



UNIVERSIDADE DO MINHO  
DEPARTAMENTO DE INFORMÁTICA  
ENGENHARIA DE SEGURANÇA

---

## Projeto 2

Ferramentas e técnicas de *Coding standards*

---

### Grupo 4

*Autores:*

Joel Gama (A82202)



Tiago Pinheiro (A82491)



11 de Maio de 2020

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>2</b>
<b>2</b>	<b><i>Coding Standards</i></b>	<b>3</b>
2.1	O que são <i>Coding Standards</i> ? . . . . .	3
2.2	Utilização de <i>Coding Standards</i> ? . . . . .	3
<b>3</b>	<b>Ferramentas de <i>Coding Standards</i></b>	<b>4</b>
3.1	<i>TICS Analyzer</i> . . . . .	4
3.2	<i>QA-C/C++</i> . . . . .	5
3.3	<i>PyLint</i> . . . . .	6
3.4	<i>FindBugs</i> . . . . .	7
3.5	Ferramentas by <i>Github Projects</i> . . . . .	8
3.5.1	<i>Swift Lint</i> . . . . .	8
3.5.2	<i>Detekt</i> . . . . .	8
3.5.3	<i>ESLint</i> . . . . .	9
<b>4</b>	<b><i>Coding Standards</i></b>	<b>10</b>
4.1	Uso de variáveis globais . . . . .	10
4.2	<i>Standard headers</i> para diferentes módulos . . . . .	10
4.2.1	Exemplo . . . . .	11
4.3	<i>Naming conventions</i> para variáveis locais e globais, constantes e funções . . . . .	11
4.4	Indentação . . . . .	11
4.4.1	Mau exemplo . . . . .	12
4.4.2	Bom exemplo . . . . .	12
4.5	<i>Error return values</i> e conveções para <i>handling</i> da exceções . . . . .	12
4.6	Evitar o uso de identificadores para múltiplos propósitos . . . . .	12
4.7	O código deve estar bem documentado . . . . .	12
4.8	As funções não devem ser muito grandes . . . . .	13
4.9	Tentar não usar o <i>GOTO statement</i> . . . . .	13
<b>5</b>	<b>Conclusão</b>	<b>14</b>

# Introdução

O presente relatório surge no âmbito da Unidade Curricular de Engenharia de Segurança integrada no perfil de Criptografia e Segurança da Informação.

Este projeto é o segundo num total de três projetos que estão previstos ser abordados nesta UC. O objetivo do projeto é investigar sobre a qualidade de software e/ou testes de software. Devido à existência de vários grupos de trabalho nesta UC foram introduzidos diferentes temas que iriam ser abordados por diferentes grupos. Ao nosso grupo de trabalho foi-nos atribuído o seguinte tema: *Ferramentas e Técnicas de Coding Standards*.

Assim, este tema divide-se em dois subtemas: ferramentas e técnicas. Primeiramente, será feita uma apresentação das principais ferramentas de *coding standards* que existem, englobando boa parte das linguagens de programação mais usadas. Depois, irão ser apresentadas as várias técnicas de *coding standards* existentes para algumas das linguagens de programação, sem entrar em grande detalhe.

# *Coding Standards*

## **2.1 O que são *Coding Standards*?**

A manutenção de *software* é uma das tarefas que mais trabalhosas realizadas por um engenheiro de *software*. Uma das principais razões pelas quais é trabalhoso fazer a manutenção do código é a dificuldade em entender código escrito à algum tempo, especialmente depois de vários *updates*. Uma maneira para reduzir os custos na manutenção é a introdução de *coding standards*

*Coding standards* são um conjunto de regras que os engenheiros devem seguir. Estas regras abrangem muitos aspetos do código, como por exemplo, construções de código a evitar, *naming conventions*, *layout*, entre outros. A maior parte das regras preocupam-se com a manutenção e facilidade de leitura mas existem regras que se preocupam com a *performance* e com a transferibilidade.

Geralmente, as organizações utilizam padrões próprios e certas diretrizes, dependendo de quais se adequam à sua organização e produtos que desenvolvem. Portanto, é fundamental que os programadores mantenham os *coding standards*, caso contrário o código será rejeitado durante a revisão.

## **2.2 Utilização de *Coding Standards*?**

*Coding standards* são utilizados por várias razões, sendo as principais razões as seguintes:

- *Coding standards* dão uma aparência uniforme aos códigos escritos por diferentes engenheiros.
- Melhoram a legibilidade e a manutenção do código e também reduzem a complexidade.
- Ajudam na reutilização de código e na deteção de erros facilmente.
- Promovem boas práticas de programação e aumentam a eficiência dos programadores.

# Ferramentas de *Coding Standards*

## 3.1 *TICS Analyzer*

O *TICS Analyzer* é um analisador que é normalmente integrado em ambientes de programação, podendo também ser utilizado na linha de comandos. Ele funciona como uma camada extra sobre os verificadores de código utilizados, gerando violações nos *coding standards* registados. Para além disso, pode também ser utilizado como forma de unir os resultados de diferentes analisadores de código. As linguagens de programação suportadas pelo *TICS Analyzer* são C, C++ e PL/SQL.

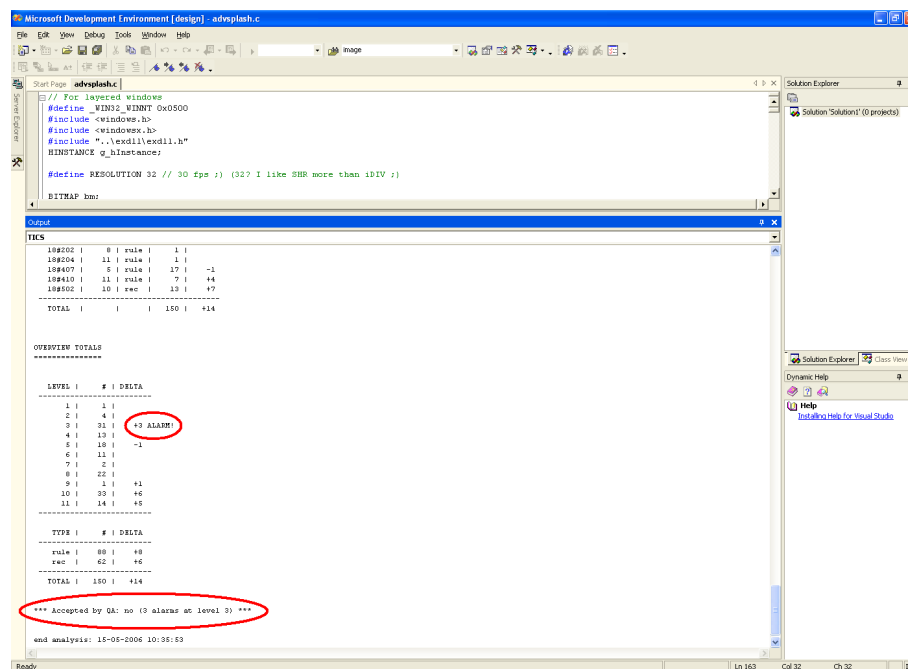


Figura 3.1: Exemplo de utilização do *TICS Analyzer*.

No exemplo, o *TICS Analyzer* faz uma análise pelos diferentes níveis verificando os problemas encontrados em cada um deles. No final atribui uma avaliação qualitativa à análise. Esta avaliação apenas diz se o código foi ou não aceite pela ferramenta e quais os problemas encontrados.

## 3.2 QA-C/C++

QA-C/C++ é utilizado como analisador de código C e C++. Identifica problemas no software na fase de desenvolvimento. Como encontra os bugs à medida que eles ocorrem reduz significativamente o custo e esforço necessário para os resolver, permite escrever código limpo e evita vulnerabilidades no código. Assim, o QA-C/C++ permite manter os *coding standards* à medida que o código é escrito.

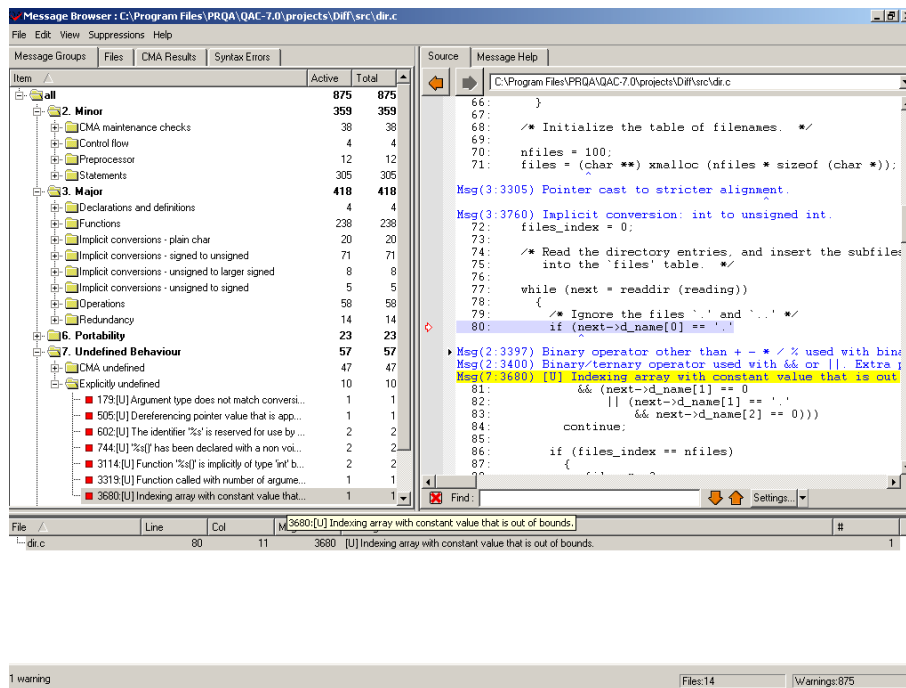


Figura 3.2: Exemplo de utilização do QA.

O modo de funcionamento do QA-C/C++ é bastante simples. Enquanto o código é escrito é feita uma verificação segundo regras da ferramenta. Caso alguma regra seja quebrada a ferramenta exibe uma mensagem ao programador durante a própria escrita do código.

### 3.3 PyLint

*Pylint* é uma ferramenta de verificação de código *Python*. Impondo *coding standards* e procurando *coding smells* no código. Pode também procurar por certos erros de tipo, recomendar refatoração de blocos de código e pode oferecer detalhes sobre a complexidade do código. Por fim, o código recebe uma classificação geral, com base no número e na gravidade dos avisos e erros.

```
pylint simple_calc.py
No config file found, using default configuration
***** Module simple_calc
R: 16:Maths.is_even: Method could be a function
R: 27:Maths.xcube: Method could be a function
R: 38:Maths.square: Method could be a function
R: 49:Maths.power: Method could be a function
R: 60:Maths.square root: Method could be a function
R: 71:Maths.factorial_number: Method could be a function
R: 82:Maths.check_prime: Method could be a function
Exception RuntimeError: 'maximum recursion depth exceeded while calling a Python
object' in <type 'exceptions.RuntimeError'> ignored

Report
=====
24 statements analysed.

Raw metrics
-----

+-----+-----+-----+-----+
|type    |number| %    |previous|difference|
+-----+-----+-----+-----+
|warning |0     |NC    |NC      |          |
+-----+-----+-----+-----+
|error   |0     |NC    |NC      |          |
+-----+-----+-----+-----+

Messages
-----

+-----+-----+
|message id|occurrences|
+-----+-----+
|R0201     |7          |
+-----+-----+

Global evaluation
-----
Your code has been rated at 7.08/10
```

Figura 3.3: Exemplo de utilização do *PyLint*.

O *Pylint* faz uma análise completa ao ficheiro *Python* no exemplo. Ele verifica o número de erros, *warnings* e classificando o código segundo os parâmetros estipulados.

### 3.4 FindBugs

O *FindBugs* é uma ferramenta que procura erros nos programas escritos em Java. É baseado no conceito de *coding standards*. O *FindBugs* usa uma análise estática para inspecionar o *bytecode* Java quanto às ocorrências de padrões de erros. A análise estática significa que o *FindBugs* pode encontrar erros simplesmente inspecionando o código de um programa. O *FindBugs* funciona analisando o *bytecode* Java (arquivos de classe compilados), para que não seja preciso o código-fonte do programa para usá-lo.

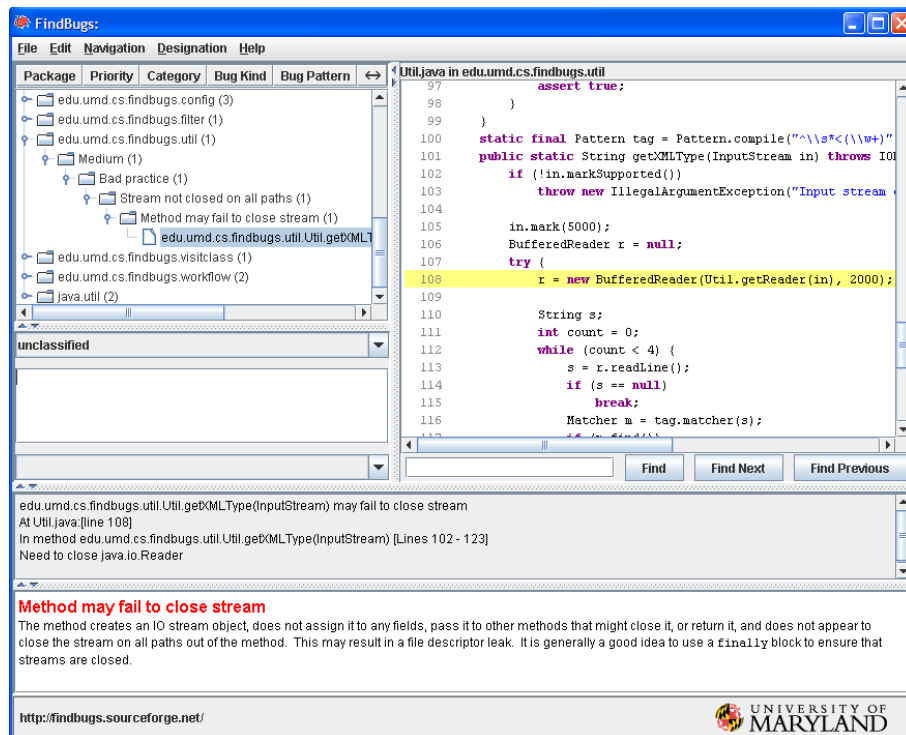


Figura 3.4: Exemplo de utilização do *FindBugs*.

Neste exemplo de utilização do *FindBugs* é feita a análise de um ficheiro Java. Podemos ver que é apontado uma possível causa de bug no programa. Sendo neste caso o facto de o *Reader* não ter sido fechado.



## 3.5 Ferramentas by Github Projects

### 3.5.1 Swift Lint

*Swift Lint* é uma ferramenta utilizada para ajudar a manter o estilo e convenções na programação em *Swift*. A ferramenta é baseada no *GitHub's Swift Style Guide* para fazer as correções e ajudar o utilizador a manter o código o mais organizado, legível e sem erros.

```
2
3 let someForceCast = NSObject() as! Int
4 let colonOnWrongSide :Int = 0
5 // SwiftLint is syntax-aware
6 // NSNumber() as! Int => no error
7 "let colonOnWrongSide :Int = 0" // => no error
8
```

Force Cast Violation: Force casts should be avoided. (force\_cast)  
Colon Violation: Colons should be next to the identifier when specifying a type. (colon)

Figura 3.5: Exemplo de utilização do *Swift Lint*.

A utilização do *Swift Lint*, assim como a do *QA-C/C++*, faz a análise enquanto o código está a ser escrito. Desta forma, é possível evitar erros no processo de desenvolvimento e não apenas numa análise posterior do código.

### 3.5.2 Detekt

O *Detekt* surge para ajudar na análise de ficheiros *Kotlin*. À semelhança das ferramentas anteriores o *Detekt* ajuda a manter o estilo de escrita de código, avisa sobre *code smells* e sugere alterações segundo um conjunto de regras definidas.

```
artur@artur-XPS15 ~/detekt$ java -jar detekt-cli/build/libs/detekt-cli-1.0.0-M0-all.jar -p /home/artur/Repos/detekt -c /home/artur/Repos/detekt/default-detekt-config.yml -i ./test/ --parallel
Analyzing 107 kotlin files: .....
Ruleset: code-smell
Ruleset: comments
Ruleset: complexity
  TooManyFunctions - 20/10 - [Findings.kt] - Line/Column=(1,1) - Path=detekt-api/src/main/kotlin/io/gitlab/arturbosch/detekt/api/Findings.kt
  NestedBlockDepth - 4/2 - [procedure] - Line/Column=(10,2) - Path=detekt-formatting/src/main/kotlin/io/gitlab/arturbosch/detekt/formatting/Indentation.kt
  ComplexCondition - 5/3 - [procedure] - Line/Column=(21,7) - Path=detekt-formatting/src/main/kotlin/io/gitlab/arturbosch/detekt/formatting/SpacingAfterComma.kt
  ComplexCondition - 7/3 - [procedure] - Line/Column=(53,7) - Path=detekt-formatting/src/main/kotlin/io/gitlab/arturbosch/detekt/formatting/SpacingAroundOperator.kt
  NestedBlockDepth - 4/3 - [procedure] - Line/Column=(22,2) - Path=detekt-formatting/src/main/kotlin/io/gitlab/arturbosch/detekt/formatting/OptionalSemicolon.kt
  NestedBlockDepth - 4/2 - [loadConfiguration] - Line/Column=(40,2) - Path=detekt-cli/src/main/kotlin/io/gitlab/arturbosch/detekt/cli/Runner.kt
  TooManyFunctions - 11/10 - [LargeClass.kt] - Line/Column=(1,1) - Path=detekt-rules/src/main/kotlin/io/gitlab/arturbosch/detekt/rules/complexity/LargeClass.kt
Ruleset: empty
  EmptyFunctionBlock - [postVisit] - Line/Column=(94,45) - Path=detekt-api/src/main/kotlin/io/gitlab/arturbosch/detekt/api/Rule.kt
  EmptyFunctionBlock - [preVisit] - Line/Column=(101,44) - Path=detekt-api/src/main/kotlin/io/gitlab/arturbosch/detekt/api/Rule.kt
  EmptyFunctionBlock - [onStart] - Line/Column=(12,35) - Path=detekt-core/src/main/kotlin/io/gitlab/arturbosch/detekt/core/FileProcessListeners.kt
  EmptyFunctionBlock - [onProcess] - Line/Column=(13,30) - Path=detekt-core/src/main/kotlin/io/gitlab/arturbosch/detekt/core/FileProcessListeners.kt
  EmptyFunctionBlock - [onFinish] - Line/Column=(14,55) - Path=detekt-core/src/main/kotlin/io/gitlab/arturbosch/detekt/core/FileProcessListeners.kt
Ruleset: exceptions
  CatchRuntimeException - [e] - Line/Column=(32,11) - Path=detekt-api/src/main/kotlin/io/gitlab/arturbosch/detekt/api/Junk.kt
Ruleset: potential-bugs
Ruleset: style
Complexity Report:
  - 1077 logical lines of code (lloc)
  - 523 McCabe complexity (mcc)
  - 311 mcc per 1000 lloc
  - 7 code smells per 1000 lloc
detekt run within 1025 ms
Exception in thread "main" io.gitlab.arturbosch.detekt.cli.SmellThresholdBuildFailure: Build failure threshold of 10 reached with 13 weighted smells!
at io.gitlab.arturbosch.detekt.cli.SmellThreshold.check(SmellThreshold.kt:31)
at io.gitlab.arturbosch.detekt.cli.Runner.execute(Runner.kt:25)
at io.gitlab.arturbosch.detekt.cli.Main$Companion.main(Main.kt:67)
at io.gitlab.arturbosch.detekt.cli.Main.main(Main.kt)
```

Figura 3.6: Exemplo de utilização do *Detekt*.

Neste exemplo de utilização do *Detekt* é feita a análise segundo determinados *Rule sets*. No final da análise das regras são apresentados os erros encontrados, *smells* e possíveis *bugs*.

### 3.5.3 *ESLint*

*ESLint* é uma ferramenta *Open Source* para *JavaScript*. É usada para encontrar padrões ou códigos problemáticos que não seguem determinadas directrizes de estilo. O *ESLint* permite encontrar erros de sintaxe e outros problemas sem ser necessário executar o código. Para além disso, o *ESLint* foi projetado para ter todas as regras completamente conectáveis.

```
vagrant@precise32:~$ eslint uploader.js

uploader.js
  1:13  error  'angular' is not defined      no-undef
  1:28  error  Strings must use doublequote  quotes
  7:15  error  Strings must use doublequote  quotes
  7:28  error  Strings must use doublequote  quotes
  7:34  error  Missing "use strict" statement  strict
  9:20  error  Expected '===' and instead saw '=='  eqeqeq
 10:17  error  Expected '===' and instead saw '=='  eqeqeq
 14:20  error  Strings must use doublequote  quotes
 27:16  error  'FormData' is not defined      no-undef
 34:17  error  'XMLHttpRequest' is not defined  no-undef
 35:1   error  Trailing spaces not allowed     no-trailing-spaces
 36:12  error  Strings must use doublequote  quotes
 45:1   error  Unexpected blank line at end of file  eol-last

â 13 problems

vagrant@precise32:~$
```

Figura 3.7: Exemplo de utilização do *ESLint*.

No exemplo de utilização do *ESLint* é possível ver a utilização da ferramenta num ficheiro *JavaScript*. No output são mostrados os problemas encontrados, dando como informação a linha e a descrição do mesmo assim como a sua categoria.

# *Coding Standards*

Anteriormente apresentamos a definição de *coding standards* e ferramentas que permitem avaliar código, ou seja, se cumprem ou não os padrões de definidos.

Os *coding standards* variam um pouco de linguagem para linguagem, principalmente, na indentação do código. De seguida apresentámos alguns dos *coding standards* existentes, assim como alguns exemplos para certas linguagens.

## **4.1 Uso de variáveis globais**

As variáveis globais devem ser usadas com moderação por várias razões:

- As variáveis globais podem ser alteradas em qualquer parte do código, sendo complicado descobrir todos os sítios onde a variável é alterada.
- As variáveis globais não tem controlo de acesso, não podem estar limitadas a certas partes do código.
- Usar variáveis globais causa poluição de nomes, ou seja, usar muitas variáveis globais faz com muito nomes já estejam ocupados e não possam ser usados ao longo do código.

## **4.2 *Standard headers* para diferentes módulos**

Para manter e entender melhor o código, os *headers* dos diferentes módulos devem seguir o formato e informação *standard*. Assim, o módulo devem conter a seguinte informação:

- Nome do módulo
- Data de criação do módulo
- Autor do módulo
- Histórico de modificação
- Um pequeno resumo sobre o que o módulo faz
- As funções que o módulo suporta assim como os seus parâmetros de *input* e de *output*
- Variáveis globais acedidas ou modificadas pelo módulo

### 4.2.1 Exemplo

```
# Name: Exemplo
# Creation Date: 05/10/2015
# Author: Grupo 4
# Modification history: never modiflicated
# Synopsis: Apenas para explicar o que é um módulo
# Functions: This module doesn't have any.
# Global variables: this module doesn't use any
global variables
```

## 4.3 *Naming conventions* para variáveis locais e globais, constantes e funções

Algumas conveções gerais de nomes:

- Os nomes das variáveis devem ter algum significado e devem ser perceptíveis (auto-explicatórias).
- As variáveis locais devem usar o tipo de escrita *camel case*, começando com letra minúscula (ex: localData). Por sua vez, as variáveis globais devem começar por letra maiúscula (ex: GlobalData). Por último, os nomes constantes devem usar apenas letra maiúscula (ex: CONSDATA).
- Evitar uso de dígitos nos nomes das variáveis.
- Os nomes das funções devem ser escritos em *camel case* começando com letra minúscula. Para além disso, o nome da função deve descrever a função e uma forma breve e clara (ex: somaNumeros).

## 4.4 Indentação

Uma indentação adequada é muito importante para facilitar a leitura do código. Para isso, existem algumas convenções simples:

- Deve haver um espaço depois da vírgula entre dois argumentos de uma função.
- Cada bloco de código deve estar bem indentado e espaçado.
- As chavetas devem estar alinhadas, uma com a outra. Os *statement* também devem estar alinhados com a chaveta de fecho.

#### 4.4.1 Mau exemplo

```
int compareNumbers(int a,int b){
    int r = 0;
    if(a<b){r=-1;}else{r=1;}
    return r;}
```

#### 4.4.2 Bom exemplo

```
int compareNumbers (int a, int b)
{
    int r = 0;

    if (a < b) {
        r = -1;
    } else {
        r = 1;
    }

    return r;
}
```

### 4.5 *Error return values* e convenções para *handling* da exceções

Todas as funções que encontrem uma condição de erro devem retornar 0 ou 1 para simplificar o *debugging*.

### 4.6 Evitar o uso de identificadores para múltiplos propósitos

Cada variável deve ter um nome descritivo e com significado, indicando a razão pela qual está a ser usada. Se um identificador for usado várias vezes por diferentes razões torna-se difícil entender o código.

### 4.7 O código deve estar bem documentado

O tópico é explica-se por ele mesmo. Para além dos nomes das variáveis e das funções serem apropriados é fundamental que o código esteja documentado para que se entenda o que as funções fazem e seja mais fácil perceber e modificá-las.

## 4.8 As funções não devem ser muito grandes

As funções não devem ser muito longas, caso sinta necessidade de fazer muita coisa numa função deve-se recorrer a funções/métodos auxiliares, tornando o código mais simples e elegante.

## 4.9 Tentar não usar o *GOTO statement*

Tentar ou NÃO USAR de todo o *GOTO statement* porque este faz com o programa perca a sua estrutura lógica e torna difícil a compreensibilidade do código e o seu *debugging*.

# Conclusão

Este trabalho foi interessante para explorar ferramentas de qualidade de software e testes de software, pelo menos, uma pequena parte deste tema bastante abrangente.

Através deste projeto foi possível aprender e interiorizar quais são as ferramentas de software usadas para classificar código através de *coding standards*. Por outro lado, também foi possível ficar a saber mais sobre quais as boas práticas para programar de acordo com os *standards* praticados pela comunidade.

Em suma, este projeto foi um meio bastante interessante de expor uma situação ao alunos e o grupo sente que foi uma maneira muito eficaz para adquirir conhecimento.