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.

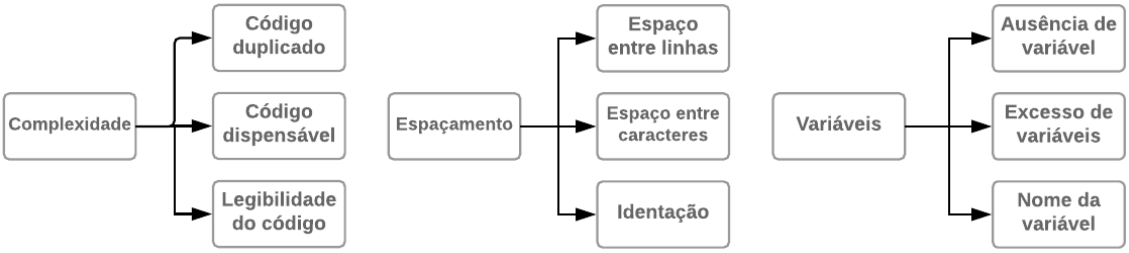

Figura 1. *Tags* dos problemas de qualidade de código analisados neste estudo.

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.

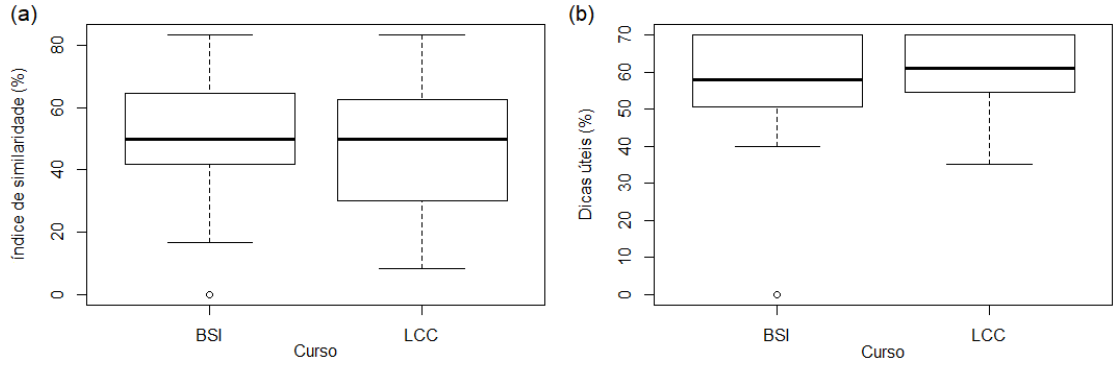

Figura 2. Visão geral dos resultados: (a) Similaridade entre aspectos de qualidade de código identificados por alunos e especialistas e (b) Dicas úteis dos alunos.

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.

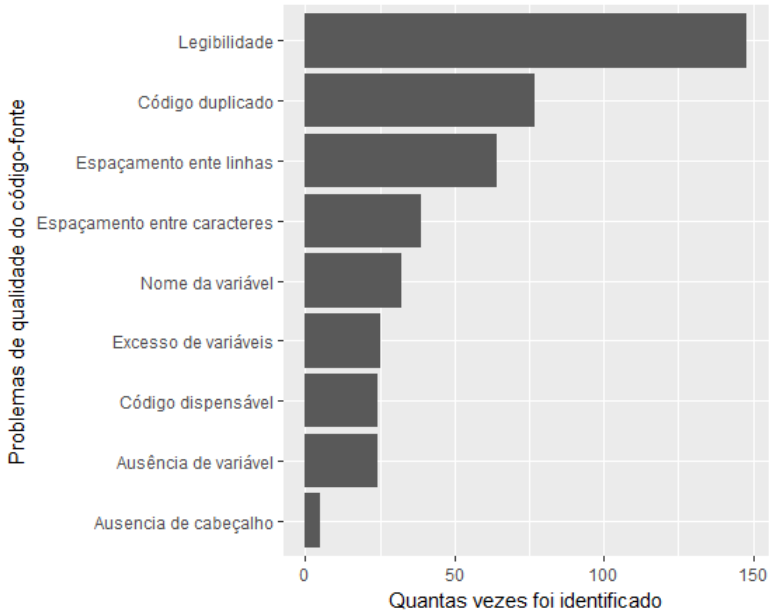

Figura 3. *Ranking* dos problemas de qualidade reportados pelos estudantes.

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-*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.