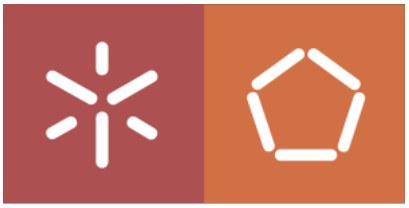

Universidade do Minho
Escola de Engenharia

Engenharia de Segurança

Ferramentas e técnicas sobre duplicação de código

Mariana Fernandes A81728
João Costeira A78073
Paulo Mendes A78203

Conteúdo

1 Introdução

Como sabemos, o custo de fazer alterações ou reparar defeitos no *software* é maior quanto mais avançada estiver a sua produção. Assim, é preciso perceber como minimizar estes custos, algo que pode ser feito atribuindo a devida importância, não só à qualidade dos requisitos e da arquitetura, como também à qualidade do próprio código.
Neste trabalho iremos definir os atributos de qualidade do *software*, explicar a importância de *code duplication* como métrica de qualidade do *software*, enunciar que programas existem que automatizem a procura de excertos de código duplicados, e dar alguns exemplos de funcionamento dos mesmos.

2 Atributos de Qualidade do *Software*

Em primeiro lugar, é útil entender de onde originam as métricas de medição da qualidade do código. Para tal, deve-se consultar o ISO 25010 que define os atributos possui um *software* de qualidade:

- **Adequação funcional:** Representa a capacidade do produto de *software* fornecer funcionalidades que atendem às necessidades declaradas e implícitas, quando o produto é usado sob as condições especificadas.
- **Eficiência de desempenho:** Esta característica representa o desempenho em relação à quantidade de recursos utilizados.
- **Compatibilidade:** Capacidade de dois ou mais sistemas, ou componentes, para trocar informações e/ou executar as funções necessárias quando partilham o mesmo ambiente de *hardware* ou *software*. Por um lado é a capacidade de coexistir com outro *software* independente, como de ter interoperabilidade com outros sistemas.
- **Usabilidade:** Capacidade de um produto de *software* ser entendido, aprendido, usado e atraente para o utilizador.
- **Confiabilidade:** Capacidade do sistema ou componente estar operacional e acessível para uso quando necessário, de operar como pretendido na presença de falha de *hardware* ou *software*, de recuperar os dados afetados e restaurar o estado desejado do sistema, em caso de interrupção ou falha.
- **Segurança:** Capacidade de proteger os dados para que pessoas ou sistemas não autorizados não possam lê-los ou modificá-los, ou seja, garantir a confidencialidade e integridade dos mesmos, bem como de garantir a autenticidade e o não-repúdio das ações efetuadas no sistema.

- **Manutenção:** Representa a capacidade do produto ser modificado de maneira eficaz e eficiente, devido a necessidades evolutivas, corretivas ou adaptativas.
- **Portabilidade:** Grau de eficácia e eficiência com o qual um sistema, produto ou componente pode ser transferido para outro *hardware* ou sistema operativo, ou seja, se se adapta a vários componentes de *hardware*, se é fácil de instalar e desinstalar ou de substituir.

No entanto, neste ISO, não está definida uma forma de medir esses atributos nem há um consenso sobre se algumas das métricas usadas realmente representam a qualidade do código. Por exemplo, pode ser difícil medir, e em particular, de forma automatizada, a facilidade de uso do *software*, pois está dependente não só dos seus componentes (Interface do utilizador) como do próprio utilizador a quem a aplicação se destina. De seguida, apresentamos uma lista das métricas mais usadas na indústria, e que podem ser automatizadas.

- Cobertura do código
- Interpretação abstrata
- Complexidade ciclomática
- Avisos do compilador
- *Standards* de codificação
- Duplicação de código
- Interdependência de módulos
- Segurança

Efetivamente, a duplicação de código surge como uma métrica importante, e passível de ser automatizada.

3 Duplicação de código

3.1 O que é?

Por vezes, encontramos partes de código que fazem algo igual ou parecido ao que pretendemos programar, e somos tentados a copiar esse código, ao invés de generalizar essa funcionalidade. No entanto, se o código precisar de ser alterado num sítio, é provável que requeira alterações nas suas cópias também.
Duas sequências de código não precisam de ser iguais caractere a caractere para serem consideradas como cópias, porque os espaços em branco, linhas em branco e comentários são ignorados.

Apesar de parecer ser um problema pouco importante, a repetição de código pode indicar uma falha no uso de mecanismos de abstração apropriados, como métodos, funções, subclasses e tipos genéricos. Consequentemente, o código é mais longo do que deveria ser, e é, portanto, mais susceptível a erros. Para além disso, instâncias repetidas de código dificultam a manutenção porque obrigam os programadores a encontrar e modificar manualmente cada cópia.
No entanto, o código nem sempre pode ser modificado para eliminar a duplicação de código. Os clones podem ser a solução mais eficaz se as abstrações forem de difícil implementação na linguagem de programação em uso.

3.2 Ferramentas para detetar duplicação de código

- **CPD** - *Copy Paste Detector*.
  Suporte para Java, C, C++, C#, Groovy, PHP, Ruby, Fortran, JavaScript, PLSQL, Apache Velocity, Scala, Objective C, Matlab, Python, Go, Swift, Salesforce.com Apex e Visualforce.
- **SonarQube** - Anáise estática de código: métricas e segurança.
  Suporte para Java, C#, C, C++, Javascript, Typescript, Python, Go, Swift, COBOL, Apex, PHP, Kotlin, Ruby, Scala, HTML5, CSS3, ABAP, Flex, Objective-C, PL/I, PL/SQL, RPG, T-SQL, Visual Basic, VB.NET e XML.
- **Simian** - Gratuito para uso pessoal e *open-source*.
  Suporte para Java, C#, C, C++, COBOL, Ruby, JSP, ASP, HTML, XML, Visual Basic, Groovy e ficheiros de texto limpo (*plain text files*).
- **Atomiq**
  Suporte para .NET.
- **ReSharper**
  Suporte para C#, VB.NET, XAML, JavaScript, TypeScript, JSON, XML, HTML, CSS, ASP.NET, ASP.NET MVC, Protobuf, NAnt e MSBuild scripts.
- **dupFinder** - via linha de comandos.
  Suporte para C# e Visual Basic .NET.

3.3 Exemplos práticos

Nota: Para os exemplos serem mais interessantes, poderiam ter aplicado as ferramentas a projetos open-source existentes no github

Simian

O *Simian* é uma ferramenta de identificação de código duplicado, capaz de actuar sobre diversas linguagens de programação, sendo até possível identificar duplicações em ficheiros contendo apenas texto simples. De modo a verificar as capacidades de detecção desta ferramenta elaboramos dois casos de teste.

Caso 1:

```java
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
        // this is a comment
        System.out.println("Hello, World!");
        System.out.println("Hello, World!");
        System.out.println("Hello, World!");
        // this is also a comment
        System.out.println("Hello, World!");
        System.out.println("Hello, World!");
    }
}
```

```
$java -jar ~/simian/bin/simian-2.5.10.jar -reportDuplicateText  test.java | tail -n +5
Found 0 duplicate lines in 0 blocks in 0 files
Processed a total of 8 significant (14 raw) lines in 1 files
Processing time: 0.081sec
$
```

Figura 1: *Output* do caso 1

Na figura 1 podemos ver que, para o caso 1, apenas usando as configurações *default* do *Simian*, este não é capaz de detectar qualquer duplicação de código.

```
$java -jar ~/simian/bin/simian-2.5.10.jar -reportDuplicateText -threshold=1  test.java | tail -n +5
Found 4 duplicate lines with fingerprint 335dfed1cf0cc6f3272e800ab2d2ab00 in the following files:
 Between lines 6 and 11 in /home/paulo/Desktop/test.java
 Between lines 4 and 10 in /home/paulo/Desktop/test.java
        System.out.println("Hello, World!");
        // this is a comment
        System.out.println("Hello, World!");
        System.out.println("Hello, World!");
        System.out.println("Hello, World!");
        // this is also a comment
        System.out.println("Hello, World!");
==========================================================================
Found 8 duplicate lines in 2 blocks in 1 files
Processed a total of 8 significant (14 raw) lines in 1 files
Processing time: 0.082sec
$
```

Figura 2: *Output* do caso 1 com *threshold* mais baixo

De modo a melhorar os resultados anteriores, se baixarmos o valor de *threshold* para 1 (originalmente estava a 6), vai reduzir o número mínimo de linhas necessárias para que seja feito um *match*. Esta alteração já vai permitir que sejam identificadas as duplicações presentes no caso 1.

Caso 2:

```java
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Goodbye, World!");
        for (int h = 0;h<7;h++) {
            // this is a comment
            System.out.println("Hello, World!");
        }
        System.out.println("Goodbye, World!");
        for (int i = 3;i<24;i++) {
            // this is also a comment
            System.out.println("Hello, World!");
            i+=2;
        }
        System.out.println("Goodbye, World!");
        for (int i = 7;i>0;i--) {
            System.out.println("Hello, World!");
        }
        System.out.println("Goodbye, World!");
    }
}
```

```
$java -jar ~/simian/bin/simian-2.5.10.jar -reportDuplicateText  test.java | tail -n +5
Found 0 duplicate lines in 0 blocks in 0 files
Processed a total of 8 significant (14 raw) lines in 1 files
Processing time: 0.081sec
$
```

Figura 3: *Output* do caso 2

De modo semelhante ao caso anterior, se executarmos o comando sem aumentarmos a sensibilidade do mecanismo de detecção, não vai ser possível obtermos os resultados desejados.

```
$java -jar ~/simian/bin/simian-2.5.10.jar -reportDuplicateText -threshold=1  test2.java | tail -n +5
Found 1 duplicate lines with fingerprint 18f0b1f483b2fdd0e8543b8bf3c99c95 in the following files:
 Between lines 23 and 26 in /home/paulo/Desktop/test2.java
 Between lines 9 and 12 in /home/paulo/Desktop/test2.java
            System.out.println("Hello, World!");
        }

        System.out.println("Goodbye, World!");
==========================================================================
Found 2 duplicate lines in 2 blocks in 1 files
Processed a total of 13 significant (29 raw) lines in 1 files
Processing time: 0.085sec
$
```

Figura 4: *Output* do caso 2 com *threshold* mais baixo

Novamente, se reduzirmos o valor de *threshold* vão ser detectadas algumas ocorrências de duplicação de código, apesar de neste caso ainda existir código duplicado que não está a ser identificado como tal.

```
$java -jar ~/simian/bin/simian-2.5.10.jar -threshold=1 -reportDuplicateText -ignoreVariableNames -ign
oreNumbers  test2.java | tail -n +5
Found 1 duplicate lines with fingerprint 06bd837dbf0ce5a85822656b86e444b1 in the following files:
 Between lines 23 and 26 in /home/paulo/Desktop/test2.java
 Between lines 9 and 12 in /home/paulo/Desktop/test2.java
            System.out.println("Hello, World!");
        }

        System.out.println("Goodbye, World!");
==========================================================================
Found 2 duplicate lines with fingerprint d63df04fbee80d2e274ce1e02cd6e34b in the following files:
 Between lines 12 and 16 in /home/paulo/Desktop/test2.java
 Between lines 5 and 9 in /home/paulo/Desktop/test2.java
        System.out.println("Goodbye, World!");

        for (int h = 0;h<7;h++) {
            // this is a comment
            System.out.println("Hello, World!");
==========================================================================
Found 6 duplicate lines in 4 blocks in 1 files
Processed a total of 13 significant (29 raw) lines in 1 files
Processing time: 0.105sec
```

Figura 5: *Output* do caso 2, ignorando nomes de variáveis e números

Finalmente, se ao comando anterior acrescentarmos as *flags* para que nomes de variáveis diferentes assim como números diferentes presentes no código de teste sejam ignorados, a ferramenta já vai ser capaz de detectar outros casos, que anteriormente lhe tinham escapado.

Sonarqube

O *sonarqube* é uma ferramenta bastante ampla que permite efectuar análise e testes de *software*, incluindo questões de segurança/manutenção, mas para o nosso caso em concreto, vamos concentrar sobre questões de repetição de código.

De seguida são descritas as ferramentas utilizadas de forma a implementar os testes semelhantes aos do grupo:

- Inicializar o servidor - Após a instalação do *sonarqube*, este servidor pode ser inicializado localmente com a execução do comando *./sonar.sh start*.
- Utilização de um *scanner* que permitirá analisar, por exemplo a *coverage* do código. O grupo decidiu utilizar o *SonarScanner*. De notar que esta ferramenta implica de configurações sobre o ficheiro *sonar-scanner.properties*.
- Executar testes sobre o código em análise. Em vez de efectuar a compilação clássica, foi utilizada a ferramenta*coverage.py* que permite compilar e gerar relatórios de *coverage* sobre o *software* em análise.
- Por fim é inicializado o *scanner (./sonar-scanner)*. Assim a análise de resultados pode ser observada localmente, por padrão na porta 9000 (http://localhost:9000/projects).

Exemplos

De forma exemplificativa, o grupo decidiu correr um código semelhante aos exemplos do *simian*, mas desta vez o código foi implementado na linguagem de programação *python*.
Complementarmente, foi implementado um código que efectua o calculo do factorial de um número de duas formas diferentes, de forma a efectuar testes de duplicação de *software*.

Análise do caso 1

O código do primeiro caso é extremamente simples: uma classe que possui um método que imprime a mesma *string* 6 vezes, ou seja, possui código repetido. Aqui encontra-se descrito o código fonte em análise:

```
1
2 class HelloWorld:
3
4     def hellos(self):
5         print("Hello, World!");
6         #this is a comment
7         print("Hello, World!");
```

```
8           print("Hello, World!");
9           print("Hello, World!");
10          #this is also a comment
11          print("Hello, World!");
12          print("Hello, World!");
13
14  def main():
15      h = HelloWorld()
16      h.hellos()
17
18  if __name__ == "__main__":
19      main()
```

Inicialmente foi efectuada uma análise sobre a *coverage* do código em análise. A seguinte imagem descreve os resultados, que como esperado, possui *coverage* de 100 por cento devido ao facto de total invocação do código.

```
❯ p caso1.py
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
❯ coverage run -m caso1
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
❯ coverage report -m
Name       Stmts   Miss  Cover   Missing
----------------------------------------
caso1.py      13      0   100%
```

Figura 6: *Coverage* do caso1

E como esperado, a ferramenta *sonarqube* foi capaz de identificar as 6 repetições existentes no código.

Figura 7: Análise da duplicação pelo *sonarqube* sobre *caso1.py*

Análise do caso 2

O segundo caso exemplificativo é extremamente semelhante ao código anterior, mas em vez de por cada *print* copiar a linha de código, recorre-se a ciclos para reduzir o número de repetições.
De seguida encontra-se uma descrição do código em análise.

```
1
2  class HelloWorld:
3
4      def hellos(self):
5          print("Goodbye, World!")
6          for _ in range(0,7):
7              #this is a comment
8              print("Hello, World!")
9
10         print("Goodbye, World!")
11
12         for _ in range(3,24,2):
13             #this is also a comment
14             print("Hello, World!")
15
16         print("Goodbye, World!")
17
18         for _ in range(7,0,-1):
19             print("Hello, World!")
20
21         print("Goodbye, World!")
22
23
24 def main():
```

```
25      h = HelloWorld()
26      h.hellos()
27
28  if __name__ == "__main__":
29      main()
```

Inicialmente foi efectuada a análise sobre a *coverage* do código, que como esperado, foi de 100 por cento devido à invocação total do código:

```
❯ coverage run -m caso2
Goodbye, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Goodbye, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Goodbye, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Goodbye, World!
❯ coverage report -m
Name       Stmts   Miss  Cover   Missing
----------------------------------------
caso2.py      17      0   100%
```

Figura 8: *Coverage* do caso2

Em relação aos resultados obtidos sobre a duplicação do código, o *sonarqube* foi capaz de identificar que os padrões *"print("Hello, World!")"* e *"print("Goodbye, World!")"* aparecem múltiplas vezes no ficheiro, por isso sugere a sua substituição por uma constante.
A seguinte imagem exemplifica o resultado obtido:

Figura 9: Análise da duplicação pelo *sonarqube* sobre *caso2.py*

Análise do caso 3

Nos exemplos anteriores conseguimos analisar a repetição literal de *código*, mas neste terceiro exemplo o grupo possui como objectivo realizar um teste diferente, testar se o sistema é capaz de identificar funções que apesar de não terem o mesmo código, estas produzem o mesmo resultado perante o mesmo *input*.

O exemplo escolhido foi a função factorial, onde possuímos duas versões diferentes da mesma função.

```
def fact1(n):
    assert n >= 0

    f=1
    for i in range(1,n+1):
        f = f * i
    return f
```

```
11 def fatorial(n):
12
13     assert n >= 0
14
15     if(n>0):
16         f = n*fatorial(n-1)
17     else:
18         f = 1
19     return f
20
21 def main():
22     for i in range(0,20):
23         print('Fatorial de '+str(i) +' e ' + str(fact1(i)) + ', e
       tambem deve ser '+ str(fatorial(i)))
24
25 if __name__ == "__main__":
26     main()
```

Em relação ao *coverage*, não existe nehuma surpresa sobre os resultados obtidos, todo o código encontra-se executado e testado:

```
__pycache__ fatorial.py
❯ coverage report -m
Name          Stmts   Miss  Cover   Missing
-----------------------------------------
fatorial.py      17      0   100%
~/Desktop/seguranca_tp2/projeto/codigo
```

Figura 10: *Coverage* do factoria

Em relação aos resultados, o *sonarqube* não foi capaz de identificar nenhuma repetição de código na aplicação.

De seguida, o código do factorial foi alterado de forma a possuir repetições de código, mais precisamente a função *fact1* e *fact2* possuem exactamente o mesmo código.

A seguinte imagem possui as alterações efectuadas sobre o código e os resultados obtidos:

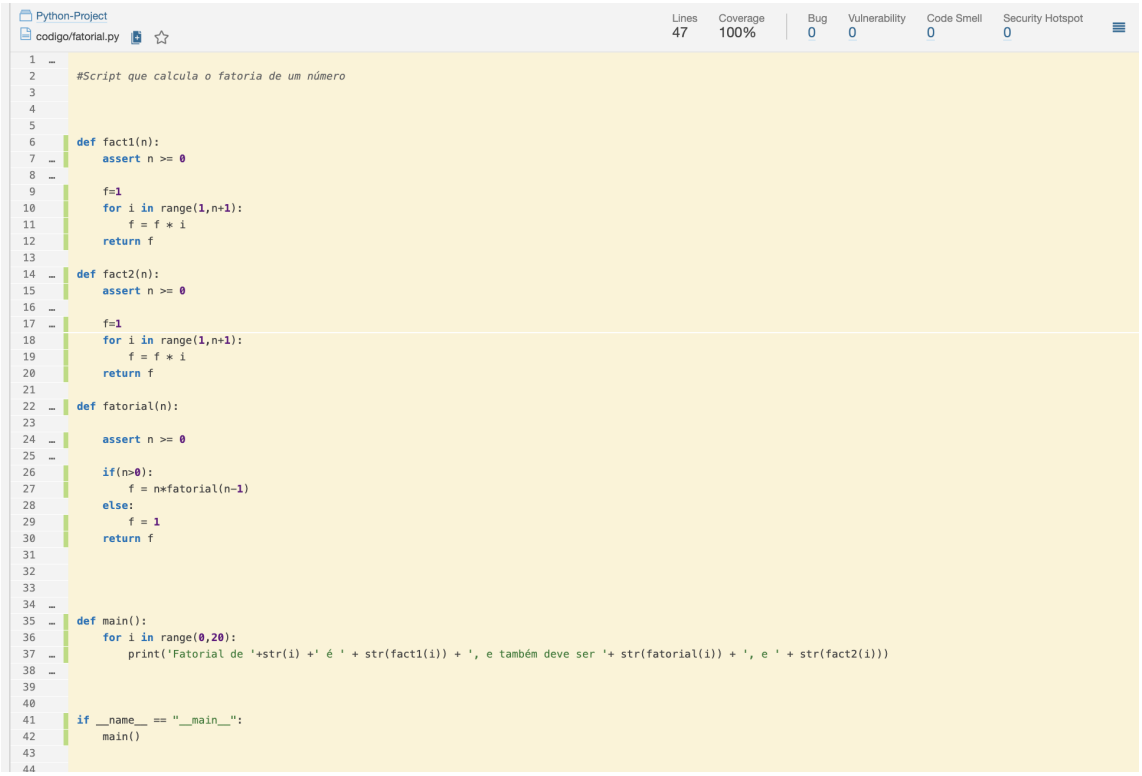

Figura 11: Análise da duplicação sobre o ficheiro de fatorial

Este resultado foi inesperado, pois apesar de existir repetições claras no código, o sistema não foi capaz de as identificar, classificando este código como perfeito em relação à *coverage* e duplicações.
Então o grupo decidiu efectuar uma terceira alteração sobre o programa factorial, simplesmente criar uma função que ocupa linhas no código fonte, no nosso caso especifico uma função com 78 linhas onde em cada linha é efectuado um *print* de um número.
A seguinte imagem descreve os resultados obtidos:

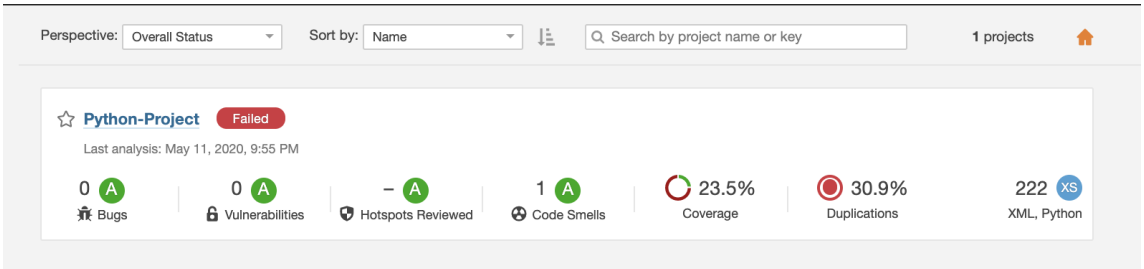

Figura 12: Análise da duplicação sobre o ficheiro de factorial

Como podemos observar, o sistema já é capaz de identificar repetições de código. Após uma pesquisa sobre a classificação de código repetido, encontramos a seguinte informação:

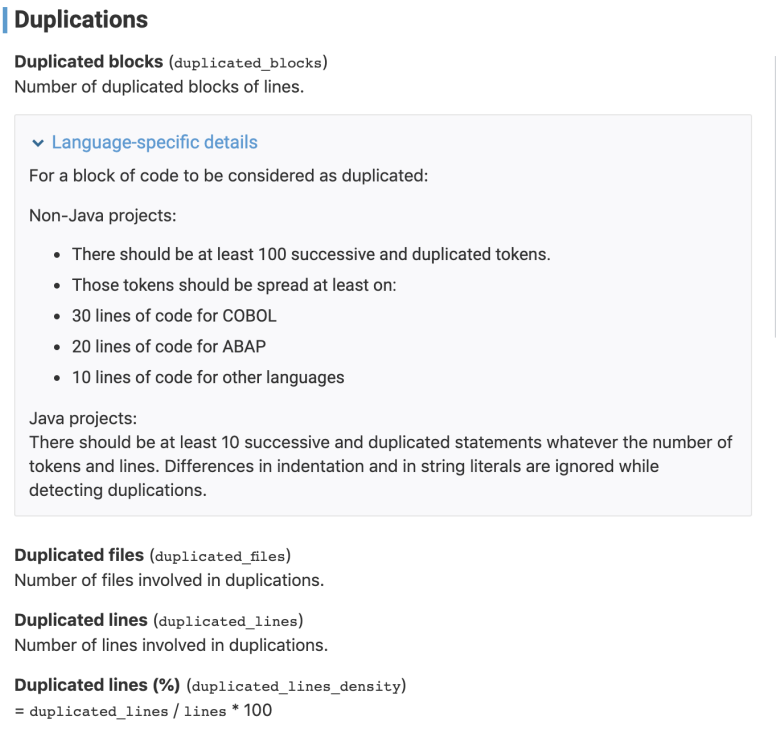

Figura 13: Análise da duplicação sobre o ficheiro de factorial

O grupo acredita que este é o motivo pelas falhas na classificação de repetições anteriormente encontradas. Pois como a função *fact1* e *fact2* encontravam-se sequencialmente definidas no código fonte, uma depois da outra, a sua proximidade

impedia o sistema de as classificar como repetição de código.

4 Conclusão

Como pudemos compreender ao longo deste relatório, a duplicação de código tem um forte impacto negativo sobre a qualidade do mesmo, e pode até revelar que uma peça de *software* foi elaborada sem ter na sua base boas práticas de codificação.
Assim sendo, a eliminação de clones ou duplicados é uma boa forma de aumentar a facilidade de manutenção e a segurança do *software*.
Para tal, podemos tirar partido de várias ferramentas já existentes, como as listadas na secção 3.2. A escolha do *programa* a utilizar dependerá da linguagem na qual se está a codificar o projeto, e no tipo de licença pretendida, porque apesar de haver licenças gratuitas, estas não podem ser usadas em contexto profissional.
Finalmente, demonstramos o uso destas ferramentas recorrendo a exemplos práticos, recomendando o uso do `Simian`, por suportar um vasto número de linguagens usadas comummente, por suportar também ficheiros de texto simples, por ter bom desempenho segundo os testes que efetuamos, e por ser fácil de instalar e usar.
Em suma, a duplicação de código é uma boa métrica a ter em conta na análise da qualidade do *software*, pela sua relevância, quer pela possibilidade de automatização da medição da mesma.

5 Bibliografia

- https://iso25000.com/index.php/normas-iso-25000/iso-25010
- https://www.tiobe.com/files/TIOBEQualityIndicator_v4_3.pdf
- https://en.wikipedia.org/wiki/Duplicate_code
- https://www.informit.com/articles/article.aspx?p=457502&seqNum=5
- https://pmd.github.io/
- https://www.sonarqube.org/features/quality-gate/
- http://www.harukizaemon.com/simian/?fbclid=IwAR1jW1n_b8RANK8Qfcqs_fVgkP_P8HMy
- http://www.getatomiq.com/
- https://www.jetbrains.com/help/resharper/dupFinder.html
- https://www.sonarqube.org/

- https://docs.sonarqube.org/latest/analysis/scan/sonarscanner/
- https://coverage.readthedocs.io/en/coverage-5.1/
- https://docs.sonarqube.org/latest/user-guide/metric-definitions/MetricDefinitions-Duplications