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

Feedback dos Alunos sobre a Qualidade de Código-Fonte para Apoiar o Aprendizado de Introdução à Programação

José Raul de Brito Andrade

Dissertação 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 obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

João Arthur Brunet Monteiro
(Orientador)

Campina Grande, Paraíba, Brasil

A553f Andrade, José Raul de Brito.
Feedback dos alunos sobre a qualidade de código-fonte para apoiar o aprendizado de introdução à programação / José Raul de Brito Andrade. – Campina Grande, 2018.
77 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2018.
"Orientação: Prof. Dr. João Arthur Brunet Monteiro".
Referências.

1. Ensino de programação. 2. Feedback. 3. Educação em computação. I. Monteiro, João Arthur Brunet. II. Título.

CDU 004.42(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA MARIA ANTONIA DE SOUSA CRB 15/398

FEEDBACK DOS ALUNOS SOBRE A QUALIDADE DE CÓDIGO-FONTE PARA APOIAR O APRENDIZADO DE INTRODUÇÃO À PROGRAMAÇÃO

JOSÉ RAUL DE BRITO ANDRADE

DISSERTAÇÃO APROVADA EM 26/10/2018

JOÃO ARTHUR BRUNET MONTEIRO, Dr., UFCG
Orientador(a)

DALTON DARIO SEREY GUERRERO, Dr., UFCG
Examinador(a)

JORGE CESAR ABRANTES DE FIGUEIREDO, Dr., UFCG
Examinador(a)

LILIANE DOS SANTOS MACHADO, Dra., UFPB
Examinador(a)

CAMPINA GRANDE - PB

Resumo

As disciplinas introdutórias de programação, tipicamente, envolvem uma grande quantidade de atividades, o que torna custoso fornecer manualmente *feedback* para cada aluno ao longo do semestre letivo. Desse modo, são propostas diversas abordagens com intuito de prover *feedback* automático sobre o código dos alunos. O foco principal desses trabalhos está no *feedback* funcional. Isto é, se o programa está correto de acordo com testes pré-definidos pelos professores. Contudo, há também a necessidade de analisar a qualidade do código dos alunos. Embora haja esforços de pesquisa nesse sentido, as abordagens propostas se concentram na análise automatizada de aspectos sintáticos, como os critérios propostos no PEP 8, mas que podem levar a um *feedback* genérico. Nesta dissertação, investigamos se, ao incluir os alunos como avaliadores, poderíamos fornecer *feedback* personalizado (detalhado) sobre a qualidade do código-fonte. Assim, nesta dissertação, a questão principal é se os alunos podem avaliar qualitativamente os programas de seus pares. Para isso, realizamos um estudo para verificar se os alunos elaboram dicas úteis e similares aos professores da disciplina. Descobrimos que a maioria dos estudantes consegue elaborar dicas úteis e identificar problemas de qualidade similares às dos professores da disciplina em um nível significativo, mesmo que não ideal. Além disso, são particularmente hábeis em elaborar dicas sobre problemas relacionados à complexidade dos programas. Replicamos esse estudo em outro cenário e obtivemos resultados semelhantes para corroborar o que observamos. Entendendo que os estudantes conseguem elaborar dicas, conduzimos uma prática de revisão de código por pares. Descobrimos que os alunos podem ser específicos e propor alternativas para os problemas identificados. Contudo, os mais avançados na disciplina elaboram dicas mais explicativas. Por fim, aplicamos métricas de qualidade de software nos códigos antes e depois dos alunos receberam as dicas e descobrimos que houve melhoria. Verificamos que o conhecimento do aluno influencia no seu resultado, mas observamos que a experiência em prover *feedback* e a motivação também são fatores de impacto no desempenho. Este estudo pode levar a investigações adicionais sobre como abordar a qualidade do código na aprendizagem colaborativa e em disciplinas mais avançadas de programação.

Abstract

Introductory programming subjects typically involve a large number of assignments, which makes it costly to provide feedback for each student throughout the term manually. In this way, several approaches are proposed to provide automatic feedback on the solutions submitted by the students. The focus of these researches is on functional feedback. That is if the program is correct according to predefined tests by teachers. However, there is also a need to analyze the code quality produced by the students. Although there are research efforts in this direction, the proposed approaches focus on the automated analysis of syntactic aspects, such as the criteria proposed in the PEP 8, but which can lead to general feedback. In this study, we investigated whether, by including students as reviewers, we could provide personalized feedback on the code quality. Thus, in this dissertation, the central question is whether students can qualitatively evaluate the program of your pairs. For this, we carried out a study to verify if the students' hints are useful and similar to those of the teachers of the subject. We found that most students can identify code quality issues related to the teachers at a significant level, even if not ideal. We have also seen that students can give meaningful hints at a considerable level, and are particularly able at finding and giving hints on issues related to programs complexity. We replicated this study in another context and obtained similar results to corroborate what we observed. After we found that students can give hints, we conducted a peer code review experiment. We found that students can be specific and propose alternatives to issues identified. However, most advanced students of the subject can give more explanatory hints. Finally, we applied quality metrics in the codes before and after the feedback and found that there was an improvement, but we did not get enough data to affirm that it was significant statistically. We verified that the student's knowledge influences in its result. However, we observed that experience in providing feedback and the motivation are also performance-impacting factors. This study may lead to further research on how to approach the code quality in collaborative learning and more advanced programming courses.

Agradecimentos

Agradeço à Deus, por todas as oportunidades concedidas em cada instante da minha vida. Por me permitir chegar até aqui e aprender tantas coisas, sempre rodeado de pessoas muito especiais.

À minha mãe, Aparecida, por seu amor, cuidado e conselhos. Todo este esforço é motivado pela sua simples existência, que mesmo distante é cada dia mais presente em mim.

Ao meu pai, Edirailton, pela confiança e investimento na minha educação. Sou imensamente grato pelo seu apoio sempre incondicional.

À minha vó Francisquinha (*in memoriam*), um poço de ternura, carinho e compreensão, pelo exemplo de força, determinação e fé.

Ao meu orientador, João Arthur, por sua paciência, conhecimentos compartilhados e orientação neste trabalho.

Aos professores de Programação I que gentilmente colaboraram com este estudo: Dalton Serey, Jorge Abrantes, Eliane Araújo e Wilkerson Andrade, da UFCG; e Eduardo Falcão e Vanessa Dantas, da UFPB. À professora Vanessa agradeço também por tudo que me ensinou ao longo de meu processo de formação desde a graduação.

À todos os colegas e amigos do SPLab, inclusive os que já saíram, por tornarem este laboratório um ótimo ambiente de trabalho. Agradeço à Lilian por todo suporte ao longo destes anos. Outras pessoas do SPLab também ajudaram, seja com um sorriso ou com uma conversa na copa desse laboratório que sempre lembrarei. Agradeço também às funcionárias da COPIN, Paloma e Liana.

Aos meus amigos da sala 104 do SPLab, Antônio, Jaziel, Rafael, Isabelle, Manoel, Marcos, Guilherme e Caio, grandes amigos que fiz em Campina Grande e que vão seguir comigo por onde eu for. Serei eternamente grato a vocês por tudo que fizeram por mim, com certeza sem vocês nada disso seria possível.

Aos demais colegas dá pós-graduação, pelas ideias, alegrias e tensões que compartilhamos. Vocês tornaram essa etapa da minha vida muito especial.

Aos meus grandes amigos, Lucas, Thayná, Marcus e João Gulherme, que mesmo na correria de nossas vidas, e nos desencontros que ela nos impõe, estão sempre comigo e me motivando a seguir pelos caminhos que escolhi.

À Josué, grande amigo que fiz na graduação e que segue comigo, por todas as revisões de texto e as mais variadas conversas.

À todos os professores com quem tive a oportunidade de trabalhar ao longo da minha vida acadêmica: Liliane Machado, Ana Liz, Flávia Costa, Ayla Rebouças, Vanessa Dantas e Thaise Costa. Vocês foram fundamentais para meu amadurecimento como pessoa e pesquisador.

À todos os estudantes que já tive a oportunidade de lecionar ou acompanhar, seja pela UFPB nas escolas Luiz Gonzaga Burity (Rio Tinto) e Senador Ruy Carneiro (Mamanguape) ou na UFCG, vocês me mostraram o quão nobre e desafiador é a docência.

À todos os familiares: Kaio César (irmão), Maria José (avó), Zé Nilton (avô), Clélia (tia) e Boni; e amigos: Julie, Fernando Mateus e Danilo Raniery, que mesmo à distância acreditaram e torceram por mim. Enfim, à todos que não citei nominalmente, mas que de alguma forma estiveram presentes nesta jornada.

À todos os pesquisadores que vieram antes de mim e de alguma forma colaboraram para tornar possível este estudo.

À CAPES, pelo auxílio financeiro através de bolsa de estudos.

Este trabalho foi parcialmente apoiado pelo contrato nº 08200.315131/2016-10 entre a UFCG e o ePol/DPF.

Conteúdo

Glossário

Para esclarecimento, definimos nesta seção os termos e siglas que usaremos no restante deste documento.

BCC: curso de Bacharelado em Ciência da Computação.

BSI: curso de Bacharelado em Sistemas de Informação.

***Feedback* efetivo:** dica capaz de produzir um efeito real na aprendizagem.

***Feedback* personalizado:** dica detalhada e específica, neste contexto, para particularidades do código que está sendo revisado.

***Feedback* qualitativo/sobre qualidade**: dicas de como melhorar a qualidade do código-fonte.

LCC: curso de Licenciatura em Ciência da Computação.

UFCG: Universidade Federal de Campina Grande.

Unidades de estudo: partes estruturadas de conteúdos que formam o currículo da disciplina. Em Programação I de BCC/UFCG, cada estrutura da linguagem de programação (como if, while, def, entre outros.) é abordada em uma unidade diferente. Isso ocorre devido a metodologia baseada no *Mastery Learning* [Blo68], na qual cada unidade é composta por poucos conceitos e com objetivos de aprendizagem bem definidos.

UFPB: Universidade Federal da Paraíba/*Campus IV*.

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Capítulo 1

Introdução

As disciplinas introdutórias de programação geralmente envolvem uma grande quantidade de atividades (tarefas), o que dificulta o acompanhamento individual do desempenho dos alunos. No curso de Ciência da Computação (BCC) da Universidade Federal de Campina Grande (UFCG), por exemplo, essa disciplina tem por volta de 100 estudantes matriculados por semestre e, em alguns momentos do período letivo, eles submetem dezenas de soluções por dia. Assim sendo, é custoso fornecer manualmente *feedback* para cada aluno ao longo do semestre letivo.

Com o intuito de minimizar esse problema, são realizados estudos acerca de abordagens para prover *feedback* automático sobre o código-fonte dos estudantes [GPL16] [SGSL13]. Em geral, as propostas implementam a ideia de Juízes Online [WAB+17], na qual corrigem automaticamente a solução submetida com testes pré-definidos pelos professores. O foco principal desses trabalhos está no *feedback* funcional, isto é, se o programa está correto de acordo com os testes (se ele faz o que deveria fazer). Contudo, mesmo que nos cursos introdutórios a corretude seja o aspecto mais significativo, também é importante analisar a qualidade do código produzido pelos alunos.

1.1 Motivação e Contextualização

Nos cursos de Ciência da Computação, as disciplinas relacionadas à programação de computadores têm papel fundamental na formação dos estudantes. No entanto, é evidente a dificuldade na aprendizagem dessas disciplinas [dJGM]. De acordo com Wilcox [Wil15],

dentre os desafios que influenciam o aprendizado está a dificuldade do professor em acompanhar individualmente o desempenho dos estudantes. Isso ocorre porque, principalmente nas disciplinas introdutórias, possuem um elevado número de estudantes matriculados. Assim sendo, é custoso para o professor fornecer efetivamente algum tipo de *feedback* manual para todos nas atividades laboratoriais.

Com intuito de abordar esse problema, são encontradas na literatura diversas propostas de ferramentas para submissão de atividades de programação que fornecem *feedback* automático. Segundo Schunk e Petermann [SP89], o *feedback* possibilita refletir sobre as diferenças entre o resultado alcançado e o que é esperado, podendo assim ser considerado um importante aliado no processo de construção do conhecimento. As ferramentas que fornecem *feedback*, em sua maioria, implementam a ideia de Juízes online [WAB+17], ou seja, os códigos submetidos pelos alunos são corrigidos automaticamente a partir de testes pré-definidos pelo professor [GPL16] [SGSL13]. O foco principal do campo de *feedback* automatizado e avaliação de atividades está na correção funcional dos programas, embora algumas ferramentas também incorporem *feedback* sobre aspectos de qualidade [KHJ17].

As propostas para *feedback* sobre a qualidade do código-fonte geralmente são de ferramentas, como o QCheck [ASF16], que automatizam a análises qualitativas baseadas nos padrões de codificação definidos pela comunidade Python no PEP 8 [VRWC01]. Além disso, algumas IDE's profissionais também detectam problemas de qualidade do código e outras refatorações, porém estas são muitas vezes complexas para iniciantes e não destinadas a apoiar a aprendizagem [KHJ17]. Esse cenário traz como consequência a busca dos alunos apenas pela corretude de seus códigos, desmotivando-os a otimizar a qualidade de suas soluções [KHJ17].

De acordo com Borges [BMSB14], para prover um *feedback* efetivo (que de fato auxilia o estudante) é importante que ele seja específico e personalizado (detalhado). Nesse sentido, Glassman *et al.* [GLCM16] propuseram uma abordagem colaborativa, baseada no conceito de *crowdsourcing*, na qual os próprios alunos elaboram dicas personalizadas para a resolução e otimização de soluções de atividades na disciplina de Arquitetura de Computadores. O objetivo do estudo foi viabilizar uma tutoria personalizada para turmas com elevado número de alunos. A ideia parte do princípio de que os próprios alunos possuem a capacidade de participar do processo de tutoria de seus colegas com base na sua experiência adquirida ao

longo da resolução de problemas anteriores.

O *Crowdsourcing* é uma abordagem de obtenção de conteúdo através da colaboração de um grupo de pessoas. Sua utilização no contexto educacional consiste na colaboração coletiva dos alunos para a elaboração de um determinado conteúdo disponibilizado para outros alunos, enquanto se envolvem em uma experiência de aprendizagem significativa [Hen09]. Essa abordagem tem apresentado resultados promissores em turmas de diferentes áreas do conhecimento e com elevado número de estudantes, tanto no acompanhamento da aprendizagem (individual ou em grupo) [AJHA+15], quanto em aspectos de qualidade da avaliação *peer-to-peer* [Ave14].

A avaliação realizada pelos alunos (auto avaliação ou em pares) é um processo que tem a interação como um mecanismo fundamental para construção do conhecimento. Nesse processo, os estudantes compartilham informações, tomam decisões, argumentam, entre outras ações. Essa prática tem se mostrado eficiente, principalmente em cursos online com muitos estudantes [KWL+15]. De acordo com Shang *et al.* [SSC01], relacionar a experiência com a interação pode auxiliar os estudantes na atribuição de significados, devido à reflexão de pontos de vista distintos. Dessa forma, incluir o aluno no processo de avaliação não somente pode minimizar o tempo de espera para obter *feedback*, mas também melhorar a formação, pois ele pode tornar-se mais crítico e autônomo na sua aprendizagem.

1.2 Problema de Pesquisa

Os estudos acerca de *feedback* e avaliação de atividades de programação abordam principalmente problemas funcionais dos programas na correção. Contudo também há ferramentas que incorporam *feedback* sobre problemas de qualidade [KHJ17]. A qualidade do código está relacionada com aspectos como complexidade da solução, tamanho do código-fonte, nomenclatura das variáveis, entre outros [KHJ17]. Embora haja esforços de pesquisa a este respeito, algumas ferramentas propostas, como o Qcheck [ASF16], automatizam análises qualitativas baseadas em aspectos sintáticos, como os critérios definidos pela comunidade Python no PEP 8 [VRWC01]. No entanto, o *feedback* fornecido algumas vezes é genérico e, para ser efetivo, ainda pode precisar do trabalho manual dos professores. Nesse contexto, uma alternativa para prover *feedback* qualitativo em menos tempo e personalizado é incluir

os estudantes como avaliadores da qualidade do código-fonte.

1.3 Objetivos da Pesquisa

Neste estudo, buscamos verificar se, ao incluirmos os alunos como avaliadores, poderíamos fornecer *feedback* personalizado para outros estudantes sobre a qualidade do código-fonte.

1.3.1 Objetivo Geral

O objetivo desta dissertação é:

- Alavancar a aprendizagem de programação introdutória acerca de aspectos qualitativos do código-fonte.

1.3.2 Objetivos Específicos

Em particular temos os seguintes objetivos específicos:

- Investigar se os alunos de programação introdutória podem elaborar *feedback* sobre a qualidade do código-fonte;
- Verificar quais são as características do *feedback* dos estudantes de programação introdutória.

1.4 *Design* do Estudo

O estudo que conduzimos nesta dissertação é caracterizado como quali-quantitativo, visando identificar e discutir as características e oportunidades a partir do *feedback* colaborativo, elaborado pelos estudantes, acerca da qualidade do código-fonte em atividades da disciplina introdutória de programação.

Dividimos este estudo em duas partes. Na primeira parte, realizamos um experimento para verificar se os estudantes conseguem prover *feedback* sobre a qualidade de código, para isso verificamos: 1) se há similaridade entre os problemas de qualidade de código-fonte identificados por eles, em relação aos problemas de qualidade de código-fonte identificados

pelos professores da disciplina; 2) quais os problemas de qualidade de código-fonte são reportados com mais frequência pelos estudantes; e 3) se as dicas dos alunos podem ser consideradas úteis. Participaram desse experimento alunos matriculados em disciplinas de Programação I dos cursos de Licenciatura em Ciência da Computação (LCC) e Sistemas de Informação (BSI) da Universidade Federal da Paraíba/*Campus IV* (UFPB) e Ciência da Computação (BCC) da Universidade Federal de Campina Grande (UFCG).

Após descobrirmos que os alunos podem elaborar dicas sobre a qualidade do código, realizamos a segunda parte do estudo que consistiu na inclusão de uma atividade prática na disciplina de Programação I do BCC/UFCG, baseada no conceito de revisão por pares. Nessa atividade, os estudantes forneceram, colaborativamente, dicas para melhorar a qualidade dos programas uns dos outros. Em seguida, cada aluno pode reescrever seus programas considerando as dicas que recebeu. Por fim, analisamos as características das dicas e o histórico de submissões dos estudantes. Também aplicamos métricas de qualidade de software (complexidade ciclomática (cc), linhas lógicas de código (lloc) e PEP 8) para verificar se houve mudanças significativas entre as versões do código antes e depois de receberem as dicas.

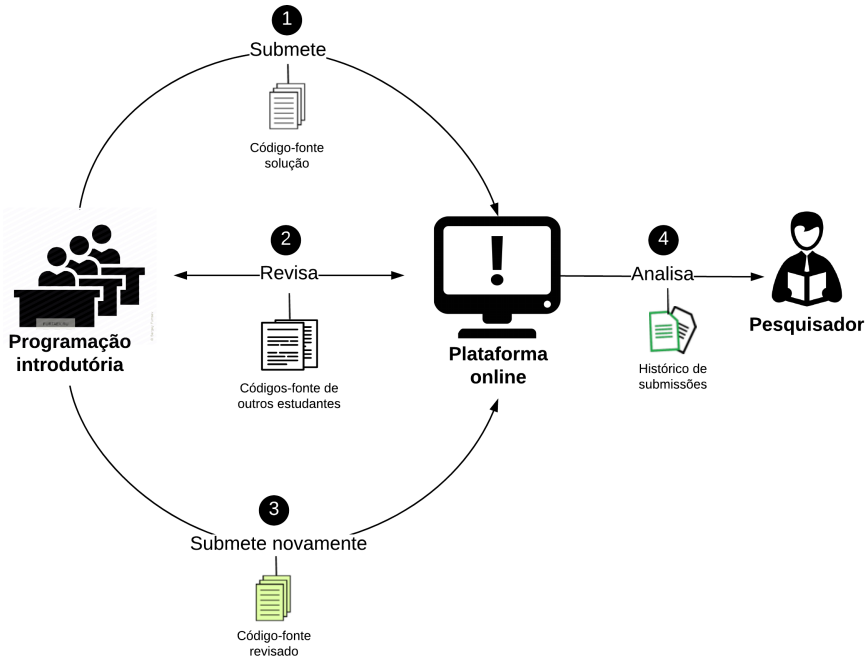

Figura 1.1: Visão geral da estratégia pedagógica proposta.

A Figura 1.1 ilustra a visão geral da estratégia de revisão colaborativa que propomos neste estudo. Onde, (1) os estudantes submetem o código-fonte da atividade; (2) os códigos

são distribuídos para que outros estudantes revisem, deixando dicas de como melhorá-los. Essas dicas são encaminhadas para o autor de cada código; e (3) o autor do código pode realizar uma nova submissão considerando as correções sugeridas por seus colegas. Por fim, (4) analisamos o histórico de submissão de cada estudante, as características das dicas e utilizamos métricas de qualidade de software para verificar se houve melhora nos códigos submetidos.

As perguntas que respondemos nesta dissertação são: **Q1**: O quão similar aos professores da disciplina os alunos de Programação I conseguem identificar problemas na qualidade do código de outros estudantes?; **Q2**: Quais problemas de qualidade de código são identificados mais frequentemente pelos alunos de Programação I?; **Q3**: O quão útil são as dicas dos alunos de Programação I sobre problemas na qualidade do código?; e **Q4**: Quais as características das dicas elaboradas pelos estudantes de programação introdutória?. Os participantes deste estudo preencheram formulários de consentimento aprovados pelo escritório de conformidade em nossa instituição.

1.5 Resultados

Ao todo participaram deste estudo 225 estudantes de Programação I. Na primeira parte (Capítulo 4) participaram 75 alunos do curso de Ciência da Computação (BCC) da UFCG e 51 alunos dos cursos de Sistemas de Informação (BSI) e Licenciatura em Ciência da Computação (LCC) da UFPB do semestre letivo 2017.2. Já na segunda parte (Capítulo 5), participaram 99 alunos do curso de Ciência da Computação da UFCG do semestre letivo 2018.1.

Na primeira parte deste estudo, ao verificarmos a similaridade entre as dicas elaboradas pelos alunos com as dicas dos professores, descobrimos que os estudantes conseguem identificar problemas de qualidade de código-fonte em um nível significativo de similaridade com os professores da disciplina. No caso do BCC/UFCG, 60%.dos estudantes e 62% dos estudantes do BSI e LCC da UFPB. Observamos que um dos fatores que afetaram na similaridade foi a quantidade de dicas que o aluno elaborou.

Descobrimos que os alunos são particularmente hábeis em encontrar e elaborar *feedback* sobre problemas na complexidade dos programas. Além disso, constatamos que 68% dos alunos do BCC/UFCG tiveram suas dicas avaliadas como úteis pelos professores da disci-

plina e que os alunos do BSI e LCC também alcançaram esse resultado pela mesma avaliação. Observamos que a pouca experiência em prover *feedback* e até o conhecimento dos estudantes sobre programação influenciou na qualidade do *feedback*.

Na segunda parte deste estudo, entendendo que os alunos conseguem elaborar *feedback* qualitativo, conduzimos uma prática de revisão de código por pares. Verificamos que alunos do BCC/UFCG conseguem ser específicos e propor alternativas para os problemas identificados em um nível significante. No entanto, os alunos mais avançados na disciplina conseguem ter desempenho melhor nesse aspecto. Verificamos também que o conhecimento do aluno em programação influencia na corretude das dicas que elabora. Constatamos que 77% dos alunos deram dicas corretas e que 80% dos alunos deram dicas claras.

Por fim, aplicamos métricas de qualidade de software (complexidade ciclomática, linhas lógicas de código e PEP 8) nos códigos dos alunos antes e depois de receberem dicas e verificamos que o *feedback* ajudou os alunos a melhorarem a qualidade de seus códigos. Contudo, observamos que a experiência em revisar código e a motivação em colaborar tem grande impacto na qualidade de seu *feedback*.

1.6 Contribuições da Pesquisa

Com o objetivo de prover *feedback* para atividades de programação, são propostas ferramentas que realizam essa atividade de forma automática, como apresentaremos no Capítulo 3. O foco principal desses trabalhos está no *feedback* funcional. No entanto, é importante que aspectos relacionados à qualidade de código também sejam abordados. Além disso, a presença do professor ainda é fundamental na maioria dos casos, seja para elaborar casos de teste ou fornecer dados para o treinamento das ferramentas.

Desse modo, esta pesquisa contribui para a área detalhando as características das dicas dos estudantes sobres aspectos qualitativos do código. Realizamos isso por meio de uma abordagem de aprendizagem ativa, baseada no conceito de revisão por pares, que inclui o aluno no processo de avaliação para possibilitar obter *feedback* detalhado e personalizado.

Com este trabalho: 1) verificamos que os estudantes podem ser incluídos como avaliadores da qualidade de código de seus colegas e 2) quais as características do *feedback* que os estudantes elaboram. Além disso, esse estudo fez as seguintes contribuições:

- Verificamos que um grupo de estudantes consegue fornecer *feedback* útil sobre a qualidade de código de seus pares;
- Verificamos quais aspectos de qualidade de código mais são identificados pelos estudantes;
- Identificamos as características das dicas elaboradas pelos alunos de programação introdutória.

Verificamos que os alunos de programação introdutória podem revisar a qualidade dos programas de outros estudantes. Este estudo pode levar a investigações adicionais sobre como abordar a qualidade do código-fonte na aprendizagem colaborativa e em turmas mais avançadas de programação.

1.7 Estrutura do Documento

Este trabalho está organizado da seguinte forma:

- **Capítulo 2 - Referencial Teórico:** Apresenta o arcabouço teórico necessário para o acompanhamento desta dissertação.
- **Capítulo 3 - Trabalhos Relacionados:** Apresenta e discute-se os trabalhos relacionados a esta dissertação.
- **Capítulo 4 - Os Alunos Conseguem Elaborar *Feedback* sobre a Qualidade do Código?:** Este capítulo apresenta o primeiro experimento que realizamos para verificar se os estudantes de turmas de programação introdutória conseguem elaborar dicas sobre a qualidade do código-fonte. Primeiro testamos com os alunos de Programação I do BCC/UFCG, mas, com intuito de verificar os resultados em contextos diferentes, replicamos na disciplina de Programação I do LCC e BSI, ambos da UFPB.
- **Capítulo 5 - Quais as Características do *Feedback* Elaborado pelos Alunos?:** Detalha o segundo estudo desta dissertação, no qual realizamos uma atividade prática de revisão de código por pares com estudantes de Programação I do BCC/UFCG. Neste capítulo, descrevemos o *design* do experimento, assim como os resultados obtidos e análise.

- **Capítulo 6 - Considerações Finais:** Resume este trabalho enfatizando o que foi feito, discute os resultados e os desafios enfrentados. Além disso, abordará algumas oportunidades e trabalhos futuros que podem ser feitos para aprofundar e melhorar esta pesquisa.

Capítulo 2

Fundamentação Teórica

O ensino de programação para alunos iniciantes é apresentado na literatura como um desafio mundial da área de Educação em Ciência da Computação [MM13] [LAF+04]. Dentre as características das disciplinas introdutórias de programação, destaca-se o elevado número de alunos matriculados por semestre letivo [Wil15]. Normalmente, essas disciplinas envolvem uma grande quantidade de atividades (tarefas), o que torna oneroso para os professores proverem *feedback* manual ao longo do período. Contudo, estudos descrevem que a impossibilidade de um acompanhamento adequado por parte do professor para cada um dos alunos é um dos fatores que motivam desistências e reprovações [Yad11] [BCCC15].

Diversos estudos têm investigado as dificuldades e características dos alunos, assim como debatido acerca de alternativas para facilitar o *feedback* em programação. Diferentes abordagens têm sido propostas para prover *feedback* efetivo para os alunos, cada uma com suas vantagens e desvantagens. Nesta dissertação, argumentamos que o *feedback* formativo, elaborado por outros estudantes, tem o potencial de auxiliar o aprendizado de programação, assim como ajudá-los a melhorar a qualidade de seu código enquanto refletem sobre diferentes formas de solucionar o mesmo problema.

De acordo com Shute [Shu08], o *feedback* formativo é a informação provida ao aluno que tem o intuito de modificar seu pensamento ou comportamento com a finalidade de melhorar a aprendizagem. Ainda de acordo com a autora, o *feedback* formativo deve ser não avaliativo, específico e oportuno. O trabalho de Shute [Shu08] apresenta uma revisão da literatura na qual são apresentadas diretrizes sobre o que deve ser considerado, ou não, ao prover *feedback* formativo. Além disso, são discutidos os tipos de *feedback* (verificação da precisão

da resposta, explicação da resposta correta, dicas, exemplos trabalhados), o momento em que o *feedback* formativo pode ocorrer (imediatamente após uma resposta ou após algum tempo) e as características dos alunos que o recebem.

Hattie e Timperley [HT07] publicaram uma revisão da literatura que analisa o impacto do *feedback* sobre a aprendizagem e a realização. Os resultados obtidos apontaram que, embora o *feedback* esteja entre as principais influências, o tipo de *feedback* e o modo como ele é dado podem ser diferencialmente efetivos. Os autores propõem um modelo de *feedback* no qual, para ser considerado efetivo, é importante que três perguntas (feitas pelos alunos ao professor) sejam respondidas, são elas: "*para onde eu vou? (quais são os objetivos?), como eu estou indo? (que progresso está sendo feito em direção ao objetivo?) e para onde ir? (quais atividades precisam ser realizadas para obter melhores progressos?)*" [HT07] (tradução nossa) [2].

Portanto, ao elaborar e fornecer *feedback* é primordial considerar aspectos, como: tipo de *feedback*, características do estudante e momento em que ele ocorrerá. A capacidade de determinada abordagem em possibilitar a geração de *feedback* adequado e oportuno está fortemente relacionada ao fato de considerar as características dos alunos. Segundo Schunk [SP89], o *feedback* possibilita refletir sobre as diferenças entre o resultado alcançado e o que é esperado, podendo assim ser considerado um importante aliado no processo de construção do conhecimento. Uma abordagem comumente utilizada na Engenharia de Software para prover *feedback* sobre o código-fonte é a revisão por pares (do inglês, *peer review* ou *peer assessment*).

A revisão de código por pares é uma abordagem utilizada por desenvolvedores e, quando realizada de forma adequada, pode melhorar significativamente a qualidade do código do projeto de software, assim como aperfeiçoar continuamente as habilidades técnicas dos profissionais [KS99]. O objetivo de adotar a revisão por pares é melhorar qualitativamente um trabalho a partir da colaboração. Assim, pesquisadores têm defendido que adotá-la (s) no contexto educacional pode alavancar o aprendizado dos estudantes que recebem revisão e também dos revisores [GLCM16] [WLF+12].

Alguns estudos têm adotado a colaboração entre estudantes para prover *feedback*

[2] "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?)"

[AJHA+15] [Ave14]. No entanto, no contexto dos cursos de Ciência da Computação, é comum ser adotada em outras disciplinas que não as de programação introdutória [GLCM16] ou com foco principalmente na funcionalidade do código, deixando de abordar aspectos qualitativos das soluções.

Neste capítulo, apresentamos uma visão geral da literatura da Educação e Ciência da Computação, abordando os assuntos desta dissertação e objetivando tornar este documento auto-suficiente. Primeiramente, apresentamos aspectos da avaliação da aprendizagem e dos tipos de avaliação, considerando que prover *feedback* é uma forma de avaliação formativa. Em seguida, discutimos o potencial do *feedback* para o aprendizado de programação. Por fim, abordamos as características da aprendizagem colaborativa.

2.1 Avaliação da Aprendizagem

O processo de avaliação é composto por um conjunto característico de atividades, como: analisar, julgar, bonificar, penalizar, entre outros [CG08]. Devido a subjetividade dessas atividades, o ato de avaliar pode tornar-se complexo e até arbitrário, de acordo com os critérios definidos pelo avaliador. Nesse contexto, esse tema vem ganhando destaque em discussões pedagógicas, de diferentes contextos educacionais, sobre técnicas mais efetivas para avaliar e padronizar os diferentes tipos de avaliação.

Na maior parte das instituições de ensino, independente no nível escolar, o processo de avaliação ainda é realizado com ênfase em aspectos objetivos [BMSB14]. Esse cenário reflete características de uma abordagem tradicional, onde o ensino e a aprendizagem acontecem tão somente pela transmissão de conhecimentos [Miz86]. Tal prática tem como consequência o incentivo, muitas vezes involuntário, para que os alunos apenas reproduzam os conteúdos comunicados em sala de aula, não instruindo-os a desenvolver criticidade e autonomia no seu processo de aprendizagem [CG08].

É de fundamental importância utilizar abordagens de avaliação mais voltadas ao diagnóstico [LUC98]. Essa ideia é ressaltada posteriormente por outros autores que afirmam que pouco é feito no sentido de reorientar o estudante com base nos resultados obtidos na avaliação [BMSB14] [CG08]. Essa prática, faz com que a avaliação se reduza ao um meio para classificar, verificar e selecionar os estudantes.

O papel social das instituições de ensino é promover aos estudantes formação baseada no desenvolvimento de suas capacidades e competências, e assim entregar para a sociedade profissionais capacitados [BMSB14]. A competência desses profissionais está diretamente ligada a qualidade da avaliação que receberam enquanto estudantes. Desse modo, se faz necessário repensar tanto a forma tradicional (somativa) que a avaliação acontece, quanto a inclusão de aspectos que tornem essa tarefa diagnóstica (formativa) [B$^+$71a]. As duas formas de avaliar são complementares e objetivam propiciar um retorno mais efetivo sobre o desempenho dos estudantes.

2.1.1 Avaliação Somativa e Avaliação Formativa

Nas disciplinas introdutórias de programação, usualmente, a avaliação é realizada de modo formal e pontualmente ao concluir determinada unidade de estudo, ou um conjunto delas, essa avaliação é nominada de somativa [B$^+$71a]. A avaliação somativa normalmente é estática, sendo definido no início do curso os momentos nos quais será realizada. O intuito de adotar esse tipo de avaliação é verificar se os estudantes assimilaram os conteúdos abordados até certo ponto da disciplina. Ela é importante para a tomada de decisões sobre a progressão do estudante [Cen94].

Na avaliação somativa, o estudante tem seu desempenho medido por notas, ou seja, ele deverá somar determinada pontuação para ter sucesso na disciplina [Tar05]. Consequentemente, ela reforça a comparação entre os estudantes e o desempenho está relacionado estreitamente às notas obtidas. Além disso, a avaliação somativa pode ignorar particularidades do processo de aprendizagem dos indivíduos, pois nela assume-se que todos aprendem da mesma forma [BMSB14].

Embora os estudantes vejam os mesmos conteúdos, a forma como a aprendizagem acontece varia para cada um. Fatores externos, como a experiência, influenciam no tempo de assimilação [CG08]. Assim, a avaliação deve ser uma atividade constante para identificar problemas na aprendizagem e tratá-los o quanto antes. Porém, o modelo de avaliação somativa não dá total suporte a essa prática. Pelo contrário, o foco que é dado ao resultado final ao invés do caminho para o aprendizado pode inclusive desmotivar os estudantes com maiores dificuldades [BMSB14].

Em contrapartida, a avaliação formativa é um método de análise e ação constante e cí-

clico, não estático. Nesse caso, a prática de avaliar é parte inerente do processo educativo, ou seja, ela é realizada de forma contínua com a finalidade de obter dados para o replanejamento do processo de ensino e aprendizagem [HFCM94]. O fato dessa avaliação ser contínua favorece a identificação precoce de problemas de aprendizagem, e não apenas em testes pontuais realizados no fim do curso, como na avaliação somativa. A avaliação formativa estimula a autonomia dos estudantes na sua aprendizagem, com intuito de desenvolver, de maneira mais consistente, os conhecimentos e habilidades deles. Ela também favorece a individualização da aprendizagem e a interação informal entre os indivíduos [B$^+$71a].

Assim sendo, a avaliação formativa, quando realizada de maneira adequada, é uma importante estratégia pedagógica que incorpora-se como parte essencial no processo de ensino e aprendizagem. No entanto, utilizar métodos adequados para essa avaliação não é tarefa trivial [Tes13]. Dentre os aspectos que dificultam as mudanças no processo de avaliação, destacam-se a estrutura curricular na qual os cursos estão inseridos, falta de discussões sobre adaptações nas abordagens de ensino e avaliação da aprendizagem e a carga de trabalho dos professores [BMSB14]. Assim, defendemos neste estudo que incluindo os alunos como avaliadores é possível minimizar essas dificuldades e prover *feedback* efetivo e personalizado sobre a qualidade do código-fonte.

Vale destacar que a avaliação somativa, quando aplicada de forma adequada, é um importante instrumento para mensurar a aprendizagem, além de ser um dos métodos de avaliação mais utilizados. A avaliação formativa e a somativa devem ser complementares [Tar05]. A utilização das duas é considerada uma boa prática no sentido de avaliação, por apresentar potencial para medir de forma mais eficiente a aquisição de conhecimentos e aptidões dos estudantes [B$^+$71a].

2.2 Efeitos do *Feedback* na Aprendizagem

O *feedback* educacional é conceituado na literatura de muitas formas. Neste estudo, utilizamos a definição de Shute [Shu08] que concebe *feedback* como: "*informação comunicada ao aluno que se destina a modificar seu pensamento ou comportamento para melhorar o apren-*

dizado" [Shu08] (tradução nossa) [1]. De acordo com Yair [Yai14], o *feedback* formativo provido pelos professores sobre o trabalho dos alunos é um fator fundamental para interação entre eles. Para possibilitar o aprendizado de determinado conceito, seja qual for, é fundamental, sempre que possível, prover *feedback* sobre o desempenho do estudante [SBS14]. Borges [BMSB14] cita algumas características desejáveis para que o *feedback* seja efetivo. Detalhamos na Tabela 2.1, de acordo com o contexto deste trabalho, algumas dessas características.

Tabela 2.1: Características de um *feedback* efetivo.

Oportuno É importante que o *feedback* seja mais próximo da conclusão da atividade. Quando o *feedback* é elaborado muito tempo depois, alguns detalhes importantes que foram observados podem ser perdidos.
Específico O *feedback* deve ser descritivo e personalizado. Deve ser evitado comentários genéricos, como "Sua solução está mais complexa do que deveria". Ao invés disso, os comentários devem ser mais específicos, como: "A solução está mais complexa do que é preciso. Não é preciso os dois primeiros comandos 'for' dentro da função, basta percorrer a lista salvando os menores e maiores. É preciso apenas um laço para isso".
Imparcial O aluno deve prover *feedback* baseado apenas no que está sendo observado no momento. Isto é, o foco da avaliação deve ser na qualidade do código desenvolvido por outro estudante, desconsiderando a personalidade ou opiniões pessoais oriundas de experiências anteriores com o avaliado.

O *feedback* efetivo fornece ao aprendiz dois tipos de informação: verificação e elaboração [KS89]. De acordo com Kulhavy e Stock [KS89], a verificação é o julgamento da corretude, ou não, da resposta. Já a elaboração está ligada a informação, como sugestões para guiar o aluno em direção a uma resposta correta. Assim, o *feedback* efetivo deve incluir elementos tanto de verificação quanto de elaboração [MB01] [BDKKM91].

[1] "Information communicated to the learner that is intended to modify his or her thinking or behavior to improve learning."

A estratégia mais frequente para realizar a verificação em atividades de programação é a avaliação baseada em testes. Inúmeras ferramentas são desenvolvidas seguindo o modelo de Juízes Online [WAB+17]. Isto é, os estudantes submetem seus códigos e, a partir de um conjunto de casos de testes elaborados pelos professores, recebem *feedback* quanto a corretude funcional. Contudo, o *feedback* vai além da indicação "certo-errado"sobre a tarefa do aluno. Inclusive, para alunos iniciantes pode não ser o suficiente para, de fato, melhorar seu entendimento.

Uma perspectiva importante para verificar a eficácia do *feedback* está relacionada às características do estudante. Diversos estudos vêm propondo a personalização de uma mensagem de *feedback* considerando as características de aprendizagem dos alunos. Estudos também têm discutido sobre o conteúdo do *feedback* para tarefas interativas de aprendizado [Nar08]. Isto é, qual e quanto de informação a mensagem de *feedback* deve apresentar. Em seu trabalho, DeNero *et al.* [DSPQ+17] propuseram melhorar as mensagens erro do compilador para que o usuário identificasse a linha de erro e conseguisse corrigir seu código mais facilmente. Eles ilustraram o erro e mostraram como corrigi-lo. No entanto, os resultados mostraram que não houve efeito sobre a capacidade dos alunos de corrigir seus erros de código. Explicações longas podem ser inúteis, não pela corretude, mas a complexidade pode fazer com que os alunos não leiam [Shu08]. Portanto, para o *feedback* ser efetivo é fundamental haver equilíbrio entre as informações que podem e as que devem existir em uma mensagem de *feedback*.

Outra característica que deve ser considerada para prover *feedback* efetivo é o tempo (atrasado ou imediato). As ferramentas de avaliação automática conseguem prover um retorno imediato, o que é vantajoso já que o aluno não precisa atrasar seu cronograma de estudos esperando resposta dos professores ou monitores da disciplina. Entretanto, esses efeitos têm sido revistos, pois essa agilidade para receber o *feedback* também pode estimular o estudante a não testar completamente seu código antes de submetê-lo e não pensar criticamente para programar seu código [BE14].

Por fim, defendemos que o *feedback* formativo pode auxiliar no processo de aprendizagem do estudante. Contudo, é necessário investigações mais aprofundadas para afirmar qual a forma mais eficaz de realizá-lo. Até então, sabemos que o *feedback* deve ser personalizado, objetivo e fornecido no momento oportuno. Acreditamos que a inclusão do aluno

como provedor do *feedback* formativo tem potencial para fornecer *feedback* útil e aprimorar o aprendizado de programação. O *feedback* elaborado pelo aluno tende a ser personalizado, pois ele estará ajudando um outro estudante de mesmo nível acadêmico e, como o *feedback* é baseado na sua experiência em resolver determinado problema, sua dica tende a ser mais clara e objetiva, já que ele acabou de desenvolver a solução. Além disso, o aluno que elabora o *feedback* tem acesso a outros códigos, o que pode ajudá-lo a refletir sobre diferentes soluções para o mesmo problema e revisar o que ele aprendeu.

2.3 Aprendizagem Colaborativa

A aprendizagem colaborativa é definida como uma estratégia de aprendizagem onde os membros do grupo apoiam uns aos outros com intuito de atingir determinado objetivo educacional [LPTA05]. Complementando essa ideia, Campos *et al.* [CSBS03] caracterizam a aprendizagem colaborativa como:

> "[...] uma proposta pedagógica na qual estudantes ajudam-se no processo de aprendizagem, atuando como parceiros entre si e com o professor, com o objetivo de adquirir conhecimento sobre um dado objeto." [CSBS03]

Embora haja diferentes maneiras de conceituar a aprendizagem colaborativa, os autores convergem que é uma abordagem para alcançar determinado objetivo educacional, a partir da construção do conhecimento em grupo e com apoio mútuo entre os membros. O princípio fundamental da aprendizagem colaborativa está na interação e ajuda entre os alunos.

Segundo Dillenbourg [Dil99], uma teoria da aprendizagem colaborativa diz respeito à 4 itens: "*critérios para definir a situação (simetria, grau de divisão do trabalho), as interações (por exemplo, simetria, negociabilidade, ...), processos (fundamentos, modelagem mútua) e efeitos.*" [Dil99]. Ainda de acordo com Dillenbourg, a chave para entender a aprendizagem colaborativa está nas relações bidimensionais entre esses quatro itens.

A ligação entre os **processos** e os **efeitos** da aprendizagem colaborativa se dá inicialmente devido os processos gerarem efeitos. Contudo, alguns processos são descritos pelos efeitos, como a internalização, e alguns efeitos são apresentados em processos, como a capacidade e trabalhar em grupo. Já entre entre a **situação** e as **interações**, a situação define as condições

para as interações. Por outro lado, a situação pode ser definida como "colaborativa" devido ocorrer interações colaborativas entre o grupo. Por fim, é necessário definir os **processos** para definir a característica das **interações** [Dil99].

Dillenbourg [Dil99] destaca ainda alguns mecanismos da aprendizagem cognitiva, centrais da aprendizagem, mas estendidos para aprendizagem colaborativa, como apresentamos a seguir:

- Indução: Segundo Schwartz [Sch95], os pares elaboram representações mais abstratas do problema em questão, pois, trabalhando em equipe, as representações individuais são expostas sendo o comum integrado.

- Auto-explicação: Acontece quando o indivíduo começa a refletir consigo mesmo sobre o problema em questão [GT90]. Na colaboração, essa explicação compartilhada permite refletir de diferentes pontos de vista a mesma solução e reforçar o que foi aprendido.

- Carga cognitiva: A divisão horizontal de tarefas reduz a carga cognitiva de trabalhado por cada indivíduo. A carga cognitiva reduzida pode explicar porque é mais fácil regular o trabalho de um colega do que a auto-regulação e, portanto, por que os membros do grupo melhoram suas habilidades reguladoras.

- Conflito: Interações sociais, onde há discrepância entre o conhecimento de dois indivíduos, levam a declarações ou posições conflitantes em relação à tarefa em questão [DMJ+13].

Os processos citados podem ocorrer com mais frequência ou mais espontaneamente em abordagens colaborativas. Porém, não são específicos desse tipo de abordagem. Os mecanismos mais específicos da colaboração, são: 1) interação, já que esse processo implica diretamente na interação social; e 2) a apropriação, por meio da qual um indivíduo reinterpreta sua própria ação ou enunciação a partir do que seu par aponta [Fox87]. A maioria das pesquisas sobre colaboração buscou mensurar seus efeitos. Os efeitos mais específicos foram em termos de melhorias na autorregulação [Gil89] ou de mudança conceitual [Ros92].

Capítulo 3

Trabalhos Relacionados

Nesta seção, apresentaremos alguns trabalhos, baseados na literatura, para prover *feedback* em atividades de programação, que são trabalhos relacionados com o nosso. O objetivo desses trabalhos é de fornecer *feedback* aos estudantes sobre o sua aprendizagem. Discutimos inicialmente sobre os sistemas de avaliação automatizada no contexto de ensino de programação. Esses trabalhos propõem fornecer *feedback* automático principalmente sobre aspectos funcionais dos códigos dos estudantes. Em seguida, analisamos as ferramentas existentes para prover *feedback* sobre aspectos de qualidade dos códigos. Por fim, abordamos a utilização da revisão de código por pares no contexto educacional. Essa foi a abordagem que utilizamos como meio para investigarmos como os alunos de programação introdutória fornecem *feedback* e os efeitos (do *feedback*) na qualidade de suas soluções.

3.1 Avaliação Automática de Atividades de Programação

Na literatura encontramos diversas propostas de ferramentas automáticas para avaliação de código-fonte que identificam falhas nas submissões e fornecem algum tipo de *feedback*. A técnica mais comum utilizada nessas ferramentas é a de testes automáticos, baseada em juízes *online*. Janzen *et al.* [JCH13], por exemplo, propuseram o WebIDE, uma estrutura que permite que os instrutores desenvolvam e entreguem conteúdo *online* com *feedback* interativo para um curso introdutório de programação. O *feedback* é fornecido através de avaliadores automatizados que variam desde a avaliação da expressão regular simples até analisadores sintáticos para aplicações que compilam e executam programas e testes de unidade. Os auto-

res verificaram que os estudantes que usaram o WebIDE apresentaram resultados melhores em todas as avaliações do que os estudantes que não usaram.

O trabalho de Gao *et al.* [GPL16], também utiliza de testes automáticos para correção de atividades de programação. Os resultados mostraram que a estrutura de geração de *feedback* automatizada pode descobrir mais erros dentro de submissões de estudantes e fornecer *feedback* oportuno e útil. Um total de 142 erros perdidos foi encontrado dentro de 446 submissões. Entretanto, é preciso tempo e esforço para projetar um bom conjunto de casos de teste que pode testar o código-fonte do estudante completamente. Na prática, os testes utilizados para a classificação são muitas vezes insuficientes para um diagnóstico preciso.

A síntese de programa é outra técnica muito utilizada nas ferramentas de *feedback*. A síntese de programa consiste em sintetizar um programa em uma linguagem de programação subjacente, para alcançar um objetivo de determinada especificação [Gul10]. Rolim *et al.* [RSD+17], propuseram a REFAZER, uma abordagem para sintetizar transformações de programas por meio de exemplos (*programming by example* - PBE) para aprender correções de código-fonte como transformações de árvores sintática abstrata (*Abstract Syntax Tree* - AST), a partir de pares de atividades corretas e incorretas submetidas pelos estudantes.

Com base no REFAZER, Head *et al.* [HGS+17] apresentam duas ferramentas, também abordando o conceito de síntese de programa, para fornecer *feedback* automático para atividades de programação, são elas: *Mistakebrowser* e *Fixpropagator*. Na primeira, são identificados erros nos códigos submetidos pelos estudantes e permite ao professor deixar comentários. Na segunda, além de possibilitar que o professor comente, ela aprende o *feedback* para que ele seja fornecido em erros semelhantes em contextos diferentes.

Kim *et al.* [KKL+16] apresentaram a APEX, uma ferramenta, baseada em comparação de códigos. Ela funciona comparando a execução da implementação correta (elaborada por um instrutor) e a execução da implementação submetida pelo estudante. São coletados tanto traços concretos de execução quanto traços simbólicos. Ele usa um algoritmo iterativo para calcular correspondências que mapeiam uma instância de instrução para alguma instância na outra versão. Os resultados de sua aplicação mostram que a APEX pode identificar com precisão as causas e explicar a causalidade para 94,5% dos 205 erros de estudante e 15 erros coletados de Internet. Os resultados apontam para uma melhora na produtividade dos estudantes.

Observamos que, de modo geral, essas ferramentas podem auxiliar no acompanhamento das turmas e na agilidade do *feedback* das atividades de programação. Porém, algumas dessas ferramentas ainda precisam do trabalho do professor em algum momento, como para elaborar testes automáticos. Elaborar casos de teste, além de exigir tempo do professor, pode ser difícil prever todos os cenários possíveis. Notamos também que há mais esforços de pesquisa no sentido do *feedback* funcional. No entanto identificamos alguns trabalhos que abordam aspectos qualitativos do código-fonte, apresentamos alguns exemplos a seguir.

3.2 *Feedback* da Qualidade do Código-fonte

Como observado, o foco principal do campo de *feedback* de atividades de programação tem sido na correção funcional dos programas, embora algumas ferramentas também incorporem *feedback* sobre aspectos de qualidade [KHJ17]. Joy *et al.* [JGB05], por exemplo, propõe o BOSS, um sistema para avaliação de trabalhos. Ele executa testes automatizados para correção e qualidade do código-fonte, verifica plágios, e fornece *feedback*. Tipicamente as ferramentas que fornecem *feedback* automático, como o QCheck [ASF16], são baseadas em métricas de engenharia de software, como: complexidade ciclomática (cc), volume de Halstead, PEP 8 e linhas lógicas de código (lloc), e integradas a ferramentas lint. Porém, em alguns casos, o *feedback* pode ser genérico e pouco eficaz para estudantes iniciantes. Nesse sentido, outros estudos foram desenvolvidos incluindo os estudantes como revisores.

Hundhausen *et al.* [HAT11] desenvolveram uma adaptação da instrução baseada em estúdio para o ensino de computação, chamada de revisão pedagógica de código (Pedagogical Code Review - PCR). O objetivo foi explorar métodos instrucionais para refinar soluções de forma interativa, por meio da revisão crítica entre estudantes e instrutores. Eles desenvolveram um ambiente *online* que permite que as PCRs ocorram de forma assíncrona fora da sala de aula e comparara os resultados entre um curso de CS1 com PCRs *online* com um curso CS1 com PCRs face-a-face. O estudo identificou que no curso com PCRs face a face a autoeficácia, qualidade das revisões e aprendizado entre pares foram significativamente maiores e os estudantes foram mais positivos quanto ao PCR.

Wang *et al.* [WLF+12] apresenta o EduPCRe, um sistema de avaliação on-line baseado no processo de inspeção de código-fonte usado na indústria de software de revisão de pares.

Nessa abordagem, os estudantes revisam o código de outros estudantes, compartilham ideias e fazem sugestões de modo interativo e colaborativo. Os professores avaliam e atribuem pontuações aos estudantes com base em seu desempenho, revisão realizada e permanência no processo de revisão por pares. Os autores observaram melhorias significativas na aprendizagem dos estudantes em vários aspectos, como: habilidades de programação, conformidade com padrões de codificação, fazer e receber críticas e trabalho colaborativo.

3.3 Revisão de Código por Pares

O processo de revisão de código por pares vem sendo utilizada em diferentes contextos. Nas wikis, por exemplo, é comum que os membros das comunidades colaborem na criação e atualização dos materiais que são compartilhados. Em ambientes científicos também é habitual sua utilização, principalmente na revisão de artigos para conferências, e na engenharia de software identificamos essa abordagem em projetos de código aberto e atividades de revisão de código por pares (RCP) [Try05].

A RCP é uma abordagem utilizada por desenvolvedores e, quando realizada de forma adequada, pode melhorar significativamente a qualidade do código do projeto de software, assim como aperfeiçoar continuamente as habilidades técnicas dos profissionais [KS99]. A finalidade de utilizar a revisão por pares é melhorar a qualidade de determinado trabalho, a partir do *feedback* colaborativo. Desse modo, alguns pesquisadores defendem que a revisão de código por pares no cenário educacional pode trazer melhoras significativas a aprendizagem de quem colabora [GLCM16] [WLF+12].

De acordo com Trytten [Try05], a RDC, quando aplicada no ambiente educacional, tem objetivos característicos, um deles é apresentar aos alunos que há comunicação entre os engenheiros de software. Como Trytten [Try05] destaca, os estudantes podem idealizar que programar é uma atividade solitária, pois é comum nas disciplinas de programação os alunos programarem seus códigos individualmente e serem ensinados que é uma prática ruim se basear em códigos de outras pessoas. Entretanto, a realidade é bem diferente. Devido ao tamanho e complexidade dos projetos de softwares atuais, é necessário que os engenheiros trabalhem grande parte da vida profissional em equipe. Assim, a RDC pode auxiliar na conscientização dos estudantes, tanto os introvertidos quanto os mais sociais, sobre a importância

de se conectarem uns com os outros e de um ambiente de colaboração para construção mútua da aprendizagem.

O outro objetivo, segundo [Try05], é estimular os alunos a aprenderem a ler o código. Ler código e escrever código são atividades distintas e ambas têm sua importância no contexto de mercado. É comum que alunos iniciantes tenham dificuldade para resolver determinados problemas de programação por não possuir maturidade para elaborar soluções alternativas para resolvê-lo. Porém, essa habilidade poderia ser desenvolvida mais facilmente se os alunos fossem incentivados a analisar outras soluções para o problema. A RDC permite que os alunos vejam diferentes formas de solucionar um mesmo problema já resolvido.

Assim sendo, a RDC é vista na literatura como uma alternativa colaborativa para atividades de sala de aula e, por ser aplicada em diversos cenários distintos, acredita-se que ela pode funcionar bem em um ambiente educacional. Propor que as atividades (tarefas) dos estudantes sejam distribuídas a um conjunto de revisores (outros estudantes) para avaliar e prover *feedback* formativo sobre a solução, pode ajudar os estudantes a se tornarem praticantes reflexivos de seu ofício e autônomos na própria aprendizagem. Na educação, a revisão por pares beneficia o aprimoramento da aprendizagem dos estudantes [KS99].

Entendendo o potencial da RDC em auxiliar o aprendizado de programação, utilizamos ela como meio para investigarmos como os alunos de programação introdutória fornecem *feedback* e os efeitos na qualidade de suas soluções. Os trabalhos citados reforçam que incluir o estudante como avaliador pode ter impacto positivo no seu desempenho e aprendizado na disciplina. Nossa proposta tem como diferencial observar o contexto da turma de Programaçao I do BCC/UFCG e abordar exclusivamente problemas relacionado à qualidade do código-fonte, por meio de dicas personalizadas elaboradas pelos próprios estudantes em atividades de revisão de código por pares.

Capítulo 4

Os Alunos Conseguem Elaborar *Feedback* sobre a Qualidade do Código?

Nesta dissertação verificamos se, ao incluirmos os alunos como avaliadores, poderíamos fornecer *feedback* efetivo para outros estudantes sobre a qualidade do código-fonte. Portanto, primeiramente buscamos investigar se os estudantes conseguem elaborar esse tipo de *feedback*. Entendemos que para incluir os estudantes como avaliadores é importante que haja equivalência entre seu *feedback* com o *feedback* elaborado pelos professores. Assim, verificamos se há similaridade entre os problemas de qualidade de código-fonte identificados pelos alunos e pelos professores da disciplina. Em seguida, observamos quais são os problemas de qualidade de código mais reportados pelos estudantes. Por fim, analisamos o quão úteis podem ser as dicas elaboradas pelos alunos, de acordo com a avaliação dos professores. Participaram desse experimento alunos matriculados em Programação I do curso de Ciência da Computação (BCC) da Universidade Federal de Campina Grande (UFCG) e dos cursos de Licenciatura em Ciência da Computação (LCC) e Sistemas de Informação (BSI) da Universidade Federal da Paraíba/*Campus IV* (UFPB). Neste capítulo descrevemos esses experimentos.

4.1 *Design* do Estudo

O método de pesquisa que empregamos neste experimento foi um *survey* supervisionado. O *survey* foi elaborado para verificar se os alunos conseguem fornecer *feedback* sobre como

melhorar qualitativamente o código-fonte de outros estudantes. Caso consigam, identificar quais problemas de qualidade são reportados e se têm similaridade com os identificados pelos professores, para assim responder às questões de pesquisa:

- **Q1**: O quão similar aos professores da disciplina os alunos de Programação I conseguem identificar problemas na qualidade do código de outros estudantes?
- **Q2**: Quais problemas de qualidade de código são identificados mais frequentemente pelos alunos de Programação I?
- **Q3**: O quão útil são as dicas dos alunos de Programação I sobre problemas na qualidade do código?

O *survey* foi aplicado presencialmente para os alunos durante o horário de aula e o tempo para conclusão foi de até 60 minutos.

4.2 Estudo de Caso 1: Ciência da Computação - UFCG

4.2.1 Visão Geral da Disciplina de Programação I do BCC/UFCG

Os professores da disciplina de Programação I da UFCG abordam aspectos práticos e teóricos de forma intercalada. Eles geralmente apresentam aspectos teóricos em sala de aula e fornecem uma série de tarefas práticas a serem desenvolvidas na linguagem de programação Python durante as aulas de laboratório.

A disciplina é composta por dez unidades de estudo sequenciais. Cada uma delas aborda um tópico diferente, conforme descrito na Tabela 4.1. Cada aluno avança em seu próprio ritmo, o que significa que pode haver alunos em unidades diferentes.

Tabela 4.1: Unidades da disciplina de Programação I do BCC.

Unidade	Descrição	Unidade	Descrição
1	Conceitos elementares de programação.	6	Funções.
2	Escrevendo programas simples.	7	Estrutura de dados: array.
3	Condições, alternativas e funções.	8	Estrutura de dados: listas.
4	Laços definidos.	9	Estrutura de dados: sequências de sequências e matrizes.
5	Laços indefinidos.	10	Estrutura de dados: mapas.

A metodologia de ensino adotada é baseada em três aspectos principais: (i) *flipped classroom*, na qual os alunos aprendem em seus próprios ambientes, assistindo vídeos disponíveis por seus professores, lendo lições detalhadas, realizando tarefas, entre outras atividades. Nessa metodologia, o professor desempenha o papel de mediador e o tempo da sala de aula é dedicado a discutir as dúvidas dos alunos; (ii) avaliação continuada, onde os alunos são submetidos a uma avaliação a cada semana para verificar seu desempenho na unidade atual; e (iii) *mastery learning* [B$^+$71b], onde, para avançar de uma unidade para outra, os alunos precisam demonstrar as habilidades e competências necessárias para dominar a unidade. Na disciplina da UFCG, eles têm que responder corretamente duas tarefas por unidade para demonstrar proficiência na unidade.

4.2.2 Participantes

Participaram do estudo 4 professores e 75 dos 114 alunos matriculados na disciplina de Programação I, do BCC/UFCG, no primeiro semestre letivo de 2017. Realizamos o experimento durante o horário de aula. Portanto, participou quem estava presente. Contudo, para verificar se a quantidade de alunos representa a turma estudada, realizamos o cálculo do tamanho amostral (1), onde n = amostra calculada, N = população, Z = variável normal padronizada associada ao nível de confiança, p = verdadeira probabilidade do evento e *e* = erro amostral.

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

Verificamos que nossa amostra representa a população do estudo com erro amostral (diferença entre número estimado e número real) próximo a 7% e com 95% de confiança (probabilidade de que o erro amostral efetivo seja menor que o erro amostral admitido).

Conforme mencionado na subseção anterior, devido à metodologia da disciplina, há alunos em diferentes unidades de estudo no mesmo assunto. A unidade 4 apresenta, historicamente, maior retenção dentre as unidades. Nesse estudo, a unidade 4 representou pouco mais de 49% dos participantes. No grupo de estudantes, não identificamos nenhum aluno da unidade 9. Na Figura 4.1, apresentamos a quantidade de alunos por unidade.

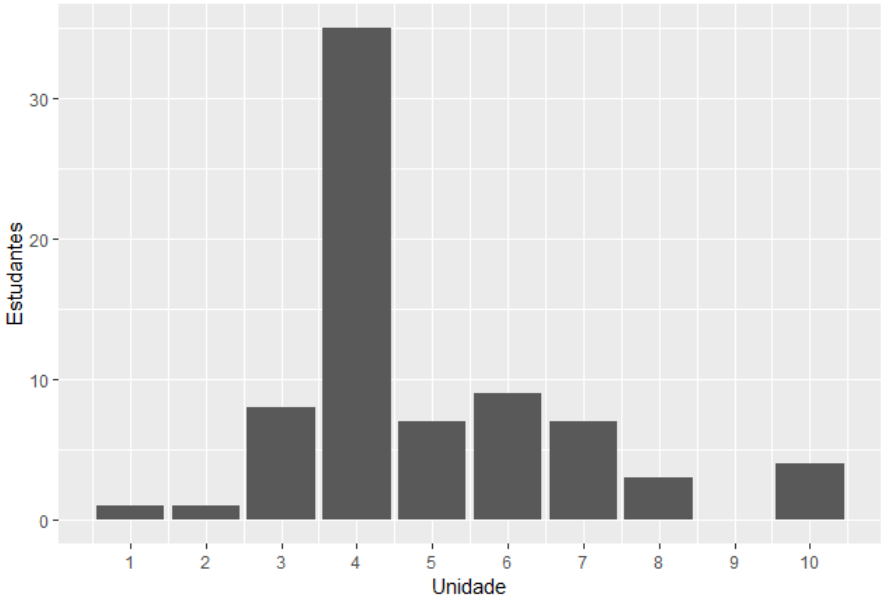

Figura 4.1: Número de alunos participantes por unidade em Programação I do BCC em 2017.1.

4.2.3 *Survey*

Aplicamos um *survey* com atividades de programação e seus respectivos códigos-fonte programados por alunos de Programação I de semestres anteriores. Todas as soluções são funcionalmente corretas, mas apresentam problemas quanto a qualidade. Nesse estudo, nos concentramos nos seguintes problemas: (i) **complexidade** (código duplicado, código dispensável e legibilidade do código); (II) **espaçamento** (espaçamento entre linhas, espaçamento entre caracteres e indentação); (iii) **variáveis** (tipo, ausência, excesso e nomenclatura) e; (iv) **cabeçalho** (ausência, dados incompletos e excessivos). A seguir apresentamos um exemplo

de um código que foi usado nesse experimento (Código-Fonte 3.1).

Código Fonte 4.1: Código-fonte correto, mas com problemas de qualidade.

```
1  # coding: utf-8
2  # xxxx.xxxxxxxx / xxxx / 201x.x
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 15
18         cont += 1
19     print cont
```

Os sujeitos deste estudo analisaram os códigos e deram dicas para melhorá-los qualitativamente. Selecionamos as atividades e suas respectivas soluções, considerando a presença dos principais problemas de qualidade do código abordados na disciplina (detalhado na Tabela 4.2).

4.2.4 Métricas

Para responder a primeira questão de pesquisa (Q1-1), comparamos e analisamos o *feedback* a partir da codificação das dicas. Para isso, utilizamos uma técnica de análise qualitativa de dados [BK15], onde: primeiramente, lemos o texto das dicas. Em seguida, rotulamos os aspectos de qualidade de código-fonte importantes. Então, definimos quais *tags* são mais importantes e categorizamos. Por fim, definimos quais são mais relevantes e como estão conectadas. Para ilustrar esse processo, vamos analisar o seguinte exemplo de dica fornecida:

Tabela 4.2: Aspectos gerais de qualidade de código avaliados no BCC.

Aspecto	O que é analisado
Cabeçalho	Verifica se a solução possui cabeçalho, se está de acordo com a questão e não há ausência de informações.
Complexidade	Verifica se a solução está mais complexa do que deveria e se é possível simplificá-la.
Variável	Verifica a ausência ou excesso de variáveis no código e aspectos relacionados à nomenclatura.
Espaçamento	Verifica a falta ou excesso de espaços entre linhas e caracteres do código e problemas relacionados à indentação.

"Há código duplicado nas linhas 14 e 22. Os prints poderiam ser só no final do programa para tornar o código mais legível". Para esse caso, definimos as seguintes *tags*: **complexidade**, **código duplicado** e **legibilidade do código**. A classificação por *tags* possibilitou comparar e analisar as dicas (presumivelmente subjetivas) com maior precisão. Cada dica é composta por uma *tag* ou um conjunto delas. A Figura 4.2 mostra as *tags* que identificamos. Para cada dica, consideramos pelo menos 1 dos 4 aspectos gerais de qualidade de código-fonte mencionados na Tabela 4.2.

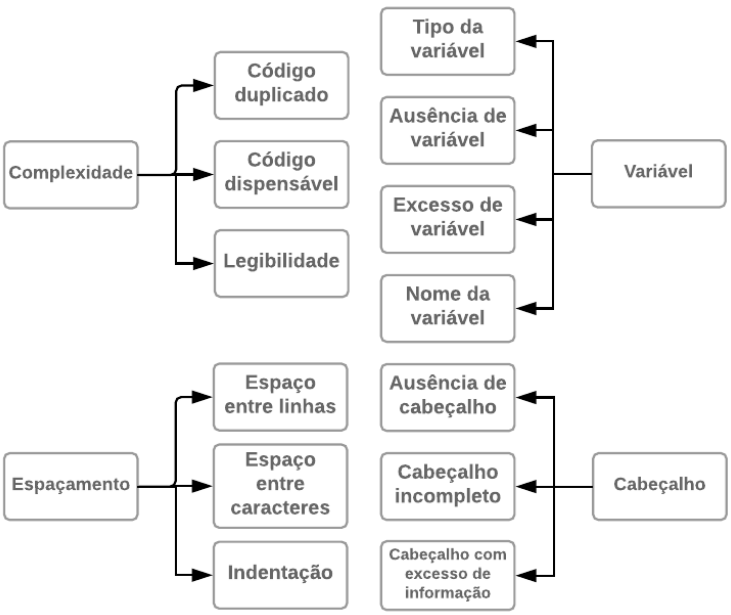

Figura 4.2: *Tags* dos problemas de qualidade de código encontrados neste estudo.

Neste estudo usamos o coeficiente de Jaccard (2) para medir a similaridade. O coeficiente de Jaccard compara o número de elementos semelhantes entre dois grupos e o número total de elementos envolvidos, excluindo o número de ausências conjugadas. Nesse modelo, o índice de similaridade varia entre 0 e 1, sendo que quanto mais próximo de 1, maior a similaridade entre os dois grupos. Calculamos a similaridade entre as dicas de cada aluno (Sf) e as dicas do *feedback* referência, elaborado a partir das dicas dos professores (Tf).

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

Primeiramente, medimos a similaridade entre os professores de acordo com as dicas que eles elaboraram. Fizemos isso para verificar se eles podem fornecer *feedback* semelhante sobre problemas de qualidade de código e, consequentemente, serem abordados como um grupo. Calculando o coeficiente de similaridade, verificamos que os professores apresentaram um índice de correlação entre 0.50 e 0.86 (comparando em pares). Em seguida, analisamos as dicas dos professores para criar um *feedback* referência sobre problemas de qualidade de código a serem identificados pelos estudantes. Esse *feedback* referência inclui dicas baseadas em sua frequência, isto é, selecionamos, para cada tarefa, dicas semelhantes elabordas por pelo menos 2 dos 4 professores. Por fim, calculamos a similaridade entre os problemas de qualidade identificado pelos alunos e professores.

Na literatura, não há índice ideal a ser alcançado com o coeficiente de Jaccard. Neste estudo, consideramos significativo, mesmo que não ideal, o índice de similaridade a partir de 0.5 (50% similar). A semelhança de 50% não é ideal, mas consideramos neste estudo um valor significativo, considerando que os alunos ainda estão aprendendo a prover *feedback*.

Para responder a Q2-1, ranqueamos os problemas de qualidade de código que os alunos mais reportaram em suas dicas. Para responder a Q3-1, apresentamos as dicas elaboradas pelos alunos para que os professores as classificassem como úteis ou inúteis. Definimos como útil a dica correta e suficientemente explicativa para melhorar o código-fonte a partir dela (por exemplo, "*Na linha 11 tem uma varíavel 'reprovados' que não é utilizada no restante do código.*"). Definimos como inútil a dica 1) correta, mas não explicativa o suficiente para melhorar o código-fonte a partir dela; 2) irrelevante para melhorar a qualidade do código; ou 3) incorreta (por exemplo, "*Há váriáveis dispensáveis no código*"). Cada dica foi avaliada por até 3 dos 4 professores da disciplina.

De modo complementar, também avaliamos quantitativamente os dados que coletamos para Q1-1 e Q3-1 usando estatística descritiva. Realizamos algumas análises estatísticas simples usando o teste de hipótese.

4.2.5 Análise e Discussão

Calculamos a semelhança das dicas dos alunos pela média dos valores do coeficiente de Jaccard que ele alcançou em cada atividade quando comparado ao *feedback* de referência. A Figura 4.3 mostra uma visão geral da taxa de similaridade que os alunos alcançaram por unidade em que se encontravam no momento do experimento.

Q1-1: O quão similar aos professores da disciplina os alunos de Programação I do BCC/UFCG conseguem identificar problemas na qualidade do código de outros estudantes?

Nesse contexto, analisamos os alunos por unidade de estudo para identificar se há relação entre a similaridade e o nível de conhecimento em programação, as unidades são sequenciais entre 0 e 10, sendo que quanto mais próximo de 10 mais avançado o aluno está.

Percebemos que há variação entre a similaridade dos alunos em todas as unidades. De acordo com a Figura 4.3, os pontos que representam a mediana de alunos por unidade, mostram que mais da metade deles elaborou dicas com similaridade igual ou superior à 50% em relação às dicas elaboradas pelos professores da disciplina. A semelhança de 50%, embora não seja ideal, é um valor significativo. De modo que, se mais de um aluno elaborar dicas para uma determinada solução, o *feedback* pode ser eficaz ou mais eficaz do que as sugestões de apenas um aluno.

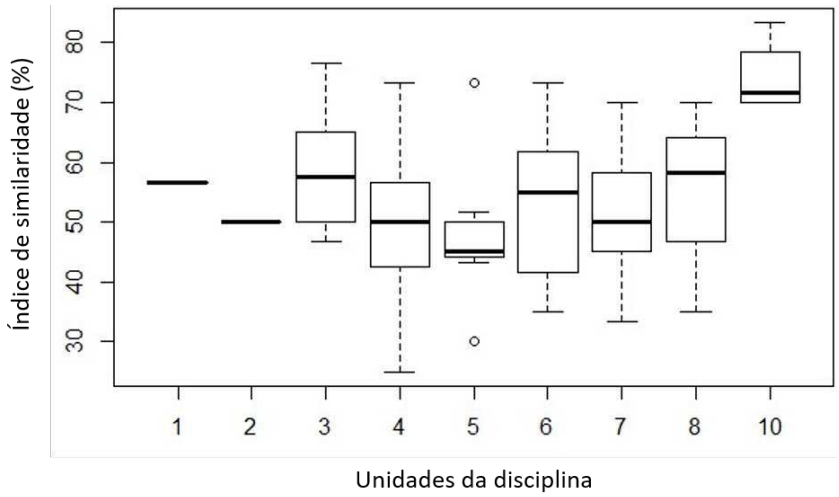

Figura 4.3: Similaridade entre as dicas dos alunos e professores do BCC.

No grupo dos alunos, não identificamos nenhum da unidade 9. O maior índice de similaridade alcançado foi de aproximadamente 75% por um aluno da unidade 10 (o que faz sentido, porque quanto mais avançado, provavelmente mais conhecimento o aluno tem sobre esses conceitos) e alguns das unidades 3 e 6 se aproximaram desse número. A unidade 5 foi a única que atingiu um número mediano inferior. Verificamos que as últimas unidades apresentaram melhores resultados. Na unidade 10, todos apresentaram similaridade maior ou igual à 70%, mas também é importante notar que esta unidade possui menos alunos que as demais.

A unidade 10 foi a que ocorreu menor variação entre similaridade dos alunos. Assim, concluímos que, nesse contexto, o *feedback* mais semelhante ao dos professores tende a ser o de alunos mais avançados em programação, e não daqueles que estão aprendendo, como menciona Glassman *et al.* [GLCM16]. Concluímos que os alunos, como grupo, são capazes de identificar uma quantidade significativa, mesmo que não ideal, de problemas relacionadas à qualidade do código semelhante aos identificados pelos professores. No entanto, um grupo de estudantes, especialmente os mais avançados na disciplina, pode identificar esses problemas com maior precisão.

De modo complementar, buscamos verificar se a proporção de alunos que elaboram *feedback* similar aos professores da disciplina é significativamente maior. Consideramos admissível a similaridade maior que 0.5 (50% similar). Usando o teste de proporção, constatamos que 60% dos alunos alcançam a similaridade maior que 50% em comparação

com os professores (*p-value* < 0.04, nível de significância de 0.05). Assim, refutamos a hipótese nula. Ou seja, para esse cenário, a maior parte dos estudantes consegue alcançar um índice admissível de similaridade com os professores da disciplina, mesmo que em alguns casos essa similaridade não seja a ideal.

Q2-1: Quais problemas de qualidade de código são identificados mais frequentemente pelos alunos de Programação I do BCC/UFCG?

Primeiramente, consideramos os principais problemas de qualidade de código analisados na disciplina (Tabela 4.2). Verificamos que a maior parte das dicas aborda problemas de complexidade, como mostra a Tabela 4.3. O segundo tipo mais presente inclui dicas relacionadas aos cabeçalhos das soluções. Essa questão considera a falta de informação e clareza no seu desenvolvimento. Dentre os fatores que podem ter influenciado nesse resultado, destacamos o fato de que o cabeçalho é, dentre os problemas analisados, o mais simples de ser reconhecido e, por ocultarmos as informações dos autores das soluções (alunos de semestres anteriores), alguns alunos podem ter mencionado isso como um problema, mesmo que tenha sido esclarecido antes que o experimento acontecesse. Não há diferença significativa entre a quantidade de dicas sobre cabeçalho, espaçamento e variáveis.

Tabela 4.3: Quantidade de dicas por aspecto de qualidade de código reportado no BCC.

Aspecto de qualidade	**Quantas vezes foi reportado**
Complexidade	203
Cabeçalho	84
Espaçamento	77
Variável	62

Os estudantes também identificaram outros problemas de qualidade de código mais específicos, além dos problemas já mencionados. Para entender melhor esse cenário, decidimos ranquear os problemas de qualidade identificados nas dicas, como mostra a Figura 4.4.

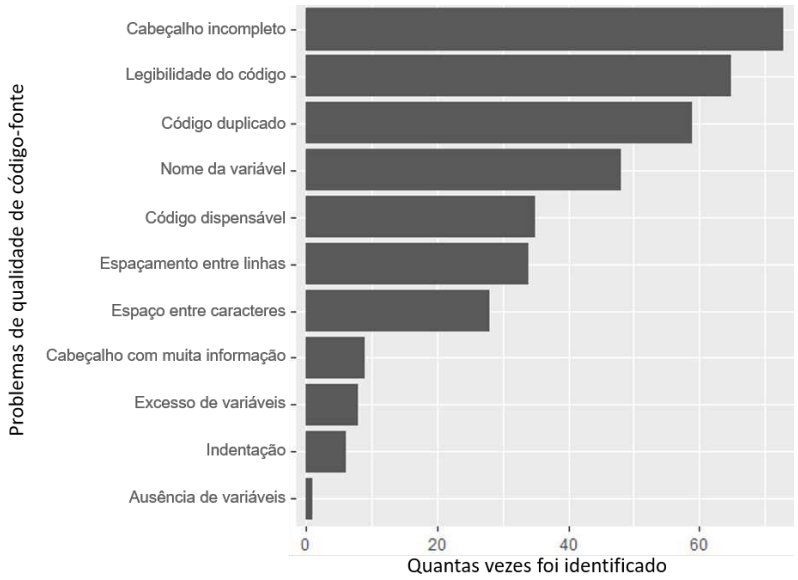

Figura 4.4: *Ranking* dos problemas de qualidade do código relatados nas dicas dos alunos do BCC.

Depois de cabeçalho incompleto, a maior quantidade de dicas foi sobre legibilidade e dispensabilidade do código. Nesta pesquisa, classificamos como legibilidade, as sugestões de boas práticas e a clareza do código sem remover ou adicionar novas funções. Enquanto código dispensável, como sugere o nome, são trechos do código que, se removidos, não influenciarão no funcionamento do programa.

Percebemos que as dicas sobre o uso de variáveis constantes e tipos de variáveis não foram identificadas pelos alunos, nesse caso, essas sugestões foram elaboradas apenas pelos professores. O oposto aconteceu com as dicas sobre muita informação no cabeçalho e adição de comentários ao código. A última foi bastante citada, mas não é considerada boa prática. Além disso, foram mencionados erros gramaticais nos textos dos cabeçalhos e nas saídas de alguns programas.

Q3-1: O quão útil são as dicas dos alunos de Programação I do BCC/UFCG sobre problemas na qualidade do código?

Para responder essa questão de pesquisa, selecionamos todas as dicas elaboradas pelos alunos e as apresentamos aos professores da disciplina para que as classificassem como úteis ou não. Neste contexto, definimos com útil a dica correta, personalizada e clara o suficiente

para melhorar o código a partir dela. Quanto a não ser útil, classificamos dicas: (i) corretas, mas não claras o suficiente para melhorar o código, (ii) irrelevantes para melhorar o código ou (iii) incorretas. Para essas definições, consideramos, além da corretude, o detalhamento da dica, pois esse é o diferencial de incluir o aluno nessa atividade. A Figura 4.5 mostra a visão geral dos dados obtidos.

Verificamos que a maior parte dos alunos tiveram mais de 50% das suas dicas classificadas como úteis. Nós também verificamos que a maioria das dicas eram sobre problemas de complexidade. Como na Q1-1, as últimas unidades obtiveram melhores resultados e a unidade 5 apresentou resultados insatisfatórios. Em média, os alunos deram 3 dicas por atividade e boa parte delas estão corretas, mesmo que não fossem úteis. No entanto, os professores mencionaram que algumas dicas foram mal detalhadas ou tiveram problemas com a terminologia de programação.

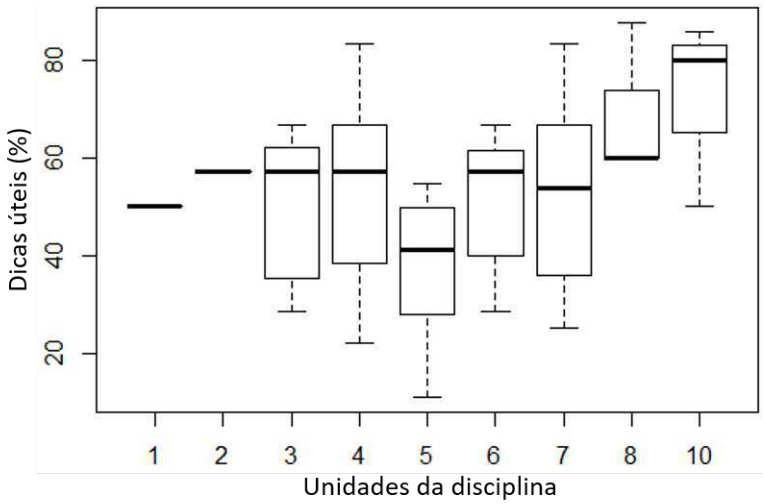

Figura 4.5: Dicas úteis dadas pelos alunos do BCC.

Para complementar a resposta da Q3-1, buscamos verificar se a proporção de alunos que elaboram dicas úteis é significativamente maior. Consideramos como casos de sucesso os alunos cujo as dicas foram avaliadas maior que 0.5 (50% úteis). Usando o teste de proporção, constatamos que 68% dos alunos tiveram suas dicas avaliadas como úteis ($p\text{-}value < 0.00091$, nível de significância de 0.05). Assim, refutamos a hipótese nula. Ou seja, para esse cenário, a maior parte dos estudantes consegue elaborar mais dicas úteis, mesmo que em alguns casos a porcentagem de dicas úteis não seja a ideal.

Desse modo, acreditamos que os estudantes podem fornecer *feedback* sobre os problemas do código-fonte de seus colegas. No entanto, defendemos que o *feedback* elaborado por mais de um aluno tende a ser mais efetivo.

4.2.6 Ameaças à Validade

No geral, buscamos com o *design* deste estudo minimizar muitas das ameaças discutidas nesta Seção. Para organizar esta Seção, foram classificadas as ameaças à validade usando as categorias: Conclusão, Interna, *Construct* e Externa [WRH+12].

Conclusão: As ameaças relacionadas a essa categoria dizem respeito à conclusão do experimento. Devido à metodologia baseada em *flipped classroom*, onde a sala de aula é para discussão e dúvidas, o tamanho da amostra do experimento pode ter sido insuficiente para obter conclusões profundas sobre os resultados. No entanto, fazendo o cálculo amostral, descobrimos que nossa amostra representa a população do estudo com erro de amostral próximo à 7% e com 95% de nível de confiança. Por fim, a classificação das dicas como úteis ou inúteis ocorreu considerando exclusivamente a avaliação de professores de Programação I. Contudo, devido a experiência e conhecimento acerca do PEP 8, afirmamos que a classificação foi adequada.

Interna: Como o estudo envolve participação humana ativa, ele se inclina para ameaças internas. Uma vez que os alunos avaliaram muitas soluções, é possível que em algum momento estivessem cansados ou entediados para avaliar com precisão. Além disso, foi necessário convidar instrutores auxiliares para que o experimento pudesse ser aplicado simultaneamente para todas as turmas (3 salas de aula), então, a orientação dada por cada instrutor pode ter influenciado a compreensão dos alunos sobre o experimento. Para minimizar essa ameaça, elaboramos um roteiro e treinamos os instrutores auxiliares, a fim de garantir que todos os estudantes participassem do estudo com as mesmas condições.

***Construct*:** O modo como construimos a métrica do *feedback* referência pode ter influenciado negativamente no indice de similaridade entre as dicas dos alunos com as dicas dos professores da disciplina. No mais, buscamos utilizar técnicas empiricamente

validadas e comumente usadas nos estudos empíricos científicos da comunidade de Educação em Ciência da Computação.

Externa: Os participantes desse estudo são representativos apenas do contexto de Programação I do BCC/UFCG e do semestre em que ocorreu. Dessa forma, talvez não possamos generalizar os resultados desse experimento para outros contextos. Esse estudo deve ser replicado em outros cursos de Computação nas disciplina de Programação I para obter resultados mais genéricos.

4.2.7 Conclusões

Como mencionamos, os estudos no campo de *feedback* automático para atividades de programação abordam principalmente aspectos funcionais dos códigos. Embora existam ferramentas que analisam a qualidade do código, seu *feedback* pode não ser detalhado o suficiente e, para serem efetivas, elas exigem a análise manual dos professores. Dessa forma, investigamos se, ao incluir alunos como avaliadores, poderíamos fornecer *feedback* personalizado. Portanto, neste estudo, a questão principal é se os alunos podem avaliar qualitativamente os programas de seus pares.

Com os resultados obtidos, verificamos que a maioria das dicas elaboradas pelos alunos estão relacionadas à complexidade do código. Verificamos também que a unidade de estudo influenciou na similaridade com as dicas dos professores e a quantidade de dicas consideradas úteis neste estudo. Ou seja, os alunos com mais conhecimento em programação identificam problemas e fornecem *feedback* melhor. Além disso, comprovamos estatisticamente que a quantidade de alunos que elaboram dicas úteis e sobre problemas de qualidade similares aos identificados pelos professores é proporcionalmente maior.

Percebemos também que, de maneira geral, os alunos podem identificar problemas de qualidade de código e elaborar boas dicas em um nível significativo, mesmo que não seja o ideal. Assim, acreditamos que é possível incluir os alunos como avaliadores da qualidade de código, porém, se mais de um aluno elaborar dicas para uma determinada solução, o *feedback* tende a ser eficaz ou mais eficaz do que as dicas de apenas um aluno. Com intuito de verificar se os resultados obtidos são semelhantes em turmas diferentes, replicamos esse estudo em outro contexto. A seguir descrevemos a experiência.

4.3 Estudo de Caso 2: Sistemas de Informação e Licenciatura em Ciência da Computação - UFPB

Com intuito de verificar se os resultados obtidos no estudo anterior se repetem em contextos diferentes, replicamos o experimento em duas turmas de Programação I. As turmas que participaram são dos cursos de LCC e SI, ambos da UFPB/*Campus IV*. Nesta seção detalhamos as particularidades desse experimento em relação ao anterior e apresentamos os resultados obtidos.

4.3.1 Participantes

Participaram 4 especialistas, alunos de pós-graduação em Ciência da Computação que pesquisam sobre ensino de programação introdutória, e 50 alunos matriculados na disciplina de Programação I, sendo 27 do curso de Sistemas de Informação (BSI) e 23 da Licenciatura em Ciência da Computação (LCC), ambos da UFPB, do segundo semestre letivo de 2017. O experimento ocorreu no horário da aula, ou seja, participou quem estava presente. A metodologia de ensino nos cursos de BSI e LCC é tradicional, ou seja, diferente do BCC não foi possivel dividir os alunos por unidade. Logo, aqui não foi realizada a análise considerando o nível de conhecimento dos alunos em programação.

4.3.2 *Survey*

Aplicamos um *survey* com atividades e seus respectivos códigos-fonte elaborados por estudantes de Programação I de LCC/UFPB em semestres anteriores. Os códigos têm problemas quanto à qualidade, como: mais complexidade que o necessário, repetições de trechos de código, excesso de espaçamento, entre outros. Neste estudo, nos concentramos nos seguintes problemas de qualidade: (i) **complexidade** (duplicação de código, código dispensável e legibilidade do código); (ii) **espaçamento** (espaçamento entre linhas, espaçamento entre caracteres e indentação); e (iii) **variáveis** (tipo, ausência, excesso e nomenclatura). Diferente do estudo realizado no BCC/UFCG, desconsideramos o cabeçalho em nossa avaliação, pois esse aspecto não é cobrado nas atividades das turmas participantes.

Tabela 4.4: Aspectos gerais de qualidade de código avaliados no BSI e LCC.

Aspecto	O que é analisado
Complexidade	Verifica se a solução está mais complexa do que deveria e se é possível simplificá-la.
Variável	Verifica a ausência ou excesso de variáveis no código e aspectos relacionados à nomenclatura.
Espaçamento	Verifica a falta ou excesso de espaços entre linhas e caracteres do código e problemas relacionados à indentação.

Os alunos analisaram os códigos-fonte e elaboraram dicas para melhorá-los qualitativamente. As atividades presentes no *survey*, assim como seus respectivos códigos, foram selecionadas considerando a presença dos principais aspectos de qualidade do código abordados neste estudo (detalhado na Tabela 4.4).

4.3.3 Métricas

Para responder a Q1-2, assim como no estudo realizado em Programação I do BCC/UFCG, também comparamos e analisamos o *feedback* a partir da codificação das dicas, utilizando a mesma técnica de análise qualitativa de dados [BK15]. Contudo, nos cursos de LCC e BSI, não consideramos a *tag* cabeçalho nem as *tags* derivadas dela, pois esse aspecto não é cobrados nas atividades realizadas nos cursos em questão.

Para cada dica, consideramos pelo menos 1 dos 3 aspectos gerais de qualidade de código-fonte (detalhados na Tabela 4.4). Outra diferença é que o *feedback* referência dos problemas de qualidade de código a serem identificados pelos estudantes for definido a partir das dicas dos especialistas. Devido a problemas de logística, não foi possível que os professores de Programação I de BSI e LCC elaborassem as dicas. No entanto, os especialistas são alunos da pós-graduação em Ciência da Computação que pesquisam e tem experiência no ensino de programação introdutória.

Para responder a Q2-2, ranqueamos os problemas de qualidade de código que os alunos mais reportaram em suas dicas. Para responder a Q3-2, apresentamos as dicas elaboradas pelos alunos para que professores da disciplina as classificassem como úteis ou inúteis. Cada dica foi avaliada por até 3 dos 4 professores da disciplina de Programação I do BCC/UFCG.

De modo complementar, também avaliamos quantitativamente os dados que coletamos para Q1-2 e Q3-2 usando estatística descritiva. Realizamos algumas análises estatísticas simples usando o teste de hipótese.

4.3.4 Análise e Discussão

Calculamos a similaridade das dicas do aluno a partir da média da similaridade (coeficiente de Jaccard (2)) que ele alcançou em cada questão, quando comparadas com o *feedback* referência definido a partir das dicas dos especialistas. A Figura 4.6a apresenta a visão geral dos dados obtidos da similaridade dos alunos por curso.

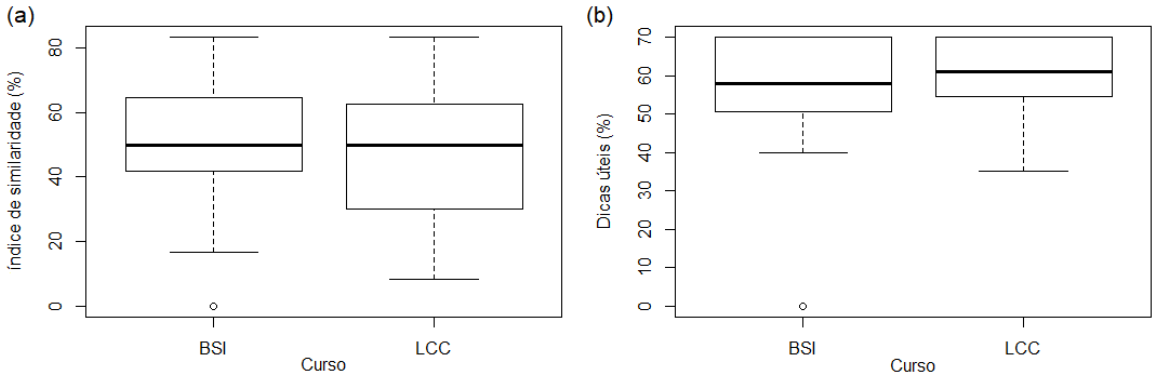

Figura 4.6: Visão geral dos resultados do BSI e LCC: (a) Similaridade entre aspectos de qualidade de código identificados por alunos e especialistas e (b) Dicas úteis dos estudantes.

Q1-2: O quão similar aos especialistas os alunos de Programação I do BSI e LCC da UFPB conseguem identificar problemas na qualidade do código de outros estudantes?

Analisamos a similaridade das dicas dos estudantes com os especialistas por curso, no intuito de verificar se existe diferença significativa quanto à forma de identificarem problemas na qualidade dos códigos-fonte. Percebemos que existe variação da similaridade entre os estudantes em ambos os cursos, a variação no LCC foi maior. De acordo com a Figura 4.6a, os pontos que representam o índice de similaridade mediano dos alunos por curso, apontam que mais da metade dos alunos de BSI elaboraram dicas com similaridade igual ou superior a 50% em relação às dicas elaboradas pelos especialistas. A similaridade de 50% por mais que não seja a ideal, é um valor significativo, considerando que os alunos ainda estão aprendendo

a prover *feedback*. O maior índice de similaridade alcançado foi de aproximadamente 83% em ambos os cursos.

De modo complementar, buscamos verificar se a proporção de alunos que elaboram dicas similares com as dos especialistas é significativamente maior. Neste contexto, consideramos admissível a similaridade maior que 0.5 (50% similar). Usando o teste de proporção, constatamos que 62% dos alunos alcançam a similaridade maior que 50% em comparação com os especialistas (*p-value* < 0.04, nível de significância de 0.05). Assim, refutamos a hipótese nula. Ou seja, para ambas as turmas, a maior parte dos estudantes consegue alcançar um índice admissível de similaridade com os especialistas, mesmo que em alguns casos essa similaridade não seja a ideal.

Apesar de haver diferenças pouco significativas, os alunos de BSI, em quantidade, apresentaram similaridade maior em relação aos alunos de LCC e também menor variação entre a similaridade dos alunos. Assim, concluímos que o grupo de alunos como um todo é capaz identificar uma quantidade significativa, mas não ideal, de problemas relacionados a qualidade de código similares aos especialistas. Porém, em ambas as turmas, um grupo de alunos consegue identificar com maior precisão esses problemas.

Q2-2: Quais problemas de qualidade de código são identificados mais frequentemente pelos alunos de Programação I de BSI e LCC da UFPB?

Primeiramente consideramos os principais problemas de qualidade de código analisados neste estudo (Tabela 4.4). Verificamos que a maior parte das dicas abordam problemas de complexidade, como mostra a Tabela 4.5, e que não há diferença significativa entre a quantidade de dicas sobre espaçamento e variáveis.

Tabela 4.5: Quantidade de dicas por aspecto de qualidade de código reportado em BSI e LCC.

Problema de qualidade do código-fonte	Quantas vezes foi reportado?
Complexidade	249
Espaçamento	101
Variável	92

Além dos problemas de qualidade de código já citados, outros aspectos mais específicos também foram identificados pelos alunos. Para entender melhor esse cenário, decidimos ranquear os problemas de qualidade identificados nas dicas, como mostra a Figura 4.7.

A maior quantidade de dicas foi sobre legibilidade e código duplicado. Nesta pesquisa, classificamos como legibilidade as dicas de boas práticas e melhoria da organização do código sem remover ou adicionar novas linhas e funcionalidades. Enquanto código duplicado, como o nome diz, definimos como trechos do código que se repetem, podendo ser reduzidos.

Observamos que dicas sobre a indentação do código não foram identificados pelos alunos, no caso, essas dicas foram elaboradas apenas pelos especialistas. O inverso disso ocorreu com as dicas sobre adicionar mensagens nas entradas dos programas e adicionar comentários ao código. O último foi citado 8 vezes, porém não é considerada uma boa prática. Além dessas, houveram dicas de alunos e especialistas sobre adicionar cabeçalho nos programas. Contudo, acreditamos que esse problema foi pouco mencionado devido os professores não cobrarem das turmas nas atividades.

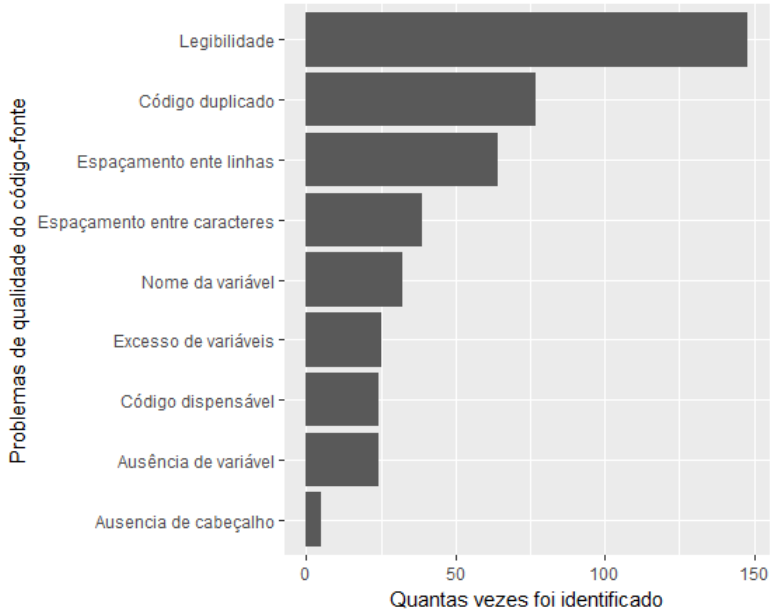

Figura 4.7: *Ranking* dos problemas de qualidade do código relatados nas dicas dos alunos do BSI e LCC.

Q3-2: O quão útil são as dicas dos alunos de Programação I do BSI e LCC da UFPB sobre problemas na qualidade do código?

Para responder esta questão de pesquisa, selecionamos todas as dicas elaboradas pelos alunos e apresentamos para que professores da disciplina as classificassem com útil ou inútil. Definimos como útil a dica correta, personalizada e suficientemente clara para melhorar o código-fonte a partir dela. Já como inútil, classificamos as dicas: (i) corretas, mas não claras o suficiente para melhorar o código-fonte a partir dela, (ii) irrelevantes para melhorar a qualidade do código ou (iii) incorretas. Para essas definições, consideramos, além da corretude, o detalhamento do *feedback* , já que esse é o diferencial de incluir o aluno nessa atividade. A Figura 4.6b apresenta a visão geral dos dados obtidos.

Verificamos que mais de 50% das dicas de 34, dos 50 alunos participantes, foram classificadas como úteis. Verificamos também que a maior parte dessas dicas eram sobre problemas de complexidade. Assim como na similaridade, não houve diferença significativa entre os resultados, porém, os alunos de LCC apresentaram mais dicas úteis. Em média, os estudantes deram 3 dicas por questão do *survey*, sendo pouco mais de 70% delas corretas, mesmo que não úteis. No entanto, os especialistas mencionaram que algumas dicas foram pouco detalhadas ou tinham problemas com a terminologia de programação.

Para complementar a resposta da Q3-2, buscamos verificar se a proporção de alunos que elaboram dicas úteis é significativamente maior. Consideramos admissível a similaridade maior que 0.5 (50% similar). Usando o teste de proporção, constatamos que 68% dos alunos tiveram suas dicas avaliadas como úteis (p-*value* $<$ 0.005, nível de significância de 0.05). Assim, refutamos a hipótese nula. Ou seja, a maior parte dos alunos do BSI e LCC consegue elaborar mais dicas úteis, mesmo que em alguns casos a porcentagem de dicas úteis não seja a ideal.

Os estudantes em sua maioria foram capazes de elaborar *feedback* útil. Assim, afirmamos que eles podem fornecer *feedback* sobre a qualidade do código de seus colegas. Com esse estudo de caso Confirmamos que os resultados são semelhantes mesmo em contextos diferentes, logo assumimos que é possível generalizar os resultados desse experimento para outros cursos.

4.3.5 Ameaças à Validade

No geral, elaboramos o *design* deste estudo como o intuito minimizar muitas das ameaças discutidas nesta seção. Para organizar esta seção, classificamos as ameaças à validade usando

as categorias: Interna, Externa, *Construct* e de Conclusão [WRH+12].

Interna: Como o estudo envolve participação humana ativa, ele inclina-se para ameaças internas. Uma vez que os alunos avaliaram muitas soluções, é possível que em algum momento estivessem cansados ou entediados para revisar os programas com precisão.

Externa: Os participantes desse estudo são representativos apenas do contexto da disciplina de Programação I dos cursos de BSI e LCC da UFPB e do semestre letivo em que esse estudo aconteceu. Contudo, ao verificarmos que os resultados são semelhantes aos do estudo de caso anterior, mesmo em contextos diferentes, assumimos que é possível generalizar os resultados desse experimento para outros cursos de Programação I.

***Construct*:** O modo como construimos a métrica do *feedback* referência pode ter influenciado negativamente no indice de similaridade entre as dicas dos alunos com as dicas dos especialistas. No mais, buscamos utilizar técnicas empiricamente validadas e comumente usadas nos estudos empíricos científicos da comunidade de Educação e Ciência da Computação.

Conclusão: A classificação das dicas como úteis ou inúteis ocorreu considerando exclusivamente a avaliação dos professores da disciplina de programação I do BCC/UFCG. No entanto, devido a experiência e conhecimento acerca do PEP 8, afirmamos que a classificação foi adequada.

4.3.6 Conclusões

Assim como no estudo realizado no BCC/UFCG, verificamos que a maioria dos estudantes elaborou dicas úteis e identificou problemas de qualidade de código com similaridade igual ou superior a 50% em comparação com os especialistas. Verificamos também que a maior quantidade de dicas dos estudantes foi sobre aspectos relacionados à complexidade dos programas.

Não houve diferença significativa entre os resultados do BSI e LCC. De modo geral, os estudantes conseguem identificar problemas de qualidade de código e elaborar boas dicas em um nível significativo, mesmo que não seja o ideal. Porém, ambos os cursos, um grupo de

alunos alcançou bons resultados. Desse modo, acreditamos que os alunos conseguem prover dicas sobre problemas na qualidade do código. Com esse estudo de caso confirmamos que os resultados são semelhantes mesmo em contextos diferentes, logo assumimos que é possível generalizar os resultados desse experimento para outros cursos. Contudo, defendemos que com a experiência em prover *feedback* e ele sendo fornecido por mais de um estudante tende a ser mais efetivo.

Concluímos então que, mesmo com adaptações no *design* do experimento, os resultados foram semelhantes aos obtidos no estudo anterior. Logo, é possível que os aspectos que observamos representem os alunos de Programação I de contextos diferentes. Nosso próximo passo é criar atividades práticas nas quais os alunos revisem colaborativamente os códigos uns dos outros, para que possamos analisar as características do *feedback* e seu impacto na da qualidade dos programas dos estudantes.

Capítulo 5

Quais as Características do *Feedback* Elaborado pelos Alunos?

A partir dos estudos descritos no capítulo anterior, percebemos que, de modo geral, os alunos conseguem elaborar boas dicas e identificar problemas de qualidade de código em um nível significativo. Entendendo isso, nosso próximo passo foi investigar as características dessas dicas quanto a corretude e clareza do texto. Para isso, realizamos uma atividade, baseada em revisão por pares, na disciplina de Programação I do BCC/UFCG onde os alunos programaram a solução para uma atividade, revisaram o código uns dos outros e modificaram seus códigos considerando as dicas que consideraram pertinentes. Neste capítulo descrevemos esse experimento.

5.1 *Design* do Estudo

Esse estudo consiste em uma atividade baseada em revisão de código por pares, na qual os estudantes resolveram uma atividade de Programação I, elaboraram dicas entre si sobre problemas identificados na qualidade do código e revisaram seus códigos baseados nas dicas que receberam. Essa atividade foi realizada presencialmente no horário da aula e teve duração de 90 minutos. As dicas e códigos dos alunos não tinham informações para identificação e a atividade selecionada é sobre os conteúdos abordados até a unidade 4 (Tabela 4.1). O objetivo foi analisar as dicas e identificar particularidades para assim responder à seguinte questão de pesquisa:

- **Q4**: Quais as características das dicas elaboradas pelos estudantes de programação introdutória?

5.1.1 *Peergrade*

Para facilitar a realização do experimento utilizamos o *Peergrade* [3] [WJH17], um sistema que dá suporte para avaliação por pares baseado na web. Para cada atividade que o professor publica nessa plataforma, os alunos seguem uma sequência de etapas para concluí-la. Primeiramente, os alunos submetem seus códigos. Em seguida, eles dão *feedback* anônimo sobre o código de outros estudantes cadastrados no curso, nesse estudo alocamos 2 revisores para cada código. Depois de fornecerem *feedback*, os estudantes reagem ao *feedback* recebido, nessa etapa é possível comentar, dar "*like*"caso concorde e dar "*flag*"se discordar. Por fim, o aluno pode consultar seus resultados (dicas elaboradas e recebidas) quando quiser. Na interface do professor é possível configurar os prazos da atividade, a quantidade de revisores por atividade, distribuição das notas e é mostrada uma visão geral (ou detalhada) do desempenho da turma na atividade.

5.1.2 Participantes

Participaram desse estudo 99 dos 108 alunos matriculados na disciplina de Programação I, do BCC/UFCG, no primeiro semestre letivo de 2018. Realizamos o experimento durante o horário de aula. Portanto, participou quem estava presente. Contudo, para verificar se a quantidade de alunos representa a turma estudada, realizamos o cálculo do tamanho amostral (1) e verificamos que nossa amostra representa a população do estudo com erro amostral (diferença entre número estimado e número real) de 5% e com 95% de confiança (probabilidade de que o erro amostral efetivo seja menor que o erro amostral admitido). A Tabela 5.1 mostra algumas informações básicas que podem nos ajudar a delinear o perfil da amostra.

[3] www.peergrade.io

Tabela 5.1: Dados demográficos da amostra de alunos de Programação I do BCC em 2018.1.

Total de alunos	99		
Gênero	**Masculino**	**Feminino**	
	88.9%	11.1%	
Idade	**<18**	**Entre 18 e 22**	**>22**
	21.9%	56.2%	21.9%
É a primeira vez que cursa Programação I?	**Sim**	**Não**	
	91%	9%	

Na Figura 5.1, apresentamos o tamanho da nossa amostra de alunos de Programação I do BCC por unidade no semestre 2018.1. Analisamos os alunos por unidade de estudo para classificamos o nível de conhecimento em programação e verificar se há relação com seu desempenho nesse experimento. Reforçando: as unidades são sequenciais entre 0 e 10, sendo que quanto mais próximo de 10 mais avançado o aluno está, e a unidade 11 são os alunos que já concluíram todas as unidades, mas ainda participam das aulas.

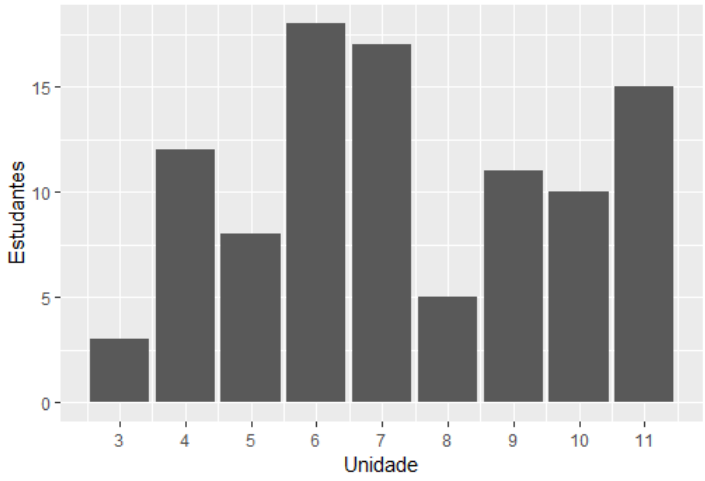

Figura 5.1: Número de alunos participantes por unidade em Programação I/BCC de 2018.1.

Questionamos sobre a frequência com que costumam ajudar outros estudantes em atividades acadêmicas e 51.6% afirmou colaborar sempre que pode, 41.1% colabora as vezes e 6.6% dificilmente colabora. A Figura 5.2 corresponde às respostas dos alunos referentes a quais fatores os motivam a colaborar com outros estudantes em atividades acadêmicas.

Nessa pergunta, os estudantes puderam selecionaram mais de uma alternativa.

Figura 5.2: Fatores que motivam a colaboração.

5.1.3 Métricas

Analisamos o *feedback* dos alunos por 2 aspectos: **corretude** e **clareza**. A corretude, assim como seu nome sugere, diz respeito ao quão corretas estão as dicas, de acordo com critérios definidos no PEP 8. Já a clareza, aborda características desejáveis para que a dica elaborada possa ser útil e efetiva. Para cada um desses aspectos a nota atribuída varia entre 0 e 5. No caso da avaliação da clareza da dica, dividimos a pontuação igualmente entre 4 características descritas na Tabela 5.2. Definidos os critérios de avaliação em conjunto com um professor de Programação I do BCC/UFCG considerando o que a literatura aponta como fundamental para o *feedback* ser efetivo. A avaliação foi feita por um especialista (aluno de pós-graduação em Ciência da computação) treinado pelo mesmo professor da disciplina.

Tabela 5.2: Critérios para avaliação da clareza da dica.

Característica	**O que é verificado**
Construtivo	É proposta uma alternativa para o problema identificado na qualidade do código-fonte?
Explicativo	É explicado o motivo pelo qual determinada prática é considerada um problema na qualidade do código-fonte?
Específico	A partir da dica é possível identificar exatamente onde está o problema na qualidade do código-fonte?
Linguagem cordial	A linguagem usada pelo revisor é cordial?

De modo complementar, aplicamos métricas de qualidade de software (complexidade ciclomática, linhas lógicas de código e PEP 8) nos códigos programados pelos alunos. O objetivo foi verificar se houve diferença significativa na qualidade dos códigos entre as versões submetidas antes e depois de receberem as dicas. Também avaliamos quantitativamente os dados que coletamos usando estatística descritiva. Realizamos algumas análises estatísticas simples usando correlação de *Spearman*, teste de hipótese e intervalo de confiança.

5.2 Resultados

Q4: Quais as características das dicas elaboradas pelos estudantes de Programação I do BCC/UFCG?

Nesta seção apresentaremos os resultados obtidos com a aplicação da prática de revisão por pares, descrita anteriormente. A Figura 5.3, mostra uma visão geral dos resultados obtidos.

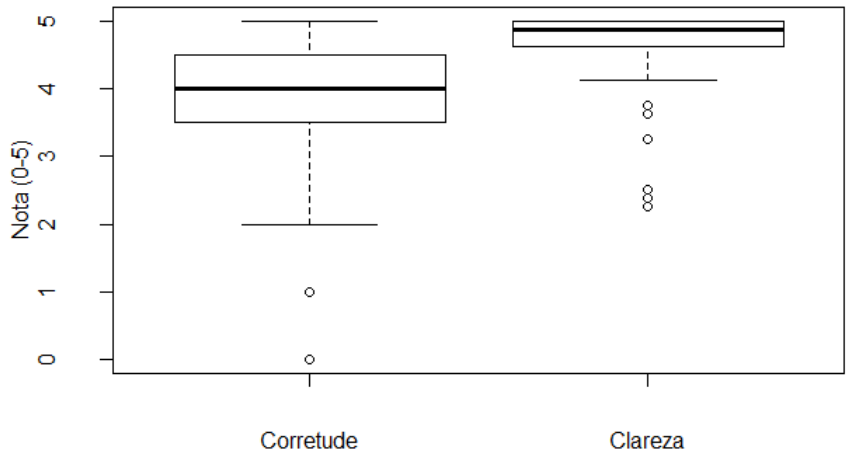

Figura 5.3: Avaliação das dicas.

Na Figura 5.3 observamos que, de modo geral, as dicas foram boas. Contudo, a nota média no aspecto clareza, da maior parte dos estudantes, foi excelente. Isto é, as dicas apresentam características importantes para um *feedback* efetivo, mesmo que parte dessas dicas não estejam corretas. Houveram casos de nota máxima tanto na avaliação da corretude, quanto na avaliação da clareza, mas no primeiro caso em menor quantidade. Diferente da corretude, que houveram 4 casos, não houve o pior caso (zero) na avaliação da clareza.

Dos 99 estudantes que elaboraram dicas, 10 tiveram notas menores que 3.5 (a pontuação mínima que consideramos para a dica ser considerada boa) no aspecto clareza, enquanto na corretude esse número foi 19. De modo complementar, verificamos se a proporção de alunos que elaboram boas dicas em relação à corretude e clareza é significativamente maior. Para isso, consideramos como casos de sucesso os alunos que alcançaram notas acima de 3.5 (equivalente a 70% boas). Usando o teste de proporção, constatamos que no contexto do BCC/UFCG mais de 77% dos alunos deram dicas corretas em um nível admissível (*p-value* < 1.622e-07, nível de significância de 0.05) e que 89% alunos deram dicas claras em um nível admissível (*p-value* < 1.013e-14 e nível de significância de 0.05). Assim, para ambas avaliações, refutamos a hipótese nula. Ou seja, para esse cenário, a maior parte dos estudantes consegue elaborar dicas corretas e claras. A Tabela 5.3, mostra informações adicionais sobre o desempenho dos alunos nas dicas.

Tabela 5.3: Informações gerais da avaliação das dicas.

	Corretude	**Clareza**
Valor mínimo	0	2.2
Mediana	4	4.8
Média	3.9	4.5
Valor máximo	5	5

Apresentamos na Figura 5.4 um exemplo de como realizamos a avaliação das dicas de acordo com os critérios definidos na Tabela 5.2.

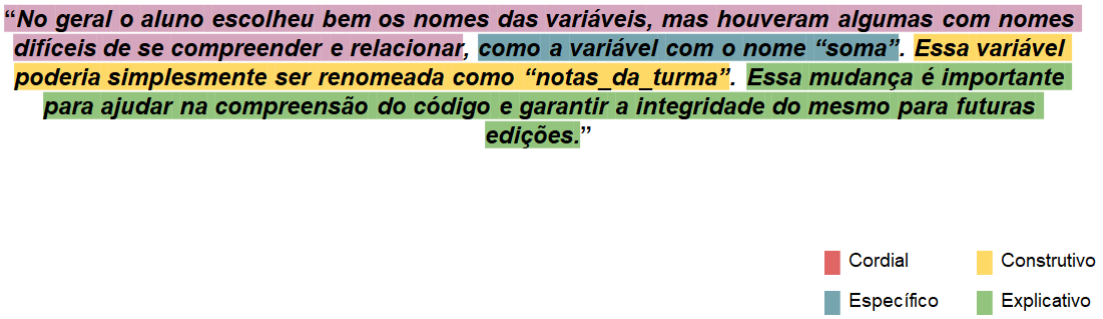

Figura 5.4: Exemplo de dica avaliada.

5.2.1 Clareza das Dicas

Como mencionamos, a avaliação da clareza das dicas foi definida a partir dos critérios detalhados na Tabela 5.2. Cada um desses critérios tem peso igual e, somados, formam a avaliação do aspecto de clareza das dicas elaboradas pelos alunos sobre problemas na qualidade do código. Assim, cada um dos critérios tem uma pontuação que varia entre 0 e 1.25. Esse método de pontuar foi proposto e aceito pelos professores da disciplina. A Figura 5.5 mostra os resultados da avaliação de cada um dos critérios.

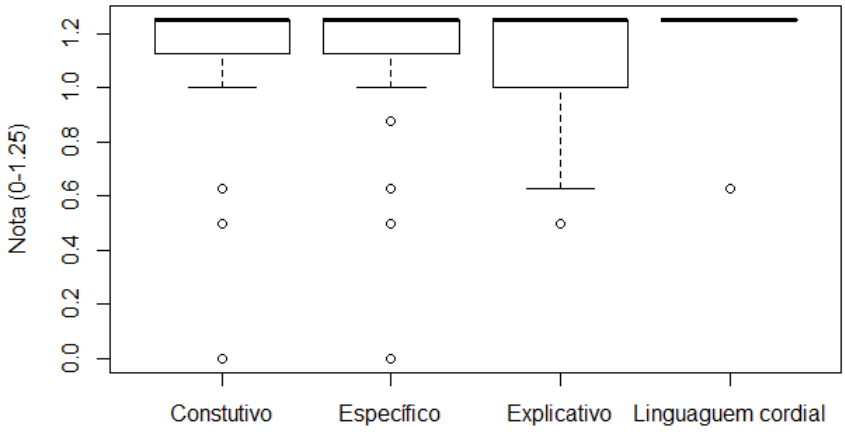

Figura 5.5: Clareza das dicas de Programação I do BCC.

Os estudantes tiveram o desempenho médio igual em todos os critérios avaliados na clareza do *feedback*. Observamos que eles são geralmente cordiais no texto das dicas que elaboram. No cenário em que este estudo ocorreu, apenas 8 estudantes não alcançaram a pontuação máxima nesse critério. Quanto às dicas serem construtivas e explicativas, os estudantes demonstraram habilidade semelhante para ambas. A característica que os estudantes apresentaram menos habilidade em relação às demais foi em ser explicativo. Essa foi a característica que mais houve variação e, apesar de não haver dicas sem explicação, algumas vezes elas foram pouco claras. Alguns fatores podem ter influenciado, como, o estudante com mais conhecimento em programação considerar óbvia a correção e por isso julgar desnecessário uma explicação mais detalhada. Além disso, a pouca experiência em elaborar dicas sobre a qualidade do código também pode influenciar, não apenas neste, mas em todos os outros critérios que destacamos nesta análise.

Com intuito de investigar mais detalhadamente as características do *feedback* dos estudantes, agrupamos pelo nível de conhecimento em programação (unidade em que eles estavam). Em seguida analisamos as características das dicas elaboradas de acordo com cada aspecto avaliado neste estudo (Tabela 5.2). Ilustramos na Figura 5.6 o desempenho dos alunos, por unidade, ao elaborar dicas de como melhorar a qualidade do código-fonte e mostramos na Tabela 5.4 exemplos de dicas dos alunos para cada critério avaliado no aspecto clareza.

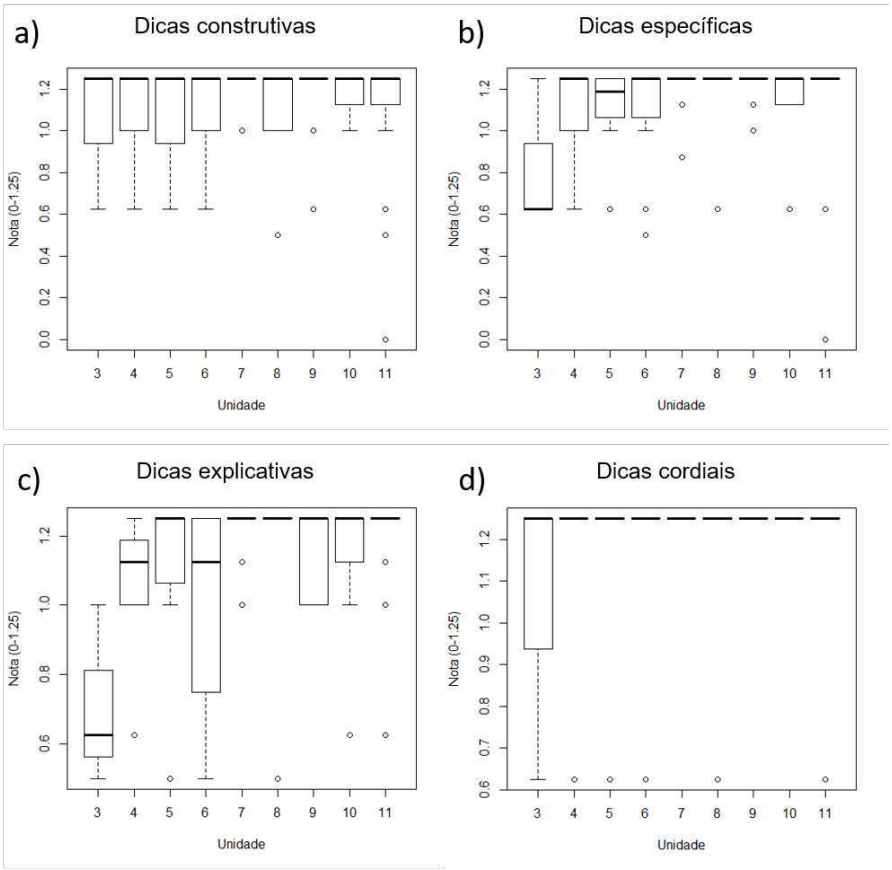

Figura 5.6: Clareza das dicas de Programação I do BCC por unidade.

Dicas construtivas: Para que o *feedback* possa ser útil é fundamental que ele seja construtivo. Isto é, o estudante deve, além de apontar onde é possível melhorar a qualidade do código, propor uma alternativa de como resolver o problema identificado. Como mencionado anteriormente, classificamos como unidade 11 os estudantes que já concluíram todas a unidades da disciplina. Esse foi o único grupo onde um aluno não pontuou. Os alunos das unidades 7, 9 e 10 apresentaram dicas mais explicativas, em comparação com as demais unidades. As outras unidades elaboraram dicas equivalentes nesse aspecto (Figura 5.6a).

Dicas específicas: Para melhorar a qualidade do código, a dica deve apontar com exatidão o problema a ser solucionado. Assim como nas dicas construtivas, a unidade 11 (alunos que concluíram todas as unidades da disciplina), foi a única em que um estudante não pontuou. Contudo, o desempenho geral da unidade foi melhor que no critério anterior. Os alunos da unidade 3 apresentaram dicas menos específicas, abordando de

modo mais genérico os problemas, mas sem definir exatamente onde estava o erro (por exemplo, *"a solução está muito complexa"*). Essa foi a única unidade que nenhum aluno atingiu a nota máxima. O desempenho médio entre a unidades foi semelhante. Os alunos das unidades 7 e 9 apresentaram as dicas mais específicas (Figura 5.6b).

Dicas explicativas: Para que o estudante compreenda que determinada prática é ruim para a qualidade do seu código é preciso justificar o motivo pelo qual ela não deve acontecer. De todos os critérios, esse é o que mais necessita que o aluno conheça padrões de codificação, no caso o PEP 8. Observamos que os alunos das unidade superiores a unidade 6 elaboraram melhores dicas nesse critério. Na unidade 3, os alunos apresentaram dicas, no geral, menos explicativas e a maior variação nos resultados foi na unidade 6. Todos os alunos pontuaram e a unidade 7 novamente apresentou dicas melhores (Figura 5.6c).

Dicas cordiais: Os estudantes como um todo foram gentis no *feedback* que elaboraram. Dentre os alunos, 8 não deram dicas cordiais. Esses alunos são de unidades diferentes da 7, 9 e 10. Nessas três unidades, pelo menos um aluno alcançou a nota máxima nesse aspecto (Figura 5.6d).

Tabela 5.4: Dicas dos alunos.

	Sim	Não
Dica construtiva	*"Na linha 14, há uma multiplicação dentro do range() que é confusa a primeira vista, pois não é mostrada o porquê da multiplicação. Isso pode ser melhorado com a criação de uma variável para a multiplicação, cujo nome explique seu significado."*	*"Condição na linha 22 e 23 confusa."*
Dica específica	*"Na linha 59 deveria ser utilizado o elif, pois não é necessário o uso do if nesse caso."*	*"Usar algo mais genérico pra resolver um problema simples"*
Dica explicativa	*"Na linha 13, existe um while dispensável que torna a compreensão mais dificil (já que não fica claro a primeira vista como aquele while vai interagir com o resto do programa, nem o número de loops planejados), podendo ser substituido com um 'for i in range()'".*	*"Código duplicado, dispensável, podendo simplificar."*

5.2.2 Corretude das Dicas

Avaliamos a corretude das dicas fundamentados no que é proposto no guia de estilo de código Python, o PEP 8. Logo, as dicas que não estavam de acordo, por exemplo, "*Atribuir todas as variáveis no começo do programa*" ou "*Poderia ter colocado alguns comentários para*

auxiliar no entendimento do código", foram consideramos incorretas. A corretude das dicas variou bastante, tanto entre alunos com mesmo nível de conhecimento em programação, quanto de níveis diferentes. Assim como nos critérios de corretude, a unidade 7 apresentou melhores dicas e menor variação. As unidades 8 e 9 tiveram variação semelhante, mas a avaliação média das dicas da unidade 9 foram melhores. Apresentamos na Figura 5.7 o desempenho dos alunos, por unidade, em relação a corretude das dicas que elaboraram.

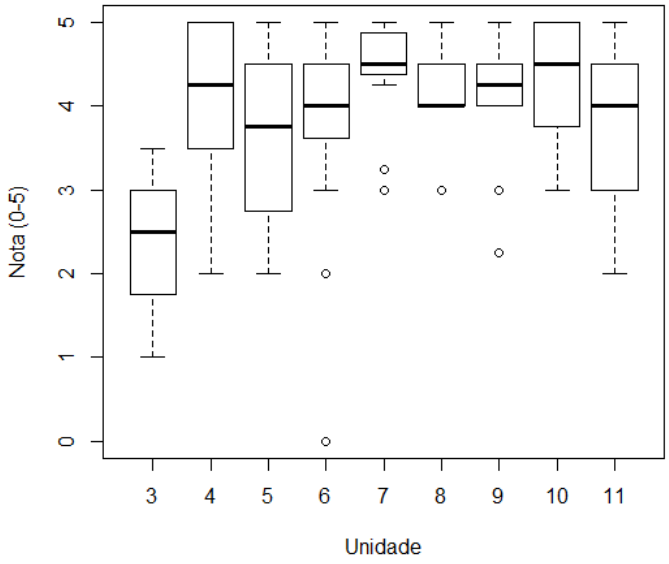

Figura 5.7: Corretude das dicas de Programação I do BCC por unidade.

No início deste estudo acreditávamos que o nível de conhecimento em programação tinha influência na corretude das dicas. Em alguns casos de fato isso aconteceu, mas as unidades 4, 7 e 11 apresentaram resultados que diferem do esperado, nos dois primeiros casos positivamente e no último negativamente. Acreditamos então que a motivação possa ser um fator de mesma, ou até maior, influência na qualidade das dicas dos alunos.

5.3 Discussão

***Feedback* construtivo:** Apontar um problema muitas vezes pode ser mais simples do que apresentar uma solução alternativa. Para propôr uma nova solução é necessário, além de conhecimento, criatividade. O esforço desprendido para essa tarefa, pode fazer com que avaliadores menos motivados não a incluam em seu *feedback*. A linguagem utilizada também pode dificultar a utilidade da dica. Adotar uma terminologia mais formal

prejudica o entendimento da dica por alunos com maiores dificuldades em programação. Esse é um dos fatores que podem ter influenciado na qualidade das dicas dos alunos que já concluíram a disciplina (unidade 11) (Figura 5.6a).

***Feedback* específico:** Para identificar um problema na qualidade do código, é importante conhecer sobre padrões de codificação. De acordo com a Figura 5.6b, percebemos que nas unidades abaixo da unidade 7 houve mais variação entre o quão específicas foram as dicas, diferente das outras unidades que atingiram a nota máxima quase em sua totalidade. O conhecimento em programação pode ter influenciado no resultado, como também a pouca prática em leitura crítica de código. Ler código e escrever código são atividades distintas e ambas têm sua importância [Try05].

***Feedback* explicativo:** A principal influência para os resultados desse critério foi a experiência. Muitos estudantes sabem que determinada prática não é correta, mas não sabem ou conseguem explicar o motivo. Parte disso deve-se ao fato de que alguns ainda não se depararam com códigos mais complexos, cujo problema ocorre e tem efeitos mais sérios. Todavia, a falta de investigação e criticidade também influem.

Feedback correto: A corretude das dicas variou entre os estudantes independente do nível de conhecimento em programação, isto é, houveram dicas corretas e incorretas em todas as unidades. Quanto às dicas incorretas, notamos que parte delas ocorrem devido aos estudantes, provavelmente, receberem a informação e, sem criticidade ou pesquisa, aceitarem como correta.

Analisando as dicas, percebemos que nesse critério alguns estudantes, principalmente das unidades iniciais, deram dicas que não seguiam o padrão proposto no PEP 8, ou seja, eles algumas vezes seguiam padrões provavelmente passados por monitores, video-aulas ou de outras formas fora da sala de aula. Percebemos nesse momento que é comum os alunos seguirem normas para melhorar a legibilidade de código que não são recomendadas na literatura. Percebemos também que o conhecimento em programação não influencia tanto quanto esperávamos na corretude das dicas. Estudantes que já concluíram todas as unidades da disciplina não deram as dicas mais corretas, acreditamos que a motivação em ajudar pode interferir nisso, pois as unidades 4 e 7 foram as que deram as melhores dicas nesse aspecto.

5.3.1 Houve Melhora nos Códigos com a Revisão por Pares?

Analisamos os códigos programados pelos estudantes a partir da aplicação das métricas de complexidade ciclomática (cc), linhas lógicas de código (lloc) e PEP 8. Em suma, estas medidas significam:

- **cc**: Métrica de qualidade de software, concebido por McCabe [McC76], usada para indicar a quantidade de caminhos de execução linearmente independentes de um programa. Cada decisão em um programa pode levar a um caminho diferente. Logo, são consideradas não apenas estruturas condicionais, mas também estruturas iterativas.

- **lloc**: Métrica de qualidade de software usada para indicar o número de linhas usadas como instruções de código-fonte. Essa medida não considera linhas em branco e comentários.

- **PEP 8**: Guia de estilo para o código Python que compreende a biblioteca padrão na distribuição principal do Python [VRWC01].

A pontuação que o código atinge em cada um dessas métricas de qualidade de software representa a intensidade do problema. Isto é, quanto maior a nota, mais problemático é o código. Para cada padrão, calculamos a diferença (Δ) entre a versão inicial (V1) do código do aluno e a versão final (V2) revisada considerando as dicas que ele recebeu (Δ = V1-V2). Logo, quando:

- $\Delta > 0$, significa que o código-fonte melhorou na métrica verificada;

- $\Delta = 0$, significa que foi indiferente na métrica verificada;

- $\Delta < 0$, significa que o código piorou na métrica verificada.

Fizemos reamostragem com *bootstrap* 2000 vezes do Δ médio e conseguimos o intervalo de confiança para cada métrica, como mostra a Figura 5.8.

Em todos os casos o Δ médio entre as versões submetidas pelos estudantes foi positiva ($\Delta > 0$, onde cc = 9.42, lloc = 39.33 e PEP 8 = 13.05). A quantidade de linhas lógicas de código foi o aspecto que mais melhorou nos programas com a revisão por pares. Todavia, apesar do Δ médio maior que 0, o intervalo de confiança (Figura 5.8) aponta que foram mudanças

pouco consideráveis e não temos dados suficientes para afirmar algo mais concreto sobre o impacto das dicas na melhoria da qualidade dos códigos, apenas que há indícios que o *feedback* pode trazer melhorias.

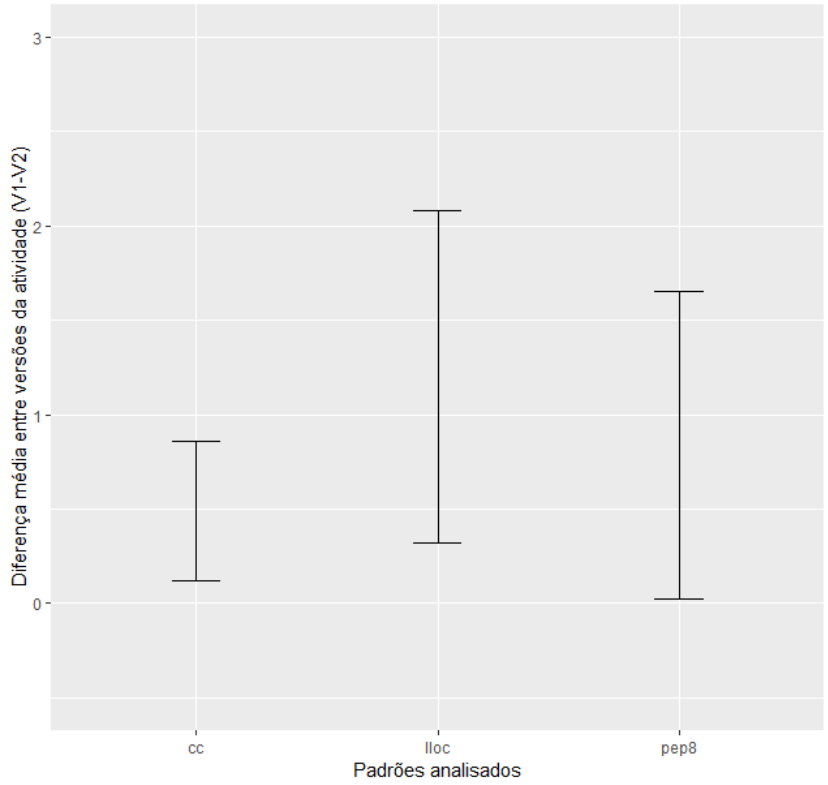

Figura 5.8: Intervalo da diferença média entre versões do código.

5.3.2 Opinião dos Estudantes

Após realizarmos a atividade de revisão, aplicamos um *survey* para saber a opinião os estudantes sobre a avaliação por pares e o quão importante o *feedback* foi para melhorar seus códigos, observamos que 63% dos estudantes consideraram muito importante a atividade, como mostra a Figura 5.9. Eles responderam essa questão em escala Likert. Observamos também que aproximadamente 63% dos estudantes consideraram as dicas que receberam como úteis ou muito úteis, aproximadamente 31% consideraram as dicas recebidas como indiferente ou pouco úteis e 5% considerou as dicas inúteis.

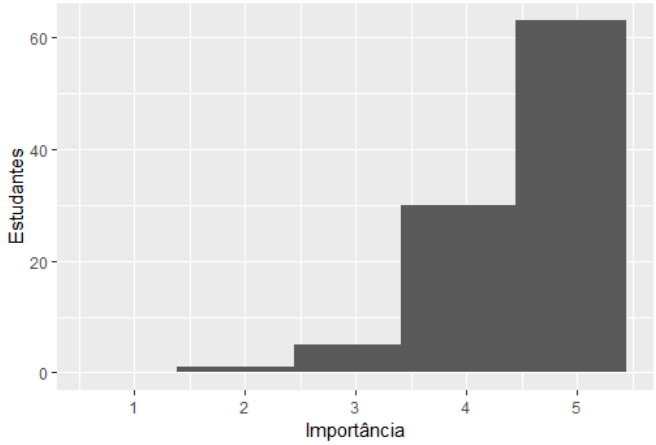

Figura 5.9: Importância do *feedback* sobre a qualidade do código.

Investigando a razão pela qual alguns alunos consideraram as dicas inúteis ou pouco úteis, descobrimos que, além de considerarem algumas dicas não pertinentes, a quantidade de dicas recebidas e o código não precisar de melhorias influenciaram na avaliação que eles deram. Quanto a importância das dicas para a aprendizagem, 33.7% dos estudantes considerou muito importante, 55,1% considerou importante e 11.2% não considerou importante para a aprendizagem.

A opinião dos estudantes sobre a avaliação por pares no geral foi positiva. Tivemos como retorno opiniões como: *"Foram úteis principalmente para simplificação e objetividade do meu código."*, *"Pude ver que ainda cometo alguns erros básicos na hora de organizar o código"* e *"Gostei muito. No começo não consegui fazer a questão, mas consegui terminar a questão com uma das dicas recebidas."*. Embora tenham se sentido ajudados pelas dicas, alguns alunos não concordaram com todas as que receberam: *"Não concordo totalmente, mas alguns comentários foram bastante úteis."* e *"Foi útil em geral, embora tiveram certas opiniões que são mais envolvidas com o estilo de programação da pessoa em sí."*.

Além disso, a resistência em aceitar o *feedback* foi um problema bastante reportado: *"Foi legal, mas as pessoas às vezes precisam absorver as dicas"* e *"De forma geral foi boa, mas em alguns casos não aceitaram minhas sugestões, nem mesmo houve intenção de discutir."*. Parte dessa resistência é devido os alunos estarem em unidades diferentes. De acordo com um dos alunos, *"O nível diferente de conhecimento entre os alunos prejudicou os feedbacks."*. Contudo, outro estudante reportou que *"[...] o curso de Ciência da Computação tem como ponto forte o trabalho em equipe e, para isso, críticas boas ou ruins são sempre*

bem vindas".

Ao todo houveram 471 *likes* (dicas que os estudantes consideraram úteis) e 108 *flags* (dicas consideradas inúteis). Nem todos os alunos usaram essas funcionalidades, mas alguns usaram bastante por não concordar com o *feedback* recebido, que muitas vezes estava correto. Percebemos que há uma certa resistência dos estudantes aceitarem dicas e conversando com alguns deles após o experimento, nos foi reportado que a resistência é motivada pela confiança no *feedback* de alunos que não estejam na mesma unidade que eles, como também por não compreender a dica recebida. Nesse último caso, a terminologia de programação pode ser menos abstrata para uns do que para outros.

Boa parte dos estudantes sugeriu a inclusão da revisão por pares como prática recorrente na disciplina. Sugeriram também que a avaliação seja realizada entre estudantes da mesma unidade, que haja previamente uma aula para esclarecer algumas dúvidas sobre padrões de codificação e que as dicas também sejam avaliadas por monitores ou professores da disciplina.

5.3.3 Ameaças à Validade

Para organizar esta seção, classificamos as ameaças à validade usando as categorias: Interna, Externa e de Conclusão [WRH+12].

Interna: Como o estudo envolve participação humana ativa, ele inclina-se para ameaças internas. Uma vez que os alunos não tem conhecimentos sólidos sobre a qualidade do código e experiência em revisar códigos, é possível que em algum momento estivessem desmotivados para revisar os programas corretamente.

Externa: Os participantes desse estudo são representativos do contexto da disciplina de Programação I dos cursos de BCC da UFCG e do semestre letivo em que o estudo aconteceu. Contudo, ao verificarmos que outros resultados desse estudo são semelhantes mesmo em contextos diferentes, assumimos que é possível generalizar os resultados desse experimento para outros contextos.

Conclusão: As preocupações relacionadas a essa categoria dizem respeito à conclusão do experimento. A avaliação das dicas foi realizada por um aluno da pós-graduação em

Ciência da Computação. No entanto, ele pesquisa e tem experiência no ensino de programação introdutória e foi treinado por um dos professores da disciplina, além disso, devido a experiência e conhecimento acerca do PEP 8, afirmamos que a avaliação foi adequada.

5.4 Conclusões

Verificamos que as dicas elaboradas pelos alunos do BCC/UFCG possuem linguagem cordial. Eles conseguem ser específicos e propor alternativas para os problemas identificados em um nível significativo. Quanto a serem explicativos, houve maior variação nos resultados, no geral os alunos com mais conhecimento em programação conseguiram ter desempenho melhor nesse aspecto.

Verificamos também que o conhecimento em programação influência na corretude das dicas que o aluno elabora. Contudo, acreditamos que a motivação tenha peso igual, pois os melhores resultados desse aspecto não foram dos alunos que já concluíram as unidades da disciplina. A unidade 7, foi a que apresentou melhores dicas, tanto em corretude quanto em clareza. Por fim, aplicando métricas de qualidade de software (complexidade ciclomática, linhas lógicas de código e PEP 8) entre as versões dos códigos dos alunos, verificamos que houve indícios de melhora.

Além das dicas recebidas, os alunos também melhoraram seus códigos baseados nos códigos que avaliavam de seus colegas. Vale destacar que as dicas podem ajudar, mas promover mudanças na aprendizagem tem relação com a forma que o estudante reage a ela. O processo de construção do conhecimento é complexo e não é o foco deste trabalho. Contudo, é importante realizar estudos para investigar o processo de aprendizagem a partir do *feedback* colaborativo, pois esse tipo de metodologia tem um importante papel no ensino de Computação, não apenas para otimizar o processo de obter *feedback*, mas também para desenvolver aprendizes reflexivos.

Capítulo 6

Considerações Finais

Como mencionamos, os estudos no campo de *feedback* automatizado para exercícios de programação abordam principalmente aspectos funcionais. Embora haja ferramentas que analisam a qualidade do código, seu *feedback* pode não ser suficientemente detalhado e, para ser efetivo, precisar do trabalho manual do professor. Além disso, pesquisadores têm utilizado e defendido a inclusão dos alunos na tarefa de revisão, tanto para facilitar a obtenção do *feedback*, quanto pelos benefícios na aprendizagem do revisor e do revisado. Portanto, investigamos se, ao incluir estudantes como avaliadores, poderíamos fornecer *feedback* personalizado. Assim, neste estudo, a questão principal foi se os alunos podem avaliar qualitativamente os programas de seus pares.

Primeiramente investigamos se os alunos conseguem identificar problemas na qualidade do código. Para isso, comparamos as dicas dos alunos do BCC do semestre 2017.1 com as elaboradas pelos professores da disciplina calculando o coeficiente de similaridade de Jaccard. A escolha desse modelo foi motivada pelos trabalhos relacionados que também o utilizam, seja para correção ou até verificação de plágio em atividades de programação. Descobrimos que os estudantes conseguem identificar problemas de qualidade de código-fonte em um nível significativo, mesmo que não ideal (similaridade maior ou igual 50%).

Em seguida, investigamos se os alunos conseguem elaborar dicas úteis. Apresentamos as dicas para que os professores avaliassem como úteis ou inúteis. Neste estudo, definimos a dica como útil quando é correta, personalizada e clara o suficiente para melhorar o código a partir dela. Definimos como inútil a dica (i) correta, mas não claras o suficiente para melhorar o código, (ii) irrelevante para melhorar o código ou (iii) incorreta. Descobrimos que

os estudantes também conseguem em maior quantidade elaborar dicas úteis em um nível admissível e que são particularmente hábeis em encontrar e elaborar *feedback* sobre problemas relacionados à complexidade dos programas. Replicamos estes estudo nos cursos de BSI e LCC do semestre 2017.2 e obtivemos resultados semelhantes, o que corrobora com o que observamos.

Também buscamos entender as características do *feedback* dos alunos. Realizamos uma atividade de revisão por pares onde os estudantes resolveram uma atividade da disciplina de Programação I, revisaram os programas de outros alunos, reagiram ao *feedback* recebido e modificaram seu código baseados no *feedback* que recebeu. As dicas foram avaliadas nos aspectos de corretude e clareza. Na corretude consideramos as recomendações do PEP 8 e na clareza consideramos se foi construtivo, específico, explicativo e cordial, de acordo com a definição descrita neste trabalho.

Verificamos que as dicas elaboradas pelos alunos do BCC/UFCG do semestre 2018.1 possuem linguagem cordial. Eles conseguem ser específicos e propor alternativas para os problemas identificados em um nível significante. Quanto a serem explicativos, houve maior variação nos resultados, no geral os alunos com mais conhecimento em programação conseguiram ter desempenho melhor nesse aspecto. Verificamos que o conhecimento do aluno influencia no seu resultado, mas observamos que a experiência em prover *feedback* e a motivação também são fatores de impacto no desempenho, pois os melhores resultados desse aspecto não foram dos alunos que já concluíram a disciplina. Por fim, aplicando métricas de qualidade de software (complexidade ciclomática, linhas lógicas de código e PEP 8) entre as versões dos códigos dos alunos, verificamos que houve indícios de melhora, mas não obtivemos dados suficientes para afirmar estatisticamente que foi significativa essa melhora.

Com base nos resultados obtidos, concluímos que os estudantes conseguem, em um nível significativo elaborar *feedback* sobre a qualidade do código-fonte de seus pares e que suas dicas também tem potencial para melhorar a qualidade do código de outros estudantes. Também observamos que um grupo de alunos, independente do curso ou nível de conhecimento em programação, alcançou bons resultados e por isso acreditamos que o estudante pode ser incluído como revisor. Porém, para isso é primordial abordar mais profundamente o conteúdo de qualidade de código, realizar atividades de treinamento e abordar aspectos de motivação para que as revisões realizadas pelos estudantes sejam mais efetivas.

Vale destacar também que, além das dicas, alguns alunos se basearam nos códigos que avaliaram para melhorar o próprio código. Assim, as dicas podem direcionar a correção, mas causar mudanças na aprendizagem está relacionado a forma como o estudante aprende, Portanto, cabe a realização de estudos futuros mais aprofundados nesse sentido.

6.1 Discussão

Verificamos que os alunos alcançam um nível significativo de similaridade com as dicas elaboradas pelos professores, principalmente os mais avançados na disciplina. Esse resultado já era esperado, pois acreditamos que o conhecimento do estudante em programação influencia diretamente na qualidade de seu *feedback*. Porém, observamos que alguns alunos, que não avançaram tanto na disciplina, também apresentaram bons resultados. Assim, concluímos que, além do conhecimento, outros fatores como motivação em colaborar e experiência em elaborar *feedback* qualitativo podem influenciar no resultado. Identificar e tratar tais fatores pode ser uma ramificação desta pesquisa.

Ao analisarmos o *feedback* dos estudantes, também observamos que houveram sugestões, como: adicionar comentários e inserir informações extras no cabeçalho, para melhoria da qualidade dos códigos. No primeiro caso, essa sugestão é considerada errada e no segundo é irrelevante para a qualidade. Dicas irrelevantes, e principalmente incorretas, são os principais desafios de incluir alunos como revisores. Além disso, os professores mencionaram que algumas dicas foram mal detalhadas ou tiveram problemas com a terminologia de programação. Porém, mesmo havendo dicas assim, elas são em menor quantidade em relação as dicas corretas. Defendemos que, se o estudante tiver seu código revisado por mais de um colega, é possível minimizar essa limitação, além disso, realizar treinamentos préviamente é uma alternativa para nivelar o conhecimento das turmas sobre qualidade de código.

Quanto a classificação das dicas como úteis ou não, notamos que há mais estudantes com *feedback* útil do que similar ao dos professores. Isso ocorreu principalmente devido a forma que analisamos a similaridade (consideramos apenas a dica elaborada por mais de um professor). Observamos que os alunos conseguem, textualmente, reportar um problema de qualidade e propor alternativas para solucioná-lo, ou seja, o *feedback* dos estudantes apresentam características que, de acordo com a literatura, são importantes para compor um

feedback efetivo. Assim como na etapa anterior de medir a similaridade, o conhecimento em programação influenciou resultados dos alunos, como esperado, mas, de modo geral, independente do curso ou desempenho na disciplina, eles apresentaram bons resultados. Isso reforça que outros fatores, além do conhecimento, influencia na qualidade do *feedback* do aluno.

Entendendo que os estudantes conseguem prover *feedback* sobre a qualidade de código, realizamos uma atividade prática, baseada na revisão de código por pares, na qual eles elaboraram dicas para melhorar a qualidade do código de seus colegas. Selecionamos aleatoriamente os revisores e alocamos dois para cada código com intuito de garantir um *feedback* potencialmente mais efetivo.

Nesse segundo momento, analisamos as dicas por dois aspectos, corretude e clareza. Em relação a corretude, os alunos, independente do conhecimento em programação, elaboraram dicas corretas e incorretas. Parte das dicas incorretas ocorreram devido aos estudantes, provavelmente, receberem sugestões de boas práticas e, sem criticidade ou pesquisa, aceitarem como verdadeira. Alguns estudantes, principalmente das unidades iniciais, deram dicas que não seguiam o padrão proposto no PEP 8, ou seja, eles algumas vezes seguiam padrões talvez passados por monitores, video-aulas ou de outras formas fora da sala de aula. Percebemos que é comum os alunos seguirem normas para melhorar a legibilidade de código que não são recomendadas na engenharia de software. Percebemos também que o conhecimento do aluno não influencia tanto quanto esperávamos na corretude das dicas, pois os estudantes que já concluíram todas as unidades não elaboraram as dicas mais corretas. Acreditamos que a motivação em ajudar pode interferir nisso, pois alunos menos avançados na disciplina elaboraram as melhores dicas nesse aspecto.

Quanto a clareza, o conhecimento sobre qualidade de código e a pouca prática em leitura crítica de código foram os maiores desafios, pois esses aspectos são fundamentais para identificar problemas na qualidade do código, apresentar uma solução alternativa e até explicar o motivo de determinada prática não ser recomendada. Verificamos que, mais uma vez, estudantes mais avançados na disciplina apresentaram melhores resultados, principalmente quanto a serem explicativos. Contudo, de modo geral, os alunos conseguem ser construtivos e específicos em suas dicas. Isto é, mesmo não estando correta, a dica tem características fundamentais para ser efetiva e por isso vale incluir os estudantes como revisores, desde que

seja trabalhado previamente a qualidade de código.

Verificamos que parte alunos conseguiu melhorar seus códigos observando a forma como seus colegas programaram os deles ao invés das dicas recebidas. Verificamos também que o conhecimento do aluno influencia consideravelmente na qualidade das dicas que ele elabora, mas a experiência em prover *feedback* e a motivação em querer colaborar também são fatores de impacto no desempenho. Por fim, recomendamos replicar a prática de incluir os alunos como revisores em outras disciplinas de programação, além da introdutória. Essa prática pode ser ainda mais eficiente e necessária para códigos de alunos com maior nível de complexidade.

6.2 Desafios

O sistema que utilizamos para realizar a atividade de revisão por pares dá suporte a interação entre o aluno e o *feedback* que ele recebeu. Essa funcionalidade se mostrou muito positiva, mas foi reportado, tanto pelos alunos quanto pelos instrutores auxiliares, que houve uma certa resistência por parte dos alunos em aceitar o *feedback* recebido. Ao questioná-los após a atividade, nos foi passado que, como eles não sabem quem está revisando seu código, o *feedback* pode ser incorreto.

Percebemos que a falta de conhecimento sobre o que é ou não recomendado em relação a legibilidade do código-fonte reforça a insegurança do *feedback* recebido. Uma alternativa para aumentar a confiabilidade do *feedback* é classificar os estudantes de acordo com o conhecimento em programação e desempenho nas dicas, de modo que alunos recebam dicas apenas de outros alunos do mesmo nível de conhecimento ou superior. Além disso, vale estimular o debate suportado pela ferramenta.

Observamos também que algumas dicas incorretas eram aceitas pelos estudantes. Mesmo não sendo uma quantidade considerável, é um problema que pode escalonar e ter efeito contrário ao que esperamos quanto a qualidade dos códigos. Por isso é importante que, antes da atividade de revisão, haja um treinamento com os estudantes para nivelar o conhecimento da turma sobre qualidade de código. Também é importante estimular a pesquisa e a criticidade para que não ocorram novamente casos de dicas incorretas serem tomadas como corretas, devido a quantidade de vezes que ela é mencionada.

Outro desafio que enfrentamos foi quanto a motivação dos alunos. No início todos estavam estimulados por ser uma atividade diferente das aplicadas na disciplina, no entanto, alguns estudantes foram perdendo o interesse ao longo da atividade, o que influenciou no desempenho final deles. Por isso é importante investigar e inserir técnicas para motivar os alunos a participarem e permaneceram na atividade de revisão da qualidade de código, assim como mostrar a importância da cooperação no ambiente de desenvolvimento de software para estimulá-los desde cedo a colaborarem. Por fim, observamos que a pouca experiência em prover *feedback* também foi um fator de grande impacto no desempenho e motivação dos alunos. Defendemos que com a prática constante os alunos podem melhorar a qualidade de suas dicas e consequentemente de seus códigos.

6.3 Trabalhos Futuros

Existem diversas oportunidades para trabalhos futuros que surgem desta pesquisa de mestrado. Listamos a seguir essas sugestões de melhorias ou trabalhos futuros.

- Verificar, a partir de estudos longitudinais, os efeitos do *feedback* qualitativo nas soluções submetidas;
- Realizar estudos qualitativos usando as dicas recebidas e submissões de códigos de alunos que não conseguiram melhorar a qualidade do código com o *feedback*;
- Investigar sobre como as percepções de *feedback* dos alunos influenciam seu envolvimento com o processo de fornecer *feedback*;
- Utilizar e investigar os efeitos de metodologias para motivar o engajamento com o processo de fornecer *feedback* qualitativo;
- Investigar relações em dados coletados a partir de estudos longitudinais, numa análise exploratória, contrastando o perfil dos alunos, o *feedback* fornecido e o desempenho pós-*feedback* nas atividades da disciplina.
- Replicar este estudo em disciplinas mais avançadas de programação.

Bibliografia

[AJHA+15] Dhiya Al-Jumeily, Abir Hussain, Mohammed Alghamdi, Chelsea Dobbins, and Jan Lunn. Educational crowdsourcing to support the learning of computer programming. *Research and Practice in Technology Enhanced Learning*, 10(1):13, 2015.

[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.

[Ave14] Jill Avery. Leveraging crowdsourced peer-to-peer assessments to enhance the case method of learning. *Journal for Advancement of Marketing Education*, 22(1), 2014.

[B+71a] Benjamin S Bloom et al. Handbook on formative and summative evaluation of student learning. 1971.

[B+71b] Benjamin S Bloom et al. Handbook on formative and summative evaluation of student learning. 1971.

[BCCC15] Alexandre Barbosa, Allan Correia, D Costa, and Evandro Costa. Um mapeamento sistemático sobre analisadores de código em disciplinas de programação. In *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*, volume 26, page 1235, 2015.

[BDKKM91] Robert L Bangert-Drowns, Chen-Lin C Kulik, James A Kulik, and MaryTeresa Morgan. The instructional effect of feedback in test-like events. *Review of educational research*, 61(2):213–238, 1991.

[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.

[BK15] S. Brinkman and S. Kvale. Interviews: Learning the craft of qualitative research interviewing. *Aalborg*, 24:2017, 2015.

[Blo68] Benjamin S Bloom. Learning for mastery. instruction and curriculum. regional education laboratory for the carolinas and virginia, topical papers and reprints, number 1. *Evaluation comment*, 1(2):n2, 1968.

[BMSB14] Marcos C Borges, Carlos H Miranda, Rodrigo C Santana, and Valdes R Bollela. Avaliação formativa e feedback como ferramenta de aprendizado na formação de profissionais da saúde. *Medicina (Ribeirao Preto. Online)*, 47(3):324–331, 2014.

[Cen94] John A Centra. The use of the teaching portfolio and student evaluations for summative evaluation. *The Journal of Higher Education*, 65(5):555–570, 1994.

[CG08] Cíntia Camargo Furquim Caseiro and Raimunda Abou Gebran. Avaliação formativa: concepção, práticas e dificuldades. *Nuances: estudos sobre Educação*, 15(16), 2008.

[CSBS03] Fernanda CA Campos, Flávia Maria Santoro, Marcos RS Borges, and Neide Santos. Cooperação e aprendizagem on-line. *Rio de janeiro: DP&A*, 168:21, 2003.

[Dil99] Pierre Dillenbourg. What do you mean by collaborative learning?, 1999.

[dJGM] Anabela de Jesus Gomes and Antônio José Mendes. *Ambiente de suporte à aprendizagem de conceitos básicos de programação*.

[DMJ+13] Willem Doise, Gabriel Mugny, A St James, Nicholas Emler, and D Mackie. *The social development of the intellect*, volume 10. Elsevier, 2013.

[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.

[Fox87] Barbara A Fox. Interactional reconstruction in real-time language processing. *Cognitive Science*, 11(3):365–387, 1987.

[Gil89] Michel Gilly. The psychosocial mechanisms of cognitive constructions: Experimental research and teaching perspectives. *International Journal of Educational Research*, 13(6):607–621, 1989.

[GLCM16] Elena L Glassman, Aaron Lin, Carrie J Cai, and Robert C Miller. Learner-sourcing personalized hints. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*, pages 1626–1636. ACM, 2016.

[GPL16] Jianxiong Gao, Bei Pang, and Steven S Lumetta. Automated feedback framework for introductory programming courses. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 53–58. ACM, 2016.

[GT90] Ronald Gallimore and Roland Tharp. Teaching mind in society: Teaching, schooling, and literate discourse. *Vygotsky and education: Instructional implications and applications of sociohistorical psychology*, pages 175–205, 1990.

[Gul10] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, pages 13–24, New York, NY, USA, 2010. ACM.

[HAT11] Christopher D Hundhausen, Pawan Agarwal, and Michael Trevisan. Online vs. face-to-face pedagogical code reviews: an empirical comparison. In *Pro-*

ceedings of the 42nd ACM technical symposium on Computer science education, pages 117–122. ACM, 2011.

[Hen09] E Hendry. Duke professor uses 'crowdsourcing'to grade. *The Chronicle of Higher Education*, 30, 2009.

[HFCM94] Charles Hadji, Júlia Lopes Ferreira, José Cláudio, and Philippe Meirieu. *A avaliação, regras do jogo: das intenções aos instrumentos*. 1994.

[HGS+17] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*, pages 89–98. ACM, 2017.

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

[JCH13] David S. Janzen, John Clements, and Michael Hilton. An evaluation of interactive test-driven labs with webide in cs0. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1090–1098, Piscataway, NJ, USA, 2013. IEEE Press.

[JGB05] Mike Joy, Nathan Griffiths, and Russell Boyatt. The boss online submission and assessment system. *Journal on Educational Resources in Computing (JERIC)*, 5(3):2, 2005.

[KHJ17] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. Code quality issues in student programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '17. ACM, 2017.

[KKL+16] Dohyeong Kim, Yonghwi Kwon, Peng Liu, I. Luk Kim, David Mitchel Perry, Xiangyu Zhang, and Gustavo Rodriguez-Rivera. Apex: Automatic programming assignment error explanation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 311–327, New York, NY, USA, 2016. ACM.

[KS89] Raymond W Kulhavy and William A Stock. Feedback in written instruction: The place of response certitude. *Educational Psychology Review*, 1(4):279–308, 1989.

[KS99] Vinícius Medina Kern and Luciana Martins Saraiva. Aplicação da revisão pelos pares no ensino de graduação. *Alcance, Itajaí, ano VI*, (3):42–49, 1999.

[KWL+15] Chinmay Kulkarni, Koh Pang Wei, Huy Le, Daniel Chia, Kathryn Papadopoulos, Justin Cheng, Daphne Koller, and Scott R Klemmer. Peer and self assessment in massive online classes. In *Design thinking research*, pages 131–168. Springer, 2015.

[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.

[LPTA05] Cristiane Luiza Köb Leite, MO de A PASSOS, Patrícia Lupion Torres, and Paulo Roberto Alcântara. A aprendizagem colaborativa na educação a distância on-line. In *Congresso Internacional de Educação a Distância*, volume 12, 2005.

[LUC98] Cipriano Carlos LUCKESI. Avaliação do aprendizado escolar: estudos e proposições, 1998.

[MB01] B Jean Mason and Roger Bruning. Providing feedback in computer-based instruction: What the research tells us. *Retrieved February*, 15:2007, 2001.

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

[Miz86] Maria da Graça Nicoletti Mizukami. *Ensino: as abordagens do processo*. Editora Pedagógica e Universitária São Paulo, 1986.

[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.

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

[Ros92] Jeremy Roschelle. Learning by collaborating: Convergent conceptual change. *The journal of the learning sciences*, 2(3):235–276, 1992.

[RSD+17] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 404–415, Piscataway, NJ, USA, 2017. IEEE Press.

[SBS14] Martijn Stegeman, Erik Barendsen, and Sjaak Smetsers. Towards an empirically validated model for assessment of code quality. In *Proceedings of the 14th Koli Calling international conference on computing education research*, pages 99–108. ACM, 2014.

[Sch95] Daniel L Schwartz. The emergence of abstract representations in dyad problem solving. *The journal of the learning sciences*, 4(3):321–354, 1995.

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

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

[SP89] N Schunk and K Petermann. Measured feedback-induced intensity noise for 1.3 μm dfb laser diodes. *Electronics Letters*, 25(1):63–64, 1989.

[SSC01] Yi Shang, Hongchi Shi, and Su-Shing Chen. An intelligent distributed environment for active learning. *Journal on Educational Resources in Computing (JERIC)*, 1(2es):4, 2001.

[Tar05] Maddalena Taras. Assessment–summative and formative–some theoretical reflections. *British journal of educational studies*, 53(4):466–478, 2005.

[Tes13] Martin Tessmer. *Planning and conducting formative evaluations*. Routledge, 2013.

[Try05] Deborah A Trytten. A design for team peer code review. In *ACM SIGCSE Bulletin*, volume 37, pages 455–459. ACM, 2005.

[VRWC01] Guido Van Rossum, Barry Warsaw, and N Coghlan. Style guide for python code. *Section: Maximum Line Length. url: http://www. python. org/dev/peps/pep-0008*, 2001.

[WAB+17] Szymon Wasik, Maciej Antczak, Jan Badura, Artur Laskowski, and Tomasz Sternal. A survey on online judge systems and their applications. *arXiv preprint arXiv:1710.05913*, 2017.

[Wil15] Chris Wilcox. The role of automation in undergraduate computer science education. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 90–95, New York, NY, USA, 2015. ACM.

[WJH17] David Kofoed Wind, Rasmus Malthe Jørgensen, and Simon Lind Hansen. Peergrade-better feedback for your students. *Peergrade. Np*, 2017.

[WLF+12] Yanqing Wang, Hang Li, Yuqiang Feng, Yu Jiang, and Ying Liu. Assessment of programming language learning based on peer code review model: Implementation and experience report. *Computers & Education*, 59(2):412–422, 2012.

[WRH+12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[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.

Apêndice A

Can Students Help Themselves? An Investigation of Students' Feedback on the Quality of the Source Code

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

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

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

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

I. Introduction

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

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

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

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

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

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

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

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

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

II. Background

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

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

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

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

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

III. Related work

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

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

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

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

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

IV. Overview of introductory programming subject from CS/UFCG

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

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

TABLE I
Units of the subject.

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

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

V. Study design

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

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

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

A. Participants

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

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

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

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

B. Survey

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

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

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

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

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

Fig. 1. Correct code but with quality issues.

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

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

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

C. Metrics

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

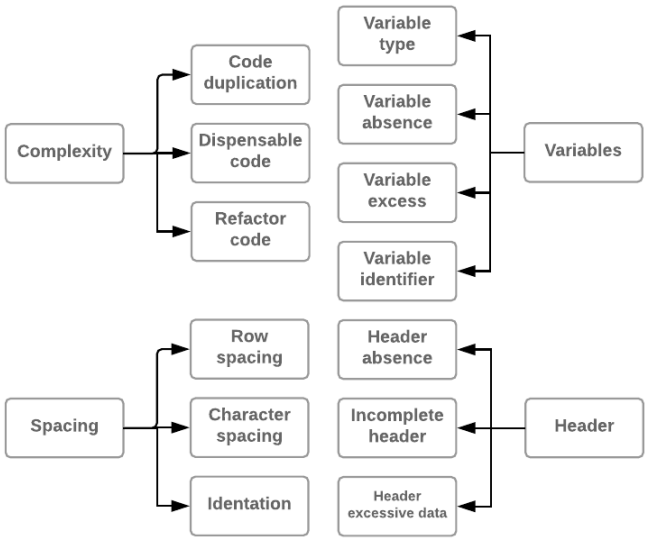

Fig. 2. Code quality issues tags found in this study.

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

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

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

VI. ANALYSIS AND DISCUSSION

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

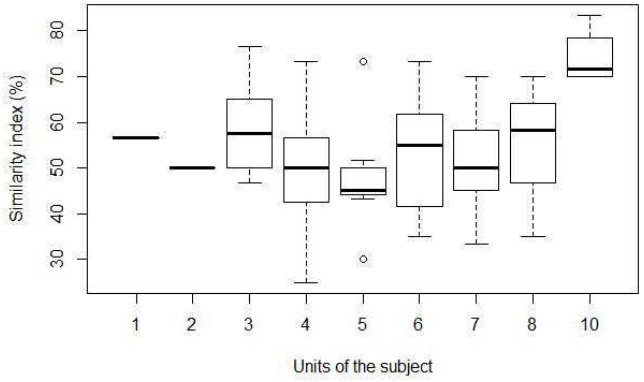

Fig. 3. Similarity between code quality aspects identified by students and teachers.

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

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

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

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

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

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

TABLE III
AMOUNT OF HINTS BY REPORTED CODE QUALITY ISSUES.

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

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

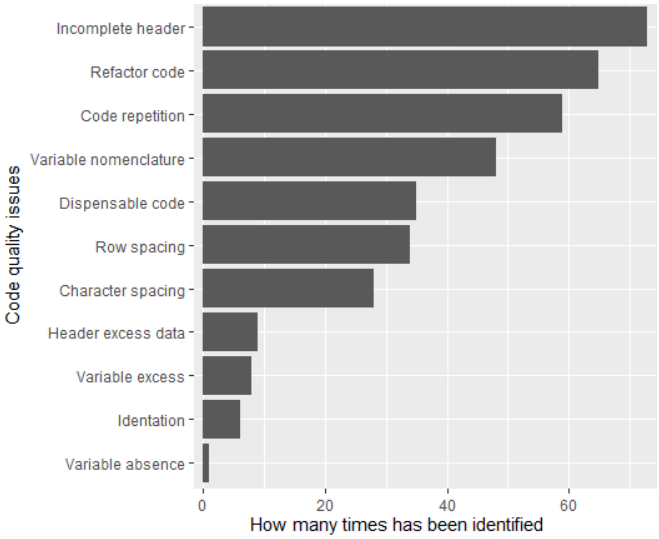

Fig. 4. Ranking of the code quality issues reported in the students' hints.

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

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

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

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

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

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

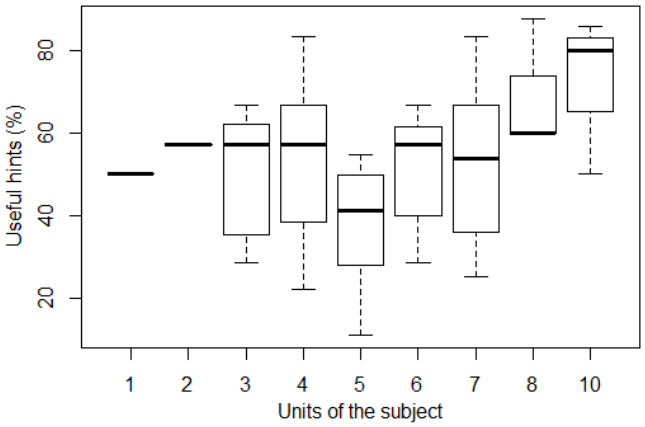

Fig. 5. Useful hints gave by students'.

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

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

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

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

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

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

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

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

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

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

TABLE IV
EXTRACTION OF METRICS.

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

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

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

VII. THREATS TO VALIDATE

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

A. Conclusion

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

B. Internal

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

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

C. Construct

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

D. External

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

VIII. CONCLUSION AND FUTURE WORK

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

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

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

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

IX. Acknowledgment

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[16] Y. Wang, H. Li, Y. Feng, Y. Jiang, and Y. Liu. Assessment of programming language learning based on peer code review model: Implementation and experience report. *Computers & Education*, 59(2):412–422, 2012.

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

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

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

Apêndice B

Investigando o Feedback dos Alunos sobre Aspectos Qualitativos do Código: Um Estudo de Caso

Investigando o *Feedback* dos Alunos sobre Aspectos Qualitativos do Código: Um Estudo de Caso

Raul Andrade [1]

[1]Departamento de Sistemas e Computação
Universidade Federal de Campina Grande
Campina Grande, Brasil

joseraul@copin.ufcg.edu.br

Abstract. *Introductory programming courses typically involve a large number of assignments, which makes it difficult for the teachers to provide manual feedback. Therefore, we investigated if by including students as reviewers we could provide useful feedback. For this, we selected a survey with assignments and their respective source codes (formulated by students in previous turns) so that specialists and students of two courses gave hints to improve the source-codes qualitatively. We found that most students elaborated useful hints, identify code quality issues similar to specialists and that students are particularly able at finding and giving hints related to the programs' complexity.*

Resumo. *A disciplina de introdução à programação normalmente envolve uma grande quantidade de atividades, o que torna custoso para os professores fornecerem feedback manual. Assim sendo, investigamos se, ao incluir os alunos como revisores, poderíamos fornecer feedback útil. Para isso, selecionamos uma lista de atividades e seus respectivos códigos-fonte (formulados por alunos em semestres anteriores) para que especialistas e alunos de dois cursos dessem dicas de como melhorar qualitativamente os códigos. Verificamos que a maioria dos alunos elaboram dicas úteis, identificam problemas de qualidade semelhante aos especialistas e que eles são particularmente hábeis em identificar e elaborar dicas sobre problemas de complexidade dos programas.*

1. Introdução

A competência em programação é fundamental para formação dos estudantes nos cursos de Computação. Entretanto, o processo de ensino e aprendizagem nessas disciplinas apresenta alguns desafios, dentre eles destaca-se o elevado número de alunos por turma, principalmente nas disciplinas introdutórias [Wilcox 2015]. A disciplina de introdução à programação, normalmente, envolve uma grande quantidade de atividades (tarefas), o que torna custoso para os professores proverem *feedback* manual ao longo do semestre letivo. Porém, na literatura identificamos estudos que apontam que a dificuldade do professor em acompanhar individualmente o desempenho de cada estudante é um dos fatores que motivam desistências e reprovações [Yadin 2011] [Barbosa et al. 2015].

Com intuito de minimizar esse problema, são realizados estudos acerca de abordagens para prover *feedback* automático sobre o código-fonte dos estudantes [Gao et al. 2016][Singh et al. 2013]. Em geral, as propostas implementam a ideia de Juízes *Online* [Wasik et al. 2017], na qual corrigem automaticamente a solução submetida

DOI: 10.5753/cbie.wcbie.2018.599

com testes pré-definidos pelos professores ou monitores da disciplina. O foco principal desses trabalhos está no *feedback* funcional. Isto é, se o programa está correto de acordo com os testes. Contudo, há também a necessidade de analisar a qualidade do código produzido pelos alunos.

A qualidade do código está relacionada a aspectos como: complexidade da solução, código duplicado, nome das variáveis, entre outros [Keuning et al. 2017]. Embora haja esforços de pesquisa nesse sentido, as propostas são de ferramentas que automatizam análises qualitativas baseadas em aspectos sintáticos, como os critérios definidos pela comunidade Python no PEP 8 [van Rossum et al. 2001] e o *feedback* fornecido pode ser genérico e ainda precisar do professor para ser efetivo. Portanto, investigamos se, ao incluirmos os alunos como avaliadores, poderíamos fornecer *feedback* útil para outros estudantes. Assim, neste estudo, investigamos se os alunos podem avaliar (formativamente) a qualidade dos programas de seus colegas.

Para isso, selecionamos uma lista de atividades e seus respectivos códigos-fonte, formulados por alunos em semestres anteriores, e solicitamos que especialistas e alunos dessem dicas de como melhorar qualitativamente esses códigos. Participaram deste estudo 51 alunos de dois cursos de introdução à programação de uma universidade do estado da Paraíba e 4 especialistas (estudantes de pós-graduação em Ciência da Computação). Analisamos um total de 200 submissões funcionalmente corretas, referentes à quatro atividades diferentes. Verificamos que a maioria dos alunos identificou problemas de qualidade de código com similaridade igual ou superior a 50% em comparação com os especialistas e que eles são particularmente hábeis em identificar e elaborar dicas sobre a complexidade dos programas.

Com base em nossa análise, os alunos conseguem elaborar, em um nível significativo, mesmo que não ideal, *feedback* útil e similar ao dos especialistas sobre a qualidade do código de outros estudantes. Este estudo pode levar a investigações adicionais sobre como abordar a qualidade do código-fonte na aprendizagem colaborativa e também apoiar o desenvolvimento de ferramentas lint, uma vez que fornece informações detalhadas sobre como os alunos elaboram *feedback* sobre a qualidade do código-fonte.

Este artigo está organizado da seguinte forma: a Seção 2 aborda a importância do *feedback* e a prática de revisão distribuída de código-fonte. A Seção 3 descreve a metodologia que adotamos para este estudo. Os resultados do estudo, incluindo uma discussão, são apresentadas na Secção 4. Nós consideramos e discutimos sobre as ameaças de validade na Seção 5. Finalmente, abordamos as conclusões do estudo juntamente com as instruções para trabalhos futuros na Seção 6.

2. Revisão por Pares

Nos cursos de Computação, as disciplinas relacionadas à programação têm papel fundamental na formação dos estudantes. No entanto, é evidente a dificuldade na aprendizagem dessas disciplinas [Wilcox 2015]. De acordo com Head [Head et al. 2017], dentre os desafios que influenciam no aprendizado está a dificuldade do professor em acompanhar individualmente o desempenho dos estudantes. Isso ocorre devido as turmas, principalmente das disciplinas introdutórias, possuírem um elevado número de estudantes matriculados por semestre. Assim sendo, é inviável para o professor fornecer efetivamente *feedback* manual para todos os alunos nas atividades.

Segundo Schunk e Petermann [Schunk and Petermann 1989], o *feedback* possibilita refletir sobre as diferenças entre o resultado alcançado e o que é esperado, podendo assim ser considerado um importante aliado no processo de construção do conhecimento. Uma abordagem comumente utilizada na computação para prover *feedback* é a revisão por pares (do inglês, *peer review* ou *peer assessment*).

A revisão de código por pares é uma abordagem utilizada por desenvolvedores e, quando realizada de forma adequada, pode melhorar significativamente a qualidade do código do projeto de software, assim como aperfeiçoar continuamente as habilidades técnicas dos profissionais [Kern and Saraiva 1999]. A finalidade de utilizar a revisão distribuída é melhorar a qualidade de determinado trabalho, a partir do *feedback* colaborativo. Desse modo, alguns pesquisadores defendem que a revisão distribuída de código no cenário educacional também pode trazer melhorias significativas à aprendizagem de quem colabora [Glassman et al. 2016] [Wang et al. 2012] .

De acordo com Trytten [Trytten 2005], a revisão distribuída de código, quando aplicada no ambiente educacional, tem objetivos característicos, um deles é apresentar aos alunos que há comunicação entre os engenheiros de software. Como Trytten destaca, os estudantes podem idealizar que programar é uma atividade solitária, pois é comum nas disciplinas os alunos programarem seus códigos individualmente e aprenderem que é uma prática ruim se basear em códigos de outras pessoas. Entretanto, a realidade é bem diferente. Devido o tamanho e complexidade dos projetos de softwares atuais, é necessário que os engenheiros trabalhem grande parte da vida profissional em equipe. Assim, a revisão distribuída de códigos pode auxiliar na conscientização dos estudantes, tanto os introvertidos quanto os mais sociais, sobre a importância de se conectarem uns com os outros e de um ambiente de colaboração para construção mútua da aprendizagem.

O outro objetivo, segundo Trytten [Trytten 2005], é estimular os alunos a aprenderem a ler o código. Ler código e escrever código são atividades distintas e ambas têm sua importância no contexto de mercado. É comum que alunos iniciantes tenham dificuldade para resolver determinados problemas de programação por não possuir maturidade para elaborar soluções alternativas para resolvê-lo. Porém, essa habilidade poderia ser desenvolvida mais facilmente se os alunos fossem incentivados a analisar outras soluções para o problema. A revisão distribuída de código permite que os alunos vejam diferentes formas de solucionar um mesmo problema já resolvido.

A revisão realizada pelos alunos (auto avaliação ou em pares) é um processo que tem a interação como um mecanismo fundamental para construção do conhecimento. Nesse processo, os estudantes compartilham informações, tomam decisões, argumentam, entre outras ações. Essa prática tem se mostrado eficiente, principalmente em cursos *online* com muitos estudantes [Kulkarni et al. 2015]. De acordo com Shang *et al.* [Shang et al. 2001], relacionar a experiência com a interação pode auxiliar os estudantes na atribuição de significados, devido à reflexão de pontos de vista distintos. Dessa forma, incluir o aluno no processo de revisão pode não somente minimizar o problema de escala para obter *feedback*, mas também melhorar a formação, pois ele pode tornar-se mais crítico e autônomo na sua aprendizagem.

3. Metodologia

O método de pesquisa que empregamos neste estudo foi um *survey* supervisionado. O *survey* foi elaborado para verificar se os alunos conseguem fornecer *feedback* sobre como melhorar qualitativamente o código-fonte de outros estudantes. Caso consigam, identificar quais problemas de qualidade são reportados e se têm similaridade com os identificados pelos especialistas, para assim responder às questões de pesquisa:

- **Q1:** O quão similar aos especialistas os alunos de Introdução à Programação conseguem identificar problemas na qualidade do código de outros estudantes?
- **Q2:** Quais problemas de qualidade de código são identificados mais frequentemente pelos alunos de Introdução à Programação?
- **Q3:** O quão útil são as dicas dos alunos de Introdução à Programação sobre problemas na qualidade de código?

O *survey* foi aplicado presencialmente para os alunos durante o horário de aula e o tempo para conclusão foi de até 60 minutos.

3.1. Participantes

Participaram deste estudo 4 especialistas, alunos de pós-graduação em Ciência da Computação que pesquisam sobre ensino de programação introdutória, e 51 alunos matriculados na disciplina de introdução à programação, sendo 27 do curso de Sistemas de Informação (BSI) e 24 da licenciatura em Ciência da Computação (LCC), ambos de uma universidade do estado da Paraíba, do segundo semestre de 2017. O experimento ocorreu no horário da aula, ou seja, participou quem estava presente. Na nossa amostra, 82.3% (42) dos alunos é do sexo masculino e 17,6% (9) do sexo feminino, Para 15,6% (8) dos alunos, não era a primeira vez que cursava a disciplina.

3.2. *Survey*

Aplicamos um *survey* com atividades e seus respectivos códigos-fonte elaborados por estudantes em semestres anteriores. Os códigos estão funcionalmente corretos, mas têm problemas quanto a qualidade, como: mais complexidade que o necessário, repetições de trechos de código, excesso de espaçamento, entre outros. Neste estudo, nos concentramos nos seguintes problemas de qualidade: (i) **complexidade** (duplicação de código, código dispensável e legibilidade do código); (ii) **espaçamento** (espaçamento entre linhas, espaçamento entre caracteres e indentação do código); e (iii) **variáveis** (tipo, ausência, excesso e nomenclatura).

As atividades presentes no *survey*, assim como suas respectivas soluções, foram selecionadas considerando a presença dos principais aspectos de qualidade do código abordados neste estudo (detalhado na Tabela 1).

Tabela 1. Aspectos gerais de qualidade de código abordados neste estudo.

Aspecto	O que é analisado
Complexidade	Verifica se a solução está mais complexa do que deveria e se é possível simplificá-la.
Variável	Verifica a ausência ou excesso de variáveis no código e aspectos relacionados à nomenclatura.
Espaçamento	Verifica a falta ou excesso de espaços entre linhas e caracteres do código e problemas relacionados à indentação.

3.3. Métricas

Para responder a Q1, comparamos e analisamos o *feedback* a partir da codificação das dicas. Para isso, utilizamos uma técnica de análise qualitativa de dados [Brinkman and Kvale 2015], onde: primeiramente, lemos o texto das dicas. Em seguida, rotulamos os aspectos de qualidade de código-fonte importantes. Então, definimos quais *tags* são mais importantes e categorizamos. Por fim, definimos quais são mais relevantes e como estão conectadas. Para ilustrar esse processo, vamos analisar o seguinte exemplo de dica fornecida: *"Há código duplicado nas linhas 14 e 22. Os prints poderiam ser só no final do programa para tornar o código mais legível"*. Para este caso, definimos as seguintes *tags*: **complexidade**, **código duplicado** e **legibilidade do código**. A classificação por *tags* possibilitou comparar e analisar as dicas (presumivelmente subjetivas) com maior precisão. Cada dica é composta por uma *tag* ou um conjunto delas. A Figura 1 mostra as *tags* que identificamos. Para cada dica, consideramos pelo menos um dos três aspectos gerais de qualidade de código-fonte mencionados na Tabela 1.

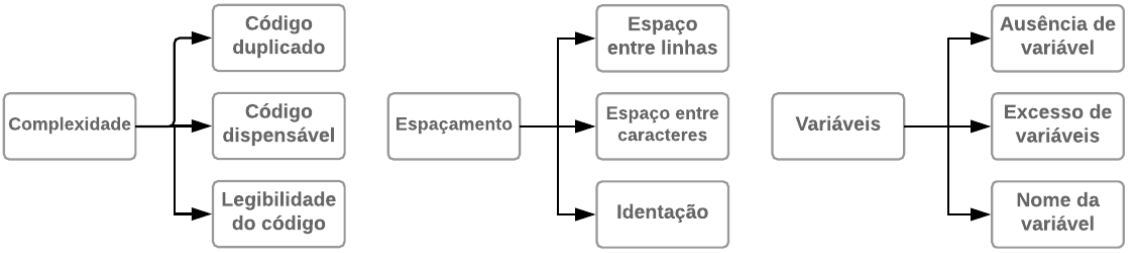

Figura 1. *Tags* dos problemas de qualidade de código analisados neste estudo.

Em seguida, analisamos as dicas dos especialistas para formar um gabarito dos problemas de qualidade de código a serem identificados pelos estudantes. Assim, para cada questão do *survey*, incluímos no gabarito as dicas dadas por pelo menos dois dos quatro especialistas. Medimos a similaridade entre as dicas dos alunos e o gabarito (dicas dos especialistas) utilizando o coeficiente de similaridade de Jaccard (I). Ele compara o número de elementos iguais entre dois grupos e o número total de elementos envolvidos, excluindo o número de ausências conjuntas. O índice de similaridade nesse modelo varia entre 0 e 1, sendo que quanto mais próximo de 1, maior a similaridade entre os grupos. Neste estudo, calculamos a similaridade entre as dicas de cada aluno (Da) e as dicas do gabarito (Dg).

Na literatura, não há índice ideal a ser alcançado com o coeficiente de Jaccard. Desse modo, consideramos significativo o índice de similaridade a partir de 0,5 (50% similar). A semelhança de 50% não é ideal, mas é um valor significativo considerando que os alunos ainda estão aprendendo a dar *feedback*.

$$S(Da, Dg) = \frac{|Da \cap Dg|}{|Da| + |Dg| - |Da \cap Dg|} \qquad (I)$$

Para responder a Q2, ranqueamos os problemas de qualidade de código que os alunos mais reportaram em suas dicas. Para responder a Q3, selecionamos todas as dicas elaboradas pelos alunos e apresentamos para que professores da disciplina as classificassem

com útil (correta e explicativa) ou inútil (a) correta, mas não explicativa; b) irrelevante; ou b) incorreta). Para essas definições, consideramos, além da corretude, o detalhamento da dica, já que esse é o diferencial de incluir o aluno nessa atividade. De modo complementar, também avaliamos quantitativamente os dados que coletamos na Q1 e na Q3 usando estatística descritiva. Realizamos algumas análises estatísticas simples usando o teste de proporção.

4. Análise e Discussão

Calculamos a similaridade das dicas do aluno a partir da média da similaridade (coeficiente de Jaccard) que ele alcançou em cada questão, quando comparadas com o gabarito definido a partir das dicas dos especialistas. A Figura 2a apresenta a visão geral dos dados obtidos da similaridade dos alunos por curso.

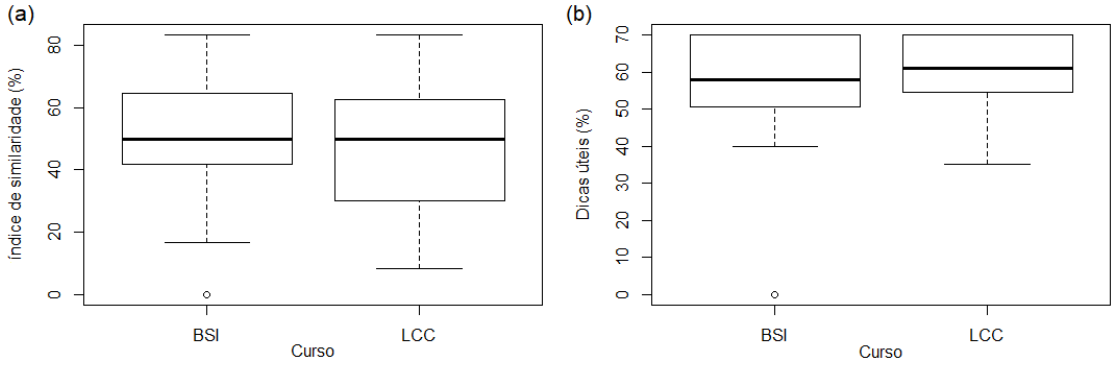

Figura 2. Visão geral dos resultados: (a) Similaridade entre aspectos de qualidade de código identificados por alunos e especialistas e (b) Dicas úteis dos alunos.

Q1: O quão similar aos especialistas os alunos de Introdução à Programação conseguem identificar problemas na qualidade do código de outros estudantes?

Analisamos a similaridade das dicas dos estudantes com os especialistas por curso, no intuito de verificar se existe diferença significativa quanto à forma de identificarem problemas na qualidade dos códigos-fonte. Percebemos que existe variação da similaridade entre os estudantes em ambos os cursos, a variação no LCC foi maior. De acordo com a Figura 3a, os pontos que representam o índice de similaridade mediano dos alunos por curso, apontam que mais da metade dos alunos de BSI elaboraram dicas com similaridade igual ou superior a 50% em relação às dicas elaboradas pelos especialistas. A similaridade de 50% por mais que não seja a ideal, é um valor significativo, considerando que os alunos ainda estão aprendendo a dar *feedback*. O maior índice de similaridade alcançado foi de aproximadamente 83% em ambos os cursos.

De modo complementar, buscamos verificar se a proporção de alunos que elaboram dicas similares com as dos especialistas é significativamente maior. Neste contexto, consideramos admissível a similaridade maior que 0.5 (50% similar). Usando o teste de proporção, constatamos que 62% dos alunos alcançam a similaridade maior que 50% em comparação com os especialistas (p-*value* < 0.04485 e nível de significância de 0.05). Assim, refutamos a hipótese nula. Ou seja, para ambas as turmas, a maior parte dos es-

tudantes consegue alcançar um indice admissível de similaridade com os especialistas, mesmo que em alguns casos essa similaridade não seja a ideal.

Apesar de haver diferenças pouco significativas, os alunos de BSI, em quantidade, apresentaram similaridade maior em relação aos alunos de LCC e também menor variação entre a similaridade dos alunos. Assim, concluímos que o grupo de alunos como um todo é capaz identificar uma quantidade significativa, mas não ideal, de problemas relacionados a qualidade de código similares aos especialistas. Porém, em ambas as turmas, um grupo de alunos consegue identificar com maior precisão esses problemas.

Q2: Quais problemas de qualidade de código são identificados mais frequentemente pelos alunos de Introdução à Programação?

Primeiramente consideramos os principais problemas de qualidade de código analisados neste estudo (Tabela 1). Verificamos que a maior parte das dicas abordam problemas de complexidade, como mostra a Tabela 2. Não há diferença significativa entre a quantidade de dicas sobre espaçamento e variáveis.

Tabela 2. Quantidade de dicas por problema de qualidade reportado.

Problema de qualidade do código-fonte	Quantas vezes foi reportado?
Complexidade	249
Espaçamento	101
Variável	92

Além dos problemas de qualidade de código já citados, outros aspectos mais específicos também foram identificados pelos alunos. Para entender melhor esse cenário, decidimos rankear os problemas de qualidade identificados nas dicas, como mostra a Figura 3.

A maior quantidade de dicas foi sobre legibilidade e código duplicado. Nesta pesquisa, classificamos como legibilidade as dicas de boas práticas e melhoria da organização do código sem remover ou adicionar novas linhas e funcionalidades. Enquanto código duplicado, como o nome diz, definimos como trechos do código que se repetem, podendo ser reduzidos.

Observamos que dicas sobre a indentação do código não foram identificados pelos alunos, no caso, essas dicas foram elaboradas apenas pelos especialistas. O inverso disso ocorreu com as dicas sobre adicionar mensagens nas entradas dos programas e adicionar comentários ao código. O último foi citado oito vezes, porém não é considerada uma boa prática. Além dessas, houveram dicas, de alunos e especialistas, sobre adicionar cabeçalho nos programas. Contudo, acreditamos que esse problema foi pouco mencionado devido os professores não cobrarem isso das turmas nas atividades.

Q3: O quão útil são as dicas dos alunos de Introdução à Programação sobre problemas na qualidade de código?

Para responder esta questão de pesquisa, selecionamos todas as dicas elaboradas pelos alunos e apresentamos para que os especialistas as classificassem com útil ou inútil.

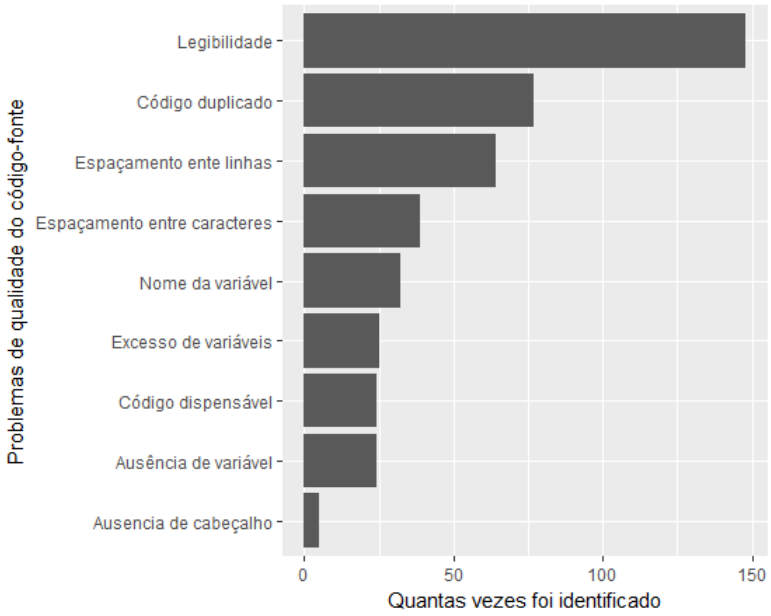

Figura 3. *Ranking* dos problemas de qualidade reportados pelos estudantes.

Definimos com útil a dica correta, personalizada e suficientemente explicativa para melhorar o código-fonte a partir dela. Já como inútil, classificamos as dicas: (i) corretas, mas não explicativas o suficiente para melhorar o código-fonte a partir dela, (por exemplo, "*O código está muito complexo*") (ii) irrelevantes para melhorar a qualidade do código ou (iii) incorretas. Para essas definições, consideramos, além da corretude, o detalhamento do *feedback* , já que esse é o diferencial de incluir o aluno nessa atividade. A Figura 3b apresenta a visão geral dos dados obtidos.

Verificamos que mais de 50% das dicas de 34, dos 51 alunos participantes, foram classificadas como úteis. Verificamos também que a maior parte dessas dicas eram sobre problemas de complexidade. Assim como na similaridade, não houve diferença significativa entre os resultados, porém, os alunos de LCC apresentaram mais dicas úteis. Em média, os estudantes deram três dicas por questão do *survey*, sendo pouco mais de 70% delas corretas, mesmo que não úteis. No entanto, os especialistas mencionaram que algumas dicas foram pouco detalhadas ou tinham problemas com a terminologia de programação.

Para complementar a resposta da Q3, buscamos verificar se a proporção de alunos que elaboram dicas úteis é significativamente maior. Consideramos admissível a similaridade maior que 0.5 (50% similar). Usando o teste de proporção, constatamos que 68% dos alunos tiveram mais de 50% de suas dicas avaliadas como úteis ($p\text{-}value < 0.005456$, nível de significância de 0.05). Assim, refutamos a hipótese nula. Ou seja, a maior parte dos alunos do BSI e LCC consegue elaborar mais dicas úteis, mesmo que em alguns casos a porcentagem de dicas úteis não seja a ideal.

Os estudantes em sua maioria foram capazes de elaborar *feedback* útil. Assim, mesmo com os riscos, acreditamos que eles podem fornecer *feedback* sobre a qualidade do código de seus colegas. Porém, defendemos que com a experiência em prover *feedback* e ele sendo fornecido por mais de um estudante tende a ser mais efetivo.

5. Ameaças à Validade

O tamanho da amostra deste estudo pode ter sido pequeno para obter conclusões profundas sobre os resultados. Por isso, os participantes deste experimento são representativos apenas das turmas e no semestre em que ocorreu. Assim, talvez não possamos generalizar os resultados para outros contextos. Esse estudo deve ser replicado em outras turmas de Introdução à Programação para obter resultados mais genéricos.

Além disso, este estudo investiga muitos problemas de qualidade de código-fonte, de diferentes aspectos, assim, alguns construtos podem não ter sido medidos pelo estudo. Para minimizar essas ameaças, selecionamos técnicas empiricamente validadas e comumente usadas em estudos empíricos científicos da comunidade de Educação em Ciência da Computação.

6. Conclusão e Trabalhos Futuros

Como mencionamos, os estudos no campo de *feedback* automático para atividades de programação abordam principalmente aspectos funcionais dos códigos. Embora existam ferramentas que analisam a qualidade do código, seu *feedback* pode não ser detalhado o suficiente e, para serem efetivas, elas exigem a análise manual dos professores. Dessa forma, investigamos se, ao incluir alunos como avaliadores, poderíamos fornecer *feedback* personalizado.

Observamos que, de modo geral, os estudantes são particularmente hábeis em identificar e elaborar dicas sobre problemas relacionados à complexidade dos programas e conseguem elaborar boas dicas de qualidade de código em um nível significativo, mesmo que não seja o ideal (principalmente LCC). Porém um grupo de alunos, em ambos os cursos, alcançou bons resultados. Portanto, defendemos que, com a experiência em prover *feedback* e se mais de um estudante elaborar dicas para determinada solução, o *feedback* tende a ser útil, ou mais útil que as dicas de apenas um estudante.

Este artigo contribui para estudos sobre os efeitos de estratégias de aprendizagem colaborativa no contexto de cursos introdutórios de programação. No entanto, ainda são necessários estudos para examinar as razões pelas quais os alunos produzem código de baixa qualidade e como lidam com problemas de qualidade. Esse tipo de metodologia tem um importante papel no ensino de Computação, não apenas para minimizar problemas de escala, mas também para desenvolver aprendizes reflexivos.

Para trabalhos futuros, pretendemos: (i) criar atividades nas quais os alunos irão revisar o código de seus colegas e verificar o efeito do *feedback* em suas soluções e (ii) desenvolver um modelo para identificar automaticamente o perfil dos alunos que elaboram *feedback* útil.

7. Agradecimentos

Agradecemos à CAPES por apoiar este trabalho com a concessão de bolsa de estudos.

Referências

Barbosa, A., Correia, A., Costa, D., and Costa, E. (2015). Um mapeamento sistemático sobre analisadores de código em disciplinas de programação. In *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*, volume 26, page 1235.

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

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

Glassman, E. L., Lin, A., Cai, C. J., and Miller, R. C. (2016). Learnersourcing personalized hints. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*, pages 1626–1636. ACM.

Head, A., Glassman, E., Soares, G., Suzuki, R., Figueredo, L., D'Antoni, L., and Hartmann, B. (2017). Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*, pages 89–98. ACM.

Kern, V. M. and Saraiva, L. M. (1999). Aplicação da revisão pelos pares no ensino de graduação. *Alcance, Itajaí, ano VI*, (3):42–49.

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

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

Schunk, N. and Petermann, K. (1989). Measured feedback-induced intensity noise for 1.3 μm dfb laser diodes. *Electronics Letters*, 25(1):63–64.

Shang, Y., Shi, H., and Chen, S. (2001). An intelligent distributed environment for active learning. *Journal on Educational Resources in Computing (JERIC)*, 1(2es):4.

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

Trytten, D. A. (2005). A design for team peer code review. In *ACM SIGCSE Bulletin*, volume 37, pages 455–459. ACM.

van Rossum, G., Warsaw, B., and Coghlan, N. (2001). Pep 8: style guide for python code. *Python. org*.

Wang, Y., Li, H., Feng, Y., Jiang, Y., and Liu, Y. (2012). Assessment of programming language learning based on peer code review model: Implementation and experience report. *Computers & Education*, 59(2):412–422.

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

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

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

Apêndice C

Survey do Estudo de Caso 1

Universidade Federal de Campina Grande - UFCG
Centro de Engenharia Elétrica e Informática - CEEI
Programa de Pós-Graduação em Ciência da Computação - PPGCC

DICAS [CONCRETAS] DE COMO MELHORAR A QUALIDADE DO CÓDIGO

Este questionário faz parte do experimento sobre a aplicação de tutoria colaborativa na disciplina de programação I, e tem por objetivo investigar os efeitos da utilização dessa técnica no contexto de alunos do curso de bacharelado em Ciência da Computação da Universidade Federal de Campina Grande.

Aluno (a): _____ semestre letivo: _____

1. Essa é a primeira vez que está cursando a disciplina de programação I?

☐ Sim. ☐ Não. Quantas vezes cursou a disciplina? _____.

2. Em qual unidade da disciplina você está?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

☐ 6 ☐ 7 ☐ 8 ☐ 9 ☐ 10

3. A seguir são apresentadas algumas atividades de programação resolvidas por alunos em semestres anteriores. Elabore dicas de como melhorar a qualidade dos códigos, como mostra o Exemplo 1. Lembrando que os principais (MAS NÃO APENAS ESSES) aspectos qualitativos são:

I – **Cabeçalho da questão**: Verificar se a solução possui cabeçalho e se ele está claro e de acordo com a questão.

II – **Complexidade**: Verificar se a solução está mais complexa do que deveria e se é possível simplificá-la (diga como).

III – **Espaçamento**: Verificar se a solução apresenta espaçamento correto.

IV – **Nome das variáveis**: Verificar se os nomes das variáveis da questão estão claros. Se é utilizado algum caractere especial.

Universidade Federal de Campina Grande - UFCG
Centro de Engenharia Elétrica e Informática - CEEI
Programa de Pós-Graduação em Ciência da Computação - PPGCC

Exemplo 1. Como elaborar dicas para melhorar a solução.

DESCRIÇÃO DA QUESTÃO

Exemplo 1. Escreva um programa que utiliza o menor dos extremos para classificar uma sequência de números inteiros. Considere que a sequência sempre possui pelo menos dois números inteiros. O menor dos extremos é definido como sendo o menor valor entre o primeiro e o último elemento dessa sequência de números. Por exemplo, se o primeiro inteiro da sequência for 3 e o último inteiro for 7, o menor dos extremos é 3.

Entrada

Na primeira linha da entrada, é lida a quantidade N de números da sequência, N > 1. As N linhas seguintes correspondem aos N números inteiros da sequência.

Saída

Na saída, seu programa deve imprimir o menor dos extremos e os totais de números abaixo e acima do menor dos extremos.

SOLUÇÃO A SER AVALIADA

```
# UFCG - ALUNO - MATRICULA - PROG1 - 11/08/** CLASSIFICA NUMEROS PEL
#MENOR DOS EXTREMOS
# -*- coding: utf-8 -*-
entradas = raw_input()
lista = []
menores = 0
maiores = 0

for i in range(int(entradas)):
        numeros = int(raw_input())
        lista.append(numeros)

if lista[-1] < lista[0]:
        print "Menor dos extremos: %d" % lista[-1]
        for i in lista:
                if int(i) < lista[-1]:
                        menores += 1
                if int(i) > lista[-1]:
                        maiores += 1
        print "%d número(s) abaixo do menor" % menores
        print "%d número(s) acima do menor" % maiores
else:
        print "Menor dos extremos: %d" % lista[0]
        for i in lista:
                if int(i) < lista[0]:
                        menores += 1
                if int(i) > lista[0]:
                        maiores += 1
        print "%d número(s) abaixo do menor" % menores
        print "%d número(s) acima do menor" % maiores

        #if (lista[-1] -1) < lista[-1]:
#               menores += 1
#               lista[i] -= 1
#       elif (lista[1]) < lista[0]:
#               menores += 1
#               lista[i] += 1
```

Dica (s) de como melhorar a QUALIDADE da solução apresentada:

Dica 1: Evite repetição de código movendo os prints para o final.
Dica 2: Remova o código comentado.
Dica 3: Melhore a descrição no cabeçalho.
Dica 4: _____
Dica 5: _____

Universidade Federal de Campina Grande - UFCG
Centro de Engenharia Elétrica e Informática - CEEI
Programa de Pós-Graduação em Ciência da Computação - PPGCC

Assim como apresentado no EXEMPLO ANTERIOR, elabore dicas de como melhorar as soluções a seguir em caráter QUALITATIVO.

DESCRIÇÃO DA QUESTÃO

Questão 1. Escreva um programa que utiliza o menor dos extremos para classificar uma sequência de números inteiros. Considere que a sequência sempre possui pelo menos dois números inteiros. O menor dos extremos é definido como sendo o menor valor entre o primeiro e o último elemento dessa sequência de números. Por exemplo, se o primeiro inteiro da sequência for 3 e o último inteiro for 7, o menor dos extremos é 3.

Entrada

Na primeira linha da entrada, é lida a quantidade N de números da sequência, N > 1. As N linhas seguintes correspondem aos N números inteiros da sequência.

Saída

Na saída, seu programa deve imprimir o menor dos extremos e os totais de números abaixo e acima do menor dos extremos.

SOLUÇÃO A SER AVALIADA

```
# coding: utf-8
# Ciência da Computação - UFCG
# Programação 1
# Questão: Classifica Números pelo Menor dos Extremos
# Aluno: **********
# Matricula: *********
# E-mail: ******@ccc.ufcg.edu.br
# ------------------------------------------------------

def qtd_menor(lista, numero):
        soma=0
        for i in range(len(lista)):
                if lista[i] < numero:
                        soma+=1
        return soma

def qtd_maior(lista, numero):
        soma=0
        for i in range(len(lista)):
                if lista[i] > numero:
                        soma+=1
        return soma


lista=[]
repeticoes=int(raw_input())

for i in range(repeticoes):
        num=int(raw_input())
        lista.append(num)

if lista[0] < lista[len(lista)-1]:

        print "Menor dos extremos: %d" %lista[0]
        abaixo=qtd_menor(lista, lista[0])
        print "%d número(s) abaixo do menor" %abaixo
        acima=qtd_maior(lista, lista[0])
        print "%d número(s) acima do menor" %acima
elif lista[0] > lista[len(lista)-1]:

        print "Menor dos extremos: %d" %lista[len(lista)-1]
        abaixo=qtd_menor(lista, lista[len(lista)-1])
        print "%d número(s) abaixo do menor" %abaixo
        acima=qtd_maior(lista, lista[len(lista)-1])
        print "%d número(s) acima do menor" %acima
else:

        print "Menor dos extremos: %d" %lista[0]
        abaixo=qtd_menor(lista, lista[0])
        print "%d número(s) abaixo do menor" %abaixo
        acima=qtd_maior(lista, lista[0])
        print "%d número(s) acima do menor" %acima
```

Dica (s) de como melhorar a QUALIDADE da solução apresentada:

Dica 1: _____
Dica 2: _____
Dica 3: _____
Dica 4: _____
Dica 5: _____

DESCRIÇÃO DA QUESTÃO

Questão 2. Um economista precisa analisar a variação do salário mínimo. Pede-se que você escreva um programa que dados os valores mensais do salário mínimo, determine quais deles ficaram acima de U$ 100.00 (cem dólares) e quais ficaram abaixo.

Entrada
Da primeira linha da entrada, seu programa deve ler o ano inicial da série de dados. Da segunda linha da entrada, deve ler o ano final da série. Em seguida, deve ler os valores do salário mínimo em dólares para cada um dos anos da faixa indicada.

Saída
Seu programa deve imprimir um relatório indicando em quantos anos o salário mínimo ficou acima de U$ 100.00 (cem dólares) e em quantos ficou abaixo (ou igual). Também deve indicar em que percentagem de anos da série o salário ficou nessas faixas de valores e a média dos salários naqueles anos. Se em nenhum ano da série o valor do salário mínimo estiver em uma das faixas, não é necessário imprimir a média de valores.

SOLUÇÃO A SER AVALIADA

```
# coding: utf-8
# **********
# Análise de Variação do Salário Mínimo

ano_inicio = int(raw_input())
ano_final = int(raw_input())

ano = ano_final-ano_inicio+1

salario_ano = []
salario_contador = 0

salario_Abaixo = 0
salario_Acima = 0
for i in range(ano):
        salario = float(raw_input())
        salario_contador += salario
        salario_ano.append(salario)

soma_Acima = 0
soma_Abaixo = 0

for i in range(len(salario_ano)):
        if salario_ano[i] >= 100.0 :
                salario_Acima += 1.0
                soma_Acima += salario_ano[i]

        elif salario_ano[i] < 100.0:
                salario_Abaixo += 1.0
                soma_Abaixo += salario_ano[i]



if salario_Abaixo > 0 and salario_Acima > 0:
        media_Acima = soma_Acima/salario_Acima

        media_Abaixo = soma_Abaixo/salario_Abaixo

        ano_Aba = (salario_Abaixo*100.0) /ano
        print '%i ano(s) abaixo (%.f%% dos anos)' % (salario_Abaixo,ano_Aba
        print 'média dos anos abaixo: U$ %.2f' % media_Abaixo
        ano_Aci = (salario_Acima*100.0) / ano
        print '%i ano(s) acima (%.f%% dos anos)' % (salario_Acima,ano_Aci)
        print 'média dos anos acima: U$ %.2f' % media_Acima

elif salario_Abaixo == 0 and salario_Acima > 0:
        media_Acima = soma_Acima/salario_Acima

        ano_Aba = (salario_Abaixo*100.0) /ano

        print '%i ano(s) abaixo (%.f%% dos anos)' % (salario_Abaixo,ano_Aba

        ano_Aci = (salario_Acima*100.0) / ano
        print '%i ano(s) acima (%.f%% dos anos)' % (salario_Acima,ano_Aci)
        print 'média dos anos acima: U$ %.2f' % media_Acima
elif salario_Abaixo > 0 and salario_Acima == 0 :

        media_Abaixo = soma_Abaixo/salario_Abaixo

        ano_Aba = (salario_Abaixo*100.0) /ano
        print '%i ano(s) abaixo (%.f%% dos anos)' % (salario_Abaixo,ano_Aba
        print 'média dos anos abaixo: U$ %.2f' % media_Abaixo
        ano_Aci = (salario_Acima*100.0) / ano
        print '%i ano(s) acima (%.2f%% dos anos)' % (salario_Acima,ano_Aci
```

Dica (s) de como melhorar a QUALIDADE da solução apresentada:

Dica 1: _____
Dica 2: _____
Dica 3: _____
Dica 4: _____
Dica 5: _____

Universidade Federal de Campina Grande - UFCG
Centro de Engenharia Elétrica e Informática - CEEI
Programa de Pós-Graduação em Ciência da Computação - PPGCC

DESCRIÇÃO DA QUESTÃO

Questão 3. Na segunda metade do século XVII, Leibniz descobriu, através do método infinitesimal, uma soma infinita que lentamente converge para o valor de pi. A série é a seguinte:

pi = 4 x (1/1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 + 1/13 - 1/15 + ...)

Além dessa série, várias outras foram descobertas que permitem calcular o valor de pi e outros números irracionais de interesse. Por convergirem muito lentamente para o valor de pi, contudo, são necessárias muitas iterações para calcular pi com significativa precisão. Pede-se que você escreva um programa que permite calcular uma aproximação de pi, usando a série descrita, *com uma precisão arbitrária*.

Entrada
O programa lê da entrada o valor do erro `e` máximo aceitável. O erro indica a diferença máxima aceitável entre dois valores da série, para que se interrompa o cálculo.

Saída
O programa deve imprimir os valores aproximados de pi calculados a cada passo, até o valor final. Todos os valores devem ser impressos com 6 casas decimais.

SOLUÇÃO A SER AVALIADA

```
# coding: utf-8
#***
#Programação 1

cont = 1
pi = 0.0
numAnterior = 0.0
valorErro = float(raw_input())

while True:
        i = 1.0
        divPi = 0.0
        bol = False
        while not i > cont:

                if bol == False:
                        divPi = divPi + (1.0/i)
                        bol = True
                else:
                        divPi = divPi - (1.0/i)
                        bol = False
                i += 2

        pi = 4.0 * divPi

        diferenca = 0.0

        if pi >= numAnterior:
                diferenca = pi - numAnterior
        else:
                diferenca = numAnterior - pi

        print "%.6f" % pi

        if diferenca < valorErro:
                break

        cont += 2

        numAnterior = pi
```

Dica (s) de como melhorar a QUALIDADE da solução apresentada:

Dica 1: _____
Dica 2: _____
Dica 3: _____
Dica 4: _____
Dica 5: _____

DESCRIÇÃO DA QUESTÃO

Questão 4. A famosa feijoada da cantina de Dona Inês acontece toda quinta-feira no DSC. Os alunos fazem seus pedidos em grupos e em fila. Ou seja, chega um grupo de alunos que pede 10 feijoadas, depois aparece outro grupo que pede 5 feijoadas e assim, por diante.

Dona Inês sempre prepara apenas feijoada suficiente para N pessoas. A demanda, contudo, é tipicamente maior ou igual ao que ela preparou. Quando a feijoada não é capaz de servir todos os alunos de um grupo, todos os alunos do grupo desistem de comer feijoada. Mesmo com feijoadas sobrando, Dona Inês também para de servir feijoadas depois que o primeiro grupo de alunos desistir de comer em sua cantina.

Pede-se que você escreva a função **quantos_comeram (N, fila)** que recebe a quantidade de feijoadas feitas por Dona Inês, e uma representação da fila de grupos de pedidos (cada valor na fila indica quantas feijoadas aquele grupo pediu), e que retorne quantas feijoadas foram, de fato, consumidas no dia.

SOLUÇÃO A SER AVALIADA

```
# coding: utf-8
# **** progl
# quantos comearam ?

def quantos_comeram(N,fila):
        cont = 0
        for i in range(len(fila)):
                if fila[i] <= N:
                        cont += 1
                        N -= fila[i]
                else:
                        break

        comem = 0
        if cont != 0:
                for i in range(cont):
                        comem += fila[i]
        return comem
```

Dica (s) de como melhorar a QUALIDADE da solução apresentada:

Dica 1: _____
Dica 2: _____
Dica 3: _____
Dica 4: _____
Dica 5: _____

DESCRIÇÃO DA QUESTÃO

Questão 5. Escreva um programa que separe uma sequência de inteiros em dois grupos: os menores e os maiores que um dado número central, denominado pivot.

Entrada
Seu programa deve ler o pivot da primeira linha de entrada. Das demais linhas deve ler os números da sequência. A sequência é terminada por um inteiro negativo.

Saída
Seu programa deve imprimir a sequência de valores menores ou iguais ao pivot na primeira linha. Na segunda linha, deve imprimir o pivot. E na terceira, os valores maiores que o pivot. Os valores menores e maiores devem ser impressos na ordem em que foram lidos da entrada.

SOLUÇÃO A SER AVALIADA

```
#!/usr/bin/env python
# coding: utf-8
# Pivot
# (C) 20** ******* / UFCG

def maioresMenores(pivot,array):
        t = 0
        for i in range(len(array)):
                if pivot >= array[i]:
                        for k in range(i,t,-1):
                                array[k] , array[k-1] = array[k-1] , arra
                        t +=1
        maiores = []
        menores = []
        for i in range(t,len(array)):
                maiores.append(array[i])
        for k in range(t):
                menores.append(array[k])
        print menores
        print pivot
        print maiores



pivot = int(raw_input())
elementos = []
entrada = int(raw_input())
while entrada >= 0:
        elementos.append(entrada)
        entrada = int(raw_input())

maioresMenores(pivot,elementos)
```

Dica (s) de como melhorar a QUALIDADE da solução apresentada:

Dica 1: _____
Dica 2: _____
Dica 3: _____
Dica 4: _____
Dica 5: _____

Apêndice D

Survey do Estudo de Caso 2

DICAS [CONCRETAS] DE COMO MELHORAR A QUALIDADE DO CÓDIGO

Este questionário faz parte do experimento sobre a aplicação de tutoria colaborativa na disciplina de Programação I, e tem por objetivo investigar os efeitos da utilização dessa técnica no contexto de alunos do curso de bacharelado em Sistemas de Informação e licenciatura em Ciência da Computação da Universidade Federal da Paraíba.

Aluno (a): _____. E-mail: _____@dce.ufpb.br

1. Essa é a primeira vez que está cursando a disciplina de programação I?

☐ Sim. ☐ Não. Essa é a __ª vez que curso a disciplina.

2. A seguir são apresentadas algumas atividades de programação resolvidas por alunos em semestres anteriores. Elabore dicas de como melhorar a qualidade dos códigos, como mostra o Exemplo 1. Lembrando: os principais aspectos de qualidade (**mas não apenas esses**) a serem considerados são:

A) **Cabeçalho da questão**: verificar se a solução possui cabeçalho, se está de acordo com a questão e não há ausência de informações;

B) **Complexidade**: verificar se a solução está mais complexa do que deveria (repetição de código, código desnecessário...) e se é possível simplificá-la;

C) **Espaçamento**: verificar a falta ou excesso de espaços entre linhas e caracteres do código e problemas relacionados à indentação;

D) **Nome das variáveis**: verificar a ausência ou excesso de variáveis no código e aspectos relacionados à nomenclatura.

Exemplo 1. Como elaborar dicas para melhorar a solução.

DESCRIÇÃO DA QUESTÃO

Exemplo 1. Escreva um programa que utiliza o menor dos extremos para classificar uma sequência de números inteiros. Considere que a sequência sempre possui pelo menos dois números inteiros. O menor dos extremos é definido como sendo o menor valor entre o primeiro e o último elemento dessa sequência de números. Por exemplo, se o primeiro inteiro da sequência for 3 e o último inteiro for 7, o menor dos extremos é 3.

Entrada
Na entrada, é lida a quantidade N de números da sequência, N > 1. As N linhas seguintes correspondem aos N números inteiros da sequência.

Saída
Na saída, seu programa deve imprimir o menor dos extremos e os totais de números abaixo e acima do menor dos extremos.

SOLUÇÃO A SER AVALIADA

```
# coding: utf-8
# Ciência da Computação - UFXX
# Programação
# Questão: Classifica Números pelo Menor dos Extremos
# Aluno: XXXXXXX
# Matricula: XXXXXXX
# E-mail: XXX@XXX.XXX.br
# ------------------------------------------------------

def qtd_menor(lista, numero):
  soma=0
  for i in range(len(lista)):
    if lista[i] < numero:
      soma+=1
  return soma

def qtd_maior(lista, numero):
  soma=0
  for i in range(len(lista)):
    if lista[i] > numero:
      soma+=1
  return soma

entradas = int(input("Informe a quantidade de entradas: "))
lista=[]
for i in range (entradas):
      valor = int(input())
      lista.append(valor)

if lista[0] < lista[len(lista)-1]:

  print("Menor dos extremos: %d" %lista[0])
  abaixo=qtd_menor(lista, lista[0])
  print("%d número(s) abaixo do menor" %abaixo)
  acima=qtd_maior(lista, lista[0])
  print("%d número(s) acima do menor" %acima)
elif lista[0] > lista[len(lista)-1]:

  print("Menor dos extremos: %d" %lista[len(lista)-1])
  abaixo=qtd_menor(lista, lista[len(lista)-1])
  print("%d número(s) abaixo do menor" %abaixo)
  acima=qtd_maior(lista, lista[len(lista)-1])
  print("%d número(s) acima do menor" %acima)
else:

  print("Menor dos extremos: %d" %lista[0])
  abaixo=qtd_menor(lista, lista[0])
  print("%d número(s) abaixo do menor" %abaixo)
  acima=qtd_maior(lista, lista[0])
  print("%d número(s) acima do menor" %acima)
```

Dica (s) de como melhorar a QUALIDADE da solução apresentada:

Dica 1: Fatorar os prints para um único lugar (fim do programa).

Dica 2: Adicionar espaços entre % e o nome da variável na formatação de strings.

Dica 3: Adicionar espaços entre os sinais de = e as expressões.

Dica 4: Contar menores e maiores em um único laço (não precisa de duas funções para isso).

Dica 5: Simplificar condicionais. Não é preciso fazer o último condicional, basta colocar <= no primeiro.

Assim como apresentado no exemplo anterior, elabore dicas de como melhorar as soluções a seguir em caráter qualitativo.

DESCRIÇÃO DA QUESTÃO

Questão 1. Escreva um programa que utiliza o menor dos extremos para classificar uma sequência de números inteiros. Considere que a sequência sempre possui pelo menos dois números inteiros. O menor dos extremos é definido como sendo o menor valor entre o primeiro e o último elemento dessa sequência de números. Por exemplo, se o primeiro inteiro da sequência for 3 e o último inteiro for 7, o menor dos extremos é 3.

Entrada
Na entrada, é lida a quantidade N de números da sequência, N > 1. As N linhas seguintes correspondem aos N números inteiros da sequência.

Saída
Na saída, seu programa deve imprimir o menor dos extremos e os totais de números abaixo e acima do menor dos extremos.

SOLUÇÃO A SER AVALIADA

```
1   # UFXX - NOME DO ALUNO - MATRICULA - PROG - 11/08/**
    # CLASSIFICA NUMEROS PELO MENOR DOS EXTREMOS
2   # -*- coding: utf-8 -*-
3
4   entradas = int(input("Informe a quantidade de entradas
5   lista = []
6   for i in range (entradas):
7           valor = int(input())
8           lista.append(valor)
9
10  menores = 0
11  maiores = 0
12
13
14
15  if lista[-1] < lista[0]:
16    print("Menor dos extremos: %d" % lista[-1])
17    for i in lista:
18      if int(i) < lista[-1]:
19        menores += 1
20      if int(i) > lista[-1]:
21        maiores += 1
22    print("%d número(s) abaixo do menor" % menores)
23    print("%d número(s) acima do menor" % maiores)
24  else:
25    print("Menor dos extremos: %d" % lista[0])
26    for i in lista:
27      if int(i) < lista[0]:
28        menores += 1
29      if int(i) > lista[0]:
30        maiores += 1
31    print("%d número(s) abaixo do menor" % menores)
32    print("%d número(s) acima do menor" % maiores)
33
34    #if (lista[-1] -1) < lista[-1]:
35  #   menores += 1
36  #   lista[i] -= 1
37  # elif (lista[1]) < lista[0]:
38  #   menores += 1
39  #   lista[i] += 1
40
```

Dica (s) de como melhorar a QUALIDADE da solução apresentada:

Dica 1:

Dica 2:

Dica 3:

Dica 4:

Dica 5:

DESCRIÇÃO DA QUESTÃO

Questão 2. A professora Júlia ensina Geografia e precisa de um programa para ler as 4 notas de cada um dos alunos de sua turma e exibir a quantidade de alunos aprovados (média igual ou superior a 8,0) e a média da turma. Além disso, ela espera que o sistema exiba a maior média obtida. Escreva esse programa.

SOLUÇÃO A SER AVALIADA

```
1   maiorMedia=0
2   quantAprovados=0
3   for x in range(1,3+1):
4       print('Aluno',x)
5       somaNotas = 0
6       for i in range(1,4+1):
7           nota = float(input('nota: '))
8           somaNotas+=nota
9       media = somaNotas/4
10      if(media > 0):
11          maiorMedia = media
12      if(media >= 8.0):
13          quantAprovados +=1
14  print('Maior media:',maiorMedia,'\nQuantidae de aprovados:',quantAprovados)
```

Dica (s) de como melhorar a QUALIDADE da solução apresentada:

Dica 1:

Dica 2:

Dica 3:

Dica 4:

Dica 5:

DESCRIÇÃO DA QUESTÃO

Questão 3. Natália abriu uma loja de bijuterias recentemente e as vendas vão muito bem. Pensando em atrair uma clientela ainda maior, ela deseja oferecer um desconto de 5% para os clientes que pagarem à vista, e parcelar as vendas feitas com cartão Visa em até 5 vezes, e com outros cartões em até 3 vezes, de acordo com a escolha do cliente. Mas, para isso, ela precisa que você escreva um programa que receba como entrada o valor da venda, a forma de pagamento e, se for o caso, a quantidade de parcelas desejadas, e exiba o valor total a ser pago, no caso das vendas à vista, e o valor de cada parcela, no caso das vendas com cartão de crédito. Você pode ajudá-la?

SOLUÇÃO A SER AVALIADA

```
1  vv = int(input('Informe o valor da venda: '))
2  fp = input('Forma de Pagamento: ')
3  parcelaMinima=0
4  parcelaMaxima1=3
5  parcelaMaxima2 = 5
6
7  if(fp == 'Cartão'):
8      cartao = input('Informe o cartão: ')
9      if(cartao == 'Visa'):
10         quantParcelas = int(input('Informe a quantidade de parcelas: '))
11         if(quantParcelas < parcelaMinima and quantParcelas > parcelaMaxima2):
12             print('Quantidade de Parcelas não disponivel')
13         else:
14             print('Valor final:',quantParcelas,'X',vv/quantParcelas )
15
16
17     else:
18         quantParcelas = int(input('Informe a quantidade de parcelas: '))
19         if(quantParcelas < parcelaMinima and quantParcelas > parcelaMaxima1):
20             print('Quantidade de Parcelas não disponivel')
21         else:
22             print('Valor final:',quantParcelas,'X',vv/quantParcelas )
23
24 elif(fp == 'A Vista'):
25     print('Valor Final:',vv - 5/100)
26 else:
27     print('Opção Invalida!')
```

Dica (s) de como melhorar a QUALIDADE da solução apresentada:

Dica 1:

Dica 2:

Dica 3:

Dica 4:

Dica 5:

DESCRIÇÃO DA QUESTÃO

Questão 4. Natanael trabalha em uma locadora de DVDs de Rio Tinto e precisa de um programa que o ajude a calcular o valor das locações e das multas dos clientes. Nessa locadora, o aluguel de cada DVD custa R$ 4 se ele for lançamento e R$ 3 para os demais itens. Para cada dia de atraso na entrega, é cobrada uma multa de R$ 5 por item. Escreva um programa que receba como entrada a quantidade de DVDs alugados, o tipo de DVD e a quantidade de dias de atraso de várias locações, até que Natanael não deseje mais usar o programa, e exiba o valor total de cada locação.

SOLUÇÃO A SER AVALIADA

```
q = 0
m = 5

while(q == 0):
    qDVD = int(input('DVDs alugados: '))
    tipoDVD = input('tpo de DVD: ')
    if(tipoDVD == 'lançamento'):
        precoFilme=4
        diasAtraso = int(input('atraso: '))
        valorIn = qDVD * precoFilme
        multaDias = m * diasAtraso
        total = valorIn+multaDias
        print('total: ',total)

        if input('Deseja continuar <s/n>') !=
            q = q -1
        else:
            q = q *1


    else:
        precoFilme = 3

        diasAtraso = int(input('atraso: '))
        valorIn = qDVD * precoFilme
        multaDias = m * diasAtraso
        total = valorIn+multaDias
        print('Valor total: ',total)

        if input('Deseja continuar <s/n>') !=
            q = q -1
        else:
            q = q *1
```

Dica (s) de como melhorar a QUALIDADE da solução apresentada:

Dica 1:

Dica 2:

Dica 3:

Dica 4:

Dica 5: