, let us analyze the following example: a provided hint was: *"You can remove the duplicate code and with that, avoid unnecessary code repetition. Check the 'prints' at the end of the program."*. For this case, we defined the following tags: **complexity**, **code duplication** and **code refactor**. The tags allow us to compare and analyze the hints with higher precision (presumably subjective). Each hint can be composed by one tag or a group of them. The data was coded manually by one of the researchers. Figure 2 shows the tags we used in this study. For every hint, we considered at least one of the 4 main aspects evaluated in the subject (Table II).

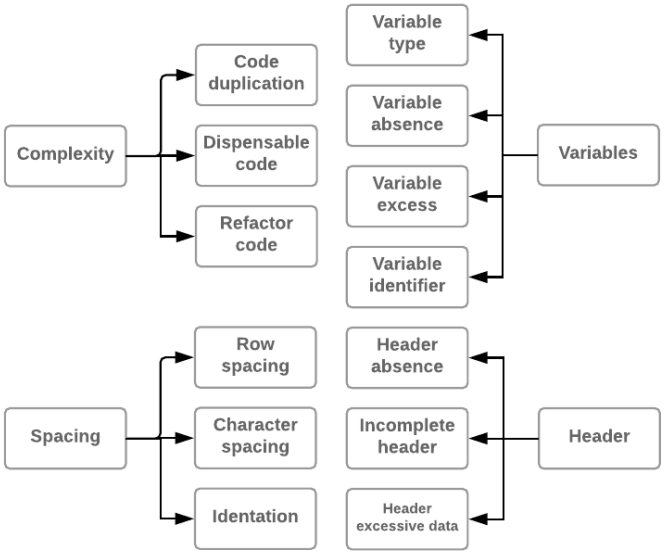


Fig. 2. Code quality issues tags found in this study.

In a first step, we measured the similarity between teachers according to the hints they elaborated. We performed that to verify if they can provide similar feedback about code quality issues and, as a consequence, be treated as a group. According to Jaccard's similarity coefficient (2), the teachers showed a correlation index between 0.50 and 0.86. The similarity index in this model varies between 0 and 1, being that the closer to 1, the bigger the similarity between the groups. We then analyzed the teachers' hints to create a reference feedback about quality code problems. This reference feedback includes hints based on their frequency. We selected, for each assignment, hints given by at least 2 from the 4 teachers.

$$S(Sf, Tf) = \frac{|Sf \cap Tf|}{|Sf| + |Tf| - |Sf \cap Tf|} \qquad (2)$$

We identified the similarity between the students and teachers hints using Jaccard's similarity coefficient. It compares the number of similar elements between two groups and the total number of involved elements, excluding the number of conjoined absences. In this study, we calculated the similarity between each student's hints (Sf) and the hints of the feedback form (Tf). In literature, there is no ideal index to be reached with the Jaccard coefficient. In this study, we considered significant, even if not ideal, the similarity index starting at 0.5 (50% similar). The 50% similarity is not ideal, but it is a significant value considering that students are still learning how to give feedback. We hope that the similarity will increase progressively in the next experiments.

## VI. ANALYSIS AND DISCUSSION

We calculated the students' hints similarity from the average of the Jaccard coefficient values, which he achieved in each task when compared to the reference feedback. Figure 3 presents the general similarity rate of students accordingly to the unit they were at the time of the experiment.

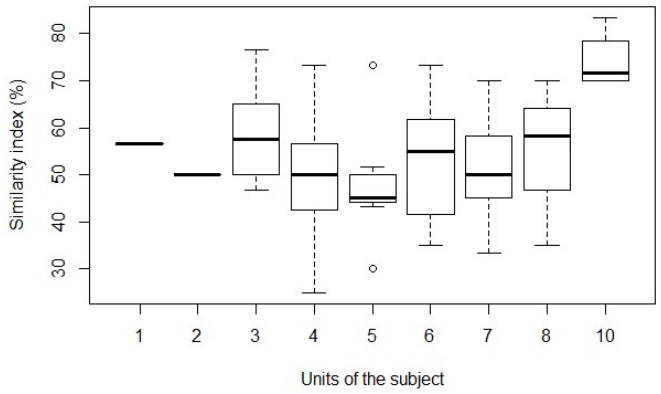


Fig. 3. Similarity between code quality aspects identified by students and teachers.

**RQ1:** Can the students' of introductory programming from CS/UFCG identify code quality issues?

We analyzed the similarity between the students' hints and the teachers' per unit, with the purpose of verifying if there is any relation with the students' advance in the subject. We noticed that there is variation between the students' similarity in all units. According to Figure 3, the points that represent the median of students per unit, show that more than half of them elaborated hints with similarity equal or superior to 50% in relation to the hints elaborated by the subject's teachers. The 50% similarity even though not ideal, is a significant value. In a way that, if more than one student elaborates hints to a determined solution, the feedback can be effective, or more effective than the hints of only one student.

In the students' group, we did not identify any student from unit 9. The highest similarity index reached was approximately 75% from students in the unit 10 (which makes sense, because the more advanced, probably the more knowledge the student has about these concepts) and some from units 3 and 6 got closer to this number. Unit 5 was the only one that reached an inferior median number, but still, a student from this unit reached similarity close to 70%. We verified that the last units showed better results. In unit 10 everyone presented similarity higher or equal to 70%, but it is also important to note that this unit has fewer students' than the others.

Unit 10 was also the one were occurred the least variation in the students' similarity. So, we believe that the feedback that is more similar with the teachers' is from students that have already concluded the units, not from those who are still concluding. The students' ability level was the factor that affected the similarity the most. In this way, we concluded that the students as a group is capable of identifying a significant amount, but not ideal, of issues related to quality code similar those identified by the teachers. However, a group of students, especially the more experienced, can identify these problems with bigger precision.

**RQ2:** What are the most frequent code quality issues identified by the students of introductory programming from CS/UFCG?

First, we considered the main code quality issues analyzed in the subject. We verified that the largest part of the hints approaches complexity problems, as shown in Table III. The second most present type includes hints related to the solutions' headers. This issue considers the lack of information and clarity in its developing. Among the factors that may have influenced in this result, we highlighted the fact that the header is, among the analyzed problems, the simplest one to be recognized and, because we hide the information from the solutions' authors (previous semester students), some students may have mentioned it as an issue, even if it was clarified before this experiment happened. There is no significant difference between the amount of hints about header, spacing, and variables.

TABLE III
AMOUNT OF HINTS BY REPORTED CODE QUALITY ISSUES.

| Code quality issues | How many times has been identified |
|---|---|
| Complexity | 203 |
| Header | 84 |
| Spacing | 77 |
| Variable | 62 |

The students also identified other more specific code quality issues, besides the issues already mentioned. To better understand this scenario, we decided to rank the quality code issues identified in the hints, as shown in Figure 4.

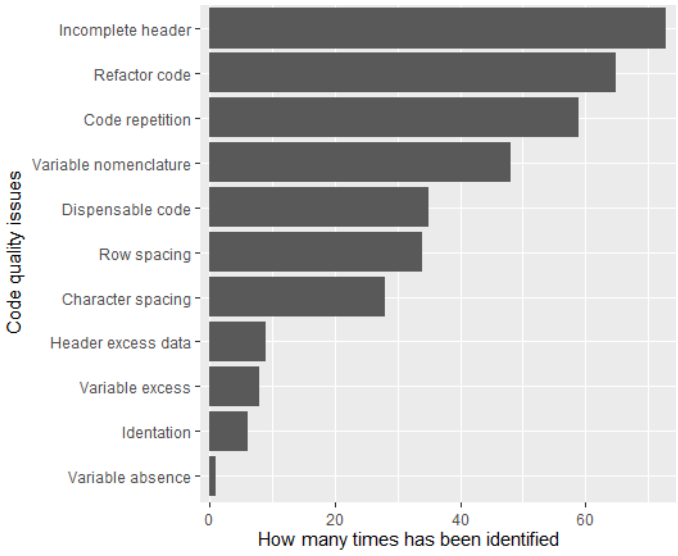


Fig. 4. Ranking of the code quality issues reported in the students' hints.

It is possible that in some moment the students were too tired or bored to continue to evaluate with precision and maybe that's why they elaborated so many hints about incomplete header, because it is an issue easy to identify, so that it was a type of hint elaborated by practically every student.

After the incomplete header, the larger amount of hints was about refactor code and dispensable code. In this research, we

classified as refactor code, the suggestions of good practice and improvement of the code's legibility without removing or adding new functions. While dispensable code, as the name already says, we defined as parts of the code that, if removed, won't influence the program's functioning.

We noticed that hints about the use of constant variables and variables types were not identified by students, in this case, these hints were elaborated only by the teachers. The opposite happened with the hints about excessive header data and adding comments to the code. The last was cited 4 times, but it is not considered good practice. Besides these, there were hints about grammatical errors in the header's texts and code prints.

**RQ3:** Can the students of introductory programming from CS/UFCG give useful feedback to improve the code quality of their pairs'?

To answer this research question, we selected all hints elaborated by the students and showed it to the teachers who classified them as useful or not. In this context, we defined with useful the hint correct, personalized and clear enough to improve source code from it. As to not useful, we classified hints: (i) correct, but not clear enough to improve source code from it, (ii) irrelevant on improving source code or (iii) incorrect. For these definitions, we considered, beyond correctness, the feedback detailing, as this is the differential of including the student in this activity. Figure 5 shows the general view of the obtained data.

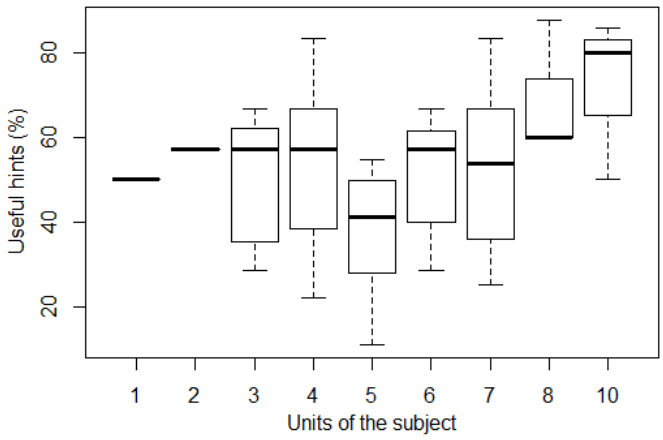


Fig. 5. Useful hints gave by students'.

We verified that more than 50% of the hints from 49 of the 75 participant students were classified as useful. We also verified that most hints were about complexity problems. As in the similarity, the last units got better results and unit 5 showed unsatisfactory results. On average, students gave 3 hints per survey question, little more than 89% of them being correct, even if not useful. However, teachers mentioned that some hints were poorly detailed or had problems with the program's terminology.

The majority of the students were capable of providing useful feedback, especially the ones in the last units.

Regarding the differential of the student's feedback be personalized, we compare a feedback given by a student and the automatic feedback of QCheck (Figure 6). We realized that the student's feedback can be more detailed, and able to help other students with greater difficulties and also allow those that gave feedback review the programming concepts.

| (a) | (b) |
|---|---|
| It appears that your program has too many operations.<br><br>. | Instead of using an "if" to check if the number is even, use an "elif".<br><br>Use space between the operators and operands used in the expression assigned to the number (inside the "if" that should be an "elif").<br><br>Increase the counter at the beginning of the loop only, to avoid code duplication.<br><br>Leave an empty line between the loop and the print. |

Fig. 6. Automatic feedback (a), given by QCheck, and manual feedback (b), given by a student, to the code of a task shown in Figure 1.

So, even with the risks, we believe they can provide feedback about their colleagues' source code problems. Nevertheless, we advocate that feedback elaborated by more than one student tends to be more effective. We verified the effects of students' feedback on pairs' solutions from the preliminary study we described in the next Section.

### *A. Preliminary study on introductory programming subject from CS/UFCG*

We conducted a pilot experiment with students from introductory programming subject from CS/UFCG to verify if the hints can, indeed, improve their solutions qualitatively. 10 students participated (8 from unit 10 and 2 from unit 7). The experiment lasted 90 minutes and was divided in 3 moments: (1) The students elaborated hints on how to improve quality of the solution code of two colleagues to a certain assignment (that was solved by them at the beginning of the semester); (2) The students corrected their solutions according to the hints received from their colleagues; and (3) They classify the hints as useful or not. We handed out randomly, at the beginning of the study, which code would be evaluated by which student.

Each student elaborated and received at least 7 hints and around 6 of them were classified as useful by the student that received it. Overall, we identified only 3 hints classified as

useless. To verify if there were improvements in the solutions, we collected software engineering metrics to the submitted code. The metrics were: lines of code (LOC), cyclomatic complexity (CC), Halstead volume and style guide for Python code (pep 8). We compared the solutions in their original version (v1) and the version after the hints (v2), as Table IV shows.

TABLE IV
EXTRACTION OF METRICS.

| | CC | | LOC | | pep 8 | | Halstead | |
|---|---|---|---|---|---|---|---|---|
| Student | V1 | V2 | V1 | V2 | V1 | V2 | V1 | V2 |
| **A** | 6 | 6 | 18 | 18 | 0 | 0 | 93.77 | 91.38 |
| **B** | 7 | 8 | 21 | 19 | 12 | 11 | 230.75 | 162.85 |
| **C** | 6 | 6 | 19 | 18 | 6 | 1 | 77.71 | 51.89 |
| **D** | 8 | 5 | 19 | 14 | 2 | 5 | 158.12 | 96 |
| **E** | 7 | 5 | 26 | 19 | 0 | 2 | 100.08 | 78.95 |
| **F** | 6 | 6 | 19 | 19 | 0 | 1 | 91.38 | 88.81 |
| **G** | 6 | 6 | 19 | 16 | 0 | 3 | 122.62 | 91.38 |
| **H** | 6 | 6 | 17 | 17 | 0 | 1 | 91.38 | 91.38 |
| **I** | 6 | 6 | 19 | 18 | 2 | 2 | 96 | 91.38 |
| **J** | 8 | 6 | 22 | 18 | 0 | 3 | 82.04 | 51.89 |

The CC metric measures the number of paths linearly independents of a program. In this metric considerers conditioned and interactive structures, like conditionals and loops [6]. Looking at Table IV, we can see that there were improvement in some cases, but most part remained the same. Only one student did not improve on the second version. The LOC metric measures the size of program, through counting the number of lines from the source code. Only two students remained equal in both versions, the others improved this aspect in their solution.

The pep 8 metric, indicates code quality based in the Style Guide for Python Code, and we also verified improvements on this aspect in most cases. The Halstead volume uses the length and the vocabulary to give a measure of the amount of code written, the measure consists in counting the number of operators and operands in a program. In this metric, all students showed significant improvements between the versions. We verified that students were able to improve the quality of their solutions with the hints, especially in aspects like excessive lines and code refactoring. We also realized that in general there were more useful hints. So, even though we can't obtain deep conclusions, due to the sample size, the collaborative learning strategy shows itself to be promising.

## VII. THREATS TO VALIDATE

Overall, the evaluation project wanted to minimize many of the discussed threats on this section. To organize this section, we classified the threats to validate using the following categories: Conclusion, Internal, Construct and External [19].

### A. Conclusion

The threats related to this category concern about the conclusion of the experiment. Due to the methodology based on the inverted classroom, where the classroom is only for discussion and doubts, the experiment sample's size could have been short to obtain deep conclusions about the results. However, making the Sample Size Calculation, we found out that our sample represents the study's population with sample error close to 7% and with 95% of confidence level. Another threat is that we assumed that teachers give useful feedback about code quality problems and based our analysis on problems mentioned and assessed by them in the subject.

### B. Internal

As the study involves active human participation, it inclines itself for internal threats. It is possible that the moment that the experiment happened (class time) and the fact that it was not previously announced could have affected the results. However, we minimized this threat by allowing the students to participate without interference from their teachers or other students. It is important to consider that, once the students evaluated many solutions, it is possible that in some moment they are too tired or bored to evaluate with precision.

We needed volunteer researchers so the experiment could be applied simultaneously, so, the explanation given by each researcher could have influenced the students' understanding of the experiment. To minimize this threat, we elaborated a script and trained for researchers volunteers.

### C. Construct

This study checks many code quality issues different from different aspects, and some constructs may not be measured by the questions. To minimize these threats, we selected empirically validated techniques and commonly used in the scientific empirical studies from the community of computer science education.

### D. External

The participants of this study are representative only to the context of introductory programming subject of CS/UFCG and to the semester where it happened. In this way, we might not be able to generalize the results of this experiment to other contexts. This study must be replicated in other introductory programming subjects to obtain more generic results.

## VIII. CONCLUSION AND FUTURE WORK

As we've mentioned, the studies in the automatic feedback field for programming assignments approach mainly functional aspects. Although there are tools that analyze code quality, their feedback may not be detailed enough and, to be effective, they require teachers' manual analysis. In this way, we investigated if, when including students as evaluators, we could provide personalized feedback. So, in this study, the main question is if students can qualitatively evaluate their pairs' programs.

With the obtained results, we verified that most of the hints elaborated by the students are related to code complexity. We also verified that students' ability level was what affected more in the similarity to the teachers' hints and in the amount of hints considered useful in this study. That is, the students from the more advanced units identify problems and provide better feedback.

We also noticed that, in a general way, the students can identify code quality issues and elaborate good hints in a significance level, even though not ideal. Nevertheless, a group of students, from different units, have shown good results. We advocate that if more than one student elaborate hints to a determined solution, the feedback tends to be effective, or more effective than hints from only one student.

As a future works, we intend to: (i) create more activities in which the students will revise code from their colleagues and verify the effect of feedback on their solutions; (ii) develop a model to automatically identify students' profile that can elaborate proper feedback; and (iii) study strategies to automate feedback in a lint-like, powered by the personalized tips elaborated by the students. This article contributes to studies about the effects of collaborative learning strategies in the context of introductory programming courses. However, studies are still needed to examine the reasons why students produce low-quality code, how they deal with quality issues and what are the effects of collaborative learning strategies use. The collaborative learning has an important role in the Computing teaching, not only to minimize scale problems but also to develop critical sense on students.

## IX. Acknowledgment

We thank CAPES for supporting this work. This work was partially sponsored by the agreement No 08200.315131/2016-10 between UFCG and ePol/DPF.

## References

[1] C. Wilcox. The role of automation in undergraduate computer science education. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 90–95, New York, NY, USA, 2015. ACM.

[2] J. Gao, B. Pang, and S. Lumetta. Automated feedback framework for introductory programming courses. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 53–58. ACM, 2016.

[3] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. *ACM SIGPLAN Notices*, 48(6):15–26, 2013.

[4] S. Wasik, M. Antczak, J. Badura, A. Laskowski, and T. Sternal. A survey on online judge systems and their applications. *arXiv preprint arXiv:1710.05913*, 2017.

[5] H. Keuning, B. Heeren, and J. Jeuring. Code quality issues in student programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '17. ACM, 2017.

[6] E. Araujo, D. Serey, and J. Figueiredo. Qualitative aspects of students' programs: Can we make them measurable? In *Frontiers in Education Conference (FIE), 2016 IEEE*, pages 1–8. IEEE, 2016.

[7] Pep 8: style guide for python code. http://legacy.python.org/dev/peps/pep-0008/. Accessed: December/2017.

[8] B. S. Bloom. Learning for mastery. instruction and curriculum. regional education laboratory for the carolinas and virginia, topical papers and reprints, number 1. *Evaluation comment*, 1(2):n2, 1968.

[9] M. Piteira and C. Costa. Learning computer programming: study of difficulties in learning programming. In *Proceedings of the 2013 International Conference on Information Systems and Design of Communication*, pages 75–80. ACM, 2013.

[10] N. Schunk and K. Petermann. Measured feedback-induced intensity noise for 1.3 $\mu$m dfb laser diodes. *Electronics Letters*, 25(1):63–64, 1989.

[11] J. Hattie and H. Timperley. The power of feedback. *Review of educational research*, 77(1):81–112, 2007.

[12] Elena L Glassman, Aaron Lin, Carrie J Cai, and Robert C Miller. Learnersourcing personalized hints. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*, pages 1626–1636. ACM, 2016.

[13] C. Kulkarni, K. Wei, H. Le, D. Chia, K. Papadopoulos, J. Cheng, D. Koller, and S. Klemmer. Peer and self assessment in massive online classes. In *Design thinking research*, pages 131–168. Springer, 2015.

[14] M. Joy, N. Griffiths, and R. Boyatt. The boss online submission and assessment system. *Journal on Educational Resources in Computing (JERIC)*, 5(3):2, 2005.

[15] C. D. Hundhausen, P. Agarwal, and M. Trevisan. Online vs. face-to-face pedagogical code reviews: an empirical comparison. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 117–122. ACM, 2011.

[16] 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*, 59(2):412–422, 2012.

[17] B. Bloom et al. Handbook on formative and summative evaluation of student learning. 1971.

[18] S. Brinkman and S. Kvale. Interviews: Learning the craft of qualitative research interviewing. *Aalborg*, 24:2017, 2015.

[19] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.