Universidade Estadual de Feira de Santana

Programa de Pós-Graduação em Ciência da Computação

Avaliação da Qualidade do Estilo de Código de Estudantes através de Analisadores Estáticos

Francisco Tito Silva Santos Pereira

Feira de Santana

2025

Universidade Estadual de Feira de Santana

Programa de Pós-Graduação em Ciência da Computação

Francisco Tito Silva Santos Pereira

Avaliação da Qualidade do Estilo de Código de Estudantes através de Analisadores Estáticos

Dissertação de Mestrado apresentada à Universidade Estadual de Feira de Santana como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Orientador: Roberto Almeida Bittencourt

Coorientadora: Elaine Harada Teixeira de Oliveira
Coorientador: David Braga Fernandes de Oliveira

Feira de Santana

2025

Ficha Catalográfica - Biblioteca Central Julieta Carteado - UEFS

P491a

Pereira, Francisco Tito Silva Santos

Avaliação da qualidade do estilo de código de estudantes através de analisadores estáticos / Francisco Tito Silva Santos Pereira. – 2025.

94 f.: il.

Orientador: Roberto Almeida Bittencourt

Coorientador(a): Elaine Harada Teixeira de Oliveira e David Braga Fernandes de Oliveira

Dissertação (mestrado) – Universidade Estadual de Feira de Santana, Programa de Pós-Graduação em Ciência da Computação, Feira de Santana, 2025.

1. Linguagem de Programação 2. Software – Qualidade. 3. Analisadores Estáticos (AE). I. Bittencourt, Roberto Almeida, orient. II. Oliveira, Elaine Harada Teixeira de, coorient. III. Oliveira, David Braga Fernandes de, coorient, Universidade Estadual de Feira de Santana. IV. Título.

CDU 004.43

Daniela Machado Sampaio Costa - Bibliotecária - CRB-5/2077

Francisco Tito Silva Santos Pereira

Avaliação da Qualidade do Estilo de Código de Estudantes através de Analisadores Estáticos

Dissertação apresentada à Universidade Estadual de Feira de Santana como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Feira de Santana, 18 de fevereiro de 2025

BANCA EXAMINADORA

Documento assinado digitalmente
gov.br
ROBERTO ALMEIDA BITTENCOURT
Data: 18/02/2025 17:36:03-0300
Verifique em https://validar.iti.gov.br

Roberto Almeida Bittencourt (Orientador(a))
Universidade Estadual de Feira de Santana

Documento assinado digitalmente
gov.br
LEANDRO SILVA GALVAO DE CARVALHO
Data: 18/02/2025 17:55:09-0300
Verifique em https://validar.iti.gov.br

Leandro Silva Galvão de Carvalho
Universidade Federal do Amazonas

Documento assinado digitalmente
gov.br
JOSE AMANCIO MACEDO SANTOS
Data: 18/02/2025 17:45:23-0300
Verifique em https://validar.iti.gov.br

José Amancio Macedo Santos
Universidade Estadual de Feira de Santana

Abstract

To improve the quality of students' source code, instructors and researchers seek alternatives to provide code feedback not only regarding its correctness, but also its quality. Thus, tools such as Static Analyzers (SA) can be used to perform code analysis, identifying style problems. Related work indicates there is a low diversity of research in the field of software quality involving programming students at more advanced levels. Therefore, this work aims to evaluate the use of SAs in the context of advanced programming learning based on log analysis from an autograder tool. To do such, this work conducts an initial investigation regarding the quality of students' programs regarding coding style. Based on the understanding of specific coding conventions of each programming language and the SAs' quality report, we had to create a Static Analyzer (NamingCheck) to evaluate variable and function naming in both C and Python. In addition, we created PerfeQ, an SA integration tool, to enable more thorough feedback on code quality, integrating the Cpplint, Pylint and NamingCheck tools – presenting their warning messages and metric values to assess the quality of student code regarding style. We designed code style quality metrics and implemented them in PerfeQ. Then, we conducted a statistical study to check differences in style metrics between i) partial versus final submissions; ii) groups of students with grades above versus below a threshold; iii) C code versus Python code. Furthermore, we computed the correlation between code quality and student performance, and the internal correlation between the metrics. The results suggest that, in general, most students do not follow code style conventions of the languages used. From the scientific understanding of student code regarding their quality, we conclude that there is a need for greater concern from instructors and students regarding the issues of code quality in the process of learning programming languages, since code conventions and standards contribute to better software maintainability.

Keywords: software quality, metrics, static analysis, coding style, autograder, feedback.

Resumo

Para melhorar a qualidade de código dos estudantes, professores e pesquisadores buscam alternativas de fornecer feedback sobre o código não somente quanto à corretude, mas também sobre a sua qualidade. Assim, ferramentas como Analisadores Estáticos (AE) podem ser utilizadas para realizar a análise do código, identificando problemas de estilo. Trabalhos relacionados indicam que existe uma baixa diversidade de pesquisas na área de qualidade de software envolvendo estudantes de programação em níveis mais avançados. Portanto, o presente trabalho busca realizar uma avaliação do uso de AEs no contexto de aprendizagem de programação avançada a partir da análise de *logs* de uma ferramenta de juiz online. Para tanto, o trabalho realiza uma investigação inicial a respeito da qualidade do software dos estudantes quanto ao estilo de codificação. A partir do entendimento de convenções de codificação específicas de cada linguagem de programação e o relatório de qualidade dos AEs, foi necessário criar um Analisador Estático para avaliação da nomeação de variáveis e funções em C e Python (NamingCheck). Além disso, foi criada uma ferramenta integradora de AEs, PerfeQ, para possibilitar um feedback mais completo quanto à qualidade do código, integrando as ferramentas Cpplint, Pylint e NamingCheck – apresentando suas mensagens de aviso e valores de métricas para avaliar a qualidade do código dos estudantes em relação ao estilo. Forem concebidas métricas de qualidade de estilo de código, implementadas em PerfeQ. Em seguida, foi realizado um estudo estatístico buscando verificar diferenças em métricas de estilo entre i) submissões parciais versus finais; ii) grupos de estudantes com notas acima versus abaixo de um limiar; iii) códigos em C versus códigos em Python, além da correlação entre a qualidade de código com o desempenho do estudante, e a correlação interna entre as métricas. Os resultados apontam que, de modo geral, a maioria dos estudantes não costuma seguir as convenções de estilo de código das linguagens utilizadas. Por fim, a partir da compreensão científica dos códigos dos estudantes em relação à sua qualidade, conclui-se ser necessária uma maior preocupação de professores e estudantes em trabalhar a questão da qualidade de código no processo de aprendizagem de linguagens de programação, pois convenções e padrões de código contribuem para melhor manutenibilidade de software.

Palavras-chave: qualidade de software, métricas, análise estática, estilo de programação, juiz online, feedback.

Prefácio

Esta dissertação de mestrado foi submetida à Universidade Estadual de Feira de Santana (UEFS) como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

A dissertação foi desenvolvida no Programa de Pós-Graduação em Ciência da Computação (PGCC), tendo como orientador o Prof. Dr. **Roberto Almeida Bittencourt**. A Prof. Dra. **Elaine Harada Teixeira de Oliveira** juntamente com o Prof. Dr. **David Braga Fernandes de Oliveira** atuaram como coorientadores deste trabalho.

Agradecimentos

Gostaria de agradecer primeiramente a Deus por continuar abençoando e iluminando todos os meus caminhos até aqui.

Além disso, agradeço grandemente à minha mãe por ter sido uma guerreira e sempre ter lutado para garantir uma boa educação para mim e minha irmã. As minhas outras mães, que me ajudaram e sempre serei grato: minha avó Diva e minhas tias Anatália e Ocsicnarf.

Minha irmã Anna Luiza, pelas uniões e brigas de cada dia, meus primos e primas que são como irmãos para mim (e estiveram nesta jornada desde o meu nascimento até o dia de hoje) meu padrasto, tios e tias (que também sempre me orientaram e vêm me ajudando).

Ao professor Roberto, que através de sua dedicação vem me orientando e sempre disposto a me ajudar e apoiar em novos projetos. Obrigado por tudo, professor. Aos professores David e Elaine, por aceitarem a coorientação neste trabalho – o feedback e apoio de vocês foram fundamentais para que o trabalho fosse realizado. Agradeço também à professora Cláudia, pelo carinho e apoio desde sempre. Além dos professores de ensino médio, graduação e mestrado, que por meio de seus ensinamentos e conhecimentos contribuíram significativamente para minha formação acadêmica e pessoal.

Aos meus amigos do colégio, graduação e do mestrado, que me apoiaram e compartilharam comigo os desafios e as conquistas ao longo dessa jornada. A July e Flávia pela amizade e parceria de sempre.

E por fim, a você, caro leitor, que de alguma forma se interessou pelo meu trabalho.

"Success is 1% inspiration, 98% perspiration, and 2% attention to detail."

– Phil Dunphy

Sumário

Alinhamento com a Linha de Pesquisa

Linha de Pesquisa: Software e Sistemas Computacionais

Esta dissertação investiga a qualidade de estilo de código produzido por estudantes de cursos não introdutórios de programação, focando na análise de estilo e aderência às convenções das linguagens C e Python. Para isso, são utilizados analisadores estáticos, além do desenvolvimento de duas ferramentas específicas: o NamingCheck, para avaliação de nomes de variáveis e funções, e o PerfeQ, que integra diferentes AEs e apresenta métricas de estilo. A partir de uma análise empírica de logs de submissões em um juiz online, a pesquisa identifica problemas de estilo, diferenças entre grupos de estudantes e correlações entre qualidade de código e desempenho acadêmico. Dada a importância de práticas de codificação para a manutenibilidade e evolução de software, esta dissertação está alinhada com a linha de pesquisa Software e Sistemas Computacionais, pois contribui para o aprimoramento da formação de desenvolvedores mais conscientes quanto à qualidade do código e promove o uso de ferramentas automatizadas que podem apoiar tanto o ensino quanto a engenharia de software.

Produções Bibliográficas, Produções Técnicas e Premiações

Pereira, F. T., Bittencourt, R. A., Oliveira, E. H., & Oliveira, D. B. (2025, Abril). PerfeQ: Um Ambiente Científico para a Análise da Qualidade de Código. In Simpósio Brasileiro de Educação em Computação (EDUCOMP) (pp. 651-662). SBC.

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Qualidade de software é um tema que vem sendo amplamente discutido nas salas de aula e fora delas. Com a necessidade de criar softwares com menos erros e que gerem menos problemas no futuro, torna-se cada vez mais importante que a sua implementação cumpra não somente os requisitos propostos para o seu funcionamento, mas que possa satisfazer requisitos de usabilidade, desempenho, confiabilidade e facilidade de manutenção. (Kan, 2003; Hedberg et al., 2007).

A possibilidade de manter um código organizado e que siga as convenções de estilo da linguagem de programação escolhida irá ajudar no melhor entendimento da lógica utilizada pelo desenvolvedor e por outros que possam utilizar o código no futuro para possíveis manutenções (Ljung, 2021). Movimentos como *clean code* existem para apresentar aos desenvolvedores os melhores princípios e práticas que podem ser utilizados para melhorar a qualidade de seu código – como, por exemplo, a sua estruturação e nomeação de variáveis (Latte et al., 2019).

Em se tratando de qualidade de software de modo geral, Börstler et al. (2018) afirmam que os desenvolvedores avaliam a qualidade dos códigos com base em métricas e convenções estabelecidas para esse propósito. Por exemplo, a norma *ISO/IEC 25010:2011* define atributos que um software deve possuir para ser considerado de qualidade. Dentre eles, destacam-se a eficiência, satisfação, compatibilidade, usabilidade, manutenibilidade e segurança. Assim, o desenvolvedor deve se atentar a esses atributos e a convenções para garantir que o seu software seja considerado de qualidade. Seguir as regras de estilo de implementação da linguagem de programação escolhida para tornar o seu código mais limpo são um elemento importante nesse contexto (Trichkova-Kashamova, 2021).

As regras de estilo de implementação de um código variam de acordo com a linguagem de programação escolhida, cada qual possuindo convenções aceitas pela comunidade. De acordo com Allamanis et al. (2014), convenção de código é uma restrição sintática não imposta pela gramática de uma linguagem de programação. No entanto, tais escolhas são importantes o suficiente para serem aplicadas por desenvolvedores de software. Essas regras também estão diretamente ligadas com a

forma como o desenvolvedor irá escrever os blocos de código, como esse será indentado e até mesmo como deverão ser formatados os nomes das variáveis e funções.

Para garantir que o código desenvolvido esteja dentro dos padrões desejáveis para a qualidade de software a partir do feedback a respeito de estrutura e convenções de código, desenvolvedores contam com a ajuda de ferramentas de análise de código como, por exemplo, analisadores estáticos. A análise estática de código é um método de verificação de código que é realizado sem a necessidade de execução do código-fonte. Os resultados da análise ajudam na identificação de potenciais *bugs* ou erros como problemas de estilo, erros de memória e de ponteiros (Edwards et al., 2017; Gomes et al., 2009; Lima et al., 2021).

O estudo de Mengel e Yerramilli (1999) explica como a utilização de ferramentas de análise automatizada, de modo geral, pode auxiliar professores e monitores ao fornecer um nível refinado de detalhes para a avaliação. Nesse estudo, a análise automatizada foi utilizada para verificar as seguintes características nos códigos dos estudantes: (i) corretude; (ii) estilo; (iii) eficiência e (iv) complexidade.

Apesar da efetividade da análise estática em identificar potenciais erros em códigos, alguns problemas são encontrados quando de sua utilização. Dentre eles, destacamos os falsos positivos – mensagens de aviso que apresentam potenciais erros no código, porém que não são problemas de verdade (Kim e Ernst, 2007). A existência dos falsos positivos pode causar problemas para os utilizadores da ferramenta. Com a grande quantidade de avisos, eles podem se sentir perdidos e não saber se determinado aviso indica um problema real ou não (Shen et al., 2011). Para buscar diminuir os problemas relacionados com falsos positivos, pesquisadores apresentam metodologias e algoritmos para a priorização de mensagens de aviso (Kim e Ernst, 2007; Shen et al., 2011; Zampetti et al., 2022; Aman et al., 2020).

De acordo com Hedberg et al. (2007), o tema de qualidade de software aparece no currículo de diversos cursos de graduação e pós-graduação. Porém, é comum os professores apresentarem de forma teórica o conteúdo e discutirem com os estudantes a respeito do assunto. Entretanto, na prática, devido à grande quantidade de alunos nas salas de aula, eles não conseguem avaliar e apresentar um feedback individual a respeito da qualidade do código entregue pelos estudantes.

O mapeamento sistemático realizado Keuning et al. (2023) apresenta que, de modo geral, a qualidade de código na educação gira em torno de ferramentas digitais - — com foco em diversos aspectos da qualidade de código, como a identificação de *code smells* e a refatoração de código, utilizando técnicas de análise estática. Em seu trabalho, os autores revisaram artigos que desenvolveram novas ferramentas de análise de código, como o *EarthWorm* e o *FooBaz*. Eles também apresentaram exemplos de ferramentas profissionais utilizadas na educação, como *PMD*, *CppCheck* e *SonarQube*.

Para automatizar a correção de códigos entregues por alunos, permitindo assim um feedback individual, pesquisadores criaram os juízes online, também chamados de

autograders em inglês. Estas ferramentas de correção automatizada são capazes de identificar, a partir de um código enviado por um estudante, se esse apresenta as saídas esperadas ou não a partir de um conjunto de casos de teste. Além de identificar a corretude da saída, algumas ferramentas de correção automatizada podem apresentar dicas de como os erros podem ser corrigidos (Galvão et al., 2016). Porém, essas ferramentas não avaliam a qualidade do software a partir de métricas pré-estabelecidas como, por exemplo, se o código segue as convenções de estilo de determinada linguagem de programação.

Segundo Keuning et al. (2017), a maioria dos estudantes não se preocupa com a qualidade de estilo dos softwares durante a sua implementação – satisfazendo-se apenas quando a saída de seu código é igual à saída esperada. Os códigos implementados com baixa qualidade de estilo podem causar sérios problemas a longo prazo, afetando atributos de qualidade como manutenibilidade, desempenho e segurança.

Para melhorar o feedback a respeito da submissão de programas por estudantes sobre as características de estilo de código e incentivar os estudantes a se preocuparem com a sua qualidade, a integração entre sistemas de juízes online e ferramentas de análise estática é uma solução possível. Os autores Liu e Woo (2020) apresentam como essa integração pode beneficiar estudantes e identificar possíveis problemas no código que possam prejudicar a sua qualidade, apresentando assim um feedback mais completo.

A partir do que foi apresentado anteriormente, nota-se a importância de introduzir o conceito de qualidade de software para os estudantes nos anos iniciais da graduação. Os trabalhos relacionados buscam apresentar a utilização de sistemas de juízes online para estudantes do primeiro semestre a fim de providenciar um feedback automatizado e individualizado (Mengel e Yerramilli, 1999; Keuning et al., 2017; Liu e Petersen, 2019; Liu et al., 2024). Por outro lado, outros trabalhos apresentam a utilização de ferramentas de análise estática de código para apresentar feedback quanto à qualidade de estilo do código do estudante, apresentando os possíveis problemas e sugestões de melhoria (Liu e Woo, 2020). Por fim, nota-se a baixa diversidade de pesquisas na área de qualidade de software envolvendo estudantes em níveis mais avançados (i.e., a partir do segundo ano de curso), com poucas exceções Plösch e Neumüller (2020). A avaliação da corretude e da qualidade de estilo de código em conjunto são relevantes nesses níveis, já que esses estudantes estão mais próximos da conclusão de seus cursos de graduação e devem ingressar em ambientes corporativos que demandam essas habilidades.

1.1 Objetivos e Questões de Pesquisa

1.1.1 Objetivo Geral

O objetivo deste trabalho é avaliar o potencial do uso de analisadores estáticos no contexto de aprendizagem de programação avançada a partir da análise de *logs* de

uma ferramenta de juiz online potencializando feedback mais completo aos estudantes, incluindo a qualidade do código.

1.1.2 Objetivos Específicos

1. Analisar os códigos dos estudantes de programação avançada da Universidade Federal do Amazonas em relação ao estilo de código nas linguagens C e Python a partir de analisadores estáticos;
2. Criar uma ferramenta integradora de analisadores estáticos para uma melhor visualização da qualidade de estilo de código;
3. Analisar estatisticamente a correlação entre estilo de código e desempenho do estudante em um juiz online.

1.1.3 Questões de Pesquisa

A partir do objetivo geral descrito acima, apresentamos as seguintes questões de pesquisa:

1. (QP1) Quais as mensagens de aviso mais comuns apresentadas pelos analisadores estáticos a partir dos códigos dos estudantes?
2. (QP2) Quais métricas podem ser desenvolvidas para mensurar problemas de estilo no código dos estudantes?
3. (QP3) Os estudantes seguem os padrões de estilo das linguagens de programação utilizadas?
4. (QP4) Existe correlação entre a qualidade do estilo de código e o desempenho do estudante nos juízes online?

1.2 Contribuições

Este trabalho apresenta as seguintes contribuições:

- Criação de um Analisador Estático para as linguagens C e Python, cujo objetivo principal é fornecer feedback a respeito do estilo do código analisado de acordo com as convenções das linguagens;
- Criação de métricas para avaliação da qualidade de estilo de código;
- Criação de uma ferramenta integradora de Analisadores Estáticos para um feedback mais completo quanto à qualidade de estilo de código;
- Estudo empírico de um *dataset* de estudantes da UFAM, verificando a sua qualidade de estilo de código.

Capítulo 2

Revisão Bibliográfica

Neste capítulo, é apresentada a fundamentação teórica para um melhor entendimento deste trabalho. Em seguida, são apresentados trabalhos relevantes que se relacionam com o tema.

2.1 Qualidade de Código

Não é recente a discussão a respeito do tema qualidade de software na comunidade acadêmica e fora dela (Miguel et al., 2014). A capacidade de desenvolver softwares completos e que permitam a sua futura manutenibilidade é uma competência cada vez mais requerida. Kan (2003) apresenta a qualidade de software como um conceito multidimensional, incluindo a entidade de interesse, o seu ponto de vista e os seus atributos de qualidade.

De acordo com Kan (2003), de um ponto de vista geral, a qualidade de software é definida como uma característica intangível, ou seja, pode ser discutida, porém não pode ser pesada ou medida diretamente. Portanto, de acordo com suas experiências passadas, as pessoas conseguem, de forma vaga, definir se algo possui uma boa qualidade ou má qualidade. Por outro lado, de um ponto de vista profissional e educacional, a qualidade de software pode ser definida como a conformidade com requisitos. Sendo assim, os requisitos devem ser previamente definidos e estruturados e métricas devem ser utilizadas para a sua avaliação.

Métrica de software é definida por Gaffney Jr (1981) como uma medida matemática objetiva que é sensível às diferenças nas características do software. Ela fornece uma medida quantitativa de um atributo que o corpo do software exibe (Nirpal e Kale, 2011). Ou seja, a partir da análise de um software, é possível obter dados que, com o auxílio da matemática, podem explicar seu comportamento e medir atributos como tamanho, complexidade, confiabilidade e qualidade (Chhillar e Gahlot, 2017).

Pesquisadores buscam apresentar métricas que podem ser utilizadas para ajudar no processo de medição de software para garantir a sua qualidade, como por exem-

plo a métrica **Complexidade Ciclomática**. Essa métrica busca identificar o grau de complexidade de um software, medindo o número de caminhos linearmente independentes dentro de um trecho de código (Feghali et al., 1994; Ebert et al., 2016). Outra métrica conhecida é o número de **Linhas de Código** (LOC), que conta as linhas com instruções ou declarações de um software – excluindo linhas em branco e comentários. Ela é geralmente utilizada para calcular e comparar a produtividade de programadores (Chhillar e Gahlot, 2017; Nuñez-Varela et al., 2017).

A revisão sistemática de Nuñez-Varela et al. (2017) identificou mais de 300 métricas de código-fonte na literatura. Esse trabalho apresenta também as métricas com o maior número de ocorrências em trabalhos acadêmicos separados por paradigmas. Dentre elas, em um paradigma procedural estão: (i) Complexidade Ciclomática; (ii) Número de Linhas de Código; (iii) Linhas de Código em branco; (iv) Linhas de Comentários; (v) Número de Operadores Distintos. O trabalho também apresenta que a grande maioria dessas métricas se relaciona com *Code Smells* – apresentando indicativos de problemas no código.

Dentre as diversas métricas existentes que podem ser consideradas para a avaliação da qualidade de um software, este trabalho utiliza métricas que se relacionam com a qualidade de estilo de código – sendo indicativas de clareza e entendimento de programas (Mengel e Yerramilli, 1999).

2.1.1 Estilo e Convenções de Código

Estilo de código está diretamente relacionado à maneira como o programador escreve e organiza o seu código. Segundo Reiss (2007), todos os desenvolvedores possuem um estilo de codificação, que afeta a organização de seus códigos e como eles enxergam códigos desenvolvidos por outros desenvolvedores. É importante ressaltar que a forma de desenvolver um software pode sofrer algumas alterações dependendo das convenções de código que são impostas por determinadas linguagens de programação como, por exemplo, a forma de indentação. Visando melhorar a manutenibilidade e garantir que a forma de codificar um programa em uma determinada linguagem seja a mesma para diferentes desenvolvedores, foram criadas as normas e convenções de código.

A norma *ISO/IEC 25010:2011* define alguns padrões para que um software seja considerado de qualidade. Ela apresenta o tópico de manutenibilidade, ou seja, a capacidade de modificação do software pelo seu criador ou por outros desenvolvedores, possibilitando a sua evolução (Trichkova-Kashamova, 2021; Kirk et al., 2022). Manutenibilidade, segundo Prause e Jarke (2015), pode ser dividida nas seguintes capacidades: (i) análise; (ii) mutabilidade; (iii) estabilidade; (iv) testabilidade; (vi) conformidade com a capacidade de manutenção.

O trabalho de Berry e Meekings (1985) apresenta uma métrica de avaliação de estilo de código em C. Em seu trabalho, são avaliadas as seguintes características do código: (i) tamanho do módulo; (ii) tamanho de identificadores; (iii) comentários; (iv)

indentação; (v) linhas em branco; (vi) tamanho da linha; (vii) espaços incorporados; (viii) definições de constantes; (ix) palavras reservadas; (x) arquivos incluídos; e (xi) presença de *goto's*.

Por outro lado, as convenções de código podem determinar as preferências a respeito de nomes de identificadores, como deve ser a formatação de uma classe, relacionamentos entre objetos e até mesmo padrões de design (Allamanis et al., 2014; dos Santos e Gerosa, 2018). Essas convenções surgiram a partir da preferência de codificação dos desenvolvedores até se tornarem lugar comum na comunidade e variam de acordo com a linguagem de programação escolhida.

Mesmo quando um código possui boa qualidade de estilo, isso não garante que ele tenha uma boa qualidade de código (Kirk et al., 2024). Embora a qualidade de estilo contribua para a qualidade geral, o código também precisa atender a outros requisitos essenciais, como desempenho, segurança e robustez, para ser considerado de alta qualidade.

Em seguida, serão apresentadas as convenções de código mais utilizadas nas linguagens de programação Python e C. Por causa de direitos de reprodução, as convenções completas não estão disponíveis nos anexos do trabalho. Porém, *links* de acesso estão disponíveis nas referências e notas de rodapé.

2.1.2 Convenções de Codificação em Python

O trabalho de Van Rossum et al. (2001) apresenta o *Python Enhancement Proposal 8* (PEP8), a convenção de estilo de codificação que é utilizada para definir os padrões de codificação em Python. Nesse guia, são apresentados tópicos para uma codificação mais legível. A seguir serão apresentados com mais detalhes os tópicos: (i) formatação de código; (ii) utilização de strings; (iii) espaços em branco; (v) convenções de nomes.

Formatação de Código

Python é uma linguagem cujos blocos de código são delimitados a partir da indentação, ou seja, pelo espaçamento. De acordo com o guia, a forma de indentação recomendada é o espaçamento ao invés de *tabs*, sendo que a última deve ser utilizada somente para manter a consistência com o código que já foi indentado com *tabs*. Segundo o guia PEP8, a indentação deve ser de quatro espaços por nível de indentação, como no exemplo abaixo:

```
def soma_valores(valor1, valor2):
    return valor1 + valor2
soma = soma_valores(1,2)
print(f'Resultado = {soma}')
```

É possível notar que o corpo da função `soma_valores` possui o valor de espaçamento apresentado anteriormente.

Este espaçamento também é recomendado para separar os argumentos em uma função. Nota-se que um novo nível de indentação foi criado para diferenciar os argumentos do corpo da função:

```
def funcao_com_muitos_argumentos(
        argumento1, argumento2,
        argumento3, argumento4)
    print(argumento1)
```

Para evitar linhas muito grandes, o recomendado é a utilização de no máximo 79 caracteres por linha. Caso necessário, o usuário pode utilizar a barra invertida (\) para dividir blocos de código:

```
with open('/caminho/para/arquivo1') as arquivo1, \
     open('/caminho/para/arquivo2') as arquivo2:
    linha_arquivo1 = arquivo1.read()
    linha_arquivo2 = arquivo2.read()
```

Para importações de código, o guia afirma que essas devem ser realizadas em linhas separadas:

```
import os
import sys
```

A única exceção é quando os módulos importados são provenientes do mesmo arquivo/biblioteca:

```
from subprocess import Popen, PIPE
```

O agrupamento dessas importações deve seguir a seguinte ordem:

1. Importação de bibliotecas padrão
2. Importação de bibliotecas de terceiros
3. Importação de aplicações/bibliotecas locais

Utilização de Strings

Em Python, não há distinção entre utilizar uma *string* com aspas duplas ou simples. O guia apresenta somente um alerta de que, ao utilizar uma string que contém aspas em seu conteúdo, caso esteja utilizando aspas duplas para definir a string – utilize aspas simples dentro dela (e vice-versa):

```
string_com_aspas = "'String contendo aspas em seu conteudo'"
```

Espaços em branco

O guia recomenda evitar a utilização extrapolada de espaços em branco em algumas situações:

- Imediatamente dentro de parênteses, colchetes ou chaves:

Forma correta:

```
converte_valores(precos[1], {moeda: 'real'}
```

Forma incorreta:

```
converte_valores( precos[ 1 ], { moeda: 'real' }
```

- Entre um *trailing comma* seguido de um parêntese:

Forma correta:

```
foo = (0,)
```

Forma incorreta:

```
bar = (0, )
```

- Imediatamente antes de uma vírgula, ponto e vírgula ou dois pontos:

Forma correta:

```
if x == 4: print x, y; x, y = y, x
```

Forma incorreta:

```
if x == 4 : print x , y ; x , y = y , x
```

- Imediatamente antes da abertura de parênteses que começa com a lista de argumentos da chamada de uma função:

Forma correta:

```
soma_valores(1, 2)
```

Forma incorreta:

```
soma_valores (1, 2)
```

- Utilizar um espaço entre operadores de atribuição, comparação e booleanos:

Forma correta:

```
i = i + 1
soma += 1
x = x*2 - 1
valor = x*x + y*y
c = (a+b) * (a-b)
```

Forma incorreta:

```
i=i+1
soma +=1
x = x * 2 - 1
valor = x * x + y * y
c = (a + b) * (a - b)
```

Convenções sobre identificadores

A nomeação de funções e variáveis em Python é considerada *Case Sensitive*, ou seja, há uma distinção entre minúsculas e maiúsculas. Por isso, o desenvolvedor deve prestar atenção nesses detalhes.

Nomes de classes devem sempre utilizar a convenção *Camel Case*, onde cada palavra é iniciada com uma letra maiúscula:

```
class ExemploClasse:
```

Os nomes de funções e variáveis devem estar em minúsculas com palavras separadas por *underscore*. Já as constantes devem ser sempre definidas no início do código e totalmente em maiúsculas:

```
EXEMPLO_CONSTANTE = 20

def exemplo_de_funcao():
    exemplo_de_variavel = 10
    print('Ola mundo!')
```

2.1.3 Convenções de Codificação em C

O trabalho de Doland e Valett (1994) e o repositório do Github CS50[1] buscam apresentar as convenções de codificação na linguagem C para uma melhor codificação e legibilidade do código. A seguir serão apresentados com mais detalhes os tópicos: (i) formatação de código; (ii) condicionais; (iii) funções e (iv) convenção de identificadores.

[1] https://github.com/cs50/cs50.readthedocs.io

Formatação de Código

Ao contrário de Python, C é uma linguagem cujos blocos de código são delimitados a partir da utilização de chaves ({ e }). O recomendado é colocar os delimitadores em linhas separadas, para deixar claro o que pertence ao bloco. É importante ressaltar que, por convenção, o tamanho máximo que uma linha pode ter em C é de 80 caracteres:

```
if (x > 0)
{
    printf("X maior que zero");
}
```

Em se tratando de indentação de código, os guias disponíveis também recomendam uma indentação de quatro espaços. Sugere-se que o desenvolvedor configure a tecla *tab* no editor de código para que seja convertida automaticamente para quatro espaços:

```
printf("\n")
for (int i = 0; i < 10; i++)
{
    printf("\%d",i);
}
```

Importações devem estar no cabeçalho do código em ordem alfabética:

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

Condicionais

Os guias recomendam o seguinte formato para condicionais:

```
if (x > 0)
{
    printf("X maior que zero");
}
else if (x < 0)
{
    printf("X menor que zero");
}
else
{
    printf("X igual a zero");
```

```
}
```

Nota-se o seguinte:

- Existe um espaço simples após cada **if**
- Cada item dentro do bloco possui uma indentação de quatro espaços.
- Existem espaços simples antes dos operadores.
- Não existe um espaço simples antes e após a utilização dos parênteses.

Funções

Um bloco de função tem a seguinte formatação:

```
int exemplo_funcao()
{
    printf("Chamada funcao");
}
```

Convenção de identificadores

Por convenção, espera-se que os nomes das variáveis estejam de acordo com o valor que está sendo armazenado – sendo objetivos e intuitivos. Assim como em **Python**, os identificadores das variáveis em **C** são **Case Sensitive**. As variáveis e funções devem ser escritas em minúsculas e, caso tenha duas ou mais palavras, essas devem ser separadas por *underscore*. Uma exceção é que constantes devem ser sempre escritas totalmente em maiúsculas:

```
#define PI 3.14
int soma_valores = 10;
float media_idade_alunos = 19.5;

function exemplo_de_funcao()
{
    printf("Hello World!");
}
```

Outra exceção é que, em *loops*, variáveis contadoras costumam usar nomes típicos de álgebra, como i, j e k:

```
for (int i = 0; i < 10; i++)
{
    for (int j = 0; j < 10; j++)
    {
        for (int k = 0; k < 10; k++)
```

```
        {
            printf("%d,%d,%d"), i, j, k);
        }
    }
}
```

Sobre ponteiros em C, o asterisco (*), cuja utilização serve para informar que uma variável armazena um endereço de memória, deve estar sempre ao lado da variável:

Forma correta:

```
int *p;
```

Forma incorreta:

```
int* p;
```

Pela necessidade de desenvolver softwares que sigam as convenções de determinadas linguagens de programação, surgiram alguns movimentos entre as comunidades de desenvolvedores para servir como ensinamento e motivação para escrever códigos considerados limpos e de qualidade. Um deles é o movimento *Clean Code*, destacado a seguir.

2.1.4 Movimento Clean Code

O movimento Clean Code surgiu em 2008 e buscou apresentar para os desenvolvedores as boas práticas de desenvolvimento, além dos princípios que devem ser seguidos para a manutenção do software no futuro. Além disso, também são apresentados os possíveis problemas que um código pode possuir tanto estruturalmente quanto funcionalmente e que podem levar a problemas no futuro, os chamados *Code Smells* (Martin e Coplien, 2009; Abidi et al., 2019). De acordo com Lacerda et al. (2020), diferente de um *bug*, um *Code Smell* não significa que o trecho de código fará a aplicação falhar, porém indica a violação de boas práticas de codificação e princípios de design, o que pode levar a outras consequências negativas como, por exemplo, dificultar a manutenção e evolução do software (Mashiach et al., 2023; Cairo et al., 2018).

Alguns *Code Smells* típicos podem ser encontrados em códigos como, por exemplo, códigos duplicados, métodos longos, classes grandes e até listas com vários parâmetros. Um exemplo bastante citado na literatura é o de *God Class/Object*, onde uma classe concentra várias responsabilidades, aumentando a sua complexidade e acoplamento e tornando difícil a sua manutenção. (Mashiach et al., 2023; Vaucher et al., 2009)

Segundo Martin e Coplien (2009); Izu et al. (2025), a qualidade de código está fortemente ligada à sua compreensibilidade. Ou seja, o código desenvolvido deve estar

bem estruturado, seguindo as convenções impostas pela linguagem de programação utilizada, fazendo com que outras pessoas consigam entender o código desenvolvido e implementar novas *features* ou realizar manutenções sem grandes problemas. A aplicação e o cumprimento dessas práticas desde o início do desenvolvimento ajudam a garantir a qualidade do software, diminuindo a probabilidade de ocorrência de erros e evitando a dívida técnica (Filazzola e Lortie, 2022).

2.2 Feedback sobre Código-Fonte

Em um ambiente de educação superior, além de garantir que os estudantes sigam as métricas e padrões de desenvolvimento de determinadas linguagens, os professores tipicamente procuram afetar diretamente a qualidade do código dos estudantes a partir do feedback do software entregue por eles (Stegeman et al., 2016).

O trabalho de Gomes e Mendes (2007) apresenta alguns problemas que são encontrados durante o processo de aprendizagem de programação. Um deles envolve o feedback individual que é fornecido pelo professor para o estudante – esse pode melhorar os índices motivacionais e de engajamento dos estudantes. Sua ausência, por outro lado, pode contribuir para a evasão e reprovação de estudantes.

Porém, em muitos casos, torna-se inviável para o professor fornecer um feedback individual acerca dos códigos por causa da grande quantidade de alunos matriculados nas disciplinas. Nesse contexto, surge a necessidade de utilizar ferramentas automatizadas que permitam avaliar o código enviado pelo aluno e apresentar um feedback quanto à sua corretude e sua qualidade. Uma das formas de oferecer esse feedback é através de ferramentas de juiz online (Messer et al., 2024; Hidalgo-Céspedes et al., 2020; Pereira et al., 2023).

2.2.1 Juízes Online

Segundo Galvão et al. (2016) e Francisco et al. (2018), os sistemas de juízes online ou *autograders* costumam ser empregados como ferramentas de apoio pedagógico em disciplinas de programação. A partir de códigos de entrada que são enviados pelos estudantes em cada exercício de programação, a sua saída é comparada com uma saída esperada e o feedback é apresentado para o estudante a partir da avaliação da corretude do seu código.

A pesquisa de Wasik et al. (2018) apresenta uma revisão acerca de ferramentas de juízes online existentes, além de as categorizar quanto à sua utilidade. Para a programação competitiva, destaca-se o *National Tsing Hua University Online Judge*, que contém mais de 10.000 desafios e dá suporte às linguagens de programação C e C++. No contexto educacional, os autores destacam a ferramenta *URI Online Judge* (atualmente conhecida como *BeeCrowd*), a qual contém desafios divididos em categorias e permite que o professor acompanhe o progresso do aluno.

O trabalho de Galvão et al. (2016) apresenta o sistema *CodeBench*, um sistema de juiz online desenvolvido na Universidade Federal do Amazonas. A partir dele, os professores disponibilizam exercícios de programação para os estudantes e o sistema julga a corretude dos códigos submetidos. Ainda sobre o *CodeBench*, para a avaliação do código enviado pelo aluno, o sistema realiza os seguintes passos: (i) análise sintática do código – caso existam erros de sintaxe na linguagem de programação escolhida, o estudante será alertado; e (ii) análise lógica – feita por casos de teste que possuem valores de entrada e valores de saída (Galvão et al., 2016). Dentro do sistema, o estudante pode submeter o código várias vezes e todas as submissões e interações do usuário com a ferramenta são armazenadas em *logs*, a partir dos quais *datasets* podem ser obtidos posteriormente para análises futuras.

De acordo com Galvão et al. (2016), no *CodeBench*, caso o programa do aluno funcione corretamente, o sistema irá informar que a submissão está correta. Caso contrário, será apresentado o percentual de casos de teste em que o aluno obteve êxito (em caso de sucesso parcial ou falha). Na plataforma, o professor pode também associar o êxito do aluno a uma nota.

Por outro lado, o feedback para os estudantes provido pelos juízes online é normalmente limitado somente à corretude do código, não avaliando a sua qualidade quanto ao estilo de codificação – ou seja, se o código submetido está de acordo com as convenções da linguagem escolhida.

Para buscar um feedback mais completo sobre código-fonte, estudantes e desenvolvedores profissionais recorrem às ferramentas de análise estática de código. Nesse contexto, o presente trabalho procura compreender como as ferramentas de análise estática podem ser aplicadas sobre códigos enviados a uma ferramenta de juiz online.

2.2.2 Análise Estática

A análise estática de código é apresentada como a identificação automática de propriedades de tempo de execução do programa, o qual considera erros de execução em tempo de compilação automaticamente, sem instrumentação de códigos ou interação com o usuário, ao contrário da análise dinâmica, que computa os estados de tempo de execução do programa com a necessidade de execução do programa (Vorobyov e Krishnan, 2010). Ou seja, uma ferramenta que realiza a análise estática de código permite a identificação precoce de potenciais *bugs*, vulnerabilidades de segurança, problemas de performance ou desvios de diretrizes de codificação do projeto (Zampetti et al., 2022; Muniz et al., 2022; Marcilio et al., 2019).

A utilização de Analisadores Estáticos (AEs) pode trazer alguns benefícios para o desenvolvimento de um software como, por exemplo, redução da necessidade de depuração de código. Permite também que a sua implementação seja direcionada para atingir as métricas de qualidade, como confiança e legibilidade (Gomes et al., 2009). É importante ressaltar que as ferramentas de análise estática não buscam

garantir a ausência de erros, mas sim encontrar o maior número possível de potenciais erros (Bessey et al., 2010).

Existem várias ferramentas já existentes no mercado que realizam a análise estática de um código. Pesquisadores buscam realizar comparações entre elas e verificar quais são as mais eficazes para a garantia da qualidade do código. A seguir serão apresentadas algumas ferramentas de análise estática que são utilizadas para as linguagens de programação Java, C/C++ e Python.

Para a linguagem de programação Java, podem ser citadas as ferramentas PMD[2] e CheckStyle[3]. A primeira busca detectar principalmente potenciais falhas de segurança, práticas ineficientes de codificação e situações suspeitas no código (Singh et al., 2017; Panichella et al., 2015; AlOmar et al., 2023). De acordo com Lenarduzzi et al. (2020b), PMD é uma das ferramentas de análise estática mais mencionadas em pesquisas, com mais de 500 mil citações. Já a segunda foca em problemas de legibilidade e padrões de código, verificando possíveis problemas de design de classes, código duplicado ou padrões de *bug* (Panichella et al., 2015; AlOmar et al., 2023; Lenarduzzi et al., 2020b).

Para auxiliar no desenvolvimento nas linguagens de programação C e C++, existem os analisadores estáticos CppCheck[4], CQMetrics (C Code Quality Metrics)[5] e CppLint[6]. O primeiro não busca detectar erros de sintaxe, sendo seu principal objetivo detectar vulnerabilidades no código, além de comportamentos indevidos que podem ser considerados perigosos para os programas, como, por exemplo, vazamentos de memória e recursos (Dos Santos e Martimiano, 2023; Joshi et al., 2014). O segundo busca analisar o código quanto às métricas de qualidade de software e estilo de código. Os padrões de desenvolvimento verificados a partir do analisador são baseados na publicação de Cannon et al. (1991). Por fim, o CppLint é um analisador estático criado pelo Google e busca apresentar os problemas de estilo do código, principalmente os que envolvem a formatação Weinberger et al. (2013).

No desenvolvimento com a linguagem Python, existem as ferramentas Pylint[7] e Pyflakes[8]. A primeira busca verificar o código com base nos padrões de desenvolvimento da linguagem, baseando-se nas métricas de qualidade definidas pelo guia PEP8[9] (Liawatimena et al., 2018; Van Rossum et al., 2001). A segunda não busca detectar erros de sintaxe, focando somente em erros lógicos (Gulabovska e Porkoláb, 2019).

Por fim, vale ressaltar a ferramenta SonarQube[10]. Além de ser utilizada na academia, essa ferramenta é bem recebida na área industrial (sendo utilizada por mais de 85.000

[2]https://pmd.github.io
[3]https://checkstyle.sourceforge.io
[4]https://cppcheck.sourceforge.io
[5]https://github.com/dspinellis/cqmetrics
[6]https://github.com/cpplint/cpplint
[7]https://pypi.org/project/pylint/
[8]https://pypi.org/project/pyflakes/
[9]https://peps.python.org/pep-0008/
[10]https://www.sonarsource.com/products/sonarqube/

organizações) além de fornecer suporte para mais de 25 linguagens de programação. A ferramenta busca analisar códigos-fonte e calcular diversas métricas, como número de linhas de código e sua complexidade, além de verificar a conformidade do código quanto a um conjunto específico de regras de codificação Marcilio et al. (2019); Lenarduzzi et al. (2020a).

As ferramentas de análise estática permitem identificar alguns problemas como, por exemplo: (i) problemas sintáticos; (ii) código-fonte não alcançável; (iii) variáveis não declaradas; (iv) variáveis não inicializadas; (v) funções e procedimentos não utilizados; (vi) variáveis utilizadas antes da inicialização; (vii) não utilização de valores de funções; (viii) uso inadequado de ponteiros (Novak et al., 2010). Para cada análise realizada pela ferramenta, é gerado um relatório contendo os erros encontrados, além de mensagens de aviso (erros que podem não causar interrupções na execução do código, porém afetam a qualidade do código).

Um problema encontrado na literatura a respeito das ferramentas que analisam estaticamente o código envolve as mensagens de aviso apresentadas. Muitas mensagens podem aparecer de forma indevida, apresentando problemas no código que não são prejudiciais de verdade, os chamados falsos positivos (Kim e Ernst, 2007). A presença desses problemas pode prejudicar a experiência do usuário, como apresenta Johnson et al. (2013): a maioria dos desenvolvedores opta por não utilizar as ferramentas de análise estática pela presença da grande quantidade de falsos positivos, diminuindo assim a sua confiança na ferramenta. Alguns trabalhos disponíveis na literatura buscam diminuir os efeitos da presença dos falsos positivos priorizando as mensagens de aviso (Kim e Ernst, 2007; Zampetti et al., 2022; Aman et al., 2020).

2.3 Trabalhos Relacionados

A seguir, apresentamos trabalhos que relacionam com a qualidade de código, analisadores estáticos e potenciais melhorias no código de estudantes, sua integração com sistemas de juízes online, e a criação de métricas relacionadas a estilo de código.

O trabalho de Izu et al. (2025) busca apresentar a importância do tema qualidade de código e como esse pode ser apresentado para cursos introdutórios de computação (CS1). Para isso, foi realizada uma revisão da literatura, onde 248 trabalhos ajudaram a catalogar 63 padrões de defeitos no código, além de 28 atividades instrucionais a respeito da qualidade de código para ajudar instrutores a abordar a qualidade de código de forma mais abrangente. Como objetivo principal, os autores esperam oferecer recursos para o incentivo de boas práticas de codificação desde o início da programação.

A revisão sistemática de Striewe e Goedicke (2014) busca analisar algumas abordagens de análise estática voltadas para estudantes do ensino superior. Os autores informam sobre a importância do *feedback* e como as ferramentas de análise estática podem auxiliar. Segundo os autores, o objetivo das ferramentas automatizadas de

avaliação e tutoria em cenários de aprendizagem é duplo: (i) permitir que os alunos desenvolvam soluções para exercícios sem assistência intensiva de um humano (através de mensagens de compilação mais compreensíveis, por exemplo); (ii) auxiliar os professores na avaliação de um grande número de tarefas. Além disso, eles realizam uma comparação entre diferentes abordagens de análise estática: código-fonte vs *bytecode*, árvores vs grafos, análise de um único arquivo vs análise de múltiplos arquivos. Por fim, os autores apresentam uma análise de ferramentas existentes no mercado e que podem ser utilizadas na educação superior, apontando suas vantagens e desvantagens.

O estudo de Mengel e Yerramilli (1999) apresenta como a utilização de análise estática automatizada pode ajudar instrutores e assistentes, fornecendo um nível refinado de detalhes para a avaliação. Nesse trabalho, foi realizado um estudo de caso das ferramentas de análise estática de código com estudantes novatos de ciência da computação utilizando C++. A análise automatizada foi utilizada para verificar as seguintes características nos códigos dos estudantes: (i) corretude; (ii) estilo; (iii) eficiência; (iv) complexidade. Os autores concluíram, através do estudo piloto, que existem correlações entre a qualidade de software e os resultados obtidos pela análise automatizada através do programa *Verilog Logiscope*.

O trabalho de Liu e Petersen (2019) apresenta como a análise estática de código pode ser utilizada na sala de aula pela utilização de duas ferramentas: *PCRS*[11] e *PyTA*[12]. A primeira é um aplicativo de código aberto para agrupar exercícios de programação. Já a segunda é um módulo do *Python* que utiliza análise estática de código para ajudar os alunos a encontrar e corrigir erros comuns na codificação de exercícios em cursos introdutórios de Python. Os autores apresentam que a linguagem *Python* faz uso de um pequeno número de mensagens, tornando difícil para os estudantes utilizarem as mensagens do interpretador para identificar a fonte do erro. A utilização da ferramenta *PyTA*, segundo os autores, pode ser um complemento eficaz no processo de aprendizagem, auxiliando os alunos na identificação e correção dos erros de programação mais comuns. Assim como o trabalho de Mengel e Yerramilli (1999), os autores Liu e Petersen (2019) possuem como objetivo principal realizar uma análise em um contexto de programação introdutória. O presente trabalho se diferencia pelo foco no contexto de programação não introdutória.

O trabalho dos autores Keuning et al. (2017) buscou verificar a qualidade do código de estudantes iniciantes em programação com a linguagem Java. Os autores puderam verificar, a partir da utilização de padrões de estilo de código da linguagem: (i) quais são os problemas de qualidade que ocorrem no código dos estudantes (a partir da utilização do analisador estático *PMD*, os autores fizeram uma categorização dos problemas encontrados com uma rubrica pré-existente); (ii) quão frequentemente os estudantes corrigem os problemas de código (a partir do histórico de submissões); (iii) quais diferenças ocorreram nos problemas que envolvem qualidade de código

[11] https://mcs.utm.utoronto.ca/ pcrs/pcrs/
[12] https://github.com/pyta-uoft/pyta

entre estudantes que utilizaram analisadores estáticos comparados com os que não usaram (o estudo verificou que não existe uma diferença significativa).

Em Plösch e Neumüller (2020), os autores buscaram verificar a utilização do analisador estático *SonarQube* com estudantes intermediários na programação. Durante todo o processo de desenvolvimento de dois diferentes softwares, os códigos dos estudantes foram analisados quanto à qualidade. Além disso, os estudantes também foram analisados quanto à sua experiência com programação. Verificou-se que estudantes que possuem uma experiência maior se preocupam mais com a qualidade do software desde o início do desenvolvimento, ao contrário de estudantes menos experientes. Além disso, percebeu-se que estudantes com maior experiência utilizam técnicas de programação mais avançadas, tornando possível o aparecimento de avisos diferentes dos encontrados em códigos de estudantes menos experientes.

A pesquisa de Liu et al. (2024) buscou analisar os erros reportados por analisadores estáticos a partir de códigos de estudantes no nível introdutório de computação. Durante o processo, os estudantes tinham acesso a um *autograder* para avaliar a corretude de seu código e o analisador estático *PythonTA* para analisar a qualidade de seu código. O trabalho buscou responder às seguintes questões de pesquisa: (i) Quais são os erros mais comuns cometidos pelos estudantes em Python, reportados pelos analisadores?; (ii) Os erros mais frequentes variam pelo nível de experiência dos estudantes?; (iii) Quais erros persistiram no trabalho final dos estudantes? e (iv) Os erros de analisadores estáticos estão relacionados com as notas dos estudantes? Respondendo à primeira questão, os autores apresentaram os dez erros mais comuns cometidos pelos estudantes em Python. Dentre eles, os dois primeiros estão relacionados com a formatação do código. Em seguida, verificaram que estudantes sem experiência prévia de programação possuem maiores taxas de erros do que os outros com alguma experiência. Em seguida, verificaram que os erros de formatação ainda persistiram nas versões finais dos códigos. Por fim, os autores verificaram a correlação entre a nota final do código dos estudantes com a porcentagem de reprovações no envio. Como resultado, encontraram uma correlação forte negativa entre as duas variáveis.

O trabalho de Saliba et al. (2024) apresentou a criação de um sistema *autograder* que também realiza uma análise do estilo de código dos estudantes a fim de melhorar as suas práticas de codificação. O sistema foi criado para a linguagem de programação C e foi utilizado com estudantes de graduação. Os resultados sugerem que a utilização da ferramenta foi útil, significativa, clara e eficaz e os ajudou a aprender a respeito das boas práticas de codificação. Do ponto de vista do professor, os feedbacks também foram positivos. A maioria informou que a ferramenta economiza tempo e permite que eles foquem em outros tópicos.

O trabalho de Liu e Woo (2020) buscou apresentar a integração de um sistema de juiz online juntamente com uma ferramenta de análise estática de códigos a fim de prover um relatório de qualidade mais completo. A pesquisa se deu com estudantes do primeiro semestre de um curso de programação, utilizando a linguagem Python.

Segundo os autores, a justificativa para a integração entre as ferramentas é que a qualidade de software deve ser colocada em primeiro lugar desde os primeiros semestres. Foram coletados mais de 2000 códigos-fonte de estudantes, presentes no banco de dados do juiz online e verificou-se que os erros mais comuns que os estudantes cometiam estão relacionados com *bad smells*.

O trabalho de Varet e Larrieu (2013) apresenta a ferramenta OpenSource Metrix[13] cujo objetivo principal é computar diferentes métricas de qualidade de software. Cada uma das métricas apresenta informações de qualidade do código-fonte (nas linguagens C e Ada). As métricas apresentadas são: linhas de código (e suas variações), métricas de Halstead e Complexidade Ciclomática de McCabe. A disponibilização dos dados é através de uma interface gráfica que apresenta diagramas, gráficos de barras e histogramas.

O trabalho de Tigina et al. (2023) apresenta uma análise de dados retrospectivos de uma plataforma de MOOC (Massive Open Online Courses), utilizando um grande conjunto de dados composto por 1.073.018 submissões em Java e 1.345.332 submissões em Python. Eles utilizaram a ferramenta Hyperstyle Birillo et al. (2022) – um agregador de ferramentas de análise estática para Python, Java, Kotlin, Javascript e Go. Dependendo do código-fonte fornecido, a ferramenta utiliza diferentes analisadores estáticos existentes para a linguagem em questão e o classifica como Excelente, Bom, Moderado ou Ruim, com base nos resultados desses analisadores e em algumas heurísticas. O resultado da análise mostrou que a maioria dos problemas identificados está relacionada a importações não utilizadas, uso incorreto de *Switch..Case* em Java ou a sobrescrita de nomes de funções embutidas (built-in) em Python.

O trabalho de Glassman et al. (2015)) apresenta uma ferramenta para fornecimento de feedback em larga escala sobre nomes de variáveis. A ferramenta oferece uma interface gráfica para que o instrutor possa comentar a respeito de nomes de variáveis inadequados ou identificar variáveis cujos nomes não correspondem aos valores que armazenam. Os resultados mostraram que a interface auxiliou os instrutores a fornecer um feedback personalizado a respeito da nomeação de variáveis.

O presente trabalho busca realizar a avaliação dos códigos de estudantes em um nível não introdutório no ensino superior, obtidos através de um sistema *autograder*. Este trabalho difere dos anteriores por focar em estudantes intermediários e com algum conhecimento prévio em programação – ao contrário da maioria dos trabalhos apresentados anteriormente, cujo foco era em estudantes em um nível introdutório. Em um primeiro momento, assim como no trabalho de Liu et al. (2024), será necessário realizar a avaliação das mensagens de aviso encontradas pelos analisadores estáticos e verificar se existem correlações entre a nota do código e os avisos provenientes do analisador. Para isso, este trabalho, assim como o de Saliba et al. (2024), apresenta também o processo de criação de um analisador estático. Também é apresentada a criação de uma ferramenta integradora de analisadores estáticos que disponibiliza como saída, métricas que se relacionam com a qualidade do código avaliado.

[13]http://recherche.enac.fr/ avaret/metrix/

Capítulo 3

Metodologia

Este trabalho utiliza uma perspectiva de pesquisa pragmática e uma metodologia de pesquisa quantitativa baseada na análise de dados retrospectivos. A Figura 3.1 apresenta uma visão geral da organização do estudo. Em seguida, os passos e os componentes serão apresentados com mais detalhes.

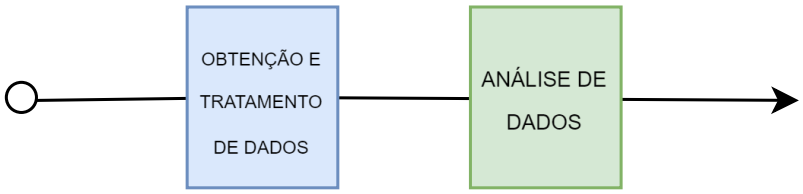

Figura 3.1: Design do Estudo

3.1 Etapas metodológicas

Após a identificação e definição do problema, foi realizada uma fase preliminar de revisão bibliográfica com o objetivo de compreender o estado da arte relacionado ao tema. Com base nesse levantamento, o desenvolvimento do trabalho foi estruturado em torno da metodologia KDD (*Knowledge Discovery in Databases*), que orienta o processo de extração de conhecimento a partir de dados (Maimon e Rokach, 2005). Seguindo essa abordagem, o trabalho foi dividido em duas etapas fundamentais: (i) obtenção e tratamento dos dados, que abrange as fases de seleção, pré-processamento e transformação; e (ii) análise dos dados, correspondente à aplicação de técnicas de mineração de dados e interpretação dos resultados. A Figura 3.2 apresenta uma representação em alto nível do fluxo de dados adotado nesta pesquisa.

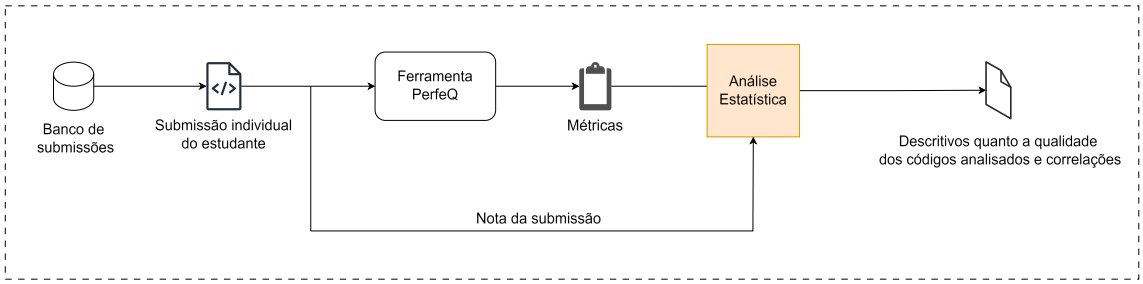

Figura 3.2: Representação do fluxo de dados

Inicialmente os dados foram extraídos do banco de submissões; cada submissão individual passou por uma ferramenta integradora de AEs (PerfeQ), na qual a partir de um código de entrada (que pode ser em C ou Python), foi realizada a análise estática do mesmo – apresentando como saída, métricas relacionadas com a qualidade do estilo do código analisado. As métricas foram computadas a partir das mensagens de aviso dos AEs e que se relacionam com a formatação do código e nomeação de variáveis/funções. O capítulo 4 apresenta com mais detalhes a construção da ferramenta integradora e dos AEs envolvidos. Também foi extraída da submissão individual a sua nota. Por fim, estes dados foram analisados estatisticamente, possibilitando ter como saída final os descritivos quanto à qualidade dos códigos analisados, juntamente com as suas correlações. A seguir, as etapas serão apresentadas com mais detalhes:

3.1.1 Primeira Etapa: Obtenção e tratamento de dados

Obtenção dos Dados

Para a obtenção dos dados, foi utilizado um *dataset* personalizado que foi disponibilizado livremente pelos mantenedores da ferramenta de correção automática de código *CodeBench*[1]. Esta personalização foi necessária visto que o presente trabalho busca analisar códigos de estudantes no nível não introdutório, sendo assim, foram filtrados os códigos e disciplinas que cumpram este requisito.

Estes dados contêm os *logs* de acesso e ações dos estudantes na ferramenta como, por exemplo, cliques na tela, modificações no código, quantidade de execuções, resultados quanto à corretude de código e a nota disponibilizada pelo professor. Os dados dos estudantes são anonimizados, identificando-os somente com um número.

Para esta pesquisa, foram utilizados somente os dados de estudantes da Universidade Federal do Amazonas (UFAM) nas disciplinas de Algoritmos e Estrutura de Dados I e Algoritmos e Estrutura de Dados II dos cursos de Engenharia de Software e Ciência da Computação realizadas no período entre 2020 e 2023. As disciplinas utilizam

[1]https://codebench.icomp.ufam.edu.br/dataset/

as linguagens de programação C e Python. A escolha pelo *dataset* da UFAM se deu pela facilidade de obtenção dos dados e pela grande utilização do *autograder* em alguns cursos de computação da instituição, já que o mesmo foi criado nesta universidade. As Tabelas 3.1 e 3.2 apresentam os conteúdos, respectivamente, para os cursos Algoritmos e Estrutura de Dados I e Algoritmos e Estrutura de Dados II.

Neste cenário, os alunos geralmente aprendem Python no primeiro semestre. Algoritmos e Estruturas de Dados I é um curso de segundo semestre, onde estruturas de dados básicas são apresentadas e a linguagem C é usada. Portanto, este curso é o primeiro contato dos alunos com a linguagem C. Algoritmos e Estruturas de Dados II é um curso de terceiro semestre, onde estruturas de dados mais avançadas são apresentadas e a linguagem Python é usada.

Tabela 3.1: Conteúdo da disciplina Algoritmos e Estruturas de Dados I

Conteúdos da Disciplina
Estruturas de Dados Elementares
Operadores
Funções e Expressões Embutidas
Condicionais
Instruções Incondicionais e de Repetição
Subprogramas: funções, procedimentos
Parâmetros Locais e Globais
Recursão
Tipos Definidos pelo Programador
Estruturas de Dados Compostas: Vetores, Matrizes e Registros
Estruturas de Dados Dinâmicas: listas
Tipos Abstratos de Dados: filas, pilhas

Tabela 3.2: Conteúdo da disciplina Algoritmos e Estruturas de Dados II

Conteúdos da Disciplina
Noções de complexidade de algoritmos
Algoritmos de ordenação quadráticos, lineares e otimizados por custo
Algoritmos de busca sequencial e binária
Tabelas de dispersão (Hash Tables)
Processamento de Strings
Árvores (Árvores Binárias de Busca e Árvores Balanceadas)
Representação de grafos e algoritmos de percurso
Aplicações: ordenação topológica e caminho mínimo

A Tabela 3.3 apresenta os dados detalhados do *dataset* utilizado.

Este *dataset* contém todas as submissões dos estudantes durante o período. Nota-se a grande quantidade de submissões dos estudantes. A Tabela 3.4 apresenta um resumo das submissões de códigos realizadas pelos estudantes, agrupadas por turma.

Tabela 3.3: Dados do *dataset* utilizado.

Descrição	Quantidade
Códigos	259.497
Códigos em C	191.462
Códigos em Python	68.035
Estudantes	327
Turmas de Disciplinas	10

Tabela 3.4: Submissões de códigos por turma

Disciplina	ID	# Alunos	Linguagem	# Submissões
Algoritmos e Estruturas de Dados I	345	32	C	32,569
Algoritmos e Estruturas de Dados I	359	20	C	7,022
Algoritmos e Estruturas de Dados I	414	51	C	54,039
Algoritmos e Estruturas de Dados I	415	60	C	20,846
Algoritmos e Estruturas de Dados I	504	45	C	39,263
Algoritmos e Estruturas de Dados I	505	54	C	8,976
Algoritmos e Estruturas de Dados II	344	50	C	28,747
Algoritmos e Estruturas de Dados II	383	48	Python	13,670
Algoritmos e Estruturas de Dados II	437	48	Python	21,819
Algoritmos e Estruturas de Dados II	534	58	Python	32,546
Total	–	327	–	259,497

Tratamento dos dados

Após a obtenção dos dados, foi possível realizar a fase de tratamento. Como o objetivo do trabalho é verificar a qualidade do código dos estudantes quanto ao estilo de codificação, é importante realizar este tratamento para obter os dados necessários para a próxima fase.

Como informado anteriormente, o *dataset* contém o histórico de submissões dos estudantes. Este histórico é armazenado em pastas de acordo com o código identificador do aluno e da classe em arquivos **.log** e, a partir desses arquivos, podem-se recuperar os seguintes dados: (i) Código da turma; (ii) Código da atividade; (iii) Id do aluno; (iv) Data e hora da submissão; (v) Código enviado; (vi) Tempo de execução; (vii) Casos de teste; (vii) Saídas esperadas e obtidas; (ix) Erros e (x) Nota final (em porcentagem). Para este trabalho somente interessam os dados de **Código da Atividade**, **Id do Aluno**, **Código enviado** pelo aluno e **Nota final**. Um exemplo de arquivo log é apresentado na Figura 3.3.

Para recuperar essas informações, foi criado um *script* utilizando a linguagem de programação Python. O objetivo desse *script* é ler os arquivos de *log* e armazenar em pastas os arquivos separadamente. É importante ressaltar que, dentro de um arquivo de *log*, é possível ter uma quantidade **N** de códigos. Para cada um deles, foi gerado um arquivo diferente, contendo o conteúdo completo e a extensão do arquivo referente à linguagem de programação utilizada (.c para C e .py para Python). Além

```
== SUBMITION (2021-08-25 16:25:54)
-- CODE:
 // CÓDIGO OMITIDO
    }
-- EXECUTION TIME:
0.180045
-- TEST CASE 1:
---- input:
4
4 3 2 1
---- correct output:
4
---- user output:
4
-- GRADE:
100%
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
```

Figura 3.3: Exemplo de arquivo de log

disso, em um documento separado foram armazenadas as notas do código que foram atribuídas pelo analisador estático.

3.2 Segunda Etapa: Análise de Dados

Após a obtenção dos dados e do agrupamento das informações relevantes para o trabalho, foi possível realizar uma análise dos dados para responder às questões de pesquisa QP1, QP2, QP3 e QP4.

3.2.1 Utilização de Analisadores Estáticos

Como apresentado anteriormente, o *dataset* possui códigos nas linguagens de programação **C** e **Python**. A fim de analisar os códigos em ambas as linguagens, a ferramenta integradora de AEs PerfeQ, foi utilizada (os detalhes da sua criação e utilização estão disponíveis no Capítulo 4). Como estamos lidando com uma grande quantidade de dados, a ferramenta irá apresentar como saída um arquivo .csv contendo as informações dos códigos analisados juntamente com os valores das métricas.

A partir da saída da ferramenta integradora, foi necessário adicionar o campo nota no arquivo para que seja possível realizar, posteriormente, análises quanto a correlação entre as métricas e a nota final do código analisado.

3.2.2 Análise Estatística dos dados

Para análises estatísticas, foi utilizado o software SPSS Statistics na versão 22.0, juntamente com a linguagem de programação Python e as bibliotecas *matplotlib*, *pandas*, *seaborn*, *numpy* e *scipy*.

A partir disso, foi possível realizar análises descritivas dos dados, verificando os seus valores de média, desvio padrão, mediana e intervalos interquartis. Complementando, a representação visual dos dados a partir de diagramas de caixa (*boxplots*) permitiu verificar a distribuição dos dados.

Testes de normalidade, hipótese e correlação também foram realizados e suas saídas complementaram a análise dos dados, permitindo assim verificar o comportamento dos dados e como eles podem ser comparados entre si. Para os testes de hipótese entre dois grupos, que serão apresentados com mais detalhes no Capítulo 5, foram consideradas as seguintes premissas:

- H_0 (Hipótese nula): Não existem diferenças significativas entre os dois grupos analisados.
- H_1 (Hipótese alternativa): Existem diferenças significativas entre os dois grupos analisados.

Os testes de normalidade revelaram que nenhum dos dados apresenta uma distribuição normal. Portanto, utilizamos o teste de hipótese não-paramétricos de Mann-Whitney para comparação entre grupos e a correlação de Spearman para verificação de relações entre variáveis.

Em casos em que houve diferenças significativas nos testes de hipótese de Mann-Whitney, o seu tamanho de efeito foi avaliado considerando a medida de correlação posto-bisserial (r_{rb}). A depender do valor de r_{rb}, pode-se inferir o tamanho do efeito, conforme sumariza a Tabela 3.5.

Tabela 3.5: Tamanho de efeito apresentado a partir da medida de correlação posto-bisserial

Valor de r_{rb}	**Tamanho do efeito**
$r_{\text{rb}} < 0{,}1$	Muito pequeno
$0{,}1 \leq r_{\text{rb}} < 0{,}3$	Pequeno
$0{,}3 \leq r_{\text{rb}} < 0{,}5$	Médio
$r_{\text{rb}} \geq 0{,}5$	Grande

Capítulo 4

Métricas e Ferramentas

Neste capítulo, apresentamos métricas criadas para quantificar a qualidade de estilo de código e também o desenvolvimento de duas ferramentas concebidas ao longo deste trabalho, cuja finalidade é integrar o processo de análise da qualidade de código proposto nesta pesquisa: NamingCheck e PerfeQ. O primeiro é um Analisador Estático (AE) para C e Python cujo foco é a verificação de problemas de nomeação de variáveis e funções. Já o segundo é uma ferramenta integradora de Analisadores Estáticos, cujo objetivo é fornecer um feedback mais completo do código analisado a partir das saídas dos AEs e de valores de métricas, que foram projetadas e implementadas para sumarizar a informação sobre mensagens de aviso em um dado código.

4.1 Métricas

Para obter uma resposta a respeito da qualidade de código dos estudantes quanto ao seu estilo, foi necessário criar métricas para servir de base para as análises. O trabalho de Stegeman et al. (2016) apresenta rubricas para o apoio na análise e feedback de boas práticas de codificação. Dentre as destacadas, serviram de apoio as categorias que envolvem: (i) nomeação; e (ii) formatação. A Tabela 4.1 apresenta os níveis de cada categoria, onde no nível 1 representa um código com problemas de qualidade, enquanto o nível 4 reflete a aplicação de boas práticas de codificação.

Tabela 4.1: Rubrica de Stegeman et al. (2016) para avaliação da qualidade de código

Nível	Nomeação	Formatação
1	Nomes são ilegíveis, sem sentido ou enganosos.	A formatação está ausente ou é enganosa.
2	Os nomes descrevem a intenção do código, mas podem ser incompletos, longos, conter erros ortográficos ou uso inconsistente de maiúsculas e minúsculas.	Indentação, quebras de linha, espaçamento e uso de chaves destacam a estrutura pretendida, mas de forma inconsistente.
3	Os nomes descrevem a intenção do código, sendo completos, distintivos, concisos, corretamente escritos e com uso consistente de maiúsculas e minúsculas.	Indentação, quebras de linha, espaçamento e uso de chaves destacam a estrutura pretendida de forma consistente.
4	Todos os nomes no programa seguem um vocabulário consistente.	A formatação destaca partes semelhantes do código de forma clara.

Uma forma automatizada de realizar a análise de um código, alinhando-se tanto às suas convenções quanto aos critérios apresentados na rubrica, é a partir da utilização de um Analisador Estático. Sendo assim, a partir das saídas dos AEs – mensagens de aviso, juntamente com uma análise a respeito do conteúdo de um código-fonte, é possível ter acesso aos atributos apresentados na Tabela 4.2.

Tabela 4.2: Atributos referentes à análise do código.

Atributo
ID do código
Quantidade de Linhas de Código (LOC)
Quantidade de mensagens de avisos (total)
Quantidade de Variáveis no código
Quantidade de Funções
Quantidade de mensagens de avisos referentes a declaração de funções
Quantidade de mensagens de avisos referentes a declaração de variáveis
Quantidade de mensagens de avisos referentes à formatação de código

Tais atributos são apresentados em valores absolutos. Desta forma, esses valores precisam ser normalizados, seja em relação ao número de linhas de código, seja em relação ao total de ocorrências de declarações. Assim, a partir da normalização dos atributos previamente mostrados, criamos as métricas utilizadas neste trabalho, apresentadas na Tabela 4.3.

Tabela 4.3: Métricas criadas e seus significados.

Métrica	Significado
WPL	Quantidade de mensagens de aviso(geral) / LOC
VWPV	Quantidade de mensagens de aviso referentes a declaração de variáveis / Quantidade de variáveis
FWPF	Quantidade de mensagens de aviso referentes a declaração de funções / Quantidade de funções
FWPL	Quantidade de mensagens de aviso referentes a formatação de código / LOC

As métricas criadas buscam apresentar tanto uma visão geral do código analisado (**WPL** e **FWPL**) – informando problemas de formatação de código como uma visão mais objetiva a respeito da nomeação de variáveis e funções (**VWPV** e **FWPF**). Assim, a partir da utilização das métricas, é possível quantificar a qualidade de estilo do código analisado.

4.2 NamingCheck

Antes da criação da ferramenta, foi realizada uma análise dos principais analisadores estáticos no mercado, apresentados no Capítulo 2. O objetivo principal desta etapa foi a verificação de pontos fortes e fracos desses analisadores.

A partir dessa análise, foi identificado que os analisadores *Cpplint* e *Pylint* oferecem feedback válido quanto a formatação do código, especialmente contemplando as convenções das linguagens C e Python, respectivamente. Porém, também verificamos que os analisadores, principalmente o *Cpplint*, possuem limitações ao analisar a nomeação de identificadores para variáveis e funções. A ausência de feedback sobre identificadores pode causar problemas de legibilidade e manutenibilidade do

código, sendo recomendável que desenvolvedores considerem a escolha de identificadores como um elemento importante em código de alta qualidade (Van Der Werf et al., 2024).

Nesse contexto, foi necessário criar um novo analisador estático para as linguagens C e Python, nomeado NamingCheck[1], com foco na análise de identificadores de variáveis e funções seguindo os padrões das linguagens, como apresentados no Capítulo 2.

O NamingCheck foi escrito em Python. Ele recebe como entrada o código-fonte com as extensões **.c** ou **.py**. Em seguida, assim como no trabalho de Saliba et al. (2024), é criada uma árvore sintática parcial de cada unidade do código, permitindo verificar quando um trecho de código se refere à declaração de variáveis, *structs* e funções. Somente a nomeação de variáveis e funções é analisada. Por fim, a depender da análise, mensagens de aviso são apresentadas ao usuário, juntamente com a linha da ocorrência, como apresentado na Tabela 4.4:

Tabela 4.4: Exemplo de saída do Analisador Estático criado

Tipo	**Linha**	**Mensagem**
WARN	[9]	Variables names should be declared in snakecase.
WARN	[16]	Functions names should be declared in snakecase.
WARN	[22]	Functions names should be declared in snakecase.
WARN	[28]	Functions names should be declared in snakecase.
WARN	[36]	Variables names should be declared in snakecase.

Todas as mensagens que podem ser apresentadas pelo Analisador Estático são apresentadas na Tabela 4.5. As mensagens de aviso do analisador estático desenvolvido foram baseadas em trabalhos prévios Doland e Valett (1994); Van Rossum et al. (2001) e na documentação do CS50[2], que apresentam as convenções de estilo de código para C e Python.

Tabela 4.5: Mensagens de aviso apresentadas pelo NamingCheck

Mensagem
Structs should be declared in lowercase.
Enums should be declared in pascalcase.
All constants should be declared in uppercase.
Functions names should be declared in snakecase.
If you initialize one variable, you should initialize the others.
Pointer variables should not be declared with no pointers variables.
Variables names should be declared in snake case.
Variables names should have length greater than one.

[1]https://github.com/franciscotis/NamingCheck
[2]https://github.com/cs50/cs50.readthedocs.io

Um detalhe a respeito da última mensagem da Tabela 4.5 é que foram excluídos os casos em que as variáveis são declaradas como i, j ou k e são utilizadas em estruturas de repetição, já que a maioria dos professores utiliza essas variáveis como contadores em loops.

4.3 PerfeQ

A fim de realizar uma integração entre Analisadores Estáticos existentes e o Analisador Estático criado, além de disponibilizar métricas relacionadas à qualidade do estilo de código, a ferramenta PerfeQ[3] foi criada.

Ela foi desenvolvida para, primeiramente, encapsular a informação de vários analisadores estáticos, incluindo o novo analisador NamingCheck. Além disso, ao apresentar informações sobre um código-fonte tanto de forma analítica (mensagens de aviso) como sintética (métricas de qualidade), PerfeQ serve como uma ferramenta de apoio à verificação da qualidade de código de estudantes com foco na aderência às convenções de codificação.

A ferramenta foi desenvolvida utilizando a linguagem de programação Python e integra os AEs *Pylint* e *Cpplint*, além do NamingCheck. Adicionalmente, ela quantifica a avaliação do código a partir de métricas. A ferramenta tem suporte para as linguagens Python e C, podendo ser utilizada tanto por pesquisadores quanto por professores e estudantes.

A Figura 4.1 apresenta uma visão arquitetônica da dinâmica da ferramenta desenvolvida.

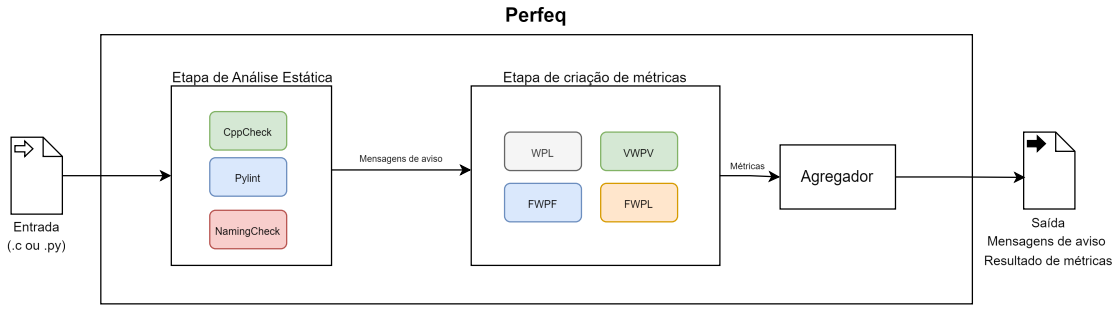

Figura 4.1: Visão arquitetônica da dinâmica da ferramenta PerfeQ

O sistema recebe como entrada a localização do código-fonte sobre o qual deseja-se realizar a análise. Na Etapa de Análise Estática, o código passa pelo módulo integrador de Analisadores Estáticos. A depender da linguagem do código, o analisador Pylint ou o CppCheck será acionado para gerar as mensagens de aviso referentes a

[3]http://github.com/franciscotis/PerfeQ

formatação de código. Em seguida, será acionado o NamingCheck, que gera mensagens a respeito da qualidade dos identificadores das variáveis e funções usadas no código. Esta etapa integra na saída as mensagens de aviso fornecidas pelos AEs.

Em seguida, o código de entrada e as mensagens de aviso geradas são encaminhados para a Etapa de Criação de Métricas. Nesta segunda etapa, são primeiramente computados os atributos referentes à análise do código (como apresentado na Tabela 4.2). Em seguida, a partir dos atributos computados, são geradas as métricas apresentadas na Tabela 4.3. Como saída, o integrador exibe os valores das métricas no terminal e gera um arquivo .csv contendo essas métricas, além de informações sobre o código, como número de linhas, variáveis e funções.

Para criar as métricas do PerfeQ e separar os tipos das mensagens de aviso disponibilizadas pelos AEs, foi necessário analisar também a documentação de cada Analisador Estático e identificar os grupos de mensagens de avisos que tratam de qualidade do código em relação ao estilo de codificação. Em **Pylint**, as mensagens apresentadas na Tabela 4.6 foram consideradas como mensagens referentes à declaração de variáveis.

Tabela 4.6: Mensagens de aviso em Pylint referentes à declaração de variáveis.

Mensagem de aviso
Constant name doesn't conform to UPPER_CASE naming style (invalid-name) / C0103
Variable name doesn't conform to snake_case naming style (invalid-name) / C0103

Ainda no **Pylint**, a Tabela 4.7 apresenta as mensagens que foram consideradas como mensagens referentes à declaração de funções. As outras mensagens foram consideradas de formatação de código.

Tabela 4.7: Mensagens de aviso em Pylint referentes à declaração de funções.

Mensagem de aviso
Function name doesn't conform to snake_case naming style / C0103

Já para C, todas as mensagens apresentadas pelo **Cpplint** foram consideradas como mensagens referentes à formatação de código. A partir da utilização do NamingCheck, as mensagens apresentadas na Tabela 4.8 foram consideradas como mensagens referentes à declaração de variáveis.

Tabela 4.8: Mensagens de aviso do NamingCheck referentes a declaração de variáveis

Mensagem de aviso
Structs should be declared in lowercase.
Enums should be declared in pascalcase.
All constants should be declared in uppercase.
Variables names should be declared in snake case.
Variables names should have length greater than one.

Para considerar uma mensagem como referente à declaração de funções, foi utilizada a mensagem apresentada na Tabela 4.9:

Tabela 4.9: Mensagens de aviso do NamingCheck referentes à declaração de funções.

Mensagem de aviso
Functions names should be declared in snakecase.

4.3.1 Cenários de Uso

Para utilizar a ferramenta, é necessário ter instalado o interpretador do Python na versão 3.12.2 ou posterior. Primeiramente, o usuário deve informar a localização da pasta ou do arquivo que deseja analisar. Caso seja informada uma pasta, todos os arquivos presentes nela serão analisados. A seguir, apresentamos como a ferramenta é utilizada de modo geral e como três tipos de usuário (professores, estudantes e pesquisadores) podem se beneficiar de seu uso.

Funcionamento

Para a sua execução, o usuário deve informar a localização absoluta do código fonte que deseja analisar como argumento do programa perfeQ.py. Em seguida, o programa realiza a análise estática do código desejado e apresenta no console as mensagens de aviso geradas, como ilustrado na Figura 4.2.

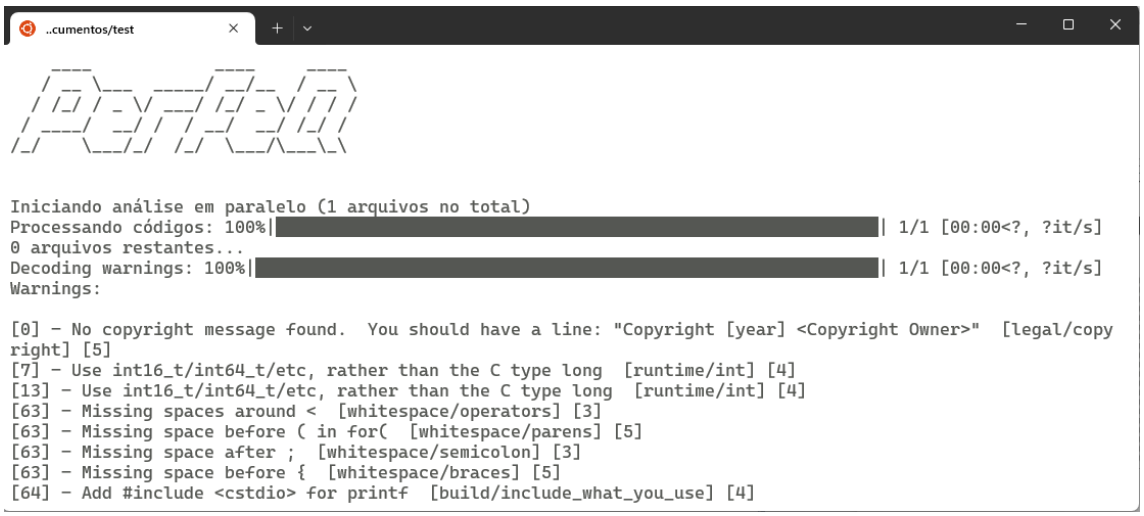

Figura 4.2: Mensagens de aviso de analisadores estáticos apresentados pelo PerfeQ

Em seguida, os valores, em percentuais, das métricas de qualidade também são apresentados, como ilustra a Figura 4.3.

A Figura 4.4 apresenta o arquivo *.csv* que também é gerado. Nele, é apresentado um resultado mais detalhado da análise.

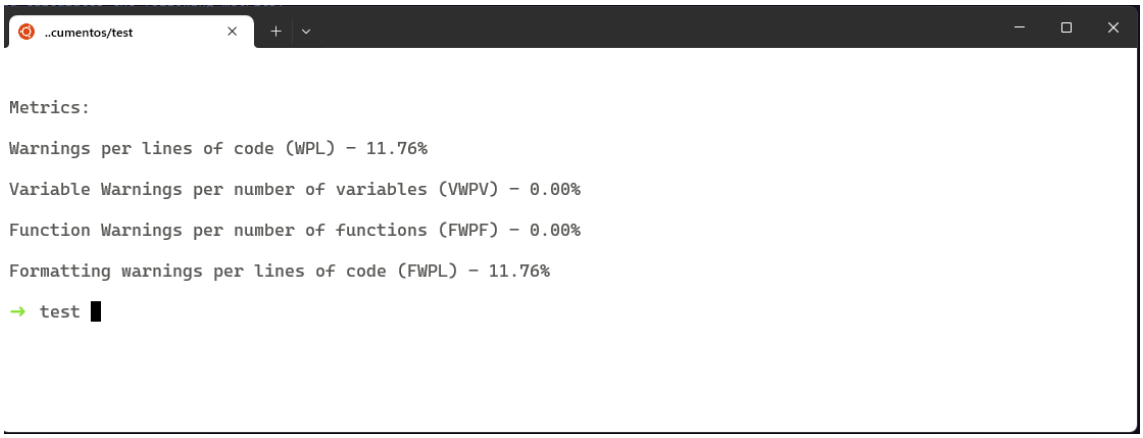

Figura 4.3: Apresentação das métricas pelo PerfeQ

```
code_id,LOC,warnings_qty,WPL,variable_warnings_qty,variables_qty,VWPV,function_warnings_qty,functions_qty,FWPF,formatting_warnings_qty,FWPL
1438_3460_3358_1.c,0,1,0.00,0,0,0.00,0,0,0.00,1,0.00
1438_3428_3325_4.c,28,49,175.00,0,1,0.00,0,1,0.00,49,175.00
1438_3417_3273_11.c,13,17,130.77,0,2,0.00,0,1,0.00,17,130.77
1438_3460_3327_1.c,71,122,171.83,2,11,18.18,0,4,0.00,120,169.01
1438_3460_3357_15.c,35,63,180.00,3,7,42.86,0,1,0.00,60,171.43
1438_3417_3783_0.c,8,11,137.50,0,0,0.00,0,1,0.00,11,137.50
1438_3428_3291_1.c,15,27,180.00,2,6,33.33,0,1,0.00,25,166.67
1438_3460_3336_0.c,33,55,166.67,1,2,50.00,0,2,0.00,54,163.64
1438_3428_3287_2.c,34,64,188.24,1,6,16.67,0,1,0.00,63,185.29
1438_3417_3260_4.c,19,20,105.26,0,0,0.00,0,1,0.00,20,105.26
1438_3417_3242_8.c,14,18,128.57,1,4,25.00,0,1,0.00,17,121.43
1438_3417_3276_1.c,18,34,188.89,0,4,0.00,0,1,0.00,34,188.89
1438_3460_3328_0.c,27,38,140.74,1,3,33.33,0,2,0.00,37,137.04
1438_3428_3271_2.c,89,163,183.15,0,4,0.00,0,2,0.00,163,183.15
1438_3460_5219_1.c,42,62,147.62,0,4,0.00,4,5,80.00,58,138.10
```

Figura 4.4: Exemplo de arquivo .csv com os resultados detalhados da análise realizada pelo PerfeQ

Caso o usuário opte por informar o caminho de uma pasta em vez de um arquivo, a ferramenta realiza a análise de todos os arquivos contidos nesta pasta. Neste caso, somente o arquivo de resultados (.csv) é gerado.

4.3.2 Utilização por professores

A ferramenta PerfeQ pode ser utilizada por professores que buscam analisar a qualidade de código de um único estudante ou de vários estudantes. É importante ressaltar que, assim como apresentado anteriormente, devido ao número de alunos, é pouco provável que o professor tenha tempo de apresentar um feedback completo e individual para os estudantes quanto à qualidade dos seus códigos. O professor pode utilizar outras ferramentas, como ambientes de juiz online, para verificar a

corretude do código e, também, utilizar PerfeQ para analisar a qualidade do código. Os resultados das métricas apresentadas ao final da análise podem servir como base para avaliação. Finalmente, o professor pode utilizar o arquivo de saída gerado e realizar processamento adicional para verificar quais são os problemas de estilo encontrados pelos estudantes durante a codificação. Por exemplo, pode-se verificar que os estudantes possuem problemas na nomeação de variáveis e funções, e em seguida reforçar esse assunto em sala de aula.

4.3.3 Utilização por estudantes

A ferramenta também permite que estudantes possam obter feedback a respeito da qualidade de seus códigos. Por exemplo, podem obter as mensagens de aviso dos analisadores estáticos que estão integrados na ferramenta, além dos valores das métricas. A partir dessas informações, os estudantes podem melhorar seus códigos, diagnosticando melhor em que aspectos eles devem trabalhar.

4.3.4 Utilização por pesquisadores

Por fim, pesquisadores também podem se beneficiar do uso da ferramenta. Há um grande potencial para pesquisa em analíticas de aprendizagem (em inglês, *learning analytics*). Com isso, seria possível analisar a evolução do aluno durante o curso e verificar todo o processo de ensino e aprendizagem. Ao analisar uma grande quantidade de dados, é disponibilizado um arquivo de saída contendo os atributos dos códigos analisados juntamente com os valores das métricas. A partir daí, o pesquisador pode utilizar outras ferramentas de análise de dados e gerar gráficos descritivos como, por exemplo, *boxplots* e outros diagramas, além de poder realizar análises estatísticas mais detalhadas.

Capítulo 5

Resultados

Neste capítulo, apresentamos os resultados do presente trabalho. A partir das métricas criadas e calculadas na seção anterior, foi possível realizar análises estatísticas para obter resultados quanto à qualidade do código dos estudantes. Para isso, foi realizado um estudo comparativo entre dois casos: o primeiro, com os resultados obtidos a partir da análise das métricas das submissões parciais dos estudantes (da primeira submissão até a penúltima); o segundo, com os resultados obtidos a partir da análise das métricas da submissão final dos estudantes. O objetivo deste estudo é verificar se existe uma evolução na qualidade de estilo de código no processo de codificação, desde a primeira submissão até a última.

Para o primeiro caso que contém as submissões parciais (todas as submissões, excluindo a final), foram analisados 245.773 códigos, sendo 180.486 referentes a códigos escritos em C e 65.287 a códigos escritos em Python. Já para o segundo caso, contendo somente as submissões finais, foram analisados 13.725 códigos, sendo 10.976 em C e 2.748 em Python. A partir da utilização do teste de Kolmogorov-Smirnov, pôde-se concluir que nenhuma das métricas apresenta uma distribuição normal ($valor\text{-}p < 0.05$).

Para uma melhor organização dos dados, separamos a análise e os resultados por linguagem de programação: C e Python. Essa separação foi necessária pois os códigos em cada linguagem são analisados por analisadores estáticos diferentes – produzindo, assim, saídas diferentes. Posteriormente, apresentamos uma comparação das linguagens para compreender melhor, a partir do *dataset* utilizado, diferenças na qualidade de estilo de código.

A partir desses casos, também realizamos uma análise dos códigos com dois grupos: (i) os que obtiveram uma nota igual ou superior a 50%; (ii) os que obtiveram uma nota inferior a 50%. Para isso, foi necessário somente utilizar as submissões finais. Para a análise, é apresentado um diagrama de caixa para ajudar na verificação da dispersão dos dados e também são apresentados os resultados do teste de hipótese para verificar se as submissões finais que obtiveram maiores notas também obtiveram melhores resultados nos valores das métricas.

A Tabela 5.1 apresenta um resumo das variáveis independentes e dependentes analisadas no estudo.

Tabela 5.1: Resumo das variáveis independentes e dependentes analisadas no estudo

Caso	Variável Independente	Variável Dependente
Submissão parcial vs. Submissão final	Tipo de submissão	Métricas do código
Nota $\geq$ 50% vs. Nota $<$ 50%	Desempenho do estudante	Métricas do código
Códigos em Python vs. Códigos em C	Linguagem de programação	Métricas do código

A seguir, para cada métrica, serão apresentados os resultados correspondentes.

5.1 Visão Geral

As Tabelas 5.2 e 5.3 apresentam as 10 mensagens de aviso que mais aparecem nos códigos observados, considerando as submissões parciais em C e Python. Todas as mensagens de aviso observadas estão disponíveis online[1]. As mensagens de aviso foram separadas em 4 categorias: (i) Formatação; (ii) Nomeação de Funções; (iii) Nomeação de Variáveis; (iv) Outros (fora do escopo deste trabalho).

Já as Figuras 5.1 e 5.2 apresentam, respectivamente, os gráficos com as frequências das categorias identificadas em todas as mensagens de aviso analisadas.

Tabela 5.2: Mensagens de aviso mais frequentes encontradas nos códigos observados em C, considerando as submissões parciais

Mensagem de aviso	Quantidade de Ocorrências	Categoria
Tab found; better to use spaces	5.613.231	Formatação
Missing space before {	1.366.562	Formatação
Line ends in whitespace. Consider deleting these extra spaces.	856.630	Formatação
Missing space after ,	468.564	Formatação
Missing spaces around =	272.761	Formatação
Functions names should be lower cased.	215.569	Nomeação de Funções
No copyright message found. You should have a line	207.830	Formatação
Should have a space between // and comment	205.525	Formatação
Weird number of spaces at line-start. Are you using a 2-space indent?	182.831	Formatação
Missing space before (in if(	176.406	Formatação

[1]https://github.com/franciscotis/warning-messages-mestrado

Tabela 5.3: Mensagens de aviso mais frequentes encontradas nos códigos observados em Python, considerando as submissões parciais.

Mensagem de aviso	Quantidade de Ocorrências	Categoria
Bad indentation. Found 1 spaces, expected 4 (bad-indentation)	692.148	Formatação
Bad indentation. Found 2 spaces, expected 8 (bad-indentation)	517.386	Formatação
Trailing whitespace (trailing-whitespace)	300.016	Formatação
Bad indentation. Found 3 spaces, expected 12 (bad-indentation)	188.023	Formatação
Missing function or method docstring (missing-function-docstring)	154.573	Outro
Use of eval (eval-used)	76.192	Outro
Missing module docstring (missing-module-docstring)	69.021	Outro
Bad indentation. Found 4 spaces, expected 16 (bad-indentation)	61.825	Formatação
Final newline missing (missing-final-newline)	41.126	Formatação
Unnecessary parens after 'if' keyword (superfluous-parens)	35.650	Outro

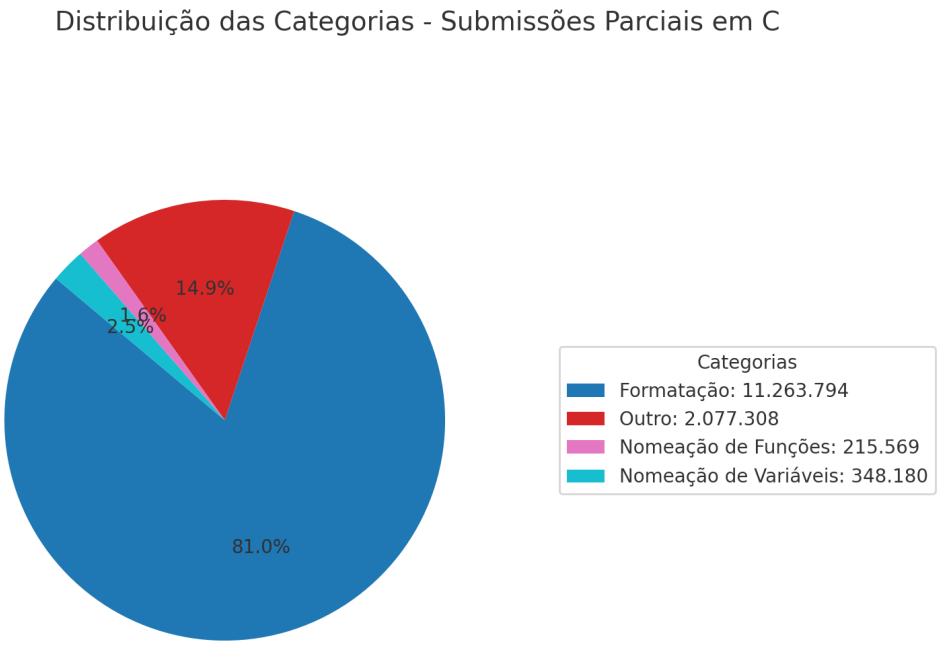

Figura 5.1: Distribuições das mensagens de aviso em submissões parciais em C.

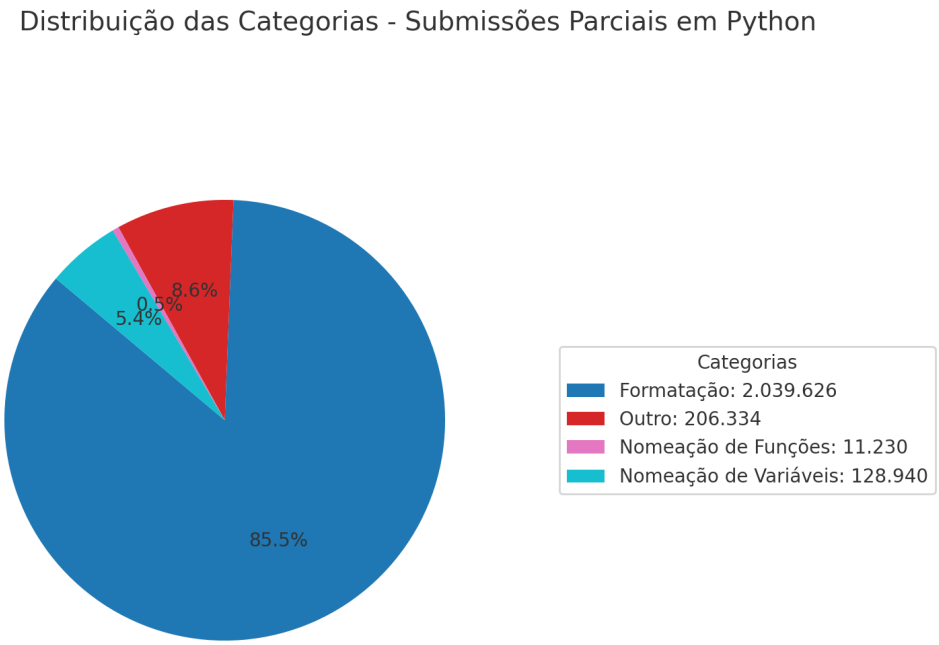

Figura 5.2: Distribuições das mensagens de aviso em submissões parciais em Python.

Percebe-se que a maioria dos avisos tem relação com a formatação do código em ambas as linguagens.

As Tabelas 5.4 e 5.5 apresentam as 10 mensagens de aviso que mais aparecem nos códigos observados, considerando a submissão final em C e Python. Todas as mensagens de aviso observadas estão disponíveis online[2]. Já as Figuras 5.3 e 5.4 apresentam, respectivamente, os gráficos com as frequências das categorias identificadas em todas as mensagens de aviso analisadas.

Tabela 5.4: Mensagens de aviso mais frequentes encontradas nos códigos observados em C, considerando a submissão final.

Mensagem de aviso	**Quantidade de Ocorrências**	**Categoria**
Tab found; better to use spaces	215.051	Formatação
Missing space before {	57.022	Formatação
Line ends in whitespace. Consider deleting these extra spaces.	38.475	Formatação
Missing space after ,	22.775	Formatação
No copyright message found. You should have a line	10.996	Outro
Weird number of spaces at line-start. Are you using a 2-space indent?	10.481	Formatação
Missing spaces around =	10.345	Formatação
Missing space before (in if(	9.956	Formatação
Functions names should be lower cased.	8.572	Nomeação de Funções
Could not find a newline character at the end of the file.	7.998	Formatação

[2]https://github.com/franciscotis/warning-messages-mestrado

Tabela 5.5: Mensagens de aviso mais frequentes encontradas nos códigos observados em Python, considerando a submissão final.

Mensagem de aviso	Quantidade de Ocorrências	Categoria
Bad indentation. Found 1 spaces, expected 4 (bad-indentation)	23.925	Formatação
Bad indentation. Found 2 spaces, expected 8 (bad-indentation)	15.910	Formatação
Trailing whitespace (trailing-whitespace)	9.828	Formatação
Missing function or method docstring (missing-function-docstring)	5.811	Outro
Bad indentation. Found 3 spaces, expected 12 (bad-indentation)	4.993	Formatação
Use of eval (eval-used)	3.044	Outro
Missing module docstring (missing-module-docstring)	2.696	Outro
Final newline missing (missing-final-newline)	1.671	Formatação
Bad indentation. Found 4 spaces, expected 16 (bad-indentation)	1.533	Formatação
Unnecessary parens after 'if' keyword (superfluous-parens)	1.316	Outro

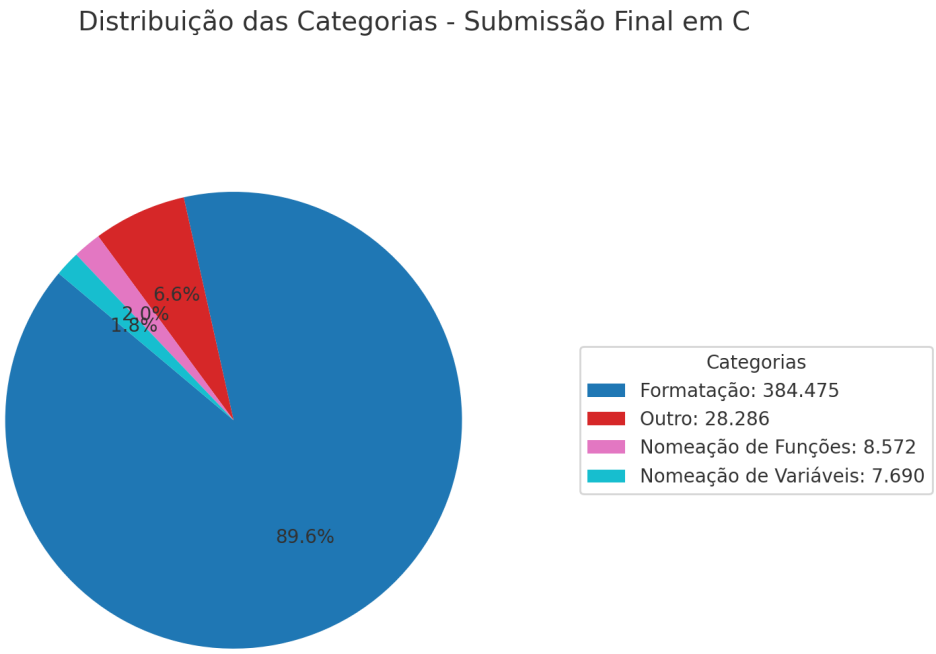

Figura 5.3: Distribuições das mensagens de aviso em submissões finais em C.

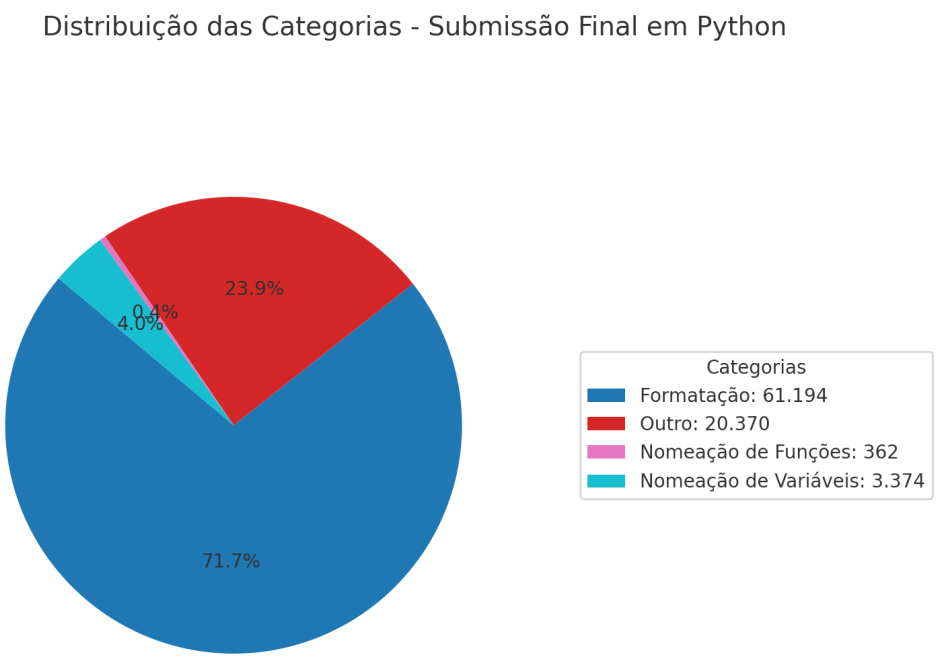

Figura 5.4: Distribuições das mensagens de aviso em submissões finais em Python.

Percebe-se, na versão final, que a maioria das mensagens de aviso tem relação com a formatação do código em ambas as linguagens, assim como no caso anterior.

Para apresentar os resultados de forma mais organizada, estes foram divididos em dois grupos: o primeiro reúne os dados em C e o segundo, os dados em Python. Para cada grupo será apresentado o resultado da análise das métricas para tipo de submissão (parcial ou final), além dos resultados dos testes de hipótese. Em seguida, será apresentada a comparação dos resultados em C e Python e, por fim, as correlações calculadas.

5.2 Análise em C

Nesta seção, são apresentados os resultados do estudo comparativo entre os dois casos com códigos escritos em C. As estatísticas descritivas de cada métrica são exibidas na Tabela 5.6. Nota-se que a coluna Tipo refere-se ao tipo de submissão na qual a métrica está analisada (final ou parcial).

Tabela 5.6: Dados descritivos das métricas para as submissões parciais e final em C.

Métrica	Tipo	$\bar{x}$	s	Md	AIQ
WPL	Parcial	1,129	0,431	1,176	0,524
WPL	Final	1,072	0,457	1,118	0,636
VWPV	Parcial	0,269	0,331	0,200	0,375
VWPV	Final	0,233	0,329	0,143	0,333
FWPF	Parcial	0,203	0,568	0,000	0,375
FWPF	Final	0,203	0,499	0,000	0,167
FWPL	Parcial	1,085	0,426	1,125	0,517
FWPL	Final	1,033	0,452	1,073	0,621

A Tabela 5.7 apresenta as estatísticas descritivas para a análise dos códigos que obtiveram uma nota igual ou superior a 50% e os que obtiveram uma nota inferior a 50%.

Tabela 5.7: Dados descritivos das métricas para os códigos com notas superiores ou inferiores ao limiar de 50% em C.

Métrica	Tipo	$\bar{x}$	s	Md	AIQ
WPL	Nota $\geq$ 50%	1,083	0,452	1,125	0,629
WPL	Nota $<$ 50%	1,021	0,478	1,058	0,648
VWPV	Nota $\geq$ 50%	0,231	0,328	0,143	0,333
VWPV	Nota $<$ 50%	0,245	0,335	0,167	0,333
FWPF	Nota $\geq$ 50%	0,170	0,374	0,000	0,000
FWPF	Nota $<$ 50%	0,220	0,399	0,000	0,333
FWPL	Nota $\geq$ 50%	1,045	0,446	1,083	0,619
FWPL	Nota $<$ 50%	0,979	0,474	1,000	0,633

5.2.1 Métrica WPL

A seguir, os resultados das análises para a métrica referente a quantidade total de *warnings* por linhas de código.

Submissões Parciais vs. Submissões Finais

Os *boxplots* da Figura 5.5 apresentam em mais detalhes os resultados da análise da métrica nos códigos. É possível verificar que ambos os casos apresentam distribuições com medianas e dispersões similares, com os dados concentrados em torno da média e mediana.

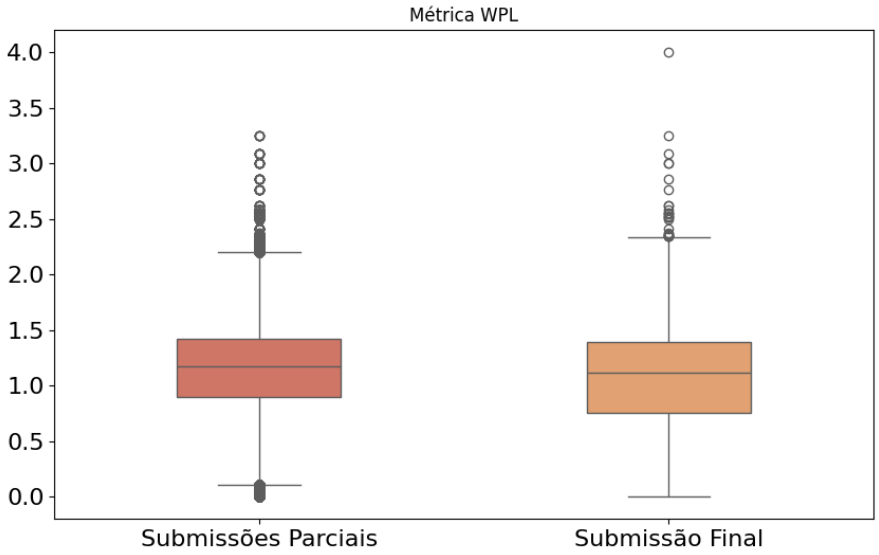

Figura 5.5: Boxplots para a métrica WPL nas submissões parciais e final em C.

A análise do *boxplot* juntamente com os seus dados descritivos sugere que os códigos analisados em ambos os casos possuem uma quantidade alta de avisos gerados pelos analisadores estáticos. Por meio do teste de Mann-Whitney, encontramos que os valores das métricas variam significativamente entre os dois grupos ($p < 0{,}001$). Porém, o tamanho de seu efeito é considerado muito pequeno ($r_{rb} = 0{,}076$).

Códigos com notas acima de 50% vs. Códigos com notas abaixo de 50%

Os *boxplots* da Figura 5.6 apresentam com mais detalhes os resultados da análise.

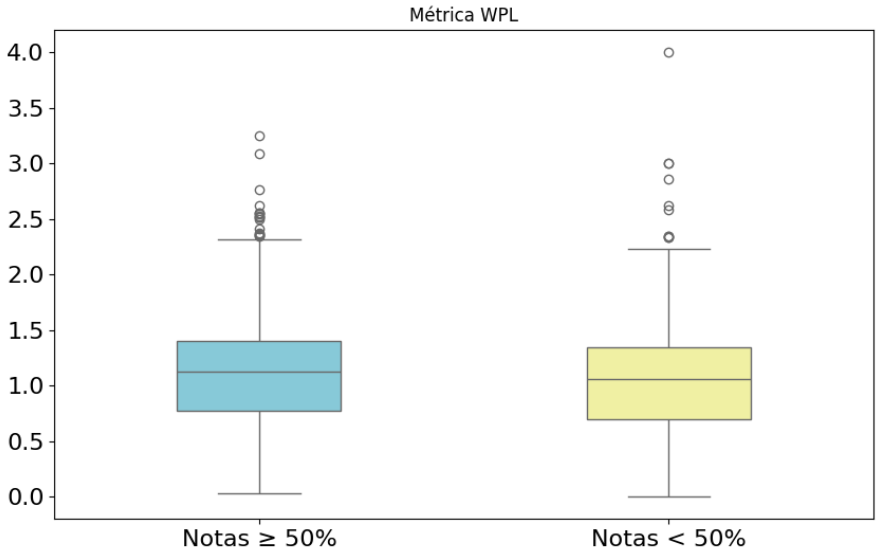

Figura 5.6: Boxplots para a métrica WPL para os códigos com notas superiores e inferiores ao limiar de 50% em C.

É possível verificar que ambos os casos apresentam distribuições com medianas e dispersões similares. Além disso, os resultados sugerem que nas duas situações, os códigos apresentam uma alta taxa de mensagens de aviso.

Por meio do teste de Mann-Whitney, encontramos que existem diferenças significativas entre os grupos de códigos com notas maiores ou iguais a 50% e aqueles com notas inferiores a 50% ($p < 0{,}001$). Porém, o tamanho de seu efeito é considerado muito pequeno ($r_{rb} < 0{,}078$).

5.2.2 Métrica VWPV

A seguir, os resultados das análises para a métrica referente a quantidade total de mensagens de aviso de variáveis por quantidade de variáveis.

Submissões Parciais vs. Submissões Finais

Os *boxplots* da Figura 5.7 apresentam em mais detalhes os resultados da análise da métrica nos códigos. É possível verificar que existe uma diferença entre as medianas dos dois grupos, indicando que os códigos nas suas versões finais apresentam menos problemas relacionados com a declaração de variáveis.

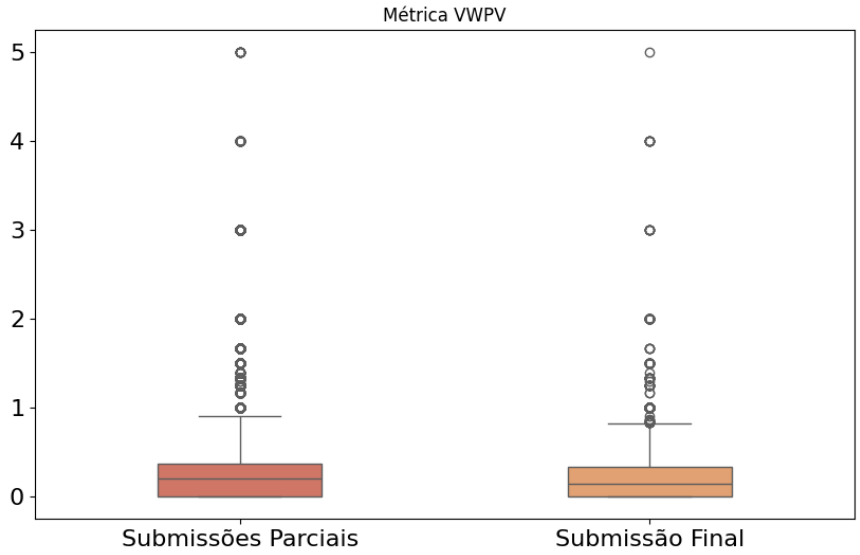

Figura 5.7: Boxplots para a métrica VWPV nas submissões parciais e final em C.

É possível verificar que ambos os casos apresentam dispersões similares, porém a mediana da submissão final apresenta um valor menor em comparação com a submissão parciais. De modo geral, existem poucos problemas relacionados à nomeação de variáveis. O teste de Mann-Whitney apresenta que os valores das métricas variam significativamente entre os dois grupos ($p < 0{,}001$). Porém, o tamanho de seu efeito é considerado muito pequeno ($r_{rb} = 0{,}093$).

Códigos com notas acima de 50% vs. Códigos com notas abaixo de 50%

Os *boxplots* da Figura 5.8 apresentam com mais detalhes os resultados da análise.

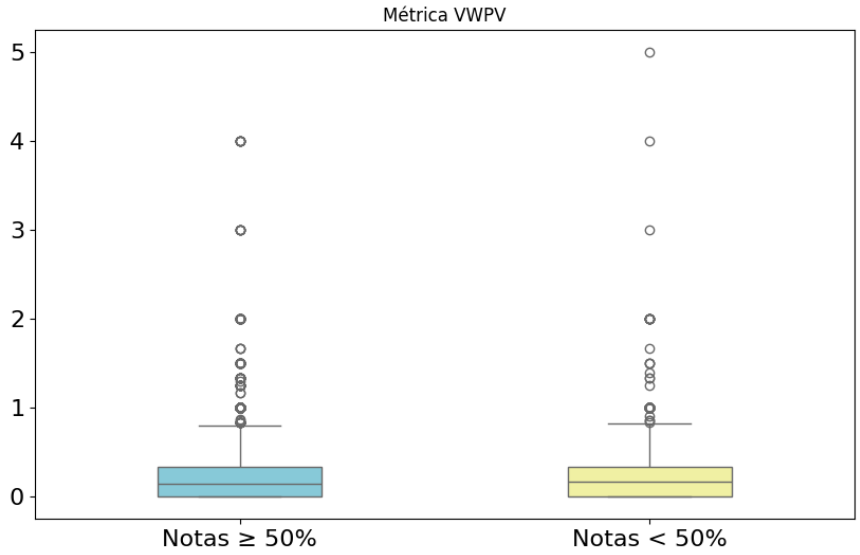

Figura 5.8: Boxplots para a métrica VWPV para os códigos com notas superiores e inferiores ao limiar de 50% em C.

É possível verificar que ambos os casos apresentam dispersões similares, com o grupo dos códigos com nota $< 50\%$ possuindo um valor de mediana inferior que o grupo dos códigos com nota $\geq 50\%$. Além disso, verifica-se que existem poucos problemas relacionados a declaração de variáveis em ambos os grupos.

Por meio do teste de Mann-Whitney, encontramos que existem diferenças significativas entre os grupos de códigos com notas maiores ou iguais a 50% e aqueles com notas inferiores a 50% ($p < 0{,}001$). Porém, o tamanho de seu efeito é considerado muito pequeno ($r_{rb} = 0{,}054$).

5.2.3 Métrica FWPF

A seguir, os resultados das análises para a métrica referente a quantidade total de mensagens de aviso de funções por quantidade de funções.

Submissões Parciais vs. Submissões Finais

Os *boxplots* da Figura 5.9 apresentam em mais detalhes os resultados da análise da métrica nos códigos. Verifica-se uma maior dispersão dos dados no primeiro grupo, porém a mediana dos dois se iguala.

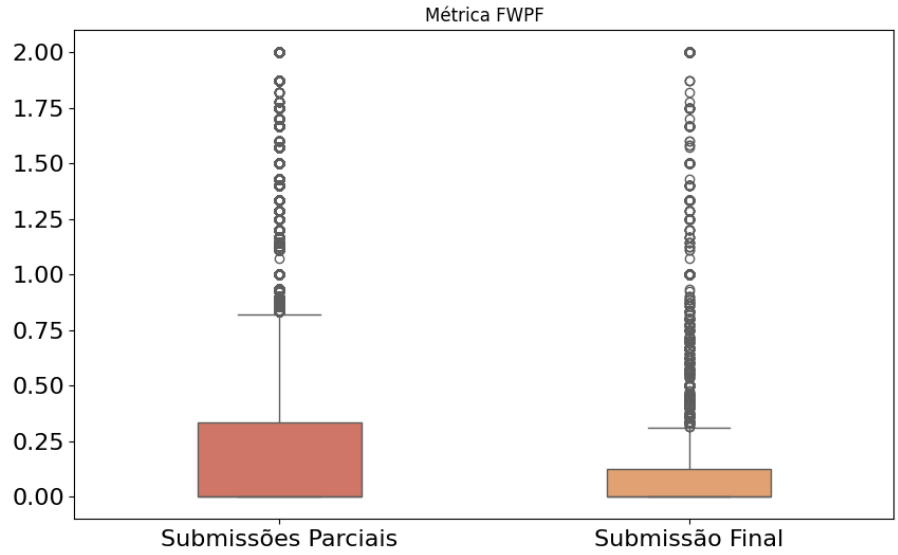

Figura 5.9: Boxplots para a métrica FWPF nas submissões parciais e final em C.

É possível verificar que a maioria dos códigos observados, em ambos os casos, não apresenta muitos problemas relacionados com a nomeação de funções. Porém, as submissões parciais apresentam mais problemas em relação à submissão final. O teste de Mann-Whitney apresenta que os valores das métricas variam significativamente entre os dois grupos ($p < 0{,}001$). Porém, o tamanho de seu efeito é considerado muito pequeno ($r_{rb} = 0{,}071$).

Códigos com notas acima de 50% vs. Códigos com notas abaixo de 50%

Os *boxplots* da Figura 5.10 apresentam com mais detalhes os resultados da análise.

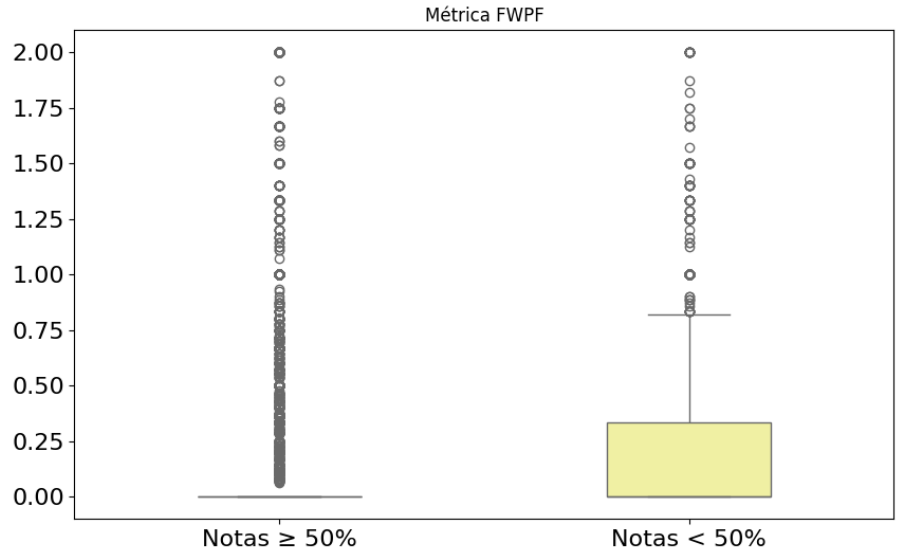

Figura 5.10: Boxplots para a métrica FWPF para os códigos com notas superiores e inferiores ao limiar de 50% em C.

É possível verificar que o grupo dos códigos com nota $< 50\%$ possui uma distribuição dos dados maior, porém a mediana de ambos os grupos seja semelhante. O resultado indica que o grupo de códigos com notas inferiores a 50% apresenta mais problemas de nomeação de funções.

Por meio do teste de Mann-Whitney, encontramos que existem diferenças significativas entre os grupos de códigos com notas maiores ou iguais a 50% e aqueles com notas inferiores a 50% ($p < 0{,}001$). Porém, o tamanho de seu efeito é considerado muito pequeno ($r_{\text{rb}} = 0{,}089$).

5.2.4 Métrica FWPL

A seguir, os resultados das análises para a métrica referente a quantidade de mensagens de aviso de formatação por linhas de código.

Submissões Parciais vs. Submissões Finais

Os *boxplots* da Figura 5.11 apresentam em mais detalhes os resultados da análise da métrica nos códigos. Verifica-se uma maior dispersão dos dados no segundo grupo, porém a mediana é maior no primeiro grupo.

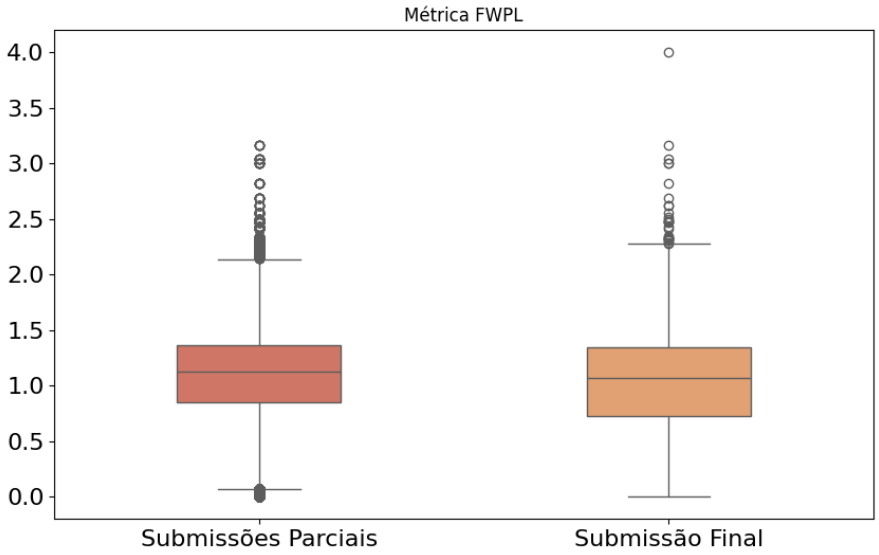

Figura 5.11: Boxplots para a métrica FWPL nas submissões parciais e final em C.

Os dados indicam que existe uma alta incidência de avisos relacionados à formatação nos códigos observados em ambos os casos, sendo maior no primeiro caso. O teste de Mann-Whitney apresenta que os valores das métricas variam significativamente entre os dois grupos ($p < 0{,}001$). Porém, o tamanho de seu efeito é considerado muito pequeno ($r_{rb} = 0{,}071$).

Códigos com notas acima de 50% vs. Códigos com notas abaixo de 50%

Os *boxplots* da Figura 5.12 apresentam com mais detalhes os resultados da análise.

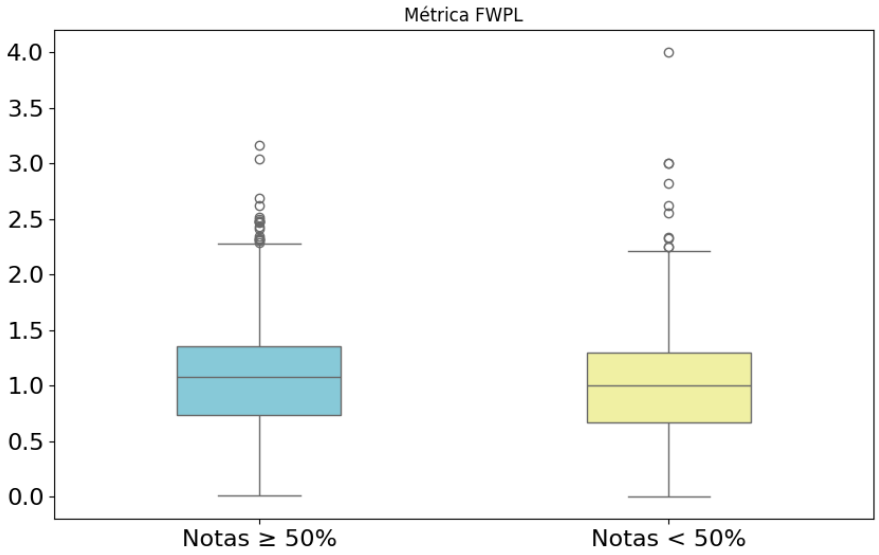

Figura 5.12: Boxplots para a métrica FWPL para os códigos com notas superiores e inferiores ao limiar de 50% em C.

É possível verificar que ambos os casos apresentam distribuições com medianas e dispersões similares. Porém, o resultado apresenta também que os códigos que possuem notas maiores ou iguais a 50% apresentam mais problemas relacionados à formatação de código.

Também por meio do teste de Mann-Whitney, encontramos que existem diferenças significativas entre os grupos de códigos com notas maiores ou iguais a 50% e aqueles com notas inferiores a 50% ($p < 0{,}001$). Porém, o tamanho de seu efeito é considerado muito pequeno ($r_{\text{rb}} = 0{,}083$).

5.3 Análise em Python

Nesta seção, são apresentados os resultados do estudo comparativo entre os dois casos com códigos escritos em Python. As estatísticas descritivas de cada métrica são exibidas na Tabela 5.8. Nota-se que a coluna Tipo refere-se ao tipo de submissão na qual a métrica está analisando (final ou parcial).

Tabela 5.8: Dados descritivos das métricas para as submissões parciais e final em Python

Métrica	Tipo	$\bar{x}$	s	Md	AIQ
WPL	Parcial	1,058	0,320	1,103	0,289
WPL	Final	1,180	0,420	1,254	0,464
VWPV	Parcial	0,242	0,256	0,182	0,333
VWPV	Final	0,807	0,418	0,800	0,444
FWPF	Parcial	0,179	0,325	0,000	0,222
FWPF	Final	0,360	0,602	0,000	0,600
FWPL	Parcial	0,975	0,299	1,033	0,276
FWPL	Final	0,911	0,383	1,000	0,4364

A Tabela 5.9 apresenta as estatísticas descritivas para a análise dos códigos que obtiveram uma nota igual ou superior a 50% e os que obtiveram uma nota inferior a 50%.

Tabela 5.9: Dados descritivos das métricas para os códigos com notas superiores ou inferiores ao limiar de 50% em Python

Métrica	Tipo	$\bar{x}$	s	Md	AIQ
WPL	Nota $\geq$ 50%	1,196	0,378	1,256	0,429
WPL	Nota $<$ 50%	1,155	0,479	1,250	0,517
VWPV	Nota $\geq$ 50%	0,811	0,402	0,800	0,429
VWPV	Nota $<$ 50%	0,804	0,444	0,800	0,471
FWPF	Nota $\geq$ 50%	0,347	0,608	0,000	0,500
FWPF	Nota $<$ 50%	0,381	0,591	0,000	0,667
FWPL	Nota $\geq$ 50%	0,923	0,350	1,000	0,424
FWPL	Nota $<$ 50%	0,891	0,430	1,000	0,474

5.3.1 Métrica WPL

A seguir, os resultados das análises para a métrica referente a quantidade total de warnings por linhas de código.

Submissões Parciais vs. Submissões Finais

Os *boxplots* da Figura 5.13 apresentam em mais detalhes os resultados da análise da métrica nos códigos. É possível verificar que tanto a mediana quanto a dispersão da submissão final possuem valores maiores que os das submissões parciais.

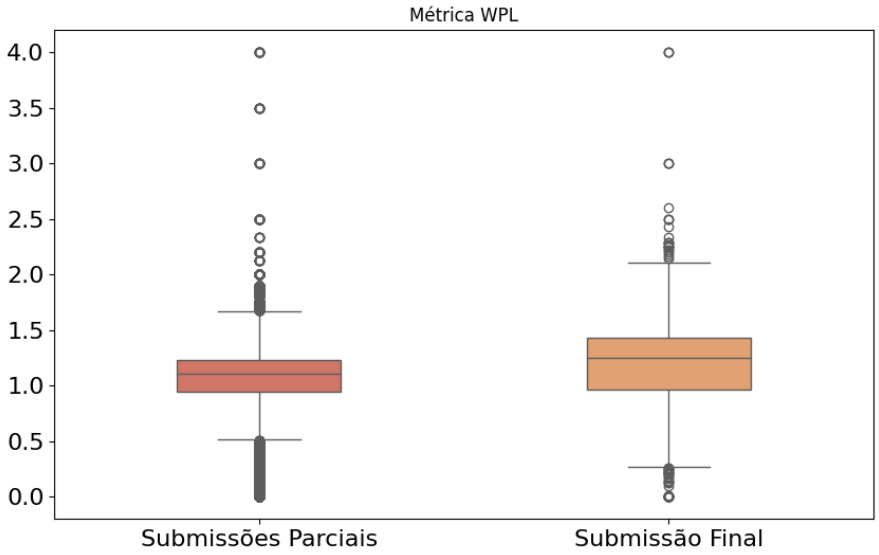

Figura 5.13: Boxplots para a métrica WPL as submissões parciais e final em Python

Estes valores sugerem que os códigos analisados em ambos os casos possuem uma quantidade alta de avisos gerados pelos analisadores estáticos, porém em quantidade maior na submissão final. O teste de Mann-Whitney apresenta que os valores das métricas variam significativamente entre os dois grupos ($p < 0{,}001$). O tamanho do seu efeito é considerado pequeno ($r_{rb} = 0{,}257$).

Códigos com notas acima de 50% vs. Códigos com notas abaixo de 50%

Os *boxplots* da Figura 5.14 apresentam com mais detalhes os resultados da análise.

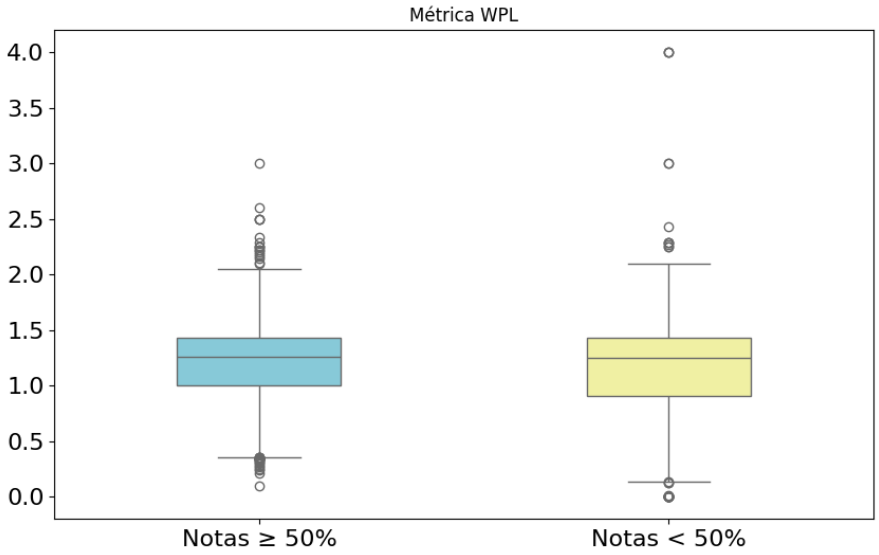

Figura 5.14: Boxplots para a métrica WPL para os códigos com notas superiores e inferiores ao limiar de 50% em Python.

É possível verificar que ambos os casos apresentam medianas similares, porém a dispersão no grupo de códigos com notas $< 50\%$ é maior, indicando que estes apresentam mais mensagens de aviso.

Por meio do teste de Mann-Whitney, encontramos que não existem diferenças significativas entre os grupos de códigos com notas maiores ou iguais a 50% e aqueles com notas inferiores a 50% ($p = 0{,}169$).

5.3.2 Métrica VWPV

A seguir, os resultados das análises para a métrica referente a quantidade total de mensagens de aviso de variáveis por quantidade de variáveis.

Submissões Parciais vs. Submissões Finais

Os *boxplots* da Figura 5.15 apresentam em mais detalhes os resultados da análise da métrica nos códigos. É possível verificar que existe uma dispersão e mediana maior no segundo grupo.

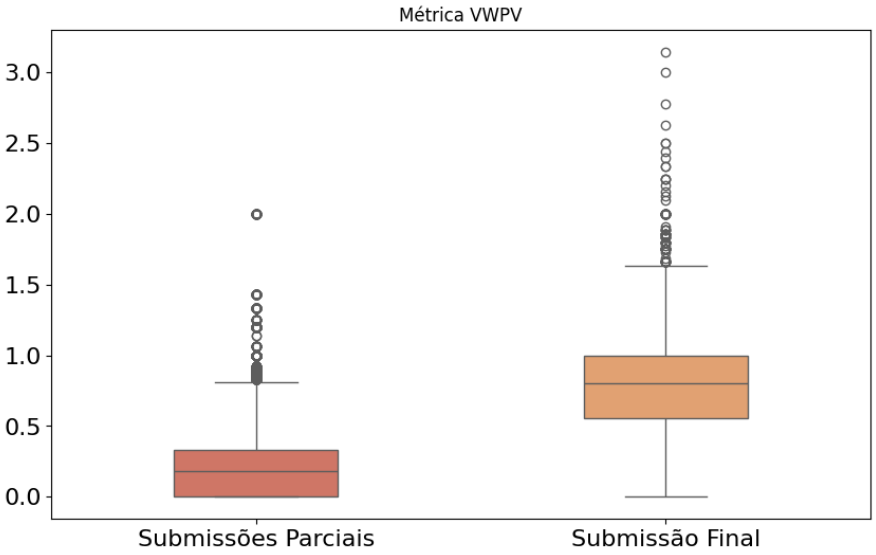

Figura 5.15: Boxplots para a métrica VWPV nas submissões parciais e final em Python

Os resultados indicam que os códigos observados na submissão final apresentam mais mensagens de aviso relacionadas à nomeação de variáveis em comparação às submissões parciais. O teste de Mann-Whitney apresenta que os valores das métricas variam significativamente entre os dois grupos ($p < 0{,}001$). O tamanho do seu efeito é considerado forte ($r_{\text{rb}} = 0{,}754$).

Códigos com notas acima de 50% vs. Códigos com notas abaixo de 50%

Os *boxplots* da Figura 5.16 apresentam com mais detalhes os resultados da análise.

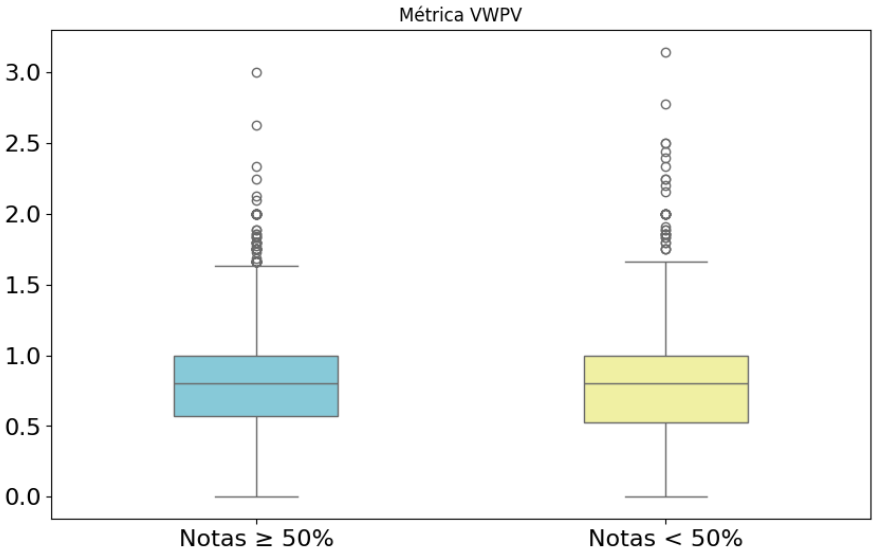

Figura 5.16: Boxplots para a métrica VWPV para os códigos com notas superiores e inferiores ao limiar de 50% em Python.

É possível verificar que ambos os casos apresentam distribuições com medianas e dispersões similares. Porém, o grupo de códigos com notas abaixo de 50% apresenta mais problemas em relação à nomeação de variáveis.

Por meio do teste de Mann-Whitney, encontramos que não existem diferenças significativas entre os grupos de códigos com notas maiores ou iguais a 50% e aqueles com notas inferiores a 50% ($p = 0{,}474$).

5.3.3 Métrica FWPF

A seguir, os resultados das análises para a métrica referente a quantidade total de mensagens de aviso de funções por quantidade de funções.

Submissões Parciais vs. Submissões Finais

Os *boxplots* da Figura 5.17 apresentam em mais detalhes os resultados da análise da métrica nos códigos. Verifica-se uma maior dispersão na submissão final.

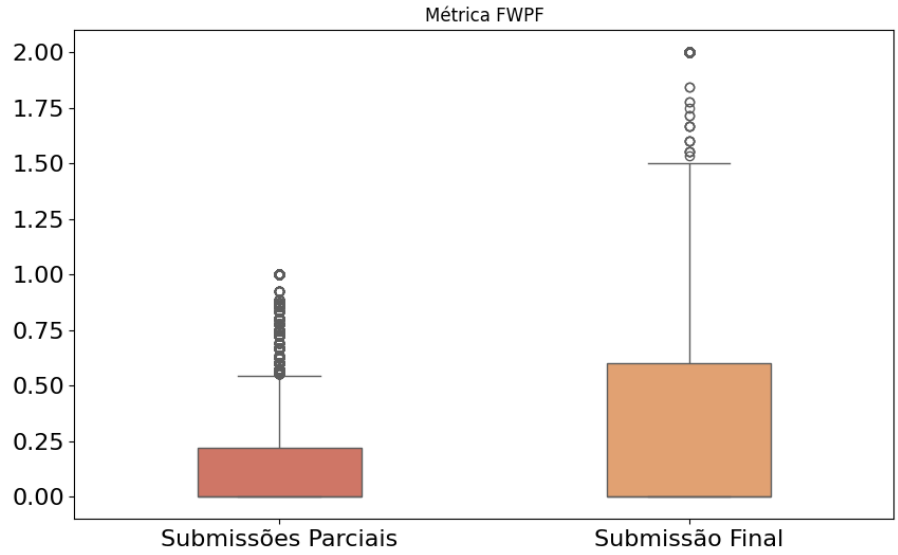

Figura 5.17: Boxplots para a métrica FWPF nas submissões parciais e final em Python.

É possível verificar que na submissão final existe uma maior quantidade de mensagens de aviso relacionadas a funções. O teste de Mann-Whitney apresenta que os valores das métricas variam significativamente entre os dois grupos ($p < 0{,}001$). Porém, o tamanho de seu efeito é considerado muito pequeno ($r_{rb} = 0{,}095$).

Códigos com notas acima de 50% vs. Códigos com notas abaixo de 50%

Os *boxplots* da Figura 5.18 apresentam com mais detalhes os resultados da análise. Já Grupo 02 se refere ao grupo dos códigos com notas abaixo de 50%.

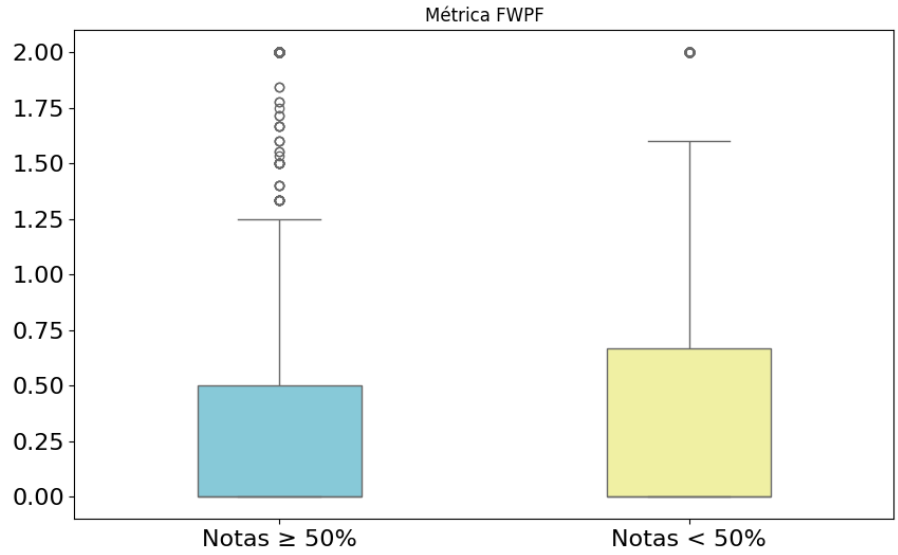

Figura 5.18: Boxplots para a métrica FWPF para os códigos com notas superiores e inferiores ao limiar de 50% em Python.

É possível verificar que o grupo de códigos com notas inferiores a 50% apresenta uma maior dispersão em comparação com o grupo de códigos com notas acima de 50%. Indicando que os códigos com menores notas apresentam mais problemas relacionados à nomeação de funções.

Por meio do teste de Mann-Whitney, encontramos que existem diferenças significativas entre os grupos de códigos com notas maiores ou iguais a 50% e aqueles com notas inferiores a 50% ($p < 0{,}001$). Porém, o tamanho de seu efeito é considerado muito pequeno ($r_{\text{rb}} = 0{,}061$)

5.3.4 Métrica FWPL

A seguir, os resultados das análises para a métrica referente à quantidade de mensagens de aviso de formatação por linhas de código.

Submissões Parciais vs. Submissões Finais

Os *boxplots* da Figura 5.19 apresentam em mais detalhes os resultados da análise da métrica nos códigos. Verifica-se que, apesar da dispersão dos dados e do valor da mediana ser maior no segundo grupo, os valores do primeiro se aproximam.

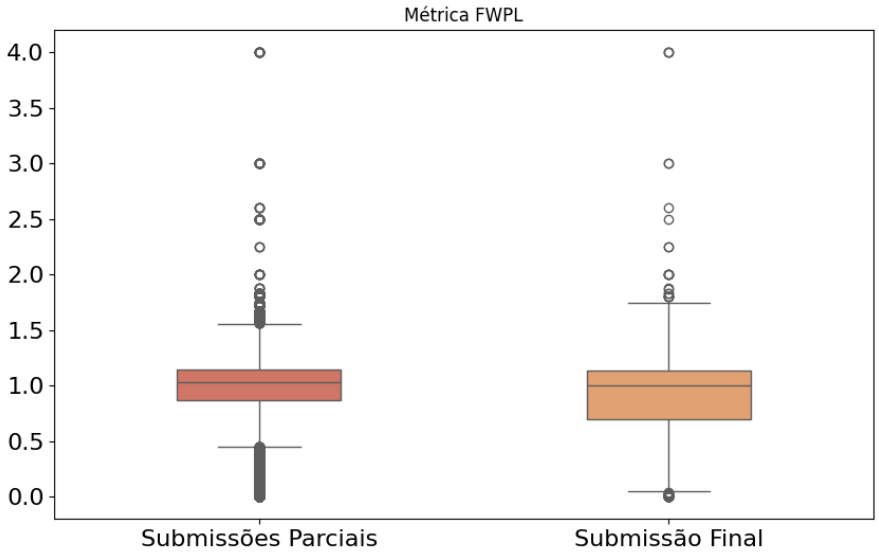

Figura 5.19: Boxplots para a métrica FWPL nas submissões parciais e final em Python.

Os dados indicam que existe uma alta incidência de avisos relacionados à formatação de código observados em ambos os casos, sendo maior no segundo caso. O teste de Mann-Whitney apresenta que os valores das métricas variam significativamente entre os dois grupos ($p < 0{,}001$). Porém, o tamanho de seu efeito é considerado muito pequeno ($r_{rb} = 0{,}092$).

Códigos com notas acima de 50% vs. Códigos com notas abaixo de 50%

Os *boxplots* da Figura 5.20 apresentam com mais detalhes os resultados da análise.

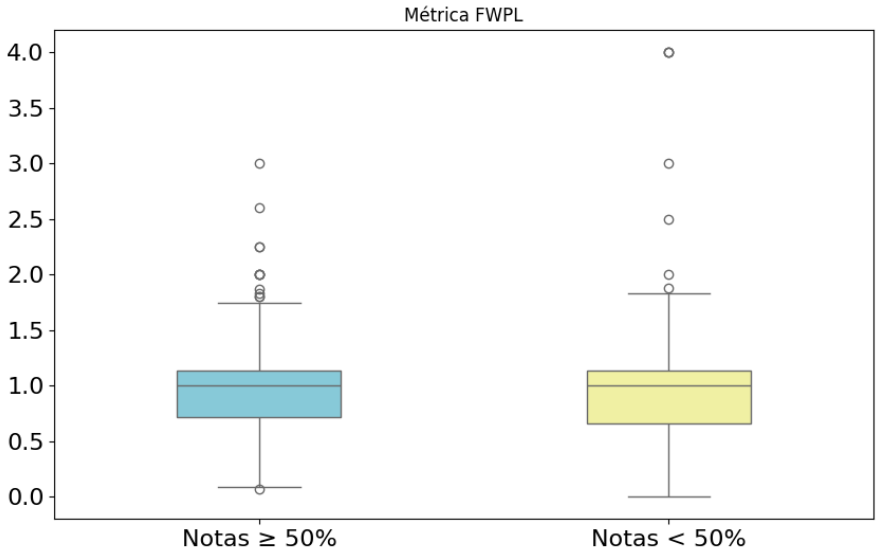

Figura 5.20: Boxplots para a métrica FWPL para os códigos com notas superiores e inferiores ao limiar de 50% em Python.

É possível verificar que ambos os casos apresentam distribuições com medianas e dispersões similares.

Por meio do teste de Mann-Whitney, encontramos que não existem diferenças significativas entre os grupos de códigos com notas maiores ou iguais a 50% e aqueles com notas inferiores a 50% ($p = 0{,}075$).

5.4 Comparação entre códigos em C e em Python

A seguir, são apresentados os resultados da comparação entre os resultados apresentados em C e os resultados apresentados em Python. Para isso, uma análise descritiva de cada métrica é realizada, seguida do teste de hipótese para verificar se os dados variam significativamente entre os dois grupos.

5.4.1 Métrica WPL

Submissões Parciais

A partir dos valores anteriormente, verifica-se que os valores da mediana dos códigos em C são maiores que em Python: 1,176 e 1,103, respectivamente. Apesar dos valores serem próximos, é uma indicação de que os códigos em C apresentam mais mensagens de aviso.

O teste de Mann-Whitney verificou que existem diferenças significativas entre os dois grupos para a métrica analisada ($p < 0{,}001$), com um efeito pequeno ($r_{rb} = 0{,}143$).

Submissões Finais

A partir dos valores apresentados anteriormente, é possível verificar que os valores da mediana dos códigos em Python são maiores que em C: 1,254 e 1,118, respectivamente. Este valor sugere que existem mais mensagens de aviso em Python do que em C.

O teste de Mann-Whitney verificou que existem diferenças significativas entre os dois grupos para a métrica analisada ($p < 0{,}001$), com um efeito pequeno ($r_{rb} = 0{,}146$).

5.4.2 Métrica VWPV

Submissões Parciais

Verifica-se que a mediana dos códigos em C é maior que a dos códigos em Python: 0,200 e 0,182, respectivamente. Apesar dos valores serem próximos, é uma indicação de que os códigos em C apresentam mais mensagens de aviso relacionadas a variáveis.

O teste de Mann-Whitney verificou que existem diferenças significativas entre os dois grupos para a métrica analisada ($p < 0{,}001$), com um efeito muito pequeno ($r_{rb} = 0{,}006$).

Submissões Finais

A partir dos valores apresentados anteriormente, é possível verificar que os valores da mediana dos códigos em Python são maiores que em C: 0,800 e 0,143, respectivamente. Porém, existe uma dispersão dos dados semelhante entre os dois grupos. Este resultado indica que existem mais mensagens de aviso relacionadas à nomeação de variáveis em Python do que em C.

O teste de Mann-Whitney verificou que existem diferenças significativas entre os dois grupos para a métrica analisada ($p < 0{,}001$), com um efeito forte ($r_{rb} = 0{,}749$)

5.4.3 Métrica FWPF

Submissões Parciais

A partir dos valores apresentados anteriormente, é possível verificar que a mediana de ambos os códigos é igual a 0, porém a dispersão dos valores é maior nos códigos em C.

O teste de Mann-Whitney verificou que existem diferenças significativas entre os dois grupos para a métrica analisada ($p < 0{,}001$), com um efeito pequeno ($r_{rb} = 0{,}041$).

Submissões Finais

A partir dos valores apresentados anteriormente, é possível verificar que os valores da mediana dos códigos em Python e em C são iguais a zero. No entanto, há uma dispersão maior dos dados em Python.

O teste de Mann-Whitney verificou que existem diferenças significativas entre os dois grupos para a métrica analisada ($p < 0{,}001$), com um efeito pequeno ($r_{rb} = 0{,}120$).

5.4.4 Métrica FWPL

Submissões Parciais

A partir dos valores apresentados anteriormente, é possível verificar que o valor da mediana dos códigos em C é maior que em Python: 1,125 e 1,033, respectivamente. Complementando, a dispersão dos dados em C também é maior. Este resultado indica que existem mais problemas relacionados à formatação em códigos em C do que em Python.

O teste de Mann-Whitney verificou que existem diferenças significativas entre os dois grupos para a métrica analisada ($p < 0{,}001$), com um efeito pequeno ($r_{rb} = 0{,}207$).

Submissões Finais

A partir dos valores apresentados anteriormente, é possível verificar que os valores da mediana dos códigos em C são maiores que em Python: 1,073 e 1,000, respectivamente. Ao mesmo tempo em que existe uma dispersão maior dos dados em C. Este resultado indica que existem mais problemas relacionados à formatação em códigos em C do que em Python.

O teste de Mann-Whitney verificou que existem diferenças significativas entre os dois grupos para a métrica analisada ($p < 0{,}001$), com um efeito pequeno ($r_{rb} = 0{,}176$).

5.5 Correlações

A seguir serão apresentados os resultados das correlações de Spearman realizadas. É importante notar que o símbolo * indica que a correlação é significativa no nível 0,05 nas duas extremidades, enquanto que ** indica que a correlação é significativa no nível 0,01 nas duas extremidades. Para as correlações, somente as submissões finais foram consideradas.

5.5.1 Correlações entre métricas e notas

A Tabela 5.10 apresenta as correlações entre as métricas com as notas dos códigos analisados em C e Python. A partir dos seus resultados, é possível verificar que não existem correlações entre as métricas avaliadas e as notas dos códigos observados em ambos os casos.

Tabela 5.10: Correlações entre as métricas com as notas dos códigos em C e Python.

Metrica	Nota (C)	Nota (Python)
WPL	0,052** (0,000)	0,023** (0,222)
VWPV	-0,035** (0,000)	0,011 (0,552)
FWPF	-0,075** (0,001)	-0,059 (0,002)
FWPL	0,055** (0,000)	0,032 (0,097)

5.5.2 Correlações entre as métricas

A seguir serão apresentadas as correlações entre as métricas para cada linguagem de programação analisada.

A Tabela 5.11 apresenta as correlações entre as métricas para a linguagem Python. Nota-se que existe uma correlação mediana entre as métricas **WPL** e **VWPV** e muito forte entre **WPL** e **FWPL**. Percebe-se também uma correlação negativa fraca entre **VWPV** e **FWPF**.

Tabela 5.11: Correlações Internas em Python

Metrica	WPL	VWPV	FWPF	FWPL
WPL	1,000 (-)	0,305**(0,000)	0,067**(0,000)	0,870**(0,000)
VWPV	0,305**(0,000)	1,000(-)	-0,051**(0,008)	0,004(0,836)
FWPF	0,067**(0,000)	-0,051**(0,008)	1,000 (-)	0,026(0,173)
FWPL	0,870**(0,000)	0,004(0,836)	0,026(0,173)	1,000 (-)

Já a Tabela 5.12 apresenta as correlações entre as métricas para a linguagem C. Nota-se que existe uma correlação fraca entre as métricas **WPL** e **VWPV**; negativa fraca entre **WPL** e **FWPF** e muito forte entre **FWPL** e **WPL**.

Tabela 5.12: Correlações Internas em C

Metrica	WPL	VWPV	FWPF	FWPL
WPL	1,000 (-)	0,290**(0,000)	-0,258**(0,000)	0,994**(0,000)
VWPV	0,290**(0,000)	1,000(-)	0,116**(0,000)	0,220(0,000)
FWPF	-0,258**(0,000)	0,116**(0,000)	1,000 (-)	-0,293**(0,000)
FWPL	0,994**(0,000)	0,220**(0,000)	-0,293**(0,000)	1,000 (-)

Capítulo 6

Discussão

Este capítulo realiza uma discussão sobre os resultados apresentados neste trabalho. A discussão se baseia nas questões de pesquisa apresentadas no Capítulo 1.

6.1 QP1 - Quais as mensagens de aviso mais comuns apresentadas pelos analisadores estáticos a partir dos códigos dos estudantes?

Para responder à QP1, foi necessário utilizar os analisadores estáticos nos códigos dos estudantes. Para os códigos em Python, foi utilizado o analisador **Pylint**. Já para C, foi utilizado o **Cpplint**. Assim como informado anteriormente, estes analisadores não possibilitam um feedback adequado a respeito da nomeação de funções e variáveis. Sendo assim foi necessário criar um novo analisador estático, o **NamingCheck**, para complementar a saída dos analisadores verificados. A partir dos resultados apresentados nas Tabelas 5.2, 5.3, 5.4 e 5.5, pôde-se perceber que a maioria das mensagens de aviso geradas pelos analisadores estáticos se relacionam com a formatação do código.

Percebe-se que o espaçamento é um tópico bastante importante nas duas linguagens. É importante ressaltar que apesar de a formatação do código não interferir na sua saída final, uma organização pode ajudar na manutenção do código, permitindo que outras pessoas possam entender o seu conteúdo sem maiores dificuldades.

Com os resultados encontrados, nota-se que, na maioria dos casos, não existe uma preocupação forte dos estudantes a respeito da formatação do código para seguir os padrões existentes das linguagens escolhidas. Essa não-preocupação pode ter uma possível explicação com o que é aprendido em sala de aula. É possível que os professores, ao apresentarem o material, não reforcem também as melhores formas de estilizar o código, principalmente em disciplinas não-introdutórias, onde frequentemente espera-se que os estudantes já possuam esse conhecimento. Focando assim na lógica de programação e no uso das estruturas corretas.

A maioria dos problemas relacionados com a formatação do código podem ser solucionadas com a utilização de um formatador de código, cujo objetivo é fazer com que a sua formatação siga os princípios nas convenções de código existentes. Este formatador pode ser utilizado em tempo real durante o desenvolvimento. Além disso, uma possível melhoria seria incorporar um formatador de código ao ambiente de desenvolvimento disponível no *autograder*.

6.2 QP2 - Quais métricas podem ser desenvolvidas para mensurar problemas de estilo no código dos estudantes?

Para poder avaliar a qualidade do código dos estudantes quanto ao estilo e responder à QP2, foi necessário entender melhor quais atributos do código podem ajudar a descrever o seu estilo. Isto pode ser parcialmente respondido pelos manuais de estilo e convenções de código de cada linguagem. A partir desta análise, e da criação um analisador estático novo, o **NamingCheck**, foi possível criar uma ferramenta integradora, o **PerfeQ**, que possibilita realizar uma analise do código mais completa, fornecendo assim ao utilizador um feedback mais completo quanto a qualidade do código quanto ao estilo.

A ferramenta **PerfeQ** possibilita a integração de três analisadores estáticos: **Pylint**, **Cpplint** e **NamingCheck** que irão apresentar um feedback quanto a organização do código (espaçamento, indentação) e também quanto a nomeação de variáveis, funções e structs. A ferramenta tem suporte as linguagens C e Python. Para este feedback, foram criadas métricas levando em consideração as convenções e padrões de estilo de cada linguagem para que juntas elas consigam descrever o código analisado quanto à sua qualidade de estilo. A utilização das métricas e seus resultados podem ajudar estudantes e professores a identificarem possíveis problemas de estilo no código-fonte. As quatro métricas desenvolvidas foram: (i) **WPL** (Quantidade de mensagens de aviso (geral) / LOC); (ii) **VWPV** (Quantidade de mensagens de aviso referentes a declaração de variáveis / Quantidade de variáveis); (iii) **FWPF** (Quantidade de mensagens de aviso referentes a declaração de funções / Quantidade de funções) e (iv) **FWPL** (Quantidade de mensagens de aviso referentes a formatação de código / LOC). É importante ressaltar que as definições das métricas já incorporam uma normalização dos dados. Assim, todas elas utilizam alguma forma de função de densidade.

A partir da criação das métricas, foi possível verificar a correlação existente entre elas. No Capítulo 5, foi verificado que nos códigos analisados, na linguagem C, foi verificada uma correlação fraca entre **WPL** e **VWPV**, indicando que, em geral, existem mensagens de aviso relacionadas à declaração de variáveis, porém a sua quantidade é baixa. Uma possível explicação é apresentada na Tabela 5.4, na qual não são apresentadas mensagens de aviso relacionadas à declaração de variáveis. Além disso,

foi encontrada uma correlação muito forte entre **FWPL** e **WPL**, indicando que a maioria das mensagens de aviso se relacionam com nomeação de funções.

Já em Python, existe uma correlação mediana entre as métricas **WPL** e **VWPV**. Uma possível explicação da correlação se dá pela presença de mensagens de aviso relacionadas à nomeação de variáveis. Apesar de não aparecerem na Tabela 5.5, é possível verificar na Figura 5.15 que as submissões finais apresentam um alto índice de avisos relacionados à declaração de variáveis. Também foi encontrada uma correlação muito forte entre **WPL** e **FWPL**, como apresentado na Tabela 5.5. Verifica-se que a maior quantidade de mensagens de aviso se relaciona com a formatação de código, explicando a correlação muito forte.

Assim como também apresentado no Capítulo 5, a partir dos valores das métricas, foi possível ter uma descrição quantitativa do código analisado, permitindo realizar análises estatísticas – gerando *boxplots* e verificando as correlações, para uma melhor compreensão sobre a qualidade de estilo de código dos estudantes. A análise das correlações, juntamente com os dados descritivos das métricas e a representação visual através dos *boxplots*, ajuda a mensurar problemas de estilo no código.

6.3 QP3 - Os estudantes seguem os padrões de estilo das linguagens de programação utilizadas?

A partir dos resultados obtidos no Capítulo 5, pôde-se perceber que a maioria dos estudantes analisados não segue os padrões de estilo das linguagens de programação utilizadas. Os altos valores dos coeficientes das métricas juntamente com a análise estatística realizada apontam que a maior parte dos problemas de estilo tem relação com a formatação do código.

Ao realizar uma comparação entre as métricas utilizadas para avaliação dos padrões de estilo, verifica-se que em ambas as linguagens (C e Python) os estudantes estão, na maior parte das vezes, nomeando corretamente as variáveis. Em Python, pôde-se perceber que a submissão final apresenta mais problemas relacionados com a nomeação de variáveis em comparação com as submissões parciais - apresentando diferenças significativas entre os grupos e com um tamanho de efeito forte.

A respeito da nomeação das funções, em C os estudantes também as nomeiam corretamente na maioria das vezes. Porém, em Python, as métricas são mais altas, sugerindo problemas também na nomeação de funções. Ao realizar essa comparação entre as linguagens C e Python, as análises indicaram que os códigos em C apresentam mais problemas de estilo de código, o teste de hipótese indicou que existem diferenças significativas nos resultados, porém na maioria das métricas o efeito é considerado pequeno, exceto com a métrica VWPV – no qual apresentou um efeito forte.

Ao fazer uma comparação entre o histórico de submissões e a submissão final do estudante, nota-se que a diferença dos resultados das métricas é bastante baixa; em

alguns casos, a submissão final apresenta mais problemas de estilo de código do que as parciais. O que também nos leva a inferir que os estudantes realizaram poucas mudanças quanto ao estilo de código durante a sua evolução.

Realizamos também uma comparação entre os códigos que obtiveram notas maiores ou iguais a 50% e os que obtiveram notas inferiores a 50%. Foi verificado que, de modo geral, não existem muitas diferenças entre os dois grupos – indicando que os códigos com notas altas não garantem bons resultados das métricas de qualidade de estilo de código.

Assim como discutido na questão de pesquisa anterior, o não seguimento dos padrões de estilo pode ter relação com o que é aprendido em sala de aula. Como o estudo é realizado levando em consideração que os códigos avaliados são provenientes de um ambiente de juiz online, é necessário entender que avaliar somente a corretude dos códigos não é o suficiente para o aprendizado. Aprender a respeito dos padrões e convenções de código é importante não somente para o meio acadêmico, mas para o ambiente profissional – onde seguir padrões de estilo de código é importante para a manutenibilidade de software.

Uma ação sugerida é realizar integrações de sistemas *autograders* juntamente com ferramentas de análise estática para que, durante o aprendizado, o estudante possa não somente se preocupar com a corretude dos códigos mas também com o estilo – verificando se este segue os padrões desejáveis.

6.4 QP4 - Existe correlação entre a qualidade do estilo de código e o desempenho do estudante nos juízes online?

A partir dos resultados da Tabela 5.10, pôde-se verificar as correlações entre as métricas utilizadas para medir a qualidade do estilo do código e o desempenho dos estudantes nos juízes online. Percebemos que não existe uma correlação entre o estilo de código e a nota, indicando que outros fatores externos podem estar contribuindo com os altos índices das métricas. Porém, o que pode-se perceber é que o sistema *autograder*, em que os estudantes submetem os códigos, somente realiza a verificação da corretude da saída dos códigos. A partir do momento em que não está sendo avaliada a qualidade de código no sistema, é possível que os estudantes comecem a ignorar alguns padrões de estilo, mantendo o seu foco na corretude, de modo a fazer com que o código gere a resposta necessária para passar nos testes.

6.5 Ameaças à Validade

Apresentamos algumas relevantes ameaças à validade relacionadas a este trabalho.

6.5.1 Validade de Construto

Consideramos uma ameaça de constructo as ferramentas criadas nesta pesquisa: o Analisador Estático **NamingCheck** e a ferramenta integradora de AEs **PerfeQ**. Ao se tratar de criações novas e apesar de passarem por testes antes da avaliação dos códigos, entendemos que estas ferramentas estão em constante evolução, portanto podem apresentar falhas ao avaliar códigos – apresentando até falsos positivos ou até mesmo a não apresentação de mensagens de aviso – prejudicando a construção das métricas.

6.5.2 Validade Interna

Consideramos uma ameaça interna a validade da pesquisa, o fato da realização da comparação dos resultados das linguagens C e Python. Entendemos que os grupos em si são diferentes, possuem sintaxes e algumas convenções de estilo diferentes, porém entendemos que existem algumas similaridades a respeito das convenções de nomeações de variáveis e funções – um dos alvos deste estudo.

6.5.3 Validade Externa

Consideramos uma ameaça externa a validade da pesquisa o fato de utilizar dados de estudantes de somente uma única universidade – Universidade Federal do Amazonas (UFAM). Sendo assim, os resultados obtidos nesta pesquisa não devem ser generalizados para outras instituições e cenários.

6.5.4 Validade de Conclusão

Para evitar possíveis ameaças à validade relacionadas à coleta de dados, estes foram coletados a partir dos *logs* do próprio sistema de juiz online (**CodeBench**). Foram realizados testes estatísticos para rejeitar as hipóteses nulas e, com isso, a validade dos resultados obtidos são aceitáveis. Como a desistência de uma disciplina pode ocorrer por diversos fatores não controláveis, optou-se por excluir da análise os alunos que trancaram ou reprovaram por falta nas disciplinas. Dessa forma, tentamos amenizar o viés causado por esses fatores, conseguindo mensurar com mais exatidão o desempenho real do aluno na disciplina.

Capítulo 7

Conclusões

Este trabalho buscou analisar códigos-fonte de estudantes provenientes do juiz online *CodeBench* através de ferramentas de análise estática com o objetivo final de desenvolver um feedback mais completo quanto destes códigos-fonte – apresentando não somente informações sobre corretude, mas destacando os problemas de estilo.

Para a análise, foi necessário inicialmente criar um Analisador Estático, **NamingCheck**, capaz de analisar códigos em C e Python e apresentar um feedback quanto a declaração de variáveis e funções – seguindo as convenções de estilo das linguagens. Em seguida, foi criado uma ferramenta integradora de AEs, **PerfeQ**, que tem suporte às linguagens C e Python, integrando os AEs **Cpplint**, **Pylint** e **NamingCheck**. Após a integração, foram criadas e apresentadas métricas que descrevem o código-fonte através de índices de qualidade relacionados a estilo de código. Estas métricas foram baseadas em atributos do código-fonte a partir do processamento da saída dos analisadores estáticos utilizados.

A partir das ferramentas mencionadas, foi possível realizar o estudo retratado neste trabalho. Inicialmente, os dados do juiz online **CodeBench**, contendo todas as submissões dos estudantes, foram coletados, que foram submetidas à etapa de Análise Estática, passando pela ferramenta **PerfeQ** e tendo como resultado os valores das métricas calculadas. Em seguida, foi realizado um estudo estatístico buscando verificar diferenças em métricas de estilo entre i) submissões parciais versus finais; ii) grupos de estudantes com notas acima versus abaixo do limiar de 50%; iii) códigos em C versus códigos em Python, além da correlação entre a qualidade dos códigos com o desempenho dos estudantes e a correlação interna entre as métricas.

Os resultados apontam que os estudantes de modo geral não seguem as convenções de estilo de código. Em sua grande maioria, buscam somente alcançar a corretude do código para passar nos exercícios propostos, ignorando os padrões e convenções existentes. Isto é tanto uma consequência da ferramenta *autograder* utilizada somente avaliar a corretude do código através de testes como de uma possível menor importância dada em disciplinas de graduação a respeito de convenções de código.

Assim, entendemos ser necessário que os estudantes comecem a se preocupar não somente com a corretude do código, mas também com a qualidade de estilo – fator que é importante não somente no meio acadêmico, mas também no mercado de trabalho, onde seguir convenções e padrões de código é extremamente importante para a manutenibilidade do software.

7.1 Trabalhos Futuros

As ferramentas criadas neste trabalho estão em constante evolução. Trabalhos futuros incluem melhorias nas ferramentas desenvolvidas, fazendo com que estas apresentem resultados mais precisos na sua utilização. Como apresentado nas análises, alguns *boxplots* apresentavam alguns *outliers* com altos valores de índices. Espera-se que eventuais melhorias possam reduzir a frequência destes *outliers*.

Além disso, outro trabalho futuro pode ser a criação de uma interface gráfica para a ferramenta integradora de AEs PerfeQ. Assim, além de fornecer as métricas e a saída em um arquivo .csv, será possível apresentar ao usuário dados mais detalhados em um *dashboard*, não demandando uma ferramenta externa para a realização da análise de dados. Por fim, além de fornecer os valores das métricas, existe a possibilidade da integração de *Large Language Models* (LLMs) por meio de Inteligência Artificial permite disponibilizar ao usuário sugestões para correção e otimização do código.

Futuras pesquisas envolvem estender o trabalho para outras universidades e disciplinas. Além disso, uma integração da ferramenta **PerfeQ** com sistemas de juiz online como o **CodeBench** pode ser realizada, a fim de apresentar aos estudantes um feedback mais completo a respeito do seu código – avaliando assim a corretude juntamente com a qualidade.

A partir da integração da ferramenta **PerfeQ** com sistemas de juízes online, é possível realizar experimentos controlados comparando grupos que são avaliados ou não pelas métricas de qualidade. Assim, permite-se identificar se o feedback sobre a qualidade do código tem um impacto nos códigos dos estudantes.

Referências

Abidi, M., Grichi, M., Khomh, F., e Guéhéneuc, Y.-G. (2019). Code smells for multi-language systems. In *Proceedings of the 24th European Conference on Pattern Languages of Programs*, EuroPLop '19, New York, NY, USA. Association for Computing Machinery.

Allamanis, M., Barr, E. T., Bird, C., e Sutton, C. (2014). Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, página 281–293, New York, NY, USA. Association for Computing Machinery.

AlOmar, E. A., AlOmar, S. A., e Mkaouer, M. W. (2023). On the use of static analysis to engage students with software quality improvement: An experience with pmd. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, páginas 179–191.

Aman, H., Amasaki, S., Yokogawa, T., e Kawahara, M. (2020). A survival analysis-based prioritization of code checker warning: A case study using pmd. *Big Data, Cloud Computing, and Data Science Engineering*, páginas 69–83.

Berry, R. E. e Meekings, B. A. (1985). A style analysis of c programs. *Commun. ACM*, 28(1):80–88.

Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., e Engler, D. (2010). A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75.

Birillo, A., Vlasov, I., Burylov, A., Selishchev, V., Goncharov, A., Tikhomirova, E., Vyahhi, N., e 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-Volume 1*, páginas 307–313.

Börstler, J., Störrle, H., Toll, D., van Assema, J., Duran, R., Hooshangi, S., Jeuring, J., Keuning, H., Kleiner, C., e MacKellar, B. (2018). "i know it when i see it"perceptions of code quality: Iticse '17 working group report. In *Proceedings*

of the 2017 ITiCSE Conference on Working Group Reports, ITiCSE-WGR '17, página 70–85, New York, NY, USA. Association for Computing Machinery.

Cairo, A. S., Carneiro, G. d. F., e Monteiro, M. P. (2018). The impact of code smells on software bugs: A systematic literature review. *Information*, 9(11):273.

Cannon, L., Elliott, R., Kirchhoff, L., Miller, J., Milner, J., Mitze, R., Schan, E., Whittington, N., Spencer, H., Keppel, D., et al. (1991). *Recommended C style and coding standards*. Pocket reference guide. Specialized Systems Consultants.

Chhillar, R. S. e Gahlot, S. (2017). An evolution of software metrics: a review. In *Proceedings of the International Conference on Advances in Image Processing*, páginas 139–143.

Doland, J. e Valett, J. (1994). C style guide. Relatório técnico, National Aeronautics and Space Administration.

Dos Santos, G. T. H. e Martimiano, L. A. F. (2023). Uso de ferramentas de análise estática para identificar vulnerabilidades em sistemas operacionais em c/c++ para dispositivos iot. *Revista Eletrônica de Iniciação Científica em Computação*, 21(1).

dos Santos, R. M. e Gerosa, M. A. (2018). Impacts of coding practices on readability. In *Proceedings of the 26th conference on program comprehension*, páginas 277–285.

Ebert, C., Cain, J., Antoniol, G., Counsell, S., e Laplante, P. (2016). Cyclomatic complexity. *IEEE software*, 33(6):27–29.

Edwards, S. H., Kandru, N., e Rajagopal, M. B. (2017). Investigating static analysis errors in student java programs. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, página 65–73, New York, NY, USA. ACM.

Feghali, I., Watson, A. H., Henderson-Sellers, B., e Tegarden, D. (1994). Technical correcspondence: Clarrification concerning modularization and mccabe's cyclomatic complexity. *Communications of the ACM*, 37(4):91–94.

Filazzola, A. e Lortie, C. (2022). A call for clean code to effectively communicate science. *Methods in Ecology and Evolution*, 13(10):2119–2128.

Francisco, R. E., Ambrósio, A. P. L., Junior, C. X. P., e Fernandes, M. A. (2018). Juiz online no ensino de cs1 - lições aprendidas e proposta de uma ferramenta. *Revista Brasileira de Informática na Educação*, 26(03):163.

Gaffney Jr, J. E. (1981). Metrics in software quality assurance. In *Proceedings of the ACM'81 conference*, páginas 126–130.

Galvão, L., Fernandes, D., e Gadelha, B. (2016). Juiz online como ferramenta de apoio a uma metodologia de ensino híbrido em programação. In *(Simpósio Brasileiro de Informática na Educação - SBIE)*, volume 27, página 140.

Glassman, E. L., Fischer, L., Scott, J., e Miller, R. C. (2015). Foobaz: Variable name feedback for student code at scale. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, páginas 609–617.

Gomes, A. e Mendes, A. (2007). Learning to program - difficulties and solutions. páginas 283–287.

Gomes, I., Morgado, P., Gomes, T., e Moreira, R. (2009). An overview on the static code analysis approach in software development. *Faculdade de Engenharia da Universidade do Porto, Portugal*.

Gulabovska, H. e Porkoláb, Z. (2019). Survey on static analysis tools of python programs. In *SQAMIA*.

Hedberg, H., Iivari, N., Rajanen, M., e Harjumaa, L. (2007). Assuring quality and usability in open soruce software development. In *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*, FLOSS '07, página 2, USA. IEEE Computer Society.

Hidalgo-Céspedes, J., Marín-Raventós, G., e Calderón-Campos, M. E. (2020). Online judge support for programming teaching. In *2020 XLVI Latin American Computing Conference (CLEI)*, páginas 522–530. IEEE.

Izu, C., Mirolo, C., Börstler, J., Connamacher, H., Crosby, R., Glassey, R., Haldeman, G., Kiljunen, O., Kumar, A. N., Liu, D., et al. (2025). Introducing code quality at cs1 level: Examples and activities. In *2024 Working Group Reports on Innovation and Technology in Computer Science Education*, páginas 339–377.

Johnson, B., Song, Y., Murphy-Hill, E., e Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, páginas 672–681.

Joshi, A., Tewari, A., Kumar, V., e Bordoloi, D. (2014). Integrating static analysis tools for improving operating system security. *International Journal of Computer Science and Mobile Computing*, 3(4):1251–1258.

Kan, S. H. (2003). *Metrics and models in software quality engineering*. Addison-Wesley Professional.

Keuning, H., Heeren, B., e 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, página 110–115, New York, NY, USA. Association for Computing Machinery.

Keuning, H., Jeuring, J., e Heeren, B. (2023). A systematic mapping study of code quality in education. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, páginas 5–11.

Kim, S. e Ernst, M. D. (2007). Which warnings should i fix first? In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, página 45–54, New York, NY, USA. Association for Computing Machinery.

Kirk, D., Crow, T., Luxton-Reilly, A., e Tempero, E. (2022). Teaching code quality in high school programming courses-understanding teachers' needs. In *Proceedings of the 24th Australasian Computing Education Conference*, páginas 36–45.

Kirk, D., Luxton-Reilly, A., e Tempero, E. (2024). Code style!= code quality. In *Proceedings of the 2024 on ACM Virtual Global Computing Education Conference V. 1*, páginas 267–270.

Lacerda, G., Petrillo, F., Pimenta, M., e Guéhéneuc, Y. G. (2020). Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, 167:110610.

Latte, B., Henning, S., e Wojcieszak, M. (2019). Clean code: On the use of practices and tools to produce maintainable code for long-living.

Lenarduzzi, V., Lomio, F., Huttunen, H., e Taibi, D. (2020a). Are sonarqube rules inducing bugs? In *2020 IEEE 27th international conference on software analysis, evolution and reengineering (SANER)*, páginas 501–511. IEEE.

Lenarduzzi, V., Sillitti, A., e Taibi, D. (2020b). A survey on code analysis tools for software maintenance prediction. In *Proceedings of 6th International Conference in Software Engineering for Defence Applications: SEDA 2018 6*, páginas 165–175. Springer.

Liawatimena, S., Warnars, H. L. H. S., Trisetyarso, A., Abdurahman, E., Soewito, B., Wibowo, A., Gaol, F. L., e Abbas, B. S. (2018). Django web framework software metrics measurement using radon and pylint. In *2018 Indonesian Association for Pattern Recognition International Conference (INAPR)*, páginas 218–222. IEEE.

Lima, Y., Fonseca, I., Chagas, J., Rodrigues, E., Silveira, M., e Silva, J. (2021). Comparação de ferramentas de análise estática para detecção de defeitos de software usando mutantes. In *Anais da V Escola Regional de Engenharia de Software*, páginas 159–168, Porto Alegre, RS, Brasil. SBC.

Liu, D., Calver, J., e Craig, M. (2024). Are a static analysis tool study's findings static? a replication. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2024, página 80–86, New York, NY, USA. Association for Computing Machinery.

Liu, D. e Petersen, A. (2019). Static analyses in python programming courses. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, página 666–671, New York, NY, USA. Association for Computing Machinery.

Liu, X. e Woo, G. (2020). Applying code quality detection in online programming judge. In *Proceedings of the 2020 5th International Conference on Intelligent Information Technology*, ICIIT '20, página 56–60, New York, NY, USA. Association for Computing Machinery.

Ljung, K. (2021). Clean code in practice: Developers perception of clean code.

Maimon, O. e Rokach, L. (2005). Introduction to knowledge discovery in databases. In *Data mining and knowledge discovery handbook*, páginas 1–17. Springer.

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

Martin, R. C. e Coplien, J. O. (2009). *Clean code: a handbook of agile software craftsmanship*. Prentice Hall.

Mashiach, T., Sotto-Mayor, B., Kaminka, G., e Kalech, M. (2023). Clean++: Code smells extraction for c++. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, páginas 441–445. IEEE.

Mengel, S. A. e Yerramilli, V. (1999). A case study of the static analysis of the quality of novice student programs. In *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '99, página 78–82, New York, NY, USA. Association for Computing Machinery.

Messer, M., Brown, N. C., Kölling, M., e Shi, M. (2024). Automated grading and feedback tools for programming education: A systematic review. *ACM Transactions on Computing Education*, 24(1):1–43.

Miguel, J., Mauricio, D., e Rodriguez, G. (2014). A review of software quality models for the evaluation of software products. *International Journal of Software Engineering & Applications*, 5:31–54.

Muniz, R. C. et al. (2022). Uma técnica para detectar fraquezas de código em programas c.

Nirpal, P. B. e Kale, K. (2011). A brief overview of software testing metrics. *International Journal on Computer Science and Engineering*, 3(1):204–2011.

Novak, J., Krajnc, A., et al. (2010). Taxonomy of static code analysis tools. In *The 33rd international convention MIPRO*, páginas 418–422. IEEE.

Nuñez-Varela, A. S., Pérez-Gonzalez, H. G., Martínez-Perez, F. E., e Soubervielle-Montalvo, C. (2017). Source code metrics: A systematic mapping study. *Journal of Systems and Software*, 128:164–197.

Panichella, S., Arnaoudova, V., Di Penta, M., e Antoniol, G. (2015). Would static analysis tools help developers with code reviews? In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, páginas 161–170.

Pereira, F. D., Fonseca, S. C., Wiktor, S., Oliveira, D. B., Cristea, A. I., Benedict, A., Fallahian, M., Dorodchi, M., Carvalho, L. S., Mello, R. F., et al. (2023). Toward supporting cs1 instructors and learners with fine-grained topic detection in online judges. *IEEE Access*, 11:22513–22525.

Plösch, R. e Neumüller, C. (2020). Does static analysis help software engineering students? In *Proceedings of the 2020 9th International Conference on Educational and Information Technology*, ICEIT 2020, página 247–253, New York, NY, USA. ACM.

Prause, C. R. e Jarke, M. (2015). Gamification for enforcing coding conventions. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, página 649–660, New York, NY, USA. Association for Computing Machinery.

Reiss, S. P. (2007). Automatic code stylizing. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, página 74–83, New York, NY, USA. Association for Computing Machinery.

Saliba, L., Shioji, E., Oliveira, E., Cohney, S., e Qi, J. (2024). Learning with style: Improving student code-style through better automated feedback. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, páginas 1175–1181.

Shen, H., Fang, J., e Zhao, J. (2011). Efindbugs: Effective error ranking for findbugs. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, páginas 299–308.

Singh, D., Sekar, V. R., Stolee, K. T., e Johnson, B. (2017). Evaluating how static analysis tools can reduce code review effort. In *2017 IEEE symposium on visual languages and human-centric computing (VL/HCC)*, páginas 101–105. IEEE.

Stegeman, M., Barendsen, E., e Smetsers, S. (2016). Designing a rubric for feedback on code quality in programming courses. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, Koli Calling '16, página 160–164, New York, NY, USA. Association for Computing Machinery.

Striewe, M. e Goedicke, M. (2014). A review of static analysis approaches for programming exercises. In *Computer Assisted Assessment. Research into E-Assessment: International Conference, CAA 2014, Zeist, The Netherlands, June 30–July 1, 2014. Proceedings*, páginas 100–113. Springer.

Tigina, M., Birillo, A., Golubev, Y., Keuning, H., Vyahhi, N., e Bryksin, T. (2023). Analyzing the quality of submissions in online programming courses. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, páginas 271–282. IEEE.

Trichkova-Kashamova, E. (2021). Applying the iso/iec 25010 quality models to an assessment approach for information systems. In *2021 12th National Conference with International Participation (ELECTRONICA)*, páginas 1–4.

Van Der Werf, V., Swidan, A., Hermans, F., Specht, M., e Aivaloglou, E. (2024). Teachers' beliefs and practices on the naming of variables in introductory python programming courses. In *Proceedings of the 46th ICSE: SEET*, páginas 368–379.

Van Rossum, G., Warsaw, B., e Coghlan, N. (2001). Pep 8–style guide for python code. *Python. org*, 1565:28.

Varet, A. e Larrieu, N. (2013). Metrix: a new tool to evaluate the quality of software source codes. In *AIAA Infotech@ Aerospace (I@ A) Conference*, página 4567.

Vaucher, S., Khomh, F., Moha, N., e Guéhéneuc, Y.-G. (2009). Tracking design smells: Lessons from a study of god classes. In *2009 16th working conference on reverse engineering*, páginas 145–154. IEEE.

Vorobyov, K. e Krishnan, P. (2010). Comparing model checking and static program analysis: A case study in error detection approaches. *Proc. SSV*, páginas 1–7.

Wasik, S., Antczak, M., Badura, J., Laskowski, A., e Sternal, T. (2018). A survey on online judge systems and their applications. *ACM Comput. Surv.*, 51(1).

Weinberger, B., Silverstein, C., Eitzmann, G., Mentovai, M., e Landray, T. (2013). Google c++ style guide. *Section: Line Length. url: http://google-styleguide. googlecode. com/svn/trunk/cppguide. xml# Line_Length*.

Zampetti, F., Mudbhari, S., Arnaoudova, V., Di Penta, M., Panichella, S., e Antoniol, G. (2022). Using code reviews to automatically configure static analysis tools. *Empirical Software Engineering*, 27(1):28.