UMA ABORDAGEM PARA OTIMIZAÇÃO DA QUALIDADE DE CÓDIGO FONTE BASEADO NA COMPLEXIDADE ESTRUTURAL

Heleno de Souza Campos Junior[1], Luís Rogério Ventura Martins Filho[2], Marco Antônio Pereira Araújo[3]

Resumo: Estudantes de programação desenvolvem códigos fonte com condições redundantes que acabam ocasionando uma complexidade estrutural desnecessária. Nesse cenário, busca-se evitar que isso aconteça. Dessa forma, este trabalho tem como objetivo apresentar uma abordagem capaz de identificar a presença de complexidade ciclomática desnecessária em um código fonte. A abordagem foi implementada como uma nova funcionalidade em uma ferramenta que apoia a criação de casos de teste de unidade para códigos fonte escritos em linguagem Java. A utilização dessa funcionalidade pode levar o usuário a simplificar seus algoritmos e despender menos esforço durante a fase de testes de unidade, impactando em uma melhor qualidade final do software desenvolvido.

Palavras-chave: Qualidade de software, teste de software, complexidade desnecessária, complexidade ciclomática, melhoria do código fonte.

INTRODUÇÃO

O desenvolvimento de aplicações de software envolve a criação de códigos fonte com diferentes finalidades. O código fonte desenvolvido pode apresentar alta complexidade. Segundo YU e ZHOU (2010), um código fonte complexo é difícil de se entender, manter e testar, podendo impactar em uma menor qualidade do software.

Uma forma de se medir a complexidade de um programa, ou unidade de código fonte, é através da complexidade ciclomática de McCABE (1976). Essa métrica é baseada na análise da estrutura do fluxo do programa analisado. Quanto mais complexo o fluxo, mais complexo é o código fonte. Para analisar a estrutura do fluxo, é utilizado o conceito apresentado por ALLEN (1970) de Grafos de Fluxo de Controle (GFC). A ideia é representar o programa analisado utilizando um grafo, onde cada vértice representa um determinado conjunto de instruções e as arestas que os conectam representam a passagem de controle entre essas instruções. Dessa forma, uma instrução condicional, por exemplo, possui duas arestas, ou caminhos possíveis a se seguir; um quando a condição é verdadeira e outro quando é falsa. O valor da complexidade ciclomática é calculado a partir da quantidade de caminhos únicos existentes nesse grafo, também chamados de caminhos independentes.

Um fenômeno que pode se manifestar no desenvolvimento de software é a presença da complexidade desnecessária, ou seja, o código fonte pode apresentar

1 Bolsista PIBICTI IF Sudeste MG - Campus Juiz de Fora/UFJF PGCC - camposheleno0@gmail.com

2 Bolsista PIBICTI IF Sudeste MG - Campus Juiz de Fora - luisrogeriojf@gmail.com

3 IF Sudeste MG - Campus Juiz de Fora - marco.araujo@ifsudestemg.edu.br

uma complexidade ciclomática que pode ser diminuída sem que o comportamento do programa seja alterado. Em um trabalho anterior, CAMPOS JUNIOR et al. (2016a), esse fenômeno foi observado no ambiente acadêmico de ensino de programação, onde se manifestou em 16% dos 482 códigos fonte analisados.

A existência de uma ferramenta desenvolvida em outro trabalho anterior que possibilita a análise estática do código fonte, juntamente com os dados sobre a ocorrência da complexidade desnecessária motivaram a criação de uma abordagem para detectar esse fenômeno de forma sistemática. Dessa forma, o objetivo deste trabalho é apresentar uma extensão às funcionalidades de uma ferramenta que auxilia os testes de software e possibilita a análise estática de códigos fonte desenvolvidos em linguagem Java.

Até o momento da escrita deste artigo, não foram encontradas outras ferramentas ou trabalhos que têm como objetivo a identificação da complexidade ciclomática desnecessária. Entretanto, alguns trabalhos citam o termo.

BENNET e RAJLICH (2000) discorrem sobre a área de manutenção e evolução de software, procurando identificar as principais oportunidades de pesquisa para os próximos dez anos. Um dos tópicos citados envolve a reestruturação do código fonte para remover complexidade desnecessária. Já BOEHM e PAPACCIO (1988) citam como um dos fatores que causam um aumento no custo do software a complexidade desnecessária.

O trabalho segue estruturado em mais três seções. Na seção Métodos, é apresentada a ferramenta desenvolvida e a abordagem de identificação da complexidade desnecessária. A seção Resultados e Discussão tem como objetivo a exemplificação do uso da ferramenta através de um cenário de uso, demonstrando também um caso em que é possível ocorrer a otimização ou diminuição da complexidade ciclomática desnecessária. Por fim, na seção Conclusão, são apresentadas as considerações finais deste trabalho.

MÉTODOS

Nessa seção será apresentada a ferramenta sobre a qual foi desenvolvida a abordagem para identificação da complexidade ciclomática desnecessária, bem como a própria abordagem.

Complexity Tool

A ferramenta utilizada é descrita nos trabalhos CAMPOS JUNIOR et al. (2016b) e CAMPOS JUNIOR et al. (2015). Possui como principal finalidade a análise estática de código fonte em linguagem Java, possibilitando o auxílio ao teste de unidade de software através da construção do grafo de fluxo de controle, cálculo da complexidade ciclomática e sugestão de condições a serem consideradas para a criação dos casos de teste.

Para criar uma abordagem sistemática para identificação da complexidade ciclomática desnecessária, é necessário que o código fonte seja analisado sintaticamente. Uma vez que a ferramenta já era capaz de efetuar certas análises sintáticas, a mesma foi expandida com a nova abordagem, gerando novas funcionalidades.

Dessa forma, as funcionalidades da ferramenta são:

- análise de arquivos de códigos fonte desenvolvidos em linguagem Java até a versão 1.7;
- geração de um grafo de fluxo de controle correspondente a cada método do código fonte analisado;
- visualização e navegação nos caminhos independentes existentes no GFC gerado;
- análise do GFC gerado, gerando informações sobre os caminhos independentes, condições para satisfazer cada caminho e o valor da complexidade ciclomática;
- indicação do trecho de código

correspondente a cada vértice do GFC gerado;
- análise da estrutura do GFC, identificando a existência da complexidade desnecessária;
- geração de novo GFC do método analisado, sem a complexidade desnecessária, denominado GFC otimizado.

Abordagem para identificação da complexidade desnecessária

Para identificar a complexidade ciclomática desnecessária e gerar um grafo de fluxo de controle otimizado, foi desenvolvida uma abordagem inicial dividida em quatro passos: (a) normalização das condições do método, (b) busca por clusters de condições, (c) identificação de otimização e (d) refatoração do GFC. A seguir são apresentados os pseudo códigos referentes a cada passo.

a. Normalização das condições do método

O objetivo desse passo é fazer com que todas as condições do código fonte analisado estejam em um mesmo padrão. O padrão adotado envolve manter do lado esquerdo da expressão condicional uma variável de comparação e do lado direito um literal ou outra variável a ser comparada. O pseudo código para executar essa etapa é apresentado na Listagem 1.

```
início
  para cada condição do método analisado
     se operando esquerdo não é uma variável e operando direito é uma variável
          inverter operadores
          inverter operandos
      fim se
   fim para cada
fim
```

Listagem 1. Pseudo código para normalização das condições de um método.

É importante notar que quando uma expressão de condição chega à esse procedimento, é sempre binária, ou seja, possui um operando esquerdo, um operador e um operando direito. A transformação de condições compostas (ex.: if (total==100 && quantidade>0)) em condições binárias ocorre em outro módulo da ferramenta que não está no escopo deste trabalho. Outro ponto importante é que os operadores que devem ser invertidos na operação "inverter operadores" são somente os seguintes: ">", "<", "≥" e "≤". Exemplo de entradas e saídas após a execução do procedimento são apresentadas na Tabela 1.

Tabela 1. Exemplos de normalização de condições.

#		Operando esquerdo	Operador	Operando direito		Saída
1	if(	total	==	1000	)	if(total == 1000)
2	if(	1000	==	total	)	if(total == 1000)
3	if(	total	>	1000	)	if(total > 1000)
4	if(	1000	>	total	)	if(total < 1000)
5	if(	1000	>=	total	)	if(total <= 1000)

b. Busca por clusters de condições

Nesse passo, as condições são agrupadas de acordo com as sua relações. Cada agrupamento é denominado um cluster de condições. Para que uma condição possa estar agrupada com outra, elas devem ter alguma ligação direta no GFC. O pseudo código dessa etapa é apresentado na Listagem 2.

```
início
   conjuntoClusters = novo conjunto de clusters
   conjuntoNós = novo conjunto de nós
   adicionar todos os nós do grafo ordenados por id ao conjuntoNós
   enquanto o conjuntoNós não está vazio
      clusterAtual = novo conjunto de nós
      nóAtual = primeiro elemento do conjuntoNós
      remover nóAtual do conjuntoNós
      nóDireito = nó à direita do nóAtual
      enquanto as condições para seguir para o nó direito forem satisfeitas
         se clusterAtual está vazio
            adicionar nóAtual ao clusterAtual
         fim se
         adicionar nóDireito ao clusterAtual
         remover nóDireito do conjuntoNós
         nóAtual = nóDireito
      fim enquanto
      adicionar clusterAtual ao conjuntoClusters
   fim enquanto
   retornar conjuntoClusters
fim
```

Listagem 2. Pseudo código para busca de clusters de condições.

A instrução do pseudo código "condições para seguir para o nó direito" se refere às seguintes regras que devem ser satisfeitas:

- o nó à direita do nó atual representa uma condição (se) e;
- essa condição não tem um senão, exceto quando o senão é seguido de outra condição, nesse caso, formando uma cadeia de condições se/senão aninhadas e;
- essa condição testa a mesma variável que é testada no nó atual e;
- os tipos das variáveis sendo testadas no nó atual e no nó à sua direita são iguais.

Dessa forma, na Tabela 2 é exemplificada a saída após a aplicação desse passo, para o código fonte dado.

Tabela 2. Exemplo de lista de clusters.

Código Fonte	Vértice	Lista Clusters	Condições cluster

int total = 200;
int quantidade = 1;
if (total == 200) {
  if (quantidade == 0) {
  print("vazio");
  } else {
  if (quantidade == 1) {
  print("1 unidade");
  } else {
  print("+ unidades");
  }
  }
} else {
  if (total == 100) {
  print("total 100")
  }
}
print("fim");
	
	1
	2
	3
	
	4
	5
	
	6
	
	7
	8
	
	9
		Cluster 1	if(total == 200)

			if(total == 100)
		Cluster 2	if(quantidade == 0)
			if(quantidade == 1)

c. Identificação de otimização

Após a normalização das condições e identificação dos clusters, nessa etapa é possível a identificação da complexidade ciclomática desnecessária, através da análise do intervalo de valores abrangidos por cada cluster. Se o intervalo abrange todo o domínio do tipo de variável do cluster, então a última condição do cluster pode ser substituída por uma instrução "senão", fazendo com que a complexidade ciclomática diminua e o comportamento do código continue o mesmo. O pseudo código para essa etapa é apresentado na Listagem 3.

```
início
   enquanto conjuntoClusters não está vazio
      clusterAtual = primeiro elemento do conjuntoClusters
      se o intervalo do clusterAtual é completo
         refatorar o grafo para o clusterAtual (passo 4)
         remover a última condição do clusterAtual
         atualizar o conjuntoClusters com a nova estrutura do grafo (passo 2)
         marcar o clusterAtual como checado
         se todos os clusters do conjuntoClusters foram checados
            se não houve modificação em nenhum cluster do conjuntoClusters
                  termina execução
            fim se
         fim se
      fim se
   fim enquanto
fim
```

Listagem 3. Pseudo código para identificação da otimização a partir dos clusters identificados.

d. Refatoração do GFC

Esse passo tem como objetivo modificar as ligações do GFC ao remover uma instrução de condição do código fonte analisado. Dessa forma, ao final da execução da abordagem, é possível apresentar um GFC otimizado que representa o código fonte analisado sem a complexidade ciclomática desnecessária.

```
início
   nóARemover = última condição do clusterAtual
   pai = nó pai do nóARemover
   Nó a direita do pai passa a ser o nó que está a esquerda do nóARemover
   enquanto nóARemover  tem nós incidentes
      conjuntoIncidentes = conjunto de nós incidentes ao nóARemover
      nóIncidente  = primeiro elemento do conjuntoIncidentes
      remover nóIncidente do conjuntoIncidentes
      se nó à esquerda do nóIncidente  = nóARemover
         nó à esquerda do nóIncidente passa a ser o 1º nó fora do escopo do clusterAtual
      senão
         se nó à direita do nóIncidente = nó à direita do nóARemover
            nó à direita do nóIncidente passa a ser o 1º nó fora do escopo do clusterAtual
         senão
            nó à direita do nóIncidente passa a ser nulo
         fim se
      fim se
   fim enquanto
fim
```

Listagem 4. Pseudo código para refatoração do GFC.

Durante a execução desse passo, o bloco de instruções que anteriormente ficava interno à condição a ser retirada passa a ficar à direita da condição externa, representando assim o "senão" dessa condição.

O fluxo de execução dos passos da abordagem é apresentado na Figura 1.

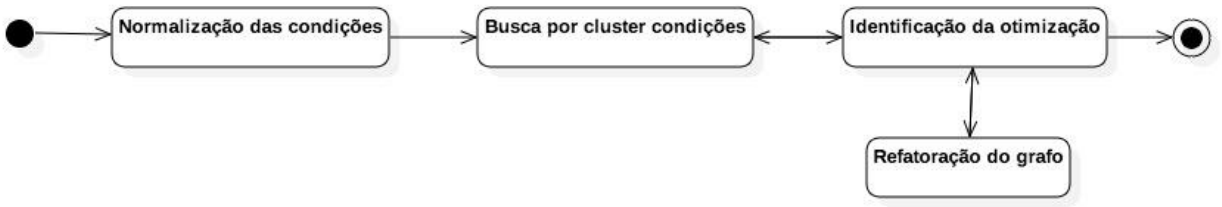

Figura 1. Fluxo de execução dos passos da abordagem.

Fluxo de atividades da ferramenta

O fluxo de uso da ferramenta era composto de carregamento da classe em linguagem Java a ser analisada; análise sintática do código fonte; geração do grafo de fluxo de controle; geração das sugestões de condições para os casos de teste. O novo fluxo, representado na Figura 2, adiciona uma nova etapa ao final do fluxo anterior, a verificação da existência da complexidade ciclomática desnecessária.

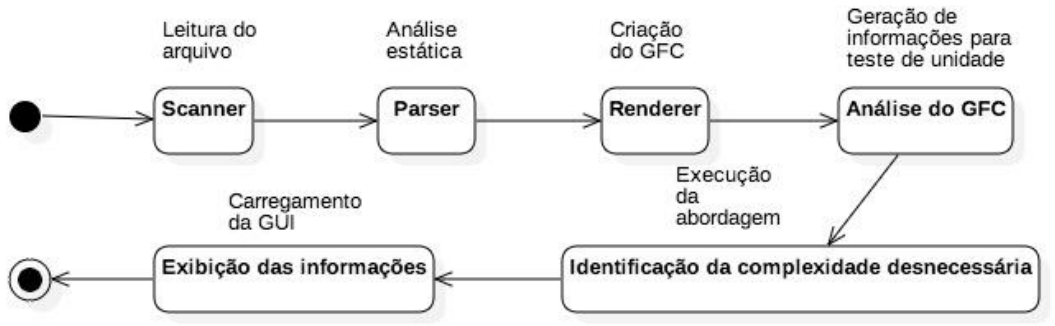

Figura 2. Fluxo de execução da ferramenta com a nova abordagem.

RESULTADOS E DISCUSSÃO

Com o desenvolvimento da abordagem e incorporação à ferramenta Complexity Tool, a mesma passou a ser capaz de identificar a ocorrência da complexidade ciclomática desnecessária em códigos fonte desenvolvidos em linguagem Java. O fato dessa funcionalidade ser incorporada à uma ferramenta que auxilia o teste de unidade de software é vantajoso pelo fato de, segundo BOGHDADY et al. (2011), a complexidade ciclomática estar diretamente ligada à quantidade de casos de teste necessários para se atingir uma boa cobertura das instruções testadas. Dessa forma, caso o desenvolvedor consiga diminuir a complexidade ciclomática de um método, consequentemente, estará diminuindo a quantidade de casos de teste necessários.

Em CAMPOS JUNIOR et al. (2016), foram executados experimentos para coletar dados quanto ao desempenho e à viabilidade da abordagem implementada na ferramenta descrita. O objetivo deste artigo, no entanto, é apresentar a ferramenta, um cenário e exemplo de seu uso. Dessa forma, a seguir é apresentado um cenário hipotético de uso da ferramenta.

CENÁRIO DE USO

Como exemplificação do uso da ferramenta com a nova abordagem, a seguir será apresentado um cenário hipotético.

Suponha que um desenvolvedor precise resolver um problema de programação com as seguintes especificações.

Tarefa: desenvolver um método em linguagem Java para calcular o Índice de Massa Corporal (IMC) de uma pessoa do sexo masculino e classificar o resultado de acordo com as regras apresentadas na Tabela 3. Os dados para o cálculo do IMC são massa (em quilos) e altura (em metros). Esses dados devem ser recebidos pelo método através de parâmetros. O cálculo é feito através da fórmula:

$$IMC = \frac{peso}{altura^2}.$$

Tabela 3. Regras para classificação do IMC.

Condição	IMC (intervalo)
"Abaixo do peso"	[-∞ , 20.7 [
"No peso normal"	[20.7, 26.4 [
"Marginalmente acima do peso"	[26.4, 27.8 [
"Acima do peso ideal"	[27.8, 31.1 [
"Obeso"	[31.1, +∞]

Baseado nessa especificação, o desenvolvedor desenvolve o código fonte representado na Figura 3.

```
public String testaIMC(float altura, float peso) {
   float imc;
   imc = altura / (peso * peso);
   if (imc < 20.7) {
      return "Abaixo do peso";
   }
   if ((imc >= 20.7) && (imc < 26.4)) {
      return "No peso normal";
   }
   if ((imc >= 26.4) && (imc < 27.8)) {
      return "Marginalmente acima do peso";
   }
   if ((imc >= 27.8) && (imc < 31.1)) {
      return "Acima do peso ideal";
   }
   if (imc >= 31.1) {
      return "Obeso";
   }
}
```

Figura 3: Código Fonte que testa o IMC

Com a tarefa resolvida, o desenvolvedor decide implementar os casos de teste de unidade para o método. Para isso, abre a ferramenta Complexity Tool e carrega a classe que contém o código fonte a ser analisado. A ferramenta apresenta o GFC exibido na Figura 4, que permite ao desenvolvedor visualizar os caminhos de execução de seu código fonte, como é o caso do caminho marcado em azul. Além disso, a ferramenta informa que a complexidade ciclomática do método analisado tem o valor 9, sugerindo que devem ser criados até 9 casos de teste de unidade para se obter a cobertura de todos os nós

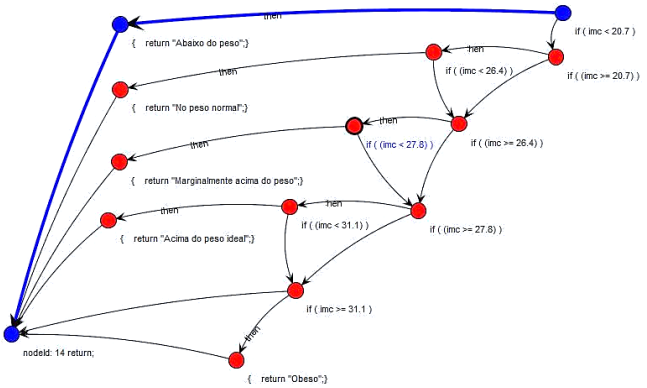

Figura 4. GFC referente ao método testaIMC.**com a**

Também é informado que o código fonte analisado apresenta uma complexidade ciclomática desnecessária e que a complexidade ciclomática ótima para esse caso seria de valor 5. Além disso, é exibido o GFC otimizado, conforme apresentado na Figura 5, para que o desenvolvedor possa entender como refatorar o método e eliminar a complexidade desnecessária.

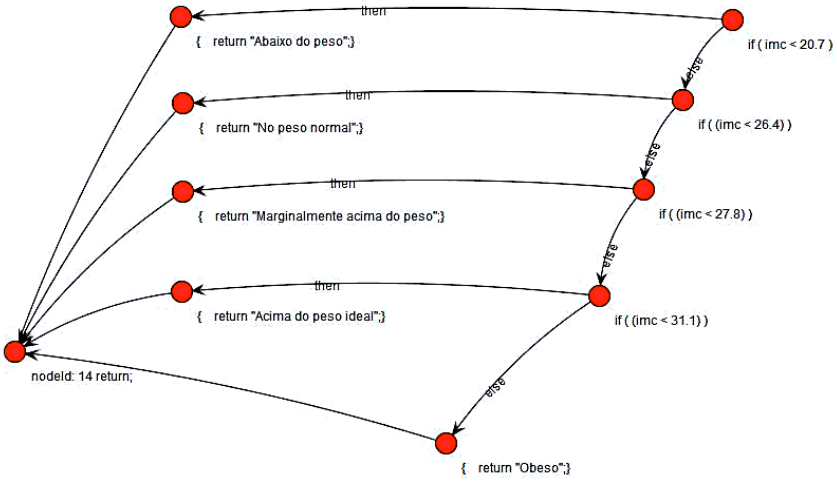

Figura 5. GFC otimizado.

O desenvolvedor refatora o método de acordo com o GFC observado na Figura 5. O código fonte refatorado pode ser observado na Figura 6. É interessante notar que, de fato, o novo código fonte aparenta ser menos complexo que a versão anterior, embora o seu objetivo continue o mesmo.

```
public String testaIMC(float altura, float peso) {
   float imc;
   imc = altura / (peso * peso);
   if (imc < 20.7) {
      return "Abaixo do peso";
   } else if (imc < 26.4) {
      return "No peso normal";
   } else if (imc < 27.8) {
      return "Marginalmente acima do peso";
   } else if (imc < 31.1) {
      return "Acima do peso ideal";
   } else {
      return "Obeso";
   }
}
```

Figura 6. Método testaIMC sem complexidade desnecessária.

O código fonte apresentado na Figura 6 tem uma complexidade ciclomática de valor 5. Comparando com a versão anterior, que tinha valor 9, o desenvolvedor agora necessita de no máximo 5 casos de teste para obter uma cobertura de 100% dos nós do GFC para seus testes de unidade, demonstrando assim, um cenário onde a ferramenta impacta na diminuição do esforço necessário para desenvolvimento dos testes de unidade para um determinado código fonte.

CONCLUSÃO

Apresentou-se neste trabalho uma ferramenta que implementa uma nova abordagem para indicação da complexidade ciclomática desnecessária em códigos fonte desenvolvidos em linguagem Java. Além disso, apresentou-se os passos necessários para implementação da abordagem, bem como o fluxo de sua execução. Como exemplificação do uso da ferramenta, um cenário de uso da ferramenta também foi apresentado, com o objetivo de esclarecer as aplicações da mesma.

Espera-se que, através do uso da nova funcionalidade da ferramenta apresentada, seja possível que desenvolvedores produzam código fonte sem complexidade desnecessária e que isso facilite a criação de casos de teste. Experimentos iniciais apresentados em trabalhos anteriores desmonstraram a viabilidade da abordagem, provendo indícios de que o seu uso é factível. Espera-se que, em trabalhos futuros, seja possível expandir o contexto estudado, que é de ensino de programação, para outros contextos.

Abstract: Programming students sometimes develop source code with redundant conditions, resulting in an unnecessary structural complexity. In this context, we aim to prevent this practice. Thus, the objective of this paper is to present an approach to detect the unnecessary cyclomatic complexity in a source code. The approach was included as a new feature in a tool that supports the unit test cases development for source code developed with the Java language. By using this feature, the user may simplify his algorithms and spend less time and effort on the unit testing phase, leading to a greater quality on the developed software.

Keywords: Software quality, software testing, unnecessary complexity, cyclomatic complexity, source code improvement.

BIBLIOGRAFIA

ALLEN, F. E. Control flow analysis, **Proceedings of a symposium on Compiler optimization**, Urbana-Champaign, Illinois, 1970, p. 1-19.

BENNETT, K. H. e RAJLICH, V. T. Software maintenance and evolution: a roadmap. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 22th, 2000, Limerick. **Proceedings of the Conference on the Future of Software Engineering**. ACM, 2000, p. 73-87.

BOEHM, B. W. e PAPACCIO, P. N. Understanding and controlling software costs. **IEEE Transactions on Software Engineering**, vol. 14, no. 10, p. 1462-1477, 1988.

BOGHDADY, P. N., BADR, N. L., HASHIM, M. A., & TOLBA, M. F. An enhanced test case generation technique based on activity diagrams. In: **Computer Engineering & Systems (ICCES), 2011 International Conference**. Cairo, IEEE, 2011, p. 289-294.

CAMPOS JUNIOR, H.S., MARTINS FILHO, L. R. V. e ARAÚJO, M. A. P. Uma ferramenta interativa para visualização de código fonte no apoio à construção de casos de teste de unidade. In: BRAZILIAN WORKSHOP ON SYSTEMATIC AND AUTOMATED SOFTWARE TESTING, 9th, 2015, Belo Horizonte. **Proceedings of the 9th Brazilian Workshop on Systematic and Automated Software Testing**. 2015, p. 31-40.

CAMPOS JUNIOR, H. S., MARTINS FILHO, L. R. V. e ARAÚJO, M. A. P. An Approach for Detecting Unnecessary Cyclomatic Complexity on Source Code. **IEEE Latin America Transactions**, vol. 14, no. 8, p. 3777-3783, 2016a.

CAMPOS JUNIOR, H. S., PRADO, A. F. e ARAÚJO, M. A. P. Complexity Tool: uma ferramenta para medir complexidade ciclomática de métodos Java. **Revista Multiverso**. v. 1, n.1, p. 66-76, 2016b.

MCCABE, T., J. e WATSON, A., H. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. **NIST Special Publication**. p. 500-235, 1996.

YU, S. e ZHOU, S. 2010. A Survey on Metric of Software Complexity. In: IEEE INTERNATIONAL CONFERENCE ON INFORMATION MANAGEMENT AND ENGINEERING. 2nd, 2010, Chengdu. **Proceedings of the 2nd IEEE International Conference on Information Management and Engineering**. IEEE, 2010, p. 352-356.