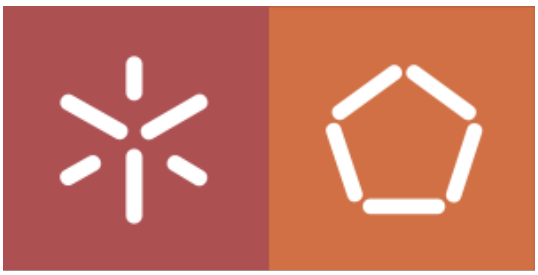

Universidade do Minho
Escola de Engenharia

Universidade do Minho
Mestrado Integrado em Engenharia Informática
Engenharia de Segurança
Projeto 2 - Cyclomatic Complexity

Diogo Duarte
(pg41843@alunos.uminho.pt)
Mateus da Silva Ferreira
(pg37159@alunos.uminho.pt)
Ricardo Cunha Dias
(pg39295@alunos.uminho.pt)

11 de Maio de 2020

Conteúdo

1. Introdução

Este projecto foi desenvolvido no âmbito da Unidade Curricular Engenharia de Software e tem como objectivo elaborar uma investigação sobre ferramentas e técnicas da métrica de qualidade de software complexidade ciclomática.

Assim, este relatório visa os resultados dessa mesma investigação realizada, explicando o que é essa métrica, para que serve e quais as principais ferramentas utilizadas para medi-la, assim como alguns exemplos de sua utilização.

A complexidade ciclomática trata-se de um exemplo de uma métrica de software e nos dias de hoje estas medições são fundamentais para o desenvolvimento de software com qualidade. Deste modo, estas métricas permitem-nos compreender e tomar decisões com base em factos, prever condições para desenvolvimentos futuros e estimar o custo e o tempo de desenvolvimento.

2. *Cyclomatic complexity*

Desenvolvida por Thomas J. McCabe em 1976, a complexidade ciclomática ou *cyclomatic complexity* é uma das mais antigas métricas para calcular a complexidade de um software.

Trata-se, provavelmente, da métrica complexa mais usada em engenharia de software e continua a ser um tema recorrente de discussão e, ainda assim, é ignorada por muitos desenvolvedores, coordenadores e gestores de empresas de software.

A complexidade ciclomática é utilizada para analisar a complexidade de um código fonte de um programa e calcular o número de caminhos de execução independente pelo código. Esta complexidade pode ser computada utilizando o grafo de controlo de fluxo do programa, onde os nós do grafo correspondem a grupos indivisíveis de comandos e uma aresta direccionada conecta dois nós apenas se o segundo comando puder ser executado imediatamente após o primeiro.

Um caminho de execução independente consiste naquele que apresenta pelo menos uma nova condição (possibilidade de desvio de fluxo) ou um novo conjunto de comandos a serem executados. Há medida que o número de caminhos aumenta, a complexidade ciclomática também amplia. Assim, sempre que o fluxo de controlo de uma função é dividido, o contador de complexidade incrementa. Cada função tem uma complexidade mínima de 1, variando conforme a linguagem utilizada.

3. Como calcular a complexidade ciclomática de um código

O valor da complexidade ciclomática resulta na contagem do número de testes que, pelo menos, precisam ser executados para que todos os possíveis fluxos que o código possa tomar sejam verificados, a fim de assegurar uma cobertura completa de testes.

Para calcular a complexidade ciclomática de um código, McCabe utiliza, primeiramente, grafos de fluxo de controlo com o objectivo de descrever as estruturas lógicas dos módulos do programa. Um módulo corresponde a uma função ou sub-rotina que tem um ponto de entrada e um ponto de saída. Os grafos são direcionados, onde cada nó representa um bloco de comando do código em análise e as arestas que os conectam representam os caminhos que os fluxos de execução podem seguir. Cada caminho de execução possível de um módulo do código tem um caminho correspondente do nó de entrada ao nó de saída do grafo de fluxo de controlo do módulo. Essa correspondência é a base da metodologia de teste estruturado.

Uma forma de calcular a complexidade ciclomática de um código é através da fórmula:

$$M = E - N + 2 \tag{3.1}$$

Onde:

E = o número de arestas do grafo

N = o número de nós do grafo

Para exemplificar o cálculo da complexidade ciclomática, foi escrita a função básica abaixo em *Python*, que recebe uma idade como parâmetro e verifica se o valor inserido é maior ou igual a 18, retornando *True* e retornando *False* caso seja menor que 18.

```
1 def idade(idade):
2 if idade >= 18:
3     return True
4 else:
5     return False
```

O grafo de fluxo de controle a seguir descreve a estrutura lógica da função acima.

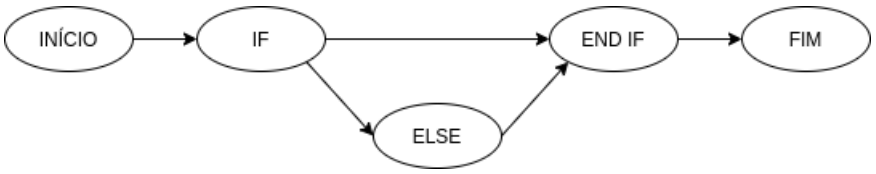

Figura 3.1: Grafo de fluxo de controle da função idade

Utilizando a fórmula citada anteriormente neste exemplo, onde o número de nós é igual a 5, o número de aresta também é 5, então: **M = 5 - 5 + 2**, resultando em uma complexidade ciclomática de 2.

Foi implementado outro exemplo para ilustrar a complexidade ciclomática envolvendo outros pontos de decisão além do *if*.

```
def euclid(m, n):
    """
    Algoritmo de Euclides
    Assumindo que m e n s o maiores que 0,
    retorna o M ximo Divisor Comum entre eles.
    :param m:
        N mero inteiro maior que 0
    :param n:
        N mero inteiro maior que 0
    :return:
        M ximo divisor comum entre m e n
    """
    if n > m:
        m, n = n, m
    r = m % n

    while r != 0:
        m = n
        n = r
        r = m % n
    return n
```

O grafo de fluxo de controle abaixo foi criado com base no função acima para auxiliar na realização do cálculo da complexidade ciclomática. Aplicando a fórmula, **M = 13 - 12 + 2**, conclui-se que a complexidade dessa função é 3, comprovando que quanto mais pontos de decisão são inseridos em um código, maior é a complexidade ciclomática.

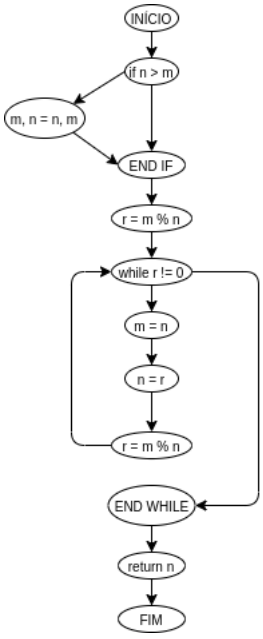

Figura 3.2: Grafo de fluxo de controle do algoritmo de Euclides

3.1 Ferramentas para medição da complexidade ciclomática

3.1.1 TICSpp e TICSsql

TICSpp é uma ferramenta para medir a complexidade ciclomática, que suporta uma grande variedade de linguagens de programação diferentes como JavaScript, Python, Scala, LUA, Java, C, C ++ e outra. Esta ferramenta tem suporte em *Windows*, *Linux* e *Solaris SPARC* e é proprietária da *TIOBE*.

TICSpp permite, verificar outras métricas de qualidade de software, porém o nosso foco centra-se na complexidade cilcomática, e como esta métrica é um pouco abstracta, esta ferramenta interpreta esta métrica da seguinte maneira.

O número de decisões é incrementado, sempre que encontra *if-statement*, *while-statement* e tratamento de expções *try-catch statement*. Como citado anteriormente, a complexidade ciclomática de uma função é o número de decisões que encontra nesta, onde 1 é o valor mínimo. Quanto à complexidade ciclomática de um ficheiro, é o número de decisões encontradas mais um, ou seja, é a soma das complexidades ciclomáticas das funções encontradas no ficheiro menos o número de funções mais um.
A média da complexidade ciclomática de um ficheiro, é dada pela soma das complexidades ciclomáticas das funções a dividir pelo número de funções no ficheiro.
A média da complexidade ciclomática do produto, é a soma das complexidades ciclomáticas dos ficheiros que fazem parte deste menos o número de ficheiros, divido pelo total número total de funções no produto(ou seja a soma de todas as funções de todos os ficheiros) mais um.

Quanto à avaliação desta métrica(*score*) a ferramenta calcula-a através da seguinte formula:

$$score = 800/(cyclomatic\_complexity^2 – 3 * cyclomatic\_complexity + 10)$$

De realçar que esta *cyclomatic complexity* é o resultado das funções de média, o que no ponto de vista do produto, será média da complexidade ciclomática do produto, que foi mencionada anteriormente.

Cyclomatic Complexity	TQI Score	Category
<= 2.57	>= 90%	A
<= 3.00	>= 80%	B
<= 3.42	>= 70%	C
<= 4.37	>= 50%	D
<= 5.00	>= 40%	E
> 5.00	< 40%	F

Figura 3.3: Score da complexidade ciclomática

3.1.2 Mlint e Checkcode

De acordo com o documento utilizado como base para este relatório, a linguagem de programação MATLAB, da empresa MathWorks, utiliza uma ferramenta, que é integrada no pacote do software, chamada "mlint". Porém, segundo a documentação mais recente do MATLAB, esta ferramenta já não é recomendada e agora deve-se usar a ferramenta "checkcode", que possui as mesmas funcionalidades. O "checkcode"foi introduzido no MATLAB na versão R2011b e é uma função para analisar possíveis problemas em ficheiros de códigos.

Para calcular a complexidade ciclomática de cada função de um ficheiro de código MATLAB, deve-se passar a opção '-cyc' como parâmetro para o "checkcode", na forma: `checkcode('filename', 'cyc')`.

3.1.3 SonarQuebe

O *SonarQuebe* consiste numa plataforma *open-source* desenvolvida pela *SonarSource* que efetua uma inspeção da qualidade do código produzido, executando revisões automáticas analisando estaticamente o código. Esta ferramenta permite-nos detetar *bugs*, *smells* e vulnerabilidades de segurança do código e está disponível em 27 linguagens de programação, sendo que em algumas destas é necessário uma licença comercial. Está integrado nos ambientes de desenvolvimento Eclipse, Visual Studio e IntelliJ IDEA através de *plugins* do SonarLint e está também incorporado em ferramentas externas como LDPA, Active Directory e GitHub.

O SonarQuebe oferece relatórios específicos de código duplicado, testes unitários, padrões de codificação, **complexidade ciclomática**, comentários, bugs e vulnerabilidades de segurança. Assim, esta plataforma fornece análises automatizadas com Maven, Ant, Gradle, MSBuild e ferramentas de integração contínua, como por exemplo Atlassian Bamboo e Jenkins.

3.1.4 Eclipse Metrics Plugin

O Eclipse Metrics Plugin é open-source e é facilmente instalado através do Eclipse Marketpalce.

Este plugin calcula diversas métricas no código durante a criação e alerta através de exibição de problemas para cada métrica, permitindo que o utilizador fique constantemente ciente da integridade do código base. Através da opção Coverage As são executados um conjunto de testes unitários onde é exibida uma lista de todas as classes e métodos da aplicação, juntamente com o cálculo da complexidade ciclomática para cada um. Esta opção computa quais os caminhos que foram abordados durante os testes e fornece uma pontuação percentual de cobertura para cada classe e método na aplicação.

3.1.5 JaCoCo Jenkins pipeline plugin

O JaCoCo é uma ferramenta de análise de *code coverage* para Java. O plugin JaCoCo Jenkins pipeline desenvolvido para o Jenkins inspeciona os resultados de vários testes JUnit executados durante a fase de testes e gera um relatório de *code coverage* utilizando o JaCoCo. Outro recurso interessante deste plugin é que ele pode ser configurado com *thresholds*, de forma que, se a métrica de complexidade ciclomática exceder 15 para qualquer método do código, o plugin indicará uma falha na compilação. Além disso, o plugin JaCoCo Jenkins também pode forçar a compilação a falhar se a porcentagem de caminhos - conforme estabelecido pela métrica de complexidade ciclomática - não for alcançada durante o teste. Portanto, ele não fornece apenas um relatório completo, mas também garante que o código não testado não entre em produção.

3.1.6 *Kuscos Application*

Kuscos, é um software de análise de aplicações e é desenvolvido pela *morphis*. Permite analisar várias métricas de software, em particular a complexidade ciclomática do código fonte. Este software suporta diferentes tipos de linguagens como java, javascript, Cobol, C, ASP.NET, VB.ET, SQL entre outros. A forma de calcular esta métrica, é igual à anteriormente descrita neste documento, e conforme o seu resultado a avaliação desta métrica é dada através do seguinte gráfico

CC	*Evaluation*	OO Languages
1-10	Simple, low complexity	Good
11-20	Moderate risk	Bad
21-50	High risk	Too complex
> 50	Not testable	Too complex

Figura 3.4: Avaliação da complexidade ciclomática

3.1.7 *Lizard*

O *Lizard* é uma ferramenta escrita em Python, também open-source para análise de complexidade ciclomática. Suporta uma extensa lista de linguagens de programação como: C/C++, Java, C, JavaScript, Objective-C, Swift, Python, Ruby, TTCN-3, PHP, Scala, GDScript, Golang, Lua e Rust. Além da complexidade ciclomática chamada pela ferramenta de CCN (*Cyclomatic Complexity Number*), o *Lizard* também calcula o número de linhas do código sem comentários (nloc), a contagem de *tokens* de funções (e a contagem de parâmetros de funções).

Esta ferramenta funciona através da linha de comandos e possui também uma plataforma de teste online. Um exemplo da sua utilização pode ser encontrado no capítulo seguinte.

4. Exemplo de utilização de algumas ferramentas

Neste capitulo decidimos explorar algumas das ferramentas mencionadas anteriormente. Uma das ferramentas escolhidas foi *TICSpp* e neste caso esta plataforma apenas disponibiliza uma *live-demo* para os utilizadores poderem experimentar antes de subscreverem à aplicação, tal como podemos observar na imagem seguinte.

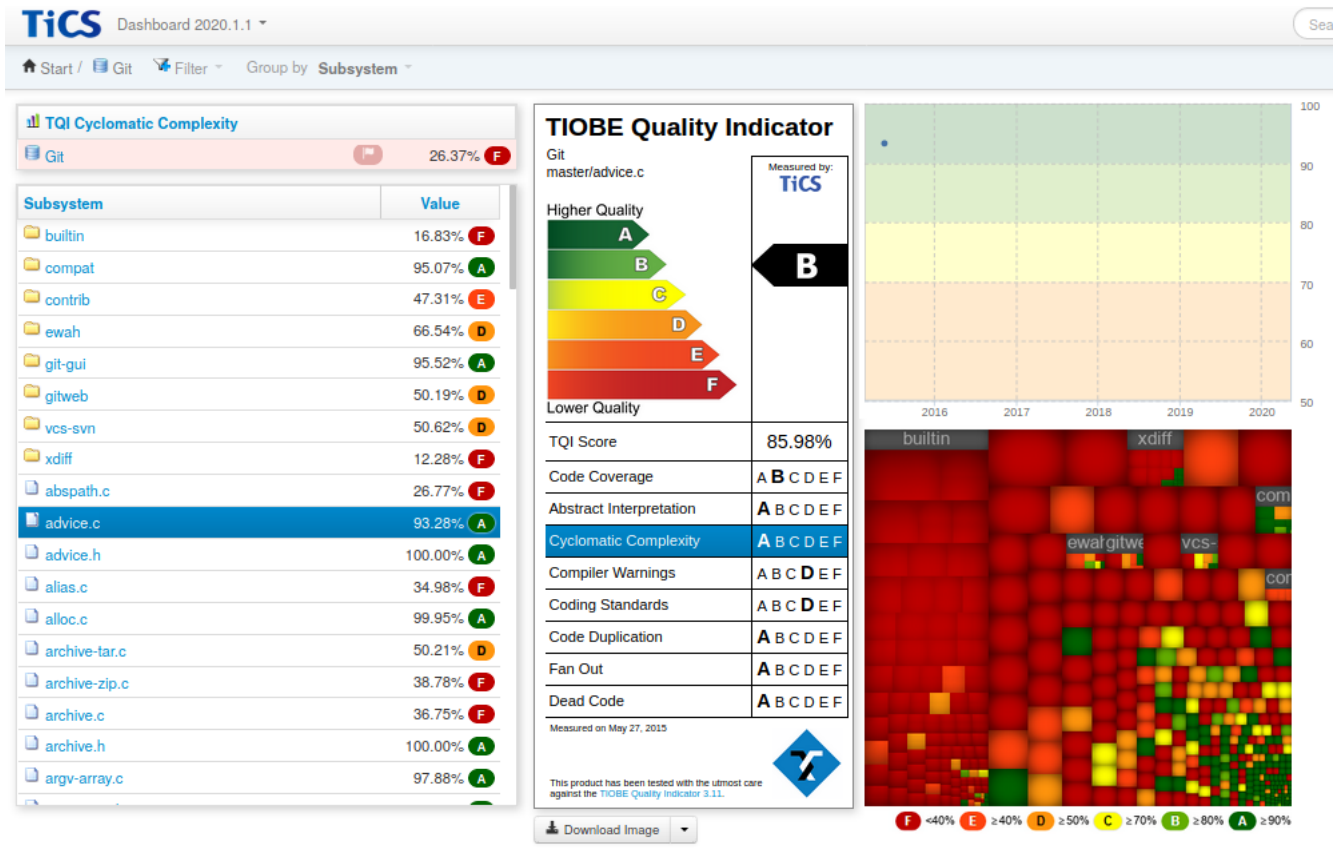

Figura 4.1:

Podemos verificar o *score* da complexidade ciclomática apresentado no lado direito de cada ficheiro. Na tabela é nos apresentado a avaliação das outras métricas, e no canto inferior direito, os scores dos diferentes ficheiros e pastas dentro desta aplicação, Git, que é fornecidas pela *live-demo*.

Outra ferramenta escolhida para a realização de testes foi a *Lizard*, que funciona via CLI. Para efetuar o teste, foi escolhido um dos ficheiros do código fonte do jogo DOOM (wadread.c), onde foi efetuado o comando abaixo e obteve-se o resultado a seguinte resultado:

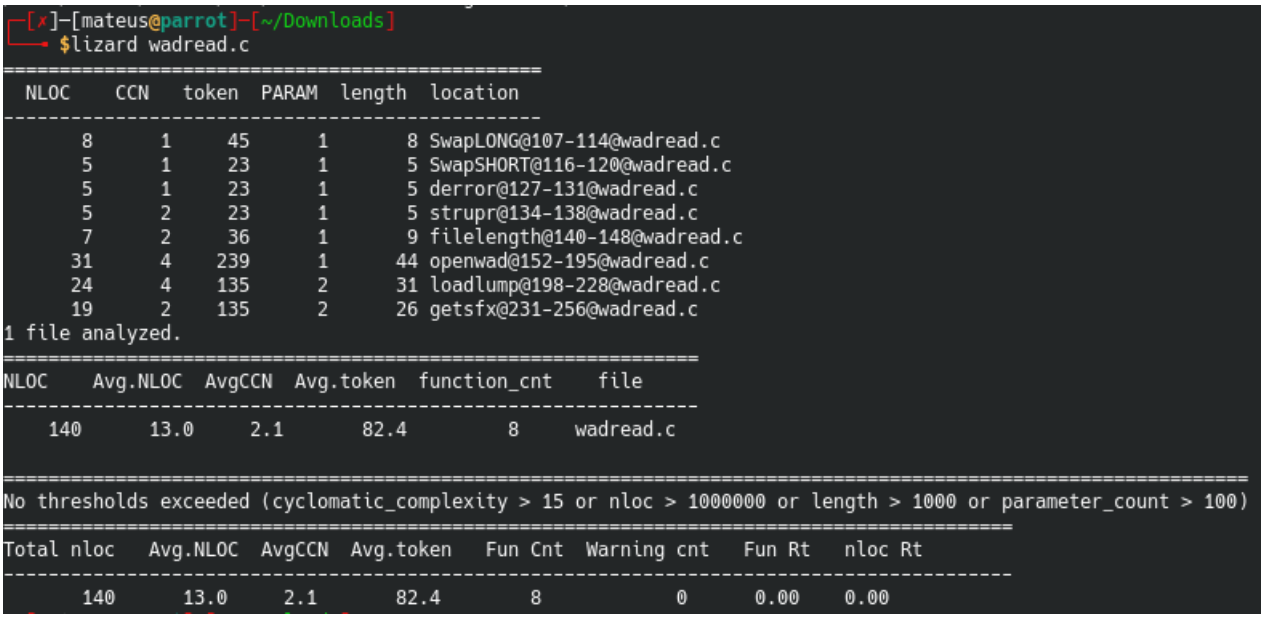

Figura 4.2: Cálculos do *Lizard* para o código "wadread.c"@

Segundo o resultado gerado pela ferramenta *Lizard* apresentado na figura acima, o código "wadread.c"possui uma complexidade ciclomática média de 7.1 e não excede o limite de 15 estabelecido pela ferramenta como o máximo aceitável.

5. Como melhorar a complexidade ciclomática de um código

Normalmente, a quantidade de código presente num programa depende das funcionalidades que um desenvolvedor pretende implementar e geralmente o software é desnecessariamente complexo. A complexidade desnecessária é um entrave para testes eficazes por três razões:

- a lógica extra requer mais testes
- os testes para esta complexidade extra requerem um esforço extra para executar cada um
- por vezes pode ser impossível testar a lógica extra devido às dependências de controlo de fluxo do software.

A complexidade desnecessária também pode indicar que o desenvolvedor original não entendeu o software o que pode provocar dificuldades na manuntenção e erros definitivos. Neste capitulo iremos abordar a complexidade desnecessária e apresentaremos técnicas para removê-la e testá-la.

5.1 Complexidade atual e Complexidade realizável

O tipo mais inofensivo de complexidade extra requer apenas mais testes que o normal, aumentando o esforço deste e escondendo a funcionalidade do software. Um tipo mais problemático de complexidade desnecessária impede testes estruturados de serem totalmente satisfeitos, pois algumas dependências dos dados podem impedir que um caminho no CFG seja percorrido.

A complexidade atual, ac, de um módulo é definida como o número de caminhos linearmente independentes que foram executados durante o teste. O critério de um teste estruturado requer que a complexidade atual seja igual à complexidade ciclomática. De referir, que a complexidade atual é uma propriedade do módulo e do teste e que aumenta a cada novo teste independente.

A complexidade realizável é a complexidade real máxima possível, ou seja, a contagem do conjunto de caminhos induzidos por todos os testes possíveis. O conceito de complexidade realizável é semelhante ao de complexidade ciclomática, exceto que alguns caminhos podem não ser abordados por algum teste.

Uma propriedade de rc é que, após uma quantidade de testes terem sido realizados num módulo, ac $\leq$ rc $\leq$ cc. No entanto quando ac $<$ cc, uma de duas situações podem ocorrer:

- pelo menos um teste independente tem de ser executado
- ac = rc, e portanto, rc $<$ cc.

Para o primeiro caso, a solução é bastante simples já que basta continuar a testar até que ac = cc ou até que o caso 2 seja alcançado.

No segundo caso, o software pode ser reestruturado para remover a complexidade desnecessária, tentando alcançar rc = cc. Em alternativa, podemos utilizar uma ferramenta que relata o conjunto de dependências de controlo do software e ajuda o desenvolvedor a decidir se a execução de mais testes adicionais pode aumentar a ac.

5.2 Remover dependências de controlo

A remoção de dependências de controlo geralmente permite uma melhoria do código, pois os módulos resultantes tendem a ser menos complexos e com uma lógica de decisão direta. Duas técnicas para as retirar são a simplificação lógica e direta e a modularização.

5.3 *Trade-offs* ao reduzir a complexidade

Ao remover o controlo de depêndencias, normalmente leva a uma melhoria no software, porém por vezes isto leva um impacto negativo no software. Um impacto negativo comum ao reduzir a complexidade é introdução de lógica não estruturada no código e pode ser resolvido através de Complexidade essencial, quantificando a lógica não estruturada, porém isto causa uma degradação estrutural. Com isto à medida que os programas são mais complexos, a degradação estrutural tende a aumentar e nesses casos, a versão original pode ser preferível. Existe também o termo de *"middle ground"* que consiste num programador avaliar a necessidade de reestruturar o programa, dependendo das suas preferências e experiência e nestes casos é mais importante documentar as razões para o *trade-off* do que reduzir a complexidade.

6. Aplicações

6.1 Limitar a complexidade durante o desenvolvimento

Uma das aplicações proporcionadas por McCabe consistia na limitação das rotinas durante o desenvolvimento do programa. McCabe recomendou que os programadores calculassem a complexidade dos módulos e os dividissem em menores sempre que este cálculo fosse superior a dez. Esta aplicação foi adotada pela metodologia NIST Structured Testing com algumas nuances, referindo que em algumas circunstâncias esta divisão dos módulos não se deveria efetuar. Assim, o cálculo poderia superar o recomendado por McCabe desde que esse valor fosse acordado previamente e fosse devidamente explicado por escrito.

6.2 Medir a "estrutura"de um programa

McCabe sugeriu uma medida numérica, nomeada de complexidade essencial, para calcular o quão próximo um programa está da programação estrutural ideal. Esta medida baseava-se nos grafos de controlo de fluxo (CFG) e nos subgráficos de programas não estruturados. Para calcular esta medida, os CFG original é reduzido iterativamente, identificando sub-gráficos substituindo-os por um unico nó. Esta medida foi mais tarde chamada de condensação. Caso um programa seja estruturado, o processo de condensação tornaria o gráfico num único nó.

6.3 Implicações para testes de software

Outra aplicação consiste na determinação do número de testes de caso necessários para obter a cobertura completa de um módulo específico. O cálculo da complexidade ciclomática tem de estar compreendido entre o número de testes necessários para obter uma cobertura completa do módulo e o número de caminhos possíveis conforme o CFG.

6.4 Correlação com o número de defeitos

Vários estudos apontam para uma relação entre o número de complexidade ciclomática de McCabe e o número de defeitos presentes num programa. Assim, programas com um maior número de imperfeições apresentam um maior número de complexidade ciclomática. Outros estudos revelaram que a complexidade ciclomática pode também estar relacionada com o tamanho do programa. Embora a relação possa ser verdadeira, esta não é conclusiva pois o tamanho do programa não é totalmente controlável.

7. Conclusão

Após o término deste projeto percebemos a importância de utilizar as métricas no desenvolvimento de software permitindo uma melhor gestão de projetos apesar de haver bastantes discordâncias sobre como medir e avaliar o resultado destas medições.

Deste modo, a complexidade ciclomática não é apenas útil para obter a complexidade de um programa, mas também permite ao utilizador definir a cobertura de testes de um programa, ou seja, o número de casos de teste.

A qualidade de um produto de software é muitas vezes avaliada pelo o número de funcionalidades presentes, o que provoca um incremento no número de linhas de código no programa. Vários estudos apontam que a complexidade ciclomática aumenta caso haja um elevado número de linhas de código presentes.

Com o aumento do número de linhas de código o número de bugs também aumenta, pois estima-se que por cada mil linhas encontramos entre cinco a cinquenta bugs, onde alguns destes são vulnerabilidades. Assim, complexidade ciclomática apresenta-se também como um dos fatores importantes que influenciam a segurança de software.

8. Bibliografia

Arthur H. Watson, A., H., McCabe, T., J., (1996). Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. NIST Special Publication 500-235. Consultado em Abril 15, 2020, em http://www.mccabe.com/pdf/mccabe-nist235r.pdf.

Wikipedia contributors. (2020). Cyclomatic complexity. Consultado em Abril, 5, 2020, em https://en.wikipedia.org/w/index.php?title=Cyclomatic_complexityoldid=947648399.

Jansen, P., (2018). The TIOBE Quality Indicator: a pragmatic way of measuring code quality. Consultado em Abril 5, 2020, em https://www.tiobe.com/files/TIOBEQualityIndicator_v4_3.pdf.

MCCABE, T. J. A., (1976). A complexity Measure. In: IEEE Trans. Software Eng. Vol. SE-2, N. 4., p. 308-320, consultado em Abril 20, 2020, em https://ieeexplore.ieee.org/document/1702388.

McKenzie, C., (2018). The 4 essential Java cyclomatic complexity testing tools. Consultado em Maio 3, 2020, em https://www.theserverside.com/feature/The-4-essential-Java-cyclomatic-complexity-testing-tools.

Yin, T., (2008). Lizard (versão 1.17.3) [*code analyzer*]. Singapore. Consultado em Maio 9, 2020, em https://github.com/terryyin/lizard.

Saraiva, J., (2019). Software Metrics. Consultado em Abril 25, 2020, em https://drive.google.com/file/d/1rHhXKchHKSc5d4n2NaKaWq1Zx6XMYZIu/view.