# Qualitative aspects of students' programs: Can we make them measurable?

Eliane Araujo, Dalton Serey, Jorge Figueiredo
Department of Computer Science
Federal University of Campina Grande
Campina Grande, Brasil
{eliane, dalton, abrantes}@computacao.ufcg.edu.br

***Abstract* — Proper feedback can leverage students to better understand their difficulties and shorten the characteristic program-submit-refactor cycle of programming exercises. The ideal feedback is the result of a human inspection and analysis considering both functional and qualitative aspects of programs produced by students. On the other hand, automated assessment systems can provide rapid, cheap and standardized feedback. In this paper, we focus on measuring aspects of code that instructors usually assess in programming assignments which are deemed unmeasurable: qualitative aspects that go beyond functional correctness. The aim of this work is to produce richer feedback messages that go beyond functional correctness as it involves code quality issues. We found that if an instructor is required to produce a reference solution for a programming assignment, then most of the expectations the instructor has about a student's code quality are concretely present in the reference solution. Based on this idea, we proposed and evaluated a set of candidate quality measures using the assignment's reference solution as a baseline. The results showed that they seem to capture what is usually considered to be the subjective and qualitative aspects of an instructors' assessment. We used these findings to generate feedback and conducted an experiment to evaluate it effectiveness. The results enforce that this kind of feedback stimulates students to improve their quality code in a higher degree than purely functional feedback, yet it still can be fully automated.***

***Keywords*— human factors; experimentation; automated assessment; programming; coding standards; software quality**

## I. INTRODUCTION

A fundamental aspect of the programming learning process is providing feedback to the students about their assignments. Further than showing that learning outcomes are being met; it can boost the student self-confidence or help her modify a recurrent behavior. Richer feedback can leverage students to better understand their difficulties and shorten the characteristic program-submit-refactor cycle of programming exercises. Currently, the richest possible feedback on students' programs is the result of human inspection and analysis of both functional and qualitative aspects of the code. In programming courses, automated tools play an important role as they allow for rapid, frequent, cheap and standardized feedback. They free instructors to direct their efforts to higher levels of analysis. For the last two decades, different strategies have been proposed to develop these tools. Several approaches for automatically assess programs were adopted [1]. Most of the systems, however, are based on functional analysis of the programs.

We focus on whether we can measure what is usually seemed unmeasurable: the so-called qualitative and subjective factors considered by instructors when they assess a program as a solution for a programming assignment. Studies discussed that functional correctness is the most important component when assessing a program [2]. Our experimental results also corroborate with this idea, since automatic grades, obtained from tests' results, are strongly correlated with manual grading (Spearman's *rho* of 0.85). While it explains at large extent the assignment score, tests results are insufficient to give students personalized rich feedback. Dedicated instructors enrich their feedback with advices on the code quality and help students to reason on their solution. We claim that while many subjective aspects are indeed unmeasurable, certain objective and measurable factors of the code can reflect most qualitative aspects reported on the feedback provided by instructors in their manual assessments.

In this paper, we intend to generate automated code quality feedback so that we can stimulate students to reflect on their code, besides functional correctness. As a baseline for such quality, we used the reference solution provided by the instructors for the assignment. This solution must convey the learning outcomes students have to master, as well as, the expected code quality. We propose a set of software measures to express qualitative aspects. They are based on software quality metrics, largely used by the industry and referred in other academic initiatives towards novice programming [1][3][4]. First, these measures are extracted from the reference solution code and from the student code. In sequence, we calculate the relation between them. Using this data, the system will generate and provide a feedback message to the student, i.e. a hint of what it could be improved in order to obtain better quality code. It is noteworthy to observe that this feedback must be delivered only to functionally correct submissions.

This proposal was evaluated following a two-pronged approach: a case study and an experiment. The case study aimed to investigate the validity of the suggested measures as surrogates of the quality expected by the instructors on the student code. Our dataset was composed by 403 functionally correct submissions, from 102 students, referring to 12 different programming assignments. Our results showed that

students whose programs have measurements close to or better than the measurements of instructor's reference solution program tend to obtain higher grades. In consequence, we could say that the proposed approach do capture most of the quality rationale behind the program's assessment performed by instructors. At the cost of providing a reference solution for each programming assignment, the measures can be fast, automatically produced and used to deliver feedback through automated assessment systems.

The experiment evaluated the impact of generating quality feedback in students' final code. As results, we perceived that the quality feedback promote reflection about the implementation and directs the student to refactor the code, as the solution is already functionally correct. We observed and also confirmed in hypothesis tests that students who receive such enriched feedback (correctness + quality) tend to make more submissions than those who do not. Also, 66.67% of the students that received quality feedback managed to deliver a better code.

This paper is organized as follows: Section II discusses how instructors consider qualitative aspects of programs when they assess students' code. It also describes a set of measures proposed to capture some of these aspects. Section III describes the case study conducted to evaluate the proposal, including a discussion about our findings. Section IV presents the experiment on generation and delivering of quality feedback. A discussion about the obtained result and its findings is presented on section V. We considered and argue about validity threats on section VI. Section VII reports some related works related. Finally, we address the conclusions of the study along with directions to further works in Section VIII.

## II. Assessment of Qualitative Aspects of Student's Code

Assessment is a central activity in higher education and is considered a core component for effective learning [5]. An essential part of assessment is the feedback it produces: to the instructor, to the student and to the educational institution. In educational research, assessments can be characterized according to its purpose as: (a) formative, to support and improve students learning skills and (b) summative, to make a judgment and verify if the learning objectives have been reached by the student [6]. Our study focuses on code quality and aims to rapidly deliver valuable formative feedback in order to motivate students to produce a better code. In the context of our work, formative feedback includes all the information and communication exchanged by students and instructors that contribute to modify an erroneous behavior and to demonstrate that expected abilities have been mastered.

We are especially interested in approaches to produce automated assessment feedback for programming assignments. In programming courses, enough practical activities are paramount to students effectively achieve learning goals. Automated assessment systems (AAS) are essential to support such a great number of programming assignments and also provide student feedback. It produces objective and consistent feedback to students, while it mitigates the heavy workload of the instructors when they perform manual assessment [1][7].

There are many automated assessment systems focusing on introductory programming assignments. Some of them provide grading support [8][9] and are classified as grader systems. They normally take into account factors such as: deadline penalty, resubmission policy, type of errors, test coverage, etc. In general, AASs employ comparable approaches when assessing students programs and provide common features [6]. A typical system executes a set of test cases, provided by instructors, and compares the expected output to the observed output from students' programs. The most common feature assessed by automated systems is functional correctness.

However, as observed by Buffardi and Edwards in [10], while "automated grading systems help students identify bugs in their code, the systems may inadvertently discourage students from thinking critically and testing thoroughly and instead encourage dependence on the instructor's tests". A similar behavior could be noticed in regards to code quality. Many students submit their programs until they pass the instructors' tests or program in a trial-and-error mode, without critically thinking on their solution. Another typical behavior is to assume that the program is finished when it receives an "ok" or "green-bar" of a test-based automated assessment tool. A careful look exposes that some programs could have their quality improved in different ways, preserving their functional correctness. If students were not pushed to review and refactor, they will simply move forward to another assignment and, maybe, will repeat the same programming pitfalls. The purpose of this work is to promote formative feedback about qualitative aspects of code, which are usually neglected by many test-based automated assessment systems.

Instructors approach the manual grading activity in different ways but usually agree whether a program is "very good" or "very bad" [11]. Besides correctness, there are other factors weighed by instructors in manual assessment in terms of code quality. For example, a program that is abnormally longer than the others and solves the same problem needs a closer look. Other common pitfalls of programming beginners are nesting multiple "if" statements and using unnecessary variables to compute temporary values. There are software metrics that could be statically extracted from the code at a low cost and serves as input to a quality analysis [1]. We evaluated in this work: logical lines of code (lloc), Halstead volume (h), cyclomatic complexity (cc) and adherence to coding standards. In short, these measures stand for:

- lloc: The number of lines effectively used as programming language code statements. This measure does not consider blank lines, comments and headings.
- h: Metrics proposed by Halstead aims to evaluate a program regarding on static analysis. The measurement consists on counting the number of operators and operands in a program [1]. In this study, we have measured the Halstead volume.
- cc: It was conceived by McCabe [12] and refers to the number of linearly independent paths of a program. Each decision in a program can lead to a different path. So to

compute cc, there are considered not only conditional structures but also iterative structures, such as for and while loops.

Ala-Mutka study pointed that: to make software metrics relevant to students they need to be comparative [1]. She argued "there is no sense in requiring students to submit a program that has a complexity number X, or contains Y lines of code". On the educational context, there is a benefit, which could not be experienced in real world software: the instructor referential solution approximates to the best possible solution to the problem. The measurements extracted from the student source code will be compared with those extracted from reference solution code. The rationale is that the measures extracted from the reference solution are an idealized target expected by instructor for all students' submissions.

We have also measured adherence to coding standards in a metric named: RPEP8. As Python is the programming language adopted by the course we have collected our data, we relied on the coding standards defined by Python community in PEP8 [13]. The number of pep8 violations indicates how distant a given code is from the defined coding standard. This measure is calculated differently from the others, as we cannot compare the violations happened in the student code with the violations that happened in the reference solution overlooking their nature. Furthermore, ideally should not exist pep8 violations in the reference code. In practice, a reduced number of violations indeed exists and are considered to be acceptable. In order to calculate this measure, we extract the number of pep8 violations for each submission for a given assignment. Then, we rank the number of violations of these submissions.

The value of RPEP8 for each submission is its ranking position. The other measures are defined as the ratio of the measurement extracted from the student submission to the measurement extracted from the reference solution.

TABLE ITable I presents the measurements we proposed to assess code quality along with its acronym. From this point forward, we are going to refer these measurements by the acronyms.

For example, if the value of RLLOC for a particular code is 1.2, it means that: the code provided by the student to that programming assignment is 20% greater than the size of the reference solution code for that assignment. Conversely, if the value of RLLOC was 0.8, the code provided by the student is 20% smaller than the reference solution code. RCC and RH calculation is done similarly.

TABLE I. MEASUREMENTS PROPOSED TO ASSESS CODE QUALITY

| Acro. | Description | Formula |
|---|---|---|
| RLLOC | Ratio between reference solution's *lloc* and student's code *lloc*. | lloc(student code) / lloc(reference solution code) |
| RCC | Ratio between reference solution's *cc* and student's code *cc* | cc(student code) / cc(reference solution code) |
| RH | Ratio between reference solution's *h* and student's code *h*. | h(student code) / h(reference solution code) |
| RPEP8 | Ranking position of the number of pep8 violations of the student's code. | |

## III. CASE STUDY: MEASURING STUDENT CODE QUALITY

Our initiative toward generating and delivering formative feedback about qualitative aspects of code started on performing an empirical study that aimed to evaluate the measures we proposed as surrogates of some extent for the human quality assessment of students' programs.

### A. Research Context and Data Collection

In the case study, we conjectured that there is a set of measurements, automatically obtained, that can capture quality aspects weighed by instructors when they assess and manual grade a student program. In order to test it, we have formulated the following research question:

**RQ1:** Can the measures RLLOC, RCC, RH and RPEP8 explain the differences observed on the grades, manually assessed, of functionally correct submissions?

In answering this research question, we investigated whether the student's code measurements were similar or better than the measurements of the reference solution, the instructor would perceive a better code quality. In consequence, it would deserve a better grade. Thus, if code quality impacts on grades, they could be captured by the proposed metrics.

We collected students' submissions of programming assignments from an introductory programming course of our university. The data was collected using an automated assessment system developed in-house and tailored for our introductory programming course. The dataset was composed by 403 functionally correct submissions, from 102 students, referring to 12 different programming assignments that appeared in midterm exams. Experienced instructors manually graded them in a scale from 0-10. In our study design, these values correspond to the dependent variable *ig*. The measures RLLOC, RCC, RH and RPEP8 are independent variables. We used radon [14], a free Python tool, to compute raw metrics: *lloc*, *h* and *cc*. The number of *pep8* violations was extracted using a script produced by Python developers' community [13]. It is worth to note that we used the reference solution version provided by the instructor who graded the assignment when extracting the measures RLLOC, RCC, RH and RPEP8 of the students' submissions.

### B. Data Distribution and Analysis

Fig. 1 shows the distribution of instructor's grades of functionally correct submissions. These submissions obtained "green-bar" as passed all automatic tests provided by instructor. If they were automatic graded, all of them would obtain the highest score: 10. However, the figure shows a left skewed distribution and only 29.5% of the evaluated submissions got the highest score. If the assessment were relied solely on automatic tests, more than 70% of the submissions would obtain a grade greater than a human instructor thinks it deserves.

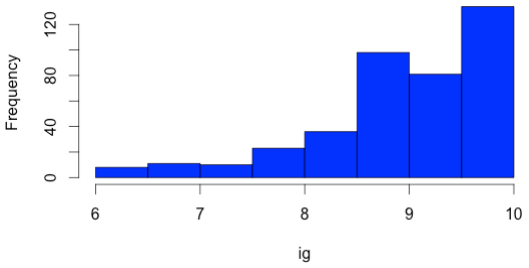


Fig. 1. Distribution of manual grades assigned to functionally correct submissions

The grades produced manually by the instructors take into consideration a set of criteria that goes beyond functional correctness, as it could be apprehended by the grades' variance. A qualitative evaluation of those submissions revealed structural code problems (such as incorrect use of conditional structures) that were not captured by traditional functional test. Fig. 1 exposes that functional correctness, alone, does not reflect the instructor manual assessment.

### C. *Results and Discussion*

This subsection reports the results of the studies to answer our research question: Whether the proposed quality measurements can explain the differences observed on the scores of functionally correct submissions.

In order to answer this question, we investigated the contrast between the student's code measurements and the reference solution measurements'. We used Wilcoxon rank sum test to compare submissions' grades. This non-parametric statistical test assesses if two independent distributions are the same. The null hypothesis is that the population is the same against the alternative hypothesis that the population differs in a location measure, in this case the median of the grades. Since this test is based on rank observations, it makes no assumptions about the normality distribution of the assessed variables.

We divided the distribution in two groups according to its measurements: (1) equal-lower than 1; meaning that the measures of student's code are equal or better than the reference solution code or (2) greater than 1; it means that measures of the student code are greater than the measures of the reference solution code. For example, in a given student submission for a programming assignment it was accounted 3 *pep8* violations. The reference solution code, for that same assignment, accounted 1 *pep8* violation. This submission is part of the group 2. In this sense, each metric was analyzed individually.

Tests results confirmed that RLLOC, RCC, RH and RPEP8 do capture the notion of quality, as the distributions differ in their medians. Instructor's grades for equal-lower group are higher, on average, than the grades of the other group with adequate statistic significance (p-value < 2.20E-16, 0.05 significance level). Hence, it rejects the null hypothesis in favour of the alternative. The results reveals, at least for these data, the better the measurements the better the grade. As practical significance of this result, we can state that stimulating students to consider not only program correctness but also its quality is indeed beneficial.

Figure 2 shows boxplots of *ig* (instructor's grades) distribution. In the first boxplot, it can be noticed a wider variation on *ig* on the first group of submissions (RLLOC(x) > 1). Apart from some outliers, the second group of submissions (RLLOC(x) <= 1) presents a narrower variation and a higher median value. A similar behavior could be observed on the other plots. Besides the hypothesis test, we performed a correlational analysis to investigate the association of each measure (RLLOC, RCC, RH and RPEP8) with *ig* using data collected from all 12 programming assignments. At this point, we must recall that RLLOC, RCC and RH are ratio metrics. It means, for example, that we are not observing the correlation between the size, in *lloc*, of a student's submission and its grade. We are measuring the relation between the size of a student's submission and the size of the instructor's reference solution. Then, whether this value correlates with the programming assignment grade.

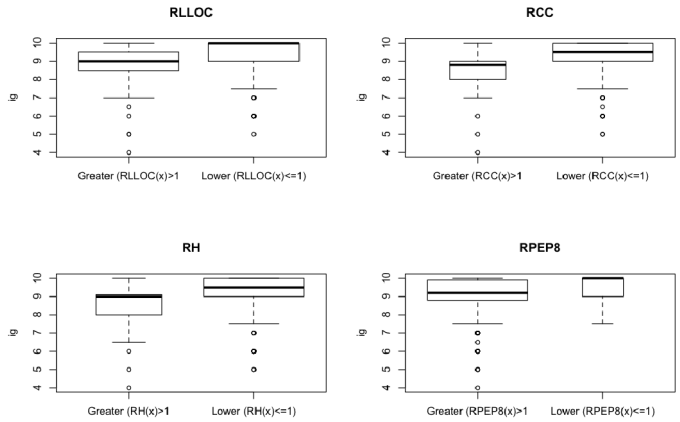


Figure 2. Boxplot of instructor's grades and the values of each RLLOC, RCC, RH and RPEP8

We used Spearman's rank correlation coefficient to measure the extent of the correlation and found that 91.67% of Spearman's rho values are negative. What means that as one variable increases, the other decreases. This behavior corroborates our hypothesis: the smaller the measure the greater the value of *ig*. The strongest correlation found, in absolute value, is between RCC and *ig* (-0.94 Spearman's *rho*). In general, the strongest correlation values were observed between RLLOC and RCC measurement. There were also *rho* values near zero, meaning that the correlation is negligible or inexistent in some cases.

## IV. Experiment: Evaluating Quality Feedback Generation

In the previous sections, we have investigated and proposed a set of measures that can give us indicators of code quality in student's programs. In this section, we will describe the experience of using these measures to generate and deliver feedback messages to students. We wanted to investigate the effectiveness of the quality feedback generation approach. If students, in fact, care about the feedback received and actuate

in their code so that it improves. We performed an experiment animated by two research questions:

**RQ2**: Students who receive quality feedback about their submission tend to make more submissions, after the first correct one?

**RQ3**: When students receive quality feedback about their submission they tend to deliver a better quality code?

### A. Experiment Setting and Data Collection

We performed an experiment in the same introductory programming course of the case study reported previously. We proposed a programming exercise to 48 students, divided into experimental and control group. Students' submissions have their functional correctness automatically tested. We considered that a student failed to solve a problem if his or her submission fails in at least one test case. Only 20.8% (13 students) failed the assignment. The quality feedback was generated and delivered only to students of the experimental group who succeeded.

We instrumented the automated assessment system already used in the course to perform quality checking and feedback generation. The warning messages are presented in a command-line interface, just after the student submits her code to automatic testing and receives the results. We empirically established a threshold for each quality measure (RLLOC, RCC, RH and RPEP8) in order to show the warnings: when it reaches 1.2, i.e. a value 20% greater than the same measurement in the reference solution, a message is produced and delivered to the student. Table II presents the warning messages generated for the other measurements. They represent advices, rather than prescriptions, in what could be done to improve the code. We have also added an extra warning message regarding to the number of lines of the heading the student are supposed to add in their code. This is an "easy-to-solve" warning aimed to make students learn by themselves how the cycle submit/receive feedback/refactor works. This was useful, because no directions were given about how to proceed after the feedback message during the experiment. RPEP8 warnings messages were translated from English and slightly modified from the original style checker implementation [13].

TABLE II. MAPPING OF WARNING MESSAGES DELIVERED TO STUDENTS. MESSAGES PRESENT HINTS ON HOW TO IMPROVE CODE QUALITY

| Measur. | Message |
|---|---|
| RLLOC | "It appears that your program has too many lines of code." |
| RCC | "It appears that your program has too many conditionals structures or loops." |
| RH | "It appears that your program has too many operations." |
| Header issues | "It appears that your program has few header lines." |

### B. Programming Assignment

The problem chosen is a typical programming assignment the students are able to solve after been exposed to conditional and repetition control structures lectures. It is the well-known 3x+1 problem, or Collatz problem. Fig. 3 presents the reference solution provided by the instructor who proposed the assignment "Collatz Life". It prompts the student to inform the number of iterations (lifes) does it take to a given number to converge to 1, repeating the process:

$$f(n) = \begin{cases} n/2, & \text{if } n \equiv 0 \pmod 2 \\ 3n+1, & \text{if } n \equiv 1 \pmod 2 \end{cases} \quad (1)$$

```
1 # Collatz life
2 # Reference solution
3
4 N = int(raw_input())
5 life = 1
6 while N != 1:
7     if N % 2 == 0:
8         N = N / 2
9    else:
10        N = 3 * N + 1
11    life += 1
12
13 print life
```

Fig. 3. Python reference solution provided by a teacher to Collatz programming assignment

### C. Results and Discussion

This subsection reports on the results of the studies performed to answer the second and third research questions of this work.

The second research question posed the investigation: Whether the students who receive quality feedback about their submission tend to make more submissions, after the first correct one. The data collected in the experiment indeed revealed that students of the experimental group (who received quality feedback) make more subsequent submissions than the students of the control group. The median of submissions performed by the subjects on the experimental group was 2.5 greater than the median of submissions performed by control group subjects. We studied this behaviour, performing Mann-Whitney nonparametric hypothesis test. As a result, we rejected the null hypothesis in favour of the alternative (p-value = 0.009, with 0.05 significance level). This means that, at least for our data, students who received warning messages as feedback about their code quality tend to make more submissions of that same assignment. As practical significance, we can state that: apparently, quality feedback messages are took into consideration by students and not ignored by them. It encourages students to reflect on their code besides it correctness.

In the third research questions we examined: Whether students that receive quality feedback about their submission tend to deliver a better quality code. It evaluated if quality

measurements of the last submissions of the students of each group differs depending on the their exposition to feedback quality warnings. We have performed the same hypothesis test and verified that it is possible to reject the null-hypothesis in favor of the alternative (p-value = 0.0267, with 0.05 significance level). This means that, at least for our data, the number of quality warnings of the last submission from the students of experimental group is lower than number of quality warnings of the last submission from the students of the control group.

We have evaluated each student's submission from the experimental group in order to verify if, provided they have access to the quality feedback, they managed to produce a better code. This qualitative analysis uncovers details that could not be captured by statistical tests. We have observed that 66.67% of the students which received at least one quality feedback warning about their first submission, presents a positive derivative: they succeed on solving the feedback warning and reduced the number of warnings obtained in relation to the previous submission. Our results indicates that students are able to actuate on their code based the quality warning feedback messages. It suggests that this type of feedback is useful and adequate to promote the improvement of student's code. Fig. 4 shows the first and the last submissions of a given student along with the quality feedback messages it received.

Data collected from control group, reveals a typical behavior of our students: they assume their submission is done when it receives an "ok" or "green-bar" from a test-based automated assessment tool. A careful look exposes that some programs could have their quality improved in different ways, preserving their functional correctness. If students were not pushed to review and refactor, they will simply move forward to another assignment and, maybe, will repeat the same mistakes in the next assignment.

## V. Discussion

Functional correctness is the most important aspect of assessment in manual grading. However, there are other features took into account by the instructors when they are grading. We claim that subjective and quality factors impact on instructors' assessment, besides functional correctness. Subjective factors, in this context, are those inherent from human beings: such as affective/emotional (willingness to give good grades or the opposite, fatigue, etc.) and errors/mistakes that may occur and are difficult to identify and to explain.

Whilst subjective factors would remain unmeasurable, this study revealed that there are some quality factors that influence instructors' assessment of programming assignments and can be automatically measured. The novel approach of this work is not the use of software metrics to assess student's code quality, but to compare student's submissions measurements with the measurements of the reference code, provided by the instructor. This indirect method reveals the target of quality aspects expected by the instructor for a given programming assignment. We claim that instructors idealize a reference solution when they assess students' code. They grade the assignment by comparing and assessing how similar the students' code is to its own reference solution code.

(a)

```
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
```

(b)

```
It appears that your program has too many operations.
```

(c)

```
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    cont += 1
8    if number == 1:
9        break
10    elif number % 2 == 0:
11        number = number/2.0
12    else:
13        number = 3 * number + 1
14 print cont
```

Fig. 4. Code (a), is the first correct submission of the student. It caused the quality warning (b) "It appears that your program has too many operations." regarding to the lines 8,12 and 15. Code (c) is the last submission made by the same student. It caused no warning messages. The student "solved the warning" making a better use of conditional structures and reducing the number of lines with duplicated code.

In fact, analogous or, even better solutions could appear when assessing students' submissions. This circumstance does not invalidate our results, rather is accommodated by the proposed metrics.

It is important to observe that this approach does not intend to provide an exhaustive analysis of the code quality of the programming assignment, including aspects such as: problem solving strategy, algorithm and solution design. It focuses on readability as a relevant indicative of code quality, mainly in introductory programming. In fact, we planned to deliver quality feedback only to functionally correct submissions. We believe that the problems the submissions we focused presents are, in great part, on readability nature and could be captured by the metrics we proposed. However, it is only an anecdotal suspicion, as our empirical studies were not intended to prove this assumption.

In a different perspective, from the observed in this study, the quality information could be delivered to students whose submissions are functionally incorrect. The measure RLLOC, for example, would help students to realize that the code is very far from the correct solution and there is a need to start over. Another possible approach is to evaluate the semantic of each measurement individually. For example, if one's submissions consistently present high RCC values, it possible suggests difficulties in mastering the concepts related to conditional or iterative structures. This type of information would be useful to instructors when monitoring the students' learning process.

## VI. Threats to Validity

### A. Internal Validity

*Human assessment:* As expected in a study that involves human assessment, human factors threaten its validity. We collected instructor's grades of a set composed by 12 programming assignments as baseline for our analysis. The grades were provided by four instructors in different moments along the course, as a result of a manual inspection. We believe that this threat is diminished in the course we collected our data, since instructors share the same marking criteria as defined in a document of assignment rubrics [15].

### B. Construct Validity

*Reference solution:* We used the programming assignment reference solution as the target of expected code quality measurements. However, different instructors may vary the way they produce their reference solutions for the same assignment. In order to mitigate this threat, we qualitatively evaluated the reference solutions provided by the instructors of the course to each programming assignment of the dataset. We analyzed their solutions and assured that they were very similar. We also found out that the metrics values extracted from their solutions code presents little variance and high degree of concordance. To perform this quantitative evaluation, we used Jaccard distance approach to measure the dissimilarity between the original reference solution and the other solutions. In this approach, we performed a pairwise comparison between each value of the vector of measurements extracted form the solutions' code. In theory, the value of Jaccard distance may vary from zero (no distance) to one. In this study, we found distance values ranging from 0 to 0.293. This means that the instructors have a strong agreement about the expected assignment solution code and about the level of quality that could be apprehended by the metrics. It shows that, even though the instructors have different background, they have a consistent thinking about the problems' solutions. Finally, we chose the reference solution code proposed by the same instructor who created the assignment.

*Set of software quality metrics:* We have chosen a set of software quality metrics that are known to be representative of good quality code [4][16] and are obtained through static analysis. We left out of the scope of this study efficiency metrics, which are obtained dynamically. Those metrics might improve students' program quality assessment [2]. However, we believe this is only a minor threat, since introductory programming assignments are usually specifications to solve a limited problem that produces relatively small programs (in our database student's programs are composed by ~30 lines of code). In this case, efficiency measurements are not as relevant in a pedagogical context.

### C. External Validity

*Application of the results to other introductory programming courses:* Caution must be taken when applying the results of this study to other introductory programming courses with different assessment methodology and different programming languages. We rely on the quality of instructor's tests to assess functional correctness through an automated assessment system. Furthermore, we took advantage of Python well-defined coding standard that focus fundamentally on code readability. Although the findings could not be generalized to every course, the ideas and research methodology applied in this work can be adapted to be used in other contexts.

## VII. Related Works

Lister, Hanks and Murphy researched about the grading process [11]. They discussed about methods used by instructors to manually grade students' programs. They show that graders have different motivations to judge and also apply different approaches in their assessments. They conclude that the teaching community must discuss grading, to learn with each other in order to benefit their students. Our work also discusses the grading process. Differently from Lister, Hanks and Murphy study objectives', the purpose is not the grade at all, but is to reveal the quality features considered by the instructors when they are grading. We propose that software metrics could capture common quality factors usually cited in grading rubrics.

The use of software metrics, as a relevant aspect to be assessed in novice programming exercises, was referred in the study of Mengel and Ulans [3] and Cardell-Oliver [16]. They proposed that metrics could be used as an indicator of student performance. Cardell-Oliver proposes that software metrics can enhance the feedback delivered to students and to the instructors. Our proposed metrics, in some extent, are similar

to those presented in Cardell-Oliver study but different in its purposes. Our work goes beyond, as it reveals, at least for our data, which metrics are really relevant to provide quality and useful formative feedback to novice programming students. In this context, the instructor played a central role as we examined their assessments and reference solutions provided to the students programming assignments.

## VIII. Conclusion and Future Work

This paper proposed a set of measures with the aim to capture code quality and generate useful feedback for novice programmers. These measures, based on traditional quality software metrics, can be automatically obtained provided we have a reference solution. This instructor's provided solution encompasses the programming abilities and code quality expected for that assignment.

Firstly, we conducted a case study on a dataset composed by more than 400 functionally correct submissions, to 12 programming assignments, from about 100 students to evaluate our proposal validity. We calculated the proposed measurements for all submissions in the dataset and assessed how they compare to instructors' grades. We tested our hypothesis with adequate statistic significance. The results confirmed that RLLOC, RCC, RH and RPEP8 indeed capture, at some extent, the notion of quality as it is reflected in the variation of the instructor's grades.

Then, we have performed an experiment on generating quality feedback using the proposed quality measurements aiming to assess its influence on students' code. As results, we observed that students manage to work on their code and improve it, after receiving the quality feedback message. We confirmed using hypothesis tests that students who received such quality feedback are more likely to submit a larger number of revisions than those who do not. Furthermore, 66.67% of the students that received quality feedback delivered a revision with better quality code.

In this work, we used the solution code provided by the instructor who manually assessed the programming assignment as a reference. In future works, we will explore other alternatives of reference solutions such as: the median of the metrics between all instructors' reference solutions, the most common solution submitted by students, etc. With regard to feedback messages, we intend to create a hierarchy of messages, for each measurement, to deliver to students. The idea is to deliver messages whose contents vary from more general to direct, as the student tries to fix the warning. So, the system does not repeat the same messages among unsuccessful fixing attempts.

Finally, it is necessary to recall that the motivation of this study was to enrich the automated feedback provided to the students about their code submissions. We envision that using those quality measures, the students will obtain useful advise in how to improve their solutions. Also, it will leverage novice programmers to adhere to software quality premises since their early coding experiences.

## Acknowledgment

The authors would like to thank Programming I instructors, at UFCG for their valuable collaboration in producing reference solutions for the studied programming assignments and providing feedback about our data analysis. We are also thankful for our Computer Science students who diligently submitted their programs. This research was partially sponsored by the agreement No 754664/2010 between UFCG and ePol/DPF.

## References

[1] K. Ala-Mutka, "A Survey of Automated Assessment Approaches for Programming Assignments". Computer science education, vol. 15, pp. 83-102, 2005

[2] B. Cheang, A. Kurnia, A. Lim and W.-C. Oon. "On Automatic Grading of Programming Assignments in an Academic Institution." Computers & Education, 41, 121-131, 2003.

[3] S.A., Mengel, and J.V., Ulans. "A case study of the analysis of the quality of novice students programs." Proc. 12th Conference on Software Engineering Education and Training, pp. 40–49, 1999

[4] R. Pettit , J. Homer , R. Gee, S. Mengel and A. Starbuck. "An Empirical Study of Iterative Improvement in Programming Assignments". *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (SIGCSE '15). ACM, pp. 410-415

[5] J.W. Gikandi, D. Morrow, N.E. Davis, Online formative assessment in higher education: A review of the literature, Computers & Education, Volume 57, Issue 4, pp 2333-2351, 2011

[6] C. Douce, "Automatic Test-based Assessment of Programming: A Review", Journal on Educational Resources in Computing, Vol. 5, Issue 3, 2006.

[7] P. Ihantola, T. Ahoniemi, V. Karavirta and O. Seppälä. "Review of recent systems for automatic assessment of programming assignments". Proc. 10th Koli Calling International Conference on Computing Education Research (Koli Calling '10). ACM, pp. 86-93. 2010.

[8] J. Carter, J. English, K. Ala-Mutka, M. Dick, W. Fone, U. Fuller, and J. Sheard. ITICSE working group report: How shall we assess this? SIGCSE Bulletin, 35(4):107–123, 2003.

[9] P. Nordquist. "Providing accurate and timely feedback by automatically grading student programming labs". J. Comput. Small Coll., 23(2):16–23, 2007.

[10] K. Buffardi and S. H. Edwards "Reconsidering Automated Feedback: A Test-Driven Approach". In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*(SIGCSE '15). ACM, New York, NY, USA, pp. 416-420, 2015.

[11] S.Fitzgerald, B. Hanks, R. Lister, R. McCauley, and L. Murphy, "What are we thinking when we grade programs?", In Proc. of the 44th ACM technical symposium on Computer science education (SIGCSE '13), ACM, pp. 471-476, 2013.

[12] T. J. McCabe "A complexity measure". IEEE Transactions on Software Engineering, vol. SE-2, num. 4, pp. 308-320, 1976

[13] PEP 8 – Style Guide for Python Code [Online. Accessed in March, 2014] http://legacy.python.org/dev/peps/pep-0008/

[14] Radon – [Online. Acessed in March, 2014] https://radon.readthedocs.org/en/latest/index.html

[15] Becker K. "Grading programming assignments using rubrics". *Proceedings of the 8th annual conference on Innovation and technology in computer science education* (ITiCSE '03). ACM, New York, NY, USA, 253-253.

[16] R. Cardell-Oliver. "How can software metrics help novice programmers?". Proc. Thirteenth Australasian Computing Education Conference - Volume 114 (ACE '11) Australian Computer Society, Inc. pp. 55-62, 2011.