



Universidade do Minho  
Escola de Engenharia

# Ferramentas e Técnicas de *Coding Standards*

Investigação sobre ferramentas de qualidade de *software* e/ou  
testes de *software*

Submetido por:

Diogo Araújo A78485

Diogo Nogueira A78957

# Conteúdo

Contextualização.....	3
Indicadores de Qualidade <i>Software Code</i> .....	4
Qualidade do Código .....	4
Características de Qualidade de <i>Software</i> .....	5
ISO 25000: <i>Software and Data Quality</i> .....	6
Métricas de Qualidade de <i>Software</i> .....	7
<i>Coding Standards</i> .....	8
Definição .....	8
Propósito/Funcionalidade .....	8
Vantagens da presença de <i>Coding Standards</i> .....	9
Desvantagens da ausência de <i>Coding Standards</i> .....	10
Práticas e Técnicas Recomendadas.....	11
Ferramentas Usadas .....	16
Linguagem de Programação <i>Java</i> .....	18
<i>Standards</i> para <i>Java</i> específicos .....	18
Ferramenta <i>open-source</i> para verificação automática .....	20
Linguagem de Programação <i>Python</i> .....	22
<i>Standards</i> para <i>Python</i> específicos.....	22
Ferramenta <i>open-source</i> para verificação automática .....	24
Conclusões .....	27
Referências .....	28

# Contextualização

O presente trabalho surge no sentido de investigar ferramentas e técnicas de qualidade de *software* e/ou testes de *software* sendo o foco primordial a métrica de *Coding Standards*, que como o próprio nome o indica refere-se ao conjunto de padrões/normas ao nível do *source code* presente em qualquer produto de *software*.

Com esta métrica em mente e com os tópicos base que permitem criar uma contextualização correta de todo o assunto da qualidade de um produto *software*, traçam-se os seguintes objetivos para esta investigação:

- Compreender a importância da qualidade do código em termos de produto de *software*;
- Como pode ser definido um modelo de qualidade para um produto de *software*;
- A necessidade de existência de métricas mais pragmáticas para criar todo um sistema de avaliação justo;
- Estudo da métrica dos *Coding Standard*, tudo aquilo que representam e como podem/devem (e porquê) ser usados consoante as diferentes linguagens de programação;

De modo a se alcançarem estes objetivos existirá uma fase inicial que se focará em entender um pouco o artigo/documento fornecido pelo docente, criando-se uma introdução e explicando a necessidade de criação de métricas que consigam avaliar de forma justa a qualidade de um produto de *software*. Com esta base mais teórica entra-se no “mundo” dos padrões de codificação, onde se detalha toda a sua definição, importância e termina-se com o seu uso para diferentes linguagens de programação.

A metodologia usada para esta investigação foi o uso do documento de apoio fornecido pelo docente e alguma pesquisa bibliográfica enriquecida com fontes *online* que abordam todo este assunto da *Code Quality* e *Coding Standards* no desenvolvimento de um *software*.

# Indicadores de Qualidade *Software Code*

Quando se fala em Indicadores de Qualidade de *Software Code*, refere-se especificamente a tudo aquilo que define a boa ou a má qualidade do código pensado para um determinado *software*. Claro que tudo isto se torna subjetivo na visão de cada programador/empresa, dado que consoante a equipa que está a desenvolver o produto em si, podem existir diferentes definições, dependendo do contexto em que o *software* se insere.

Este capítulo surge no sentido de tentar compreender a importância e o impacto destes indicadores na qualidade geral de um *software*, para dar depois entrada/foco em toda a forma como os *Coding Standards* se encaixam nestes indicadores e como atuam positivamente no auxílio para as equipas de desenvolvimento.

## Qualidade do Código

A tentativa de definir a qualidade do código pode muitas vezes ser difícil de pôr em prática, já que cada programador tem uma perspetiva diferente em relação aquilo que constitui um código bem pensado e desenvolvido. Dessa forma, não existe uma definição única de qualidade, sendo então necessário abordar um conjunto de fatores que contribuem e afetam a mesma, criando-se uma conformidade na avaliação final.

O trabalho/documento fornecido pela TIOBE de seu nome **The TIOBE: *Quality Indicator a pragmatic way of measuring code quality*** propõe-se a definir um sistema de medição de qualidade de software de um modo pragmático/simples, tomando como base mais de 15 anos de experiência e análise de 1 bilhão de linhas de código de *software*.

Esse documento refere uma definição ISO de qualidade de *software*, que se torna basilar estudar para contextualizar perante a métrica dedicada para este projeto – *Coding Standards*.

# Características de Qualidade de *Software*

Conforme se tentou deixar patente na secção anterior, existe uma ISO 25000 que surge no sentido de definir um modelo de qualidade para um produto de *software*. Este padrão ISO é nomeado de ISO/IEC 25010 e **estipula oito fatores principais de qualidade**, que permitem ter uma ideia mais substancial daquilo que pode ajudar a tornar um produto de *software* o melhor possível à vista geral dos utilizadores:

1. ***Functional Suitability*** – o produto apresenta integridade, correção e adequa-se ao propósito pretendido;
2. ***Performance Efficiency*** – o produto faz uma boa gestão dos recursos ao longo do tempo, consoante a capacidade disponível;
3. ***Compatibility*** – o produto tem uma boa compatibilidade em termos de troca de informações e/ou execução de funções necessárias perante os vários sistemas a partilhar o mesmo ambiente de *hardware* ou *software*;
4. ***Usability*** – o produto demonstra facilidade em reconhecer o propósito/algoritmo do programa em si, tendo em conta o visual que é apresentado ao utilizador consoante as necessidades pensadas para o mesmo e ainda na acessibilidade caso o utilizador possua alguma incapacidade;
5. ***Reliability*** – o produto apresenta maturidade, disponibilidade, tolerância à falha e capacidade de recuperar em situações necessárias;
6. ***Security*** – o produto deve garantir a confidencialidade, integridade, não repúdio, autenticação e ainda um bom ambiente de testes;
7. ***Maintainability*** – o produto deve ser modular de modo a poder ser reutilizável e dessa maneira ser mais fácil de testar, analisar e modificar;
8. ***Portability*** – o produto deve estar pronto para ter partes substituíveis, adaptando-se a novas e futuras instalações/atualizações.

## ISO 25000: *Software and Data Quality*

Quando se fala no desenvolvimento de *software* como um todo, é essencial definir/abordar toda a ideia de qualidade do produto, juntamente com a qualidade do processo em si.

A ISO/IEC 25000 compreende uma série de normas, cujo objetivo principal passa por orientar o desenvolvimento de produtos de *software* através da especificação de requisitos de qualidade e das características de qualidade da avaliação. Esta norma foi criada no sentido de definir um modelo de qualidade do produto, que determina as características de qualidade a ser levadas em consideração ao avaliar as propriedades de um produto de *software*.

Apesar de toda a ideia deste padrão fazer sentido e servir como ponto de partida para criar um sistema de avaliação da qualidade de um produto num estado inicial, apresenta dois entraves que devem ser tomados em conta:

- Note-se que este padrão não define/especifica nenhuma forma de medir estas características de qualidade, muito menos a sua unidade de medida em si;

Basta pensar em no atributo *Portability* e nos seus subatributos. Como poderia ser feita a medição da *Adaptability* de um produto e qual a unidade de medida usada para tal?

- Estas características/atributos que supostamente delineiam a qualidade de um produto podem ter diferentes significados dependendo do contexto. Mesmo que se consiga medir uma destas características, é impossível definir critérios 100% objetivos sobre o que é realmente bom ou mau.

O atributo *Performance Efficiency* mostra bem isso, dado que para um determinado programa uma resposta de 1 segundo é insignificante, mas para outro programa pode já não ser.

Neste caso, como podemos definir o que é realmente bom ou mau de uma forma justa para todos os produtos de *software*?

# Métricas de Qualidade de *Software*

Apesar da norma ISO/IEC 25010 ser uma base para determinar a qualidade de um produto de *software*, compreende-se a sua ambiguidade e os seus inconvenientes para ser credível ao ponto de servir como sistema de medição universal.

De forma a se obter uma maneira mais sistemática de medir e qualificar estas medições, mitigando toda a abordagem feita pela ISO 25000, **surgem oito métricas de qualidade de código de *software* mais usadas atualmente no setor e que podem ser medidas de um modo automatizado:**

1. *Code coverage;*
2. *Abstract interpretation;*
3. *Cyclomatic complexity;*
4. *Compiler warnings;*
5. *Coding standards;*
6. *Code duplication;*
7. *Fan out;*
8. *Security.*

Tendo em conta o propósito pensado para o trabalho, vai-se focar toda a atenção na métrica de *Coding Standards* e dessa forma compreender a importância destas normas na qualidade geral de um produto de *software*.

# ***Coding Standards***

## **Definição**

*“Coding standards are a set of industry-recognized best practices that provide a variety of guidelines for developing software code. There is evidence to suggest that compliance to coding standards in software development can enhance team communication, reduce program errors and improve code quality.”*

Com isto se entende que os *Coding Standards* consistem num conjunto de regras e diretrizes que determinam o estilo, procedimentos e métodos de programação para uma determinada linguagem de programação.

## **Propósito/Funcionalidade**

A ideia de escrever código sem convenções e padrões pode facilmente tornar-se algo confuso e insustentável, principalmente quando se está perante uma grande base de código pertencente a um produto de *software*. Ao trabalhar em equipa, torna-se crucial definir regras e diretrizes que cada elemento deve então seguir de modo garantir que o produto final seja confiável e com uma manutenção consistente.

Isto significa que para existir um código bem estruturado e limpo, os *developers* precisam de estar a par dos padrões/convenções usados pela sua equipa no geral, dado que dessa forma estão a facilitar a compreensão do mesmo por parte de *developers* que estejam a entrar pela primeira vez na base de código.

Por isso, é imperativo que existam cláusulas pré-definidas para as várias linguagens de programação, criando-se assim um propósito evidente no uso destas convenções.



## Vantagens da presença de *Coding Standards*

Depois de se compreender o propósito da existência de padrões de codificação no desenvolvimento de um produto de *software*, consegue-se criar desde logo um conjunto de vantagens fortes o suficiente para justificar a presença destes *Coding Standards*.

Assim, e pensando-se em todo o trabalho de equipa existente ao desenvolver um *software* e nas possíveis futuras intervenções/correções do código nele envolvido, enumeram-se as seguintes vantagens:

- **Aumento da eficiência na qualidade do código:** Uma vez que os *developers* gastam uma quantidade significativa de tempo a tentar resolver problemas que poderiam ter sido evitados, estabelecer *Coding Standards* permite que a equipa detete determinados problemas de uma forma antecipada ou até mesmo os evite por completo, aumentando assim a eficiência no processo do *software* como um todo.
- **Aumento da facilidade de manutenção:** Seguindo-se um conjunto base de regras e padrões, o código revela-se mais consistente, tornando toda a sua manutenção mais simples e intuitiva.

Basta pensar em termos de *developers* futuros e no uso de grandes bases de código. Quanto mais universal for a escrita da linguagem de programação em si, mais fácil se torna entender/modificar por qualquer programador.

- **Aumento da facilidade de deteção/correção de *bugs*:** Toda a tarefa de deteção e posterior correção de possíveis *bugs* existentes no *software* torna-se mais simples quando se está perante um código mais conciso.
- **Redução nos riscos de falhas:** Dado que muitos projetos acabam por falhar devido a problemas provenientes do desenvolvimento do *software*, a implementação de padrões e convenções ao nível do código pode efetivamente reduzir um conjunto de problemas e consequentemente o risco de falhas do projeto em si.

- **Redução da complexidade do código:** Sabe-se por experiência própria e do que se vê de grandes empresas, que quanto mais complexo é o código que está por detrás de todo o motor de *software*, mais vulnerável o mesmo é relativamente ao aparecimento de erros.

Com uso de *Coding Standards*, consegue-se produzir/escrever um código mais simples e legível, diminuindo-se a complexidade e consequentemente a ocorrência de erros.

- **Redução do custo do desenvolvimento de *software*:** Um código bem pensado e desenvolvido dá aos *developers* a oportunidade de reutilizar o mesmo sempre que necessário. Esse pensamento/ideia pode reduzir de forma radical o custo do projeto, reduzindo também todo o esforço necessário na parte de desenvolvimento.

Com tudo isto se entende a importância significativa que os *Coding Standards* têm em qualquer desenvolvimento de *software* que seja realmente bem sucedido.

Não só cria a possibilidade de trabalhar em equipa de uma forma entendível para todos os elementos, como facilita na manutenção de todo o código num futuro próximo, promovendo boas práticas de programação.

## Desvantagens da ausência de *Coding Standards*

Estando enumeradas algumas das principais vantagens da existência de *Coding Standards* em qualquer projeto de *software*, quase que se torna trivial deduzir o impacto negativo que a inexistência destas regras pode trazer.

Note-se que sem qualquer tipo de padrão ao nível do código, os *developers* acabarão por usar os seus próprios métodos para programar, criando-se assim uma grande quantidade de diferenças, o que pode resultar em:

- **Problemas ao nível da segurança do *software*:** Os principais motivos das vulnerabilidades de *software* comumente exploradas são devido a inconsistências, *bugs* e erros ao nível lógico.

Isso significa que alguns desses mesmos problemas podem surgir devido a erros de código que são resultantes de práticas inadequadas ao nível da programação.

- **Motivação reduzida por parte de toda a equipa:** Normalmente, um projeto engloba um conjunto significativo de *developers*, que trabalham todos sobre o mesmo código através de tarefas específicas para cada um deles.

Ao existir uma disparidade muito grande ao nível de código, cria-se uma dificuldade maior em compreender o mesmo, reduzindo a motivação por parte de toda a equipa.

- **Aumento do tempo de desenvolvimento:** Pegando-se na ideia estipulada no tópico anterior, facilmente se entende que quanto maior for o tempo usado para entender a tarefa desempenhada por cada elemento, maior será o tempo total do desenvolvimento de *software*.
- **Estrutura complexa da base do código:** Quantas mais regras e normas existirem, e sendo elas todas diferentes perante cada elemento da equipa, maior será a complexidade da estrutura base do código.

## Práticas e Técnicas Recomendadas

Ao pensar-se no trabalho por parte de grandes empresas que têm uma quantidade grande de *developers* a trabalhar sobre grandes projetos de código, depressa se deteta a necessidade de criar uma conformidade universal para todos os envolvidos no projeto.

Conforme mencionado nas secções anteriores, uma maneira fácil de fazer isso passa por simplificar o código desenvolvido através da ajuda dos *Coding Standards*. A ideia

é manter um estilo de código padrão e bem definido, através de padrões e diretrizes, que normalmente vão ao encontro do tipo de *software* que se está a desenvolver.

Apesar de existirem padrões específicos a cada empresa/projeto, existem sempre padrões mais comuns e gerais que devem sempre ser aplicados apesar da linguagem da programação em si:

1. **Documentação adequada do código:** É boa prática comentar o código à medida que se vai escrevendo/desenvolvendo o mesmo, dado que dessa forma se está a facilitar outros *developers* a entender todos os métodos e declarações feitas.
  - O uso de uma IDE apropriada para a linguagem de programação em causa e até mesmo outras ferramentas pode ajudar a utilizar/criar os vários comentários de diversas maneiras;
  - O aconselhável é sempre comentar cada método/função faz, bem como os seus parâmetros recebidos, valores retornados e até mesmo erros e exceções caso se aplique.

```
/**
 * Função que trata de adicionar um novo elemento
 * @param argumento1 Argumento 1
 * @param argumento2 Argumento 2
 * @return int          Valor de Saida
 */

int adicionaElemento(argumento1, argumento2) {

    // Código
    // (...)

}
```

Neste caso, fez-se uso da linguagem *Java* para demonstrar a prática em questão. Note-se que este modelo pré definido de documentação é muitas vezes semi criado pelos próprios IDE's, tal como se referiu anteriormente.

2. **Formatar e Indentar de forma legível:** Um bom código deve apresentar um formato e indentação padronizados. Dependendo da linguagem de programação em causa, podem existir regras específicas de formatação/indentação, no entanto, existem algumas convenções que devem ser respeitadas.

- Existência de um espaço após a vírgula entre dois argumentos/parâmetros de uma função/método;
- As habituais chavetas devem começar após a declaração da função e dos seus parâmetros. Após isso, uma nova linha e a chaveta final.
- Cada pedaço de código deve estar devidamente recuado e espaçado.

```
def adicionaElemento(argumento1, argumento2):  
    print("Teste")  
  
    if True:  
        print("Testado")  
    else:  
        print("Não Testado")  
  
    print("Código Fora da Definição")
```

O exemplo dado faz todo o sentido, dado que a linguagem Python tem já no seu interior toda a ideia de se trabalhar com indentações ao invés dos habituais parenteses. Dessa forma se entende a importância de saber empregar estas regras e a facilidade que se cria em compreender aquilo a que cada elemento pertence.

No caso de linguagens, o uso de chavetas tem de ser sempre bem pensado e espaçado tal como o exemplo anterior assim o demonstra.

3. **Evitar comentar partes de código desnecessárias:** Menos é sempre mais e muitas das vezes torna-se desnecessário comentar certas partes do código. Não só desocupa espaço visual como dá uma ideia mais profissional acerca do mesmo.

4. **Esquema adequado/consistente para nomenclatura** : Usar uma nomenclatura para nomear os vários métodos, classes, variáveis, etc, é sempre uma das melhores formas de criar uma harmonia visual e tornar a identificação dos vários nomes ao longo do código muito mais simples.

- Uso de *camel case* que define que a primeira letra de cada palavra é maiúscula, exceto a primeira palavra;
- Uso de *underscore* que define o uso do caracter *underscore* a delimitar cada palavra;
- Dependendo da linguagem de programação, podem existir *guidelines* para as variáveis locais e globais, através do **CamelCase** ou **UnderScore**.
- Evitar o uso de números na nomenclatura geral;
- O nome dado a uma variável, função, etc, deve ser alusivo ao motivo por detrás do uso em si, de uma forma clara e breve.

```
// Exemplo Camel Case

function adicionarElemento() {

    // elementoResultante = (...)

}

// Exemplo Under Score

function adicionar_elemento() {

    // elemento_resultante = (...)

}
```

5. **Evitar usar muitas estruturas/condições aninhadas:** A existência de muitas estruturas/condições aninhadas pode efetivamente complicar a interpretação de como toda a função/método funciona.

```
if() {  
    if() {  
        if() {  
            // Código  
        }  
    }  
}
```

6. **Linhas e funções de comprimento curto:** Ao usar-se linhas com um comprimento mais curto, cria-se uma facilidade visual mais confortável e legível. O mesmo se aplica à construção de toda as funções. Dessa forma, é sempre preferível criar funções mais pequenas, dividindo trabalho/tarefas em outras funções mais auxiliares se necessário.

```
// Prática Não Recomendada  
int valor1; int valor2; char valor2;  
  
// Maior Conforto Visual  
int valor1, valor2;  
  
char valor2;
```

7. **Organizar o código por pastas e arquivos:** Evitar escrever todo o código num único ficheiro, de modo a evitar problemas relacionados à manutenção futura. Torna-se sempre melhor organizá-lo em diferentes pastas e ficheiros indo ao encontro da funcionalidade de todo o *software*.

Estas são algumas das técnicas universais a todas as linguagens e que ao serem empregues criam desde logo uma facilidade muito grande no entendimento do código.

A secção que se segue demonstra que, consoante a linguagem de programação usada, existem outras tantas regras e padrões que devem ser tomados em consideração, e que através de ferramentas criadas para o efeito é possível manter o código de acordo com os mesmos, mitigando erros de escrita ou ambiguidades existentes.

## Ferramentas Usadas

O *Stack Overflow's Annual Developer Survey* é a maior e mais abrangente pesquisa de pessoas que praticam a programação espalhada pelas suas várias linguagens existentes. Através dessa pesquisa anual, é-nos fornecida uma visão geral acerca de várias temáticas ligadas ao mundo dos programadores, sendo uma delas referente às linguagens de programação usadas ao longo do ano corrente.

O gráfico que se segue diz respeito aos dados obtidos para o ano de 2019, e através dele consegue-se perceber as linguagens mais populares entre os *developers* ao redor do mundo.

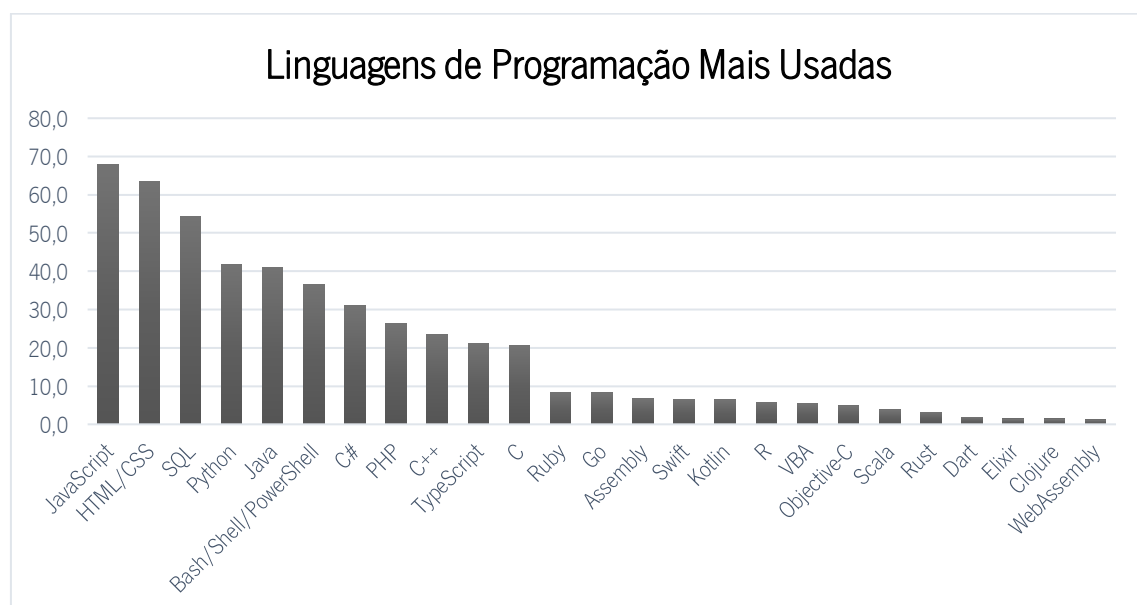


Figura 1 Gráfico com dados Linguagens de Programação Mais usadas em 2019



Linguagem	Ferramenta 1	Ferramenta 2	Ferramenta 3
JavaScript	ESLint	Google's Closure Compiler	JSLint
Python	PyCharm	Pylint	—————
Java	PMD	IntelliJ IDEA	Checkstyle
C#	CodeRush	.NET Compiler Platform	StyleCop
C, C++	Lint	Visual Studio	Clang
Multi Linguagem	Coverity Kiuwan	Kiuwan	DeepCode

Tabela 1 Ferramentas *Coding Standards* para Top 5 Linguagens 2019

A tabela supracitada deixa uma nuance de algumas ferramentas usadas para o top 5 de linguagens apresentadas no gráfico e ainda algumas que acabam por ser universais a mais do que uma linguagem. Na secção das referências encontra-se um *link* que serviu de base para esta escolha e que lista um conjunto muito grande de outras ferramentas.

Nesta secção iremos focar em duas linguagens de programação amplamente usadas no mundo de programação que tiveram no seu período de concepção métodos diferentes para abordar um *Coding Standard* universal a todos os *developers* que futuramente utilizassem a dita linguagem de programação.

- Python tem na sua génese uma filosofia nuclear que consiste em:

*“Beautiful is better than ugly. Explicit is better than implicit. Simple is better than complex. Complex is better than complicated. Readability counts.”*<sup>1</sup>

Tornou-se uma linguagem que desde a sua concepção tinha em ideia existir um *Coding Standard* universal, chamado de PEP8 (Guido van Rossum, 2001) que se tornou a base para a convenção universal de qualquer programador em *Python*.

<sup>1</sup> Fonte: PEP20, The Zen of Python: <https://www.python.org/dev/peps/pep-0020/>

- *Java*, na sua génese não seguia um *code guideline* que fosse universal e ditado pelo desenvolvedor da mesma.

No entanto, com a sua enorme popularidade e crescimento, principalmente na partilha de código e importação de bibliotecas, houve a necessidade de standardizar algumas regras e por consequência a criação de ferramentas automáticas para verificação e correção das mesmas.

## Linguagem de Programação *Java*

### *Standards* para *Java* específicos

Após a análise de *standards* gerais falados na secção acima, conseguimos pesquisar alguns *standards* específicos à linguagem *Java* aqui em estudo. Dessa forma, iremos demonstrar alguns exemplos encontrados bem como um código *Java* que não seja standardizado de forma a depois numa secção seguinte o testar e verificar os erros ditos pela ferramenta escolhida.

Quando falamos em termos de linguagem *Java* associamos de imediato uma organização própria em termos de *source code*, dado que no próprio *Java Development Kit* se encontra incluído um programa denominado *javadoc* cujo objetivo passa por processar os vários arquivos de código em *Java* e produzir uma documentação externa, no formato de arquivos HTML, para os seus programas *Java*.

Este gerador é desde logo a base de uma boa prática pertencente à linguagem, já que através dele se entende a grande importância que uma boa documentação de todo o código tem sobre um produto final de *software*.

Além desta parte da documentação ser essencial para a produção de um bom código, existem algumas normas basilares a ter em conta:

- Primeiro devem ser declarados os *packages* e só depois todo os *imports*;
- A existência de comentários (análogos à documentação);

- Declarar as variáveis do mesmo tipo na mesma linha, por forma a evitar linhas desnecessárias;
- As classes em *Java* devem seguir regras de espaçamento pelos vários parenteses;
- Os métodos devem ser separados por uma linha em branco;
- O uso de *camel case* como prática para escrever os nomes dos métodos, variáveis, classes, *packages* e constantes.

Dessa forma, o código *Java* abaixo tem uma série de falhas tais como:

- Uso de uma negação na parte do *if* com a cláusula *else*;
- Fazer o *import* de todo o *package* *java.lang* mesmo sem ser necessário para o código em causa;
- Uso de *Dollar Sign* no nome do método;
- Criação de um ciclo *for* para definir uma ideia de continuidade de uma condição quando poderia ser usado um ciclo *while*;

```
package com.engseg.grupo5;
import java.lang.*;

public class Main {

    private static void teste$codigo(int x){

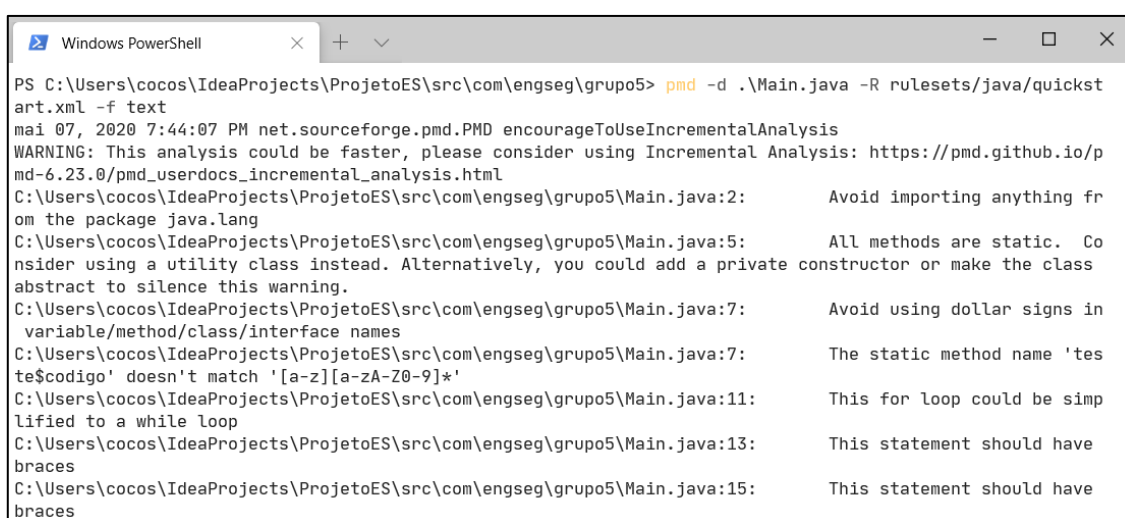
        int y = 4;

        for(;x==3;){
            if (x != y)
                System.out.println("Diferentes");
            else
                System.out.println("Iguais");
        }
    }

    public static void main(String[] args) {
        teste$codigo(3);
    }
}
```

## Ferramenta *open-source* para verificação automática

Uma das razões por se ter escolhido *Java* como uma das linguagens para estudo de ferramentas de *Coding Standards* foi o facto da mesma não apresentar um *guideline* pensado pela própria fundação. Isto nota-se pela diferença de resultados, mas ao utilizar a ferramenta PMD recomendada pelo TIOBE conseguimos verificar na figura abaixo que todas as “falhas” do *guideline* seguido aparecem na análise final verificando assim o poder da ferramenta open-source.



```
PS C:\Users\cocos\IdeaProjects\ProjetoES\src\com\engseg\grupo5> pmd -d .\Main.java -R rulesets/java/quickstart.xml -f text
mai 07, 2020 7:44:07 PM net.sourceforge.pmd.PMD encourageToUseIncrementalAnalysis
WARNING: This analysis could be faster, please consider using Incremental Analysis: https://pmd.github.io/pmd-6.23.0/pmd_userdocs_incremental_analysis.html
C:\Users\cocos\IdeaProjects\ProjetoES\src\com\engseg\grupo5\Main.java:2:      Avoid importing anything from the package java.lang
C:\Users\cocos\IdeaProjects\ProjetoES\src\com\engseg\grupo5\Main.java:5:      All methods are static. Consider using a utility class instead. Alternatively, you could add a private constructor or make the class abstract to silence this warning.
C:\Users\cocos\IdeaProjects\ProjetoES\src\com\engseg\grupo5\Main.java:7:      Avoid using dollar signs in variable/method/class/interface names
C:\Users\cocos\IdeaProjects\ProjetoES\src\com\engseg\grupo5\Main.java:7:      The static method name 'testecodigo' doesn't match '[a-z][a-zA-Z0-9]*'
C:\Users\cocos\IdeaProjects\ProjetoES\src\com\engseg\grupo5\Main.java:11:     This for loop could be simplified to a while loop
C:\Users\cocos\IdeaProjects\ProjetoES\src\com\engseg\grupo5\Main.java:13:     This statement should have braces
C:\Users\cocos\IdeaProjects\ProjetoES\src\com\engseg\grupo5\Main.java:15:     This statement should have braces
```

Figura 2 Output ferramenta PMD para programa teste *Java*

Como título de exemplo temos uma porção do código-fonte da ferramenta **PMD** que define a classe que verifica a regra de evitação do carácter \$ no nome dos objetos da linguagem *Java*, tornando-se assim visível a standardização do código.

```
public class AvoidDollarSignsRule extends AbstractJavaRule {

    @Override
    public Object visit(ASTClassOrInterfaceDeclaration node, Object data) {
        if (node.getImage().indexOf('$') != -1) {
            addViolation(data, node);
            return data;
        }
        return super.visit(node, data);
    }
}
```

Para uma pequena comparação colocamos o mesmo código *Java* em teste no IDE comercial e *closed-source* IntelliJ IDEA que tem implementado uma ferramenta de análise de código intensiva e pudemos confirmar que a análise tem similaridades com a ferramenta *open-source* escolhida, mas ainda tem mais análise de redundância e ainda uma avaliação sobre possíveis *bugs* do valor das constantes.

Isto acontece porque o *code standard* do *Java* é algo que não foi acordado pela empresa que o mantém e cria, logo as diversas ferramentas neste ecossistema podem pensar em variâncias de regras, fugindo assim a uma standardização universal para todos os desenvolvedores *Java*.

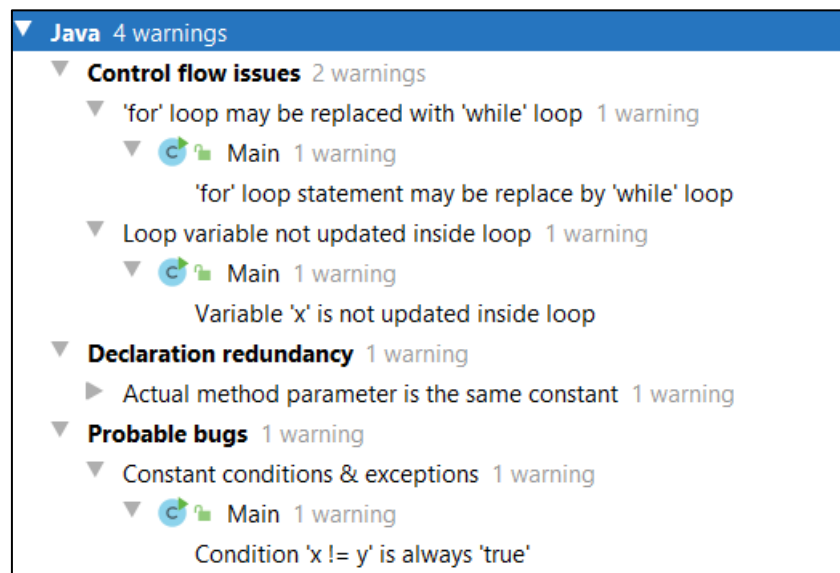


Figura 3 Output IntelliJ IDEA para programa teste *Java*

# Linguagem de Programação *Python*

## Standards para *Python* específicos

A linguagem de programação multifacetada e preferida pelos *developers* mundialmente foi lançada em 1991 e toda a filosofia enfatizava uma legibilidade de código melhor que os concorrentes na altura, focando-se no uso útil do “espaço em branco”. Com tal em mente a fundação que gere e mantém a linguagem de programação criou todo um documento evolutivo para ser a regra base do estilo e estandardização de como a legibilidade e utilização das ferramentas devia tomar de forma a tornar a compreensão de código *Python* universal e fácil de aprender e analisar.

PEP 8<sup>2</sup> pertence a um grupo de propostas de melhoramento da linguagem (daí o *Python Enhancement Proposals (PEP)*) e a oitava proposta serve para esse propósito de melhorar a legibilidade e entendimento do código *Python*. Como o criador do *Python* disse: “Código é lido muito mais vezes que é escrito” e daí vem a importância do seguimento de certas regras que realmente melhorem todo o processo de programação.

Existem assim várias convenções pensadas desde logo para a linguagem *Python*:

- Existe convenções de nomes dos objetos, como a não utilização das letras “O,I,l” que podem ser confundidas com os valores de 0 e 1, respetivamente.
- Os nomes das funções, variáveis, métodos devem ser todos com letra minúscula separando palavras com o carácter *underscore*.
- As classes devem seguir o estilo de *camel case* de forma a se distinguir facilmente.
- As constantes devem ser sempre com caracteres maiúsculos.

---

<sup>2</sup> Presente e sempre atualizado em: <https://www.python.org/dev/peps/pep-0008/>

- Existem também convenções sobre o número de linhas brancas após diferentes tipos de objetos no código, sobre a preferência de espaço invés de *tabs* e também sobre a indentação correta a ter nos vários sítios do código desde argumentos dos métodos a corpos de código dentro de ciclos ou condições.

Todo este processo de aglomerar o estilo de código e a standardização ao nível mais alto numa linguagem, tornou-se efetivamente algo que toda a gente começou a seguir e usar, tornando assim a comunidade de programação mais célere sempre que fosse preciso analisar código *Python* e a sua consequente aprendizagem.

Como forma de teste e similar à outra linguagem de programação analisada anteriormente neste documento, iremos criar um pedaço de código funcionalmente compilável, mas com falhas no seguimento dos *standards* pedidos pelo **PEP 8**.

Dessa forma, o código *Python* abaixo tem uma série de falhas tais como:

- Múltiplas importações na mesma linha;
- Mais do que 2 linhas a separar o código anterior do novo método;
- O nome do método não segue a convenção *snake\_case*, ou seja, letras minúsculas e palavras separadas por *underscore*;
- O nome das variáveis não segue a mesma convenção supracitada;
- A indentação de espaços na declaração do método e os seus argumentos está errada confundindo-se com corpo de código;

```
import os, sys

def FuncaoErrada(VARIAVEL,
                 VARIAVEL2):

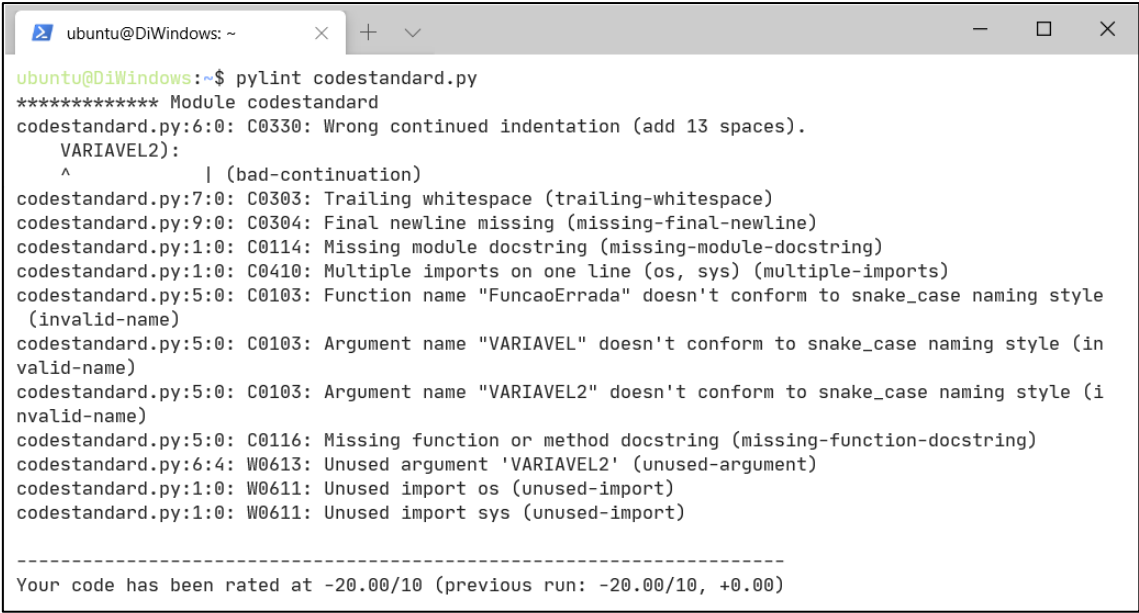
    if VARIAVEL is int:
        print("É um inteiro")
```

## Ferramenta *open-source* para verificação automática

Uma ferramenta recomendada pela TIOBE é utilizada amplamente pela grande parte dos programadores *Python* e a o **PyLint**. Este *linter* instalado de maneira simples em qualquer instância *Python* e suportado pela maioria dos editores de texto utilizados, tornou-se uma ferramenta obrigatória para cuidar do standard do código, bem como a continua análise do mesmo fornecendo ajudas visuais e avisos para as falhas que o código possa ter. Na sua versão de linha de comandos ainda providencia um sistema de avaliação por notas de 0 a 10 de forma a quantificar melhor o quão bom ou não o código produzido está da estandardização do **PEP 8**.

Ao correr o programa para a porção de código mencionada acima, vemos automaticamente uma série de recomendações que o código falhava e a sua posição respetiva. Desde os *multiple-imports* a *invalid-name* passando pelo *trailing-whitespace* e também outras verificações que a ferramenta como importações não utilizadas, ou seja, redundantes para o código em si.

Dessa forma, no fim acontece a avaliação e aquela porção do código tinha um *score* de -20 em 10, o que demonstra facilmente o quão fora da estandardização estava.

A terminal window titled 'ubuntu@DiWindows: ~' shows the command 'pylint codestandard.py' being executed. The output lists various Pylint errors and warnings for the file 'codestandard.py'. The errors include 'Wrong continued indentation', 'Trailing whitespace', 'Final newline missing', 'Missing module docstring', 'Multiple imports on one line', and 'Function name doesn't conform to snake\_case naming style'. The warnings include 'Unused argument', 'Unused import os', and 'Unused import sys'. At the bottom, the overall score is displayed as -20.00/10.

```
ubuntu@DiWindows:~$ pylint codestandard.py
***** Module codestandard
codestandard.py:6:0: C0330: Wrong continued indentation (add 13 spaces).
    VARIAVEL2):
        ^      | (bad-continuation)
codestandard.py:7:0: C0303: Trailing whitespace (trailing-whitespace)
codestandard.py:9:0: C0304: Final newline missing (missing-final-newline)
codestandard.py:1:0: C0114: Missing module docstring (missing-module-docstring)
codestandard.py:1:0: C0410: Multiple imports on one line (os, sys) (multiple-imports)
codestandard.py:5:0: C0103: Function name "FuncaoErrada" doesn't conform to snake_case naming style
    (invalid-name)
codestandard.py:5:0: C0103: Argument name "VARIAVEL" doesn't conform to snake_case naming style (in
valid-name)
codestandard.py:5:0: C0103: Argument name "VARIAVEL2" doesn't conform to snake_case naming style (i
nvalid-name)
codestandard.py:5:0: C0116: Missing function or method docstring (missing-function-docstring)
codestandard.py:6:4: W0613: Unused argument 'VARIAVEL2' (unused-argument)
codestandard.py:1:0: W0611: Unused import os (unused-import)
codestandard.py:1:0: W0611: Unused import sys (unused-import)

-----
Your code has been rated at -20.00/10 (previous run: -20.00/10, +0.00)
```

Figura 4 *Output* ferramenta Pylint para programa teste Python



Como o código-fonte da ferramenta **PyLint** é de livre acesso, podemos analisar a variedade de classes e métodos que o programa tem para verificar estas falhas no *standard Python*.

Como exemplo, temos em baixo o método que verifica se os *imports* estão a ser colocados todos numa linha apenas, colocando assim a mensagem desse erro que não cumpre o **PEP 8**.

```
(...)  
  
@check_messages(*MSGs)  
def visit_import(self, node):  
    """triggered when an import statement is seen"""  
    "  
        self._check_reimport(node)  
        self._check_import_as_rename(node)  
        self._check_toplevel(node)  
  
        names = [name for name, _ in node.names]  
        if len(names) >= 2:  
            self.add_message("multiple-  
imports", args=", ".join(names), node=node)  
  
(...)
```

Para uma pequena comparação colocamos o mesmo código *Python* em teste no IDE comercial e *closed-source* **PyCharm** que tem implementado uma ferramenta de análise de código intensiva e pudemos confirmar que a análise foi completamente idêntica à ferramenta *open-source* estudada na página anterior.

Isto deve-se ao facto de ter existido um esforço basilar da fundação *Python* para criar um *guideline* geral e universal para toda o ecossistema envolvendo a sua linguagem, tornando-se assim uma estandardização correta e implementada por todos, desde os criadores, até aos desenvolvedores, passando ainda pelos leitores ou entusiastas.

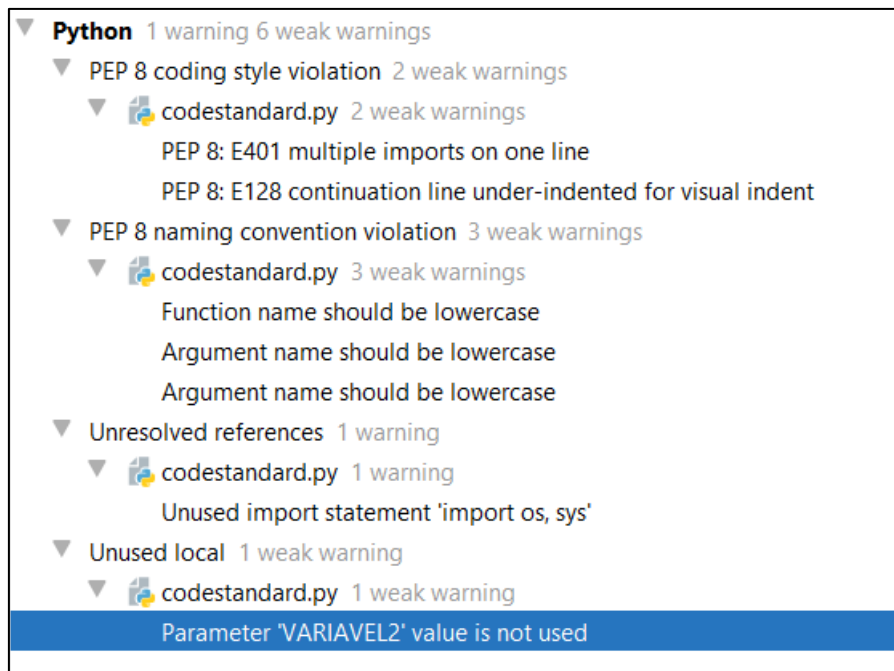


Figura 5 *Output* PyCharm para programa teste Python

# Conclusões

Este projeto de investigação conseguiu aprofundar as necessidades de uma standardização para o estilo e funcionamento de código de programação. Ao analisarmos todas as características que definem a qualidade de *software* e as métricas associadas, tornou-se essencial verificar o que na verdade eram os *standards* de código.

A ideia de escrever código não padronizado pode tornar-se confuso, principalmente quando se está perante uma grande base de código dum *software*. Ao trabalhar em equipa, torna-se crucial definir regras e diretrizes para existir uma consistência pelo menos a nível da equipa.

Na sequência da análise e pesquisa sobre o tema decidimos focar-nos em duas linguagens utilizadas por milhões de desenvolvedores que, na sua génese, tiveram abordagens diferentes tomando consequencialmente divergências na standardização da escrita de código nos ambientes respetivos. Ao analisar a diferença do *Python* com os seus *guidelines* definidos pela própria fundação e os seus criadores contrastava com a liberdade dos *guidelines* seguidos pelas diversas ferramentas de análise a código *Java*. Isto via-se a nível prático quando para ambientes *Python* existia uma ferramenta *open-source* que se tornou a *de facto* para toda a comunidade e *Java* tinha na verdade uma panóplia que divergia em certos termos sobre como devia ser a standardização do código.

A qualidade do *software* vem sempre da facilidade de o ler, compreender, modificar e corrigir de maneira mais rápida, logo é importante existir pelo menos uma tentativa universal para isso acontecer de forma a ajudar toda a comunidade de desenvolvedores que diariamente produzem milhões de linhas de código. Essa organização é imperativa para um ciclo de vida e reutilização de código saudável, elevando a comunidade de programação para algo com regras, sugestões que irão sobretudo ajudar e poupar trabalho futuro.

# Referências

- Boff, R. J. (s.d.). *Padrões de Codificação no Desenvolvimento de Sistemas*. Obtido de Micreiros: <https://micreiros.com/padroes-de-codificacao-no-desenvolvimento-de-sistemas/>
- ckan. (s.d.). Obtido de Python coding standards: <https://docs.ckan.org/en/2.8/contributing/python.html>
- Guido van Rossum, B. W. (5 de julho de 2001). *PEP 8 – Style Guide for Python Code*. Obtido de <https://www.python.org/dev/peps/pep-0008/>
- List of tools for static code analysis*. (8 de maio de 2020). Obtido de Wikipedia: [https://en.wikipedia.org/wiki/List\\_of\\_tools\\_for\\_static\\_code\\_analysis](https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis)
- Rezvi, M. (7 de janeiro de 2020). *gitconnected*. Obtido de Write Better Code with Coding Standards: <https://levelup.gitconnected.com/write-better-code-with-coding-standards-546faf3fd4d1>
- Sengayire, P. (25 de novembro de 2019). *CODING STANDARDS AND CONVENTIONS IN SOFTWARE DEVELOPMENT TEAM*. Obtido de Medium: <https://medium.com/@psengayire/the-importance-of-coding-standards-and-conventions-in-the-software-development-team-how-they-can-5d252556a05>
- TIOBE Coding Standard Methodology*. (s.d.). Obtido de TIOBE: <https://www.tiobe.com/coding-standard-methodology/>