Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Automatização de Feedback para Apoiar o Aprendizado no Processo de Resolução de Problemas de Programação

Eliane Cristina de Araújo

Tese submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Educação em Ciência da Computação

Ph.D. Dalton Dario Serey Guerrero (Orientador)
Ph.D. Jorge Cesar Abrantes de Figueiredo (Orientador)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

A663a Araújo, Eliane Cristina de.

Automatização de feedback para apoiar o aprendizado no processo de resolução de problemas de programação / Eliane Cristina de Araújo. – Campina Grande, 2017.

167 f. : il.

Tese (Doutorado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2017.

"Orientação: Prof. Dr. Dalton Serey Guerrero, Prof. Dr. Jorge César Abrantes de Figueiredo".

Referências.

1. Educação em Ciência da Computação. 2. Ensino de Programação. 3. Feedback Automático. I. Guerrero, Dalton Serey. II. Figueiredo, Jorge César Abrantes de. III. Título.

CDU 004.41:37(043)

Resumo

No ensino de programação, é fundamental que os estudantes realizem atividades práticas. Para que sejam bem sucedidos nessas atividades, os professores devem guiá-los, especialmente os iniciantes, ao longo do processo de programação. Consideramos que o processo de programação, no contexto do ensino desta prática, engloba as atividades necessárias para resolver um problema de computação. Este processo é composto por uma série de etapas que são executadas de forma não linear, mas sim iterativa.

Nós consideramos o processo de programação adaptado de Polya (1957) para a resolução de problemas de programação, que inclui os seguintes passos [Pól57]: (1) Entender o problema, (2) Planejar a solução, (3) Implementar o programa e (4) Revisar. Com o foco no quarto estágio, nós almejamos que os estudantes tornem-se proficientes em corrigir as suas estratégias e, através de reflexão crítica, serem capazes de refatorar os seus códigos tendo em vista a boa qualidade de programação.

Durante a pesquisa deste doutorado, nós desenvolvemos uma abordagem para gerar e fornecer feedback na última fase do processo de programação: avaliação da solução. O desafio foi entregar aos estudantes feedback elaborado e a tempo, referente ás atividades de programação, de forma a estimulá-los a pensar sobre o problema e a sua solução e melhorar as suas habilidades. Como requisito para a geração de feedback, comprometemo-nos a não impor mais carga de trabalho aos professores, evitando-os de criar novos artefatos. Extraímos informações a partir do material instrucional já desenvolvido pelos professores quando da criação de uma nova atividade de programação: a solução de referência.

Implementamos e avaliamos nossa proposta em um curso de programação introdutória em um estudo longitudinal. Os resultados obtidos no nosso estudo vão além da desejada melhoria na qualidade de código. Observamos que os alunos foram incentivados a melhorar as suas habilidades de programação estimulados pelo exercício de raciocinar sobre uma solução para um problema que já está funcionando.

Abstract

In programming education, the development of students' programming skills through practical programming assignments is a fundamental activity. In order to succeed in those assignments, instructors need to provide guidance, especially to novice learners, about the programming process. We consider that this process, in the context of programming education, encompasses steps needed to solve a computer-programming problem.

We took into consideration the programming process adapted from Polya (1957) to computer programming problem-solving, that includes the following stages [Pól57]: (1) Understand the problem; (2) Plan the solution; (3) Implement the program and (4) Look Back. Focusing on the fourth stage, we want students to be proficient in correcting strategies and, with critical reflection, being able to refactor their code caring about good programming quality.

During this doctoral research, we developed an approach to generate formative feedback to leverage programming problem-solving in the last stage of the programming process: targeting the solution evaluation. The challenge was to provide timely and elaborated feedback, referring to programming assignments, to stimulate students to reason about the problem and their solution, aiming to improve their programming skills. As a requirement for generating feedback, we compromised not to impose the creation of new artifacts or instructional materials to instructors, but to take advantage of a usual resource already created when proposing a new programming assignment: the reference solution.

We implemented and evaluated our proposal in an introductory programming course in a longitudinal study. The results go beyond what we initially expected: the improved assignments' code quality. We observed that students felt stimulated, and in fact, improved their programming abilities driven by the exercise of reasoning about their already functioning solution.

Contents

1 Introduction **1**

1.1 General Problem and Proposed Solution 3

1.2 Overview of this Thesis 4

1.3 Contributions 5

1.4 Thesis' Outline 6

2 Background **8**

2.1 Considerations About Learning and the Science of Instruction 9

2.2 Programming Learning Challenges 11

2.3 Feedback and its Effects on Learning 16

2.4 Related Works 20

2.4.1 Automated Assessment in Programming Education 20

2.4.2 Intelligent Tutoring Systems 21

2.4.3 TST – Programming Assignments Testing System 25

3 Feedback Generation to Support Computer Programming Problem-Solving **28**

3.1 Research Roadmap 29

3.2 Methods 31

3.2.1 Context 31

3.2.2 Data Collection 32

3.2.3 Demographics 33

3.2.4 Metrics 34

3.2.5 Analysis 36

3.2.6 *Qcheck* 37

3.2.7 Setting Up Activities 38
3.2.8 Summary of Studies 39

4 Generation of Automated Code Quality Improvement Feedback 42
4.1 Context 44
4.2 Measuring Students' Code Quality Through Software Metrics 44
4.2.1 Methods 44
4.2.2 Metrics 45
4.2.3 Data Collection 47
4.2.4 Results and Analysis 48
4.2.5 Discussion 50
4.3 Assessing Students' Code Quality with *qcheck* Support 50
4.3.1 Methods 51
4.3.2 Metrics 51
4.3.3 Data Collection 52
4.3.4 Results and Analysis 53
4.3.5 Qualitative Evaluation 54
4.3.6 Summary and Discussion 58

5 Code Quality Improvement Prompted by Automated Feedback 60
5.1 Context 61
5.2 On the Impact of Code Quality Feedback Generation with *qcheck* 62
5.2.1 Methods 62
5.2.2 Metrics 63
5.2.3 Data Collection 64
5.2.4 Results and Analysis 65
5.3 On the Use of Code Quality Feedback Messages 67
5.3.1 Methods 67
5.3.2 Metrics 68
5.3.3 Data Collection 69
5.3.4 Results and Analysis 69
5.3.5 Qualitative Evaluation 72

5.4 On Producing Summative Feedback 74
5.4.1 Methods 74
5.4.2 Metrics 75
5.4.3 Data Collection 77
5.4.4 Results and Analysis 78
5.4.5 Discussion 78

6 Consequences of Code Quality Improvement Feedback on the Learning of Programming 80
6.1 Context 81
6.2 Evaluation of Providing Code Quality Feedback in a Programming Course . 81
6.2.1 Methods 82
6.2.2 Data Collection 84
6.2.3 Metrics 85
6.2.4 Results and Analysis 85
6.3 Do Learners Think that *Qcheck* is Useful? 92
6.3.1 Methods 92
6.3.2 Participant Selection 93
6.3.3 Data Collection 95
6.3.4 Results and Analysis 95
6.3.5 Discussion 99

7 Discussion 101
7.1 Theoretical Implications 108
7.2 Pedagogical Implications and Opportunities 109
7.2.1 Learning Conversations and Interactions 109
7.2.2 Critical Reflection About Code 109
7.2.3 Clear Marking Criteria to Programming Assignments 110
7.2.4 Summative Assessment of Code Quality Produced by Students . . . 110

8 Concluding Remarks 111
8.1 Future Works 112

A Uma revisão sobre sistemas automáticos para a avaliação de atividades de programação 121

B Qualitative aspects of studentS' programs: Can we make them measurable? 132

C Applying Spectrum-based Fault Localization on Novice's Programs 141

D Questionnaire – Do learners think that *qcheck* is useful? 150

E Avaliação da Legibilidade de Programas Escritos por Alunos Iniciantes 153

List of Figures

1.1 Stages of Computer Programming Problem-Solving Stages. 2

2.1 Students' View of TST Web Interface. 26

3.1 Students Performance During the Longitudinal Study. 33
3.2 Students Course Performance Expectancy. 34

4.1 Distribution of Manual Grades Assigned to Functionally Correct Submissions. 48
4.2 Distribution of Instructors' Grades and Each Metric. 49

5.1 Problem Specification of 'Life Collatz' Programming Assignment. 63
5.2 Number of Quality Warnings per Group. 70
5.3 Warnings Distribution on each Group. 71
5.4 Distribution of ΔW According to Groups. 71
5.5 Quality Improvement According to Groups. 71
5.6 Code Changes Categorization per Group. 73
5.7 Code Production Quality Report Automatically Generated by *Qcheck*. . . . 74
5.8 Assessment of Students (a) and (b) Programming Assignments. 77

6.1 Aggregated Number of *Qcheck* Use by Students by Date. 84
6.2 Occurrences of *Qcheck* Uses X Activities Performed by Students. 86
6.3 Distribution of ΔW per Assignments. 87
6.4 Number of Warnings According to *Qcheck* User Category. 88
6.5 Mean of *W* and *S* According to *Qcheck* Use. 90
6.6 Qualitative Evaluation - Questions of the Interview and Questionnaire. . . . 95
6.7 Students' Perception About Code Improvement Directed by *qcheck* Hints. . 99

7.1 Summary of Costs and Benefits on Providing Assistance. 108

List of Tables

2.1 Pass and Failure Rates of Computer Science Students at the UFCG Introductory Programming Course. 15
2.2 ITS Objectives and Strategies. 23
2.3 Conceptual Components of ITS. 24

3.1 Demographic Data of Students of 2017.1 Programming 1 Course. 34

4.1 Measurements Proposed to Assess Code Quality. 47
4.2 Pairwise Code Quality Evaluations Among Raters X Tool. 52
4.3 Agreement Index Value Among Raters and *qcheck* Tool. 54
4.4 Disagreement Among Ratings (Rn and *qcheck*). 55

5.1 Number of Correct Submissions. 64
5.2 Distribution of Quality Warnings Account According to Each Group 69
5.3 Dataset Summary. 77
5.4 Contingency Table Contrasting Students' *qcheck* Usage Proficiency to Grades. 78

6.1 Students Code Production Data Collection. 85
6.2 Students Usage Pattern of *Qcheck* Relating to Occurrences and Activities. . 87
6.3 Contrasts of Final Number of Warnings According to *qcheck* Usage per Assignment. 88
6.4 Value of the Mean of Quality Warnings – $\mathbf{W}$ 90
6.5 Values of the Mean of Style Warnings – $\boldsymbol{S}$. 90
6.6 Data Set of Qualitative Users' Study. 96

Chapter 1

Introduction

A central activity in programming courses is the development of students' programming skills with practical programming assignments. Enough practical activities are paramount to students to effectively achieve learning goals. The assessment of these activities and the feedback provided by instructors about them is a fundamental aspect of the learning process. Besides showing that learning outcomes are being met; literature has shown that feedback can affect the learning process at various levels and have different functions, such as: stimulating, informing, correcting, making suggestions, completing knowledge, advising and so on [Nar08].

In fact, one of the central pillars of the interaction between instructor and learner is the feedback provided by the first about the work produced by the last [Yai14]. In his essay, Yair argues that grades offer the obvious and tangible type of feedback, they are merely a 'right-wrong' indication and have a rather limited benefit to students. Useful feedback goes beyond right-wrong or pass-fail information. Students need to be aware of how they are performing regarding the instructors' expectations and how they can improve. This type of feedback is usually referred in educational research as "formative feedback". Shute explains that formative feedback "can signal a gap between a current level of performance and some desired level of performance or goal" [Shu08] .

In 2005, Bennedsen and Caspersen discussed the idea of "revealing the programming process" as fundamental to teach novice programming students [BC05]. According to them, the idea of perceiving the development of computer programs as "programming process" is not new. The process encompasses a set of activities that are interactively executed and can

be revisited, differently from a linear process.

In a similar perspective, Polya's methodology on how to solve mathematical problems [Pól57] was mapped and adapted to programming teaching and learning scenario. In fact, learning how to program goes beyond than acquiring abilities on language syntax or managing a development environment. It requires analytical skills that must be trained. We believe that the development and strengthening of problem-solving skills may help to learners cross the chasm between understanding the problem specification and programming an adequate solution. Figure 1.1 summarizes the programming problem-solving stages, adapted from Polya's methodology [Tho97].

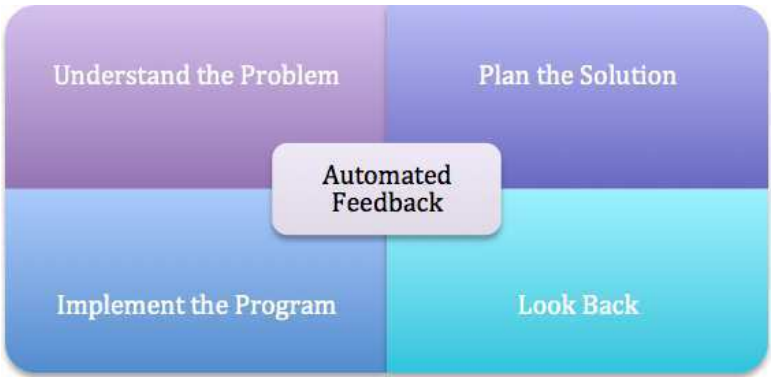

Figure 1.1: Stages of Computer Programming Problem-Solving Stages.

In this work, we use Thompson (1997) definition of the programming process, which encompasses a set of steps needed to solve a computer-programming problem. These steps are: (1) Understand the problem, (2) Plan the solution, (3) Implement the program and (4) Look back. This process can be seen as a natural and effective pathway for students accomplish their programming assignments. However, the whole process can be too complex to novices whilst struggling with their first programming experiences, increasing their need for guidance and assistance.

As the number of enrollments in programming courses is steadily growing and teachers' duties go far beyond teacher-student time, one-to-one personalized feedback about programming assignments is rare. In this context, Automated Assessment Systems (AAS) play an important role as they allow for rapid, frequent, cheap and standardized feedback. Furthermore, data acquired about students' interaction with the course instructional materials by these systems, increase teachers' ability to track, map and assess students learning

patterns [Yai14]. They empower and aid instructors to direct their efforts to higher analysis levels.

In almost every programming course nowadays, AAS provides delivering, submission and assessment of programming assignments. Those systems give to students different levels of feedback about their assignments, depending on the AAS specific strategy. Several strategies to automatically assess programs have been adopted by those systems [AM05]. Most of them provide feedback on functional correctness test-based analysis.

Nowadays, AAS have extended their scope and include features such as: gamification [IE14], test coverage analysis [JU97], managing human-authored feedback, contest adjudication [Mil11], secure remote code execution [MM13], and more [DSPQ+17].

A positive aspect highlighted by Gulwani (2014) about AAS, is that they provide immediate automated feedback and can enable new pedagogical benefits such as allowing resubmission opportunities to students who have submitted imperfect solutions and providing an immediate diagnosis for teachers on class performance, allowing them to adapt instruction accordingly [GRZ14]. On the other hand, students still need a better support from them to deal with the complexity of programming process. It is not rare that students feel helpless, as they cannot make progress in their programming assignments autonomously. Sometimes, they are only able to move on after a personal interaction with instructors or teachers assistants. The problem is that many students do not receive this qualified feedback because they avoid human contact or simply do not have a chance to make it on time.

In a broader sense, this work is in the context of computer supported learning tools aimed at helping to teach and learn how to program. It explores automated assessment as a mean to provide personalized instruction to students about their programming assignments in introductory programming courses. Given that, it relies on fully automated strategies and can be applied to distance learning or online courses such as MOOCs and others.

1.1 General Problem and Proposed Solution

The problem is that the feedback provided by automated assessment systems is focused mainly on the "Implement the Program" phase of the programming process, which comprises the production of a functional correct program according to a set of tests. The "Look Back"

phase, when the problem solution is evaluated, is usually neglected. Students need a better support from AAS to deal with the complexity of this phase and its particular difficulties.

We claim that timely automated feedback can be generated and delivered to students to satisfy this need. We intend to generate rich feedback, which is useful formative feedback typically provided by human instructors. We argue that there are aspects of program validation feedback, in terms of code quality, that can be automated.

Our proposal is to provide feedback to the last phase of the programming process, typically neglected by usual AAS. We intend to obtain information at a low cost, using instructional materials already produced by instructors to the programming assignment. The feedback will be timely delivered to the students by an AAS.

We claim that it is possible to assist novice programmers with adequate and useful feedback, in order to improve programming problem-solving support, increase student programming skills with the aim to leverage introductory programming learning.

1.2 Overview of this Thesis

During this doctoral research, we explored the generation and delivery of automated feedback to students during the programming process. The challenge was to provide timely and enriched feedback that stimulates students to reason about problems and their solution and to improve programming skills. We intended to leverage programming problem-solving teaching and learning generating enriched automated feedback, regarding students programming assignments, with information typically delivered by human instructors. Furthermore, we constrained our strategies of feedback generation to obtain information at a low cost from instructional materials already produced by teachers, aiming to minimize burdens imposed to them.

We focus deeply on providing feedback with respect to the last phase (4 - Look Back), when the program is revisited and refactored. Usually, when the program passes all tests cases and is considered functionally correct, students move on to a new assignment. In fact, we have observed in our empirical studies that the vast majority of students neglect this fourth phase, as they do not make new submissions after the first correct one. In other situations, when the program is manually evaluated, they are assessed under qualitative and

subjective factors considered by instructors. However, manual and personalized feedback is produced at a high cost and, depending on the number of students and assignments, may be prohibitive. We dedicated our efforts on generating automated feedback on code quality for novice's programming assignments.

In an effort to achieve this purpose we conducted research studies, such as experiments, surveys and case studies, to gather empirical evidence to answer the following research questions:

> RQ1: How can we generate automated code quality feedback based on introductory programming teachers' expectations?
>
> RQ2: Can students improve the code of their programming assignments prompted by timely and automated feedback?
>
> RQ3: Is it possible to improve students programming skills stimulating reflection about their code quality?

We proposed and evaluated a set of software metrics that could be used to provide qualitative feedback about novice programming assignments. We proposed and developed an automated feedback tool, which was plugged into an AAS and evaluated its efficacy. Finally, we evaluated in a real introductory programming course its effects during a period of time and found positive results. Although, there is room for adjusts and customizations so that we could improve the approach.

1.3 Contributions

In summary, this Ph.D. research proposes and evaluates strategies to improve automated feedback provided by AAS to support computer-programming problem solving of introductory programming learners. Students lack elaborated feedback, typically provided by humans, in key parts of this process. The main contribution of this Ph.D. thesis relies on the automated generation of code quality feedback, targeted to aid students on the fourth stage, known as "Look back", of the programming problem-solving process.

We proposed a set of metrics that are able to capture, in some extent, teachers notion of students' code quality; then we implemented and evaluated in a real programming course

a proof-of-concept feedback tool plugged into their AAS. Results found in our longitudinal evaluation goes beyond what we initially expected: the improved assignments' code quality. We observed that students felt stimulated, and in fact, improved their programming abilities driven by the exercise of reasoning about their already functioning solution. To sum up, the contributions we aim to deliver with this work are:

1. A proposal to provide feedback generation, based on software measures, to aid students in improving their code on instructors' code quality perspective;
2. A proof-of-concept tool – *qcheck* – built to refine and evaluate the proposal. It is publicly available to use;
3. A set of lessons learned on providing automated feedback related to program quality improvement through an automated assessment tool in an introductory programming course.

1.4 Thesis' Outline

This document is organized as follows:

Chapter 2 – Background: This chapter presents the background in computer science education that motivated us to pursue this research and the theoretical framework relating to concepts we used to develop this work. Experienced readers may safely skip it.

Chapter 3 – Feedback Generation to Code Quality Improvement: This chapter details our proposal and strategies on providing automated feedback about code quality. It summarizes the roadmap of this research and discusses the methodology we followed to construct knowledge and evaluate our claims.

Chapter 4 – Generation of Automated Code Quality Improvement Feedback: This chapter details our proposal and strategies on providing automated feedback about code quality. It shows our proposal on how to generate automated code quality feedback.

Chapter 5 – Code Quality Improvement Prompted by Automated Feedback: This chapter discuss the possibility of the code quality improvement feedback delivered to students directs the improvement of their programming assignments' code.

Chapter 6 – Consequences of Code Quality Improvement Feedback on the Learning of Programming: Lastly, this chapter considers the consequences to learners of providing feedback about code quality improvement using the proposed tools during a programming course. We are going to present our main findings in a longitudinal study and discuss its implications to learners, instructors and the course itself. Furthermore, we are going to present an evaluation of the approach performed with the students that used the tool in their activities.

Chapter 7 – Discussion: This chapter sums up the discussion about the proposal and ideas we brought to light in this doctoral research. We will observe the practical significance of those approaches and pedagogical implication arisen by them. In addition, we will briefly contrast it with other related works and present some threats to our conclusions validity.

Chapter 8 – Conclusions: This chapter will concisely wrap up this work emphasizing what was done and why it was good. Furthermore, it will address some opportunities and future works that might be done to extend and improve this research.

Chapter 2

Background

Multi-national and multi-institutional studies on Computer Science Education (CSE) literature showed that teaching programming to novices is a worldwide challenge [LAF+04][MM13] [MAD+01]. Many researchers and academics discuss alternatives to better teaching and learning programming skills [SS89] [RRR03]. Others examine characteristics and difficulties of novice students [ESPQ+09] [LAMJ05]. Different approaches have been proposed to minimize the hurdle imposed to learners, each one with their own advantages and drawbacks.

In this thesis, we argue that formative feedback has the potential to leverage programming learning in helping students to improve their code quality while reasoning about another potential solutions. Hattie and Timperley (2007) published a thorough review of the potential of feedback on learning and achievement. They propose that effective feedback must answer three major questions asked by students to her teacher/instructor: "Where am I going? (What are the goals?), How am I going? (What progress is being made toward the goal?), and Where to next? (What activities need to be undertaken to make better progress?" [HT07]. However, there are some issues to consider when generating and delivering feedback, for example, type of feedback, characteristics of the learner that will receive it, timing (delayed or immediate). Shute (2008) presented a broad literature review and summarized recommendations and guidelines for formative feedback. Her study presents suggestions about "what to do" and "what to avoid" when delivering formative feedback [Shu08] . She also discusses formative feedback timing and issues regarding learners' characteristics. Technology that supports programming learning challenges and opportunities or grader systems, has the capability

to empower the generation of adequate and timely feedback, bearing in mind the learner characteristics.

Research and development of this kind of systems are not new as reviewed by [AM05] [IAKS10]. In a comprehensive survey, Douce discussed the development of those systems for the last forty years [DLO05]. Following a chronological approach, he classified the systems into generations. Recently, communications technologies have pushed the frontier of e-learning forward with the advent of massive open online courses. So, systems that support programming learning have become even more complex and important.

In this chapter, we present an overview of CSE literature addressing the subjects in this thesis and aiming to make this document self-contained. Firstly, we discuss aspects of cognition and learning. Then, we present some programming learning challenges, based on the literature and in our practice as teachers. We also identify opportunities to contribute and advance the research in this field. Next, we examine the potential of feedback to programming learning. For clarification, we define feedback related terms that will be used in the rest of this document. Lastly, we provide an overview of the existing assessment system for introductory programming learning and intelligent tutoring systems, which are works related to ours. We discuss different strategies employed by those systems to support learning. We devoted special attention to an AAS developed in-house to our university introductory programming course - TST. This system has been used for, at least, five years and it is in constant evolution. We used data collected by TST in our studies.

2.1 Considerations About Learning and the Science of Instruction

Learning theory is a vast research area that congregates psychologists, neuroscientists, educators and other professionals. Numerous and important specialists contributed to constructing knowledge about the science of learning and its challenges, such as: Jean Piaget (1896-1980), Lev Vygotsky (1896-1934), Benjamin Bloom (1913-1999), Seymour Papert (1929- 2016), Richard Mayer (1947-) and so on. Due to their ideas, they are usually cited in researches and works that ultimate intend to improve learning.

According to cognitive sciences, human learning and working activities rely on two types

of memory: working memory and long-term memory. When people are in learning mode, the new information acquired from the environment is processed in working memory to form knowledge structures that are stored in long-term memory. When new information enters in working memory they must be integrated into pre-existing structures of the long-term memory [Cas07].

> "Learning depends on the learner's cognitive processing during learning and includes (a) selecting - attending to the relevant incoming material; (b) organizing - organizing the incoming material into coherent mental representation; and (c) integrating - relating the incoming material with existing knowledge from long-term memory" [May08].

In this work, we propose to provide elaborated feedback during the programming process to improve students' learning. We searched for the theoretical basis on the "Science of Instruction" so that we could guide our work. We shall refrain from discussing in more details this theory here, as it is not our intention to provide a comprehensive survey about the theme in this session. In summary, the key elements of the science of instruction, which are:

1. Reducing extraneous processing - cognitive processing that does not support the instructional goal and is attributable to confusing instructional design;

2. Managing essential processing - cognitive processing needed to mentally represent the incoming material and that is attributable to the complexity of the material;

3. Fostering generative processing - cognitive processing aimed at making sense of the incoming material, including organizing and integrating it with prior knowledge.

Mayer (2008) presented and discussed a set of principles to elaborate and design multimedia instruction so that it could achieve the above-mentioned principles [May08]. We also tried to adhere to some of these principles when proposing and implementing our proof-of-concept tool for code quality improvement feedback – *qcheck*. Our proposal in providing feedback on code quality improvement seeks the alignment with his theory and cognitive load reducing principles. These principles were targeted on the "Design of Multimedia Instruction" [May08], but we consider that it can also be applied to our context of programming instructional material.

We attempted to "reduce extraneous processing" using the Coherence principle. Reducing extraneous processing is important so that learners do not waste cognitive capacity in activities that do not contribute to their final goals. To follow the coherence principle, implemented a plugin to the already existing AAS - TST - used by the introductory programming course.*Qcheck* installation and usage were extremely similar to the way students are used to doing with TST. In fact, we tried to adhere *qcheck* to TST as much as we could, but they are independent software. In this sense, students did not have to learn a new instruction on how to use it, but only how to make use of it.

Then, we followed the principle of the Personalization, in order to foster generative processing. On the first experiments and versions of *qcheck* software the textual style of the feedback messages were too impersonal (example: "There are too many lines of code"). This principle claims: "People learn better from a lesson when words are in conversational style rather than in formal style." [May08]. The theoretical rationale of this technique is that the conversational style induces learners to create a sense of partnership with the message narrator, so they will try harder to make sense of what is being advised. According to this principle, we updated all feedback messages to conversational style (example: "Your program has too many lines of code."). We did not have empirical evidence about the effect size of this change in our research.

2.2 Programming Learning Challenges

Programming is considered to be a central and a distinguished feature of Computer Science curriculum [Fin99]. Programming learning in higher education challenges researchers and educators with different issues experienced worldwide. Lahtinen and colleagues, in a study about the difficulties of novice programmers (2005), argued:

> "Programming is not an easy subject to be studied. It requires the correct understanding of abstract concepts. Many students have learning problems due to the nature of the subject. In addition, there are often not enough of resources and students suffer from a lack of personal instruction. Also, the student groups are large and heterogeneous and thus it is difficult to design the instruction so that it would be beneficial for everyone. This often leads to high drop-out rates

on programming courses" [LAMJ05].

In this subsection, we are going deep into this claim resembling discussions about those issues reported in the literature and presenting our own personal experience as an instructor. At the end of this section, we will discuss the results of Lahtinen studies and debate if it still remains valid 10 years later.

Programming encompasses abstract concepts and problem-solving skills that usually frighten students that have just started a university course. Many students that cannot make progress on introductory programming course simply consider dropping out the course, as they feel inadequate to it. Difficulties in this subject may mine self-esteem and deeply affect psychologically students. High dropout rates are considered to be an important issue in computer science education and are referred as motivation by many studies [VAW14] [Yad11]. Another issue referred by educators is that "students suffer from a lack of personal instruction" [LAMJ05].

Teachers' assistants and tutors are other critical resources to practical programming classes. They support learners with their assignments and, sometimes, assess their code or test production providing them with feedback on how to make it better. However, it is expected that assistance and assessment provided by them, follow the same instructors' criteria and orientations. For example, the course instructors explained that there are situations to better use 'While True' statement. This same orientation must be replicated in laboratory classes. It may be difficult to standardize teachers' assistants and tutors procedures since they have their own judgments and limitations. One-to-one tutoring provided by them is simply prohibitive for the number of students enrolled in some courses, mainly in online courses [SGSL13]

A different approach that some courses are adopting to deal with the challenge of providing feedback about programming assignments to a large number of students is peer-review. In this case, other "peers" review the student code and generate a feedback with suggestions on how to improve the code or even grading it. But this approach has its own problems, besides standardization mentioned before. It has been reported that students may wait for a long time to get any feedback [SGSL13].

In a different perspective, there are controversial debates about what are the most effective or adequate methodology and teaching approach to teach programming. As

programming issues were considered one of the top grand challenges in Computer Science education, McGettrick and colleagues stated the following, as research objectives in this area:

> "(...) Teach using the right methods by choosing between different approaches: for example, those based on formal definitions of syntax and semantics and those relying on informal description and example; between conventional lectures and practical classes and e-learning, collaborative learning, peer tutoring and other approaches, and using the right assessment and evaluation strategies" [MBI+05].

Fincher, more than a decade before, comparatively evaluated methodologies and models for teaching programming. She presented four approaches that used to be adopted by different instructors, named and classified them regarding their degree of abstraction: literacy, computation as interaction, problem solving and syntax-free [Fin99]. As a conclusion, the study argues that practitioners need to be reflective and to know the possible approaches in order to adhere to them. It is also pointed that there is no quantitative evidence of the success of any of those teaching approaches and this is a challenge for researchers in computer science education.

Years later, Vihavainen and colleagues systematically reviewed the literature in teaching approaches for programming in order to measure the effects of each proposed intervention and, finally, to yield quantitative evidence of success [VAW14] [VPL11]. They evaluated the study of thirteen approaches, or teaching interventions, using pass rates as a success metric. The interventions were grouped on activities tagged as:

- collaboration;
- content change;
- contextualization;
- CS0 (it means a preliminary course, before the introductory programming course);
- game-theme;
- grading schema;

- group work;
- media computation;
- peer support;
- support activities (such as tutors, more teaching hours, etc);

They found that the novel approach improved pass rates, on average, in one third in comparison with the traditional method previously adopted. Vihavainen et al. assured that it was not possible to choose the most effective approach. They suggested that perhaps just the willingness of educators to change and their move from a traditional way of teaching to a new one will be responsible for improvements in pass rates.

In fact, low pass rates or high failure rates in introductory programming courses have been used as motivations of hundreds of studies [BC07]. Still, there are few studies devoted to quantitatively evaluate what is considered to be this high failure rate in introductory programming courses worldwide. Bennedsen and Caspersen, in a first attempt of producing evidence about this, conducted a survey among different institutions about failure rates [BC07]. The study results found on average 33% of failure to the number of enrollments. They concluded that this number is not "alarmingly high", but they cannot make firm conclusions about this since the number of respondents of their survey was considerably low.

In a worthy initiative, Watson (2014) revisited Bennedsen (2007) study animated by the same aim: to find "substantial evidence" of introductory programming courses failure-rates [WL14] [BC07]. This study was awarded the best paper at the 19th edition of the annual conference on Innovation and Technology in Computer Science Education - ITiCSE/ACM. They systematic reviewed the literature and performed statistical analysis to find the average of introductory programming course failure-rate worldwide. Their study sample size was a double of the previous work sample size. However, it is still statistically not sufficient to make firm global conclusions. The authors were very cautious when stating their conclusions. Interestingly, the average failure rate found was 32.3%. It was almost the same as the one obtained by Bennedsen and Caspersen survey. The study also concluded, just like their predecessors, that this number is not "alarmingly high", but it has a "considerable potential for improvement" [WL14].

The reality of our introductory programming courses at UFCG is not as different as the worldwide scenario as we can observe in Table 2.1. It is worthy to observe that on the term 2014.2 there was a drastic change in pedagogical direction of that course. Instructors have adopted flipped classroom with mastery learning strategy and this change might have caused the improvement of pass/fail numbers.

Motivated by this possibility of improvement and by the need to teach programming more effectively there is a great academic effort in this area. Mcgettick et al. claims "when we set out to teach programming skills to students, we are less successful than we need to be and ought to be." Also, "we might teach programming more effectively, making better use of resources and with greater student and staff satisfaction." [MBI+05].

Table 2.1: Pass and Failure Rates of Computer Science Students at the UFCG Introductory Programming Course.

Term	**Passed** (%)	**Failed** (%)
2011.1	69.0	31.0
2011.2	51.0	49.0
2012.1	77.0	23.0
2012.2	54.0	46.0
2013.1	72.0	28.0
2013.2	63.4	36.6
2014.1	70.6	29.4
2014.2	84.5	15.5

Many educators fiercely study to better understand the programming process and how the novice programmer comes to understand and major this cognitive ability [SS89] [RRR03]. In general, they report that students have greater difficulties in understanding the "big picture" of the programming process, such as abstraction or how to solve the problem programmatically, than details about it, such as programming language syntax.

Lahtinen and colleagues deep dived into novice programmers' universe to find out, in minor details, what were their struggles. Specifically, they aimed at contrasting if programming courses that are adopting Java/C++ corroborate with difficulties reported in

previous studies. They conducted a multi-institutional survey answered by more then 500 students, that perceived as the most difficult issues in programming were [LAMJ05]: "understanding how to design a program to solve a certain task; dividing functionality into procedures and finding bugs from their own programs." The findings corroborates with previous studies: the biggest issue educators need to deal with is to help students to master programming abstract concepts. Furthermore, they find that students and teachers have a different perception about content understanding. Students tend to overestimate their understanding about the subject. On the other hand, teachers have a more realistic view of students' difficulties as they assess their exams [LAMJ05].

In fact, students need to be aware, through instructor's feedback, about how they are performing in a particular task and, certainly, in the whole course. This knowledge may drive their attitudes about giving up or keeping on trying to succeed. Shute declares that feedback can reduce the uncertainty about how well the student is performing, as it closes the gap between learners understanding and, the desired understanding [Shu08]. We also agree with this claim and recognized the opportunity to improve the support to students through providing formative feedback along the programming process to let them achieve the expected learning outcomes.

The challenge we address is narrowing the student's self-referential assessment of their knowledge, code production and expectations with the teacher's assessment about them. In fact, teachers have their own expectations about what is supposed to be mastered in a given moment in the course, according to the exercises or lectures students have been exposed to. These expectations are the so-called learning outcomes. Formative feedback seems to be the key to uncover to the student the teachers' expectancies, in a giving moment of the programming process, about his/her code production.

2.3 Feedback and its Effects on Learning

There are numerous definitions about educational feedback in the literature. In this document, we adhere to Hattie and Timperley conceptualization that conceive "feedback as the information provided by an agent (e.g. teacher, peer, book, etc) regarding aspects of one's performance or understanding" [HT07]. Though, feedback is a consequence of a

process started before. For example, a teacher proposes a set of programming assignments to students. As a result, students produce programs as responses to those assignments. It is possible to provide feedback about the product, the code itself, and also to students' attitudes towards the programming process.

In educational research, feedback 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 to declare that learning objectives have been reached by the student [DLO05]. In a simplistic analysis, we can say that formative feedback is addressed to students and teachers and drives improvements on their teaching and learning activities along the process. Summative feedback is provided at the end of a cycle, aiming to measure the student growth, for example, students' grades or progress reports. This kind of feedback is important not only to students but also to teachers and educational institutions, in order to re-configure courses or curricula.

In fact, formative feedback provided by instructors about the work produced by learners is considered to be one of the central pillars of the interaction between them [GMD11] [Yai14]. In his essay, Yair argues, "Grades offer an obvious and tangible type of feedback (...) and have a rather limited benefit to students". Effective formative feedback assesses the learners' work and is composed of, at least, two components: verification and elaboration [Shu08].

The most frequent strategy to perform verification in programming assignments is the test-based assessment. A set of test cases produced by instructors is used to dynamically verify students' programs. On the other hand, feedback elaboration on programming assignments has a lot of variations. It can address different topics, such: test coverage, when the programs were delivered (is it on time or delayed?), discuss code quality, guide students to further studies on a given topic, propose instructional material to improve understanding, etc. Formative feedback goes beyond "right-wrong" indication about the student's task. In the context of our work, the formative feedback includes all the information and communication exchanged by learners and instructors that may contribute to modify an erroneous behavior and to demonstrate that expected abilities have been mastered. In fact, feedback closes the gap between learners' current understanding and the desired goal, according to instructors [Shu08]. Feedback has the potential to enhance learners'

performance, but when it effectively does?

There are several characteristics that must be observed to generate useful feedback. In regards to timing, it can be classified as delayed or immediate feedback. For example, in programming assignments, automated assessment systems can provide immediate test-based feedback about each submission of the student's code. It allows for students to instantly discover if the submitted code meets the requirements expected to solve that given problem. This automation can fasten studies sessions since students do not need to wait for an instructor to assess the code. However, researchers are reconsidering this kind of immediate feedback arguing that it may refrain students to think critically and thoroughly test their code before submitting it [BE14]. This practice has possibly changed the student behavior on the programming process, as they discourage the testing phase. Students can rely on the instructor's tests, which are executed when the program is submitted. Petit and colleagues proposed an alternative to mitigate this issue delaying the feedback. They investigated a throttling dynamic to accept code submissions, restricting students to submit only 3 times in a period of 15 minutes [PHG+15]. In doing such, students are forced to submit a more mature version of the program; delaying the feedback that students would receive. Delayed feedback might be seen as positive as it provides the opportunity for reflection when a task is difficult or it involves a deeper degree of processing [HT07].

In Narciss (2008) study, she discusses the content of the feedback for iterative learning tasks [Nar08]. She presents a content-related classification that provides a "structured overview of simple and elaborated feedback components by organizing the components with regard to which aspect of the instructional context is addressed". Simple feedback messages can be categorized according to their components as:

- **KP – Knowledge of performance**: Provide learners with a summative feedback (e.g., percentage of correctly solved tasks, number of errors, grade)
- **KR – Knowledge of result/response**: Provide learners with information on the correctness or quality of their actual response or outcome (e.g. correct/incorrect, flagging errors, good job)
- **KCR – Knowledge of correct results**: Provide the correct response or a sample solution to a given task.

Elaborated feedback messages can be categorized according to their components as:

- **KTC – Knowledge about task constraints**: Provide information on task rules, task constraints and/or task requirements.
- **KM – Knowledge about mistakes**: Provide information on errors or mistakes (e.g. correct/incorrect, flagging errors, good job).
- **KC – Knowledge about concepts**: Address conceptual knowledge by providing for example response hints on concept attributes or attribute-isolation examples.
- **KMC – Knowledge about metacognition**: Address and elicits meta-cognitive knowledge and strategies necessary for self-regulating the learning process (e.g., topic-contingent hints about useful sources of information).

In a similar perspective, other researchers characterize the feedback message in its complexity. It refers to how much and what information feedback message must contain. One can think that the more specific the feedback message, the better. However, a more specific feedback that includes long texts can dilute the message that instructors want to communicate. Lengthy or complicated explanations may be useless, as students will not read them [Shu08]. In a recent study, Denny and colleagues proposed to enhance the compiler errors messages in order to the user identify the error line and correct their code easily. They included concrete examples illustrating the error that occurred and how to correct that kind of error in a given situation. Surprisingly, the evaluation of this approach revealed that there was no effect on students' ability to correct their code errors [DSPQ+17] [DLRC14]. This study illustrates that there must exist a balance between what is important and what is necessary to exist in a feedback message to be useful.

Finally, another important perspective in order to evaluate feedback effectiveness is related to the learner characteristics. There are many studies in feedback research proposing to craft a different feedback message according to the learners' characteristics. Personalized feedback, the core of an adaptative/personalized learning environment, seems to be a prominent area in computer-based education research.

Another studies, claims that there are gender differences in feedback consuming: boys benefit less than girls from feedback [NSS+14] [TWV15]. Meaning that girls are more

enticed than boys to modify their erroneous behavior in a given task, according to feedback guidance and to improve their performance on it. However, we cannot assure that this behavior would be reproduced in programming assignments. To the best of our knowledge, none of those studies were replicated on introductory programming courses.

In closing, we claim that formative feedback potentially has the power to enhance programming learning and teaching, but we still need more evidences to elect what are the most effective approaches to do so. We know that feedback messages need to be objective and clear. It must provide enough information to push the learner further on his or her current understanding. It must be timely and adaptive, according to learners' characteristics. Certainly, it must be provided automatically.

2.4 Related Works

In this section, we are going to present some computer-based learning environments which are in line with our proposal in this research. They provide students with valuable feedback on the learning process. Initially, we discuss about automated assessment systems – AAS – in the context of programming education. These are works related to ours in the sense that the strategies we intend to use to deliver the generated feedback must be associated with an AAS. Next, we summarize some aspects of intelligent tutoring systems. These systems "are typically found toward the high end of the interactive spectrum" [KA07] of computer-based learning environments. The discussion about feedback provided by those systems was important to our work. Finally, we present the AAS adopted by Programming 1 course at UFCG. This system was used to help us in producing and gathering data about students interaction with programming assignments during the course. Furthermore, it is important to present TST to clearly delineate the scope of our research and existing related works.

2.4.1 Automated Assessment in Programming Education

There exist many available automated assessment systems (AAS) focusing on introductory programming assignments, such as *WEB-Cat* [Edw03], *Mooshak* [RSKPFVI14], *Marmoset* [SHP+06], *BlueJ* [Jad05], among others. AAS became widely used in programming courses.

However, they are not exactly a new approach, as the first appeared in 1960. Thenceforth, they promise to produce objective and consistent feedback to students, while mitigate the heavy workload of the instructors when performing manual assessment [DLO05] [AM05] [IAKS10].

They are indeed fundamental to provide feedback to students and help instructors to deal with modern challenges of programming courses such as increasing enrollments number, e-learning, and MOOCs. Nevertheless, in this myriad of systems, it is worth to evaluate what kind of information they provide on feedback and if students are effectively using it to improve their programming practice.

In general, AAS employ comparable approaches and provide similar features [IAKS10]. The most common feature of automated assessment systems is code functional correctness evaluation. A typical system executes a set of test cases, provided by the instructors, and compares the expected output to the observed, obtained from students' programs. Another feature provided by AAS is grading [CAMF$^+$03] [Nor07]. Grader systems may weigh other factors, besides correctness, such as deadline penalty, resubmission times, type of errors, test coverage, etc.

We claim that the richest possible feedback on students' programs is the result of human inspection and analysis of both functional and qualitative aspects of the code. Good instructors enrich their feedback with impressions about the code quality to help students to reason on their solution and leverage their critical abilities. Certainly, many subjective aspects are indeed immeasurable, but we think that it is possible to find objective and measurable factors on the code that reflect most of the so-called qualitative aspects reported on the feedback provided by instructors in their assessments. This is part of our proposal and will be explained afterward in this document. Appendix A presents a review on AAS in a paper we submitted for publication at SBIE - Simpósio Brasileiro de Informática e Educação: "Uma revisão sobre sistemas automáticos para a avaliação de atividades de programação".

2.4.2 Intelligent Tutoring Systems

Intelligent Tutoring Systems (ITSs) are computers' programs intended to provide personalized instruction and feedback to learners usually without requiring human teachers intervention. These systems were born into Artificial Intelligence laboratories, evolved and

spanned including knowledge from different areas such as cognitive sciences, psychology and others. Usually, their development is highly grounded in human learning theories or models. Even though, "no single teaching environment has been shown to be appropriate for a majority of people or even a majority of domains, in part because human learning is imperfectly understood" [Woo10].

The technological approach adopted by ITS potentially may produce highly individualized, pedagogically effective, and accessible instructional material and to involve more students in effective learning. They might unveil the extent to which students of different gender, cognitive abilities, and learning styles learn with different forms of teaching, given the capability of being sensitive to learners differences [Van06].

There are many projects which endeavor is to build and distribute an ITS for a given knowledge domain. Also, it can be found in literature meta-analysis contrasting ITS initiatives learning outcomes, suitability and effect sizes [KF16] [MANL14]. These works first challenge is to come up with a common definition about such diverse systems. In general, an ITS is a student-centered computer program whose objectives and strategies are presented in Table 2.2 [MANL14].

Typically, an ITS targets to achieve the benefits of one-to-one tutoring [Blo84], in contexts where students would otherwise have access to one-to-many instruction from a single teacher (e.g., classroom lectures), or no teacher at all (e.g., online homework) [Van06].

> "(...) this age of rapidly changing technology and Internet support of meaningful interactions, intelligent tutors have the potential to provide a skilled teacher, or community of teachers, for every student, anywhere, at any moment" [Woo10].

Furthermore, learners can benefit from impartiality, flexibility and standardized quality of instruction from intelligent tutors. Also, they can evolve at their own pace, in order to construct their own knowledge.

Achieving these benefits are possible given the design of the conceptual components of an ITS. They are composed of rich and dynamic models which are: the domain model that contains what is being taught, the student model that may contain common learners' conceptions and misconceptions and the tutor model that represents the instructional strategy adopted by the system. The following Table 2.3 describes the conceptual model of an ITS

Table 2.2: ITS Objectives and Strategies.

Objectives? Performs tutoring functions by (a) presenting information to be learned, (b) asking questions or assigning learning tasks, (c) providing feedback or hints, (d) answering questions posed by students, or (e) offering prompts to provoke cognitive, motivational or metacognitive change
Strategies? By computing inferences from student responses constructs either apersistent multidimensional model of the student's psychological states (such as subject matter knowledge, learning strategies, motivations, or emotions) or locates the student's current psychological state in a multidimensional domain model
How? Uses the student modeling functions identified using those Strategies to adapt one or more of the tutoring functions identified in Objectives.

[MANL14].

Table 2.3: Conceptual Components of ITS.

Interface Specially tailored for the ITS purpose. Through the system interface the learner communicates presenting and receiving information. Often constrained to the subject domain (e.g., algebra), the interface determines the moves the learner can make in solving problems, seeking information or responding to questions.
Domain model Represents the knowledge the student is intended to learn. The model is a set of logical propositions, production rules, natural language statements, or any suitable knowledge representation format.
Student model Represents relevant aspects of the student's knowledge determined by the student's responses to questions or other interactions with the system it also represents common misconceptions or other faults in the student's knowledge.
Tutor model Represents instructional strategies such as feedback and content elaboration and delivering. Example: When to offer a hint to a learner that is unable to generate a correct response or assign problems that requires knowledge only slightly beyond the current student model.

All things considered, we can see that ITSs are an approach that can bring great benefits to computer aided education and e-learning. There exist advances in different research lines, motivated by ITS development and evolution, which can be used in other computer science areas. However, it seems that these systems are conceptually more complex that what, in fact, has been delivered, since the challenges imposed nowadays to on-line education are huge. As an example of these challenges, Baker (2016) argues that will became essential that ITS developers build models that are robust to instructor dynamic behavior and that change as their context of application changes [Bak16].

Even though our proposal in this work aims to deliver personalized feedback in learning activities as ITS do, it presents great differences in scope. We intended to present an approach

that can be used by existing automated assessment systems of introductory programming assignments. In this sense, as a conceptual requirement, we want to provide a low-cost solution to instructors (and AAS developers) so they do not have to create more artifacts than they usually do when proposing a new assignment (i.e. usually, the problem specification, a set of tests and a reference solution) nor to learn a different language or model to implement it. Our proposal is lightweight in comparison with a whole ITS. However, aspects such as feedback and hints delivering, vastly studied in AIED (Artificial Intelligence in Education), literature must certainly be considered.

2.4.3 TST – Programming Assignments Testing System

In this subsection, we will briefly present TST, the automated programming assignment testing system adopted by Programming 1 course. This system was used as a test-bed for our proposals in this thesis. For this motive, it is worthwhile to know its scope in order to recognize the boundaries of our work. TST has been developed and evolved by Dalton Serey, Programming 1 faculty at UFCG, with valuable collaboration of other colleagues, instructors of the same course; especially Jorge Abrantes, pioneer on the present course configuration.

TST was tailored to support the Programming 1 course programming assignments and other activities from students and instructors perspective. Instructors use TST system to produce, test and publish instructional materials such as programming assignments, quizzes and laboratory scripts. They also create, configure and control programming practical activities, such as exams and marathons. Students use TST to have access to their assignments and reports about their course performance. TST provides a personalized experience for students, as they can only have access to activities according to their performance on the exams (regarding the content unit). Besides, the list of activities is randomly chosen for each student.

TST is a cloud application whose backend is built on Google App engine platform. This server is accessed, through REST API, from a web application – tst-online, as seen in Figure 2.1, and command line clients. There are also TST workers aimed at executing assignments' tests that run at another local cloud facility at UFCG. They were created to cope with the server access overloading during exams. Authentication and authorization issues are dealt using students institutional email provided by Google accounts service. Data referring to

students and their submissions are stored at Google's cloud storage facilities. A smaller amount of data is shared through a Dropbox account so that other instructors can have an easier access to them.

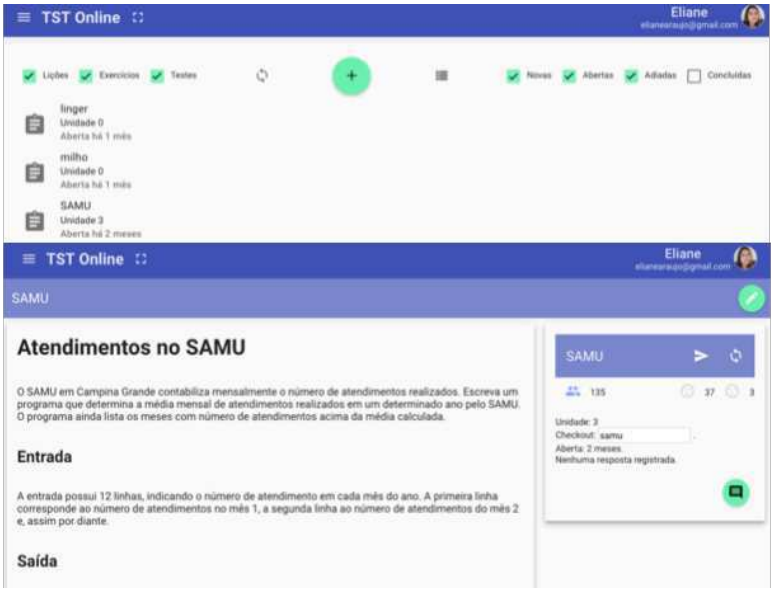

Figure 2.1: Students' View of TST Web Interface.

Students' typical routine to solve programming assignments using TST starts with a command line invocation to login into the system. Next, they access tst-online to login using Google accounts. In sequence, the students have access to their dashboard of ongoing work assignments. They can ask for new activities, of an informed unit, according to their performance in the course. From this point on, students interact with TST using command line commands. These commands are used to check out the assignment's files and test the program. TST provides feedback about passed and failed test cases. Initially, students are stimulated to locally test their program against a set of public tests. The test feedback referring to public test cases are more elaborated as it shows the expected and obtained output to a given input. When the program passes all public tests, it must be submitted to TST server so it is possible to test it against a set of secret tests. When the activity is finished, it must be closed at tst-online.

Instructor's typical routine to create programming assignments to TST starts with the problem creation. The specification must adhere to a pattern with the following sections: problem title, problem description, input, output and examples of execution. Next, the

instructor creates a test file composed of public and secret tests. The tests are specified in markdown language in a .json file. Then, he creates a reference solution to the problem and commits its file with private visibility. The programming assignment must be associated with a unit and marked as available to use. There are many other features provided by TST, such as activities versioning, but they are out of the scope of this summary.

TST modular architecture allows for the adherence of new commands as third party plugins. We used this feature to implement tst-*qcheck* and tst-oracle software as custom commands. They are the proof-of-concept tools we have created to test our proposals on code quality improvement and problem specification clarification, respectively. These programs use the same structure of TST installation, such as configuration files and libraries. However, they are not distributed along TST and must be installed apart. It means that it is up to the user to set their TST instance as they wish, including or not plugins.

In order to *qcheck* works, the instructors must include in their routine to the creation of programming assignments a generation and inclusion of a configuration file: *qcheck*.json. It is very simple to create a *qcheck*.json file. It is just necessary to execute a *qcheck* command informing the reference solution and the file will be created and saved to the current directory. This file must be committed with public visibility, to be distributed to users when they check out the assignment.

On the students' perspective, the use of *qcheck* is included in their routine to solve programming assignments just after their first correct submission to TST Server. At this point, it's known that the program is functionally correct. Students run *qcheck* commands to obtain feedback regarding code quality. This feedback is generated considering the information contained on *qcheck*.json file. Next, the student refactors the code until she finds "no warnings" message. A new submission must be done to TST server in order to register the changes done locally. Older submissions to TST server are overwritten and it is only took into consideration the last one.

Chapter 3

Feedback Generation to Support Computer Programming Problem-Solving

This chapter presents the general idea in providing automated formative feedback to support the programming process and how we have conducted the research toward this intent. Initially, it summarizes the roadmap of our studies and discusses the methodology we have followed to construct knowledge and evaluate our claims. Then, it provides an overview of the context in which the research happened and the strategies used to collect and filter data used in our analysis. Lastly, it examines the metrics and briefly exposes the statistical tools employed in the study.

In order to undertake the empirical studies on feedback generation, we need a tool support. For this reason, we have implemented a software, that would later become a TST plug-in: *qcheck*. This system is publicly available [1] and can be used for research purposes. A brief overview of *qcheck* tool is provided in this chapter. This can be useful to better understand the studies and the overall proposal of this thesis. Then, we will explain the environmental set up we performed to conduct the studies. We consider as setting up activities the instructional material we have produced to students, such as a guide to code quality in Programming one and programming assignments used on the qualitative exam, computational environment adapted to the experimental studies and more than one

[1] https://github.com/elianearaujo

hundred activities that were instrumented to *qcheck* usage. The environmental setting up was especially complex when we designed the longitudinal study. It required a long-term planning and resilience to change plans as new situations appear.

3.1 Research Roadmap

The ultimate goal of this research is to leverage introductory programming learning. As long as solving programming assignments plays a central role in this process, we focused our attention on this learning activity. It is known that computer programming goes far beyond crafting a code. In fact, teachers want that their students become proficient on solving programming problems.

The automated generation and delivering of formative feedback are essential to support this process mainly nowadays when we face a crescent interest on learning how to program. For this motive, the scale is an important issue we have to deal with. Automated assessment systems are an essential support for programming courses that deal with a high number of students, assignments and scarce human resources. AAS, typically, delivers feedback to students during the execution of programming assignments.

In this thesis, we propose an approach to generate automated feedback addressing the last stage of the programming process. At this moment, the proposed solution to the problem will be evaluated. In the original Polya method [Pól57], reasoning about other possible solutions and the encouragement to improve the existing one happens in this stage. When mapping this method to the programming process, is natural to think about software verification and validation. Which means to verify if the software meets the specified functional and non-functional requirements.

Apart from that, it is also necessary to assess the code in terms of readability, simplicity, efficiency, among others. Those aspects are fundamental to assure software quality. The process of perfecting code internal structure, improving its non-functional attributes and maintaining the same external behavior, is known as refactoring. We proposed to provide automated feedback so we can stimulate code refactoring aimed at code quality. Although refactoring in software engineering has a broader definition and includes different concerns, from this point of the document on we will refer to refactoring with this narrower perspective.

We performed a study and found that functional correctness alone, verified by automated assessment through tests, is not enough to explain the human assessment of a given assignment. After that, we proposed a set of measures, inspired by software metrics and the reference solution provided by instructors, to capture instructors quality expectations in regards to students' code to a given assignment. We performed a retrospective case study to evaluate the effectiveness of this approach in explaining teachers' manual grades. From this point on, we conducted a sequence of experiments, case studies, exploratory analyses, including a longitudinal study to refine and evaluate the proposal.

Firstly, we implemented a proof-of-concept tool, named *qcheck*, so that we can test our proposals in providing automated feedback. In an initial experiment, we examined if students would feel stimulated to refactor their code and improve its quality. We also observed if they effectively were able to improve their code quality. Secondly, we wanted to test if they improved their code because they used the tool *qcheck*, or if it happens in spite of it. This behavior was assessed in another controlled experiment using code quality feedback along the process. Then, we evaluated the validity of *qcheck* assessment. We wanted to check if the notion of code quality expected by experts, who are introductory programming course instructors, agreed with the tool code quality assessment. To this end, we conducted a blind-study to examine the agreement and better understand the situations when they disagree.

Finally, we conducted a longitudinal study in which we could perform quantitative and qualitative analysis regarding students' pattern of *qcheck* usage, post-feedback behaviors, code quality comparisons and evolution of programming skills. During the period of study, we were in laboratory classes and witnessed students' successes and difficulties in using the tool. It was a rich experience so we could gather insights about what is good and what needs to improve in our approach. Lastly, we conducted another human evaluation, by this time with students, in order to get their impressions about *qcheck* and the process of improving code quality using the feedback it provides. In the whole period of this study, we collected, under authorization, *qcheck* usage data. There is a lot more to discover mining this dataset. We refer these activities as future works.

3.2 Methods

This subsection presents the overall procedures applied on the empirical studies we have conducted throughout this research. Even though they belong to different categories: experiments, case studies, surveys and exploratory analyses, they share some common characteristics that we grouped in this section. Furthermore, methodological particularities of each study are detailed in its respective chapter.

3.2.1 Context

This research took place at UFCG in Computer Science undergraduate course. In particular, we analyzed students and used data produced during their interaction in Programming 1 and Laboratory of Programming 1 courses. These courses may be considered as one (theoretical and practical classes respectively). They are the first programming course of Computer Science major; so we can consider it equivalent to a CS1 course.

The research happened under the supervision of my advisors who were also part of the academic staff of Programming 1 course. The staff is composed of 4 instructors, graduate and undergraduate students that provide support as teacher assistants or students' tutors. Students are divided into three classes of Programming 1 and 4 classes of Laboratory of Programming 1. The programming language adopted in the course is Python. There are some peculiarities of the course that is worth to mention as it may impact in our research, given that it is our context of the study.

Programming 1 course does not follow a traditional teacher-centered approach: based on lectures and few exams. Nowadays, they employ flipped classroom, continuous assessment and self-paced with mastering learning. The course is divided into 10 thematic units, each of them with specific learning outcomes students have to master. The self-paced allows students to be in different units, according to the knowledge they have mastered and coexist in the same course. There is a set of programming assignments for each unit. Students unlock the access to a subsequent unit when they have correctly solved a 65% of the total number of assignments for that unit. Exams are composed by a set of assignments, from various units, similar to those students are used to do in laboratory classes and at home. There are exams every week. Students are encouraged to solve assignments as much as they can in the exam

so that, they can evolve to the next unit (at their own pace).

The course greatly stimulates students' code production through programming assignments. They are used, for example, as a starting point of classroom discussions, as they are flipped. To support this intense activity, the course relies on TST. This already described system is intended to randomly assign activities to students (observing their unit), gather students submissions and provide feedback on code functional correctness. With a view to evaluate our proposals, we evolved our proof-of-concept tools to be plugged into TST. This allowed students to have a seamless experience of the use of our "under evaluation" tools.

Since 2016, this research was subscribed under the committee of ethics in research involving human beings under the number CAAE 54944716.1.0000.5182. [2].

3.2.2 Data Collection

In general, the studies of this research were performed using data collected from the interaction between students and the instructional material through TST. We also used data produced by instructors during Programming 1 course evaluations, such as: grades resulting from a manual assessment of students' programs; annotations made on students code; students' performance status according to their evolution on the course, etc. Besides, we gathered data from human studies, such as the interview and survey with students and the blind-study with experts.

At the longitudinal study, we instrumented *qcheck* tool to send reports of its usage, under the user authorization. The software sends to a central server minimal data, so it does not overload the net nor reduce its performance, regarding its use in an assignment. This data, for example, contains: datetime, user, IP, activity, metrics, among others.

An important aspect of this dataset refers to the total number of students. As *qcheck* can only be used in assignments of the 3rd, 4th and 5th unit, our students' sample was changing along the time. Initially, students with greater performance composed the sample. By the end of the study, students that were retained on those units lasted on the sample. We plotted in Figure 3.1 students' exams performance over five weeks, the period of observation. We can observe that students' retention on unit 4 is outstanding /footnoteThe subject of the 3rd,

[2]More information about the project we have submitted to that board can be found at http://plataformabrasil.saude.gov.br/

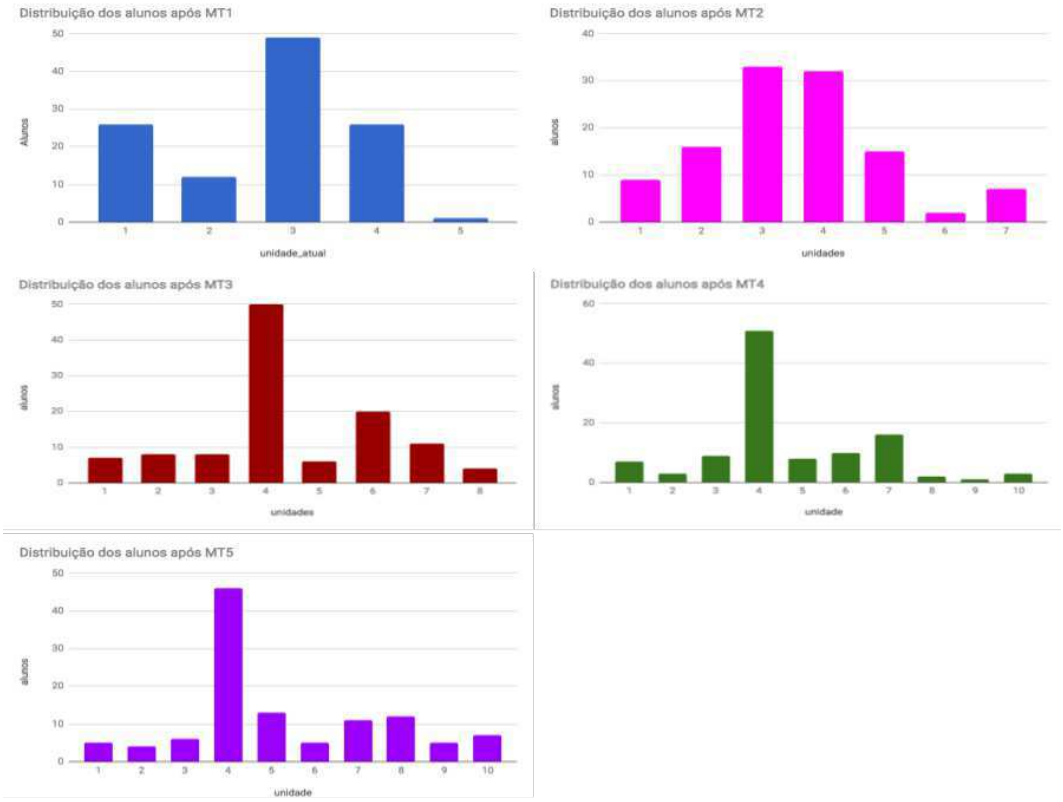

Figure 3.1: Students Performance During the Longitudinal Study.

4th and 5th units refer to, conditional structures, iterations with *for* and *while*, respectively..

Data filtering was done in different ways according to the objective of the empirical study. In general, we have used TST to separate correct from incorrect submissions and *qcheck* to count the number of quality warnings of a given code.

3.2.3 Demographics

Each experimental study we have performed in this research was conducted with different subjects (students from different semesters). But, the data we have collected during the longitudinal study was used in various analyses. For this motive, we are going to provide some information about this group here. We obtained this data at the beginning of the semester when we explained to the students that they were taking part of a research study and its purposes. They signed the term of agreement and filled a socio-demographics questionnaire.

There were 115 students enrolled in Programming 1 course in 2017.1, by the time we

finished the longitudinal study data collecting, at least 109 students have advanced for the first unit. The Table 3.1 shows some basic information that might help us to delineate the subjects' profile.

Table 3.1: Demographic Data of Students of 2017.1 Programming 1 Course.

Total of students	115		
Gender	Male	Female	
	80.9 %	19.1 %	
Age	<18	18 to 22	>22
	25.2 %	64.3 %	10.5 %
Previous programming experience	Yes	No	
	47.8 %	52.2 %	

The Figure 3.2 corresponds to students' answers referring to their self-confidence about their performance in the course. They have answered a Likert scale question.

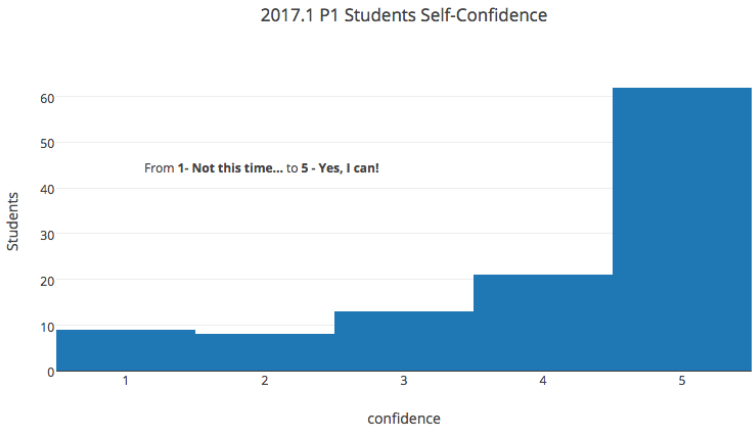

Figure 3.2: Students Course Performance Expectancy.

3.2.4 Metrics

The metrics that we have used in the studies are detailed in their respective chapter. For instance, the metrics created to reflect code quality according to teachers' expectations. However, in those involving *qcheck*, there are common measures that will be described in this subsection in order to avoid information replication.

The metric *nsub* refers to the number of submissions done by a student to a programming assignment to TST server. It differs from the number of revisions of the assignment that is the number of attempts to the correct answer, according to a set of tests. In TST dynamic, students receive a set of public tests when check out the assignment and can use it to test the program in their own environment. When they submit the solution to the server, another set of tests, namely secret tests, is added and the code is evaluated under more restrictive conditions. The metric *nsub* accounts only functional correct submissions, code that meets functional requirements and passes public and secret input/output tests proposed to that assignment.

The metric *W* summarizes the number of code quality issues of a given program according to *qcheck* tool assessment. The tool produces warnings referring to code and style. Style warnings (*S*) refers to the number unconformities with Python programming language coding standards orientation registered under pep8 [Pep15], for example, lack of white spaces between operators or indentation problems. Code warnings (*C*) are based on the set of software metrics proposed in this work to assess introductory programming assignments' code.

W = C + S'	C - value may vary from 1 to 4, as four metrics are evaluated by *qcheck*; S' - value vary from 0 to 1, standing for the presence or absence of style warning related by *qcheck*.

Other important metrics are the derivative of metrics S and W, respectively ΔS and ΔW. These metrics are only computed using functionally correct submissions according to TST server tests. They are computed as the difference between the measures extracted from the code of last correct submission and the first correct submission. ΔS refers to style warnings *S* and ΔW to the normalized number of warnings *W*. We expect these values to be negatives.

$\Delta S = S_f - S_0$	S_f - Is the number of style warnings of the last correct submission S_0 - Is the number of style warnings of the first correct submission.
$\Delta W = W_f - W_0$	W_f - Normalized number of warnings of the last correct submission W_0 - Normalized number of warnings of the first correct submission.

From metric ΔW, we came up with three situations:

- Δ**W < 0**: It means that the student was capable to recognize and fix aspects in his/her code to improve its quality.
- Δ**W = 0**: It means that the program's number of warning remained the same.
- Δ**W > 0**: It means that the student was not capable to improve the quality of his/her code; instead the last code submission became worse than the first submission.

The same analysis is valid for ΔS.

3.2.5 Analysis

We quantitatively evaluated the data we have collected using descriptive and inferential statistics. We performed some simple statistical analyses using hypothesis test. As a rule, we used the nonparametric Wilcoxon signed-rank test to compare two distributions. As response variables, we generally used the metrics previously described. We used Wilcoxon test instead of the usual t-tests, since we could not find normality on our distributions.

We performed other analyses using Pearson's chi-squared tests when investigating the relation among categorical variables. Using this test we assessed if the observed differences among the values in the distributions happened by chance.

We analyzed longitudinal data carefully, as this type of data has characteristics that elevate them to a different level of analysis:

> "The distinguished features of a longitudinal study is that the response variable of interest and a set of explanatory variables are measured several times on each individual in the study. The main objective of the study is to characterize the change in the repeated values of the response variables and to determine the explanatory variables most associated with any change" [HE06].

In this study, we have summarized, using ΔS and ΔW, a sequence of observations regarding the tuple (student, activity) along the time. We filtered a sequence of observations (student, activity) in only one entry and composed a new dataset. Certainly, we missed the benefits that a more complex longitudinal data analysis could potentially have provided to us. But we could still achieve significant results, employing a more modest analysis approach.

3.2.6 *Qcheck*

Our approaches to provide automated feedback to aid students' code quality improvement were made concrete on a proof-of-concept tool: *qcheck*. This tool is an instantiation of the proposed approach. Its code is publicly available at GitHub under AGPL-3.0 license. *Qcheck* was created to assess our proposals validity with students of an introductory programming course in a set of empirical studies, especially the longitudinal study. Naturally, the conclusions we came up with this research were drawn regarding our studies' context, but the ideas here exposed can be used and adapted to other programming courses and AAS.

In short, *qcheck* is a TST custom command that is used to check student's solution code quality to a given problem. It is based on well-known static metrics that help to evaluate software maintainability, such as logical lines-of-code, cyclomatic complexity, and others. *qcheck* takes the canonical solution provided by the assignments' author as the reference to generate feedback hints to advise students in what aspect their code can improve. It has its own installation process, separated from TST bundle. It must be installed in each user environment.

The tool was conceived to use in an environment with TST already installed. When a new programming assignment is created, the author needs to execute a *qcheck* command informing the reference solution. This command creates a file *qcheck*.json, containing the values used by *qcheck* on the client side to generate the feedback. This file must be uploaded along with other files of the assignment. When a student checks out a new activity, this file is downloaded to her or his working directory.

```
$ tst qcheck -s reference.py
```

From the student point of view, *qcheck* usage is very simple. We advise students to use *qcheck* after their program is completely tested. *Qcheck* produces two blocks of warning messages: it gives hints about code (programming solution to the problem) and style (Python coding standards). Style hints are based on PEP8 - Python community canonical style guide [Pep15]. An example of use is following listed: (a) first *qcheck* invocation and (b) second *qcheck* invocation after student change her code including more header lines.

(a)

```
$ tst qcheck pedro_filho.py
```

```
# pedro_filho.py

**6 Warning(s)**

### Code
- Your program header is too short.
- Your program has too many decision points.
- Your program has too many lines of code.
- Your program has too many operations (Example: +,-,==, etc).
### Style
- 1:1: E265 block comment should start with '# '
- 2:1: E265 block comment should start with '# '
```

(b)

```
$ tst qcheck pedro_filho.py
# pedro_filho.py

**5 Warning(s)**

### Code
- Your program has too many decision points.
- Your program has too many lines of code.
- Your program has too many operations (Example: +,-,==, etc).
### Style
- 1:1: E265 block comment should start with '# '
- 2:1: E265 block comment should start with '# '
```

3.2.7 Setting Up Activities

Setting the environment up to perform the longitudinal study was a complex and time-consuming activity. It included a study design foreseen 5 weeks of observations and data collecting, settlements with the course staff including meetings and lectures, adaptation of more than a hundred activities from Programming 1 course asset to *qcheck* usage, computational environment setting up at university laboratories, availability to students'

requests at discussion forum, they use Slack [3] as an official communication channel, among others.

An already existing system simplified the environmental set up at computing laboratories at the UFCG. The system Prog1Box [4] was developed in-house and works as a virtual box. It is used during Programming 1 exams and in some laboratories activities, such as marathons. It restricts users' access only to authorized Internet sites, physical locations or hardware devices. Using Prog1Box we could easily set up different software configurations for the experimental and control group in different computing laboratories physical locations.

As a preparation for the longitudinal study, that happened during 2017.1 classes of Programming 1 course, we adjusted 106 assignments available to students referring to unit 3rd, 4th, and 5th. It was necessary to adapt all available activities in the asset since TST randomly chose the activities to students as they request. Legacy questions do not have a *qcheck*.json file and some of them, also, do not have a reference solution associated with it. Each assignment preparation included to:

- Search for the assignment author;
- Find the original reference solution on historical database;
- Produce or, when possible, having the author to produce a reference solution;
- Validate the reference solution according to intended learning outcomes;
- Create *qcheck*.json files;
- Upload assignments' files and
- Commit the activity to TST system.

3.2.8 Summary of Studies

In the following chapters, we are going to present studies and results of our research on generating formative feedback aimed at programming assignments' code quality improvement. Each chapter refers to a broad research question and details a sequence of

[3] http://slack.com

[4] https://prog1box.appspot.com

empirical studies intended to investigate it. We decided not to follow a chronological order of the studies developed during our doctoral research but a logical order. The idea was to emphasize the claims we came up with, as a result of the research effort, using distinct methods.

These claims summarize the main contributions of this work. In order to construct knowledge about them, we took into consideration the mature and vast literature on programming education. Yet, refurbished by new challenges brought by the growing number of students' enrollment and the need to scale pedagogical practices with quality.

- C1 - We generate automated feedback based on teachers of introductory programming code quality expectations;
- C2 - Students improve programming assignments' code prompted by timely and automated feedback;
- C3 - Students improve programming skills stimulated by the reflection on their programming assignments code with the purpose to improve its quality.

The chapter 4 examines the "Generation of automated code quality feedback". In this chapter, we will investigate and demonstrate that it is possible to attain this objective with *qcheck*. Initially, we present our first study when we investigated software metrics that could explain the difference in manual grades of functionally correct programming assignments. In sequence, we use these metrics as a foundation to generate feedback messages about code quality improvement. The second study reports an evaluation with human specialists of the direct and indirect effects of *qcheck*.

In chapter 5, discuss the possibility of the code quality improvement feedback delivered to students directs the improvement of their programming assignments' code. The first and second studies present experiments with randomized controlled samples. We assessed the use of *qcheck* in experimental groups. The main difference of these studies was the evaluation of the willingness of improving code quality and its consequence. On the first study, we wanted to investigate if the novelty of an instrument to produce feedback about code quality would motivate students to attempt to generate a better code and if they, in fact, succeed. On the second study, both experimental and control group were stimulated

to improve their codes quality, and they also received a written material to help them: "Programming 1 code quality expectations". However, only students from the experimental group were able to use *qcheck* feedback. The last study presented in this section discusses and contrasts the summative assessment of the quality of students' code production in a given period of time in the Programming 1 course with the use of *qcheck* by the students during this same period.

The chapter 6, finally, considers the consequences to learners of providing feedback about code quality improvement using the proposed tools during a programming course. Recently, De Nero and colleagues (2017) discussed that there were many initiatives and advances in automated feedback platforms aimed at programming education, but few studies on its effects on real programming courses [DSPQ+17]. In this chpater, we are going to present our main findings in a longitudinal study and discuss its implications to learners, instructors and the course itself. Furthermore, we are going to present an evaluation of the approach performed with the students that used the tool in their activities.

Chapter 4

Generation of Automated Code Quality Improvement Feedback

In this chapter, we will investigate and demonstrate that it is possible to generate automated feedback of code quality and stimulate students to reflect on their code, besides functional correctness. Initially, we present our first study when we investigated software metrics that could explain the difference in manual grades of functionally correct programming assignments. In sequence, we use these metrics as a foundation to generate feedback messages about code quality improvement. The second study reports an evaluation with human specialists of the direct and indirect effects of our proposal.

We are going to report studies regarding the information we used to compose feedback about students' code quality since its proposal until the feedback validation by experts. The general question we made was:

> "Is it possible to generate automated feedback based on code quality expectations of introductory programming teachers?"

The starting point of this research is the generation of data that will be used to compose code quality feedback. Initially, we performed an empirical study, reported in the paper [AGF13], aimed at identifying whether the adherence to coding standards would be an indication of better code quality. In sequence, we sought for other measures to help us identify what could be automated from the human quality assessment of students' programs. The retrospective case study conducted to achieve these goals was discussed in the paper

[ASF16].

The first study of this chapter, reports an investigation of the validity of using measures as surrogates of the quality expected by instructors on students' code. As a baseline for such code quality, we use the reference solution provided by the instructors when creating the programming assignment. It is important to notice that this reference solution must convey the learning outcomes students have to master, as well as, the expected code quality.

We propositioned a set of software measures that can be used to express qualitative aspects: RLLOC, RCC, RH and RPEP8. They are based on software quality metrics, largely used by the industry and referred to in other academic initiatives towards novice programming [AM05] [MY99] [PHG+15]. 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 can generate and provide a feedback message to the student, i.e. a hint of what it could be improved in order to obtain a better quality code. The research question that directed the study was:

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

The second study was intended to evaluate the contrast between a human expert assessment and the assessment provided by a tool that took into consideration the proposed metrics, implemented in our proof-of-concept tool - *qcheck*. We proposed the research question:

The research question that directed the study was:

> RQ2: Does *qcheck* approach capture expert notion of code quality?

In this study, experts evaluated a pair of students' programs versions of a program before and after *qcheck* feedback. Those pairs of codes are functionally correct versions of students' assignments that were randomly presented to the expert. We contrasted the agreement on quality assessment among experts and *qcheck*.

4.1 Context

The proposed studies took place at the UFCG, in the Computer Science undergraduate course, in the context Programming 1 course. The studies happened in different academic semesters between 2013.2 to 2015.1

In the study 4.2, a retrospective case study, we used data gathered using TST in a former course edition – 2013.2. Our dataset was composed by 403 functionally correct submissions, from 102 students, referring to 12 different programming assignments. Each assignment was manually assessed, annotated and graded by at least one instructor. By this time, 4 instructors composed the course staff – 3 colleagues and me.

In the study 4.3, we collected programs produced by students that took part in a controlled experiment and have instructors to assess them. This study, which happened in 2015 with students of Programming 1 course. The experts that we have recruited to this study were three instructors of the course. They have a compared background in teaching the course.

4.2 Measuring Students' Code Quality Through Software Metrics

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.

4.2.1 Methods

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: Do we have good measures to capture some aspect of assessment subjectivity?

In answering this research question, we investigated whether the the measures RLLOC, RCC, RH and RPEP8 explain the differences observed on the grades, manually assessed, of functionally correct submissions. In practice, if 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. Three experienced instructors manually graded them on a scale that ranges 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 [Rad14], 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 [Pep15]. 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.

4.2.2 Metrics

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 to in other academic initiatives towards novice programming [AM05] [MY99] [PHG+15].

Instructors approach the manual grading activity in different ways. However, they usually agree whether a program is "very good" or "very bad" [FHL+13]. 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 serve as input to a quality analysis [AM05]. 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 of counting the number of operators and operands in a program [AM05]. In this study, we have measured the Halstead volume.

- cc: It was conceived by McCabe [McC76] 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 [AM05]. 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 reference 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 the 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 [Pep15]. 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. 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 real-world extracted from the reference solution.

The Table 4.1 presents the measurements we proposed to assess code quality along with its acronym. From this point forward, we are going to refer to these measurements by the

Table 4.1: Measurements Proposed to Assess Code Quality.

Acronym	**Description**	**Formula**
RLLOC	Ratio between reference solution's lloc and student's code lloc.	$\frac{lloc(studentCode)}{lloc(referenceSolutionCode)}$
RCC	Ratio between reference solution's cc and student's code cc.	$\frac{cc(studentCode)}{cc(referenceSolutionCode)}$
RH	Ratio between reference solution's h and student's code h.	$\frac{h(studentCode)}{h(referencesolutionCode)}$
RPEP8	Ranking position of the number of pep8 violations of the student's code.	-

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.

4.2.3 Data Collection

The Figure 4.1 shows the distribution of instructor's grades of functionally correct submissions. These submissions obtained "green-bar" as passed all automatic tests provided by the instructor. If they were automatically 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 relied solely on automatic tests, more than 70% of the submissions would obtain a grade higher than a human instructor thinks it deserves.

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 the traditional functional test. The Figure 4.1 exposes that functional correctness, alone, does not reflect the instructor

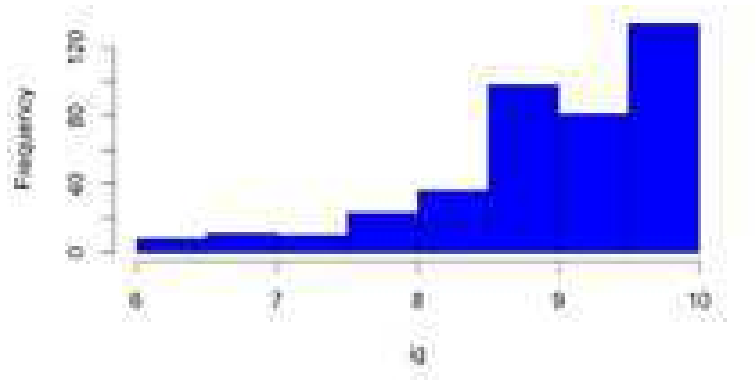

Figure 4.1: Distribution of Manual Grades Assigned to Functionally Correct Submissions.

manual assessment.

4.2.4 Results and Analysis

This subsection reports the results of the studies to answer our research question: Whether the proposed quality measurements can explain the differences observed in 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 ranking sum test to compare 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 into 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 grades medians. Instructor's grades for the equal-lower group are higher, on average than the grades of the other group with adequate statistic significance (p-value < 0.001 and 0.05 significance level.). Hence, we can reject the null hypothesis in favor of the alternative. The results reveal, at least for these data, the better the measurements the better are the grades. As the 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 4.2 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.

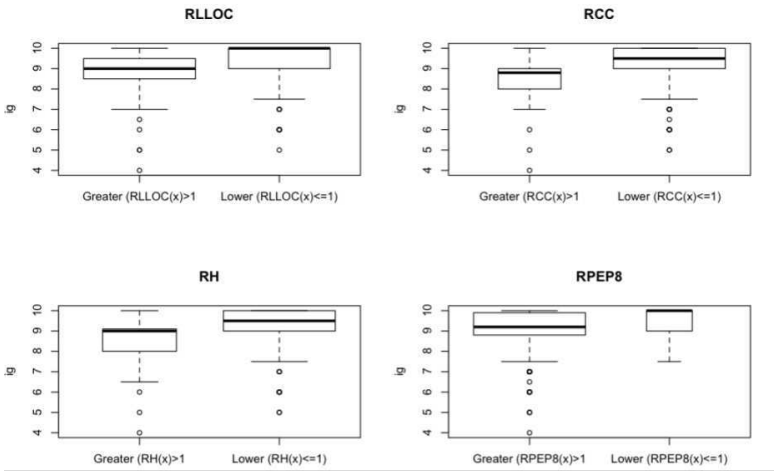

Figure 4.2: Distribution of Instructors' Grades and Each Metric.

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 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.

4.2.5 Discussion

In this study, we focused on measuring aspects of the code that instructors usually took into consideration when manually assess programming assignments: qualitative aspects that go beyond functional correctness. We wanted to investigate program features regarding code quality issues. We conjectured that instructors' reference solution for a programming assignment includes most of his expectations about a student's code quality. 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 subjective: the qualitative aspects of an instructors' assessment. Our aim is to use these findings to generate feedback regarding code quality about introductory programming assignments.

4.3 Assessing Students' Code Quality with *qcheck* Support

This study intends to gather evidence that the feedback we generate using the proposed metrics to a given program indeed reveals instructors expectations about code quality to that program. We conducted the study motivated by the research question:

> RQ2: Does *qcheck* capture expert notion of code quality in programming assignment assessments?

We had experts to assess a set of programs made by students that used *qcheck* and its feedback messages to improve their code quality. Different versions of the programs were evaluated by the experts. We expected that the study confirmed our conjecture that *qcheck* really captures this notion.

4.3.1 Methods

From a set of 35 students, we selected a set of submissions that meet the following criteria: students that make more than one submission and whose submissions differ from each other. After applying these filters, we composed our dataset with the lasting 16 pairs of submissions (S_0, S_n) from students of control and experimental groups. In sequence, we used *qcheck* to calculate the value of W (number of *qcheck* warnings) for each submission pairs.

Finally, we set up a single-blind study to have experts' evaluating these codes. We recruited three domain experts that are experienced teachers of Programming 1 course. We printed and handed them out a booklet with all programs. Each page had a pair of students' programs and the question: "Is code A better than B?" They had to choose an answer among: Yes, Equivalent or No. The printing order of the pair (S_0, S_n) on each page was randomly chosen and also the order of the pages in the booklet - they may vary from teacher to teacher. Teachers were invited to a meeting room and, together, received a brief explanation of the study purpose and how to proceed. They performed their evaluation individually. The experiment lasted less than 30 minutes. At the end, we collected data evaluation of each expert – R_1, R_2 and R_3 – and contrasted them among each other and *qcheck* tool – T.

4.3.2 Metrics

The metric used to assess programs, regarding *qcheck*, is W that represents the number of code quality warnings captured by the tool in the student code. As *qcheck* flags quality issues with warnings, the lower the value of W, the better the program code quality.

We took into consideration a set of submissions from a student to a programming assignment ($S_0, S_1, S_2, \ldots, S_n$), we selected a pair composed by the first and last functional correct submissions (S_0, S_n) and calculated the value of W_0 and W_n.

As we have seen, *qcheck* code quality evaluation yields:

- Y: S_n presents better code quality than S_0, if $W_n < W_0$.
- N: S_n does not present better code quality than S_0, if $W_n > W_0$.
- E: S_n code quality is equivalent to S_0, if $W_n = W_0$.

Experts used their own judgment, based on Programming 1 course's criteria, to evaluate programs' quality. From this point of the document on, we will also refer these experts as raters. Provided we know that the notion of code quality is predominantly a subjective measurement, we want to verify if *qcheck* can assess code quality just as the experts do. Given that, we pose the following hypotheses:

H1: Raters R_1, R_2, R_3 judgment about code quality is consistent among each other.

H1-0: Raters R_1, R_2, R_3 judgment about code quality is inconsistent among each other.

H2: R_m judgment about code quality is consistent with *qcheck* judgment T.

H2-0: R_m judgment about code quality is inconsistent with *qcheck* judgment T.

By "consistent judgment", we mean that their agreement is statistically significant.

4.3.3 Data Collection

The following Table 4.2 summarizes ratings of each pair (R_n,T). Each cell accounts the occurrence of a pair of evaluations between (R_n, T) that matched. The diagonal line in each table represents the highest rating agreement. The last column and row summarize the total of evaluations of each category, provided there were 15 programs.

Table 4.2: Pairwise Code Quality Evaluations Among Raters X Tool.

Rater1	**Tool**			**Rater2**	**Tool**			**Rater3**	**Tool**		
	N	E	Y		N	E	Y		N	E	Y
N	**3**	1	1	N	**5**	0	1	N	**5**	1	1
E	2	**3**	0	E	0	**3**	0	E	0	**3**	0
Y	0	1	**5**	Y	0	2	**5**	Y	0	1	**5**
TOTAL	5	5	6		5	5	6		5	5	6

4.3.4 Results and Analysis

In accordance to the proposed hypotheses, we first investigated raters agreement (H1.1) and, as we found they were consistent, we proceeded to the agreement evaluation (H2.1) between *qcheck* tool and the raters.

In order to investigate agreement consistency among three raters, we computed the *Fleiss' kappa* inter-raters reliability index. Unlike *Cohen's kappa*, which is a more common statistical measure that assesses agreement between two raters, Fleiss' kappa can be used to a fixed number m of raters when evaluating a trait with categorical rates. This index aims to calculate the degree of raters' agreement when evaluating a trait over it would be expected to happen by chance. This index ranges from 0 (zero) to 1 (one). The agreement improves as Kappa's index value approximates to one.

The value of kappa index calculated among R_1, R_2 and R_3 is approximately 0.81 (the statistic is significant as p-value < 0.001), which indicates that agreement is almost perfect [LK77]. Thus we can reject the null hypothesis in favor of the alternative (H1.1) and, as a result, we have strong evidence to assume that experts' judgment is consistent.

Next, we computed *Cohen's kappa* index for each rater (R_1, R_2, R_3) and *qcheck* tool (T). In this study, we used this measure to compute the disagreement of each rater with the tool and the raters among each other. We intended to investigate if raters' judgment was, individually, similar or unlike of those provided by the tool.

Ratings provided in this experiment for code quality are considered ordered-categorical data N, E, Y. It means that a pair of raters agrees more if they answer the experimental question "code A is better than B?" with Y and E, than Y and N. So, we mapped the values of N, E, Y to 1, 2, 3 to quantify this distance in agreement. As the traditional *Cohen's kappa* measurement does not take into consideration the degree of disagreement, we used the modified weighted Cohen's kappa. This method ponders disagreements between two raters with a set of weights for each possible categorical rate: the higher the disagreement the higher the weight. The following Table 4.3 shows *Cohen's kappa* index for each pair (R_n, T) and raters among each other.

The values presented indicate a substantial agreement among raters and tool. The p-value for all tests was less than 0.01 (ranging from 0.01 to 0.0002), meaning that Kappa index value is statistically significant. Although the interpretation of this index is controversial [LK77],

Table 4.3: Agreement Index Value Among Raters and *qcheck* Tool.

	R1	**R2**	**R3**	**Tool**
R1	-	0.75	0.92	0.69
R2	0.75	-	0.85	0.80
R3	0.92	0.85	-	0.80
Tool	0.69	0.80	0.80	-

it directs us towards interesting analysis. The best agreement found was between teachers (R1, R3 - 0.92), meaning that their assessment of code quality is almost perfect, at least in the proposed study. In general, agreement among teachers is slightly more significant than each rater with the tool. The worse agreement index value was found between (R1, T - 0.69). Although this absolute value is the smallest found, it can still be considered an indicator of significant agreement.

In order to investigate the second hypothesis (H2.1) of this study, if raters and tool consistently agree, it was necessary to determine a consensus among raters evaluation on the study items. We were inspired by a real course situation to find a consensus: when two or more instructors are assessing a student assignment. If there is any divergence in their opinion, they usually agree to assess the assignment with the more frequent rate. For this reason, we chose the statistical measure *mode* to represent the consensual evaluation among raters. In sequence, we computed weighted Cohen's kappa between (T, Rm), where Rm stands for the mode among raters evaluation.

As a result, the computed index value revealed a significant agreement between human raters and the *qcheck* tool. Cohen's kappa index value was 0.80 for a p-value of 0.001. So, we can reject the null hypothesis in favor of the alternative and claim that "raters judgment about code quality is consistent with *qcheck* judgment, at least for this study".

4.3.5 Qualitative Evaluation

This qualitative evaluation was motivated by the question: "If raters agreed so much, when do they disagree?". It is perhaps even more insightful to discover what are the divergences among ratings than to confirm their agreement, at this point. We found that there were only

4 items in the dataset that made evaluations controversial, shown in Table 4.4.

Table 4.4: Disagreement Among Ratings (Rn and *qcheck*).

ID	**R1**	**R2**	**R3**	**Tool**	**Mode**
S67	2	1	1	1	1
S70	2	1	1	1	1
S62	1	3	1	2	1
S64	1	1	1	3	1

The first and second line of the table shows similar ratings: raters 3 and 2 agreed with the tool that assesses 1 to both pairs of codes. It means that they consider that code B presents better quality than code A. Conversely, rater 1 assesses them as an equivalent code. Although the difference among rating weight is small, they are consistent as can be seen in the code excerpts below.

We list both versions of the student S67 code in order to discuss the disagreement among ratings. The difference between the code (A, B) of student S67 is on lines 5 and 14. On code A in line 5, the variable "*cont*" initiates with the value 0, while on code B, on line 5, this same variable initiates with value 1. This change causes suppression of the operation "+1" on line 14, on code B. It appears that rater 2, rater 3, and *qcheck* tool considered it as positive when assessing code B better than code A.

The issue found on student S70's code A is a redundant line: it increments a variable inside an "if and else" block (lines 10 and 13). As can be seen in code B version, S70 excludes one of these lines bringing the line out of the conditional structure (line 12). Both codes are listed in sequence.

S67 and S70 eliminated one operation in their codes. It seems that, in Rater 1 judgment, this is not sufficient to assert, in terms of code quality, that one code is better than the other. He/she considered both pairs equivalent. We believe that this is a positive case for our study in two aspects. First, it shows the consistency of judgment of the rater assessment. We assume that there is a component of subjectivity in the human evaluation of code quality, for this reason, the tool cannot, and is not intended to, perfectly mimic human assessments. Second, the tool evaluation agreed with the majority of raters in this case. This signals

```
# coding: utf-8
# vida collatz
# Xxxx Xxxx / programacao 1
Ni = int(raw_input())
cont = 0
while True:
    if Ni % 2 == 0:
        Ni = Ni / 2
        cont += 1
    else:
        Ni = (Ni * 3) + 1
        cont += 1
    if Ni == 1: break
print cont + 1
```

```
# coding: utf-8
# vida collatz
# Xxxx Xxxx / programacao 1
Ni = int(raw_input())
cont = 1
while True:
    if Ni % 2 == 0:
        Ni = Ni / 2
        cont += 1
    else:
        Ni = (Ni * 3) + 1
        cont += 1
    if Ni == 1: break
print cont
```

Listing 1: Student S67 Assignment's Code Comparison (A) and (B).

```
#coding: utf-8
#Ler da entrada um numero.

numero = int(raw_input())

cont = 1
while numero != 1:
    if numero % 2 == 0:
        numero = numero / 2
        cont += 1
    else:
        numero = 3 * numero + 1
        cont += 1
print cont
```

```
#coding: utf-8
#Ler da entrada um numero.

numero = int(raw_input())

cont = 1
while numero != 1:
   if numero % 2 == 0:
        numero = numero / 2
   else:
        numero = 3 * numero + 1
   cont += 1
print cont
```

Listing 2: Student S70 Assignment's Code Comparison (A) and (B).

that there exist cases that evaluations produced by *qcheck* are not identical to those from a particular teacher, but it is still representative and useful.

The evaluation of student S62's code, listed in sequence, revealed the biggest discrepancy among raters. Rater 1 and 3 assess code B better than A, while in rater 2 opinion is the opposite. *Qcheck* evaluates the pair of code as equivalents. In fact, the pair of code is very similar. It main difference is that one uses the programming idiom "while True/ If / break" and the other employs "while <condition>". In this introductory programming course, students are encouraged to use the first construction "while True/ If / break", when possible. However, this is a controversial point in programming style and, sometimes, can be considered a matter of personal taste. Again, we recall the human subjectivity issue of code quality analysis. In this case, the tool assessed both codes as equivalent, in the middle distance from both opposite ratings (Y, N). Once again, we considered *Qcheck* evaluation disagreement as positive as it can be.

```
# coding: utf-8
# vida_collatz
# Xxxx Xxxxx

num_ini = int(raw_input())
contador = 0

while True:
    contador +=1
    if num_ini == 1: break
    elif num_ini % 2 == 0:
        num_ini /= 2
    else:
        num_ini = 3 * num_ini + 1

print contador
```

```
# coding: utf-8
# vida_collatz
# Xxxx Xxxxx

num_ini = int(raw_input())
contador = 1

while num_ini != 1:
   if num_ini % 2 == 0:
       num_ini /= 2
   else:
       num_ini = 3 * num_ini + 1
   contador +=1
print contador
```

Listing 3: Student S62 Assignment's Code Comparison (A) and (B).

Student S64 code represents the most discrepant case in terms of the agreement between *qcheck* tool and the raters. Raters agreed among each other that code B is better than code A. But, *qcheck* assessed code A better than code B. The following code listing shows the pair of code in order to depict this divergence.

As can be seen, the main difference between both codes is that code B encapsulates code

```
# coding: utf-8

numero = int(raw_input())
conta = 0
while True:
    if numero % 2 == 0:
        numero = numero/2
        conta += 1
    else:
        numero = 3*numero + 1
        conta += 1
    if numero == 1:
        conta += 1
        break

print conta
```

```
# coding: utf-8

numero = int(raw_input())

def collatz(numero):
    conta = 0
    while True:
        if numero % 2 == 0:
            numero = numero/2
            conta += 1
        else:
            numero = 3*numero + 1
            conta += 1
        if numero == 1:
            conta += 1
            return conta
print collatz(numero)
```

Listing 4: Student S64 Assignment's Code Comparison (A) and (B).

A in a function named *collatz(numero)* and invokes it just after its definition. Perhaps raters' judgment about code quality touched a wider layer not covered by *qcheck* metrics as all human raters agreed that code B is better than code A. Or, maybe, it is just another case of human subjectivity that could not be captured by the tool. In fact, this case is controversial and it reaches the limitations of *qcheck* assessment when compared to human evaluations.

4.3.6 Summary and Discussion

In this empirical study, we summoned three Programming 1 instructors to act as raters evaluating students' code quality between submissions for a given programming assignment. We also used *qcheck* to evaluate the code quality of the submissions. We investigated if *qcheck* approach is able to capture experts' notion of code quality.

Firstly we examined the consistency of the evaluations among instructors. We found that their level of agreement is almost perfect, according to Fleiss Cohen's kappa, a statistical measurement of the degree of agreement that ponders agreements that occurred by chance.

Then, we made a pairwise evaluation and found that *qcheck* and each instructors' agreement were statistically relevant. We use the mode of instructors' ratings as a consensual measurement and calculated its agreement with *qcheck* ratings. At least for this study, raters'

judgment is consistent with tool judgment, so *qcheck* is able to capture experts' notion of code quality.

Finally, we qualitatively evaluated each code that caused divergences among raters and tool. In general, they seem to be related to a matter of personal taste of raters. When proposing *qcheck* tool, we assume that there is a component of subjectivity in the human evaluation of code quality. The tool is not intended to, perfectly mimic human assessments. It tries to capture, in a great extent, code quality standards using instructors' reference solution as a beacon. This study shows that there are cases that instructors and *qcheck* evaluations are not identical, but, in general, *qcheck* evaluations are representative. The bottomline is that this results give us confidence that *qcheck* evaluation are perfectly useful as representative of instructor's assessment. In this sense, we can rely on the approach to generate feedback that helps code quality improvement or when evaluating students' code quality.

Chapter 5

Code Quality Improvement Prompted by Automated Feedback

In this Chapter we discuss the possibility of the code quality improvement feedback delivered to students directs the improvement of their programming assignments' code. The first and second studies present experiments with randomized controlled samples. We assessed the use of *qcheck* in experimental groups. The main difference of these studies was the evaluation of the willingness of improving code quality and its consequence. On the first study, we wanted to investigate if the novelty of an instrument to produce feedback about code quality would motivate students to attempt to generate a better code and if they, in fact, succeed. On the second study, both experimental and control group were stimulated to improve their codes quality, and they also received a written material to help them: "Programming 1 code quality expectations". However, only students from the experimental group were able to use *qcheck* feedback. The last study presented in this section discusses and contrasts the summative assessment of the quality of students' code production in a given period of time in the Programming 1 course with the use of *qcheck* by the students during this same period.

We 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 solution must encompass the programming abilities and code quality expected by the instructor for the programming assignment. We proposed the generation of automated feedback using that

information. In the studies that we report in this chapter, we gathered evidences to ground the claim:

> "Students can improve code of their programming assignments prompted by timely and automated feedback"

5.1 Context

The studies we report in this section happened in February/2015, September/2016, and June/2017, in this sequence. They all occurred at UFCG, in the Computer Science undergraduate course, in the context Programming 1 course. We will briefly summarize the nature of the studies.

In the study reported on section 5.2, we present a controlled experiment to assess the idea of using proposed metrics to generate code quality feedback and whether students adhere to the idea and improve their code.

Furthermore, the study presented on section 5.3 discusses another controlled experiment to assess the potential of the feedback messages in providing effectively help on code quality improvement. In this study, students in both groups were explicitly asked to submit the best version of their programs according to written directions about code quality. Only students from the experimental group could have access to *qcheck* tool and follow their code quality improvement hints.

In section 5.4, we report what happened during the longitudinal study undertaken with 2017.1 class of Programming 1 course. Instructors of this course requested a summative feedback, based on the whole code production of each student, to be produced automatically using *qcheck*. We implemented this request and used the obtained information to perform an analysis contrasting the summative assessment with students' *qcheck* pattern of usage.

5.2 On the Impact of Code Quality Feedback Generation with *qcheck*

In this section, we will present our first experience on using a set of measures, inspired on software metrics, 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.

By this time, the facility of providing quality feedback generation with *qcheck* was implemented in a modified version of TST automated assessment system differently from the actual integration solution, as a software plugin. We designed an experiment with a randomized sample, providing *qcheck* tool to just one of the two groups. We posed the following research questions:

> RQ1: Do students who receive quality feedback about their submission tend to make more submissions, after the first correct one?
>
> RQ2: Do students who receive quality feedback about their submissions tend to deliver improved quality code?

5.2.1 Methods

We invited students of Programming I course to take part in the study and **45** voluntarily accepted. We randomly divided students into two groups: control and experimental.

Students from both groups received one programming assignment to solve and submit to TST assessment system in **60** minutes. The activity, listed below, is a typical assignment the students are able to solve after being exposed to conditional and repetition control structures' classes. It is based on the well-known mathematician Collatz' conjecture. It asks the student to inform the number of iterations it takes to a given number to converge to 1. The complete specification is presented on Figure 5.1.

Students from the control group performed the activity using the computational resources and the automated assessment system of the course – TST in the usual manner. Students from the experimental group counted on the same usual support but also, had access to code quality feedback messages *qcheck*.

Considere uma sequência de inteiros iniciada em um número natural qualquer e em que cada um dos termos é determinado em função do anterior, pela seguinte regra:

1) se o número anterior, Ni, for par, o número seguinte será Ni / 2;

2) se o numero anterior, Ni, for ímpar, o número seguinte será 3 * Ni + 1.

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

O matemático Lothar Collatz propôs que toda sequência formada através dessa regra, sem importar a partir de qual número natural seja iniciada, converge para 1. Até hoje, essa proposição nunca pode ser comprovada (ou desmentida). Por isso é conhecida como "conjectura de Collatz".

Pede-se que você escreva um programa que imprima a quantidade de números que compõem uma sequência de Collatz, até produzir o número 1. Este número é o que chamamos de "vida" da sequência. Por exemplo, para o número 16, têm-se a sequência: 16, 8, 4, 2, 1. A vida desta seqência é 5.

Figure 5.1: Problem Specification of 'Life Collatz' Programming Assignment.

The messages were presented in the command-line interface, just after the student submits her/his code to automatic testing and receives the results. It was 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. RPEP8 warnings messages were translated from English and slightly modified from the original style checker implementation. The current version of *qcheck* kept the original, messages in English.

It was also added an extra warning message regarding the number of lines of the heading the student is supposed to add in their code. This is an "easy-to-solve" warning aimed at making students learn by themselves how the cycle submit/receive feedback/refactor works. There was an explanation on how to use *qcheck* command but no directions on how to proceed after receiving the feedback message were given during the experiment. It was only advised that those messages could help to improve their code quality.

5.2.2 Metrics

The metric *nsub* is a response variable and refers to the number of functional correct, passed in all test cases, submissions are done by a student referring to a programming assignment

to TST server. The metric *W* is a response variable that summarizes the number of *qcheck* quality warnings on the last correct submission of the programming assignment.

In an effort to investigate those proposed research questions for this study, we formulated the following concrete hypotheses referring to each research question:

> *H1.1 Students who uses qcheck make more submissions(nsub) than those who do not use.*
>
> *H1.0 There is no difference between the mean value of nsub relating to students' group origin.*
>
> *H2.1 Students who uses qcheck presents a lower mean value of W on their final submissions.*
>
> *H2.0 There is no difference between the mean value of W relating to students' group origin.*

The study happened in two computing laboratories so that we could isolate each group of students. The experiment was supervised by teacher assistants and graduate students, part of the Programming 1 course staff, usinf an application script provided by the researcher.

5.2.3 Data Collection

The experiment data was collected by TST. As a premise, only functionally correct submissions were considered to the study. A small amount of 10 students (22.2%) failed the assignment and 35 (77.7%) succeeded.

As shown in Table 5.1, the number of correct submissions after the first correct one that is greater in students of the experimental than the others. It could be observed that the mean value was drawn up because of some individual cases; therefore, we consider that median statistic provides a more reliable measure in this context.

Table 5.1: Number of Correct Submissions.

	N	**Min.**	**1st Qu.**	**Med**	**Mean**	**3rd Qu.**	**Max**
Control	17	1	1	1	1.588	2	4
Experimental	18	1	2	2.5	2.667	3.750	5

5.2.4 Results and Analysis

In RQ1 we investigated if 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 in the experimental group was 2.5; which is greater than the median of submissions performed by control group subjects. We studied this behavior, performing a Mann-Whitney nonparametric hypothesis test. As a result, we rejected the null hypothesis in favor of the alternative. For this test, p-value = 0.009 with 0.05 significance level. This means that 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 taken into consideration by students and not ignored by them. It encourages students to reflect on their code beside it correctness.

In RQ2 we examined if students that receive quality feedback about their submission tend to deliver a better quality code. It was evaluated if quality measurements of the last submissions W of the students' submissions of each group differ depending on their exposition to feedback quality warnings. We have performed the same hypothesis test successfully. It was possible to reject the null-hypothesis in favor of the alternative as 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 the experimental group is lower than the number of quality warnings of the last submission from the students of the control group.

Then, we took into consideration all functional correct submissions of each student in the experimental group, not only the last one. 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 in solving the feedback warning and reduced the number of warnings obtained in relation to the previous submission. Our results indicate 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, at least for this study.

The following Listing 5 shows an example of *qcheck* usage that happened during the experiment. The code on the left is the first (a) and the code on the right is the last (b) correct submissions of a student. Code (a) is the first correct submission of the student. It caused the warning "It appears that your program has too many operations." due to lines 10, 15 and 18. Code (b) is the last submission made by the same student. It caused "No warnings" message. The student "solved the warnings" making a better use of conditional structures and reducing the number of lines of duplicated code.

```
1  # coding: utf-8
2  # xxxx.xxxxxxxx / xxxx / 2014.2
3  # Collatz life
4
5  number = int(raw_input())
6  cont = 0
7
8  while True:
9      if number == 1:
10         cont += 1
11         break
12
13     if number % 2 == 0:
14         number = number/2.0
15         cont += 1
16     else:
17         number = 3 * number + 1
18         cont += 1
19 print cont
```

```
1  # coding: utf-8
2  # xxxx.xxxxxxxx / xxxx / 2014.2
3  # Collatz life
4
5  number = int(raw_input())
6  cont = 0
7
8  while True:
9      cont += 1
10     if number == 1:
11         break
12     elif number % 2 == 0:
13         number = number/2.0
14     else:
15         number = 3 * number + 1
16 print cont
```

Listing 5: Contrast Between Submissions – Before CQI Feedback.

On the other hand, data collected from the control group, reveals a typical student behavior: 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.

5.3 On the Use of Code Quality Feedback Messages

In the previous study, we have observed that students who receive quality feedback with *qcheck* tool tend to deliver a code with better quality, fewer quality warnings. However, we want to better understand if this result occurs due to the feedback we have provided or if it resulted only of the student willingness to improve their code.

In the present study, we contrasted two groups of students that were both stimulated to improve their assignments' code quality after finishing it. The groups were instructed on Programming 1 course code quality standards through a written document we produced and delivered to them. Only the experiment group had access to *qcheck* tool. We contrasted quantitatively and qualitatively the final outcomes gathered from both groups in terms of the number of warnings and what they have changed in their code (churn). As a result, we found that, in fact, students that use *qcheck* are able to make more relevant code changes towards a better code quality.

The research questions that directed this study were:

> RQ1: Do students who receive quality feedback about their submission tend to deliver a better quality code?
>
> RQ2: Do students tend to improve their code quality considering quality feedback?

5.3.1 Methods

Firstly, students were invited to an extra-class activity in order to train them to qualitative evaluation instructors perform in Programming 1. In this activity, they were asked to answer at most 5 programming assignments, from units three to five. In these content units, students are exposed to programming concepts such as conditional structures and iterative structures (for and while loops). The activity lasts the same of a regular laboratory class 120 minutes. It was given to students a written document, which can be found in this thesis Appendix, containing Programming 1 code quality standards and orientation. Students were explicitly stimulated to do the best as they can to improve their code, after making it work. They were divided into two groups experimental and control group and also located in different computing laboratories (LCC2 and LCC1).

During the activity, students from the experimental group could have access to *qcheck* quality feedback warnings. There was a previous pre-training session on how to use install and use *qcheck* with the whole class. Students from control group answered the assignment just like they were used to do, under TST support. The experiment was conducted under the orientation of graduate students that were also part of the course staff. At the end, we evaluated the proposed research questions under quantitative and qualitative analysis.

5.3.2 Metrics

The metric *W* is a response variable that summarizes the number of *qcheck* quality warnings on the last correct submission of the programming assignment. ΔW is the difference between the number of warnings of last correct submission and the first correct submission: ΔW = W_f - W_0. From this metric we came up with three situations:

ΔW < 0: The student was able to recognize and fix aspects in his/her code to improve its quality.

ΔW = 0: The student was not able to improve his/her code quality.

ΔW > 0: The student was not able to improve his/her code quality, instead, the last code submission became worse than the first submission.

Considering the posed research question we established the following concrete hypotheses:

> *H1.1 Students from the experimental group that had access to qcheck messages presented a lower mean value of W on their final submissions.*
>
> *H1.0 There is no difference between the mean value of W in relating to students' group origin.*
>
> *H2.1 Students who use qcheck present $\Delta W < 0$ among their submissions.*
>
> *H2.0 There is no difference on the mean value of ΔW relating to students' origin group.*

5.3.3 Data Collection

We collected a total of 242 programs, from 35 students, 20 from the control group and 15 from the experimental. Since student's participation on the experiment was volunteer, some of them were absent and unbalanced the groups size. We discarded unsuccessful submissions, which are those that did not pass functional tests. This restriction greatly reduced our dataset. Furthermore, many students, despite being stimulated to produce a better version of their correct code in both experimental and control groups, simply do not deliver a different version of their submitted solutions for a given assignment. In this sense, each entry of the dataset is a submitted set of solutions to a given assignment of the experiment. This set is composed only by functionally correct submissions. Submissions must have lines of code that cause relevant difference from each other, for example, blank lines are not enough to make two submissions different. Conversely, comment lines, such as header lines, are considered to be a relevant difference. As we are measuring the evolution of quality warnings, we also excluded from our dataset submissions that accounted zero quality warnings (no warnings).

During the evaluation of the RQ2, we define as an entry in our dataset a pair composed by the first and last submission for a given programming assignment. At the end, the dataset accounted 28 entries, referring to 56 programs. The evaluation of RQ1 was done using only the last correct submission. Table 5.2 summarizes those data.

Table 5.2: Distribution of Quality Warnings Account According to Each Group

	Control	**Experiment**	**TOTAL**
Entries	19	9	28
TOTAL number of submissions	38	18	56

5.3.4 Results and Analysis

In RQ1 we investigated the number of quality warnings on the final submissions according to students' origin group. The Figure 5.2, presents the values of *W* for students from the experimental group are more concentrated and left shifted than the W values from the control group, which are more scattered and right-shifted. This indicates that subjects from

experimental group submit code containing fewer quality warnings - most of them with zero warnings. Furthermore, it can be observed that this behavior is a strong characteristic of the group, as data are more concentrated on the left portion of the plot. Conversely, data from control group are more dispersed, indicating that subjects from data group cannot be characterized by a common behavior. The distribution of "number of quality warnings" from control group presents a wide range of values.

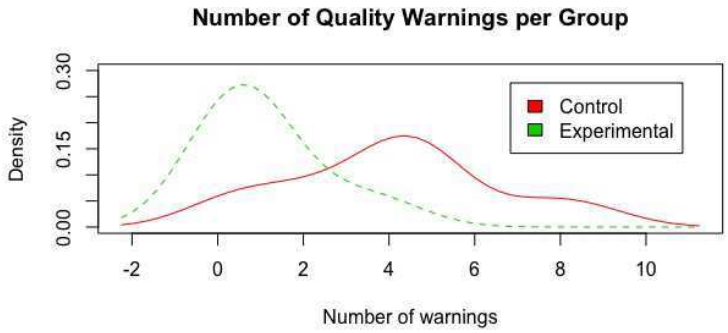

Figure 5.2: Number of Quality Warnings per Group.

Finally, we verified our conjecture expressed on the RQ1 in a nonparametric hypotheses test and found it is statistically valid. We performed Wilcoxon test, our dataset was composed of 28 observations in a non-normal distribution, as found a p-value < 0.001. So we could reject the null hypotheses in favor of the alternative. Our data shows that the mean of quality warnings found on students code in control group is approximately 4.4 and in the experimental group is 1.0. The Figure 5.3 shows the number of warning's distribution in each group. Our conjecture that "students that receive quality warnings during development can produce code with better quality than those who receive only written orientation" finds statistical support in this study.

In RQ2 we investigated if "students tend to improve their code quality considering quality feedback" according to ΔW. In Figure 5.4 the continuous red curve represents ΔW distribution of control group and the dashed green curve represents ΔW distribution of the experimental group.

It can be observed that ΔW values from students of the experimental group are lower than zero, meaning that they were able to recognize and fix aspects in his/her code to improve its quality. We also showed in finer detail, on Figure 5.5, a barplot of the raw data regarding

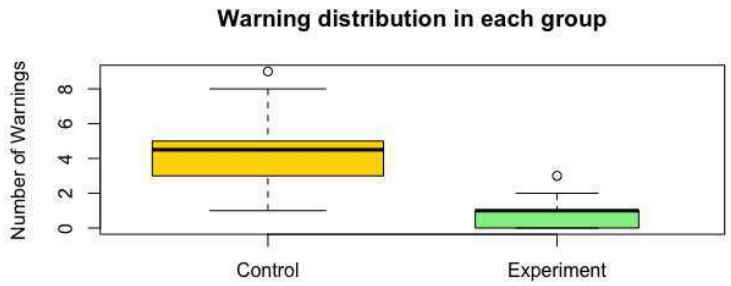

Figure 5.3: Warnings Distribution on each Group.

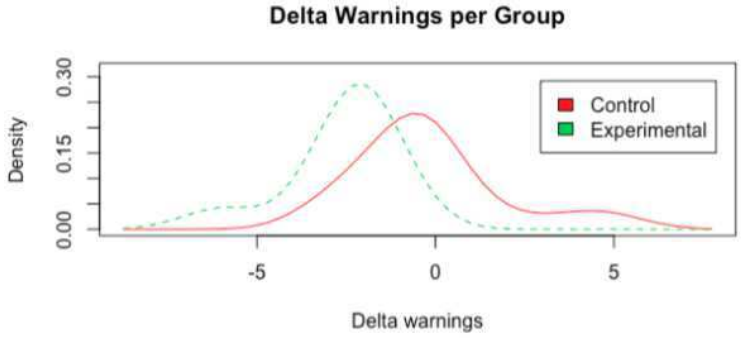

Figure 5.4: Distribution of ΔW According to Groups.

ΔW of each group. Notice that there is not a superposition of the bars.

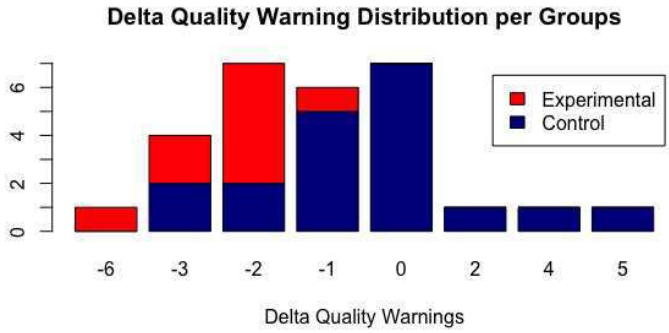

Figure 5.5: Quality Improvement According to Groups.

We also verified our conjecture expressed on the RQ2 in a nonparametric hypotheses test and found it is statistically valid. We performed a Wilcoxon test and found a p-value $<$ 0.001. So we could reject the null hypotheses in favor of the alternative. Our data shows that the mean ΔW of each entry of students code from control group is -0.22 and in students'

code from the experimental group is -2.56. This data shows a greater Delta reduction from students' code of the experimental group. The 95% confidence interval of the difference in mean Delta is between 0.96 and 3.72. Our conjecture that "students that receive quality warnings during development improve their code, using the feedback messages delivered" finds statistical support in this study.

5.3.5 Qualitative Evaluation

We manually evaluated each dataset entry in order to discover what students were thinking when trying to improve their code. We compared the difference on codes of the first and the last correct submission, S_0 and S_n respectively. Then, we classified those code improvements into six categories: code, style, code+style, header, naming and comments. In order to categorize each change we observed the following:

- Code: Includes changes that impact on coding strategy. We find indications of this kind of improvement when we observed changes in the number of lines of code, code statements such as conditional structures and iterations, among others.
- Style: It refers to modifications relating to adherence to coding standards or attempt to make the code more readable including, for instance, spaces before operators.
- Code+Style: This improvement category exists because we found some cases that the code difference includes changes in both categories. We considered being worth to highlight those cases, instead of categorizing it in a single one.
- Header: Changes to include code heading.
- Naming: Refers to changes in variable names.
- Comments: Refers to the addition of comments lines in the code.

We also observed that the number of submissions *nsub* in each group is very similar, in general 2 or 3 submissions, with a few outliers. We credit this behavior to this experiment design: subjects of both groups were stimulated to submit the best possible functional correct code. The Figure 5.6 summarizes code changes executed by students and manually

categorized on 28 entries of the dataset: 9 entries from the experimental group and 19 entries from the control group.

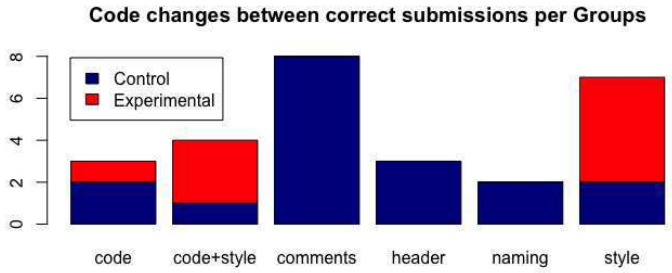

Figure 5.6: Code Changes Categorization per Group.

We observed in this qualitative analysis that students from the control group were not able to act in terms of code improvement. The majority of students in this group tried to improve their code quality including comments to their code. We find it awkward, as the already mentioned quality advice document does not mention comments as a way to improve code quality. Students from control group worried about header and naming while students from experimental group centered their attention in code and style improvements. We can infer that students from the control group felt stimulated and had the genuine intention to improve their code but do not have the correct guidance for that. They lack guidance in what really matters.

On the other hand, students that had access to *qcheck* managed to make significant changes in their code. Their changes in code quality include improvements that impacts algorithm simplicity, better conditional/logical constructions, readability and adherence to coding standards, among others. We believe that the notion of these premises is fundamental to educate and train a good programmer.

Finally, we notice that students who do not receive *qcheck* feedback, even when stimulated to improve their code quality, feel lost and are not able to make significant changes. It appears that *qcheck* act as a relevant guide showing what can be done towards code quality improvement.

5.4 On Producing Summative Feedback

In this study, we will present our efforts on producing summative feedback about students' code production, requested by Programming 1 instructors during the longitudinal study. This feedback information was used to compose a report that was semi-automatically generated using *qcheck*. Students had access to their individual report in TST environment, at the middle of the term, advising about the quality level of their code production until the moment. This was intended to stimulate students to care about code quality issues as they were going to be assessed in this aspect soon. The Figure 5.7 shows some reports examples in TST screen shots.

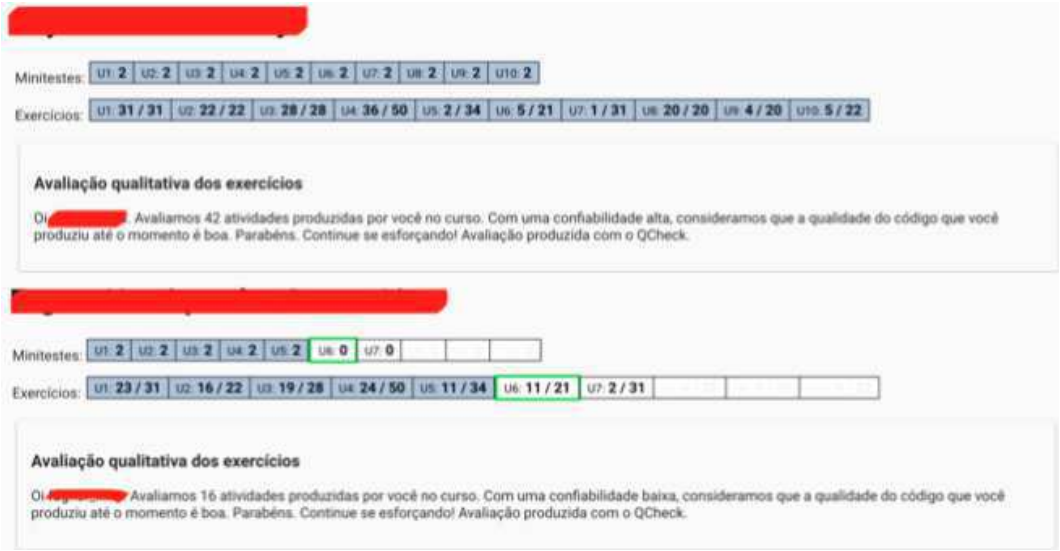

Figure 5.7: Code Production Quality Report Automatically Generated by *Qcheck*.

Besides, we took advantage of these data produced regarding all students to contrast with *qcheck* users to find out if: *"Are qcheck users producing code with better quality than the other students?"*

We conducted this study using two datasets: the first referring to general code production gathered from TST server and the last referring to data collected through *qcheck* usage logs.

5.4.1 Methods

Qcheck was adapted to assess the quality of students code production and summarize the feedback in a grade. This grade ranged from A to D, meaning: *A - Good*; *B - Can improve*;

C - Must improve and *D - Undefined*. This adaptation was challenging and we faced many problems. The first is that students' code production is a skewed distribution: few students solve many assignments and many students solve few assignments. For this motive, care must be taken with this summative feedback interpretation and how to communicate it to students.

In this sense, it was not possible to express the grade considering just one dimension. We proposed to use confidence as a second assessment dimension. This measure refers to the number of programming assignments evaluated to produce such grade. It can be interpreted as how confident we are that a given code production deserves the grade. We defined confidence values as: *null*, *low*, *medium* and *high*. The summative feedback produced can be viewed as a tuple (grade, confidence).

We started the study by fixing a period of 16 days to collect data from TST Server regarding students' submissions to programming assignments from units 3, 4, and 5. Then, we tested all submissions using TST to select only functional correct submissions. Next, we used *qcheck* to compute *W* for each entry (student, activity) and produced a mark for them as a ratio of *W*. We classified this marks in 3 ranges, according to the course intended learning objectives. The final grade (from A to D) was calculated as a proportion of activities in each range. This assessment approach was defined in accordance with Programming 1 course staff. Finally, we created the script to produce a personalized report including message and metadata. The feedback message that was personalized and delivered to 95 students considering grade and confidence dimensions. We here list a translated instance of the feedback we provided to students:

> "Hello *Maria*. We assessed 11 activities that you produced during the course. We consider that the code quality that you produced until now is good, with low confidence. Congratulations! Keep up the good working. Automatically produced with *qcheck*."

5.4.2 Metrics

In order to answer the research question that motivated this study, we have created a concrete hypothesis, considering some metrics detailed below.

RQ1: Do students who use *qcheck* presents a better code production quality then the other students?

H1.1 Students of category "users" presents a better grade of code production quality.

H1.0 There is no difference regarding the category of students on the grade of code production quality.

In a detailed *qcheck* log analysis, we observed that students' usage patterns were very different. Some of them just installed and used one or two times while others used the tool as much as they could. For this reason, we categorized students regarding their proficiency in *qcheck* use. This sounds to be fairer, when conducting whichever study aiming to evaluate *qcheck* effect. A more detailed explanation in how to compute these categories will be presented later on.

In the present study, we want to evaluate the performance of students who are *qcheck* users and compare them to those who do not. We intend to gather evidence that, even if these students do not use the tool they care about quality and produce good code. So, we verified if proficient *qcheck* users achieved the highest grades with adequate confidence in relation to other students, by chance or not with adequate statistical significance.

We categorize students according to their proficiency of using *qcheck*. First, we calculated the correlation and found it significant (Spearman's rho 0.97 with p-value < 0.01), between the occurrences of *qcheck* use with the number of activities solved by students. Then, using the distribution of the number of activities solved in the period we classified students' *qcheck* proficiency in a categorical order as users, testers, curious, non-users. The category is used as a factor in this study.

The metric students' grades, a response variable of this study, varied from A to D, meaning: A - Good; B - Can improve; C - Must improve and D - Undefined. The Figure 5.8 presents a practical example of this grade as the methodology to calculate it was previously described. It shows the assessment of the code production of two students. Each cell represents one programming assignment. The mark is a ratio considering W. Ranges depicted by different colors in the image, relates to course pedagogical assessment decisions. The final grade is a proportion of the activities in each range. Distribution (a) was graded as A and

distribution (b) was graded as C.

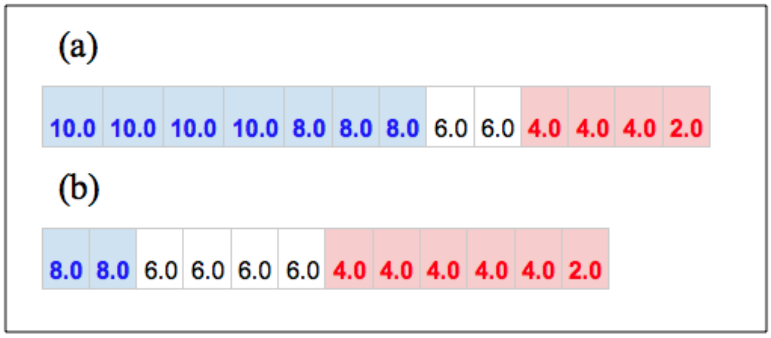

Figure 5.8: Assessment of Students (a) and (b) Programming Assignments.

The *confidence* metric ranges in four categories, regarding the number of programming assignments completed by the students: null < 10; low < 20, medium < 30 and high >=30. Although these numbers might appear too high, we would rather keep high standards in this aspect. It means that our approach is conservative: A high confidence in the assessment is only possible if we evaluate many assignments. For this reason, we pruned the dataset and discarded entries with null *confidence*.

5.4.3 Data Collection

The Table 5.3 presents a summary of the data collection of code production quality assessment and *qcheck* usage. Note that, regarding users' proficiency, those in "Not" category are students that did not use *qcheck* in the period of the study. It differs from "Non-users" as this group used *qcheck* only once in the period.

Table 5.3: Dataset Summary.

Total of students 47

User Proficiency	User	Tester	Curious	Non-user	Not
	17	5	8	5	12
Confidence	High	Medium	Low		
	7	12	28		
Grade	A	B	C		
	23	19	5		

5.4.4 Results and Analysis

Observing the data collected, (grade, confidence), in order to evaluate the research question we found that all (100%) students (A, high) are categorized according to *qcheck* proficiency as *users*. If we consider a broader confidence, keeping the same grade (A, high + medium) we found that 83.3% of the students are categorized as users and 16.7% as not.

In order to evaluate the whole distribution and gather statistical evidence that it did not occur by chance we produced the contingency table and tested this hypothesis. As a result, we found it relevant and could reject the *null* hypothesis in favor of the alternative that in fact there is a difference between the distributions. We applied Pearson's Chi-squared test to assess whether the differences in the distributions happened by chance. The result was relevant with df = 8 and p-value =0.008. The contingency table 5.4 used in this test is presented in sequence.

Table 5.4: Contingency Table Contrasting Students' *qcheck* Usage Proficiency to Grades.

	A	**B**	**C**
User	13	4	0
Tester	2	3	0
Curious	4	3	1
Non-user	0	5	0
Not	4	4	4

5.4.5 Discussion

In this study, we conjectured whether students who were *qcheck* users perform better on the assessment of their code production, in a given period of time than others student that did not use the tool. We came to very positive results observing and testing the data distribution we collected. However, care must be taken while interpreting these results due to some bias that could threaten its validity.

Confidence metric is directly related with the number of assignments the student solves. It perhaps may be an indication that these are students most motivated or better performing in the course. On the other hand, we collected data from programming assignments regarding

3, 4 and 5 units and top performing students in the course have passed by this units solving few assignments.

Another issue is that this result must not be considered a final evaluation about code quality in students' production. They were not warned that we were going to collect their submission in a given period to assess them. In fact, students have contacted us after the report release questioning that they had produced much more activities than it was assessed. We advised that this report was just a checkpoint and is not intended to provide a final diagnostic, but to warn students about how they are performing.

In fact, 49 students with *null* confidence were excluded from the study. It means that 51.4% of the students have not completed enough programming assignments, by the time we performed the study, to have their code production qualitatively assessed.

Chapter 6

Consequences of Code Quality Improvement Feedback on the Learning of Programming

In this chapter we consider the consequences to learners of providing feedback about code quality improvement using the proposed tools during a programming course. Recently, De Nero and colleagues (20017) discussed that there were many initiatives and advances in automated feedback platforms aimed at programming education, but few studies on its effects on real programming courses [DSPQ+17]. We are going to present our main findings in a longitudinal study and discuss its implications to learners, instructors and the course itself. Furthermore, we are going to present an evaluation of the approach performed with the students that used the tool in their activities.

In order to evaluate how students consume code quality improvement feedback and how it is effective in helping them to improve their code, we proposed a longitudinal empirical study. At this point, we have already proposed, implemented an adjusted *qcheck* as a proof-of-concept instrument that was plugged into TST, the automated assessment system used in our introductory programming course at the UFCG. The theoretical goal of this study was to find evidence that could help us to answer the research question:

> *"How learners that used qcheck increased knowledge about code quality and transferred it to their programming practice?"*

However, answering this question and finding a causal link between the use of the instrument and its repercussion in ones' knowledge it's impracticable, at least in the scope of this work. For this motive, we rephrased this research questions into more modest ones, in order to gather empirical evidence of the repercussion of our proposal on students programming practice and skills along a period of time in their introductory programming course. The following research questions guided the studies reported in this section:

> *RQ1: Do learners incorporate "code improvement" as part of their programming process cycle?*
>
> *RQ2: Do learners that consume qcheck code quality feedback is succeeded in improving their programming assignments code quality?*
>
> *RQ3: Do learners that consume qcheck code quality feedback improve their programming abilities regarding code quality?*
>
> *RQ4: How is students' perception about qcheck usefulness in the aid of improving programming assignments' code quality?*

6.1 Context

This study took place at UFCG in the Computer Science undergraduate course in Programming 1 course under the supervision of our advisors. They were also part of the academic staff the course that was composed by 4 instructors and 15 graduate and undergraduate students that provide support to the course as teacher assistants or students' tutors. It happened in the academic period of 2017.1 with an enrollment of 115 students. This is the official number, which includes those students who have dropped out the course later on.

6.2 Evaluation of Providing Code Quality Feedback in a Programming Course

In order to evaluate how students consume *qcheck* feedback and how effective it is in supporting code quality improvement, we took into consideration two outcomes: the

programming assignment code and the student that used or not *qcheck* tool when solving programming assignments.

6.2.1 Methods

The nature of this empirical study aimed at contrasting and exploring the consequences of using *qcheck* on the introductory programming course is quantitative. However, in order to unveil relations and results, it was necessary to recur to qualitative analysis of students programs and behaviors. In general, this study had the following characteristics:

- Analytic - as it uses statistical methods to uncover relations and make inferences;
- Longitudinal - as we collected a set of measurements along time of the study;
- Prospective - as students under the research will be followed by their exposure to *qcheck* to the outcome (response variables);
- non-Randomized - in fact, we did not intentionally divide two groups of study, but it naturally happened when students chose to use or not the instrument. Certainly, *qcheck* users' group is biased by their willingness to invest in code quality improvement. For this motive, care must be taken when reporting conclusions about the hypothesis that contrast users/non-users groups.

In order to investigate each research questions mentioned above, we made a set of concrete hypothesis considering some metrics and the data collection context. We also used descriptive statistic in an exploratory analysis to report data about the observed phenomena.

RQ1: Do learners incorporate "code improvement" as part of their programming process cycle?

> *EA1.1 What was the proportion of students that used qcheck to improve their code quality?*
>
> *EA1.2. How students that used qcheck behaves in terms of frequency of its use?*

RQ2: Do learners that consume *qcheck* code quality feedback is succeeded in improving their programming assignments code quality?

EA2.1 Among students that used qcheck what is the proportion of students that managed to reduce their W?

EA2.2 Among students that used qcheck what is the proportion of students that managed to reduce to 0 (they received the message "No warnings, congratulations!") their W?

H2.1.1 The value of W in the code of the last submission is smaller in students that consumed qcheck feedback.

H2.1.0 The value of W in the code of the last submission does not depend if students consumed qcheck or not.

H2.2.1 Students that uses qcheck presents $\Delta S<0$ among their submissions of the same assignment.

H2.2.0 The value of ΔS among submissions of the same assignment of students that uses qcheck are not necessarily less than 0.

H2.3.1 Students that uses qcheck presents $\Delta W<0$ among their submissions of the same assignment.

H2.3.0 The value of ΔW among submissions of the same assignment of students that uses qcheck are not necessarily less than 0.

RQ3: Do learners that consume *qcheck* code quality feedback improve their programming abilities regarding code quality?

H3.1.1 The value of S in the code of the last submission is less than in students that use qcheck.

H3.1.0 The value of S in the code of the last submission does not depend if students use qcheck or not.

H3.2.1 The value of W in the code of the last submission is less than in students that use qcheck.

H3.2.0 The value of W in the code of the last submission does not depend if students use qcheck or not.

6.2.2 Data Collection

We used two collections of data to investigate the proposed hypotheses of this study. The first one was the data gathered using *qcheck* log bot and saved in our server from the period 2017-05-29 to 2017-06-26. We filtered this data in order to prune the distribution. In the first three days of study, when the tool was presented to the students, it was proposed an activity to teach them how to use *qcheck*. During these days, almost all students used *qcheck*, as it was a laboratory activity. After that, using *qcheck* was no longer asked or incentive by the course instructors. So, as it can be observed in the Figure 6.1, the number of *qcheck* occurrences of use declined.

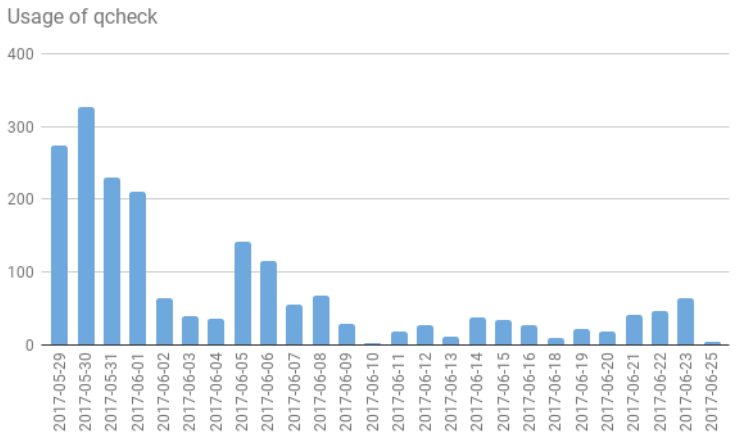

Figure 6.1: Aggregated Number of *Qcheck* Use by Students by Date.

The distribution is skilled revealing an abnormal pattern of usage at the beginning of data collecting. It is worth to note that these data refer only to students that used *qcheck* at least once during this period. Another unusual pattern of usage was detected at 2017-06-26 when happened the first exam to assess students' code quality as an official activity the course. Students could use *qcheck*, if they wished to. As this day, we registered more than 800 occurrences of *qcheck* invocation. We omitted this data in the plot and excluded them from our analysis of *qcheck* usage.

The second data set was collected from TST server and is composed by submissions done by students during the same period of time. Differently from the first mentioned data set, this one contains data from students that used or not *qcheck* tool. However, it contains only the last functional correct submission of each student for a given assignment. In summary,

Table 6.1 presents a summary of the data set.

Table 6.1: Students Code Production Data Collection.

Summary

Users		96
Activities		85
Units		3, 4 and 5
Used qcheck	Yes	464
	No	1030
Consumed qcheck feedback	Yes	356
	No	1138
Total number of submissions		1494

6.2.3 Metrics

The metric *W* is a response variable that summarizes the number of *qcheck* quality warnings on the last correct submission of the programming assignment. ΔW is a difference between the number of warnings of last correct submission and the first correct submission: ΔW = W_f - W_0. Similarly, ΔS is a difference between the number of style warnings of last correct submission and the first correct submission: ΔS = S_f - S_0. We expect these values to be negatives.

6.2.4 Results and Analysis

In order to investigate whether learners incorporated 'code improvement' as part of their programming process, we first searched what proportion of students used *qcheck* during the study (EA1.1). We took into consideration that the initial 3 days of data collection corresponded to a specific activity to install and present *qcheck* to students, then we found that more than 66.3% of students used *qcheck* in theirs everyday activities while 33.7% did not.

Among those students that used *qcheck* in their routine, we found, as could be expected, that the number of *qcheck* invocations (occurrences of use) are strongly positively correlated

to the number of programming assignments (activities) the student did. We used Spearman's rank correlation method since it is less sensitive to outliers. The value of rho is $= 0.87$, p-value=2.2 x 10_{-16}.The Figure 6.2 shows this distribution.

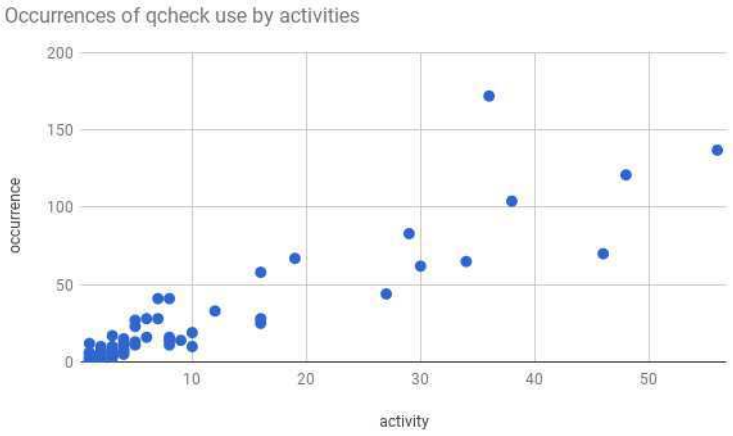

Figure 6.2: Occurrences of *Qcheck* Uses X Activities Performed by Students.

Furthermore, the pattern of *qcheck* usage is diverse (EA1.2). We categorized students, according to the number of activities they submitted and used *qcheck* to improve its quality, as: users, testers, curious and non-users. Considering that occurrences of use and activities are strongly correlated, we found that number of activities was more significant to make this categorization, observe a summary is shown on Table 6.2. Students in the category of users are those that the number of activities performed using *qcheck* tool is above the 3rd quartile, referring to 30.4 %. Students in the category of testers are above the median and below the 3rd quartile, referring to 21.7%. Students in the category of curious are 26.1% that are students that submitted fewer activities than the median, but more than the 1st quartile. The category of non-users corresponds to students that installed, tested and used *qcheck* but did not incorporate it into their programming process routine as they used when performing just one or two activities. It differs from those 33.7% proportion, previously mentioned, that did not use the tool (besides in activities proposed by instructors) at all.

In the second research question (RQ2) we wanted to evaluate if "Learners that consume *qcheck* feedback is succeeded in improving their programming assignments code quality". Initially, we explored data collected with *qcheck* log bot in the longitudinal study to know what was the proportion of students that used *qcheck*, receive its feedback and managed

Table 6.2: Students Usage Pattern of *Qcheck* Relating to Occurrences and Activities.

	Min	**1* Quartile**	**Median**	**3* Quartile**	**Max**
Occurrences	1	5	11	27	172
Activities	1	2	4	8	56
CATEGORY	Non-user	Curious	Tester	User	

to reduce the value of metric W among their programming assignments, $\Delta W < 0$ (EA2.1). We found that 63.04% reduced the number or quality warnings of its codes, 32.4% did not increase nor reduced the value of W and a small amount of 4.4% increased the value of W. It is important to notice that, since the tool runs in the client side, it means that an unsuccessful attempting in improving their code quality does not compromise students code assessment. They only submit to TST server the best version of their code. The Figure 6.3 shows the frequency of distributions of the value of ΔW.

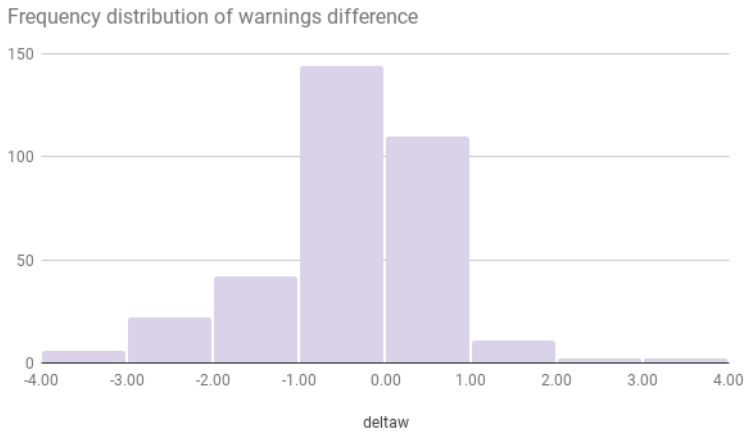

Figure 6.3: Distribution of ΔW per Assignments.

Furthermore, we looked for students that managed to eliminate the warnings emitted by *qcheck*, so that W = 0 in their activities (EA2.2). Considering 618 assignments (user, activity) in our dataset, we observed that, 24.75% of the students that used *qcheck* tool were able to accomplish "No warnings" at their first attempt. A proportion of 25.56%, consumed *qcheck* feedback and managed to also accomplish W = 0. A considerably large number of students, 20.37%, just checked the quality feedback and 29.28% of them were not capable to eliminate the warnings Table 6.3.

Table 6.3: Contrasts of Final Number of Warnings According to *qcheck* Usage per Assignment.

	W = 0	**W >0**	**TOTAL**
Checked quality feedback	153	126	279
Consumed quality feedback	158	181	339
TOTAL	311	307	618

In order to better understand this post-feedback behavior, we contrasted these results with the categories of *qcheck* users. We found that "users" and "testers" are more proficient in consuming quality feedback message and improving their code than other categories of users, as can be seen in the boxplot in Figure 6.4. Recall that "users" and "testers" are those students who have more occurrences of *qcheck* use. It can represent a signal that students in these categories have learned "how to use *qcheck* to improve their code".

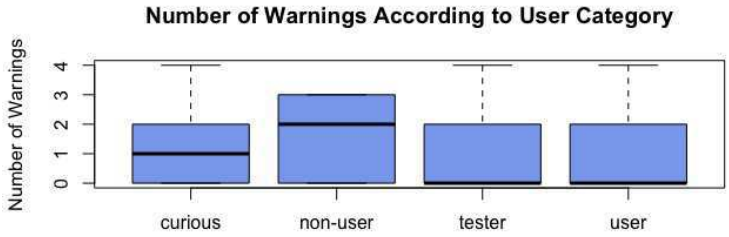

Figure 6.4: Number of Warnings According to *Qcheck* User Category.

We also tested and found significantly, the hypothesis that the mean of W on assignments depends on the use of feedback (if it was just checked or consumed). We reject the null hypothesis in favor of the alternative that there are differences in terms of the mean location of W (response variable) in the distribution (feedback "just checked" or "consumed") executing the Wilcoxon signed rank test, with p-value < 0.001.

H2.1.1 The value of W in the code of the last submission are smaller in students that consumed qcheck feedback.

H2.1.0 The value of W in the code of the last submission does not depend if students consumed qcheck or not.

We investigated if the derivative of metrics W and S (ΔW and ΔS) in this study performing some statistical tests. We also made some assumptions about the ability of learners in consuming feedback and improving their codes. We hypothesized that ΔW and ΔS tends to be smaller than 0 in the observations of the distribution, meaning students were able to reduce the initial value of W and S on the final submission. The concrete hypothesis H2.2 used to perform the statistical test is presented in sequence. We omitted H2.3 because is similar H2.2, but using ΔW.

> *H2.2.1 Students that uses qcheck presents $\Delta S<0$ among submissions of the same assignment.*
>
> *H2.2.0 The value of ΔS among submissions of the same assignment of students that uses qcheck are not necessarily less than 0.*

We computed ΔS on the distribution and found that 56.05% reduced the number of S, which is the style warning. A proportion of 40.41% did not increase nor reduce the value of S and few students, 3.54%, increased the value of S. We executed statistical tests that were capable to support our initial claims for both variables. For both variables ΔS and ΔW, using Wilcoxon signed rank it was possible to reject the null hypothesis in favor of the alternative, with p-value «0.001.

Finally, in the last question (RQ3) we examined if "learners that consume *qcheck* code quality feedback improve their programming abilities regarding code quality in relation to those who have not used". Until this point, we have only dealt with students that executed *qcheck*. Certainly, a considerable proportion of students in this group has the willingness to improve their code. Although, we have observed that not all of them could accomplish it. In this final study, we are going to contrast metrics *W* and *S* from students of the whole class: that used or not *qcheck* in their final submission to TST Server.

We observed in this sample distribution, of given period of time in Programming 1 course, that 31.05% of students have used *qcheck* in when doing their assignments, while 68.95% did not. From the amount of those who used *qcheck* 76.72% consumed the quality feedback, meaning that they executed *qcheck* more than once or obtained "No warnings" in the first execution. Considering the mean of W and S metrics as response variables and qchecked and consumed as factors we have the following data presented in Figure 6.5, Table

6.4 and Table 6.5.

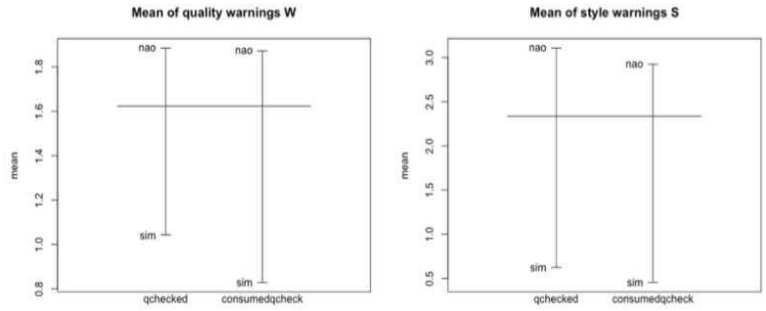

Figure 6.5: Mean of *W* and *S* According to *Qcheck* Use.

Table 6.4: Value of the Mean of Quality Warnings – **W**

Mean of W		Consumed *qcheck* feedback	
		No	Yes
Used *qcheck*	No	1.88	-
	Yes	1.75	0.82

Table 6.5: Values of the Mean of Style Warnings – ***S***.

Mean of S		**Consumed *qcheck* feedback**	
		No	Yes
Used *qcheck*	No	3.10	-
	Yes	1.18	0.45

We conjecture that students that used *qcheck* tool might improve their programming abilities regarding code quality in relation to those who have not used that used the tool. In order to investigate the validity of this conjecture, we proposed the hypotheses H3.1 and H3.2, which are similar and refers to *S* and *W* respectively. We recall only the first one here:

> *H3.1.1 The value of S in the code of the last submission is smaller in students that use qcheck.*
>
> *H3.1.0 The value of S in the code of the last submission does not depend if students use qcheck or not.*

It was observed the mean value of S and W, as response variables for students that used or not *qcheck*. The results of the hypotheses tests we have used to investigate those conjectures were significant for both variables. It was used a nonparametric test as we cannot observe normality in the distribution. Its results yield that we can reject the null hypothesis in favor of the alternative that there is a difference on the mean (mean_sim < mean_nao) of the distributions. As we previously discussed in this section, if we considered students that effectively processed *qcheck* feedback (variable *consumed qcheck*), we would observe even greater impact on the value of W and especially on S.

We deeply evaluated the relations among the factors and their effects on the response variables W and tried to adjust a model to observe the statistical significance of them. As the variable W represents an observed count (normalized number of warnings ranging from 0-5) and we could not guarantee the assumptions of normality for a trivial linear regression model. For this motive, we fitted a generalized linear model with Poisson distribution.

We took as explanatory variables: *consumedqcheck*, *unit*, lloc, *cc*, *vhalstead*, utype. As a result, we found that the intercept and the values of *consumedqcheck*, *lloc* and *utype=user* were highly significant for the value of W (p-value approximately 3 X 10-5). The final model could be written as:

W= 0.510 - 0.660 consumedqcheck + 0.020 lloc - 0.275 utypeuser

A free interpretation of the model coefficients, at least for our dataset, may represent that:

- When consume *qcheck* feedback, we observe an decreasing W value;
- Longer programs impacts the number of defects (lloc positive), we observe an increasing W value;
- When the student is a *qcheck* proficient user (utypeuser), we observe decreasing *W* value;

Other interesting conclusions can be observed regarding the coefficients that were not relevant to the overall explanation of W value. The factor "unit of the programming assignment" that could be related to additional difficulty on the programming level and also on the programmer maturity was not relevant for W observed value. It possibly gives us

an indication that code quality is a transversal concern, at least in these initial units of an introductory programming level.

It is worth to note that when we use a Poisson regression model, we assume that the response variable follows this distribution and so, that the mean and variance are equal. In these data, for the response variable W, mean and variance estimates were approximately 1.62 and 1.86, respectively. Thus, we have no reason to suspect about Poisson modeling.

We have also modeled *S* variable using the same methodology, however, results were not acceptable.

6.3 Do Learners Think that *Qcheck* is Useful?

The aim of this study was to gather observations and comments of students that were using *qcheck* tool in order to evaluate its experience during their programming process. The research question raised by this study was:

> RQ4: How is students' perception about *qcheck* usefulness in the aid of improving programming assignments' code quality?

We want to better understand student's post-feedback behavior from their own perspective: (1) how students use *qcheck* in their programming process; (2) how students consume *qcheck* quality feedback; (3) What preclude or motivate them to use *qcheck* and (4) If students agree that their code is effectively improved if they follow *qcheck* hints.

Firstly, we selected and informally interviewed students during their laboratory classes. Then, we asked them to register our conversation and other important information in a form. They were asked to answer a questionnaire composed by almost the same questions but in a more structured way. Next, we processed the answers, coding and categorizing the answers for each posed question. Lastly, we summarized the results for each topic and drew conclusions about students' perception of the usefulness of *qcheck*.

6.3.1 Methods

In this study we interviewed a set of students that voluntarily used *qcheck* in their programming process during laboratory classes. After three weeks of in situ observation, we

started the fourth week remotely monitoring *qcheck* usage log. This distance was intended to reduce researcher's influence on the natural order of events. When we detected an occurrence of *qcheck* use, that met our requirements, we 'rushed into the scene' in order to get fresh impressions about the event.

We adopted a research methodology inspired by as 'firehouse research' [BBBL15]. In this methodology, the process begins with the researcher 'at-the-ready', prepared to act as soon as the event starts or happens. Researcher's actions are driven by the events not by him or her. It requires a careful design plan previously defined, monitoring and controlling of tool's usage and automated subjects' selection. It requires less formalism; in contrast with others social science research methods, but more automation and readiness. Another positive aspect of firehouse research methodology is the way it transports researchers to subjects' reality. In our study, we adapted this methodology as we included a questionnaire to the subjects themselves register their information. While this can hinder students from speaks freely about their impressions on the subject it gives us the agility to talk to others students at the same class and avoid the step of interviews transcriptions.

6.3.2 Participant Selection

The process of participant selection for this study was based on two aspects: the student and the event - an occurrence of *qcheck* use.

We chose students considering their temporal distribution on each one of the four laboratory classes of Programming 1 course: T1, T2, T3 and T4 (two of them occur at the same time). This was intended to maximize the number of respondents and mitigate biases regarding the student class and instructor.

Furthermore, in the beginning of the course students were advised that they would take part in a research; they may opt to decline or accept. Also, during *qcheck* tool installation they are asked if they want to report or not data for research purposes. If they accept to report their data, they are warned that are contributing to our research. According to our dataset, only 4.2% students declined to report data. We can only count on participants who accepted to take part in the research.

We also prioritized selecting users from different clusters based on their code quality level. We took into consideration students' code production and performed a careful analysis,

using *qcheck* tool, on their assignments submitted for TST server. We adapted *qcheck* to produce a summative feedback and using this information we categorized students into three groups based on the level of quality of their code production, until this moment. Ideally, we wanted to select students at least from each one of the three clusters.

In regards to the event *qcheck* use, we set the following criteria:

- It needs to happen on the course of a laboratory class;
- It must be observed at least two occurrences of *qcheck* invocation;
- The number of warnings (style and code warning) must be zero on the last invocation.

The first two student's criteria were established for convenience: as we know where the student is. We contacted students during laboratory classes, upon an agreement with the teacher. Laboratory classes of T1, T2, T3 and T4 occur on the first three days of the week, so we concentrated our interviews activities on those days. The second criterion was fixed to guarantee that it happened at least one cycle of feedback: (1) report-feedback, (2) feedback-consumption and (3) code-refactoring. Finally, we fixed, as a tiebreaker, an ideal event: students managed to solve all warnings. Certainly, if we only choose participants that can meet this last criterion, our results would be biased, and the contrary is also true. So, we were careful to ensure that we have a balanced selection.

After the subject selection, the interviews occurred sequentially. During laboratory classes, we remotely monitored *qcheck* activity log aiming to find a pair (student+event) that meet those specified criteria. When it happens, we contacted the student, asked if we can talk about their *qcheck* usage and ask them to answer the online questionnaire with the same discussion.

The Figure 6.6 lists the questions we have discussed with the study participants in order to gather elements to answer the research question (RQ4). The main topic is listed in the first line and in sequence, the question elaboration is maintained in Portuguese to show what was effectively asked to the students. The instrument used is in the Appendix.

RQ4) How is students' perception about qcheck usefulness in the aid of improving programming assignments' code quality?

1. How students use *qcheck* in their programming process?
- Como ou em que momento você está usando o *qcheck* nas suas atividades de programação?
2. How students consume *qcheck* feedback?
- Como você usa ou processa a mensagem fornecida pelo *qcheck*?
3. What preclude or motivate them to use *qcheck*?
- Quais sãos os pontos positivos e negativos do *qcheck*? - O que impede ou motiva você a usar a ferramenta?
4. Do students find that their code can be improved following qcheck hints?
- Após a resolução dos warnings do qcheck, seu código se apresenta: Pior que a primeira versão… Melhor que a primeira versão (Likert scale answer allowed)

Figure 6.6: Qualitative Evaluation - Questions of the Interview and Questionnaire.

6.3.3 Data Collection

At last, we interviewed 19 students selected using the reported approach and other different reasons. For example, some students asked to take part in the research during the laboratory class. We included these students because they already passed the units allowed to use *qcheck* tool and they could not be selected looking at *qcheck* log. The final subject selection included students from a diverse profile according: to laboratory class provenance, experience with programming, performance on the course, grade of qualitative evaluation regarding code production and proficiency of *qcheck* use, as can be seen in Table 6.6. We naturally preferred users with a higher level of *qcheck* proficiency so they could assess it more properly.

6.3.4 Results and Analysis

The first topic was intended to verify if students were including *qcheck* quality verification correctly on the programming process: on the fourth stage - "Look back". In the proposed approach, students are encouraged to verify their code quality after their program verification. In Programming 1, students can tell their program is functionally correct when it passes public and secret TST tests.

In response to (Q1) "How students use *qcheck* tool in their programming process?" the majority of the respondents 47.35% answered that they used after TST server tests when the program is correct. A proportion of 31.6% answered that they execute *qcheck* before

Table 6.6: Data Set of Qualitative Users' Study.

Student	**lab**	**newbie**	**unit**	**grade**	**proficiency**
S1	1	yes	4	A	user
S6	1	yes	8	A	user
S15	1	yes	4	B	user
S2	2	yes	5	A	user
S5	2	yes	5	A	user
S8	2	yes	10	A	tester
S16	2	yes	9	A	user
S17	2	yes	4	A	user
S3	3	yes	4	B	user
S4	3	yes	7	A	user
S7	3	yes	10	A	user
S11	3	yes	7	A	user
S12	3	yes	8	D	tester
S13	3	yes	10	D	tester
S18	3	yes	9	A	user
S9	4	yes	6	B	user
S10	4	no	4	A	non-user
S14	4	no	4	D	curious
S19	4	no	4	D	user

submitting the code to the server. This can concern us, as it means that students do not know the basic recommendation about the stages of programming process and how to include quality improvement on it. It also can be the cause of students' alleged difficulties in solving the warnings - if the code is not functionally correct the metrics used by *qcheck* to generate code warnings are not valid. The rest of 21.05% of students answered this question with other information. Curiously, student S8 misunderstood the question and revealed "where" he used the tool *"Only when I am solving programming assignments at home"*. This may corroborate to other arguments raised in subsequent questions that it takes a time to improve the code.

In the second topic, we wanted to investigate the quality of the feedback messages and how students process them. *Qcheck* feedback message is divided into two parts: code warnings and style warnings. Style warnings hints are composed by a message and the numerical indication of column and code line where the problem can be found. Code warnings are those related to the program structure and the algorithm used to solve the problem. Sometimes, to solve those warnings it is necessary to reason about other solution - a better algorithm. For this motive, solve this type of warning is more difficult. Possibly a good working strategy is to start solving style warnings and then proceed to solve code warnings.

In response to the topic raised by (Q2) "How students consume *qcheck* feedback?" we noticed that 31.6% of the respondents use the strategy of solving code warnings first. Some of them argued they proceed this way, as style warnings are easier to solve. Only one student answered that he starts solving code warnings. The other responses address diverse issues. Student S9 manifests his concern in *"fixing the quality warnings and keep the code still working"*. Another student (S17) asserts "*There were cases that I left unsolved warnings because it was more difficult to solve them than to solve the programming assignment"*. We agree that solving code warnings are indeed more difficult than solving style warnings as it needs more reasoning about the solution recruiting more depth cognitive work.

In the third topic, we intended to understand what precludes or stimulates students in using *qcheck* tool. The answer to this question may be evaluated in a twofold perspective: the tool and the task. The first and more direct is the tool evaluation: if *qcheck* meets the requirements of their users. The second is the task: if students felt stimulated to invest their

time and efforts in improving code quality in the context of programming 1 course.

In order to investigate this topic we have formulated two questions:

(Q3.1) Which are the positive and negative aspects of qcheck tool?

(Q3.2) What precludes or stimulates you to use qcheck tool?

Evaluating Q3.1 answers we found many interesting and revealing views. As positive aspects, 42.1% of students evaluated the tool as *"an excellent way to help them to improve their code"*. Other 21.05% argued that it stimulates good programming practices. Other students reported that the process stimulates them to reason about other ways of solving the problem. It was also mentioned twice that it raises students' confidence that they are producing good code. Student S10 declares that as a positive aspect of *qcheck* is that *"it stimulates you to create new ways to solve the same problem, though we learn deeper"*. Regarding negative aspects, 21.05% of the respondents reinforced that there is no negative aspect. Three students sustained that *qcheck* did not consider their solution as a valid one and "complained" about their chosen programming structures. We observe that students feel that the system could provide a better support for the refactoring process itself: *"needs to improve integration with TST"*, *"it is difficult when we change the code to solve a warning and break it"* and *"somehow show the references on how to improve code warnings"*. A student that related (anecdotally?) OCD – obsessive-compulsive disorder – about correctness, reported frustration when keep receiving warning messages and could not get rid of them. To sum up, we claim that it is possible to fine tune to *qcheck* warning generation, to make it more resilient to students' solution. However, we propose it as future work in Chapter 8, as we consider that is necessary to evaluate the motivations to make these parameters changing and ground it with an experimental evaluation using evidence-based approach.

Evaluating Q3.2 answers we found as the main motivation to invest in quality improvement, on 36.84% of the respondents, is the *"willingness to write code clearer, cleaner and more efficient"*. Other 15.8% of the students reported that they felt stimulated observing *"the results"* they have obtained using the tool. Student S2 reports *"a visible improvement in his code"*. Interestingly, S17 posed that what motivates him is the feedback message *"No warnings! Congratulations!"*, reinforcing the motivational power of the feedback. Few impediments to *qcheck* usage were reported. The most relevant declared were: *"quality*

accounts on 10% of our grade" and *"the time spent on code improvement"*. Three students answered, regretting, that what preclude them to use *qcheck* is *"it is unavailable in higher units of the course"*.

In the last topic, of this evaluation we intended to understand if students considered that their code is effectively improved after their actuation following *qcheck* hints. We asked the following question to the students:

> *(Q4) Do you think that final version of your code, after solving qcheck warnings, is worse or better than the first version?*

This question was answered using a Likert scale ranging from 1 - worse than the first version to 5 - better than the first version. The responses are shown in the following Figure 6.7.

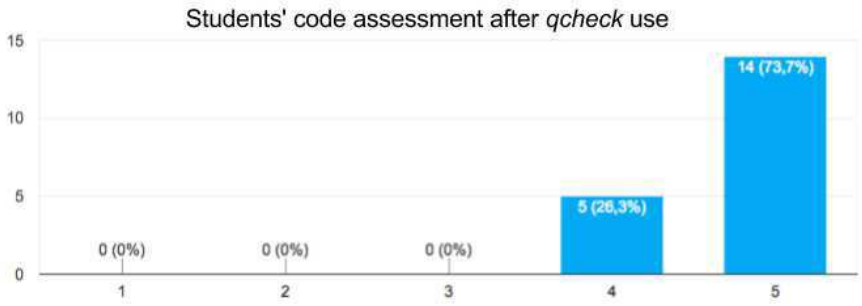

Figure 6.7: Students' Perception About Code Improvement Directed by *qcheck* Hints.

6.3.5 Discussion

We highlight student S16's testimonial about his experience with *qcheck* on the course as this discussion starting point. His opinion summarizes many observations of other students collected in the study and raises other concerns:

> *"The most important thing about qcheck is that it leverages the 'learning of programming' to the 'learning of good programming'. I found that the experience of learning how to program in a system that is explicitly worried about producing code with quality was more fulfilling than the experience of 'what is important is that your code works'. However, I think that in*

> *Programming 1 classes it must be emphasized the importance of the skills such a tool helps students to develop. Many students of my class do not feel that using qcheck it's important due to this lack of incentive."*

In fact, we realize that having code quality improvement on the list of concerns of an introductory programming course is challenging not only to students but also to instructors. In order to be successful, we perceived that including code quality improvement refactoring activity needs to be part of the pedagogical orientation of the course. Furthermore, repeating the study to improve the tool according to quantitative and qualitative evaluation is necessary. Consequently, the overall proposal can be refined and improved.

In general, we gathered very positive evaluations and good suggestions on how to improve the tool. It was important to observe that some students were not using *qcheck* as we expected: after the program is functionally correct. Also important, it was to hear from them what precludes them to use the tool. Some affective issues were reported, both negatives "frustration and anger" as they cannot overcome their difficulties and positives "joy" when they obtain the message 'No warnings. Congratulations!' and "hope" to become a better programmer.

Chapter 7

Discussion

In this work, we presented an approach to generate formative feedback to leverage programming problem-solving in the last stage of the programming process: targeting the solution evaluation. As long as solving programming assignments plays a central role in learning the skills of programming, we focused our attention on this task. This research results can be applied on introductory programming courses supported by automated assessment systems to programming assignments. As a requirement for generating feedback, we compromised not to impose the creation of new artifacts or instructional materials to instructors, but to take advantage of a usual resource already created when proposing a new programming assignment: the reference solution.

To design and implement the proposed feedback strategy, we took into consideration the programming process adapted from Polya (1957) [Pól57] to computer programming problem-solving. We intend that students become proficient on solving programming problems and successfully attend the goals of a programming process that includes: (1) Understand the problem; (2) Plan the solution; (3) Implement the program and (4) Look Back. Considering the fourth stage, we want students to be fluent in correcting strategies and, with critical reflection, being able to refactor their code caring about good programming quality.

Our proposal on providing feedback regarding code quality improvement – CQI was initially motivated by contrasting the manual and automated assessment and questioning: why do human instructors grade functionally correct programs so differently? Instructors approach the manual grading activity in different ways but usually agree whether a program

is "very good" or "very bad" [FHL+13] [Fin99].

Besides correctness, there are other factors weighed by instructors in manual assessment in terms of code quality. In this sense, we seek for measures that could help us reveal the quality expected by instructors in students' programs and evaluate its validity in a case study with real data. Our tests confirmed that the proposed metrics RLLOC, RCC, RH and RPEP8 do capture that notion of code quality.

After validating this construct, we designed, implemented and evaluated a feedback strategy regarding CQI. In order to perform the studies required for this approach evaluation, we implemented *qcheck*, a proof-of-concept tool. We also conducted a blind-study with human experts to confront the agreement of their evaluation and the one provided by the tool. We achieved very positive results regarding the agreement level and insightful disagreements. So, we can sustain the claim that: we can generate automated feedback based on teachers of introductory programming code quality expectations.

We designed and conducted another evaluative study with human experts, during the longitudinal study. However, as it happened along Programming 1 course and the term has not finished by the time of this writing, we could not report its results. We summarize the studies conducted to evaluate if the feedback we provide in terms of code quality improvement reflects experts' expectancies of students code quality.

C1 - We generate automated feedback based on teachers of introductory programming code quality expectations.

O1	Proposal and investigation of measures to reveal the quality expected by instructors on students' code.
RQ1	*Can the measures RLLOC, RCC, RH and RPEP8 explain the differences observed on the grades, manually assessed, of functionally correct submissions?*
O2	Evaluation of the contrast between a human expert assessment and *qcheck* assessment.
RQ2	*Does qcheck capture expert notion of code quality?*

We used *qcheck* to assess the quality of the feedback generation regarding CQI. We found significant results when performed the first controlled experiment and introduced feedback on CQI: students made more submissions after the first correct one, acting differently from

they used to do and, in fact, they were able to improve the code quality.

In a second study, intrigued by these first findings, we wanted to assess if students improved their code quality directed by *qcheck* hints or because they were motivated to do so. In this sense, we designed and conducted another controlled experiment having both groups stimulated to produce the best code they could in accordance with a set of directions. Again, we could find positive results on *qcheck* users group. Students from control group acted on the code and tried to improve it in less relevant aspects.

Finally, we evaluated students' code quality and contrasted those who had used *qcheck* from those who had not in a more realistic context, during the longitudinal study of the Programming 1 course. Differently from the previous studies when the first and last correct submissions of an assignment were evaluated, we now assessed a snapshot of the students' code production. We provided a summative feedback: intended to evaluate the student achievements in this aspect. We assessed 1497 submissions, regarding 85 programming assignments of 96 students. Our statistical results showed that experienced *qcheck* users achieved better performance in contrast with other students. This performance was expressed in terms of grade – a summative mark based on metric W and reliability – referring to the number of solved assignments.

In summary, we consider that we have got enough evidence to claim that: students can improve the code of their programming assignments prompted by timely and automated feedback. Following, we present a summary of the studies and claims to assess the feedback messages of CQI and it impacts on students' code.

C3 - Students improve code of their programming assignments prompted by timely and automated feedback

O1	Evaluation of the quality feedback generation
RQ1	*Students who receive quality feedback about their submission tend to make more submissions, after the first correct one?*
RQ2	*When students receive quality feedback about their submission they tend to deliver a better quality code?*
O2	On the use of *qcheck*'s feedback messages
RQ1	*When students receive quality feedback about their submission they tend to deliver a better quality code?*
RQ2	*Do students they tend to improve their code quality considering CQI feedback messages?*
O3	Assessing the quality of students' code production: summative feedback to teachers - study performed in the context of the longitudinal research
RQ1	*Are qcheck users producing code with better quality than other students?*

We designed and conducted a longitudinal study, in a real introductory programming course. We aimed at evaluating the experience of providing automated quality feedback during a programming course based on students' performance perspective. Besides, we wanted to catch a glimpse of the relations and patterns of use and post-feedback behaviors of students.

In the first study, we investigated if students included CQI in their programming assignments solving process routine, after *qcheck* has been presented in the course. Although we observed that a high proportion of students (66.3%) have installed and used *qcheck* at least once during the observation period, a smaller fraction of users engaged in the cycle of consuming CQI feedback and actuate in their code. In [Nar08], she observes that:

> *"...even the most sophisticated feedback is useless if learners do not attend to it or are not willing to invest time and effort in error correction."*

Considering the concepts defined by [Nar08] that external feedback comes from an external source of information while internal feedback is resultant from learners reasoning,

we can evaluate some possible post-feedback behaviors that may hinder code quality improvement feedback intended effect. In [CB93] they list factors that make the effect of the external feedback small: (1) ignore feedback; (2) reject the feedback; (3) judge the feedback irrelevant, (4) consider external and internal feedback unrelated, (5) reinterpret external feedback to make it conform to the internal feedback and (6) make superficial rather than fundamental changes to their knowledge and beliefs.

Contrasting those factors with the data gathered and our observations in the reported study, it was possible to observe each one of them. (1) Some students ignored the CQI feedback provided by *qcheck*. In our studies, there were some students that had not used *qcheck* tool in their programming assignments solving routine. Those students had their proficiency of *qcheck* use labeled as not in such situations. In some cases, it happens due to the scope of the study design: only programming assignments of units 3, 4 and 5 were prepared for *qcheck* use. So, extremely high performing students went through these units without solving any exercise, just solving the exams. On the other hand, extremely poor performing students had not the chance to use *qcheck* either, as they had only solved assignments from units 1 and 2. By the time we finished the data collection, 7.83% students in the class were in this situation.

Situation (2) and (3) are different and indeed happened, but produce the same result: students just checked but did not consume the CQI feedback. The discourse analysis of students' questionnaire about *qcheck* evaluation and our observation in laboratory classes gave us some hints why it happened. Some students argued that (2) they had rejected the feedback because it raised so many warnings that "solving them will be more difficult than solving the task.", said S15. Some of them judged the CQI feedback irrelevant (3), mainly when it is related to style warnings. As an example, we may cite PEP8 restriction "Limit all lines to a maximum of 79 characters". This may also be mixed with (4) situation. Another style warning that is usually rejected is the PEP8 recommendation that compound statements (multiple statements on the same line) must be discouraged. It happens because the course instructors usually use such construction in class: if a_condition: break. Students mimic instructors examples and, for this reason, internal and external feedback conflicts in this situation.

In situation (5) and (6) students may have the will but not the skill to solve the problems

raised by the warnings. This can be observed in many situations that students interact and engages the CQI cycle but cannot completely eliminate all code quality warnings and attain $W = 0$ and the "No warnings" message.

In fact, on RQ2 we asked if learners that consume *qcheck* code quality feedback are succeeded in improving their programming assignments code quality. We observed that 24.75% of the students that used *qcheck* tool were able to accomplish *"No warnings"* in their first attempt. Considering only students who engaged the CQI cycle, 25.56% that consumed *qcheck* feedback also accomplish $W = 0$, 20.37% just checked CQI feedback and 29.28% were not capable to eliminate the warnings. In order to better understand why some students managed to reduce the number of *W*, we deeply evaluated their patterns of *qcheck* usage. We categorized students according to their proficiency in using *qcheck* and found that most proficient users produce better code. In practice, it means that not all students understood or played "the game", but those who did achieve good results regarding code quality. Although not all students accomplished total warning elimination, we observed that the majority of them reduced their number of warnings (ΔW and ΔS). It means that students who checked and consumed CQI were able to improve their code in at least one point. We also find statistical significance when comparing two distributions.

In practice, we conclude that *qcheck* usage is positive as users improve their code quality and reduced the number of observed warnings in the final submission. Eliminating warnings is more common in *qcheck* proficient users. Besides analyzing causes of success, we need a deeper investigation on why some did not succeed. Apart from the lack of skill to accomplish the task, there are other factors that might be involved. We conjectured that some of them might be related to the tool adjustments and feedback strategies improvement. We address this as future work.

Lastly, an investigation was performed to contrast the students who have used and those who have not used *qcheck* during their programming assignments resolution. The results confirm previous experimental findings, whether the mean of warnings tends to be smaller among *qcheck* users and even smaller among those who consumed CQI feedback. Using statistical tools, we were able to observe that some factors influences on the decreasing of the number of defects *W*: *qcheck* feedback consumption and if the students are *qcheck* user proficient. We also found that lloc, influence increasing *W*, as expected.

We performed a qualitative evaluation considering users' perspective using a semi-structured interview and having students to register their impressions in a questionnaire. This instrument was valuable to uncover important usage pitfalls. Some students, for example, used *qcheck* before the code is thoroughly tested. This may be the cause that some of them could not manage to eliminate their warnings (W=0). As for negatives aspects, some students refer "frustration" and "difficulties" when trying to reduce the number of warnings. As it can be lack of skill to accomplish the task, it may also be a signal that we need to evaluate such situations in order to understand if it is the level of feedback we really intend to deliver. Many positive aspects were reported regarding *qcheck* usage and the support it provides.

Those comments were stimulating and revealed that the strategy of CQI we proposed successfully fill a gap on these students learning opportunities. Overall, we considered that we have gathered minimal support to claim that: Students improve programming skills stimulated by the reflection on their programming assignments code with the purpose to improve its quality. We summarize in sequence the studies we conducted to support the claim.

C3 - Students improve programming skills stimulated by the reflection on their programming assignments code with the purpose to improve its quality

O1	Evaluation of providing automated quality feedback along a programming course - study performed in the context of the longitudinal research
RQ1	*Do learners incorporate "code improvement" as part of their programming process cycle?*
RQ2	*Do learners that consume qcheck code quality feedback succeed in improving their programming assignments code quality?*
RQ3	Do learners that consume *qcheck* code quality feedback improve their programming abilities regarding code quality?
O2	*Users evaluation of qcheck tool*
RQ4	*How is students' perception about qcheck usefulness in the aid of improving programming assignments' code quality?*

7.1 Theoretical Implications

An important discussion referring to our approach on providing feedback to code quality improvement refers to how much information to deliver. In general, computer-based learning environments differ in terms of whether or how they give or withhold information or assistance. In the proposal hereby discussed we raised the question: Do we need more elaborate hints on 'how to solve' the warnings? Or when to deliver such elaborate and direct hint?

This problem on *"how should learning environments balance information or assistance giving and withholding to achieve optimal student learning"* is known as the 'assistance dilemma' [KA07]. In this sense, we have to consider the possible benefits and costs of information delivering versus omitting in order to design effective instruction, summarized on Figure 7.1 [KA07].

	Benefits	Cost
Giving information or assistance	Accuracy Efficiency of communication Thrill of (supported) success	Lack of attention May not engage long-term memory Stealing chance to shine
Withholding information or assistance	Forces attention Engages long-term memory Thrill of independent success	Cost of errors Floundering, confusion, wasted time Frustration of failure

Figure 7.1: Summary of Costs and Benefits on Providing Assistance.

There are many benefits that we observed in withholding information "on how to solve the warning" with *qcheck* during the longitudinal studies. We can cite the opportunity students have to think about other ways to solve the task. It pushes their knowledge boundaries and as they are impelled to improve their skills to, for example, solve warnings that demand reduction: the lines of code or the number of conditional statements they use in their solution.

We witnessed students, motivated by *qcheck* warnings, debating on how to improve a functioning code on the discussion board (Slack), an extremely rare phenomenon in the course. In another situation, a graduate teachers assistant reacted with *"Wow! I need to study more Python."* when we assisted him with a student code that *qcheck* raised some warnings. During an experiment when *qcheck* was first introduced, a student declared, expressing hesitation, that *"using qcheck, I feel that I have been watched"*.

We argue that the threshold of the dilemma assistance is not a one-size-fits-all solution. It depends on the maturity of the approach. This maturity is only attained after many evolutions and evaluations in real programming courses. Furthermore, this balance is extremely related to the pedagogical approach of the course. Decisions about what to reveal or withdrawn must not be dissociated of instructors orientation.

7.2 Pedagogical Implications and Opportunities

There are some pedagogical implications and other opportunities raised by our approach and feedback strategy. We highlighted some of them in this section.

7.2.1 Learning Conversations and Interactions

In a study by Robinson and Udall (2006), they demonstrate that interactive "learning conversations" with engineering students lead to a greater sense of ownership in students' learning [RU06]. The feedbacks we intend to provide stimulate conversation and discussion about how to (better) solve the problem. We believe that interactions are extremely beneficial to engage students in learning activities. Furthermore, it inverts the information flow: from the student to the instructor. Instructors' assistance is now directed by students request, promoting the significant learning process.

7.2.2 Critical Reflection About Code

Feedback on CQI provides to the student the opportunity to reflect about their code, after a task completion. Literature shows that such opportunities to "critically evaluate the quality of their own work during, as well as after, its production" can foster the desired self-regulation learning [SBCP11]. It means that it can stimulate students to actively engage in evaluating their performance, including the processes underlying their performance, which are aimed at the regulation of learning [TWV15]. In the users' evaluation, many students stated that *qcheck* use was stimulated by their willingness to "became a better programmer", "write better and cleaner code", "learn more about how to improve my code" and other self-assessment about their code production and role as a programmer.

7.2.3 Clear Marking Criteria to Programming Assignments

It is important that students be aware and stimulated that code quality matters. *Qcheck* users reported that what may hinder them to invest time in CQI, in this particular course, is that code quality represents only "10% of the final grade". More students will consider and actively engage in refactoring activities if it was rewarded in the course context. In arguing about effective feedback for students in CS1 courses, Claudia Ott and colleagues declare:

> *"* (...) *marking criteria for a programming task should not only address the program's functionality but also the programming process and matters of good programming style. A clear communication of those marking criteria up front would help students understand the expectations and act accordingly. If process-related goals are clearly stated, feedback can address unsatisfactory programming approaches and assist improvement toward these goals for the next tasks."[ORS16]*

7.2.4 Summative Assessment of Code Quality Produced by Students

While formative assessment goal is to monitor student learning, summative assessment goal is to evaluate students' learning [Shu08]. However, we argue that the opportunity of providing a summative assessment about code quality production automatically using *qcheck* can be beneficial to not only to instructors but also to students. The proposal aims to aid students in the process of self-monitoring. In our experience, in this longitudinal study, we personalized the feedback to each student in the course and delivered in his or her personal AAS dashboard. Claudia Ott and colleagues also refer this approach as highly valuable:

> *"To help students in the process of self-monitoring their performance providing course information seems highly valuable. Adding meaning to available course data would inform students about their prospects in the course. Based on individual performance data, feedback could be personalized by adopting performance goals (e.g., goals to catch up or attend labs more regularly), relating students' actual performance to what was observed as "successful" performance in the past, and pointing out aspects to improve." [ORS16]*

Chapter 8

Concluding Remarks

In this work, we proposed and evaluated an approach to generate and deliver feedback to students in the programming process. The challenge was to provide timely and enriched feedback that stimulates students to reason about the problem and their solution and to improve their programming skills. We intended to leverage programming problem-solving learning generating enriched automated feedback, regarding students programming assignments, with information typically delivered by human instructors. Furthermore, we constrained our strategies of feedback generation to obtain information from instructional materials already produced by teachers, what aims to minimize burdens imposed to them.

We focused on providing feedback with respect to the last phase of the programming process, when the program is revisited and refactored. The main contribution of this Ph.D. thesis relies on the lessons learned with the proposal and evaluation of automated generation of code quality feedback to an introductory programming course. We conducted research studies, such as experiments, case studies and a survey to gather empirical evidence to support the following claims:

(1) We can generate automated feedback based on teachers of introductory programming code quality expectations;

(2) Students can improve code of their programming assignments prompted by timely and automated feedback;

(3) Students improve programming skills stimulated by the reflection on their programming assignments code with the purpose to improve its quality.

We performed an evaluation of this approach with students of an introductory programming course in a longitudinal study. We used a proof-of-concept tool – *qcheck* to materialize our proposal. Results found in our longitudinal evaluation goes beyond what we initially expected: the improved assignments' code quality. We observed that students felt stimulated, and in fact, improved their programming abilities driven by the exercise of reasoning about their already functioning solution. Furthermore, we conducted a users assessment among students. We gathered very positive evaluations and good suggestions on how to improve the tool. It was important to observe that some students were not using *qcheck* as we expected: after the program is functionally correct it was also important, to hear from them what precludes them to use the tool.

8.1 Future Works

There are several opportunities for future works that arise from this doctoral research. Some are specific directions to improve our work and others are proposals of new studies regarding the data collection of the longitudinal study on code quality improvement.

- Evaluation of other metrics to assess code quality such as dynamic software metrics, vocabulary related, design constraints, and so on;
- Investigation on how students' perceptions of feedback influence their engagement with the feedback process, in CQI context;
- Performing qualitative studies using the code submissions of students who could not manage to improve their code quality using feedback messages;
- Studis exploring the 'assistance dilemma' and search for the balance in providing feedbacks on CQI;
- Uncovering relations on data gathered from longitudinal studies in, an exploratory analysis, contrasting students profile and post-feedback behavior and performance on the task and course.

Bibliography

[AGF13] E.C. Araujo, D.S. Guerrero, and J.A. Figueiredo. Avaliando a legibilidade em programas de iniciantes. *Workshop de EducaÃğÃčo em ComputaÃğÃčo*, 2013.

[AM05] Kirsti M Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer science education*, 15(2):83–102, 2005.

[ASF16] Eliane Araujo, Dalton Serey, and Jorge Figueiredo. Qualitative aspects of students' programs: Can we make them measurable? In *Frontiers in Education Conference (FIE), 2016 IEEE*, pages 1–8. IEEE, 2016.

[Bak16] Ryan S Baker. Stupid tutoring systems, intelligent humans. *International Journal of Artificial Intelligence in Education*, 26(2):600–614, 2016.

[BBBL15] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 134–144. IEEE Press, 2015.

[BC05] Jens Bennedsen and Michael E Caspersen. Revealing the programming process. In *ACM SIGCSE Bulletin*, volume 37, pages 186–190. ACM, 2005.

[BC07] Jens Bennedsen and Michael E Caspersen. Failure rates in introductory programming. *ACM SIGCSE Bulletin*, 39(2):32–36, 2007.

[BE14] Kevin Buffardi and Stephen H Edwards. A formative study of influences

on student testing behaviors. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 597–602. ACM, 2014.

[Blo84] Benjamin S Bloom. The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational researcher*, 13(6):4–16, 1984.

[CAMF+03] Janet Carter, Kirsti Ala-Mutka, Ursula Fuller, Martin Dick, John English, William Fone, and Judy Sheard. How shall we assess this? *SIGCSE Bull.*, 35(4):107–123, June 2003.

[Cas07] Michael Edelgaard Caspersen. *Educating novices in the skills of programming*. PhD thesis, Department of Computer Science, 2007.

[CB93] Clark A Chinn and William F Brewer. The role of anomalous data in knowledge acquisition: A theoretical framework and implications for science instruction. *Review of educational research*, 63(1):1–49, 1993.

[DLO05] Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)*, 5(3):4, 2005.

[DLRC14] Paul Denny, Andrew Luxton-Reilly, and Dave Carpenter. Enhancing syntax error messages appears ineffectual. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 273–278. ACM, 2014.

[DSPQ+17] John DeNero, Sumukh Sridhara, Manuel Pérez-Quiñones, Aatish Nayak, and Ben Leong. Beyond autograding: Advances in student feedback platforms. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 651–652. ACM, 2017.

[Edw03] Stephen H Edwards. Improving student performance by evaluating how well students test their own programs. *Journal on Educational Resources in Computing (JERIC)*, 3(3):1, 2003.

[ESPQ+09] Stephen H Edwards, Jason Snyder, Manuel A Pérez-Quiñones, Anthony Allevato, Dongkwan Kim, and Betsy Tretola. Comparing effective and ineffective behaviors of student programmers. In *Proceedings of the fifth international workshop on Computing education research workshop*, pages 3–14. ACM, 2009.

[FHL+13] Sue Fitzgerald, Brian Hanks, Raymond Lister, Renee McCauley, and Laurie Murphy. What are we thinking when we grade programs? In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 471–476. ACM, 2013.

[Fin99] Sally Fincher. What are we doing when we teach programming? In *Frontiers in Education Conference, 1999. FIE'99. 29th Annual*, volume 1, pages 12A4–1. IEEE, 1999.

[GMD11] Joyce Wangui Gikandi, Donna Morrow, and Niki E Davis. Online formative assessment in higher education: A review of the literature. *Computers & education*, 57(4):2333–2351, 2011.

[GRZ14] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. Feedback generation for performance problems in introductory programming assignments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 41–51. ACM, 2014.

[HE06] Torsten Hothorn and Brian S Everitt. *A handbook of statistical analyses using R*. CRC press, 2006.

[HT07] John Hattie and Helen Timperley. The power of feedback. *Review of educational research*, 77(1):81–112, 2007.

[IAKS10] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, pages 86–93. ACM, 2010.

[IE14] Alexandru Iosup and Dick Epema. An experience report on using gamification in technical higher education. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 27–32. ACM, 2014.

[Jad05] Matthew C Jadud. A first look at novice compilation behaviour using bluej. volume 15, pages 25–40. Taylor & Francis, 2005.

[JU97] David Jackson and Michelle Usher. Grading student programs using assyst. In *ACM SIGCSE Bulletin*, volume 29, pages 335–339. ACM, 1997.

[KA07] Kenneth R Koedinger and Vincent Aleven. Exploring the assistance dilemma in experiments with cognitive tutors. *Educational Psychology Review*, 19(3):239–264, 2007.

[KF16] James A Kulik and JD Fletcher. Effectiveness of intelligent tutoring systems: a meta-analytic review. *Review of Educational Research*, 86(1):42–78, 2016.

[LAF+04] Raymond Lister, Elizabeth S Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, et al. A multi-national study of reading and tracing skills in novice programmers. In *ACM SIGCSE Bulletin*, volume 36, pages 119–150. ACM, 2004.

[LAMJ05] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. In *Acm Sigcse Bulletin*, volume 37, pages 14–18. ACM, 2005.

[LK77] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

[MAD+01] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. *ACM SIGCSE Bulletin*, 33(4):125–180, 2001.

[MANL14] Wenting Ma, Olusola O Adesope, John C Nesbit, and Qing Liu. Intelligent tutoring systems and learning outcomes: A meta-analysis., 2014.

[May08] Richard E Mayer. Applying the science of learning: Evidence-based principles for the design of multimedia instruction. *American psychologist*, 63(8):760, 2008.

[MBI+05] Andrew Mcgettrick, Roger Boyle, Roland Ibbett, John Lloyd, Gillian Lovegrove, and Keith Mander. Grand challenges in computing: EducationâĂŤa summary. *The Computer Journal*, 48(1):42–48, 2005.

[McC76] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[Mil11] Dejan Milojicic. Autograding in the cloud: interview with david o'hallaron. *IEEE Internet Computing*, 15(1):9–12, 2011.

[MM13] Tommy MacWilliam and David J Malan. Streamlining grading toward better feedback. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, pages 147–152. ACM, 2013.

[MY99] Susan A Mengel and Vinay Yerramilli. A case study of the static analysis of the quality of novice student programs. In *ACM SIGCSE Bulletin*, volume 31, pages 78–82. ACM, 1999.

[Nar08] Susanne Narciss. Feedback strategies for interactive learning tasks. *Handbook of research on educational communications and technology*, 3:125–144, 2008.

[Nor07] Pete Nordquist. Providing accurate and timely feedback by automatically grading student programming labs. *Journal of Computing Sciences in Colleges*, 23(2):16–23, 2007.

[NSS+14] Susanne Narciss, Sergey Sosnovsky, Lenka Schnaubert, Eric Andrès, Anja Eichelmann, George Goguadze, and Erica Melis. Exploring feedback and student characteristics relevant for personalizing feedback strategies. *Computers & Education*, 71:56–76, 2014.

[ORS16] Claudia Ott, Anthony Robins, and Kerry Shephard. Translating principles of effective feedback for students into the cs1 context. *ACM Transactions on Computing Education (TOCE)*, 16(1):1, 2016.

[Pep15] Pep8. Style guide for python code. `http://legacy.python.org/dev/peps/pep-0008/`, March 2015.

[PHG+15] Raymond Pettit, John Homer, Roger Gee, Susan Mengel, and Adam Starbuck. An empirical study of iterative improvement in programming assignments. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 410–415. ACM, 2015.

[Pól57] George Pólya. How to solve lt. *Princeton University*, 1957.

[Rad14] Radon. Radon. `https://radon.readthedocs.org/en/latest/index.html`, March 2014.

[RRR03] Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and teaching programming: A review and discussion. *Computer science education*, 13(2):137–172, 2003.

[RSKPFVI14] Manuel Rubio-Sánchez, Päivi Kinnunen, Cristóbal Pareja-Flores, and Ángel Velázquez-Iturbide. Student perception and usage of an automated programming assessment tool. *Computers in Human Behavior*, 31:453–460, 2014.

[RU06] Alan Robinson and Mark Udall. Using formative assessment to improve student learning through critical reflection. *Innovative assessment in higher education*, pages 92–99, 2006.

[SBCP11] Kay Sambell, Elizabeth Barry-Cutter, and Nicholas Price. Rethinking feedback in higher education: an assessment for learning perspective; learning to learn: learning to cook: the f-word, feedback. 2011.

[SGSL13] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. *ACM SIGPLAN Notices*, 48(6):15–26, 2013.

[SHP+06] Jaime Spacco, David Hovemeyer, William Pugh, Fawzi Emad, Jeffrey K Hollingsworth, and Nelson Padua-Perez. Experiences with marmoset: designing and using an advanced submission and testing system for programming courses. *ACM Sigcse Bulletin*, 38(3):13–17, 2006.

[Shu08] Valerie J Shute. Focus on formative feedback. *Review of educational research*, 78(1):153–189, 2008.

[SS89] E. Soloway and J. Spohrer. *Studying the Novice Programmer*. Lawrence Erlbaum Associates, Hillsdale, New Jersey. 497 p., 1989.

[Tho97] Simon Thompson. Where do i begin? a problem solving approach in teaching functional programming. In *International Symposium on Programming Language Implementation and Logic Programming*, pages 323–334. Springer, 1997.

[TWV15] Caroline F Timmers, Amber Walraven, and Bernard P Veldkamp. The effect of regulation feedback in a computer-based formative assessment on information problem solving. *Computers & education*, 87:1–9, 2015.

[Van06] Kurt Vanlehn. The behavior of tutoring systems. *International journal of artificial intelligence in education*, 16(3):227–265, 2006.

[VAW14] Arto Vihavainen, Jonne Airaksinen, and Christopher Watson. A systematic review of approaches for teaching introductory programming and their influence on success. In *Proceedings of the tenth annual conference on International computing education research*, pages 19–26. ACM, 2014.

[VPL11] Arto Vihavainen, Matti Paksula, and Matti Luukkainen. Extreme apprenticeship method in teaching programming for beginners. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 93–98. ACM, 2011.

[WL14] Christopher Watson and Frederick WB Li. Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 39–44. ACM, 2014.

[Woo10] Beverly Park Woolf. *Building intelligent interactive tutors: Student-centered strategies for revolutionizing e-learning*. Morgan Kaufmann, 2010.

[Yad11] Aharon Yadin. Reducing the dropout rate in an introductory programming course. *ACM inroads*, 2(4):71–76, 2011.

[Yai14] Yoav Yair. Did you let a robot check my homework? *ACM Inroads*, 5(2):33–35, 2014.

Appendix A

Uma revisão sobre sistemas automáticos para a avaliação de atividades de programação

Uma revisão sobre sistemas automáticos para a avaliação de atividades de programação

Abstract. *In this article we will review the literature on automatic assessment systems (AA) for programming activities. These systems have been developed over time with different characteristics as to: feedback given to the student, approaches to assess student's programs, security issues, among others. In this research, we seek to evolve existing studies in the literature that presents those systems published until 2010. We shall present our perspective on the surveyed systems relating to those characteristics and provide our perception of future works in the area.*

Resumo. *Neste artigo faremos uma revisão da literatura sobre sistemas automáticos para avaliação de atividades de programação (AA). Estes sistemas têm sido desenvolvidos ao longo do tempo com características distintas quanto: ao feedback dado ao aluno, à abordagem utilizada para avaliar os programas, às questões de segurança, dentre outras. Nesta pesquisa, procuraremos evoluir estudos existentes na literatura que apresentam dados até o ano de 2010, com o propósito de ampliar o corpo de conhecimento da área. Apresentaremos a nossa perspectiva sobre os sistemas catalogados de acordo com as características investigadas, além de enriquecer o trabalho apontando as direções sobre os trabalhos futuros.*

1. Introdução

Nos cursos da área de computação, o ensino de programação ocupa papel de destaque. Um estudo realizado por Pears *et al* [Pears, 2005] classificou e mapeou a literatura na área de ciência da computação. Neste estudo, constatou-se que os relatos e pesquisas sobre ferramentas e sistemas de auxílio ao ensino compõem o grupo com mais artigos publicados, em veículos da ACM – Association for Computing Machinery[1], estando à frente de temas como: currículo, pedagogia e linguagens de programação.

No contexto das ferramentas computacionais de auxílio ao ensino, estão os sistemas automáticos para avaliação de atividades de programação (AA). A avaliação automática de programas é um recurso didático que vem sendo cada vez mais utilizado, especialmente devido ao aumento do tamanho das turmas de cursos de programação. Com o uso dos sistemas de AA, é possível manter a consistência das correções, garantir que será dado algum feedback rapidamente ao aluno sobre suas atividades e diminuir a carga de trabalho sobre o professor. Tais sistemas são especialmente úteis em cursos de programação onde há ênfase na realização de muitos exercícios.

Embora haja muitos sistemas de AA já desenvolvidos, professores, pesquisadores ou outros desenvolvedores continuam criando seus próprios sistemas para suprir necessidades específicas. Creditamos este fenômeno ao desconhecimento dos

[1] http://www.acm.org/

sistemas existentes, à impossibilidade de adaptação dos sistemas a peculiaridades do curso de programação, ou à necessidade de implementar o resultado de uma pesquisa com fins de avaliação. Com este trabalho de revisão de literatura, pretendemos ampliar o corpo de conhecimento sobre as iniciativas já empreendidas nesta área, de modo que professores, desenvolvedores e pesquisadores possam melhor direcionar os seus esforços de pesquisa e implementação.

Neste trabalho, evoluímos os estudos sobre o desenvolvimento de sistemas catalogados até o ano de 2010 nos trabalhos publicados por Douce *et al* (2005), Ala-Mutka (2005) e Ihantola *et al* (2010). Descrevemos as características dos sistemas e os diferenciais apresentados por eles. Identificamos as oportunidades de atuação e trabalhos de pesquisa, ao identificar lacunas nos sistemas desenvolvidos desta época até os dias atuais. Enfatizamos, principalmente, as questões que dizem respeito ao feedback dado ao aluno com respeito à qualidade de seu código e ao caminho que o conduziu a uma solução bem sucedida para o problema.

Ao término do estudo, pudemos observar que aspectos que foram considerados relevantes em estudos anteriores não eram mais mencionados nos sistemas atuais, como o caso das políticas de re-submissão de exercícios. Por outro lado, as preocupações com os problemas de escala são mais presentes nos sistemas contemporâneos. O estudo revela, ainda, que não houve evolução na ênfase dada pelos sistemas de AA nas questões de segurança, que é, em geral, baixa.

Este artigo apresenta uma revisão sobre a literatura de sistemas automáticos de avaliação para atividades de programação. O restante documento está organizado como segue: a seção 2 apresenta o referencial teórico do estudo e ressalta os trabalhos relacionados, a seção 3 detalha a metodologia empregada para a condução da revisão da literatura, a seção 4 exibe os resultados encontrados tanto na coleta de dados quanto na análise dos sistemas que são, posteriormente, discutidos na seção 5, onde são delineadas algumas conclusões.

2. Referencial Teórico e Trabalhos Relacionados

Os sistemas para avaliação automática de exercícios de programação (AA) são utilizados em diversos cursos introdutórios de programação no mundo [Cheang, 2003][Ala-Mutka, 2005]. Impulsionados pelos juízes online das maratonas de programação [Kolstad, 2009][Revilla, 2008], os testadores foram sendo adaptados para a realidade de cada curso ou laboratórios de programação. Estudos mostram que já existem e, continuam sendo desenvolvidos, muitos sistemas com este propósito [Ihantola, 2010][Ala-Mutka, 2005]. Poucos têm seu código aberto, o que prejudica a tentativa de reutilização e adaptação para outros cursos diferentes daquele para o qual foram projetados.

Uma revisão de literatura na área de sistemas de avaliação automática que pretende dar uma perspectiva histórica da área é o trabalho de Douce *et al* (2005). Neste trabalho, ele classifica o sistemas em “gerações” com base na abordagem tecnológica utilizada. A primeira geração inclui os primeiros sistemas, desenvolvidos na década de 60 até o final da década de oitenta. Para usar esses sistemas, era necessário muito conhecimento, o que na prática significava que o desenvolvedor e o usuário eram, em geral, a mesma pessoa. A segunda geração é composta por ferramentas que poderiam ser operadas via linha de comando ou com uma interface gráfica para o usuário (GUI)

bem simples. A terceira geração é marcada por sistemas orientados à interface Web. Além disso, Douce ainda discute os sistemas de avaliação automáticos sob a perspectiva pedagógica.

Outro trabalho fundamental no tocante ao mapeamento dos sistemas de avaliação automáticos até 2005 é o de Kristi Ala-Mutka. Neste trabalho as diferentes técnicas de avaliação dos sistemas analisados são ressaltadas. São elencadas muitas vantagens da utilização de tais sistemas como a velocidade, a disponibilidade a consistência e a objetividade das avaliações. Ala-Mutka adverte que é necessário que haja uma cuidadosa justificativa pedagógica para cada uma das decisões de projeto, nas avaliações empreendidas e, também, no feedback que é fornecido ao aluno.

Finalmente, o trabalho de Ihantola *et al* de 2010, procura revisar sistematicamente a literatura na área de sistemas de avaliação automáticos no período que abrange 2006 a 2010. Os sistemas que interessam àqueles autores são 1) sistemas de avaliação automática para competições de programação e 2) sistemas de avaliação para apoio de cursos de programação introdutória. Como parte de suas conclusões, estão os principais pontos de diferenciação entre os sistemas estudados: a forma como lidam com as re-submissões de questões; a forma como os testes automáticos são definidos e como questões de segurança são tratadas.

3. Metodologia

Para conduzir este trabalho de revisão sobre a literatura de sistemas automáticos para avaliação de atividades de programação é necessário definir o escopo dos sistemas que são de nosso interesse nesta pesquisa. Inicialmente, esclarecemos o que consideramos como sendo "atividades de programação" e "avaliação automática".

As atividades de programação são quaisquer código ou trecho de código gerado pelos estudantes em resposta a um problema ou especificação passado pelo professor/tutor em um curso de ensino de programação. Estes códigos são artefatos que podem ser analisados dinâmica ou estaticamente. Não necessariamente trata-se de código executável. Muitas vezes, as respostas a estas atividades de programação, ou seja, o código dos programas produzidos pelos estudantes são chamados simplesmente de "submissões", pois são "submetidos" ao AA. Observe que atividades de programação que incluam diagramas, especificações ou documentações estão fora do escopo deste trabalho.

A avaliação automática refere-se à analise de quaisquer dados produzidos a respeito submissão do estudante (data, hora, quantas vezes foi submetido, etc) além de dados a respeito do próprio programa, tais como a correção funcional, caso ele seja submetido a testes automáticos. Os sistemas de AA, que interessam a este trabalho de pesquisa, devem fornecer algum tipo de feedback. Por este motivo, o nosso estudo engloba e ultrapassa o conjunto dos sistemas de avaliação automática com o propósito estrito de dar nota ou pontuar o programa do estudante (*grading systems*).

Este trabalho evolui a pesquisa de Ihantola et al (2010). As questões de pesquisa que nos motivaram para a realização desta revisão são:

1. Quais são as características dos sistemas automáticos de avaliação relatados na literatura após 2010?
2. Para quais direções estes sistemas impulsionam os trabalhos futuros?

Há muitos sistemas desenvolvidos com o propósito da avaliação automática. É bastante comum que professores criem sua própria solução para cursos específicos de programação ao invés de aderir ou adaptar sistemas já existentes. Além do mais, muitos sistemas que estão em uso na prática sequer foram publicados em artigos. A metodologia adotada procura seguir os passos propostos por Brereton *et al* (2010) para revisões sistemáticas no domínio de Engenharia de Software.

Na fase inicial do planejamento da revisão, foram definidos os serviços de indexação utilizados como base para a pesquisa, bem como os termos ou palavras-chave da busca. A revisão será realizada sobre as consagradas bases de dados em ciência da computação: ACM Digital Library, IEEE Xplore, Science Direct (Elsevier), Taylor and Francis on-line. Os artigos entre 2010 e 2014 dos anais da Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE) e das revistas científicas Computer Science Education (CSE), Olympiads in Informatics International Journal (OI) e Transactions on Education (ToE) foram considerados, para manter a consistência com o estudo de Ihantola *et al*. Além disso, ampliamos o nosso espectro de pesquisa, incluindo os trabalhos das conferências Anual Conference of the Special Interest Group on Computer Science Education (SIGCSE), Simpósio Brasileiro de Informática na Educação (SBIE), Workshop de Informática na Escola (WIE), da Revista Brasileira de Informática na Educação (RBIE) e do jornal Computers & Education (C&E).

O processo de seleção dos trabalhos é iterativo [Brereton, 2010], de modo que após uma triagem inicial feita através da consulta na base de dados pelos termos adequados, uma nova triagem foi realizada considerando as restrições que guiam os interesses deste trabalho. Nas publicações da SBC, a pesquisa foi realizada de forma semi-automática. Na primeira triagem utilizamos o termo "programação" na consulta das bases de dados. Em seguida, de modo manual, os artigos anteriores a 2010 foram descartados e foram considerados apenas os artigos relevantes de acordo com os termos. Nas demais bases de dados, a sentença de pesquisa utilizada para a recuperação dos artigos foi derivada das questões de pesquisa deste trabalho. Ela foi composta usando as seguintes palavras-chave e conectores: ("automatic" OR "automated") AND ("assessment" OR "grading") AND "programming". A restrição temporal, nestes casos, foi aplicada na sentença do engenho de busca.

Após a fase inicial de busca por palavras chave, ao conjunto de artigos selecionados foram aplicados alguns filtros. Consideramos apenas os sistemas inéditos para o meio acadêmico, ou seja, publicados pela primeira vez em artigos de revistas, jornais científicos ou em periódicos de conferências. Os sistemas devem ser voltados ao ensino de programação de computadores para nível superior/universitário e fornecer algum tipo de feedback ao aluno ou instrutor.

As características consideradas relevantes para a avaliação dos sistema de AA foram definidas tomando-se por referência aquelas que foram levantadas no trabalho de Ihantola *et al* (2010) e incluindo a nova categoria "feedback". Embora existentes no estudo anterior, as características "Re-submissão" e "Especialidades" não aparecem no estudo atual. A primeira por não haver menção à política de re-submissão de códigos nos sistemas pesquisados e a segunda por não haver espaço suficiente no trabalho. Mais informações sobre os sistemas podem ser encontradas no apêndice disponível on-line em: http://goo.gl/upcekv.

4. Resultados

A apresentação dos resultados desta pesquisa, inicia-se com a visualização dos dados sobre a busca dos artigos seguindo a mesma abordagem adotada por Aureliano e Tedesco (2012). Em seguida, daremos ênfase nas características que guiaram as decisões pedagógicas e de projeto dos sistemas selecionados.

4.1 Coleta de dados

A pesquisa foi capaz de recuperar 132 artigos completos atendendo às restrições temporal e dos termos da sentença de pesquisa. Foram selecionados 10 artigos que atendiam as demais restrições descritas na metodologia deste trabalho. O quadro com o resultado geral encontra-se na **Tabela 1**

Publicação	Base de Dados	Artigos Recuperados	Artigos Selecionados
OI	VU[2]	6	2
C&E	Elsevier	30	2
CSE	Taylor and Francis	6	0
ItiCSE	ACM	18	1
SIGCSE	ACM	12	1
ToE	IEEE	0	0
SBIE	SBC	30	4
WIE	SBC	16	0
RBIE	SBC	10	0
TOTAL		132	10

Tabela 1. Resultado das buscas nas bases de dados

A **Figura 1** mostra: (a) a distribuição das publicações no tempo e (b) nos meios em que elas apareceram. Observa-se que o ano de 2012 foi responsável pela maior concentração de publicações sobre sistemas de avaliação automáticos. Uma justificativa possível para que não apareçam publicações nos anos subsequentes é a tendência de criação de ferramentas de apoio ao ensino mais abrangentes, onde o AA é parte de um ecossistema maior. Sendo assim, publicações deste tipo de sistema não foram capturadas pela nossa pesquisa. Analisando pela perspectiva das publicações, observamos que o SBIE concentrou o maior número de artigos. Possivelmente, este fenômeno seja decorrente da busca semi-automática realizada na base das publicações nacionais que tornou a sentença de pesquisa mais abrangente.

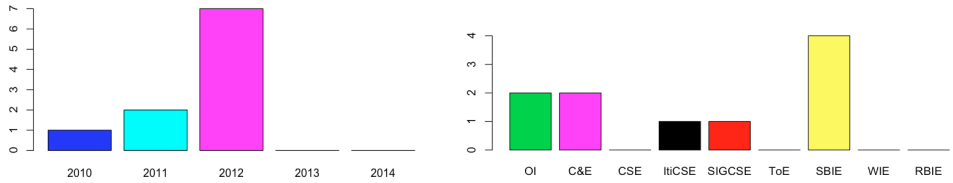

Figura 1. (a)Distribuição temporal dos artigos e (b) Distribuição dos artigos nas publicações

[2] Vilnius University Institute of Mathematics and Informatics

4.2 Linguagens de Programação

Os sistemas avaliados neste estudo, suportam, em sua maioria a linguagem Java de programação. Em seguida, vêm os sistemas que suportam C/C++ e depois Python e Java. Esta escolha reflete a tendência dos cursos introdutórios de programação de adotarem tais linguagens para o seu processo de ensino. Os sistemas Jutge [Petit, 2012] e Pythia [Combéfis, 2012] oferecem seus serviços de modo independente de linguagem, ou seja, as soluções submetidas pelos alunos podem ser escritas em uma linguagem previamente escolhida, dentre o elenco de linguagens de programação disponibilizados pelo sistema. Jutge, por exemplo, suporta submissões em mais de 22 linguagens de programação.

4.3 Avaliação dos Programas

O processo de avaliação das submissões dos alunos é o ponto chave do estudo das funcionalidades do AA. As avaliações podem ser estáticas ou dinâmica, manuais ou automáticas; ou uma combinação destas abordagens como no ProgTest [de Souza, 2012]. Além disso, há sistemas que promovem, o aprendizado da disciplina de testes de software. Neste sistemas, os casos de testes criados pelos aluno são fornecidos para o AA junto com a solução dos programas e são avaliados de modo adequado.

Nas avaliações de programas de modo dinâmico, os códigos são exercitados frente a uma bateria de testes provida pelo instrutor/autor do problema. Este processo permite que o sistema dê feedback sobre a correção funcional da solução. Além disso, é possível verificar a evolução da execução do programa na CPU da máquina, fornecendo feedback sobre a complexidade do programa [Combéfis, 2012][Brown, 2012]. A avaliação dinâmica é a abordagem mais prevalente entre os sistemas avaliados. Nas avaliações estáticas, é possível detectar erros de sintaxe e compilação além da verificação da similaridade a modelos de soluções cadastradas no sistema. No AutoLEP [Wang, 2011], o programa do aluno é comparado a modelos de programas que representam a forma correta de resolver o problema. Quanto mais semelhante aos modelos cadastrados, maior será a nota do aluno. Mesmo que o programa esteja incompleto ou apresente erros de sintaxe, ele terá uma nota considerada justa, pelos autores. O ProgTest [de Souza, 2012], segue abordagem semelhante.

Alguns sistemas, principalmente os que fornecem notas, adotam o processo semi-automático de avaliação a fim de corrigir ou mitigar distorções causadas nas notas dos alunos devido a processos muito estritos de avaliação. O processo de avaliação adotado pelo Jutge [Petit, 2012] é uma alternativa interessante para lidar com esse problema. O processo é configurável através de elementos chamados *checkers*. O professor pode definir se a avaliação das submissões será estrita ou se cabe algum nível de flexibilização. Isto é particularmente interessante do ponto de vista do ensino de programação. Há problemas de programação, por exemplo, que podem ser resolvidos de maneiras distintas, de modo que o resultado apresentado na saída, não siga exatamente a mesma ordem para todas as abordagens de solução possíveis. Forçar que a saída do aluno esteja exatamente na mesma ordem da saída produzida pela solução de referência pode causar impacto negativo na liberdade criativa do aluno. O Jutge oferece diferentes níveis de flexibilizações na avaliação das submissões usando os *checkers*.

4.4 Testes

Em geral, os sistemas baseiam-se em testes de entrada e saída ou testes de unidade previamente cadastrados pelo autor do problema. No Putka [Trampus, 2012], os testes são realizados em máquinas de diferentes arquiteturas e configurações de hardware, de modo que o tempo de execução e o consumo de memória possa ser corretamente monitorado.

Nos sistemas em que os testes dos alunos são avaliados, deve ser criado um arcabouço para viabilizar esta avaliação e, definir como o resultado da mesma influenciará na nota do aluno. No ProgTest [de Souza, 2012], o autor do problema fornece, além de uma solução de referência um conjunto de casos de testes com total cobertura do programa. Na nomenclatura adotada pelo sistema, este é o "trabalho oráculo". Os testes são realizados utilizando ferramentas próprias, que integram-se ao sistema como plugins para a realização de testes de: unidade, estruturais e baseados em erros. A avaliação dos testes fundamenta-se, especialmente, na teoria de Análise de Mutantes. O professor define os pesos dado à avaliação dos testes e do programa para compor a nota do estudante.

4.5 Segurança

O principal problema de segurança experimentado pelos sistemas de AA é comum para todos eles: as submissões podem conter códigos maliciosos que afetem o sistema onde eles são executados para avaliação. Ou, por outro lado, o código submetido pode sofrer adulteração na máquina em que ele está sendo avaliado. Surpreendentemente, poucos sistemas relatam alguma preocupação ou medida tomada para enfrentar estes desafios, como já relatado por Ihantola *et al* (2010) em sua revisão. Os AA que surgiram a partir de juízes online ou da comunidade de competições de programação são os que incorporam alguma medida de segurança. No Jutge [Petit, 2012], por exemplo, os programas dos alunos são executados em um ambiente *sandbox* com privilégios e acessos ao sistema restritos. O tempo de acesso à CPU, ao *clock* e o uso de memória são controlados. Além disso, as conexões remotas usam protocolos de comunicação SSH ou HTTPS. O Pythia [Combéfis, 2012] segue abordagem semelhante. Já no Putka [Trampus,2012], as chamadas ao sistema realizadas pelos programas são interceptadas e analisadas, para posteriormente, serem autorizadas.

4.6 Feedback

A principal motivação para o uso de AA nos cursos de programação é a possibilidade de se obter feedback, rápido, padronizado e relevante. É importante avaliar como o feedback sobre os submissões dos alunos vêm sendo produzidos e se, efetivamente, eles têm contribuído para a melhora da qualidade dos programas produzidos pelo aluno, bem como sua motivação e consequentemente, o seu desempenho no curso. Os AA que fornecem uma nota automática para o aluno devem ser capazes de oferecer algum grau de flexibilidade para a configuração da composição das notas.

A maioria dos artigos não dá a devida ênfase ao feedback ou não detalham de que forma ele é produzido. Em geral, é mostrado para o aluno os erros de compilação ou sintaxe caso existam e os erros nos testes providos pelo professor. Alguns sistemas mostram os casos de entrada que fizeram o programa falhar. Essa abordagem costuma, fazer com que o aluno programe usando o método tentativa-e-erro. Preocupação

semelhante é relatada em [Pelz, 2012], já que aquele sistema permite testar se a submissão do aluno apresenta algumas estruturas obrigatórias para a criação do código da solução. O feedback fornecido neste tipo de teste era usado pelos alunos para tentar "adivinhar" como o programa deveria ser escrito. As soluções para mitigar estes problemas podem ser simples como, caracterizar alguns casos de testes como "secretos" e não divulgar as entradas destes casos. Ou, como em [Brown, 2012] mostrar relatórios de feedback diferentes: antes e depois do prazo para a entrega das atividades. Os relatórios mais completos são apresentados ao aluno posteriormente.

Nos sistemas em que os AA estão integrados a ambientes de aprendizagem, o feedback fornecido, muitas vezes, inclui sugestões de um novo conjunto de exercícios que o aluno pode começar a responder ou qual unidade de conhecimento ele deve ler ou trabalhar mais. A personalização dos caminhos que levam ao aprendizado, a partir dos resultados das avaliações automáticas, parece ser a tendência mais forte na evolução dos sistemas desta área.

5. Discussão e Conclusões

Os sistemas de avaliação automáticos (AA) são uma peça importante no processo de ensino e aprendizagem dos cursos de programação. A adoção de tais sistemas amplificou a possibilidade de dar feedback aos alunos sobre suas respostas aos exercícios de programação o que, por conseguinte, permitiu que os professores disponibilizassem mais exercícios para os alunos. Estudos mostram que a quantidade de exercícios realizados pelos alunos tem papel importante no desempenho ao final do curso [Araujo, 2013], o que parece estar de acordo com as experiências que muitos professores têm em sala de aula.

Neste trabalho, evoluímos os estudos sobre o desenvolvimento de sistemas automatizados de avaliação de atividades de programação [Ihantola, 2010] partindo do ano de 2010 até os dias atuais. A seção 4 procurou responder a primeira pergunta de pesquisa apresentada na metodologia deste trabalho: "Quais são as características dos sistemas de avaliação automatizados relatados na literatura após 2010?".

Já a segunda pergunta de pesquisa questiona: "Para quais direções estes sistemas impulsionam os trabalhos futuros?". A tendência da integração dos AA com ambientes virtuais de aprendizagem promete sistemas mais abrangentes. Isto permite a centralização de esforços do professor em um só conjunto de software, do qual o AA faz parte, para a criação e gerência de seus cursos. Além disso, a personalização do aprendizado é outra área de pesquisa para a qual os sistemas de AA podem ser de grande relevância. Atualmente, os dados produzidos e mantidos por estes sistemas são foco de bastante interesse nas pesquisas que utilizam as técnicas de *Learning Analytics* (LA). Percebemos que a integração dos AA com os ambientes de ensino, a personalização do ensino e LA é o que impulsiona, agora, os trabalhos nesta área.

Como limitações deste trabalho, apontamos os possíveis erros de execução no processo de revisão descrito na metodologia. Principalmente, devido ao fato de usarmos na, mesma pesquisa, bases de dados nacionais e internacionais. Além disso, a restrição imposta pela sentença de pesquisa adotada e pelos veículos de publicação escolhidos, que são assunções do trabalho, podem não cobrir todos os sistemas que seriam de nosso interesse.

Como trabalhos futuros, vislumbramos ampliar o espectro de pesquisa para outros jornais, conferências e workshops, como o *Frontiers in Education* e o workshop de Informática na Escola – WIE, não contemplados neste primeiro estudo. Além disso, pretendemos incluir sistemas existentes, que são referência na área, e que ainda passam por ativo desenvolvimento e pesquisa: como o Web-CAT [Edwards, 2004] e Marmoset [Spacco, 2006]. Tais sistemas ficaram de fora de nosso estudo por haverem sido criados antes de 2010, muito embora figurem em publicações recentes, mostrando novos avanços nos sistemas.

Referências

Ala-Mutka, K. (2005). A survey of automated assessment approaches for programming assignments, Journal of Computer Science Education 15(2), 83 102.

Araujo, E. C., Gaudencio, M., Menezes, A., Ferreira, I., Ribeiro, I., Fagner, A., Ponciano, L., Morais, F., Guerrero, D. S., e Figueiredo, J. A. (2013). O papel do hábito de estudo no desempenho do aluno de programação. In *Workshop de Educação em Computação, Congresso anual da SBC 2013,* Maceió, Brasil. SBC.

Aureliano V. C. O., Tedesco P. C. A. R. (2012) Ensino-aprendizagem de Programação para Iniciantes: uma Revisão Sistemática da Literatura focada no SBIE e WIE. In: 23º Simpósio Brasileiro de Informática na Educação, Rio de Janeiro, Brasil.

Brereton P. , Kitchenham B. A., Budgen D., Turner M., e Khalil M. (2007). Lessons from applying the systematic literature review process within the software engineering domain. *Journal System Software* 80, 4, 571-583.

Brown, C., Pastel, R., Siever, B., & Earnest, J. (2012). JUG: A JUnit generation, time complexity analysis and reporting tool to streamline grading. In*Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education* (pp. 99-104). ACM.

Cardell-Oliver, R. (2011). How can software metrics help novice programmers? Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114 (ACE '11) Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 55-62.

Cheang, B., Kurnia, A., Lim, A., e Oon, W. (2003). On automated grading of programming assignments in an academic institution. Comput. Educ. 41, 2, 121-131.

Combéfis, S., de Saint-Marcq, V. L. C. (2012). Teaching programming and algorithm design with pythia, a web-based learning platform. *Olympiads in Informatics*, *6*, 31-43.

de Souza, D. M., Maldonado, J. C., & Barbosa, E. F. (2012). Aspectos de Desenvolvimento e Evolução de um Ambiente de Apoio ao Ensino de Programação e Teste de Software. In *Anais do Simpósio Brasileiro de Informática na Educação,* Vol. 23(1).

Douce C., Livingstone D., e Orwell J. (2005). Automatic test-based assessment of programming: A review. ACM *Journal of Educational Resources in Computing*. Vol 5(3) – 4.

Edwards, S. H. (2004). Using software testing to move students from trial-and-error to reflection-in-action. SIGCSE Bulletin, Vol 36(1), 26–30.

Gikandi J.W., Morrow D., Davis N.E. (2011), Online formative assessment in higher education: A review of the literature, *Computers & Education,* Vol 57(4), 2333-2351.

Ihantola, P., Ahoniemi, T., Karavirta, V. e Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. In *Anais do 10º Koli Calling International Conference on Computing Education Research (Koli Calling '10)*. ACM, 86-93.

Kolstad, R. Infrastructure for contest task development (2009). *Olympiads in Informatics*, 3:38--59.

Pears, A., Seidman, S., Eney C., Kinnunen P. e Malmi L. (2005). Constructing a core literature for computing education research. *SIGCSE Bulletin*. 37, 4, 152-161.

Pelz, F. D., de Jesus, E. A., Raabe, A. L. (2012). Um Mecanismo para Correção Automática de Exercícios Práticos de Programação Introdutória. In*Anais do Simpósio Brasileiro de Informática na Educação*. Vol. 23, No. 1.

Petit, J., Giménez, O., & Roura, S. (2012). Jutge. org: an educational programming judge. In *Anais do 43º ACM Technical Symposium on Computer Science Education*. 445-450.

Píccolo, H. L., Sena, V. D. F., Nogueira, K. B., da Silva, M. O., & Maia, Y. A. (2010). Ambiente Interativo e Adaptável para ensino de Programação. In *Anais do Simpósio Brasileiro de Informática na Educação* Vol. 1, No. 1.

Revilla, M. A., Manzoor, S., e Liu, R.. Competitive learning in informatics: The uva online judge experience (2008). *Olympiads in Informatics*, Vol 2, 131-148.

Santos, J., Ribeiro, A. R. (2011). JOnline: proposta preliminar de um juiz online didático para o ensino de programação. In *Anais do Simpósio Brasileiro de Informática na Educação*. Vol 1, 964-967.

Spacco, J., Pugh, W., Ayewah, N., and Hovemeyer, D. (2006). The Marmoset project: an automated snapshot, submission, and testing system. *In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, 669–670.

Trampus, M., & Urbančič, J. (2012). Putka: A web application in support of computer programming education. *Olympiads in Informatics*, Vol *6*, 205-211.

Verdú, E., Regueras, L. M., Verdú, M. J., Leal, J. P., de Castro, J. P., & Queirós, R. (2012). A distributed system for learning programming on-line. *Computers & Education*, *58*(1), 1-10.

Wang, T., Su, X., Ma, P., Wang, Y., Wang, K. (2011). Ability-training-oriented automated assessment in introductory programming course. *Computers & Education*, *56*(1), 220-226.

Appendix B

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

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}@dsc.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.

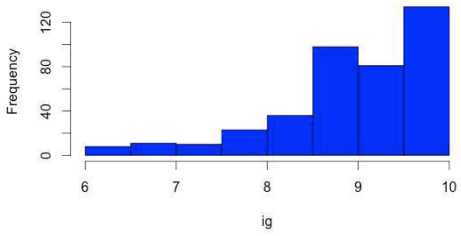

Fig. 1. Distribution of manual grades assigned to functionally correct submissions

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.

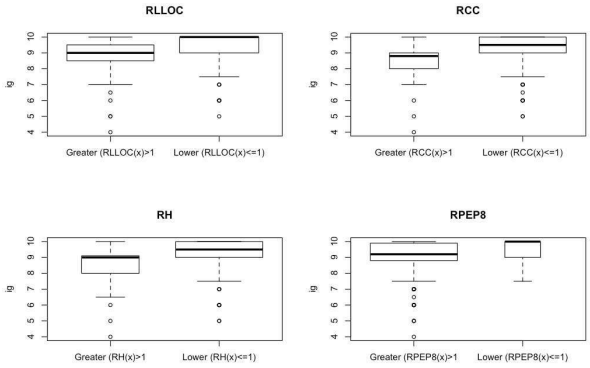

Figure 2. Boxplot of instructor's grades and the values of each RLLOC, RCC, RH and RPEP8

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:

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

```
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.

Appendix C

Applying Spectrum-based Fault Localization on Novice's Programs

Applying Spectrum-based Fault Localization on Novice's Programs

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

***Abstract*—Most introductory programming courses count on automated assessment systems (AAS) to support practical programming assignments and give fast feedback. AAS usually rely on tests results to check the program's functional correctness to provide feedback to students. Novice programmers, however, may find it difficult to map such feedback to the root failures' cause in their programs. It can be even more frustrating when the code is "almost right". In this paper we investigated the use of a fault localization technique on programs produced by students of introductory programming. Our proposed approach is grounded on spectrum-based fault localization (SBFL). The results of our empirical study showed that this lightweight technique is promising. It can be easily adapted to different AAS to generate useful feedback not only to students but also to instructors. We also delineate the scope where SBFL use is jeopardized. The main contribution of this paper is to present the benefits and drawbacks of applying SBFL, in the context of programming learning, as a novel source of information about students' programming assignments faults.**

Keywords— programming, automated assessment; software fault diagnosis; novices, experimentation.

I. INTRODUCTION

Nowadays, many programming courses are supported by automated assessment systems (AAS) that provide feedback to the students and also collect data about their interaction with the instructional material. However, the feedback provided by those systems about the students' difficulties in programming assignments are distant from the instructors' enriched feedback. The problem is that AAS may not provide adequate feedback in some phases of the programming process, so that students may feel frustrated and face difficulties to proceed autonomously on their learning pathway.

AAS usually rely on tests results to check the program's functional correctness to provide feedback to students. Novice programmers, however, may find it difficult to map errors in their code with failing test cases [1]. It can be even more frustrating when the code is "almost right". Sometimes, even if the student knows how to solve the proposed problem, she may fail in producing a functionally correct implementation. The failure revealed by tests can be caused by a wrong operator ("greater than" instead of "greater than or equal to"), a wrong value on the "if" conditional statement or even a misplaced parenthesis. An adequate feedback in this situation would help and stimulate the student to solve the problem and move on. In fact, there are on the literature different strategies proposed to find bugs on student code [1][2]. However, they may not be easily adopted by whichever programming course, as they increase instructors' duties requiring the production of new artifacts.

This paper investigates the use of a lightweight fault localization technique on programs produced by students of introductory programming. Spectrum-based fault localization (SBFL) has been used successfully in different areas of software development [3][4][5]. This technique relies on program spectra: program traces that reveal which parts of the code are active during a failed or successful execution. SBFL predicts the likelihood of a software component, for example, to be responsible for faulty executions. This research focuses on programming assignments proposed along with a test cases suite that are automatically executed by an AAS. In this sense, those systems would compute SBFL measures at a low cost. It does not demand artifacts different from those instructors are used to provide.

We conducted an empirical study to investigate the suitability of using SBFL on novice's programs as a novel source of information aimed to AAS feedback generation. We collected data from an entire edition of an introductory programming course comprising more than 10,000 Python programs, referring to almost 300 programming assignments, from approximately 100 students. We analyzed the tests results of each program submission to characterize them. We observed that 25.9% of the submissions in the data set were considered incorrect, as they did not pass the complete set of tests. In order to be adequate to SBFL use, the submission has to pass at least one test. In this sense, 61.6% of incorrect submissions are initially adequate to SBFL application. A broader exploratory study was able to characterize these programs and provides a more comprehensive knowledge of the extent of situations where the technique could be relevant.

Then, we performed a quantitative study with 5 programming assignments to assess the quality of SBFL diagnostics. We used as baseline instructor's assessments and annotations on the programs. On average, using SBFL, it is necessary to look in only ~20% of the program's lines of code to find the flaw. This study also corroborated with the previous findings on literature. We discuss situations where SBFL was inappropriate to provide feedback about the programs' faults.

The contributions of this work, addressed to instructors and AAS developers, are the following:

- We present and adapt SBFL as lightweight alternative to find faults in students programs. It is a new source of information for feedback generation to instructors or students. Instructors or AAS developers must be responsible for modulate the information before deliver it to students, so it could make better sense in pedagogical context.
- We discuss the use limitations of this technique towards introductory programming assignments, in particular Python procedural programs, as lessons learned from an exploratory study.
- We report a case study evaluation, on real programming assignments, highlighting good results in terms of diagnostic accuracy.
- We summarize strategies on how to maximize SBFL use in programming learning context and propose them as future works.

II. Related Works

Automated assessment systems are used for decades in programming learning context [6]. In general, AAS employ comparable approaches and provide similar features [7]. The most common feature assessed by them is code functional correctness. A typical system executes a set of test cases, provided by the instructors, and compares the expected output to the observed output produced by students' programs. Some systems, characterized as grading systems [8], use those results to grade the programming assignment. Grading systems may weigh another factors, besides correctness, such as deadline penalty, resubmission times, type of errors, test coverage [9][10], etc. AAS may also provide features such as quality assessment, in terms of: efficiency [11], static software metrics [12] and programming style [13]. The work hereby described, focuses on fault localization [1][2] and code repair strategies [1], which are discussed in more details in the following subsection.

A. Fault localization and repair

The approach adopted in [2], to localize bugs in student code and provide feedback, is based on the automatic generation of program execution traces. An execution trace is a list of each program execution step, line by line, and the value of the variables at each time. By reading these traces, students can understand their program execution path and how it has evolved to reach the end. In order to generate feedback to students, the authors suggested comparing students' trace to the one generated from the instructors' reference solution. This works resembles the approach here presented, as it is also based on execution traces. However, SBFL can go further as it can map faults to software elements. The code is an artifact students are used to deal with, differently from an execution trace.

In another way, Singh and colleagues' work tries to identify the error in the students' code and guides them to it correction [1]. The authors argue that most of students' errors in programming assignments are predictable as students who are solving the problems were exposed to the same classes and learning materials. For these and other reasons, their errors tend to follow a typical pattern. They generate feedback based on possible fixes to error models that are typically found in particular programming assignments. Their approach could provide detailed information about the error localization and how to solve it. It also allows the message customization according to the level of feedback the teachers want students to see.

However, to use this approach instructors must provide, in addition to the assignment's reference solution, the model of typical mistakes that could be made by students in that assignment. Errors must be described in an Error Model Language - EML proposed by the authors. This approach has been successfully evaluated in MIT online and regular introductory programming courses. We argue that the overhead required to use Singh and colleagues' proposal is higher than to use our approach. We speculate that having the instructors to foresee every error possibility and also learn a new language to model them is a big hurdle to impose. SBFL is simple and easily adaptable to existing AAS, as it does not require additional artifacts besides the test cases already provided by instructors. In contrast, the precision level of faults localization in our approach is lower than the observed with Singh's approach.

In a very recent work, Edmison and Edwards evaluated the use of SBFL on object-oriented programming learning context [14]. They recognized it as a "feasible strategy" to provide feedback on where to look for faults on programs. Differently from our work, addressed to novice programmers, the authors focus CS2 students, which are not complete beginners as they are taken their second or third programming course. Furthermore, the work deals with objected-oriented Java programs with the aim to locate and identify what methods are most likely to contain the fault. Our proposal has a finer granularity as it ranks the lines where the fault could be found in a procedural Python program. In addition, our research goes a step further as it discusses when not to apply SBFL. As a result of an exploratory study, performed in a dataset from over 10,000 programs submissions, we characterized the students' solutions and discussed the scope of the technique: when and why it is useful. The present work considers practical significance of the results as it gives insights into how to make better use of the SBFL in programming learning context.

III. Background

In this section we describe the key concepts related to Spectrum-based Fault Localization (SBFL) technique and how we have adapted it to introductory programming learning context. Introductory programming assignments are usually well-formed specifications of problems to produce relatively small programs. These programs receive inputs and transform them in testable outputs. In this setting, *faults* can be seen as bugs in the programs and *failures* are evidenced by unexpected outputs for a given input [3].

A. Spectrum-based Fault Localization

SBFL is a technique that dynamically analyses a program in order to calculate the likelihood of a given *component* to be faulty. For diagnosis purpose, the concept *component* stands for an element of the system considered to be atomic. In multiple application of SBFL, *components* can be mapped to different targets: blocks of code when analyzing industry software systems [3]; cells in case of spreadsheets analysis [4]; agents when examining multi-agent systems [5] and methods in the study of object-oriented student programs [14].

The idea is to observe multiple runs of the program, where components are exercised in failed and passed executions and calculate how a component is "suspicious" to be faulty. Failure detection is a precondition to fault localization: it is necessary to recognize that something is wrong before trying to locate the fault [3]. In this scenario, we use test cases provided by instructors to each programming assignment. However, seeking for failures through test cases are an elementary way of detecting faults. Some of them may not be disclosed if the set of test cases were not complete. Provided we cannot guarantee this completion, we assume, in this study, that all program's faults are revealed by test cases. In this sense, a failed run occurs when an error is detected – the expected output is different from the observed. On the other hand, a passed run occurs when the output is equal to the expected.

The data collected from failed/pass runs are used to compose a hit-spectra matrix, see Fig. 1. This is an NxM matrix; where N represents the number of *components* inspected in the program and M the number of runs (test executions, for example). Each a_{ij} element of the matrix corresponds to a binary value: (1) if it was hit in that particular run and (0) in the contrary [4]. In practice, this means that we aim to identify which component is "involved" in a failure. Another necessary element used to calculate components' suspicious in SBFL is the error vector. This N-length binary vector holds the information about "fail" and "pass" to N runs, see Figure 1.

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & \dots & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ \dots \\ e_n \end{bmatrix}$$

$$s_1 \quad s_2 \quad \dots \quad s_n$$

Fig. 1. MxN hit-spectra matrix and N-length error vector. Each column represents a *component* spectrum

After computing the hit-matrix and error vector, the next step is to identify which column in the matrix resembles most the error vector. Similarity coefficients, which are largely known in the literature, are used in this activity. Passos and colleagues cited in their work the use of more than 40 heuristics to compute similarity between vectors [5]. The idea is simple: the more a *spectrum* of a given component is similar to the error vector, the more it is suspicious to be the cause of the detected error.

In this work, we used Jaccard similarity coefficients in order to calculate the value of "suspiciousness index" for a given component. Refer to [3] in order to obtain more information about how to compute those coefficients in SBFL context.

Finally, the coefficient values assigned for each component are ranked in descending order (most similar figures on top most positions). It means that, in order to find the fault in a given code, it is recommended to inspect the components following the SBFL ranking order. It can be noticed that the accuracy of this technique diagnosis is limited: it is a recommendation not a prescription. However, SBFL merit is to greatly reduce the range of code inspection. In the work reported by [3], it exonerated, on average, 80% of the blocks of code of being faulty.

B. Students' Programs Fault Localization with SBFL

We argue that using SBFL to generate information to provide automatic feedback in the context of programming learning is relatively simple. AAS are increasingly been used in programming courses. They can be used to calculate SBFL coefficients and compose the rank, as they already count on a set of tests provided by instructors.

In this paper, we applied SBFL to Python programming assignments of an introductory programming course. Each *component* of the technique is mapped to one line of the program, excluding comments or blank lines. We rely on the set of tests provided by the instructors in order to thoroughly test the code. In this sense, the diagnostic accuracy of the strategy also relies on instructors test quality.

Fig. 2 presents an example of real student code. This assignment specification asks students to write a program to calculate the body mass index of males and females. The value of suspiciousness index *s,* for each line, is showed on the left side. It can be observed that the last two lines (7, 8) obtained the highest values of *s*. The faulty line of code is indeed the last line (8), as the variable used should have been *bmi_female* instead of *bmi_male*.

```
0.5 | 1. genre = raw_input()
0.5 | 2. height = float(raw_input())
0.5 | 3. bmi_male = 72.7 * height - 58
0.5 | 4. bmi_female = 62.1 * height - 44.7
0.5 | 5. if genre == "m" or genre == "M":
0.0 | 6.     print "%.03f" % bmi_male
1.0 | 7. elif genre == "f" or genre == "F":
1.0 | 8.     print "%.03f" % bmi_male
```

Fig. 2. Sample of student code is on the right. Values of suspiciousness index of each line are on the left.

IV. Research Methodology

In this section, we are going to present the research methodology applied in the empirical study of this work. In order to investigate the applicability of SBFL on novice programs, we followed a two-pronged approach: an exploratory study and a case study. The first aimed at having a broader look on students' code production, qualitatively evaluating their errors and evaluating the soundness of SBFL

in the context of programming learning. The latter intended to quantitatively evaluate the approach, measuring its accuracy and other metrics, in a given set of real students' programs sample.

A. Data Collection

This research was performed in an introductory programming course with undergraduate students of Computer Science. In this course students learn programing skills using Python language. Students learn how to use expressions, alternative statements, collections, strings, collection-controlled loops, conditional-controlled loops and functions. Their laboratory activities focus on solving problems by coding programs and submitting them to an AAS to be automatically tested. Each programming assignment presents a basic input and output test. Students are used to test their programs with this basic test before submitting their solution to the system. After the code submission to the AAS, additional hidden tests are executed. As a result, students receive the number of tests failed and passed for that code. They are not penalized for multiple submissions for the same question.

In the exploratory study, we collected students' submissions for programming assignments of a complete course edition. They were collected using an instrumented AAS, which was developed in-house and tailored for our course. The dataset of the exploratory study comprises 10,357 programs, referring to 277 programming assignments, from 115 students. On average, each student submitted 90 programs along the course. Some of them were selected to a more in-depth qualitative analysis.

In the case study, we selected a set of 5 assignments focusing on different programming learning outcomes expected in the course, such as conditional structures (if), iterations (for and while), simple algorithms with data structures (lists). We seek for assignments that students faced difficulties to succeed. For this reason, of 181 submissions that composes this study dataset, 53.6% failed in at least one test case. It is worthy to note that each assignment has their own set of tests and they were also necessary to compute SBFL values.

B. Exploratory Study

In general, the purpose of an exploratory study is to answer research questions about the studied phenomena without formulating any previous hypothesis [15]. In this study we investigated the suitability of using SBFL in different configurations of defects observed on students' programs. From this point on, we are going to refer this "configurations of defects" as scenarios. Our main purpose is to define the scope of action of SBFL in programming learning context. For this reason, we want to answer the research questions:

RQ1) What are the preconditions to use SBFL in students' programs, according to their tests' results?

RQ2) SBFL performance depends on the programs' defects configuration?

RQ3) Which are the scenarios of defects configuration in which SBFL performance is good or is jeopardized?

SBFL is a technique that dynamically analysis a program in order to calculate the likelihood of a given *component* to be faulty. In this study, each logical line of the code (program lines excluding comments, headers and blank lines) is considered a *component*. It means that, the likelihood of being faulty is calculated for each line according to the technique algorithm. This value is referred as index *s*, for suspicious. It represents the similarity between the line *spectrum* and error vector (see Section III.A). We chose Jaccard heuristic to compute the similarity coefficient in this study because it is a simple strategy that yields good results on related work [5].

Initially, we inspected the dataset and determined which program submissions could be used as subject of this investigation. Such definitions of criteria helped us to answer the first research question. In sequence, we executed SBFL strategy to calculate *s* indexes for each program line and to create the rank. These are the steps of the process: Firstly, for each test of the test suite associated with that programming assignment, we generate an execution trace of the program. Each trace has a set of lines that represents the lines exercised during the test execution. Secondly, we compute the hit-matrix and the error vector (Fig. 1) for those executions. Then, we calculate the *s* value of each line using the Jaccard similarity coefficient S_J described in (1) with the values from Table I [3].

In Table I, C_{11} represents the number of failed tests that executed that line. C_{10} is the number of passed tests that hit such line. C_{01} is the number of failed tests that do not exercise the line and C_{00} is the number of passed tests that do not hit the line. All those values are calculated from the hit-matrix and error vector. Finally, we create a rank with the values of *s*. In theory, the lines more likely to be faulty are on the initial ranking positions.

$$S_J = \frac{C_{11}}{C_{11} + C_{10} + C_{01}} \quad (1)$$

TABLE I. DICHOTOMY TABLE REPRESENTING THE STATES OF A LINE

Test Result	**Line Hit**	
	Yes = 1	***No = 0***
Failed = 1	C_{11}	C_{01}
Passed = 0	C_{10}	C_{00}

The index *s*, calculated using Jaccard similarity heuristic, can range from 0 to 1. With this in mind, we perceived four possible situations when we observed the SBFL rank and compared with the real localization of defects in students' programs. Table II presents these scenarios.

In order to answer posed research questions, we searched the dataset and found programs that match each one of these situations. We performed a qualitative evaluation in such programs to better understand SBFL performance on those cases. Furthermore, there were other programs that did not fit in those categories and presented notable characteristics such as: multiple lines of errors, dead code and runtime errors. They also helped on the definitions of SBFL applicability scope.

TABLE II. SCENARIOS OF DEFECTS OBSERVATIONS IN REGARDS TO SBFL SUSPICIOUSNESS INDEX.

s	**Real Defects Found**	
	Yes = 1	***No = 0***
High (>=0.5)	S_1	S_2
Low (< 0.5)	S_3	S_4

C. Case Study

In the case study, we conjectured if SBFL could really help us to identify which are the lines responsible for the faults observed in students' programming assignments. So that, the values obtained through the technique could be used to generate feedback in the context of programming learning. In order to test it, we applied SBFL and used a set of evaluative metrics to analyze its results and practical significance. To conduct this study, we formulated the following research question:

RQ4) Is the quality of diagnosis delivered by SBFL good enough to generate useful feedback about faults localization in novice programming assignments?

In answering this research question, we applied SBFL technique to a set of programming assignments, as described on the previous section. This set of assignments was collected from mid-term exams of different course editions of Programming I. Experienced instructors manually assessed and annotated the programs highlighting its errors. We executed automatic tests and collected faulty program submissions. We also manually analyzed these submissions to make sure that the faulty lines were indeed identified. This inspection was work intensive, but it was fundamental to this study. Its results was used to compose the baseline of the study, an oracle of "true positives" faults, used to compute the evaluation metrics.

We instrumented the AAS to calculate *s* indexes and generate SBFL rank. The order in this rank indicates the likelihood of a line to be faulty. After applying the technique for each submission, we evaluated the success of the fault localization in contrast to the baseline using different metrics. Precision and recall are traditional metrics of information retrieval. They are used in this study with the following meaning:

- *Precision*: Measures the fraction of the "number of lines marked as faulty by SBFL, which are real faulty lines according to the baseline" by the "total number of lines indicated by SBFL".
- *Recall:* Measures the fraction of the "number of lines marked as faulty by SBFL, which are real faulty lines according to the baseline" by the "number of faulty lines according to the baseline".

In addition, we used another metrics proposed by Abreu and colleagues to evaluate SBFL diagnosis quality in terms of *accuracy* (*qd*) and *quality of the error detection* (*qe*) [3]. We are going to briefly describe the equations and the meaning of its compound values. Finer details about the underlying motivation can be found in [3].

Accuracy represents the quality of diagnosis of the technique in locating a faulty line along the program. It means the percentage of the program lines that does not need to be inspected when searching for a fault by traversing the ranking.

Let $d \in \{1, \ldots, N\}$be the index of the faulty line. For all $j \in \{1, \ldots, N\}$, s_j is the similarity coefficient calculated for the line *j*. The ranking amplitude τ also considers that when two lines have the same similarity coefficient, we use the average ranking position for them. The first term $|\{j|s_j > s_d\}|$ counts the number of lines ranked before the faulty line. The term $|\{j|s_j \geq s_d\}|$ calculates the number of lines with the same or higher similarity coefficient compared to the faulty line [5].

$$\tau = \frac{|\{j|s_j > s_d\}| + |\{j|s_j \geq s_d\}| - 1}{2} \quad (2)$$

The value of accuracy is calculated considering the rank amplitude τ and the total number of lines of code N, according to (3).

$$q_d = \left(1 - \frac{\tau}{N-1}\right).100\% \quad (3)$$

The metric *error quality detection* aims to quantify a problem of diagnosis based on the observation of tests results. An error only appears when the faulty line is exercised by a test case. In this sense, the purpose of this metric is to measure the "unambiguity of the passed/failed" data in relation to the fault being exercised [3]. Equation 4 computes the metric using the definitions of Table I:

$$q_e = \frac{C_{11}}{C_{11} + C_{10}}.100\% \quad (4)$$

V. RESULTS

A. Exploratory Study

The dataset of this study contains 10,357 students' programs: 7,670 passed all tests and 2,687 failed at least one test. We are investigating what are the preconditions to use SBFL in students' programs, according to their tests' results. As SBFL is a technique to locate faults, clearly we are not interested on functionally correct submissions, i.e. when they pass all test cases. The requirements to use SBFL in students' programs, is to have traces of failed and passed executions. In the subset of codes that failed at least one test 1,031 codes did not passed any test (38.37%). It means that we can apply SBFL in 1,656 of 2,687 codes with defects (61.63%).

Although this result could be considered a large number, for the dataset we evaluated, it is important to understand what kind of submission is not "suitable" for using SBFL as a strategy for fault localization. Code submissions that did not pass in any test case may present failures on the algorithm or strategy to solve the problem. Possibly the student does not understood the problem specification or does not know how to program it correctly and need to start its code over. Depending on the test suite, it also may occur that a program passes "by

chance" in few test cases. Overall, SBFL is not suitable for these cases, as its purpose is to help to pinpoint faults. Students whose code submissions are "almost right" can benefit better of the information obtained from SBFL results.

In order to study the functioning of the technique in regards to the programs' defects configuration, we sought the dataset in order to locate programs that fits on the scenarios S_1, S_2, S_3 and S_4, as defined on Table II. We present examples of real student code for each scenario and discuss how the suspiciousness index *s* can be interpreted. Then we highlight the lessons learned when looking for defects using SBFL, in programs with such defect configurations. For each example code, the values of *s* are on the left side and the real defects are underlined.

S_1) Defect found in line with high *s* value. In this scenario, the faulty line is on the top positions of SBFL rank. This could be considered the ideal case: as *s* index is high. One who is looking for defects in the code can find it almost directly. Fig. 2 (of Section III) shows an example of program of this scenario. This program was tested against four test cases. It passes two and failed other two tests. Lines whose *s* value is 0.5 were executed in all test cases. The value of *s* of line 6 is 0.0, as this line was not executed in a failed test. The value of *s* of line 7 and 8 is 1.0, as they are executed in failed runs. In fact, line 8 is only executed when the test fails, as the real defect is found on it. Lines 7 and 8 are the top-ranked lines according to SBFL.

S_2) Defect not found in line with high *s* value. In this scenario, the faulty line is not on the top position of SBFL rank. The program example, showed in Fig. 3, was tested against 5 test cases and failed 2 tests. In this scenario, the faulty line (line 7) is not the top most line on SBFL rank. The highest value of *s* indexes is 1.0, corresponding to line 6. This happens because this line is executed in all the failed runs. Although the faulty line is not on the first rank position, it is one of the top most lines ranked. In this sense, one who is looking for defects in the code can find it in few lines attempts. So, this scenario also represents a successful case of SBFL use. This program can be fixed if we substitute the *if* statement on line 7 by an *elif* when checking if the sum is divisible by three.

```
0.4 | 1. num1 = int(raw_input())
0.4 | 2. num2 = int(raw_input())
0.4 | 3. num3 = int(raw_input())
0.4 | 4. sum= num1 + num2 + num3
0.4 | 5. if sum % 3 == 0 and sum % 5 == 0:
1.0 | 6.     print "fizzbuzz"
0.6 | 7. if sum % 3 == 0:
0.0 | 8.     print "fizz"
0.0 | 9. elif sum % 5 == 0:
0.0 | 10.    print "buzz"
```

Fig. 3. Student code with the suspicious value on the left side and the faulty statement underlined. Example of scenario 2.

S_3) Defect found in line with low *s* value. In this scenario, all lines have low values of index *s*. This indicates that all lines were executed in at least one successful test. However, some lines were not executed on failed runs. Although, no line in this example presented high values for the suspiciousness index *s*, when the rank is composed the faulty line (line 9) is in one of the highest positions of the rank, see Fig. 4. The problem of the faulty statement of this program is a truncate division operation. In Python version 2.7, the result of an integer division is truncated which may result in a failure for some inputs: when *b* is not a multiple of *2*a.* To effectively correct this code, it is necessary to convert *int* values into *floats*. This could be done in line 9 or in lines 2 and 3. This ambiguity may make it harder to find a way to correct the fault but the technique still gives a good hint. For this reason, we argue that SBFL, in this scenario, also can help to produce useful information about the fault localization.

```
0.1 | 1. import math
0.1 | 2. a = int(raw_input())
0.1 | 3. b = int(raw_input()
0.1 | 4. c = int(raw_input())
0.1 | 5. delta = b**2 - 4*a*c
0.1 | 6. if delta < 0:
0.0 | 7.     print "no real roots"
0.2 | 8. elif delta == 0:
0.3 | 9.     x = -(b)/(2*a)
0.3 | 10.    print "%s = %.2f" % ('x', x)
0.0 | 11.else:
0.0 | 12.    x1 = (-b+math.sqrt(delta))/(2*a)
0.0 | 13.    x2 = (-b-math.sqrt(delta))/(2*a)
0.0 | 14.    print "%s = %.2f" % ('x1', x1)
0.0 | 15.    print "%s = %.2f" % ('x2', x2)
```

Fig. 4. Student code and the value of suspicious index on the left side of each line and the faulty statement underlined. Example of scenario 3.

S_4) Defect not found in line with low *s* values. In this scenario, the faulty line does not appear on top most positions of SBFL rank. It means that SBFL failed in suggests the lines that contain the real defect. In this example, presented by Fig. 5, the faulty line is line 7. To better understand this situation, we debugged this code and we found that the failure happened because we short-circuit (don't evaluate) the *elif* statement when the second input (dna_2) is lower than the first input (dna_1). Thus, in any situation in which "dna_3 < dna_2 < dna_1" we will observe the same problem. To correct this program, it is necessary to substitute *elif* to an *if* condition.

This scenario is not common to happen. In fact, it was hard to find a situation where a defect was not indicated correctly by the highest SBFL value. We identified that situations like this can happen when the defect is located in the conditional structure, such as if/elif/else. When the alternative condition is accepted the other conditional test is not executed, even in failed runs. This situation poses a great challenge to this strategy.

Other perceptions about SBFL functioning and interpretation were observed on programs that does not fit on the characteristics of the scenarios previously described. However, SBFL can be applied and give us insights about the code execution or defect location. For example, when you have information about lines that were not executed (dead code) is possible to reason about a possible defect on the condition that make that code unreachable.

```
0.1 | 1. dna_1 = raw_input()
0.1 | 2. dna_2 = raw_input()
0.1 | 3. dna_3 = raw_input()
0.1 | 4. small = dna_1
0.1 | 5. if len(dna_2) < len(smal):
0.5 | 6.     smallest = dna_2
0.0 | 7. elif len(dna_3) < len(smal):
0.0 | 8.     smallest = dna_3
0.1 | 9.print "%s %d" % (smal,len(smal))
```

Fig. 5. Student code with the suspiciousness index value on the left side and the faulty statement underlined. Example of scenario 4.

B. Case Study

In this subsection we report on the results of the study that helped us to answer the research question that drove our study: Is the quality of diagnosis delivered by SBFL good enough to generate useful feedback about faults localization in novice programming assignments?

We studied five programming assignments (PA), emphasizing different learning outcomes of programming learning: (1) for loops, (2) sorting, (3) conditional structures, (4) lists and (5) while loops. These PA were chosen due to its high failure rate (53.59%). Table III shows each PA and the data about their total number of submissions and the number of failed submissions. Additionally, it shows the number of submissions suitable for SBFL use, failed submissions that pass at least one test.

TABLE III. PROGRAMMING ASSIGNMENTS DATA ABOUT SUBMISSIONS AND TESTS FAILURES

	Submissions	***Failed***	***Passed at least 1 test***
PA_1	29	24	19
PA_2	31	17	3
PA_3	18	10	5
PA_4	16	8	3
PA_5	87	38	28
TOTAL	181	53.59%	59.79%

The quality of SBFL diagnosis was assessed using the metrics defined in Section IV: precision, recall, accuracy and quality of error diagnosis. PA_2 and PA_4 values were omitted from the results table. Each one has only 3 submissions. The manual evaluation of these programs revealed severe defects caused by multiple lines in 4/6 programs. As we learned in the exploratory study, programs with such characteristics are not suitable to SBFL strategy. Table IV shows the average of the evaluative metrics for the others programming assignments.

TABLE IV. AVERAGE VALUES OF EVALUATION METRICS

	Recall	***Precision***	***qd***	***qe***
PA_1	1	0.18	0.84	0.17
PA_3	1	0.13	0.82	0.79
PA_5	1	0.08	0.73	0.30
Avg	1	0.10	79.67%	42.00%

The first two metrics (*recall* and *precision)* present contrasting values. *Recall* value is perfect for each programming assignment presented on Table 4. It means in practice, that all real faulty lines are successfully detected by SBFL technique. However, the precision value is low, meaning that many lines are detected, but some of them are false positives. This behavior was expected since SBFL strategy includes creating a rank of possible faulty lines to be inspected in a given order, so that a great part of the code could be exonerated of being inspected. It is likely that the top-ranked lines present the defect.

The quality of defect diagnosis is measured on metric *qd*. This measure shows the accuracy of SBFL in terms of the percentage of lines of code that do not need to be considered when searching for a defect on the code. The values obtained for each PA are considerable high. The metric quality of the error detection *qe* measures, in practice, how good is the test suite used to apply SBFL to a given programming assignment. It seems that the values obtained for this metric does not follow any trend. They can be considered low values, meaning that the quality of error detection for our dataset is not good. However, in a deeper analysis, we could not observe correlation between the measurements of: accuracy (*qd*) and the quality of error detection (*qe*). This result corroborates Abreu and colleagues' findings [3]. It means that even if the test suite used to apply SBFL technique results in low error detection quality, the diagnosis accuracy is still high.

VI. DISCUSSION

There is a set of necessary preconditions, regarding tests' results, to use SBFL to locate faults in a given programming assignment. Besides failing in at least one test case, what is obvious as our aim is to locate failures, it is also necessary to pass at least on test case. Though, the test suite must have at least 2 test cases. In dataset we evaluated on the exploratory study, 61.63% of code with failures met these requirements and could be used to assess SBFL.

The performance of SBFL technique depends on the programs' defect configuration. In the exploratory study, we identified four error scenarios and characterized other errors configurations. It helped us to better understand the meaning of SBFL suspiciousness indexes and devise hints on how to look for the fault. For example: If there is dead code in the program, it is important to understand why the test suite did not exercise such lines. Possibly, the defect is on the statement that precludes that code to be executed. Another lesson learned on using SBFL strategy is that if the defect were not found on the lines top-ranked consider analyze the neighborhood. It is worthy to verify conditional expressions near those lines to see if there is some defect there, especially if the defect were not find on top ranked lines.

There are indeed some scenarios in which SBFL performance is jeopardized. This technique is useful to highlight obvious and punctual defects. Programs containing

multiple lines of errors, structural problems or error on its problem solving strategy is not suitable for applying SBFL. Even when manually evaluating punctual defects, there existed few situations where SBFL was not able to locate the fault. The type of feedback generated with SBFL information, in programming learning context, is useful and ideal for situations in which the program is "almost right".

Overall, we considered the quality of diagnosis delivered by SBFL good enough to generate useful feedback about fault localization in novice programming assignments. At least for the dataset on the performed case study, SBFL accuracy was on average 80%. It means that guided by SBFL rank, one just need to scan 20% of the program to find it defect. This result approximates to the values obtained by other studies that applied SBFL in contexts such as spreadsheets [4], multi-agent systems [5] and software products [3].

VII. Conclusions and Future Work

This paper investigated the use of SBFL, a fault localization technique, on programs produced by students of introductory programming. This technique relies on program spectra, defined as a set of program' statements that were active during an execution. It predicts the likelihood of each program statement to be responsible for faulty executions. In order to make better use of this information, regarding to pedagogical context, instructors or AAS developers must fine-tune it before delivering to students.

We discussed how to interpret the values of SBFL suspiciousness index and the limitations of use of this technique. Our exploratory study characterized programs according to their errors configurations in scenarios. We claim, as lessons learned from this study, that SBFL is useful to pinpoint punctual defects. It is worthy to note that this technique is not fail-proof and there exist scenarios where looking only for the top most positions in SBFL rank may not be enough to find the fault. We report a case study evaluation, using real programming assignments, highlighting good results in terms of diagnostic accuracy: using SBFL we just need to look at 20% of the code in order to find the fault. This result corroborates with other studies found on the literature and obtained an approximate result when applying SBFL to other software engineering contexts.

The main contribution of this work, to instructors and AAS developers, is the investigation of SBFL benefits and limitations, as promising lightweight alternative to find faults in students programs and, as a new source of information, for student feedback generation. As future work we address the need of further investigation in the scenario in which the technique performance was jeopardized: when the fault is found on the conditional statement. Furthermore, it would be worthwhile to use item response item theory in order to validate the test suite provided to the programming assignment, since its quality is fundamental to this approach.

Acknowledgment

The authors would like to thank Programming I instructors and students at UFCG for their valuable collaboration. This research was partially sponsored by the agreement No 754664/2010 between UFCG and ePol/DPF.

References

[1] R. Singh, S. Gulwani, and A. Solar-Lezama. "Automated feedback generation for introductory programming assignments". In PLDI, pp 15–26, 2013.

[2] M. Striewe and M. Goedicke. "Using run time traces in automated programming tutoring". In ITiCSE, pp 303–307, 2011.

[3] R. Abreu, P. Zoeteweij, and A. van Gemund. "On the Accuracy of Spectrum-based Fault Localization". In Proc. of Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION), pp 89–98, Windsor, UK, 2007. IEEE Computer Society.

[4] R. Abreu, J. Cunha, J. P. Fernandes, P. Martins, A. Perez, and J. Saraiva, "Smelling faults in spreadsheets," in Proc. 30th IEEE Int. Conf. Softw. Maintenance Evol., 2014, pp. 111–120

[5] L.Passos, R. Abreu and R. J. F. Rossetti., "Spectrum-based fault localisation for multi-agent systems", In Proc. 24th International Conference on Artificial Intelligence (IJCAI'15), 2015, AAAI Press, pp. 1134-1140

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

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

[8] B. Cheang, A. Kurnia, A. Lim, and W. Oon. "On automated grading of programming assignments in an academic institution". Comput. Educ. 41, 2. pp. 121-131, September 2003.

[9] S. H. Edwards, "Improving student performance by evaluating how well students test their own programs", In J. Educational Resources in Computing, Vol 3, 2003, pp.1–24.

[10] H. S. Edwards, J. Snyder, M. A. Pérez-Quiñones, A. Allevato, D. Kim, and B. Tretola. 2009. "Comparing effective and ineffective behaviors of student programmers". In Proc. of the fifth international workshop on Computing education research workshop (ICER '09), pp. 3-14.

[11] S. Gulwani, I. Radiček, and F. Zuleger. "Feedback generation for performance problems in introductory programming assignments". In Proc. of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014), pp. 41-51.

[12] R. Cardell-Oliver, "How can software metrics help novice programmers?". In Proc. of the Thirteenth Australasian Computing Education Conference. Australian Computer Society, Inc. Vol 114, 2011, pp. 55-62.

[13] H. Blau and J. E. B. Moss, "FrenchPress Gives Students Automated Feedback on Java Program Flaws", In Proc. ACM Conference on Innovation and Technology in Computer Science Education, 2015, pp. 15-20.

[14] B. Edminson and S.H. Edwards, "Applying Spectrum-based Fault Localization to generate Debbuging Suggestions for Student Programmers", In Proc. of the 2015 IEEE International Symposium on Software Reliability Engineering Workshops, pp. 93-99.

[15] R. A. Bittencourt, D. M. B. dos Santos, C. A. Rodrigues, W. P. Batista and H. S. Chalegre, "Learning programming with peer support, games, challenges and scratch," *Frontiers in Education Conference (FIE), 2015. 32614 2015. IEEE*, El Paso, TX, 2015, pp. 1-9.

Appendix D

Questionnaire – Do learners think that *qcheck* is useful?

Minha experiência usando o Qcheck

Queremos melhorar a experiência de uso do Qcheck - uma ferramenta que dá dicas de como melhorar a qualidade dos códigos dos exercícios de programação 1. Para isso contamos com o seu valioso feedback. Estas perguntas objetivam descobrir como foi a sua experiência usando o Qcheck.

1. **Endereço de e-mail** *

2. **Em que momento (ou como) você está usando o qcheck na resolução de suas atividades de programação?**

3. **Como você processa (ou interpreta) a mensagem fornecida pelo teste?**

4. **Quais são os pontos positivos e negativos da ferramenta?**

5. **O que impede ou o motiva/motivou a utilizar o qcheck?**

6. **Após uso a resolução dos warnings do qcheck, seu código se apresenta:**
   Marcar apenas uma oval.

	1	2	3	4	5	
Pior que a primeira versão	◯	◯	◯	◯	◯	Melhor que a primeira versão

☐ Envie para mim uma cópia das minhas respostas.

Appendix E

Avaliação da Legibilidade de Programas Escritos por Alunos Iniciantes

Avaliação da Legibilidade de Programas Escritos por Alunos Iniciantes

Eliane Cristina de Araujo[1], Dalton Serey Guerrero[1], Jorge César Abrantes de Figueiredo[1]

[1]Departamento de Sistemas e Computação
Universidade Federal de Campina Grande – Campina Grande, PB – Brasil

{eliane,dalton,abrantes}@computacao.ufcg.edu.br

Abstract. *In this paper, we report on a study that was carried out, in the context of an introductory programming course, to investigate how code readability correlates with the students' achievements. We suggest a simple metric to automatically assess beginners code readability. The study revealed a correlation between code readability and students' course performance. In parallel, we brought to light other factors which, taken together with the readability metric, can better explain students performance in the course.*

Resumo*. Neste artigo, apresentamos um estudo realizado, no contexto de um curso de introdução à programação, em que investigamos a relação entre a legibilidade dos programas produzidos pelos alunos e o seu desempenho na disciplina. Propusemos e avaliamos uma métrica simples de legibilidade de código Python para programas produzidos por estudantes de programação introdutória. Nossos resultados confirmam que há uma correlação entre a métrica de legibilidade dos programas e o desempenho dos alunos, indicando que a métrica captura um aspecto considerado pelos professores na avaliação dos programas.*

1. Introdução

O desenvolvimento de programas por estudantes em cursos introdutórios de programação é uma das atividades que mais pode contribuir na aprendizagem desta disciplina. Entretanto a avaliação destes programas e o consequente feedback dado ao aluno, demanda muito trabalho do professor. Por essa razão, professores têm optado por diminuir a quantidade de programas/exercícios propostos para os alunos (Cheang et al 2003).

Os sistemas de verificação e testes automáticos de programas podem reduzir a carga dos professores e viabilizar o estímulo à produção de muitos programas pelos estudantes. Nesse modelo, os programas são verificados automaticamente e se produz um feedback imediato para o estudantes sobre o programa-solução produzido (Campos et al, 2004). Os verificadores, por se nortearem pelos testes automáticos, consideram para avaliação apenas a resposta produzida pelos programas para um conjunto pré-definido de casos de teste, desconsiderando as qualidades internas do código submetido pelo estudante.

Neste trabalho, apresentamos os resultados de um estudo cujo objetivo foi analisar como a legibilidade dos códigos dos alunos pode ser avaliada de forma automática e como ela se relaciona com o desempenho dos estudantes no curso. Para isso, propusemos uma métrica para avaliar a legibilidade do código produzido por

programadores iniciantes e que pode ser obtida de forma automática. Neste estudo procuramos verificar em que medida a capacidade de escrever programas mais legíveis pode estar correlacionada com o desempenho do aluno no curso.

O trabalho está organizado da seguinte forma: a Seção 2 apresenta o referencial teórico e alguns trabalhos relacionados à legibilidade de códigos. A seção 3 explica a metodologia adotada para realizar o estudo e descreve o contexto de produção dos dados relatando a forma como foram coletados e tratados. A Seção 4 apresenta os resultados alcançados pelo estudo, que são analisados e discutidos na seção 5. Os comentários finais, bem como as idéias para refinar e melhorar o trabalho, são tratados na seção 6.

2. Referencial Teórico e Trabalhos Relacionados

Estudos apresentam diferentes definições sobre a noção de legibilidade. Posnett et al. (2011) consideram que a legibilidade é a impressão subjetiva que os programadores têm sobre o quão difícil de entender é determinado código. Para estes autores, um trecho de código é dito legível se for fácil de ler, compreender e manter. No entanto, esta noção de legibilidade está fortemente relacionada a fatores humanos cognitivos, o que a torna difícil de quantificar e medir. Buse et al. (2010) propuseram uma métrica para medir a legibilidade do código com base em uma pesquisa experimental envolvendo anotadores humanos. Posnett apresentou uma outra proposta, seguindo a mesma linha.

Os modelos para legibilidade de código tanto de Buse quanto de Posnett, foram obtidos experimentalmente observando códigos produzidos sob condições diferentes dos programas de alunos iniciantes. Em geral, os programas de iniciantes são respostas para exercícios de programação propostos pelos professores, podendo ser abertos ou não (Blinkstein, 2011). Estes programas são compostos geralmente por poucas linhas de código. Além disso, cursos introdutórios de programação tendem a não optar pelo paradigma de orientação a objetos, mesmo utilizando linguagens que dêem este suporte. Os programas avaliados pelos autores citados na obtenção de suas métricas são escritos em linguagens orientadas a objetos. Só este fator já é motivo de impedimento para a utilização das métricas por eles propostas.

Neste trabalho, propusemos uma métrica para avaliar a legibilidade dos programas dos estudantes considerando a adesão ao padrão de codificação estabelecido pela comunidade Python que é a linguagem de programação adotada para a escrita de programas no curso. Esta simplificação da análise da legibilidade é amparada pela experiência da comunidade de prática da linguagem através dos objetivos do Guia de Estilo para Código Python, conhecido como PEP08 (Python, 2013) "(...) as regras aqui descritas objetivam melhorar a legibilidade dos códigos Python e torná-los consistentes com o amplo espectro de códigos Python desenvolvidos mundialmente.".

Com uma ferramenta de verificação de conformidade ao padrão PEP08, medimos a quantidade de itens do código que estão em desacordo com este guia de estilo. Neste estudo, a métrica legibilidade foi reduzida em sua complexidade e quantificada de forma inversa: medindo o número de PEP08-defeitos (ao que chamaremos apenas de *defeitos*). Por exemplo: o programa 1 será mais legível que o programa 2 se ele apresentar menos *defeitos* de legibilidade que o outro.

3. Metodologia

A pesquisa foi realizada no contexto da disciplina Programação I do curso de Bacharelado em Ciência da Computação da UFCG. É dado um grande enfoque à resolução de problemas e ao desenvolvimento de muitos programas. Essa abordagem é suportada por um ferramental técnico de apoio ao ensino desenvolvido pela própria equipe pedagógica. Em síntese, do ponto de vista do estudante, o ferramental permite que ele: tenha acesso ao enunciado dos exercícios propostos cuja resposta é um pequeno programa ou função e envie sua resposta ao professor. Estes sistemas ampliaram a possibilidade de interação com o aluno fora de sala de aula através da Web. Desta forma foi possível, atendendo às necessidades da disciplina, aumentar a quantidade de exercícios propostos sem prejuízo para o feedback dado ao aluno. O sistema viabiliza a correção mais rápida e padronizada dos exercícios.

Neste contexto, realizamos um estudo de caso utilizando dados do curso ministrado no período de 2011.2. Para a realização do estudo verificamos os programas dos alunos coletados após submissão ao sistema de correção e armazenados no banco de respostas. A submissão do programa indica, em sua maioria, que o aluno considera que aquela é uma implementação válida para especificação do programa dada no exercício. Assumimos que cada exercício proposto ao aluno é a especificação de um programa. Os exercícios que são propostos ao longo do curso são corrigidos automaticamente através do testador automático. Os exercícios propostos nas provas são testados automaticamente e também corrigidos por um professor, ou seja, há uma avaliador humano para checar além da corretude outros atributos como: desempenho, eficiência, estilo, complexidade, legibilidade, etc.

3.1. Medidas, variáveis e conjunto de dados

A questão de pesquisa que norteou o estudo foi: Como a qualidade – corretude e legibilidade – e a quantidade de código produzido pelo estudante ao longo do curso podem influenciar seu desempenho?

A *corretude* dos programas foi medida em função da nota emitida pelo testador automático, que é uma ponderação entre os casos de testes em que o programa é bem sucedido, a relevância e a quantidade de casos de testes totais formuladas pelo professor. A *quantidade* de código produzida pelo aluno no período de estudo foi medida em termos de questões para as quais foi submetida pelo menos uma solução e também em número de linhas de código destes programas (LoC, tradicional métrica da engenharia de software). O *desempenho* do estudante no curso foi medido em função de sua nota na segunda prova da disciplina. Optou-se por analisar as notas da segunda prova por ser uma avaliação central no curso, cobrindo aproximadamente 70% do conteúdo. A *legibilidade* dos programas foi medida considerando o padrão de codificação estabelecido pelo PEP08. Como já ressaltado, medimos o número de não-conformidades ao padrão encontradas no código, ou seja o número de *defeitos*. A *densidade* de defeitos é a medida que procura quantificar a habilidade que o aluno tem em escrever programas com menos defeitos, ou seja mais legíveis. Para tanto, medimos a quantidade de *defeitos* de um conjunto de programas e dividimos pelo seu número de *linhas de código* (LoC).

Os dados que utilizamos no estudo referem-se aos programas desenvolvidos pelos estudantes e as notas atribuídas pelo testador automático, com base nos testes propostos

pelos professores. Foram coletados 170 programas de 76 alunos. Consideramos os programas submetidos pelos estudantes no período compreendido entre duas provas da disciplina (intervalo de aproximadamente de 30 dias). Apenas a última versão submetida de cada questão enviada ao sistema por cada aluno, foi considerada para a análise. Os dados coletados neste período correspondem a 64 exercícios diferentes. No total, para esta fase, coletamos 3080 programas de 76 alunos.

4. Resultados

Observamos uma série de programas escritos pelo aluno e calculamos o índice *densidade defeitos*. Procuramos estabelecer a correlação entre este valor e a nota do aluno na segunda prova da disciplina, mostrado na Tabela 1. Além disso, aprofundamos a análise dos mesmos fatores através do agrupamento dos indivíduos de acordo com a nota. Usamos o método de Spearman para o cálculo da correlação, já que não verificamos normalidade no conjunto de dados.

	Questões	LoC	Defeitos	Densidade
Nota do aluno	0,550	0,530	0,365	0,056

Tabela 1 – Valores dos coeficientes de correlação

A Tabela 1 mostra os coeficientes de correlação entre a nota da prova e as variáveis: número de exercícios resolvidos, número de linhas de códigos produzidas - LoC, número de defeitos de legibilidade e densidade de defeitos de legibilidade. Há uma correlação moderada/forte (0,550) entre o número de questões resolvidas por cada aluno com sua nota na segunda prova. A mesma força, também é observada na correlação entre o número de linhas de código produzidas pelo aluno e a nota da prova (0,530). Chama à atenção o valor positivo encontrado para o coeficiente de correlação entre a nota na prova e a densidade de defeitos de legibilidade (0,056), embora em valor absoluto, seja inexpressivo. Este valor parece contrariar resultados obtidos em um estudo correlacional anterior quando avaliamos os programas individualmente, mostrando que para programas corretos, quanto menos legível o programa menor será sua nota por um avaliador humano. Tal diferença nos motivou a refinar o estudo, agrupando estudantes em função de seu desempenho geral na disciplina, cujo resultado pode ser visto na Figura 1.

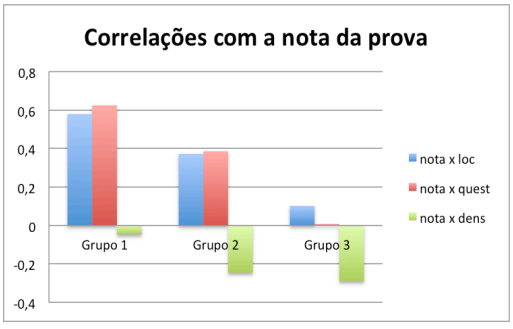

Figura 1 - Coeficientes de correlação entre a nota atribuída pelo professor e os fatores número de linhas de código produzidas, número de questões resolvidas e densidade de defeitos de legibilidade em cada grupo de estudantes. Grupo 1 = regular ou fraco. Grupo 2 = bom. Grupo 3 = excelente.

Os alunos foram agrupados em três grupos. No grupo 1 estão os estudantes com notas inferiores a 7.5, com desempenho considerado de regular a fraco. No grupo 2, estão os estudantes com notas entre 7.5 e 8.7, com desempenho considerado bom. E, finalmente, no grupo 3, os estudantes com notas acima de 8.7, com desempenho considerado excelente pelos professores. Novamente produzimos os coeficientes de correlação entre as notas na segunda prova e os fatores LoC, número de questões e densidade de defeitos de legibilidade.

O gráfico da Figura 1 mostra que para o Grupo 1, os alunos com desempenho regular ou fraco, o volume de programação realizado até a data da prova está mais correlacionado com nota do aluno do que a densidade de defeitos de legibilidade. Em contraste, para os alunos do Grupo 3, o valor absoluto da densidade de defeitos de legibilidade é bem superior aos outros fatores neste grupo. Isto indica que na análise das notas dos alunos do Grupo 3, as maiores notas são dos alunos com menor densidade de defeitos de legibilidade em seus códigos. Os resultados encontrados para o Grupo 2, mostram uma correlação negativa moderada entre densidade de defeitos e a nota dos alunos. Contudo, o valor absoluto da força desta correlação é menor que a dos outros dois fatores, consideradas correlações moderadas.

5. Análise e Discussão

A correlação moderada entre a quantidade de código produzida e a nota do aluno na prova corrobora com a noção intuitiva de que quanto mais o aluno pratica, produzindo código, melhor será a sua nota. Entretanto, a análise da qualidade do código, no quesito legibilidade deve ser ponderada mais cuidadosamente. A correlação observada entre o total de defeitos de legibilidade acumulados nos programas do aluno até a data da prova e nota nesta prova é de 0,365. O que parece indicar que, mesmo de forma moderada, quanto maior a quantidade de defeitos maior a nota do aluno. Temos que considerar, contudo, que a quantidade de defeitos é função das variáveis número de questões resolvidas e total de linhas de código produzidas. Uma forma melhor de interpretar essa correlação seria: quanto mais o aluno pratica e, portanto, quanto mais se dispõe a errar, mais chances tem de ter um bom desempenho.

A medida de densidade de defeitos procura eliminar a influência dos fatores ligados ao volume de prática de cada estudante, dividindo a quantidade de defeitos pelo número de linhas de código do aluno. Neste caso, contudo, a correlação observada foi praticamente inexistente (0,056). Agrupamos os estudantes de acordo com sua nota, a fim de obter resultados mais esclarecedores. Observa-se que a correlação entre a nota do aluno e a densidade de defeitos de legibilidade dos seus programas aumenta à medida que aumenta a nota do estudante (em valores absolutas, temos para o grupo 1: 0,05; para o grupo 2: 0,27; e para o grupo 3: 0,33). Esse fenômeno parece indicar que há uma lógica na forma de avaliação dos programas que é dominada pela componente corretude. Isto é, o professor só parece levar em conta a questão da legibilidade dos programas, depois que o programa é considerado minimamente correto, em termos do número de casos de teste a que satisfaz. Ainda assim, é necessário observar que ou a densidade de defeitos de legibilidade não é o único fator envolvido ou a métrica não captura adequadamente o conceito de legibilidade adotado pelo professor.

6. Conclusões e Trabalhos Futuros

Neste trabalho realizamos um estudo para investigar como a legibilidade dos códigos dos alunos relaciona-se com o desempenho dos estudantes no curso. Para isso, propusemos uma métrica simples e automática para o cálculo da legibilidade dos códigos produzidos por programadores iniciantes.

Como resultado deste trabalho, verificamos que o desempenho do estudante é fortemente correlacionado com a quantidade de código por ele produzida ao longo do curso. No geral, o estudo mostra que produzir programas legíveis é menos relevante para o desempenho do que produzir muitos programas. Na análise por grupos de acordo com o desempenho, no entanto, a legibilidade dos códigos é um fator relevante quanto melhor for o desempenho do aluno.

O estudo mostra que, além da corretude, a legibilidade é um fator analisado pelo professor na composição da nota. É importante questionar, como um caminho natural para uma futura pesquisa, quais são os outros fatores? Também interessaria repetir o estudo com outros conjuntos de dados, a fim de dar maior sustentação aos resultados.

Incluir legibilidade de código e outros aspectos de qualidade interna em uma suíte de testes automáticos enriquece o feedback dado aos estudantes. Isto pode ser útil não só para os cursos regulares/presenciais de programação que são fortemente baseados em resolução de problemas como também nos MOOCs – Massive Open Online Courses onde a escala é uma questão relevante.

Referências

Blinkstein, P.. 2011. “Using learning analytics to assess students' behavior in open-ended programming tasks”. In: Anais do 1st International Conference on Learning Analytics and Knowledge (LAK '11). ACM, New York, NY, USA, 110-116.

Buse, R. P. L. e Weimer, W. R.. 2010. “Learning a Metric for Code Readability”. In: IEEE Trans. Softw. Eng. 36, 4 (Julho 2010), 546-558.

Campos, C. P. e Ferreira, C. E. . “BOCA: um sistema de apoio a competições de programação (BOCA: A Support System for Programming Contests)”. In: Workshop de Educação em Computação (Brazilian Workshop on Education in Computing), 2004, Salvador. Anais do Congresso da SBC, 2004.

Cheang B., Kurnia A., Lim A., e Oon W.. 2003. “On automated grading of programming assignments in an academic institution”. In: Comput. Educ. 41, 2 (setembro 2003), 121-131. 30-7

Posnett, D., Hindle, A., e Devanbu, P. “A simpler model of software readability”. In: Anais do 8th Working Conference on Mining Software Repositories (MSR '11). ACM, New York 2011, NY, USA, 73-82.

Style Guide for Python Code. http://www.python.org/dev/peps/pep-0008/#introduction. [Online. Acesso 01-março-2013].