UMA ABORDAGEM PARA GERAÇÃO AUTOMATIZADA DE TESTES DE UNIDADE EM CÓDIGO FONTE SEM COMPLEXIDADE DESNECESSÁRIA

Gabriel Felix Vaneli[1], Marco Antônio Pereira Araújo[1,2]

[1] Instituto Federal do Sudeste de Minas Gerais, Juiz de Fora, Brasil
[2] Universidade Federal de Juiz de Fora, Juiz de Fora, Brasil

gabrielfelix.guitar@gmail.com, marco.araujo@ifsudestemg.edu.br

Abstract. *Programmers may choose to prioritize how the source code works and disregard aspects of its quality, making it difficult to be maintained and tested. The cyclomatic complexity, one of these quality aspects, measures the difficulty to understand a given source code. The occurrence of redundant conditional structures is seen as superfluous complexity, which can lead to unnecessary unit tests. In previous works, we developed a tool in order to assist the developer in the construction of these tests as well as to identify the presence of unnecessary complexity without the intention of eliminating it by a refactored suggestion of the source code. To improve developer support when building unit tests, we have developed a new approach that generates drafts of the unit tests. The generated cases indicate the data input that satisfy each condition existing in its independent paths, and the value that must be obtained in the data output to meet each test case. The purpose of this article is to describe the functioning of the new approach and its implementation in the tool as a new functionality.*

Resumo. *Programadores podem escolher priorizar o funcionamento do código fonte e desprezar aspectos de sua qualidade, tornando-o difícil de ser mantido e testado. A complexidade ciclomática, sendo um desses aspectos de qualidade, mede a dificuldade de entendimento para um determinado código fonte. A ocorrência de estruturas condicionais redundantes é caracterizada como complexidade desnecessária, podendo ocasionar em testes de unidade desnecessários. Uma ferramenta foi desenvolvida em trabalhos anteriores para auxiliar o desenvolvedor na*

construção desses testes e identificar a presença da complexidade desnecessária no intuito de eliminá-la por uma sugestão refatorada do código fonte. Para melhorar o apoio ao desenvolvedor na construção dos testes de unidade, foi desenvolvida uma nova abordagem que gera esboços dos casos de teste de unidade. Os casos gerados indicam os valores da entrada de dados que satisfaçam cada condição presente em seus caminhos independentes, e o valor que deve ser obtido na saída de dados para atender a cada caso de teste. O objetivo deste artigo é descrever o funcionamento da nova abordagem e a sua implementação na ferramenta como uma nova funcionalidade.

Palavras-chave: Complexidade Ciclomática, Manutenção e Evolução de Software, Teste de Software.

1 Introdução

Desenvolvedores de software podem escolher priorizar o funcionamento do código fonte e desprezar aspectos de sua qualidade, tornando-o difícil de ser mantido e testado. Segundo Pfleeger [1], um software que apresenta dificuldades em sua manutenção e também em ser testado se torna mais propício a apresentar defeitos para o usuário final, pois de acordo com Yu e Zhou [2], um código fonte de alta complexidade é difícil de ser compreendido, podendo assim, ocasionar em altos custos em sua manutenção.

Teste de software é uma técnica que apoia a qualidade e manutenção do mesmo, pois, de acordo com Delamaro et al. [3] serve para revelar defeitos durante o desenvolvimento, manutenção e evolução de um sistema. Assim, o uso e desenvolvimento dos testes trazem grande impacto não somente em sua boa estruturação e qualidade, mas também na parte financeira do projeto.

Em trabalhos anteriores, Campos Junior et al. [4] desenvolveu uma ferramenta capaz de auxiliar o desenvolvedor na construção de testes de unidade para software em linguagem Java através da exibição de caminhos do fluxo de execução. Também, identifica a presença de condições redundantes no código fonte a fim de evitar casos de teste desnecessários. Magalhães et al. [5] incrementou nessa ferramenta uma funcionalidade de refatoração para eliminação das condições redundantes do código fonte. O objetivo deste trabalho é apresentar uma abordagem de construção automática dos casos de teste, que funcione em conjunto com a abordagem da ferramenta [4, 5] de identificação da complexidade desnecessária em código fonte, com intuito de evitar a geração de casos desnecessários de teste. Assim busca-se apoiar a manutenção do software assegurando que não seja alterado seu comportamento externo, analisando a validação dos testes.

O trabalho segue estruturado da seguinte forma: Na Seção 2 é apresentado o referencial teórico, explicando conceitos sobre a abordagem de geração de testes de

unidade, e trabalhos relacionados. Na Seção 3, é descrito o funcionamento da nova abordagem de geração automatizada dos casos de teste de unidade. Na Seção 4 a abordagem é avaliada através de uma demonstração em um cenário de uso junto de um estudo experimental, e por fim, na Seção 5, são apresentadas as considerações finais.

2 Referencial Teórico

A medição da complexidade de um código fonte pode ser feita através da métrica complexidade ciclomática, proposta por McCabe [6]. Seu valor pode ser obtido através da quantidade de caminhos únicos presentes na estrutura dos fluxos de execução do código. Quanto maior for o valor dessa métrica, maior será a dificuldade de se entender, modificar e testar um código fonte. A análise dessa estrutura pode ser feita sobre o conceito de Grafos de Fluxo de Controle (GFC), apresentado por Allen [7], composto por vértices que fazem referência a blocos ou linhas do código fonte, e arestas que conectam os vértices formando caminhos do fluxo de execução, identificando assim quais são os possíveis caminhos de execução de um algoritmo.

Segundo IEEE [8], cada caminho único presente na estrutura do código fonte é a representação de um caso de teste de unidade, baseado no critério de cobertura todos-nós. Para cobrir as saídas e condições de um algoritmo, a quantidade de casos de teste deve ser no máximo igual à quantidade de caminhos independentes no grafo (GFC) da complexidade ciclomática, exceto quando algum desses caminhos não é atingível. Esse tipo de teste tem como foco testar o funcionamento das menores partes testáveis do código fonte em cada método, verificando se cada um deles retorna determinado resultado esperado para uma entrada de dados específica. Essas pequenas partes testáveis do código são conhecidas como unidades. Uma tabela pode ser utilizada com intuito de descrever os valores que as variáveis devem assumir para atingir cada um dos resultados. Essa tabela apresenta entradas (caso de teste) com base nos critérios de Análise do Valor Limite, definidos por Clarke et al. [9], que tem como premissa utilizar valores próximos à fronteira dos valores estabelecidos pelas condições em que são testados. Na **Tabela 1** são mostrados exemplos de valores limite definidos para as condições binárias serem verdadeiras, com base no tipo numérico de sua variável (`n`) abrangida.

Tabela 1. Condições e seus valores (limite) com base no tipo da variável.

Condição		Valor Limite	Tipo
`(n < 10)`	`(n != 10)`	`n = 9`	`Integer`
		`n = 9.9`	`Float`

<table>
<tr><td>(n <= 10)</td><td>(n == 10)</td><td>(n >= 10)</td><td>n = 10</td><td>Integer</td><td>Float</td></tr>
<tr><td colspan="2" rowspan="2">(n != 10)</td><td rowspan="2">(n > 10)</td><td>n = 10.1</td><td colspan="2">Float</td></tr>
<tr><td>n = 11</td><td colspan="2">Integer</td></tr>
</table>

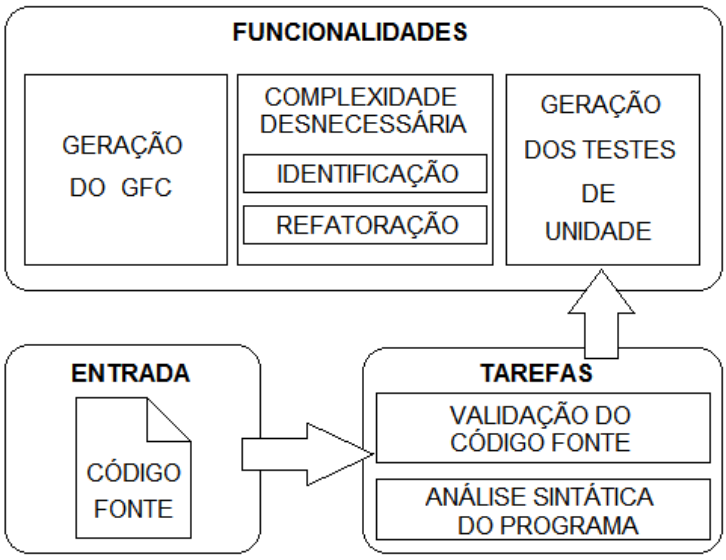
Fonte: Elaborada pelo autor

2.1 Complexity Tool

Em trabalhos anteriores foi desenvolvida uma ferramenta, chamada *Complexity Tool*, que presta auxílio na criação de testes de unidade de software em linguagem de programação Java. O seu desenvolvimento original se encontra descrito nos trabalhos de Campos Junior et al. [10, 11]. A ferramenta foi evoluída para identificar um fenômeno chamado complexidade ciclomática desnecessária, e sugerir uma correção do código fonte através do uso de um segundo GFC. Este fenômeno, observado por Campos Junior et al. [4] em 16% de 482 tarefas de programação no ambiente acadêmico, ocorre quando o código fonte possui instruções condicionais redundantes, que se não eliminadas, culmina na geração de casos de teste desnecessários. A implementação dessa abordagem se encontra descrita nos trabalhos de Campos Junior et al. [4, 12]. A implementação da funcionalidade que permite a ferramenta sugerir uma versão refatorada do código fonte lido sem a complexidade desnecessária, é descrita no trabalho de Magalhães et al. [5]. Estudos experimentais foram executados ao longo do desenvolvimento da ferramenta para validação de suas funcionalidades, cujos resultados também estão descritos nos trabalhos anteriores [4, 5, 10]. Uma agregação desses resultados pode ser encontrada em Magalhães et al. [13]. A **Figura 1** retrata a arquitetura da ferramenta com suas funcionalidades planejadas e implementadas como um esquema básico do funcionamento geral da ferramenta.

A geração dos testes de unidade é a última e mais recente dessas funcionalidades, cuja abordagem constrói a estrutura dos casos de teste para cada método do código fonte analisado, e com os casos em quantidade equivalente ao valor da complexidade ciclomática do método, sendo que em alguns casos, são apresentados valores de entrada e saída do caso de teste. Mais detalhes da abordagem desenvolvida são explicados na seção seguinte.

Figura 1. Arquitetura da ferramenta *Complexity Tool*.



Fonte: Elaborada pelo autor

2.2 Trabalhos Relacionados

Para apoiar o desenvolvimento de uma abordagem que gere automaticamente casos de teste de unidade, trabalhos e ferramentas com objetivos semelhantes foram buscados na literatura de forma ad-hoc.

EvoSuite é uma ferramenta criada por Fraser e Arcuri [14] para geração automática de conjuntos de testes JUnit em código fonte Java. A abordagem utilizada por essa ferramenta gera testes utilizando algoritmos genéticos para aleatoriamente construir testes de modo que se busquem os que possuem melhor cobertura do código fonte.

Pacheco et al. [15] apresentam uma ferramenta chamada *Randoop* para geração automática do conjunto de testes JUnit. Esta cria, para cada classe testada, uma chamada sequencial de seus métodos e construtores que, por sua vez, manipulam o estado dos objetos instanciados com base no comportamento do código a ser testado.

Silvia et al. [16] observaram que ambas as ferramentas gratuitas são amplamente utilizadas por desenvolvedores. Com isso, decidiram realizar um estudo exploratório para averiguar a eficiência dos conjuntos de testes gerados por essas ferramentas [14, 15], em relação à capacidade de prevenir erros de refatoração. O estudo evidenciou que, tanto o *EvoSuite* quanto o *Randoop* são falíveis em identificar defeitos de refatoração dos códigos em que são gerados os testes.

3 Abordagem Automatizada para Construir Casos de Teste

Nesta seção é descrita a abordagem responsável por gerar os esqueletos dos casos de teste de unidade para cada um dos caminhos únicos do fluxo de execução do código fonte a ser testado.

A ferramenta lê um código fonte de arquivo Java que possua pelo menos uma classe, lista os seus métodos, executa a abordagem de detecção da complexidade desnecessária em cada método lido, e gera uma versão refatorada de seu código fonte, caso aplicável, para que não sejam gerados casos de teste desnecessários.

Antes da execução da abordagem é verificado se cada método lido possui alguma instrução de retorno. A abordagem não suporta a geração de testes para métodos sem retorno e por isso, é exibida uma mensagem informando que aquele método não tem retorno.

Considerando que o método lido possui instrução de retorno, a abordagem é iniciada gerando e construindo a estrutura básica da classe que conterá os casos de teste conforme a **Listagem 1** exibe. Através do método **`setUp()`** a classe de teste inicializa uma nova instância da classe para realizar as chamadas do método a ser testado em cada caso.

```
public class ClasseTestadaTest(){

    ClasseTestada ct;

    @Before
    public void setUp(){
        ct = new ClasseTestada();
    }

    /* casos de teste de unidade */

}
```

Listagem 1. Estrutura básica dos casos de teste.

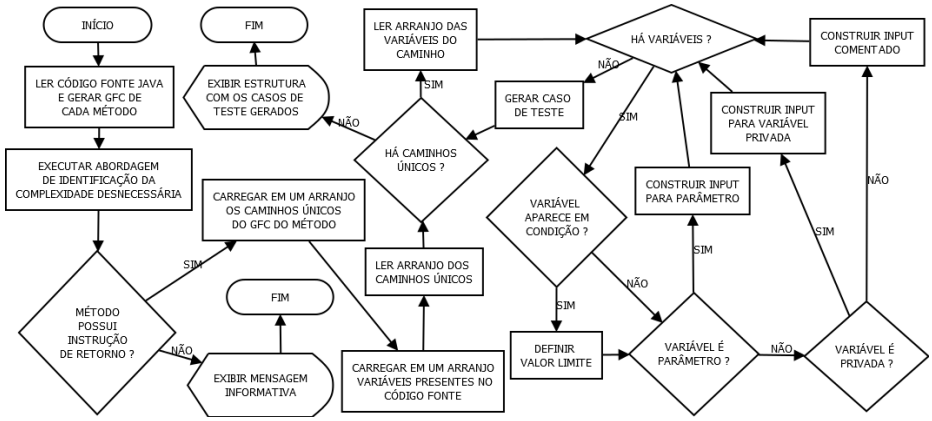
Em seguida é listado cada caminho único de execução do GFC desse método. Um laço de repetição se inicia para ler os nós do grafo de cada um desses caminhos e então construir a estrutura do seu caso de teste.

Para cada caminho lido, listam-se as variáveis utilizadas no caminho, no intuito de que seus valores limite sejam definidos com base nos critérios de Clarke et al. [9], e assim, já preencher a estrutura do caso de teste com esse valor a fim de facilitar o trabalho do desenvolvedor.

Começando a construir o caso de teste, cada variável é identificada, e é construído um *input* com o seu valor limite. Esse valor limite é definido a partir da

expressão presente na última instrução condicional em que a variável aparece no caminho sendo testado, antes da instrução de retorno. Caso a variável não se encontre em nenhuma das condições presentes no caminho, cabe então ao desenvolvedor definir o valor que esta assumirá no teste de unidade. Um fluxograma esquemático da geração dos casos de teste está retratado na **Figura 2**.

Figura 2. Fluxograma de geração dos casos de teste.



Fonte: Elaborada pelo autor

O *input* com o valor limite definido de cada variável que aparece em pelo menos uma instrução condicional pode ser construído de três formas diferentes conforme o modo em que esta é declarada na classe. As próximas listagens para cada forma descrita nos parágrafos abaixo são exemplos de teste para o método da **Listagem 2**.

```
public class ClasseTestada {

    public String metodoTestado(double num){
        if (num < 10){
            return "Menor que 10";
        } else {
            return "Maior ou igual a 10";
        }
```

```
        }

    }
```

Listagem 2. Método sendo testado.

Se a variável é um parâmetro do método sendo testado, é criada uma declaração dessa variável com o valor limite já inserido, de acordo com a **Listagem 3**, e na chamada do método no *output* é atribuída essa variável como parâmetro.

```
    @Test
    public void test_1_metodoTestado(){
        double num = 9.9;
        String output = "Menor que 10";
        assertEquals(output, ct.metodoTestado(num));
    }
```

Listagem 3. *Input* como declaração de variável.

Se a variável é uma declaração privada da classe, é inserida uma chamada para o método que define o valor dessa variável encapsulada (com prefixo "set-") conforme mostra a **Listagem 4**.

```
    @Test
    public void test_1_metodoTestado(){
        ct.setNum(9.9);
        String output = "Menor que 10";
        assertEquals(output, ct.metodoTestado(num));
    }
```

Listagem 4. *Input* como chamada de método *set*.

Se a variável é declarada internamente dentro do método ou então a variável não se encaixa em nenhuma dessas três formas diferentes, então é inserida uma declaração comentada dessa variável, conforme a **Listagem 5**, para que o desenvolvedor saiba que o valor dessa variável precisa ser definido com o valor limite nela atribuída.

```
    @Test
    public void test_1_metodoTestado(){
        //double num = 9.9;
        String output = "Menor que 10";
        assertEquals(output, ct.metodoTestado(num));
    }
```

Listagem 5. *Input* como declaração comentada de variável.

E por fim, a instrução de retorno é buscada no final do caminho para determinar o valor da saída que o caso de teste deverá ser validado. O output é construído com o método "*assertEquals*()", próprio da biblioteca JUnit, tendo como parâmetros o valor que se espera na saída do método sendo testado e a sua chamada, conforme as **Listagens 3**, **4** e **5** já mostram.

Para cada método lido do código fonte, a abordagem agrupa os seus casos de teste em uma única estrutura conforme mostra a **Listagem 6,** que é um exemplo dos casos de teste gerados para o código fonte da **Listagem 2**.

```
public class ClasseTestadaTest {

    ClasseTestada ct;

    @Before
    public void setUp(){
        ct = new ClasseTestada();
    }

    @Test
    public void test_1_metodoTestado(){
        double num = 9.9;
        String output = "Menor que 10";
        assertEquals(output, ct.metodoTestado(num));
    }

    @Test
    public void test_2_metodoTestado(){
        double num = 10;
        String output = "Maior ou igual a 10";
        assertEquals(output, ct.metodoTestado(num));
    }

}
```

Listagem 6. Casos de teste de unidade gerados.

4 Avaliação da Abordagem

Nesta seção são apresentados, um cenário hipotético com o objetivo de exemplificar o uso da abordagem desenvolvida e um estudo experimental com a abordagem desenvolvida para comparar os resultados de cobertura de seus testes gerados com os testes das ferramentas *EvoSuite* [14] e *Randoop* [15].

4.1 Cenário de Uso

Suponha que um desenvolvedor precise desenvolver, dentro de uma classe, um método com um algoritmo para calcular o Índice de Massa Corporal (IMC) de uma pessoa através da fórmula:

$$\text{IMC} = \frac{\text{peso}}{(\text{altura})^2}$$

O método deve receber como parâmetros o peso e a altura da pessoa, ambos de tipo *float*. O resultado correspondente a cada valor calculado é mostrado na **Tabela 2**.

Seguindo o enunciado o desenvolvedor então desenvolve o código fonte conforme a **Listagem 7**.

Tabela 2. Regras para classificação do IMC.

Mensagem	IMC
Abaixo do peso	IMC < 18.5
Peso normal	18.5 <= IMC <= 24.9
Pré-obesidade	24.9 < IMC <= 29.9
Obesidade	29.9 < IMC <= 34.9
Obesidade mórbida	IMC > 34.9

Fonte: Elaborada pela autor

```
public class TaxasSaude {

    public String classificaIMC(float peso, float altura){
        float imc = peso / (altura * altura);
        String s = "";
        if (imc < 18.5){
            s = "Abaixo do peso";
        } else if (imc >= 18.5 && imc <= 24.9){
            s = "Peso normal";
        } else if (imc > 24.9 && imc <= 29.9){
            s = "Pré-obesidade";
        } else if (imc > 29.9 && imc <= 34.9){
            s = "Obesidade";
        } else if (imc > 34.9){
            s = "Obesidade mórbida";
        }
        return s;
    }

}
```

Listagem 7. Código fonte desenvolvido conforme a tarefa apresentada.

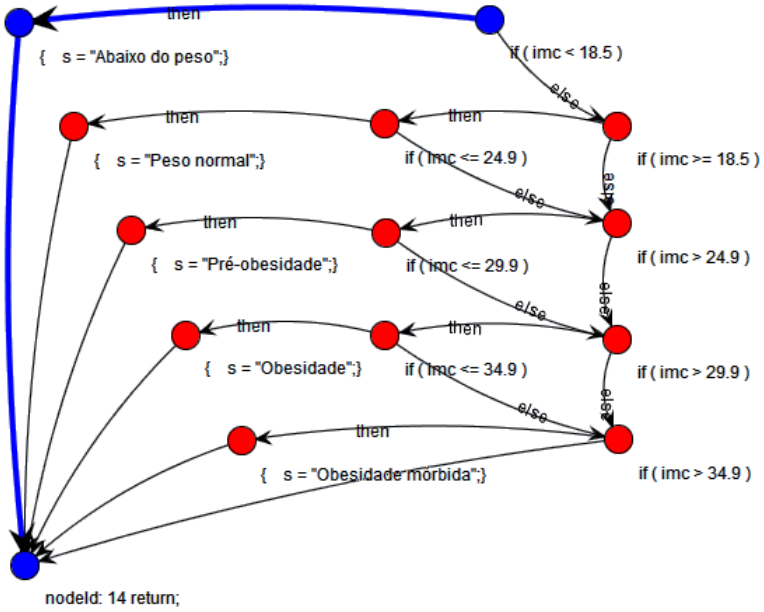
Com a tarefa concluída, o desenvolvedor decide utilizar a ferramenta *Complexity Tool* na codificação dos casos de teste de unidade. Para isso, carrega na ferramenta o arquivo que contém o código fonte a ser analisado. A ferramenta então apresenta o seu GFC, conforme a **Figura 3**, e na aba *Source Code* é visualizado o código fonte do método desenvolvido, conforme a **Figura 4**.

Na aba *Analysis*, é informado ao usuário que o código fonte lido possui uma complexidade ciclomática de valor 9, sugerindo assim que devem ser criados até 9 casos de teste de unidade para cobrir todos os vértices do GFC. Cada caminho único, referente a um dos casos a serem testados se encontra destacado, e é exibido em azul no GFC, e estes representam casos de teste de unidade a serem construídos, também é permitido selecionar o caminho escolhido através de um combobox na ferramenta *Complexity Tool*. É exemplificado um desses caminhos possíveis na **Figura 3**. O critério de teste todos-nós garante que todos os nós do GFC são visitados pelo menos uma vez durante a execução dos testes de unidade.

É informado também que o código fonte apresenta complexidade ciclomática desnecessária, e é sugerida uma complexidade ciclomática ideal de valor 5 para esse código fonte, sugerindo então que devem ser criados até 5 casos de testes ao invés de 9. O GFC refatorado é exibido conforme a **Figura 5** e a sugestão de alteração do código fonte, baseada no GFC refatorado é exibida na aba *Refactored Code*, conforme **Figura 6**.

A ferramenta exibe na aba *Unit Test* os 5 casos de teste de unidade construídos para o método desenvolvido, conforme a **Figura 7** mostra, com um dos *inputs* e os *outputs* já preenchidos. O desenvolvedor então visualiza os testes gerados pela ferramenta e estes podem ser selecionados e copiados para a área de transferência, a fim de facilitar a construção dos casos de teste de unidade.

Figura 3. GFC referente ao método classificaIMC().



Fonte: Print Screen da ferramenta *Complexity Tool*

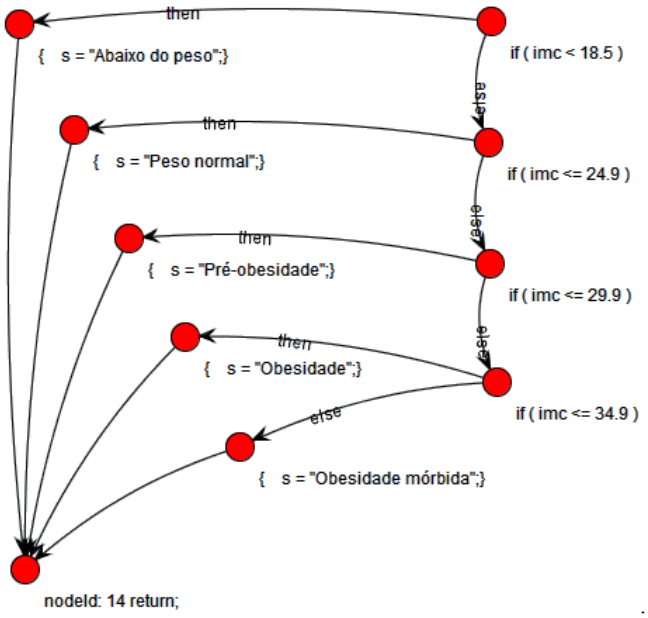
Figura 4. Exibição do código fonte analisado.

Source Code | Analysis | Refactored Code | Unit Test

```
1  public String classificaIMC(float peso, float altura) {
2     float imc = peso / (altura * altura);
3     String s = "";
4     if (imc < 18.5) {
5        s = "Abaixo do peso";
6     } else if (imc >= 18.5 && imc <= 24.9) {
7        s = "Peso normal";
8     } else if (imc > 24.9 && imc <= 29.9) {
9        s = "Pré-obesidade";
10    } else if (imc > 29.9 && imc <= 34.9) {
11       s = "Obesidade";
12    } else if (imc > 34.9) {
13       s = "Obesidade mórbida";
14    }
15    return s;
16 }
```

Fonte: Print Screen da ferramenta *Complexity Tool*

Figura 5. GFC refatorado do método classificaIMC()



Fonte: Print Screen da ferramenta *Complexity Tool*

Figura 6. Exibição da sugestão de refatoração.

Source Code | Analysis | Refactored Code | Unit Test

```
1  public String classificalMC(float peso, float altura) {
2     float imc = peso / (altura * altura);
3     String s = "";
4     if (imc < 18.5) {
5        s = "Abaixo do peso";
6     } else if (imc <= 24.9) {
7        s = "Peso normal";
8     } else if (imc <= 29.9) {
9        s = "Pré-obesidade";
10    } else if (imc <= 34.9) {
11       s = "Obesidade";
12    } else {
13       s = "Obesidade mórbida";
14    }
15    return s;
16 }
```

Fonte: Print Screen da ferramenta *Complexity Tool*

Figura 7. Casos de teste de unidade gerados para o método classificalMC().

Source Code | Analysis | Refactored Code | Unit Test

```
1  import org.junit.Test;
2  import static org.junit.Assert.*;
3  import org.junit.Before;
4
5  public class TaxasSaudeTest {
6
7     TaxasSaude ts;
8
9     @Before
10    public void setUp() {
11       ts = new TaxasSaude();
12    }
13
14    @Test
15    public void test_1_classificalMC() {
16       // float imc = 18.4;
17       float peso = /*insert value*/;
18       float altura = /*insert value*/;
19       String output = "Abaixo do peso";
20       assertEquals(output, ts.classificalMC(peso, altura));
21    }
22
23    @Test
24    public void test_2_classificalMC() {
25       // float imc = 24.9;
26       float peso = /*insert value*/;
27       float altura = /*insert value*/;
28       String output = "Peso normal";
29       assertEquals(output, ts.classificalMC(peso, altura));
30    }
31
32    @Test
33    public void test_3_classificalMC() {
34       // float imc = 29.9;
35       float peso = /*insert value*/;
36       float altura = /*insert value*/;
37       String output = "Pré-obesidade";
38       assertEquals(output, ts.classificalMC(peso, altura));
39    }
40
41    @Test
42    public void test_4_classificalMC() {
43       // float imc = 34.9;
44       float peso = /*insert value*/;
45       float altura = /*insert value*/;
46       String output = "Obesidade";
47       assertEquals(output, ts.classificalMC(peso, altura));
48    }
49
50    @Test
51    public void test_5_classificalMC() {
52       // float imc = 35.0;
53       float peso = /*insert value*/;
54       float altura = /*insert value*/;
55       String output = "Obesidade mórbida";
56       assertEquals(output, ts.classificalMC(peso, altura));
57    }
58
59 }
```

Fonte: Print Screen da ferramenta *Complexity Tool*

Neste exemplo o desenvolvedor despende menos tempo com o desenvolvimento dos casos de teste, pois quase tudo que ele precisa codificar já é gerado automaticamente pela abordagem implementada na ferramenta, basta que seja copiado para a área de transferência. Também, menos casos de teste precisam ser construídos já que a complexidade ciclomática do código diminuiu.

Resta apenas o usuário indicar os valores que as variáveis `peso` e `altura` precisam assumir para que a variável `imc` obtenha o respectivo valor limite definido em cada um dos casos de teste. Como exemplo, é proposto para este caso, que em cada teste o `peso` assuma o valor limite definido, e a `altura` seja de valor 1, pois conforme a fórmula do IMC definida, dividindo o peso pelo quadrado da altura (que será 1 também), resultará, para o `imc`, o respectivo valor limite atribuído ao `peso`.

4.2 Estudo Experimental

Com o objetivo de avaliar as funcionalidades providas pela ferramenta desenvolvida, foi executado um estudo experimental para comparar as coberturas obtidas pelos testes de unidade gerados pela abordagem desenvolvida com o das ferramentas *EvoSuite* versão 1.0.5 e *Randoop* versão 3.1.5. Utilizou-se o código fonte da **Listagem 7** como exemplo para geração dos conjuntos de testes de unidade. Foi utilizado o plug-in "JaCoCo" da IDE NetBeans para obtenção da cobertura dos casos de teste. Os resultados dessas coberturas se encontram nas **Figuras 8, 9** e **10**.

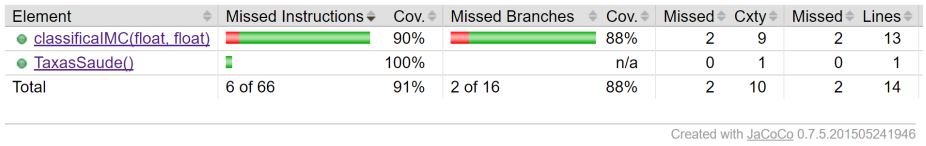
Figura 8. Cobertura dos testes gerados pela ferramenta *Evosuite*.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines
classificalMC(float, float)		100%		100%	0	9	0	13
TaxasSaude()		100%		n/a	0	1	0	1
Total	0 of 66	100%	0 of 16	100%	0	10	0	14

Created with JaCoCo 0.7.5.201505241946

Fonte: Print Screen da ferramenta *JaCoCo*

Figura 9. Cobertura dos testes gerados pela ferramenta *Randoop*.

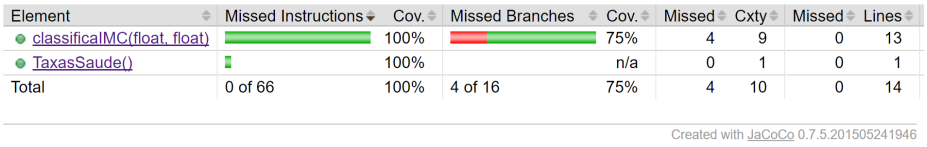


Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines
classificalMC(float, float)		90%		88%	2	9	2	13
TaxasSaude()		100%		n/a	0	1	0	1
Total	6 of 66	91%	2 of 16	88%	2	10	2	14

Created with JaCoCo 0.7.5.201505241946

Fonte: Print Screen da ferramenta *JaCoCo*

Figura 10. Cobertura dos testes gerados pela ferramenta *Complexity Tool*.



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines
classificalMC(float, float)		100%		75%	4	9	0	13
TaxasSaude()		100%		n/a	0	1	0	1
Total	0 of 66	100%	4 of 16	75%	4	10	0	14

Created with JaCoCo 0.7.5.201505241946

Fonte: Print Screen da ferramenta *JaCoCo*

Os testes gerados pela *EvoSuite* obtiveram cobertura de 100% das saídas possíveis e das condições também. Já o *Randoop* gerou testes com coberturas de 90% e 88%. Os testes gerados pela abordagem implementada na ferramenta *Complexity Tool* obtiveram cobertura de 100% das saídas, porém, 75% das condições, pois seus testes gerados são para cobrir os 5 casos possíveis da versão sem complexidade desnecessária do código fonte.

Apesar de evidente que a *EvoSuite* não tenha detectado condições redundantes no código fonte, não significa portanto que tenha gerado testes desnecessários; pois foi observado que a quantidade de seus testes gerados se apresentou menor do que a quantidade limite de 9 casos. O *Randoop* por sua vez constrói testes com valores de input fora do valor limite [9], mas definidos de forma aleatória.

5 Considerações Finais

Como a demanda por software se faz aumentar a cada dia, acredita-se que a geração automatizada dos testes de unidade junto com a eliminação da complexidade desnecessária proporcionem maior agilidade no desenvolvimento e melhor manutenção de qualidade do software.

A aplicação dessa solução em uma ferramenta que detecta complexidade desnecessária pode fazer com que se evite a geração de testes de unidade desnecessários. A abordagem, no entanto, ainda é limitada para apenas métodos que possuam instrução de retorno em sua estrutura e nem todos os casos de teste gerados terão seus *inputs* preenchidos pela abordagem. Também por ser baseada no valor limite

das condições presente no fluxo de execução, não é garantido que os testes gerados terão sempre cobertura total do código fonte.

Como trabalhos futuros, novos estudos experimentais envolvendo estudantes do ensino superior serão realizados para averiguar o funcionamento da abordagem desenvolvida e também validar a hipótese de que a ferramenta consegue diminuir o tempo de desenvolvimento dos testes de unidade, pois no estudo experimental anterior [5] a ferramenta não conseguiu diminuir o esforço de teste.

6 Referências

1. Pfleeger, S. L. **Engenharia de software: teoria e prática**. Prentice Hall, 2004.
2. 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, p. 352-356, 2010.
3. Delamaro, M.; Jino, M.; Maldonado, J. **Introdução ao teste de software (2ª Edição)**. Elsevier Brasil, 2017.
4. Campos Junior, H. S.; Martins Filho, L. R. V.; Araújo, M. A. P. "An Approach for Detecting Unnecessary Cyclomatic Complexity on Source Code". **IEEE Latin America Transactions**, vol. 14, no. 8, 2016.
5. Magalhães, N. M.; Campos Junior, H. S.; Araújo, M. A. P.; Neves, V. O. "An Automated Refactoring Approach to Remove Unnecessary Complexity in Source Code". In: **Proceedings of the 2nd Brazilian Symposium on Systematic and Automated Software Testing.** ACM, p. 3, 2017.
6. McCabe, T. J. "A complexity measure". In: **IEEE Trans. Software Eng**. Vol. SE-2, N. 4, p. 308-320, 1976.
7. Allen, F. E. "Control flow analysis", **Proceedings of a symposium on Compiler optimization**, Urbana-Champaign, Illinois, p. 1-19, 1970.
8. ISO/IEC/IEEE: 24765-2010 - Systems and software engineering – Vocabulary, pp. 1–418 (2010).
9. Clarke, L. A.; Hassell, J.; Richardson, D. J. "A close look at domain testing". **IEEE Transactions on Software Engineering**, n. 4, p. 380-390, 1982.
10. Campos Junior, H. S.; Martins Filho, L. R. V.; 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. p. 31-40, 2015.

11. Campos Junior, H. S.; Prado, A. F.; Araújo, M. A. P. "Complexity Tool: uma ferramenta para medir complexidade ciclomática de métodos Java". **Revista Multiverso**. v. 1, n.1, p. 66-76, 2016.

12. Campos Junior, H. S.; Martins Filho, L. R. V.; Araújo, M. A. P. "Uma Abordagem para Otimização da Qualidade de Código Fonte Baseado na Complexidade Estrutural". **Revista Multiverso**. v. 2, n.1, p. 13-21, 2017.

13. Magalhães, N. M.; Campos Junior, H. S.; Araújo, M. A. P. "Melhoria da Qualidade de Software Através da Eliminação da Complexidade Desnecessária em Código Fonte". **Revista Multiverso**. Aceito para publicação, 2017.

14. Fraser, G.; Arcuri, A. "Evosuite: automatic test suite generation for object-oriented software." In: **Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering.** ACM, p. 416-419. 2011.

15. Pacheco, C., Lahiri, S. K., Ernst, M. D., & Ball, T. "Feedback-directed random test generation." In: **Proceedings of the 29th international conference on Software Engineering.** IEEE Computer Society. p. 75-84. 2007.

16. Silva, I. P. S. C.; Alves, E. L. G.; Andrade, W. L. "Analyzing automatic test generation tools for refactoring validation." In: **Proceedings of the 12th International Workshop on Automation of Software Testing.** IEEE Press, p. 38-44. 2017.