Análise da Qualidade do Código-Fonte de Educadores que lecionam em Plataformas MOOC

Otávio Vinícius G. S. Rocha[1]

[1]Bacharelado em Engenharia de Software
Instituto de Ciências e Informática - PUC Minas
Ed. Fernanda. Rua Cláudio Manoel, 1.162, Funcionários, Belo Horizonte - MG

otaviovgsr@gmail.com

Abstract. *Frequently, students rely on Massive Open Online Courses (MOOC) to learn new technologies. In these platforms, educators can provide source code to students to perform practical exercises. In this context, the code quality is a relevant factor since issues may impact hundreds of students. Also, the students learn by using these source code examples. In this paper, we investigate the source code quality of these courses. Specifically, we analyze 352 projects, involving five programming languages, Java, JavaScript, TypeScript, Python, and Go. Overall, we detect about 8K issues in over 11K files. Most issues refer to code smells of low gravity. Besides, there is not a significant correlation between the popularity and code quality. In summary, the results suggest that these platforms provide a source code of satisfactory quality for students.*
Keywords: *code quality; course; educators; MOOC Plataform*

Resumo. *Frequentemente, estudantes utilizam Cursos Online Abertos e Massivos (MOOC) para aprender novas tecnologias. Nessas plataformas, os educadores podem fornecer código-fonte dos projetos para que os alunos realizem exercícios práticos. Nesse contexto, a qualidade do código é um fator relevante, visto que problemas podem impactar centenas de alunos. Além disso, os alunos aprendem usando estes exemplos. Neste trabalho, investiga-se a qualidade do código-fonte destes cursos. Especificamente, analisam-se 352 projetos, envolvendo cinco linguagens de programação: Java, JavaScript, TypeScript, Python e Go. Detectou-se cerca de 8 mil problemas de código em mais de 11 mil arquivos. A maioria dos problemas refere-se a code smells de baixa gravidade. Além disso, não existe uma correlação significativa entre a popularidade e a qualidade do código. Em resumo, os resultados sugerem que a qualidade do código disponibilizado nestas plataformas é satisfatória para os estudantes.*
Palavras-chave: *qualidade de código; curso; educadores; plataforma MOOC*

Bacharelado em Engenharia de *Software* — PUC Minas
Trabalho de Conclusão de Curso (TCC)

Orientador de conteúdo (TCC I): Cleiton Tavares — cleitontavares@pucminas.br
Orientadora de conteúdo (TCC I): Simone de Assis — simone@pucminas.br
Orientador acadêmico (TCC I): Laerte Xavier — laertexavier@pucminas.br
Orientadora (TCC II): Aline Brito — alinebrito@pucminas.br

Belo Horizonte, 7 de junho de 2023.

1. Introdução

Dentro da indústria de *software*, existem certos padrões de codificação necessários para a implementação bem sucedida de qualquer produto [Martin 2013]. Para incentivar o crescimento, as organizações precisam de um *software* que funcione perfeitamente e que atendam diversos requisitos de qualidade [Spinellis 2006]. Diante disso, é notável que a qualidade do código-fonte tem uma influência importante no processo de desenvolvimento de um *software*. Embora a *expertise* e a técnica do desenvolvedor sejam de suma importância, tendo papel central na qualidade da escrita de um código, estudos sugerem que o aprendizado em qualidade de *software* deve começar o mais cedo possível [Keuning et al. 2017, Gomes et al. 2021]. Surgindo como um modelo de ensino de programação, os *Massive Open Online Courses* (MOOCs) oferecem cursos abertos, ofertados em plataformas *online* e projetados para receber um número ilimitado de participantes, sem restrição de local ou tempo [Bey et al. 2018].

Muitos pesquisadores se interessam em compreender como os alunos resolvem seus problemas de programação e quais são os erros mais comuns cometidos por eles, que afetam a qualidade do código que produzem. Keuning et al. (2017) examinaram a presença de problemas na qualidade de código em programas estudantis, entendendo a frequência e a diferença de quem faz uso de extensões de análise de código estático. Andrade Gomes et al. (2017) propuseram uma ferramenta que utiliza relatórios de qualidade para analisar o código-fonte dos alunos e fornecer um *feedback* sobre sua codificação. Nesse contexto, tendo em vista que os artigos mencionados visam compreender a qualidade do aprendizado dos alunos numa perspectiva de seu código, pouco se discute sobre a qualidade do código-fonte utilizado pelos educadores. Portanto, o problema a ser tratado neste artigo é **a escassez de análises da qualidade do código-fonte de educadores que ministram cursos de programação para iniciantes em plataformas MOOCs**.

Estudos apontam que os exemplos fornecidos em aulas exercem significativas influências nas fases iniciais da aquisição de habilidades cognitivas, onde os alunos utilizam tais exemplos como modelos em seus próprios projetos [Börstler et al. 2007]. Com a crescente popularidade dos cursos *online* de programação e sabendo que educadores devem estimular seus discentes a melhorarem a qualidade do seu código-fonte [Andrade Gomes et al. 2017], Gomes e Mendes (2014) entrevistaram instrutores de cursos introdutórios de programação e concluíram que determinados instrutores enfatizam que uma boa escolha de exercício de programação e bons materiais didáticos desempenham um papel importante no processo de ensino e aprendizado do curso. Dessa forma, para identificar a falta de boas práticas em códigos-fonte de instrutores de cursos *online* de programação, destaca-se a importância da resolução do problema apresentado neste trabalho.

Perante o exposto, o objetivo geral deste estudo é **analisar a qualidade do código-fonte de educadores que ministram cursos de programação para iniciantes em plataformas MOOCs através de análise estática de código**. Para atingir o objetivo geral, foram definidos os seguintes objetivos específicos: i) avaliar a frequência de problemas de codificação nesses projetos, sendo eles, *code smells*, vulnerabilidades e *bugs*; ii) analisar quais são os problemas encontrados e classificá-los de acordo com sua gravidade; iii) avaliar a relação entre a qualidade do código-fonte do curso e sua popularidade.

Através da coleta de código-fonte de cursos de programação, juntamente com ferramentas de análise de código estático, obtiveram-se informações relacionadas à qualidade do código-fonte utilizado pelos educadores em seus cursos de programação. Sendo assim, através

deste estudo, foram obtidos resultados quantitativos a respeito de más práticas de programação nos exemplos fornecidos pelos educadores. Por fim, espera-se que os resultados obtidos auxiliem diversos educadores nas escolhas dos exemplos de código fornecidos em seus cursos. Dessa maneira, os educadores poderão melhorar suas estratégias de ensino, além de estimular um código limpo, princípios e padrões de *design* de *software* para iniciantes em programação.

As próximas seções do artigo estão divididas da seguinte forma: a Seção 2 apresenta a fundamentação teórica. Na Seção 3, são mostrados os trabalhos relacionados. A Seção 4 traz consigo a metodologia para a realização da pesquisa. As Seções 5 e 6 apresentam e discutem os resultados obtidos no trabalho, respectivamente. A Seção 7 detalha as ameaças a validade e suas mitigações. Finalmente, a Seção 8 apresenta a conclusão e possíveis trabalhos futuros.

2. Fundamentação Teórica

Esta seção apresenta uma fundamentação teórica sobre: Análise Estática de Código-Fonte, Ferramentas de Análise de Código Estático e *Massive Open Online Courses*.

2.1. Análise Estática de Código-Fonte

Análise de código estático (SCA, do inglês *Static Code Analysis*) é o tipo de análise de *software* realizada no código-fonte sem que seja necessário a execução do programa. Essa análise pode revelar possíveis vulnerabilidades, defeitos ou problemas de design em um estágio inicial da fase de desenvolvimento [Louridas 2006]. A SCA é significativamente mais rápida do que testes manuais, pois, assim como todas as formas de testes automatizados, ela garante que as verificações sejam realizadas consistentemente, fornecendo *feedback* rápido sobre as alterações mais recentes no código [Stefanović et al. 2020].

A análise estática de programas são utilizada desde o início da década de 1960 na otimização de compiladores. Mais recentemente, provou ser útil também para encontrar e verificar erros em ferramentas e em Ambientes de Desenvolvimento Integrado, (IDEs, do inglês *Integrated Development Environment*) no apoio de desenvolvimento de *software* [Moller and Schwartzbach 2012]. Os erros encontrados nessas análises ocorrem em programas sintaticamente válidos que podem ser compilados com sucesso. Eles representam problemas que possam ocorrer em um programa, mesmo que sua estrutura sintática siga as regras da linguagem de programação. Esses problemas são chamados de erros de análise estática por serem identificados por ferramentas de análise estática [Molnar et al. 2020].

2.2. Ferramentas de Análise de Código Estático

Ferramentas de análise de código estático são ferramentas desenvolvidas para realizar a inspeção contínua de código-fonte através da análise estática. Seu objetivo é gerar relatórios e identificar as violações e falhas causadas por más implementações, apontando desvios dos padrões de qualidade de códigos existentes, bem como uma estimativa aproximada da qualidade geral do *software* em questão [Andrade Gomes et al. 2017]. Essas ferramentas fornecem inúmeras métricas de qualidade de *software*, como: tamanho do código (Linhas de Código), complexidade (Complexidade média por método e complexidade média por classe), métricas CK, violações de convenções de codificação, entre diversas outras métricas [Tomas et al. 2013].

Em geral, as métricas fornecidas pelas ferramentas de análise de código estático indicam os seguintes tipos de problemas:

- *Bugs:* são erros de codificação que podem levar a um comportamento inesperado em tempo de execução. É recomendado que qualquer *bug* encontrado deve ser resolvido imediatamente [Andrade Gomes et al. 2017];
- *Vulnerabilidade*: são violações de segurança que se originam em erros de programação, que podem levar a incidentes de segurança, comprometendo a confiabilidade do *software* [Gasiba et al. 2021];
- *Code Smells*: são estruturas no código que indicam violação de princípios fundamentais de *design* que impactam negativamente em sua qualidade. *Code smells* geralmente não são *bugs*, o seu código aparenta estar tecnicamente correto e não impedem o funcionamento do programa. Entretanto, eles indicam fraquezas no código que podem retardar o desenvolvimento ou aumentar o risco de *bugs* ou falhas no futuro [Tufano et al. 2015].

Normalmente, as métricas fornecidas pelas ferramentas de análise são reportadas em relatórios, constituídos de uma lista de alerta. Estes avisos descrevem uma falha em potencial juntamente com uma série de recursos, como a linha de código correspondente, tipo e gravidade da falha [Yüksel and Sözer 2021]. Frequentemente, estas ferramentas suportam linguagens de programação populares no desenvolvimento de *software* e *plugins*. Além disso, as ferramentas não ajustam de forma automática os defeitos. A alteração do código é uma decisão dos desenvolvedores de *software* [Stefanović et al. 2020].

2.3. *Massive Open Online Courses*

Desde 2012, os *Massive Open Online Courses* (MOOCs) são considerados uma forma de educação disruptiva e transformadora que cresce exponencialmente pelo mundo [Hyman 2012]. O termo MOOC foi criado em 2008 para descrever um curso *online* aberto a ser oferecido pela Universidade de Manitoba, no Canadá [Liyanagunawardena et al. 2013]. Os MOOCs podem variar muito em formato, destacando-se algumas características definidoras, como:

- *Massive*: são cursos projetados para inscrição de um número ilimitado de participantes, dado que, se o número de participantes aumentar, nenhum esforço adicional será necessário para conduzi-lo [Mahajan et al. 2019];
- *Open*: o aspecto aberto dos MOOCs pode-se referir ao conceito de que geralmente são gratuitos e acessíveis a qualquer pessoa, não exigindo um processo formal de admissão [Blackmon and Major 2017];
- *Online*: os cursos são ministrados por meio de recursos *online* via Internet [Mahajan et al. 2019];
- *Course*: um curso completo é oferecido incluindo o plano de metas de aprendizagem e o material didático é disponibilizado na plataforma com o conteúdo do curso. Em geral, a avaliação dos participantes é formativa,[1] feita por meio de questionários, podendo ser empregado exames somativos para fins de certificação [Blackmon and Major 2017];

Dado a essas características, os MOOCs vêm se tornando um fenômeno global que está transformando o ensino e o aprendizado com plataformas *online* como, Coursera, Udemy, Udacity, Alura [Hyman 2012]. Em dezembro de 2021 e de acordo com a *Class Central*,[2] 220

[1]Disponível em: `https://eric.ed.gov/?id=ED123239`. Ultimo acesso: 25 out. 2022

[2]Disponível em: `https://www.classcentral.com/report/mooc-stats-2021`. Ultimo acesso: 15 out. 2022

milhões de alunos relataram seu registro em um dos mais de 19,4 mil cursos dentre as 950 plataformas. Sendo isso, mais que o dobro de alunos e cursos em comparação aos 4 anos anteriores, considerando que em 2017 foram registrados 81 milhões de alunos em 9,4 mil cursos.

3. Trabalhos Relacionados

Esta seção explora e discute cinco artigos relacionados ao tema de qualidade de código envolvendo educadores de *software* ou, cuja abordagens e técnicas utilizadas pelos autores inspiram o presente trabalho.

Nesse contexto, Kirk et al. (2020) propuseram um estudo investigativo visando compreender e avaliar o conhecimento técnico dos professores no que tange às boas práticas em qualidade de código, e ainda, compreender, sob a ótica destes professores, os fatores que afetam a dinâmica e a eficácia dos *feedbacks* aos seus discentes. Foram realizadas diversas perguntas abertas sobre o tema, a 8 professores entrevistados. Além disso, esses professores foram solicitados a avaliar a qualidade de código em exemplos de exercícios de alunos. Como resultado, observou-se que alguns professores não se atentaram em algumas correções que faziam, ou então, não possuíam conhecimento suficiente para avaliar a qualidade do código ali presente. Dessa forma, foi possível concluir que alguns alunos poderiam estar recebendo informações incorretas através do *feedback* fornecido por determinados docentes . Tal artigo assemelha-se a este estudo proposto, pois ambos possuem como objetivo avaliar o conhecimento técnico de professores em relação qualidade de código-fonte, todavia, neste espera-se que a avaliação seja realizada através de análise estática de código (SCA).

De maneira análoga, Boutnaru e Hershkovitz (2015) analisaram programas de computadores de alunos e professores através de exercícios produzidos em um treinamento de segurança cibernética. A fim de explorar as diferenças na aprendizagem dessas populações, os autores compararam medidas de qualidade e segurança de *software* por meio de SCA. Foram investigados trechos de código escritos em Python desenvolvidos por 18 professores e 31 alunos. Os resultados indicaram que os códigos dos professores participantes tiveram qualidade superior em relação ao código dos alunos, no entanto, em relação aos recursos de segurança, o oposto foi observado. A pesquisa é relevante para este trabalho, pois ambas concentram-se em código produzido por professores, examinados através da perspectiva de análise estática. Além disso, espera-se também observar as más práticas em segurança de *software* através dos alertas de vulnerabilidades fornecidos pelos relatórios das ferramentas de análise.

Gomes et al. (2021) definiram propostas na melhoria do processo de ensino de programação em *software*. Eles forneceram orientações em conteúdos e materiais oferecidos aos professores, elaborados com base nas dificuldades encontradas pelos alunos, com a observância de violações de qualidade submetidas por eles em exercícios. Para análise dos códigos, foram utilizadas as ferramentas SonarQube e TeacherMate, ambas as ferramentas inspecionam código estaticamente. Como resultado do estudo, observaram determinados conjuntos de regras que mais sofreram violações dos alunos, destacando-se problemas relacionados a: *Data Input/Output*, *Loggers*, *StackTrace*, *Exceptions* e *Throwable*, *Garbage Colletion* e *Constructors*. De forma similar, este trabalho pretende utilizar os relatórios obtidos através da ferramenta SonarQube como artefato para descobrir quais violações de qualidade são mais frequentes pelos educadores. Como um de seus objetivos, espera-se também obter um conjunto de regras que mais sofreram violações. Assim como no estudo de Gomes et al. (2022),

esta abordagem servirá de insumo para educadores, possibilitando direcionar as atividades fornecidas em aulas, aspirando apenas as boas práticas de codificação.

Birillo et al. (2022) apresentaram a ferramenta Hyperstyle, utilizada para fornecer feedback detalhado sobre a qualidade do código de soluções de programação. Desenvolvida principalmente para integração em plataformas MOOC, a ferramenta tem como objetivo alvo as linguagens de programação: Python, Java, Kotlin e JavaScript. Como estudo da eficiência da ferramenta em comparação com outras existentes do mercado, foi realizada uma análise empírica comparativa de 250 mil soluções para cada um das quatro linguagens suportadas. Por fim, foi constatado que o Hyperstyle detecta centenas problemas de qualidade de código divididas em cinco categorias. A pesquisa faz-se relevante para este estudo, pois ambas têm como objetivo a análise de código em plataformas MOOC, além de ter como alvo as linguagens de programação Python, Java e JavaScript. Espera-se também, detectar problemas de qualidade em categorias semelhantes ao do trabalho citado.

Andrade Gomes et al. (2017) apresentam a ferramenta SMaRT (*Software Maintenance, Report and Tracker*) cujo objetivo é apoiar os alunos a melhorar seu código-fonte e, consequentemente, suas habilidades de programação. A ferramenta proposta utiliza relatórios de qualidade do SonarQube Scanner, a fim de obter medidas sobre o código-fonte e, então, apresentá-las aos alunos como forma de *feedback* de suas tarefas. Foi realizado um estudo piloto de um experimento controlado, onde, alunos foram solicitados a realizar tarefas de manutenção em um código-fonte predefinido que possuíam alguns defeitos comuns. Foi comprovado que a ferramenta SMaRT tem potencial em auxiliar os alunos a entender o que é um código-fonte com qualidade. A metodologia utilizada no presente trabalho se assemelha com a proposta de avaliação desse estudo, uma vez que a análise de qualidade é realizada através de relatórios da ferramenta SonarQube. No entanto, o trabalho se difere a este estudo, ao ter como objeto de estudo o código de educadores.

4. Materiais e Métodos

O estudo proposto neste trabalho é de caráter quantitativo com objetivo exploratório [Wohlin et al. 2012], onde, identificou-se a existência de más práticas em código-fonte produzidos por educadores. Para isso, analisou-se esta qualidade por meio de coleta de dados com aplicação de métodos estatísticos. Em complemento a esta justificativa, esta metodologia se comprovou eficaz em trabalhos relacionados [Boutnaru and Hershkovitz 2015, Gomes et al. 2021, Birillo et al. 2022]. Os materiais e métodos empregados neste trabalho baseiam-se nos conceitos e estudos relacionados, que estão descritos na Seção 3 . As subseções seguintes detalham os procedimentos adotados para execução do estudo.

4.1. Procedimentos

Extração, Transformação e Carga (ETL, do inglês *Extract Transform Load*) são procedimentos de uma técnica de *Data Warehouse* (DW), sendo um processo chave para reunir todos os dados em um ambiente padrão e homogêneo, modelando os dados relevantes do sistema de origem em informações úteis a serem armazenadas [Gour et al. 2010]. Baseado nesta técnica, os dados utilizados neste estudo são extraídos de um sistema-fonte, convertidos e formatados para análises e persistidos em um banco de dados. Por fim, para este estudo, faz-se necessário a realização de análises dos dados obtidos a fim de se alcançar os resultados dos objetivos propostos, dessa forma, adiciona-se ao final do fluxo ETL a etapa de análise de dados. A

demonstração desse fluxo completo é apresentada na Figura 1 e seu detalhamento é descrito nas subseções a seguir.

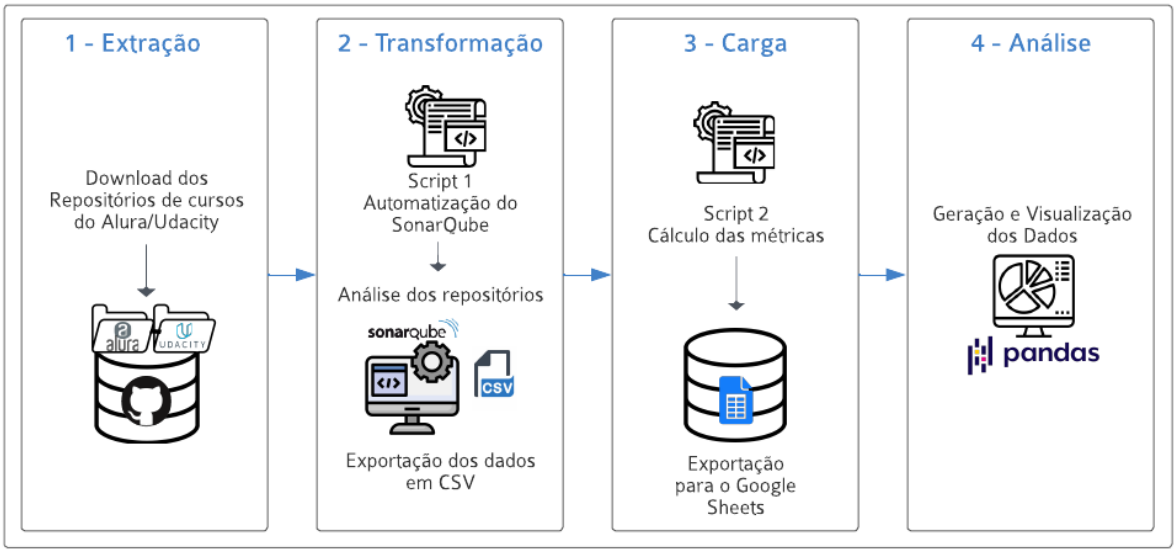


Figura 1. Fluxograma Metodológico

4.1.1. Extração dos Dados

Nesta pesquisa, estudaram-se cursos de programação em linguagens relevantes e expressivas no cenário atual de desenvolvimento. Consideram-se as linguagens Javascript, Java, Python, Typescript e Go, sendo essas encontradas entre as mais populares no *rankings* do GitHub[3] e TIOBE[4]. Além disso, estas linguagens de programação são alvo de estudos recentes sobre qualidade de código [Lopes and Hora 2022]. O código-fonte dos educadores foram extraídos das plataformas de cursos Alura,[5] e do Github oficial da plataforma Udacity[6]. Visto que os cursos visam capacitar alunos, considera-se os mesmos como cursos voltados para iniciantes no conteúdo ministrado. Nos próximos parágrafos, discutem-se as principais características das plataformas estudadas.

Alura: Como uma das maiores plataformas MOOC de ensino de programação paga no Brasil, a Alura pertencente ao grupo Caelum, fundada em 2011 e atualmente possui em seu catálogo mais de 1400 cursos voltados para a área digital, além de possuir mais de 70 mil alunos ativos.[7] Para obtenção do código-fonte dos cursos na plataforma, buscaram-se aqueles: (i) cursos categorizados como programação; (ii) com o código final do projeto disponível para *download*; (iii) que possuam como linguagem principal as selecionadas anteriormente para o estudo, sendo em sua forma base ou *frameworks*.

[3]Ranking GitHub de linguagens populares de 2022 disponível em: https://octoverse.github.com/2022/top-programming-languages. Ultimo acesso: 22 fev. 2023

[4]Ranking TIOBE de linguagens populares de 2023 disponível em: https://www.tiobe.com/tiobe-index/. Ultimo acesso: 22 fev. 2023

[5]Disponível em: https://cursos.alura.com.br/courses. Ultimo acesso: 25 out. 2022

[6]Disponível em: https://github.com/udacity. Ultimo acesso: 15 abr. 2022

[7]Disponível em: https://www.alura.com.br/empresas/sobre-nos. Ultimo acesso: 25 out. 2022

Udacity: A Udacity foi fundada em 2012 por Sebastian Thrun, David Stavens e Mike Sokolsky, professores da Universidade de Stanford.[8] Desde então, a plataforma tem crescido exponencialmente e já conta com mais de 11 milhões de usuários em todo o mundo. Os cursos oferecidos pela Udacity são criados em colaboração com grandes empresas de tecnologia, como Google, Facebook, IBM e Amazon. Para obtenção do código-fonte dos cursos na plataforma, buscaram-se repositórios que: (i) possuem código disponibilizado durante o curso ou *templates* de exercício disponibilizado para os alunos; (ii) possuem como linguagem principal Javascript, Java, Python, Typescript ou Go.

4.1.2. Transformação dos Dados

A segunda etapa constituí-se no desenvolvimento de um *script* em Python, onde, possui como propósito a automatização das análises estáticas dos repositórios dos cursos clonados na etapa anterior. Esta automatização foi utilizada para todos os cursos das linguagens, em exceção a Java, onde foi necessário realizar a integração manualmente. Para a análise, realiza-se a integração do código do curso com a ferramenta SonarQube (SQ). Desenvolvida pela SonarSource, a plataforma SonarQube realiza inspeções no código-fonte, identificando violações na qualidade, conforme descrito na Seção 2.2 deste artigo. [9] Esta plataforma foi desenvolvida como um conjunto de ferramentas *open source* que vem sendo utilizadas em diversos projetos, como PMD, Findbugs, Checkstyle, entre outros [Campbell and Papapetrou 2013].

Este estudo concentra-se nos seguintes grupos de métricas da ferramenta SonarQube:

- *Manutenibilidade*: baseado na Dívida Técnica (DT) do projeto, sendo o esforço (tempo) estimado para corrigir todos os *code smells* encontrados. Através do DT, encontramos o Índice de Dívida Técnica (IDT), sendo a relação entre o custo para desenvolver o *software* e o custo para corrigi-lo.
- *Confiabilidade*: baseia-se no número de problemas do tipo *bugs* encontrados no código;
- *Segurança*: se baseia no número de problemas do tipo vulnerabilidade, ou seja, locais suspeitos no código que indicam possíveis falhas de segurança;
- *Tamanho*: essas métricas descrevem o tamanho do código e como ele é comentado.

As *issues* do SonarQube são classificadas em categorias, de acordo com sua gravidade:

- *Blocker*: problemas graves que impedem o funcionamento correto do *software* ou afetam diretamente a segurança do sistema;
- *Critical*: problemas que podem causar falhas graves no *software* ou afetar negativamente a segurança do sistema;
- *Major*: problemas que afetam a qualidade do código e podem levar a problemas operacionais, mas não são tão graves como os classificados como críticos ou bloqueadores;
- *Major*: problemas menores de qualidade de código que geralmente não afetam a funcionalidade do *software*, mas podem ser corrigidos para melhorar a qualidade do código;
- *Info*: problemas de baixa gravidade, porém é importante corrigi-los, uma vez que eles podem se acumular ao longo do tempo e afetar negativamente a qualidade geral do código.

[8]Disponível em: https://www.udacity.com/us. Ultimo acesso: 8 mai. 2023.

[9]Lista das regras detectadas pela ferramenta SonarQube, disponível em: `https://rules.sonarsource.com/`, Ultimo acesso: 5 jun. 2023

A ferramenta além de informar o número de violações, seu tipo e gravidade, fornece também a classificação para cada um dos grupos de métricas. Sendo, de A a E, onde A é pontuação mais alta. A definição de classificação apresentada pelo SonarQube está descrita na Tabela 1, assim como, mais detalhadamente, na documentação oficial da ferramenta.[10]

Tabela 1. Classificação de problemas de código por categoria (SonarQube)

Nota	Classificação de Confiabilidade	Classificação de Segurança	Classificação de Manutenibilidade
A	Nenhum *bug*	Nenhuma vulnerabilidade	Índice de Dívida Técnica $< 5\%$
B	Pelo menos um *bug (minor)*	Pelo menos uma vulnerabilidade *(minor)*	Índice de Dívida Técnica $< 10\%$
C	Pelo menos um *bug (major)*	Pelo menos uma vulnerabilidade (major)	Índice de Dívida Técnica $< 20\%$
D	Pelo menos um *bug (critical)*	Pelo menos uma vulnerabilidade*(critical)*	Índice de Dívida Técnica $< 50\%$
E	Pelo menos um *bug (blocker)*	Pelo menos uma vulnerabilidade (*blocker)*	Índice de Dívida Técnica $>50\%$

Após realizado a análise do código do curso, exporta-se o relatório fornecido para o formato CSV (*Comma-separated values*), onde, para cada repositório, obteremos os seguintes atributos que são utilizados para atingir os objetivos específicos propostos: (i) nome do repositório/curso; (ii) numero de linhas de código (LOC); (iii) linguagem primária do repositório; (iv) lista de violações encontradas do tipo *bugs*, vulnerabilidades, *code smells*, sendo que para cada violação seja definido sua descrição e gravidade; e (v) classificação de qualidade dos critérios de confiabilidade, segurança e manutenibilidade.

Além desses atributos, para resolução do terceiro objetivo específico deste trabalho, para cada curso, se faz necessário a coleta do número de alunos inscritos. No final desta etapa são executadas a limpeza e filtragem dos dados obtidos, removendo àqueles que são irrelevantes para o estudo.

4.1.3. Persistência e Análise dos Resultados

Na terceira etapa, tem-se como objetivo a persistência das métricas e atributos em uma planilha. Neste estudo, tal etapa visa facilitar a manipulação dos dados obtidos, aumentando também sua confiabilidade e acessibilidade. A etapa constitui no desenvolvimento e execução do segundo *script* em Python, nele as métricas são calculadas e adicionadas na planilha.

Por fim, para solucionar o problema proposto, realizamos a análise em cima dos dados coletados. Dessa forma, na etapa final, utiliza-se a ferramenta *Pandas* para plotagem de gráficos, melhorando assim visualização das informações. Pretende-se comparar e encontrar relações entre a qualidade do código dos educadores, através de sua classificação nas métricas de qualidade, linguagens de ensino, plataforma, tamanho e popularidade do curso.

[10]Disponível em: https://docs.sonarqube.org/latest/user-guide/metric-definitions/. Ultimo Acesso: 30 out. 2022.

5. Resultados

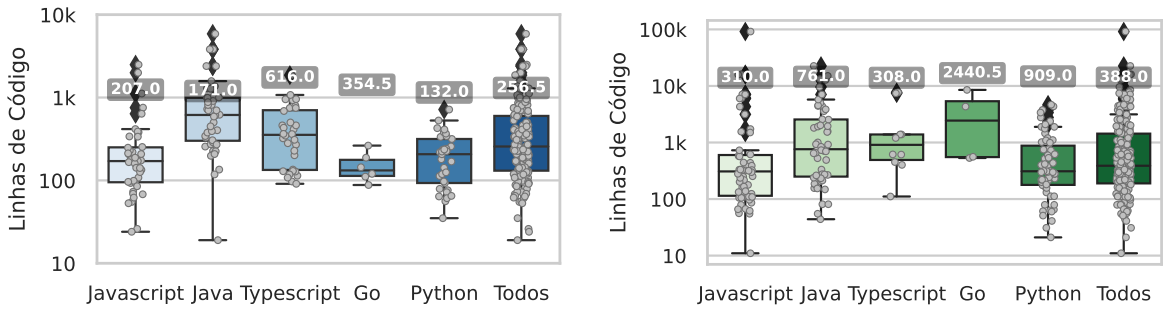
Nesta seção, apresenta-se os resultados obtidos. A Seção 5.1 inclui a caracterização do conjunto de dados, enquanto as subseções seguintes apresentam as questões de pesquisa.

5.1. Caracterização do Conjunto de Dados

A caracterização dos dados coletados é realizada pela elaboração de gráficos *boxsplot* na escala logarítmica, que permitem a visualização da distribuição dos valores obtidos (Figura 2). Em resumo, descreve-se o tamanho dos 352 cursos, provenientes das plataformas Alura e Udacity.

Cursos Alura: Seguindo os critérios descritos na Seção 4.1.1, analisam-se 162 projetos da plataforma Alura (43 Javascript, 32 TypeScript, 48 Java, 33 Python e 6 em Go). A Figura 2(a) apresenta a distribuição do tamanho dos projetos, considerando o número de linhas de código por linguagem de programação. Avaliando todas as linguagens, a mediana é igual a 256, sendo que 75% dos cursos possuem até 600 linhas. O curso com o menor tamanho é "Maven: Gerenciamento de dependências e build" com 19 linhas, enquanto o curso com a maior quantidade de linhas é "Kafka: Fast deletate, evolução e cluster de brokers" com 5.866 linhas na linguagem Java. Ao todo, foram obtidos 2.523 arquivos das extensões dessas linguagens, sendo Java a linguagem com o maior número de arquivos (1.286), seguida por TypeScript (510), JavaScript (418), Python (282) e Go (27).

Cursos Udacity: Considerando todas as linguagens da plataforma Udacity, a mediana da quantidade de linhas de código é de 388, sendo que 75% dos cursos possuem projetos com até 1.435 linhas. O projeto de menor tamanho é "Hashing Code Exercise", com apenas 11 linhas de código, enquanto o maior é "Building High Conversion Web Form", com 91.651 linhas de código, ambos desenvolvidos utilizando Javascript. Além disso, observa-se que a mediana do número de linhas varia de acordo com a linguagem utilizada (Figura 2(b)): 308 em Typescript, 909 em Python, 761 em Java, 310 em Javascript e 2440 em Go. No total, analisa-se 8.807 arquivos. Existem 5.063 arquivos referentes à linguagem de programação Java, 1.910 arquivos JavaScript, 870 arquivos Python, 821 em TypeScript, e 143 arquivos em Go.



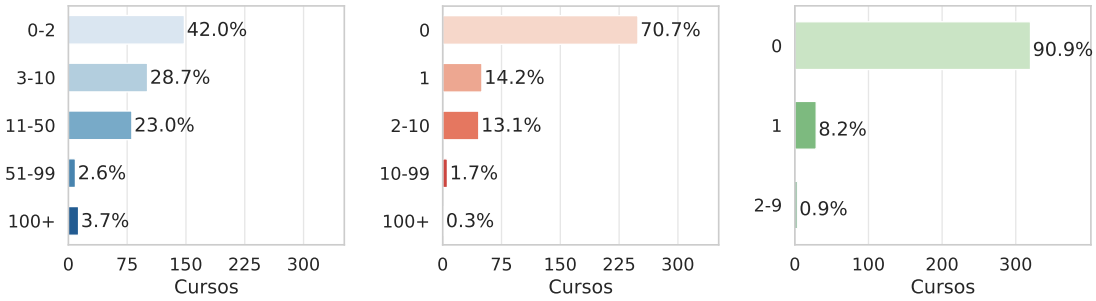
(a) *Alura* (b) *Udacity*

Figura 2. Distribuição da quantidade de linhas de código por linguagem

5.2. Frequência de problemas de código em plataformas MOOC

Para responder a primeira questão de pesquisa, analisa-se a taxa de problemas de código por curso. A Figura 3 mostra a frequência de problemas de código por categoria: *code smells*, *bugs*, e vulnerabilidades. Para avaliar a distribuição dos problemas encontrados, foram gerados

três gráficos de barras para representar a quantidade de problemas por tipo. Nota-se que a maioria dos cursos avaliados (42%) apresenta entre zero e dois *code smells*, enquanto, cerca de 29% dos cursos apresentaram entre três e dez. A maioria dos cursos avaliados (70,7% e 90,9%, respectivamente) não apresentou problemas de *bugs* e vulnerabilidades. No entanto, uma parcela significativa de cursos teve apenas uma ocorrência desses problemas: 50 cursos com um único *bug* (14,20%) e 29 cursos com uma única vulnerabilidades (8,23%).



(a) *Code Smells* (b) *Bugs* (c) *Vulnerabiliadde*

Figura 3. Quantidade de problemas de código por curso

Dentre os 352 projetos, existem 7.854 problemas de código, sendo 1.286 identificados na plataforma Alura (16,37%) e 6.568 na plataforma Udacity (83,62%). Nos próximos parágrafos, discute-se os resultados por plataforma.

Cursos Alura: A Tabela 2 exibe a frequência de problemas identificados nos cursos da plataforma Alura. Observa-se que os *code smells* foram detectados em 70,99% dos cursos (com 1.157 ocorrências de problemas na categoria). Destaca-se que somente os cursos ministrados nas linguagens Java e Typescript apresentaram vulnerabilidades. Assim, das 162 avaliações realizadas, apenas 18 cursos (11,11%) foram identificados com algum tipo de vulnerabilidade, sendo que nenhum deles apresentou mais de uma vulnerabilidade. A linguagem Go apresenta o maior percentual de cursos com *bugs* (87,5%), distribuídos em cinco projetos. O curso *Python Collections parte 2: conjuntos e dicionários* apresenta a maior taxa de ocorrências de *bugs* (23 ocorrências), correspondendo a 69,69% dos *bugs* encontrados na linguagem.

Tabela 2. Frequência de problemas de código (Alura)

Linguagem	N.º de cursos	Bugs		Vulnerabilidade		Code Smells	
		Ocorr.	%	Ocorr.	%	Ocorr.	%
Go	6	5	87,50	0	0,00	4	16,67
Java	48	37	41,67	16	33,33	412	79,17
Javascript	43	17	13,95	0	0,00	480	81,40
Python	33	33	21,21	0	0,00	193	57,58
Typescript	32	19	40,63	2	6,25	68	68,75
Total	162	111	31,48	18	11,11	1.157	70,99

O Listing 1 mostra um exemplo de problema de código, onde o trecho contém um *code smell* de gravidade *major*.[11] Essa regra procura identificar blocos de código que podem ser simplificados para melhorar a legibilidade e manutenibilidade.

[11]Disponível em: `https://rules.sonarsource.com/python/type/Code\%20Smell/RSPEC-1066`

```
calcular_regras(self):
    col = self.pacman.coluna_intencao
    lin = self.pacman.linha_intencao
    if 0 <= col < 28 and 0 <= lin < 29:
        if self.matriz[lin][col] != 2:
            self.pacman.aceitar_movimento()
```

Listing 1. Exemplo de problema de código (Code Smell em Python)

Cursos Udacity: A Tabela 3 apresenta a frequência dos problemas encontrados na plataforma Udacity. A linguagem Go não apresenta ocorrências de *bugs* e vulnerabilidades. No entanto, todos os cursos da linguagem Go apresentaram problemas de Code Smell (279 ocorrências). Java e Javascript possuem um número significativo de ocorrências de *bugs*, 73 e 270, respectivamente, o que representa mais de 30% dos cursos avaliados em ambas as linguagens. Em Python, existem 2.725 ocorrências de *code smells* (95,24%).

Tabela 3. Frequência de problemas de código (Udacity)

Linguagem	N.º de cursos	Bugs		Vulnerabilidade		Code Smells	
		Ocorr.	%	Ocorr.	%	Ocorr.	%
Go	4	0	0,00	0	0,00	279	100,00
Java	55	73	34,55	13	21,82	998	72,73
Javascript	58	270	34,48	12	3,45	2.003	86,21
Python	63	55	15,87	0	0,00	2.725	95,24
Typescript	10	3	30,00	0	0,00	137	90,00
Total	190	401	27,37	25	7,37	6.142	85,79

5.3. Gravidade dos problemas de código em plataformas MOOC

Para responder a segunda questão de pesquisa, analisam-se as métricas de confiabilidade, segurança, e manutenibilidade, considerando a classificação de qualidade disponibilizada pela ferramenta SonarQube.

Cursos Alura. A Tabela 4 mostra a classificação dos problemas na plataforma Alura. Observa-se que a maioria dos cursos na linguagem Go possuem a nota C em Confiabilidade (83.3%, 5 cursos). Assim como Python, todos os cursos em Go receberam boas classificações em relação métricas de segurança e manutenibilidade (Nota A). Javascript é a linguagem com maior porcentagem de notas A (37 cursos, 86%). Por fim, vale destacar que, embora cada linguagem tenha suas características próprias, Javascript e Python apresentaram o maior percentual de cursos com boas notas nas três métricas analisadas.

Tabela 4. Classificação de qualidade por tipo de problema de código (Alura)

Linguagem	Bugs Confiabilidade (%)					Vulnerabilidades Segurança (%)					Code Smell Manutenabilidade (%)				
	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
Go	16,7	0,0	83,3	0,0	0,0	100	0,0	0,0	0,0	0,0	100	0,0	0,0	0,0	0,0
Java	58,3	8,3	25,0	4,2	4,2	66,7	12,5	4,2	10,4	6,3	72,9	20,8	4,2	2,1	0,0
Javascript	86,0	0,0	9,3	4,7	0,0	100	0,0	0,0	0,0	0,0	93,0	7,0	0,0	0,0	0,0
Python	78,8	0,0	21,2	0,0	0,0	100	0,0	0,0	0,0	0,0	100	0,0	0,0	0,0	0,0
Typescript	56,3	6,3	31,3	0,0	6,3	93,8	0,0	0,0	0,0	6,3	100	0,0	0,0	0,0	0,0

Cursos Udacity. A Tabela 5 apresenta a classificação dos problemas de código na plataforma Udacity. Pode-se observar que a maioria dos cursos em Go e Python possuem classificação A. Em Typescript, 20% de seus cursos tiveram uma nota intermediária nas métrica de confiabilidade. Assim como na plataforma Alura, pela Udacity a linguagem Java é a que possui mais pontos fracos, apresentando notas mais baixas em todas as três métricas avaliadas.

Tabela 5. Classificação de Qualidade por tipo de problema de código (Udacity)

Linguagem	Bugs Confiabilidade (%)					Vulnerabilidades Segurança (%)					Code Smell Manutenabilidade (%)				
	A	**B**	**C**	**D**	**E**	**A**	**B**	**C**	**D**	**E**	**A**	**B**	**C**	**D**	**E**
Go	100	0,0	0,0	0,0	0,0	100	0,0	0,0	0,0	0,0	100	0,0	0,0	0,0	0,0
Java	65,5	7,3	18,2	3,6	5,5	78,2	1,8	1,8	12,7	5,5	61,8	14,5	9,1	7,3	7,3
Javascript	72,4	0,0	20,7	1,7	5,2	94,8	1,7	0,0	0,0	3,4	98,3	1,7	0,0	0,0	0,0
Python	88,9	0,0	9,5	0,0	1,6	100,0	0,0	0,0	0,0	0,0	98,4	1,6	0,0	0,0	0,0
Typescript	80,0	0,0	20,0	0,0	0,0	100,0	0,0	0,0	0,0	0,0	100	0,0	0,0	0,0	0,0

5.4. Relação entre a popularidade e qualidade dos cursos em plataformas MOOC

Por fim, na terceira questão de pesquisa, analisa-se a correlação entre a popularidade e a quantidade de problemas de código detectados nos cursos. A análise concentra-se em cursos da plataforma Alura, visto que a mesma disponibiliza informações sobre o número de alunos matriculados, mesmo atributo utilizado em estudos anteriores [Qaralleh and Darabkh 2015]. Especificamente, obtém-se $rho = 0{,}2503$ e $p - value = 0{,}0013$, sugerindo uma correlação insignificante entre as variáveis [Borges and Valente 2018, Hinkle et al. 2003].

A Figura 4 apresenta o gráfico de dispersão gerado. Pode-se observar a grande concentração dos pontos é encontrada em valores baixo, tanto de número de *issues*, quanto em número de alunos do curso.

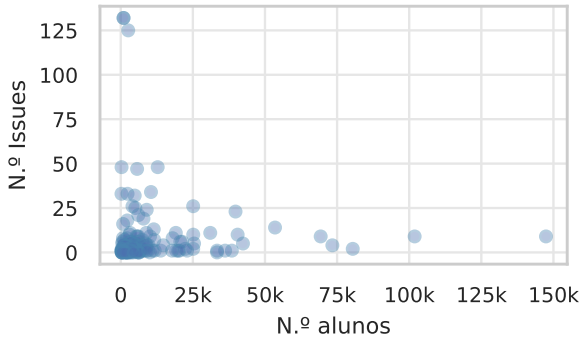


Figura 4. Dispersão dos problemas de código por popularidade dos cursos

6. Discussão dos Resultados

Nesta seção são apresentas discussões geradas pelas questões respondidas neste estudo.

Os problemas de código mais frequentes em plataformas MOOC referem-se à *code smells*. Dentre os 352 cursos analisados, apenas 37 deles não apresentaram nenhum tipo de problema nas três categorias (*bugs*, *code smells* e vulnerabilidades). Identificaram-se mais

problemas de código relacionados com *code smells*. De fato, *code smells* são problemas recorrentes e menos graves [Pereira et al. 2018], sendo alvo de pesquisas recentes em outros contextos [Bibiano et al. 2019, Cedrim et al. 2017]. Embora um grande número de cursos apresente problemas de código, é importante destacar que a taxa de problemas por curso é baixa. Por exemplo, 51,98% dos cursos possuem até 5 problemas de código. Entretanto, os resultados sugerem que os autores de cursos podem melhorar a qualidade dos projetos, fazendo uso, por exemplo, de ferramentas de análise da qualidade do código-fonte, como *SonarQube*. Dessa forma, os resultados inspiram futuras linhas de pesquisa sobre a adoção de *linters* e outras ferramentas para a análise estática de código por educadores em plataformas MOOC.

A maioria dos problemas de código em plataformas MOOC possuem baixa gravidade. Mais de 50% dos cursos em todas as linguagens receberam a classificação A em todas as categorias, sugerindo que os problemas encontrados possuem uma baixa gravidade. Por exemplo, em Python, todos os cursos receberam classificação A em relação a problemas de segurança. Entretanto, observou-se a ocorrência de problemas de níveis intermediários, por exemplo, em cursos Java. É importante ressaltar que o impacto dos problemas pode variar de acordo com o contexto e o uso da linguagem de programação. Falhas de vulnerabilidade, por exemplo, podem causar danos significativos [Iannone et al. 2023]. Dessa forma, educadores podem se beneficiar também de ferramentas de análise de qualidade de código para mitigar problemas graves, que podem impactar os estudantes em plataformas MOOC.

Frequência de problemas em sistemas Java. Os sistemas Java possuem problemas de código em todas as categorias nas plataformas Alura e Udacity (A até E). Este resultado pode estar relacionado com a sua popularidade e nível de maturidade, visto que é uma linguagem reconhecida e amplamente adotada. SonarQube, por exemplo, define 656 regras para a linguagem Java, enquanto apenas 38 regras são definidas para a linguagem Go.

Correlação entre a popularidade e qualidade dos cursos em Plataformas MOOC. A análise de correlação entre problemas de código e popularidade não mostrou resultados significativos, isto é, a variação do número de alunos não impacta necessariamente na qualidade do código-fonte disponibilizado nos cursos. Por exemplo, o curso "Javascript e HTML: desenvolva um jogo e pratique lógica de programação" apresentou poucos problemas, apesar de ter muitos alunos matriculados. Por outro lado, o curso "React com Javascript: lidando com arquivos estáticos" teve muitos problemas, mesmo tendo um número relativamente baixo de alunos.

7. Ameaças à Validade

Nesta seção, apresenta-se as ameaças à validade deste estudo e as estratégias utilizadas para mitigá-las [Wohlin et al. 2012]. Primeiro, como usual em estudos empíricos em Engenharia de *Software*, os resultados não podem ser generalizados para outros cenários. Entretanto, o trabalho inclui 352 cursos em duas plataformas MOOC populares, abrangendo cinco linguagens de programação distintas. Especificamente, analisa-se 11.315 arquivos.

Neste estudo, utilizou-se a configuração padrão da ferramenta SonarQube, que inclui um conjunto predefinido de regras. Portanto, existe a possibilidade de falsos positivos. Entretanto, a ferramenta é popular no contexto de análise estática de código [Vassallo et al. 2018, Marcilio et al. 2019]. Conforme mostrado no Apêndice A, visando mitigar falsos positivos, removeu-se *tags* que não se aplicam ao contexto deste trabalho. Por exemplo, foram removi-

das *tags* relacionadas a uma versão específica da linguagem de programação, como problemas relacionados com a *tag* $java17$.[12] Entretanto, essas *issues* específicas de versões representam apenas uma pequena fração dos resultados.

Por fim, existe a possibilidade de que alguns arquivos do curso sejam dependências externas. Para atenuar esta ameaça, pacotes usualmente utilizados para versionar bibliotecas externas foram removidos, como as pastas $node\_modules$, lib, $site\_packages$ e $vendor$. Além disso, consideraram-se apenas os arquivos nas extensões específicas de linguagem de programação, como $.js$, $.py$, $.ts$, $.go$ e $.java$.

8. Conclusão

Este estudo teve como objetivo investigar a presença de problemas de código em cursos de programação oferecidos por plataformas de ensino *online*. Ao todo, analisaram-se aproximadamente 400 cursos das plataformas Alura e Udacity, totalizando 11.315 arquivos. Especificamente, o trabalho concentrou-se em cursos baseados nas linguagens de programação Java, Javascript, Typescript, Python e Go. A análise de qualidade do código foi realizada através da ferramenta SonarQube. Apresenta-se a seguir os principais resultados:

- A maioria dos problemas de código referem-se à *code smells*. Especificamente, detectou-se aproximadamente 8 mil ocorrências: 7.299 *code smells* (92,93%), 512 *bugs* (6,51%), e 43 vulnerabilidades (0,40%).
- Apenas 37 cursos (10,51%) não apresentaram nenhum tipo de problema, isto é, sem *bugs*, *code smells* e vulnerabilidades.
- Em relação às linguagens de programação, detectou-se a maioria dos problemas em cursos Python (38,27%), seguidos por Javascript (35,42%) e Java (19,72%).
- A maioria dos problemas de código encontrados não são graves, mostrando que a qualidade do código disponibilizado nessas plataformas pode ser considerada satisfatória para os estudantes que utilizam os cursos online para aprendizado. Por exemplo, no caso de JavaScript, apenas 5.94% (6 ocorrências) dos cursos possuem a métrica de confiabilidade as classificações D ou E.

Como trabalhos futuros, sugere-se realizar um estudo qualitativo para aprofundar a análise dos problemas de código apresentados nesta pesquisa. Por exemplo, *surveys* com estudantes para compreender o impacto dos problemas de código. Além disso, sugerem-se estudos considerando a perspectiva dos educadores, sobre o uso de ferramentas de análise estática de código para garantir a qualidade dos cursos. Também é possível utilizar outras ferramentas de análise da qualidade de código, avaliando, por exemplo, categorias distintas de problemas além de *code smells*, vulnerabilidades e *bugs*.

Pacote de Replicação

O pacote de replicação deste trabalho encontra-se disponível em:

`https://github.com/ICEI-PUC-Minas-PPLES-TI/plf-es-2022-2-tcci-5308100-pes-otavio-guimaraes`

[12]Disponível em: `https://rules.sonarsource.com/java/tag/java17`

Referências

Andrade Gomes, P. H., Garcia, R. E., Spadon, G., Eler, D. M., Olivete, C., and Correia, R. C. M. (2017). Teaching software quality via source code inspection tool. In *2017 IEEE Frontiers in Education Conference (FIE)*, pages 1–8. Ieee.

Bey, A., Jermann, P., and Dillenbourg, P. (2018). A comparison between two automatic assessment approaches for programming: An empirical study on moocs. *Journal of Educational Technology & Society*, 21(2):259–272.

Bibiano, A. C., Garcia, E. F. D. O. A., Kalinowski, M., Fonseca, B., Oliveira, R., Oliveira, A., and Cedrim, D. (2019). A quantitative study on characteristics and effect of batch refactoring on code smells. In *13th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11.

Birillo, A., Vlasov, I., Burylov, A., Selishchev, V., Goncharov, A., Tikhomirova, E., Vyahhi, N., and Bryksin, T. (2022). Hyperstyle: A tool for assessing the code quality of solutions to programming assignments. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, pages 307–313.

Blackmon, S. and Major, C. (2017). Wherefore art thou mooc: Defining massive open online courses. *Online Learning Journal*, 21(4).

Borges, H. and Valente, M. T. (2018). What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112–129.

Börstler, J., Caspersen, M. E., and Nordström, M. (2007). Beauty and the beast–toward a measurement framework for example program quality.

Boutnaru, S. and Hershkovitz, A. (2015). Software quality and security in teachers' and students' codes when learning a new programming language. *Interdisciplinary Journal of E-Skills and Lifelong Learning*, 11:123–148.

Campbell, G. A. and Papapetrou, P. P. (2013). *SonarQube in action*. Manning Publications Co.

Cedrim, D., Garcia, A., Mongiovi, M., Gheyi, R., Sousa, L., de Mello, R., Fonseca, B., Ribeiro, M., and Chávez, A. (2017). Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In *11th International Symposium on the Foundations of Software Engineering (FSE)*, pages 465–475.

Gasiba, T. E., Hodzic, S., Lechner, U., and Pinto-Albuquerque, M. (2021). Raising security awareness using cybersecurity challenges in embedded programming courses. In *2021 International Conference on Code Quality (ICCQ)*, pages 79–92. IEEE.

Gomes, P. H., Garcia, R. E., Eler, D. M., Correia, R. C., and Junior, C. O. (2021). Software quality as a subsidy for teaching programming. In *2021 IEEE Frontiers in Education Conference (FIE)*, pages 1–9. IEEE.

Gour, V., Sarangdevot, S., Tanwar, G. S., and Sharma, A. (2010). Improve performance of extract, transform and load (etl) in data warehouse. *International Journal on Computer Science and Engineering*, 2(3):786–789.

Hinkle, D. E., Wiersma, W., and Jurs, S. G. (2003). *Applied statistics for the behavioral sciences*, volume 663. Houghton Mifflin college division.

Hyman, P. (2012). In the year of disruptive education. *Communications of the ACM*, 55(12):20–22.

Iannone, E., Guadagni, R., Ferrucci, F., De Lucia, A., and Palomba, F. (2023). The secret life of software vulnerabilities: A large-scale empirical study. *IEEE Transactions on Software Engineering*, 49(1):44–63.

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*, pages 110–115.

Kirk, D., Tempero, E., Luxton-Reilly, A., and Crow, T. (2020). High school teachers' understanding of code style. In *Koli Calling'20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research*, pages 1–10.

Liyanagunawardena, T. R., Adams, A. A., and Williams, S. A. (2013). Moocs: A systematic study of the published literature 2008-2012. *International Review of Research in Open and Distributed Learning*, 14(3):202–227.

Lopes, M. and Hora, A. (2022). How and why we end up with complex methods: a multi-language study. *Empirical Software Engineering*, 27(5):115.

Louridas, P. (2006). Static code analysis. *IEEE Software*, 23(4):58–61.

Mahajan, R., Gupta, P., and Singh, T. (2019). Massive open online courses: concept and implications. *Indian pediatrics*, 56(6):489–495.

Marcilio, D., Bonifácio, R., Monteiro, E., Canedo, E., Luz, W., and Pinto, G. (2019). Are static analysis violations really fixed? a closer look at realistic usage of sonarqube. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 209–219.

Martin, R. C. (2013). *Clean Code-Refactoring, Patterns, Testen und Techniken für sauberen Code: Deutsche Ausgabe*. MITP-Verlags GmbH & Co. KG.

Moller, A. and Schwartzbach, M. I. (2012). Static program analysis. *Notes. Feb*.

Molnar, A.-J., Motogna, S., and Vlad, C. (2020). Using static analysis tools to assist student project evaluation. In *Proceedings of the 2nd ACM SIGSOFT International Workshop on Education through Advanced Software Engineering and Artificial Intelligence*, pages 7–12.

Pereira, R., Henriques, P. R., and Vieira, M. (2018). The effects of code smells on software maintainability: A replication study. *Journal of Software: Evolution and Process*, 30(1):e1941.

Qaralleh, E. A. and Darabkh, K. A. (2015). A new method for teaching microprocessors course using emulation. *Computer Applications in Engineering Education*, 23(3):455–463.

Spinellis, D. (2006). *Code quality: the open source perspective*. Adobe Press.

Stefanović, D., Nikolić, D., Dakić, D., Spasojević, I., and Ristić, S. (2020). Static code analysis tools: A systematic literature review. *Annals of DAAAM & Proceedings*, 7(1).

Tomas, P., Escalona, M., and Mejias, M. (2013). Open source tools for measuring the internal quality of java software products. a survey. *Computer Standards & Interfaces*, 36(1):244–255.

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., and Poshyvanyk, D. (2015). When and why your code starts to smell bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 403–414. IEEE.

Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Zaidman, A., and Gall, H. C. (2018). Context is king: The developer perspective on the usage of static analysis tools. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 38–49.

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

Yüksel, U. and Sözer, H. (2021). Dynamic filtering and prioritization of static code analysis alerts. In *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 294–295. IEEE.

Apêndice A - Lista de *tags SonarQube* desprezadas neste trabalho

O presente apêndice contém a lista de descrições das *tags* do Sonarqube que foram desconsideradas durante o processo de análise estática de código-fonte do estudo, devido à sua relação com uma versão específica de uma linguagem de programação.

Regra	Tag	Descrição
S1135	cwe	Rastrear usos de tags “TODO”
S3504	es2015	Variáveis devem ser declaradas com “let” ou “const”
S125	unused	Seções de código não devem ser comentadas
S1788	es2015	Parâmetros de função com valores padrão devem ser os últimos
S3863	es2015	As importações dos mesmos módulos devem ser mescladas
S1128	es2015	Importações desnecessárias devem ser removidas
S3353	es2015	Variáveis locais inalteradas devem ser “const”
S3525	es2015	Métodos de classe devem ser usados em vez de atribuições de “prototype”
S2208	es2015	As importações curinga não devem ser usadas
S3513	es2015	“arguments” não devem ser acessados diretamente
S3524	es2015	Chaves e parênteses devem ser usados consistentemente com funções de seta
S3514	es2015	A sintaxe de desestruturação deve ser usada para atribuições
S3512	es2015	Template strings devem ser usadas em vez de concatenação
S3499	es2015	As propriedades abreviadas do objeto devem ser agrupadas no início ou no final de uma declaração de objeto
S3498	es2015	A sintaxe abreviada do literal de objeto deve ser usada
S3317	es2015	Os nomes de exportação padrão e os nomes de arquivo devem corresponder
S3500	es2015	Não devem ser feitas tentativas para atualizar variáveis “const”
S1598	es2015	A declaração do pacote deve corresponder ao diretório do arquivo de origem
S6212	java10	A inferência de tipo de variável local deve ser usada
S5194	java12	Use a expressão “Switch” do Java 12
S5664	java14	O espaço em branco para o recuo do bloco de texto deve ser consistente
S6205	java14	Os rótulos de seta de alternância não devem usar palavras-chave redundantes
S5665	java14	Sequências de escape não devem ser usadas em blocos de texto

S5663	java14	O literal de string simples deve ser usado para strings de linha única
S6208	java14	Rótulos separados por vírgula devem ser usados no “Switch” com dois pontos
S6126	java15	A concatenação de várias linhas de string deve ser substituída por blocos de texto
S6209	java16	Os membros ignorados durante a serialização do registro não devem ser usados
S6218	java16	O método equals deve ser substituído em registros contendo campos de matriz
S6216	java16	A reflexão não deve ser usada para aumentar a acessibilidade dos campos dos registros
S6207	java16	Construtores/métodos redundantes devem ser evitados em registros
S6206	java16	Registros devem ser usados em vez de classes comuns ao representar estrutura de dados imutável
S6204	java16	O método “Stream.toList()” deve ser usado em vez de “coletors” quando uma lista não modificável for necessária
S6219	java16	O campo “serialVersionUID” não deve ser definido como “0L” nos registros
S6201	java16	A correspondência de padrão para o operador “instanceof” deve ser usada em vez de simples “instanceof” + “cast”
S6211	java16	O método getter personalizado não deve ser usado para substituir o comportamento do getter do registro
S6217	java17	Os tipos permitidos de uma classe selada devem ser omitidos se forem declarados no mesmo arquivo
S4719	java7	As constantes “StandardCharsets” devem ser preferidas