# PerfeQ: Um Ambiente Científico para a Análise da Qualidade de Código

**Francisco T. S. S. Pereira**[1], **Roberto A. Bittencourt**[1,2], **Elaine H. T. Oliveira**[3], **David B. F. Oliveira**[3]

[1] UEFS – Universidade Estadual de Feira de Santana
Av. Transnordestina, s/n, Novo Horizonte
Feira de Santana – BA, Brasil – 44036-900

[2]University of Victoria
3800 Finnerty Rd
Victoria – BC, Canada – V8P 5C2

[3]UFAM – Universidade Federal do Amazonas
Av. Gen. Rodrigo Octávio 6200 — Coroado I
Manaus – AM, Brasil – 69080-900

franncisco.p@gmail.com,rbittencourt@uvic.ca,

{elaine,david}@icomp.ufam.edu.br

***Abstract.*** *Faced with the need to improve the quality of students' source code, researchers seek alternatives to provide feedback on the code not only regarding its correctness, but also its quality. However, large classrooms do not allow for individual and complete feedback. In this context, tools such as Static Analyzers (SAs) can be used to analyze code without executing it, providing reports on its quality, such as style issues. This paper presents a tool to evaluate code quality regarding the use of programming language style conventions. To this end, we created metrics to summarize the quantification of code quality based on the results of the SAs. Finally, we created an environment to simplify the scientific analysis of student code datasets. The work presents a detailed description of the environment, as well as illustrative examples of its operation.*

***Resumo.*** *Para melhorar a qualidade do código dos estudantes, pesquisadores buscam alternativas de fornecer feedback sobre o código não somente quanto à corretude, mas também sobre a sua qualidade. Entretanto, percebe-se que a realidade de salas de aula com muitos alunos não permite um feedback individual e completo. Neste contexto, ferramentas como Analisadores Estáticos (AEs) podem ser utilizadas para realizar a análise do código sem executá-lo, fornecendo relatórios sobre sua qualidade como, por exemplo, problemas de estilo. A partir deste entendimento, o presente trabalho apresenta uma ferramenta para avaliação da qualidade de código em relação ao uso de convenções de estilo de linguagens de programação. Para tanto, criamos métricas para resumir a quantificação da qualidade do código a partir dos resultados dos AEs. Por fim, criamos um ambiente para simplificar a análise científica de datasets de código de estudantes. O trabalho apresenta uma descrição detalhada do ambiente, além de exemplos ilustrativos do seu funcionamento.*

## 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 também satisfaça requisitos de usabilidade, desempenho, confiabilidade e facilidade de manutenção [Finkbine 1996, 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.

As convenções de estilo variam de acordo com a linguagem de programação escolhida e, em geral, são amplamente adotadas pela comunidade de programadores de cada linguagem. Convenção de código é uma restrição sintática não imposta pela gramática de uma linguagem de programação [Allamanis et al. 2014]. No entanto, tais escolhas são importantes o suficiente para serem aplicadas por desenvolvedores de software. Essas regras influenciam diretamente a maneira como o desenvolvedor escreve os blocos de código, incluindo a indentação e a formatação dos nomes de variáveis e funções.

O tema de qualidade de software aparece no currículo de diversos cursos de graduação e pós-graduação, onde é comum os professores apresentarem esse conteúdo e discutirem a respeito do assunto [Hedberg et al. 2007]. Entretanto, devido à grande quantidade de alunos nas salas de aula, os professores não conseguem avaliar e apresentar um feedback individual a respeito da qualidade do código entregue pelos estudantes.

Para automatizar a correção de códigos entregues pelos alunos, permitindo assim um feedback individual, pesquisadores criaram os juízes online, também chamados de *autograders* em inglês. Essas ferramentas são capazes de identificar se o código do estudante 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 automática podem apresentar dicas de como os erros podem ser corrigidos [Galvão et al. 2016]. A utilização de ferramentas de análise automatizada pode auxiliar professores e assistentes ao fornecer um nível refinado de detalhes para a avaliação, utilizando a análise automatizada para verificar as seguintes características nos códigos dos estudantes: (i) corretude; (ii) estilo; (iii) eficiência e (iv) complexidade [Mengel and Yerramilli 1999].

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

Para assegurar que os códigos desenvolvidos por estudantes atendam aos padrões desejáveis de qualidade de software, ferramentas de análise de código como analisadores estáticos (AEs) podem ser utilizadas por professores para fornecer feedback sobre estrutura e convenções de estilo. 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*, problemas de estilo e até mesmo erros de memória e de ponteiros [Edwards et al. 2017, Gomes et al. 2009, Lima et al. 2021].

Por outro lado, analisadores estáticos apresentam limitações no contexto educacional, seja pela interface de usuário oferecida, pelo excesso de mensagens geradas ou pelas limitações dos tipos de mensagens capturadas. Procuramos resolver estes problemas na forma de uma ferramenta integrada voltada para pesquisadores e professores. Assim, este trabalho apresenta PerfeQ, uma ferramenta integrada de avaliação da qualidade de código de estudantes de programação, com foco na aderência às convenções de estilo de linguagens de programação, incluindo métricas criadas para quantificar essa aderência.

## 2. Fundamentação Teórica

Nesta seção, apresentamos os fundamentos para um melhor entendimento deste trabalho.

### 2.1. Análise Estática de Código

A análise estática de código é definida como o processo automático de examinar o código-fonte de um programa sem executá-lo. Esse tipo de análise identifica erros de execução durante a compilação, sem necessidade de instrumentação do código ou interação com o usuário. Essa análise se diferencia da análise dinâmica, que consiste na computação dos estados durante o tempo de execução do programa, sendo necessária a execução do código para realizar a análise [Vorobyov and Krishnan 2010].

A utilização de analisadores estáticos pode trazer alguns benefícios para o desenvolvimento de um software como, por exemplo, a redução da necessidade de depuração de código. Além de permitir também que a sua implementação seja direcionada para atingir métricas de qualidade como a confiabilidade e a legibilidade [Gomes et al. 2009].

A seguir serão apresentadas algumas ferramentas de análise estática que são utilizadas com as linguagens de programação C/C++ e Python.

Para realizar a análise estática de código nas linguagens de programação C e C++, foram criados os analisadores estáticos CppCheck[1], CQMetrics (C Code Quality Metrics)[2] e CppLint[3], dentre outras ferramentas. O CppCheck não busca detectar erros de sintaxe, sendo seu principal objetivo identificar vulnerabilidades no código e comportamentos indefinidos que podem ser perigosos para os programas, como vazamentos de memória e recursos[Dos Santos and Martimiano 2023, Joshi et al. 2014]. O CQMetrics busca analisar o código quanto às métricas de qualidade de software e estilo de código[Cannon et al. 1991]. Por fim, o CppLint é um analisador estático criado pelo Google, focado em identificar problemas de estilo no código, especialmente aqueles relacionados à formatação[Weinberger et al. 2013].

Para apoiar a análise estática com a linguagem Python, foram criadas, dentre outras, as ferramentas Pylint[4] e Pyflakes[5]. A primeira busca verificar o código com base nos padrões de desenvolvimento da linguagem, baseando-se nas métricas de qualidade definidas pelo guia *Python Enhancement Proposal 8* (PEP8)[6] [Liawatimena et al. 2018,

[1]https://cppcheck.sourceforge.io
[2]https://github.com/dspinellis/cqmetrics
[3]https://github.com/cpplint/cpplint
[4]https://pypi.org/project/pylint/
[5]https://pypi.org/project/pyflakes/
[6]https://peps.python.org/pep-0008/

Van Rossum et al. 2001]. A segunda não busca detectar erros de sintaxe, focando somente em erros lógicos [Gulabovska and Porkoláb 2019].

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 retornados por funções; (viii) uso inadequado de ponteiros [Novak et al. 2010].

### 2.2. Estilo e Convenções de Código

O estilo de código está diretamente relacionado à maneira como o programador escreve o seu código. Todos os desenvolvedores possuem um estilo de codificação, o que afeta a organização do seu código e como eles entendem códigos escritos por outros desenvolvedores [Reiss 2007]. Visando melhorar a manutenibilidade e garantir que a forma de codificar um programa em uma determinada linguagem seja similar para diferentes desenvolvedores, foram criadas as normas e convenções de código.

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* [Berry and Meekings 1985].

As convenções de código podem determinar as preferências a respeito de nomes de identificadores, como deve ser o layout de uma classe, relacionamentos entre objetos e até mesmo padrões de design [Allamanis et al. 2014, dos Santos and Gerosa 2018].

O *PEP8* é a convenção de estilo de codificação que é utilizada para definir os padrões de codificação em Python [Van Rossum et al. 2001]. Nesse guia, são apresentados tópicos para uma codificação mais legível. Já em C, a literatura e o repositório do GitHub CS50[7] buscam apresentar as convenções de codificação na linguagem para melhor codificação e melhor legibilidade do código [Doland and Valett 1994].

## 3. Metodologia

Nessa seção, será descrita a metodologia adotada para o desenvolvimento da ferramenta proposta neste trabalho.

### 3.1. Criação de um Analisador Estático

Antes da criação da ferramenta, realizamos uma análise dos principais analisadores estáticos no mercado, apresentados na Seção 2.1. O objetivo principal desta etapa foi a verificação de pontos fortes e fracos desses analisadores.

Para realizar esta análise, utilizamos códigos reais 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 utilizaram as linguagens de programação C e Python. A escolha pelo *dataset* da UFAM se deu pela facilidade

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

de obtenção dos dados a partir da ferramenta de juiz online *CodeBench*. Os códigos serviram como entrada para o analisador estático e as suas saídas (feedback com mensagens de aviso) foram analisadas quanto ao resultado esperado.

A partir dessa análise, identificamos que os analisadores *CppLint* e *Pylint* oferecem feedback válido quanto ao layout 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 – por exemplo, não apresentam mensagens sobre o tamanho dos identificadores.

Nesse contexto, foi necessário criar um novo analisador estático para as linguagens C e Python, nomeado NamingCheck[8], com foco na análise de identificadores de variáveis e funções seguindo os padrões das linguagens da Seção 2.2.

O NamingCheck foi escrito em Python. Ele recebe como entrada o código-fonte com as extensões **.c** ou **.py**. Em seguida, realiza uma análise sintática do código. 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. A Tabela 1 apresenta todas as mensagens de aviso apresentadas pela ferramenta.

**Tabela 1. 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. |

As mensagens de aviso do analisador estático desenvolvido foram baseadas em trabalhos prévios [Doland and Valett 1994, Van Rossum et al. 2001] e na documentação do CS50[9], que apresentam as convenções de estilo de código para C e Python.

Um detalhe a respeito da última mensagem da Tabela 1 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 as utilizam como contadores em *loops*.

### 3.2. Integração dos analisadores

Para a criação de uma ferramenta cujo objetivo seja fornecer um feedback mais completo a respeito de convenções de código, foi necessário realizar a integração dos analisadores estáticos existentes (Cpplint e Pylint) com o analisador criado (NamingCheck). A partir dessa integração, pode-se fornecer avisos sobre o código tanto em relação ao seu layout quanto em relação à nomeação de identificadores. 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 2.

[8]https://github.com/franciscotis/NamingCheck

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

**Tabela 2. Atributos referentes à análise estática do código.**

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

Esses dados são bastante úteis para a criação de métricas que servirão como uma forma de caracterizar quantitativamente a qualidade dos códigos analisados.

### 3.3. Métricas

A partir dos resultados dos analisadores estáticos apresentados anteriormente, criamos métricas para quantificar de forma sintética a aderência de um dado código-fonte às convenções de estilo da linguagem de programação.

Os atributos presentes na Tabela 2 resumem características do código e das mensagens dos AEs de forma absoluta. Para uma melhor visão sobre esses códigos, normalizamos as quantidades, seja em relação ao número de linhas de código, seja em relação ao total de ocorrências de declarações de identificadores. Assim, a partir da normalização dos atributos previamente descritos, criamos as métricas apresentadas na Tabela 3.

**Tabela 3. Métricas criadas e suas definições.**

| Métrica | Definição |
|---|---|
| WPL | Warnings per Lines Of Code (LOC) |
| VWPV | Variable warnings per number of Variables |
| FWPF | Function warnings per number of Functions |
| FWPL | Formatting warnings per LOC |

A métrica **WPL** informa a quantidade de mensagens de aviso por linhas de código. **VWPV** mensura a quantidade de mensagens de aviso referentes a variáveis pelo número de variáveis. **FWPF** informa a quantidade de mensagens de aviso referentes a funções pelo número de funções. Por fim, **FWPL** mensura a quantidade de mensagens de aviso referentes à formatação de código pelo número de linhas de código. Estas métricas podem ser apresentadas como frações ou percentuais.

## 4. PerfeQ

### 4.1. Visão Geral

A partir das limitações e desenvolvimentos apresentados na Seção 3, criamos a ferramenta PerfeQ[10]. Ela foi desenvolvida para, primeiramente, encapsular a informação de vários analisadores estáticos, incluindo o novo analisador NamingCheck que desenvolvemos, em um único ambiente. Além disso, ao apresentar informações sobre um código-fonte

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

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 em relação à aderência a convenções de código.

A ferramenta foi desenvolvida utilizando a linguagem de programação Python. Integra os AEs *Pylint* e *CppLint*, além do NamingCheck, o AE desenvolvido e apresentado na Seção 3. Adicionalmente, quantifica a avaliação do código a partir das métricas apresentadas na seção anterior. A ferramenta tem suporte para as linguagens Python e C, podendo ser utilizada tanto por pesquisadores quanto por professores e estudantes.

### 4.2. Arquitetura da Ferramenta

A Figura 1 apresenta uma visão arquitetural dinâmica da ferramenta desenvolvida.

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 usada, o analisador Pylint ou o Cpplint será acionado. Em seguida, será acionado o NamingCheck, que gera mensagens sobre a nomeação de identificadores. 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 2). Em seguida, a partir dos atributos computados, são geradas as métricas apresentadas na Tabela 3. Como saída, o integrador disponibiliza ao usuário os valores das métricas no terminal e um arquivo .csv contendo essas métricas, além de informações sobre o código, como número de linhas, variáveis e funções.

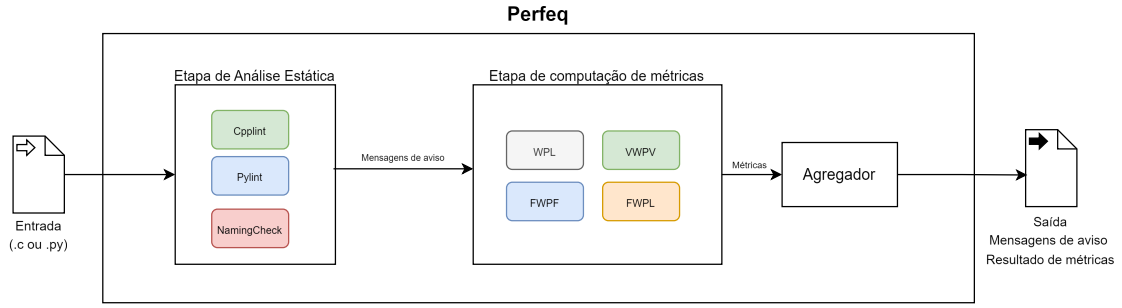


**Figura 1. Visão arquitetural dinâmica da ferramenta**

### 4.3. Limitações

Como descrito previamente, a ferramenta atualmente possui suporte apenas para as linguagens de programação C e Python. Outras linguagens que são frequentemente utilizadas em disciplinas de programação não estão disponíveis para análise.

Pelo estágio preliminar de desenvolvimento, a ferramenta não possui uma interface gráfica para uma melhor visualização das análises de código (e.g., exibição de gráficos, boxplots ou um *dashboard* integrado). A utilização e visualização de resultados se dá pela utilização de um terminal em modo texto. Entendemos que a ferramenta pode ser integrada a outros ambientes como juízes online, e manter a interface simples pode

facilitar esta integração futura. Por outro lado, pesquisadores podem utilizar os dados de saída da ferramenta em outras ferramentas de análise de dados para a geração de gráficos e realização de análises estatísticas mais detalhadas.

## 5. Cenários de Uso

Nesta seção, o funcionamento da ferramenta é apresentado. Para utilizá-la, 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.

### 5.1. Funcionamento

Nesta seção, apresentamos o comportamento do sistema ao realizar a análise de arquivos. A sua utilização se dá por meio do terminal, onde o usuário deve informar a localização absoluta do código-fonte que deseja analisar.

Inicialmente, o programa realiza a análise estática do código desejado, e apresenta no console as mensagens de aviso geradas pelos analisadores estáticos, como ilustrado na Figura 2. Em seguida, os valores, em percentuais, das métricas de qualidade também são apresentados, como ilustra a Figura 3. A Figura 4 apresenta o arquivo *.csv* que também é gerado. Nele, é apresentado um resultado mais detalhado da análise.

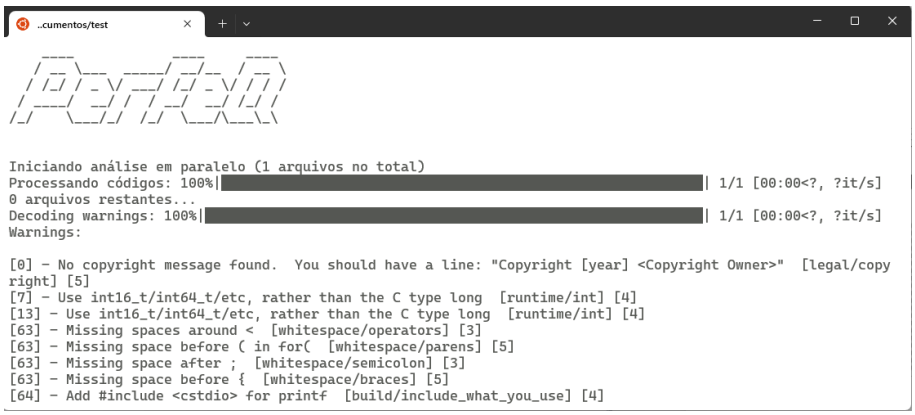


**Figura 2. Saída da ferramenta PerfeQ**

Caso o usuário opte por informar o caminho de uma pasta, a ferramenta realiza a análise de todos os arquivos na pasta. Neste caso, somente o arquivo .csv é gerado.

### 5.2. 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

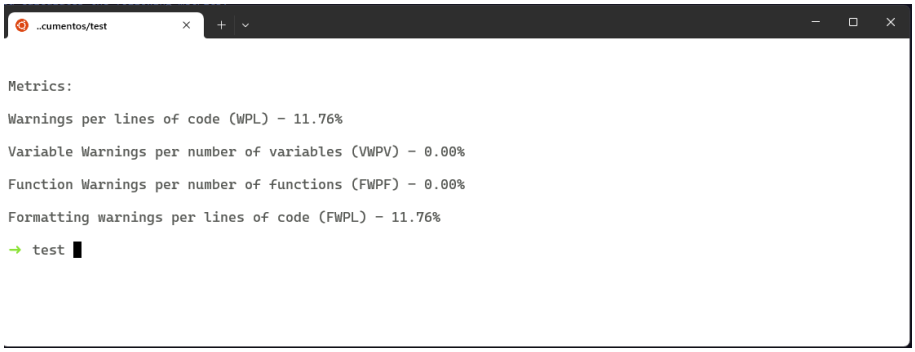


**Figura 3. Apresentação das métricas**

```
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. Arquivo .csv com os resultados detalhados da análise**

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 verificar que os estudantes possuem problemas na nomeação de variáveis e funções, e em seguida reforçar esse assunto em sala de aula.

### 5.3. Utilização por estudantes

A ferramenta também permite que estudantes possam obter feedback a respeito da qualidade de estilo dos 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.

### 5.4. 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.

## 6. Trabalhos Relacionados

Poucos artigos criam ferramentas voltadas para a melhoria da qualidade de código de estudantes de programação. Saliba et al. (2024) apresentam uma análise do estilo de código dos estudantes a fim de melhorar as suas práticas de codificação. Esse 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.

Outros trabalhos discutem como a análise estática de código pode ser utilizada na sala de aula pela utilização da ferramenta *PyTA*. Esta pode ser um complemento eficaz para os erros de programação que os alunos cometem e pode ajudá-los a identificar e resolver os erros mais comuns [Liu and Petersen 2019].

A ferramenta criada e apresentada neste trabalho (NamingCheck) buscou suprir uma lacuna que os AEs atuais apresentam: a falta de análise de nomeação de identificadores de variáveis e funções. Diferentemente dos AEs existentes, este tem como foco principal verificar se essa nomeação segue as convenções das linguagens C e Python.

Em uma breve comparação com os trabalhos disponíveis na literatura, verifica-se que o presente trabalho se destaca pelo objetivo de fornecer um feedback mais completo quanto à qualidade de código. Além disso, ao ser disponibilizado como uma ferramenta *open source* e com uma interface simples, permite oferecer maior suporte para os principais usuários: professores, estudantes e pesquisadores. Finalmente, além de disponibilizar as mensagens de aviso fornecidas pelos AEs, a ferramenta também procura quantificar a qualidade de código por meio das métricas voltadas para convenções de estilo.

## 7. Conclusões

Este trabalho apresentou a ferramenta PerfeQ, cujo objetivo é fornecer um melhor feedback a respeito da qualidade do código em termos de aderência a convenções de estilo de linguagens. Atualmente, a ferramenta analisa código nas linguagens C e Python.

A ferramenta integra dois analisadores estáticos conhecidos, Cpplint e Pylint, por oferecerem um bom feedback quanto à aderência a convenções de layout de código, embora não o façam em relação à nomeação de identificadores. Por isso, criamos um novo analisador estático, NamingCheck, que oferece feedback sobre aderência a convenções de nomeação de identificadores. Ao utilizar a ferramenta, o usuário tem acesso às mensagens de aviso geradas pelos AEs e também o resultado de quatro métricas criadas para quantificar a qualidade do código de forma sintética.

Trabalhos futuros envolvem a integração da ferramenta com sistemas de juízes online, criação de uma interface gráfica com um *dashboard* para uma melhor visualização e análise dos dados, além de possíveis usos de inteligência artificial para melhorar o feedback para os estudantes – apresentando uma melhor análise a respeito de nomeações, verificando também o seu significado, além de apresentar sugestões para melhorar o código. Análise de usos da ferramenta também pode servir para o seu melhoramento.

## Agradecimentos

Este trabalho recebeu apoio do Conselho Nacional de Desenvolvimento Cientifico e Tecnologico (Processo 303443/2023-5).

## Referências

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

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

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.

Doland, J. and Valett, J. (1994). C style guide. Technical report, National Aeronautics and Space Administration.

Dos Santos, G. T. H. and 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. and Gerosa, M. A. (2018). Impacts of coding practices on readability. In *Proceedings of the 26th conference on program comprehension*, pages 277–285.

Edwards, S. H., Kandru, N., and 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, page 65–73, New York, NY, USA. ACM.

Finkbine, R. B. (1996). *Metrics and Models in Software Quality Engineering*, volume 21. Association for Computing Machinery, New York, NY, USA.

Galvão, L., Fernandes, D., and 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, page 140.

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

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

Hedberg, H., Iivari, N., Rajanen, M., and 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, page 2, USA. IEEE Computer Society.

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

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

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

Lima, Y., Fonseca, I., Chagas, J., Rodrigues, E., Silveira, M., and 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*, pages 159–168, Porto Alegre, RS, Brasil. SBC.

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

Mengel, S. A. and 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, page 78–82, New York, NY, USA. Association for Computing Machinery.

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

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

Saliba, L., Shioji, E., Oliveira, E., Cohney, S., and 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*.

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

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

Weinberger, B., Silverstein, C., Eitzmann, G., Mentovai, M., and Landray, T. (2013). Google c++ style guide.