MELHORIA DA QUALIDADE DE SOFTWARE ATRAVÉS DA ELIMINAÇÃO DA COMPLEXIDADE DESNECESSÁRIA EM CÓDIGO FONTE

Nathan Manera Magalhães[1], Heleno de Souza Campos Junior[2], Marco Antônio Pereira Araújo[1,2]

RESUMO: Programadores iniciantes podem escolher priorizar o funcionamento de um código-fonte ao desprezar sua qualidade, tornando-o difícil de ser mantido e testado. Com base nisso, um fenômeno chamado complexidade estrutural desnecessária pode ocorrer quando um método tem o valor de complexidade ciclomática que pode ser reduzido sem mudar o seu comportamento. Em trabalhos anteriores, foram desenvolvidas abordagens capazes de identificar a presença do referido fenômeno e mostrar ao desenvolvedor uma sugestão para reestruturar o código-fonte. Ainda, abordagens para auxiliar na construção de casos de teste de unidade. O objetivo deste artigo é agregar as evidências empíricas obtidas ao se avaliar experimentalmente as abordagens desenvolvidas e implementadas em uma ferramenta nomeada *Complexity Tool*. As evidências obtidas nos estudos, realizados em trabalhos anteriores, sugerem que as abordagens impactam significativamente em um aumento da cobertura de testes de unidades desenvolvidos e na identificação e remoção do fenômeno de complexidade ciclomática desnecessária. Não foram encontradas, no entanto, evidências que apontem para a diminuição do esforço necessário para criação de casos de teste de unidade.

PALAVRAS-CHAVE: Complexidade Ciclomática, Teste de Software, Refatoração do Código Fonte.

INTRODUÇÃO

Desenvolvedores de software, sobretudo iniciantes, podem optar por desenvolver código fonte se preocupando somente com que esse seja funcional, deixando sua qualidade de lado. Segundo LEHMAN (1980), conforme um software evolui, seu código fonte tende a tornar-se cada vez mais complexo e, de acordo com YU e ZHOU (2010), um código fonte com alta complexidade é de difícil compreensão, podendo ocasionar em custos elevados para sua manutenção.

Uma forma de se medir o quanto complexo é um código fonte, é através da métrica complexidade ciclomática, proposta por MCCABE (1976). Essa métrica se baseia na estrutura do fluxo de um código fonte para calcular a quantidade de fluxos de execução (caminhos únicos) presentes. Quanto maior for o valor dessa métrica, maior será a dificuldade de se entender, modificar e testar um código fonte. ZHANG e BADDOO (2007) afirmam que o uso da complexidade ciclomática para encontrar *bugs* em unidades de software tem maior desempenho do que outras métricas de complexidade testadas. A análise da estrutura do fluxo de um código fonte pode ser feita sobre o conceito de Grafos de Fluxo de Controle (GFC), apresentado por ALLEN (1970), 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 al-

[1] Nathan Manera Magalhães; IF Sudeste MG - Campus Juiz de Fora; nathan.manera@gmail.com
[2] Heleno de Souza Campos Junior; Universidade Federal de Juiz de Fora ; heleno_scj@hotmail.com
[1,2] Marco Antônio Pereira Araújo; IF Sudeste MG - Campus Juiz de Fora; Universidade Federal de Juiz de Fora; marco.araujo@ifsudestemg.edu.br

goritmo.

Conforme a complexidade ciclomática de um software aumenta e o desenvolvedor não se preocupa em manter sua qualidade, podem surgir condições redundantes no código fonte, que podem ser removidas, pois não mudam o comportamento externo do programa. Este fenômeno, chamado de complexidade ciclomática desnecessária, foi observado por CAMPOS JUNIOR et al. (2016b) no ambiente acadêmico de ensino de programação, onde se manifestou em 16% de 482 tarefas de programação analisadas.

Baseado no fenômeno observado, ao longo de trabalhos anteriores, criou-se uma ferramenta que permite a visualização de código fonte através de GFC, informando sua complexidade ciclomática, CAMPOS JUNIOR et al. (2016a) e casos de testes de unidade a serem criados, CAMPOS JUNIOR et al. (2015). Permite ainda, identificar complexidade ciclomática desnecessária para sugerir a sua eliminação através uso de um segundo GFC CAMPOS JUNIOR et al. (2016b e 2017) e provê uma sugestão de refatoração do código fonte, MAGALHÃES et al. (2017), sem a ocorrência do respectivo fenômeno. As etapas do seu desenvolvimento se encontram na próxima seção.

O objetivo deste artigo é agregar e discutir resultados empíricos obtidos por meio de estudos experimentais sobre as abordagens e ferramenta desenvolvida.

O trabalho segue estruturado em mais quatro seções. Na seção Material e Métodos são apresentados, os conceitos de complexidade desnecessária, a abordagem que faz a sua identificação e eliminação, e a arquitetura da ferramenta desenvolvida como também as suas funcionalidades e aplicações. Na seção Resultados, é feito um sumário dos estudos experimentais realizados para avaliar as funcionalidades da ferramenta. A análise dos resultados desses estudos é debatida na seção Discussão. E por fim, na seção Conclusão, são apresentadas as considerações finais.

MATERIAL E MÉTODOS

Nesta seção, serão apresentados os conceitos necessários para a compreensão da abordagem de identificação e eliminação da complexidade desnecessária. São também descritas a arquitetura e as funcionalidades / aplicações presentes na ferramenta.

Complexidade ciclomática desnecessária

Conforme já mencionado, a complexidade ciclomática desnecessária ocorre quando um código fonte possui instruções condicionais redundantes ou também condições inalcançáveis. Para eliminar a sua ocorrência, foi desenvolvida uma abordagem que analisa o GFC de um código fonte e detecta o referido fenômeno. Antes de explicar a abordagem, será dissertado o uso de intervalos numéricos para detecção da complexidade desnecessária.

Instruções condicionais com variáveis de tipo numérico são avaliadas como verdadeiras quando o operando de valor variável assume valores que pertençam a um intervalo numérico delimitado. Esse intervalo é definido por dois valores numéricos que representam os limites mínimo e máximo que a variável pode assumir na condição em que se encontra. Os limites são definidos pelo operador relacional e pelo operando de valor constante presentes nessa mesma condição. Dependendo do operador presente, pode acontecer de um desses limites ser um valor infinito, que caso seja o valor da esquerda do intervalo, representa o menor valor possível para o tipo da variável numérica, e caso seja o valor da direita do intervalo, representa o maior valor possível para o tipo da variável numérica. Esses valores infinitos são definidos conforme limitações de linguagem de programação e arquitetura do processador.

Como exemplo, na **Tabela 1** são apresentadas expressões binárias de condições com variável (n) de tipo numérico no ope-

rando de valor variável da expressão, que assume (de acordo com o operador relacional) valores maiores, menores, iguais ou diferentes do valor numérico (10) expresso no operando de valor constante, para que essa condição seja verdadeira. A condição com a expressão (n < 10), por exemplo, é considerada como verdadeira se (n) for um valor numérico que esteja entre o menor valor possível (–infinito) para seu tipo, e o valor (10). Isso é representado por um intervalo aberto com início em (–infinito) e fim em (10).

Tabela 1. Exemplo de definição dos intervalos para variáveis de tipo numérico.

Expressão	Intervalo definido
(n < 10)	] –∞; +10 [
(n <= 10)	] –∞; +10]
(n > 10)	] +10; +∞ [
(n >= 10)	[+10; +∞ [
(n == 10)	[+10; +10]
(n != 10)	] –∞; +10 [U] +10; +∞ [

Os intervalos de condições com variáveis de tipo não numérico (*String* e *boolean*) são definidos através de regras específicas elaboradas para fazer uso da abordagem com um intervalo numérico. Nessas regras, o fato de o operando de cada um desses tipos não possuir valores numéricos, permite o uso de valores fixos para definir os intervalos internamente. Estes intervalos, definidos pelas regras apresentadas nos parágrafos seguintes, diferentemente do tipo numérico, não representam um conjunto delimitado de valores que tornam a condição verdadeira. Apenas representam o valor (*true* ou *false*) análogo ao operador presente na expressão.

Na linguagem de programação Java, a utilização de expressões binárias para comparação de *Strings* é considerada uma má prática. O ideal é a utilização de métodos tais como "equals()" ou "equalsIgnoreCase()" em uma expressão unária. Uma expressão unária com operando desse tipo tem seu intervalo definido com base na aparição do operador lógico de negação (!). A ausência desse operador na expressão é representada por um intervalo fechado com início em (–1) e fim em (+1). Outra expressão que faz parte do mesmo cluster que a anterior e com a presença desse operador, é representada por um intervalo aberto de uma união entre dois intervalos, cujo primeiro tem início em (–infinito) e fim em (–1), e o outro tem início em (+1) e fim em (+infinito). A **Tabela 2** apresenta exemplos de expressões com uma variável (p) de tipo *String* que se diferenciam pela presença do operador (!), formando assim os intervalos pré-definidos.

Expressões unárias com operando do tipo *boolean* tem seu intervalo definido da mesma forma que o do tipo *String*. Porém o uso do tipo *boolean* em uma expressão binária admite apenas dois valores: *true* ou *false*. Os operadores lógicos (==) e (!=) são análogos a esses valores do operando. Com isso, o intervalo é definido através da combinação do valor do operando com o do operador. Em uma expressão com operando de valor igual ao do operador, é usado o intervalo fechado com início em (–1) e fim em (+1). Outra expressão que faz parte do mesmo *cluster* que a anterior, e com operando de valor oposto ao do operador, é usado o intervalo aberto de uma união entre dois intervalos, cujo primeiro tem início em (–infinito) e fim em (–1), e o outro tem início em (+1) e fim em (+infinito). Um exemplo de uso dessa regra está na **Tabela 2**, com expressões de uma variável (b) do tipo *boolean*, e que se diferenciam pela combinação do operador com o valor do operando constante.

Expressões binárias que possuam chamadas de métodos em seu operando variável, utilizam o intervalo respectivo ao tipo de retorno do método (numérico ou não). Na **Tabela 2**, é demonstrado um exemplo de expressão com o método getNumero(), retornando um valor de tipo numérico.

Tabela 2. Definição de intervalos para tipos de variáveis não numéricas.

Tipo	Expressão	Intervalo definido
String	`(p.equals("valor"))`	[–1; +1]
	`(!p.equals("valor"))`	] –∞; –1 [U] +1; +∞ [
Boolean	`(b == true)` ou `(b != false)`	[–1; +1]
	`(b != true)` ou `(b == false)`	] –∞; –1 [U] +1; +∞ [
Método	`(getNumero() < 10)`	] –∞; +10 [

Identificação da complexidade desnecessária

Vértices do GFC que possuem pelo menos uma aresta ligando diretamente um ao outro, e seguem padrões definidos em CAMPOS JUNIOR et al. (2016b, 2017), são agrupados em estruturas denominadas *clusters* de condições. Cada *cluster* forma um intervalo composto pela união dos intervalos formados de cada condição abrangida. Esse intervalo, ao ser analisado, permite com que a complexidade ciclomática desnecessária seja identificada quando ele abrange todo o domínio do tipo da variável sendo usada, significando então que a última condição do *cluster* pode ser eliminada e substituída por um *else*.

Antes de identificar o fenômeno da complexidade desnecessária, a abordagem faz uma normalização das expressões binárias dos vértices condicionais para formarem um padrão em que o operando da esquerda seja sempre uma variável e o operando da direita seja sempre uma constante, obedecendo assim à notação Variável / Operador / Constante. Caso não haja constantes nas condições, a abordagem não pode ser executada, uma vez que a mesma não considera o valor que as variáveis podem assumir ao longo da execução do programa.

Depois de definidas quais condições foram eliminadas, é previsto na abordagem a geração de um segundo GFC **(Figura 2)** para demonstrar como deve ser a estrutura do fluxo de execução do código fonte lido, sem a complexidade ciclomática desnecessária. Esse GFC é gerado a partir da otimização do primeiro **(Figura 1)**. Como exemplo, o código fonte **(Listagem 1)** é refatorado pela abordagem para ficar sem a complexidade desnecessária **(Listagem 2)**. Mais detalhes do processo de refatoração do código fonte se encontram na subseção seguinte.

```
public static String numeros(int num){
    String m = "";
    if (num < 0) {
        m = "Negativo";
    } else if (num >= 0 && num < 10) {
        m = "Positivo e menor que 10";
    } else if (num >= 10) {
        m = "Maior ou igual a 10";
    }
    return m;
}
```

Listagem 1. Exemplo de código fonte com complexidade ciclomática desnecessária.

```
public static String numeros(int num){
    String m = "";
    if (num < 0) {
        m = "Negativo";
    } else if (num < 10) {
        m = "Positivo e menor que 10";
    } else {
        m = "Maior ou igual a 10";
    }
    return m;
}
```

Listagem 2. Exemplo do código fonte da Listagem 1 sem a complexidade ciclomática desnecessária.

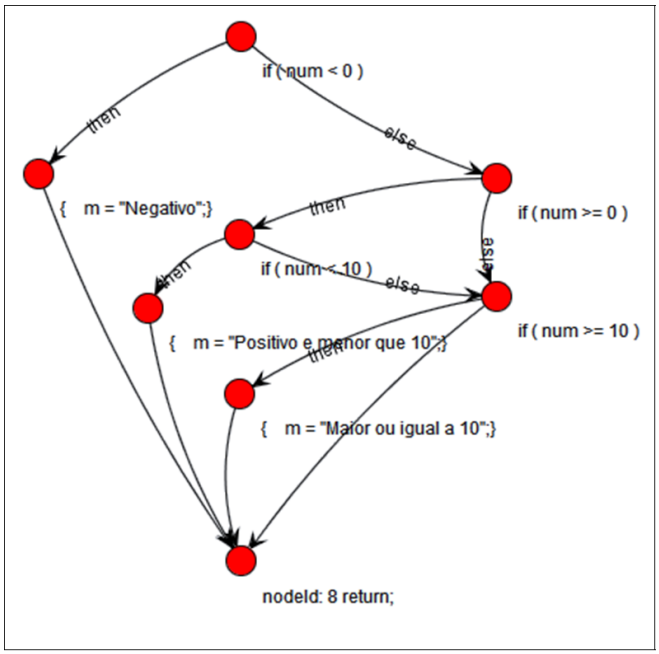

Figura 1. GFC relativo ao código fonte da Listagem 1.

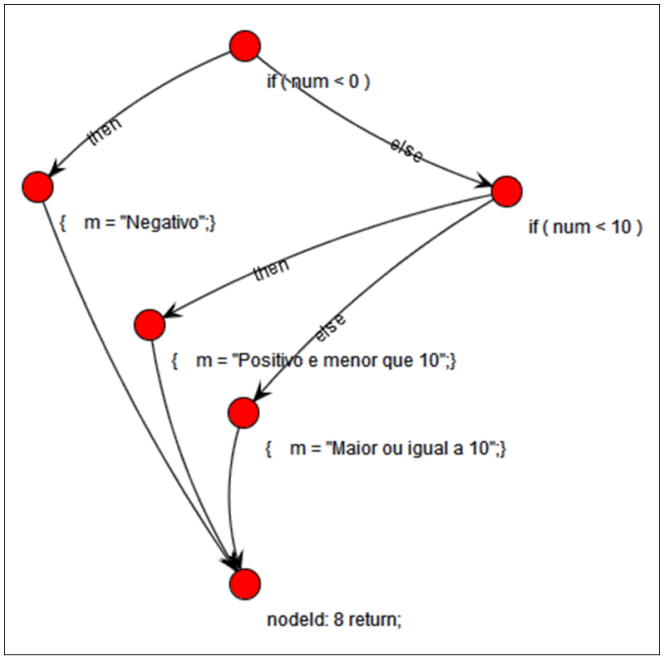

Figura 2. GFC relativo ao código fonte da Listagem 2.

No exemplo da **Listagem 1** são formados dois *clusters* de condições, sendo um com (num < 0) e (num >= 0), e outro com (num < 10) e (num >= 10). Em cada *cluster*, a união dos intervalos de suas condições forma um intervalo com valor mínimo de (–infinito) e valor máximo de (+infinito). Portanto essas instruções condicionais em um mesmo *cluster* podem ter sua variável (num) assumindo qualquer valor numérico entre o valor mínimo e o valor máximo do domínio de seu tipo, que neste caso é *int*. Assim, a complexidade ciclomática desnecessária é identificada e eliminada, quando a última condição presente em cada *cluster* é excluída do código fonte por meio de refatoração, gerando assim o exemplo da **Listagem 2.**

Refatoração do código fonte

Depois que a complexidade desnecessária é identificada no código fonte e são gerados o GFC representando o código fonte original e o GFC representando a sua otimização (a partir de agora chamado de refatorado), o código fonte é refatorado de forma automatizada através do processo descrito nos parágrafos abaixo.

Antes de iniciar a refatoração, são feitas listagens de todos os vértices do GFC original (GFCO) e do GFC refatorado (GFCR), a fim de que sejam determinados quais vértices foram excluídos do GFCO para formar o GFCR. Os vértices identificados são inseridos em uma lista denominada arranjo. Um laço de repetição é iniciado com a leitura do primeiro vértice do arranjo de vértices excluídos, cuja instrução condicional é denominada condição atual.

Ao iniciar esse laço, é verificado primeiro se a condição atual está no interior de uma expressão composta (com operadores lógicos de conjunção (&&) ou de disjunção (||)), e caso esteja, é feita a sua decomposição para o formato de expressão simples (binária ou unária), separando a condição atual das outras condições, a fim de preparar o código fonte para a eliminação da condição atual.

Em seguida, é procurado no GFCR, um vértice representando uma condição que, através de uma aresta, se liga diretamente com o vértice da estrutura interna da condição atual.

Caso não haja algum vértice condicional com essas características, então a condição atual é considerada como inalcançá-

vel, sendo esta eliminada do texto do código fonte junto com a sua estrutura interna por meio da refatoração.

Caso o vértice condicional com essas características tenha sido encontrado, então a condição atual é considerada como redundante. Antes de refatorar o código fonte, esse mesmo vértice é buscado na lista do GFCO para que seja feita uma comparação entre ele e o seu equivalente da lista do GFCR, a fim de determinar se este possuía um *else* ou não em sua estrutura. Se esse vértice na primeira lista não possuía um else e na segunda lista possui um *else,* faz-se então a refatoração do código fonte excluindo a condição atual, e adicionando em seu lugar um comando *else*. Caso contrário, deve-se apenas excluir a condição atual. Quando a condição atual é considerada como redundante, a sua estrutura interna permanece preservada.

Depois de um desses casos, o laço atual é encerrado para que se inicie um novo laço ao ler o próximo vértice do arranjo. A refatoração do código fonte se encerra quando todos os vértices do arranjo são lidos.

Testes de unidade

O GFC, que é a representação da estrutura de um código fonte, mostra os caminhos de execução possíveis de um algoritmo. Segundo IEEE (1990), cada caminho único representa um caso de teste de unidade a ser desenvolvido baseado no critério de cobertura todos-nós. A quantidade desses caminhos presentes em um algoritmo é equivalente ao valor de sua complexidade ciclomática. Para cobrir as saídas e condições de um algoritmo, a quantidade de casos de teste deve ser correspondente à quantidade de caminhos independentes no grafo da complexidade ciclomática, exceto quando algum desses caminhos não é atingível. Esse tipo de teste tem como foco testar o funcionamento de unidades do código fonte conhecidos como métodos, verificando se cada um deles retorna determinado resultado esperado para uma entrada de dados específica. É utilizada uma tabela para 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. (1982), que tem como premissa utilizar valores próximos à fronteira dos valores estabelecidos pelas condições em que são testados.

Complexity Tool

A abordagem de identificação e eliminação da complexidade ciclomática desnecessária foi implementada em uma ferramenta chamada *Complexity Tool.* Esta ferramenta foi originalmente desenvolvida para prestar auxílio na criação de testes de unidade de software em linguagem de programação Java, a partir de análise estática de código fonte. O seu desenvolvimento se encontra descrito nos trabalhos CAMPOS JUNIOR et al. (2015, 2016a). A ferramenta foi evoluída para identificar o fenômeno da complexidade ciclomática desnecessária, e sugerir uma correção do código fonte através do uso dos GFCs. A implementação dessa abordagem se encontra descrita nos trabalhos CAMPOS JUNIOR et al. (2016b, 2017). A implementação da funcionalidade que permite a ferramenta sugerir uma versão refatorada do código fonte lido, é descrita no trabalho MAGALHÃES et al. (2017). A arquitetura dessa ferramenta está retratada na **Figura 3**.

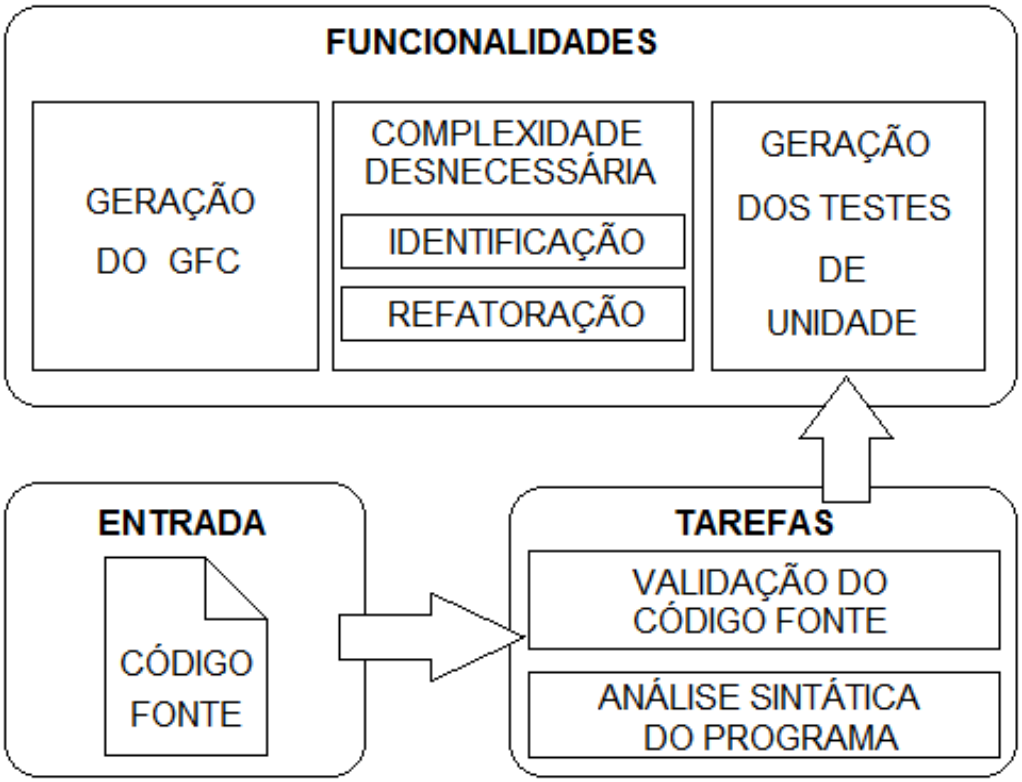

Figura 3. Arquitetura da ferramenta Complexity Tool.

Todas as funcionalidades planejadas estão implementadas, com exceção da geração dos testes de unidade que ainda precisam ser criados manualmente pelo usuário. As demais características são retratadas nas **Figuras 4 e 5**, onde cada uma é marcada com números em azul.

O código fonte analisado é exibido na guia *Source Code* (1). É gerado e exibido na guia *Graph* (4) um GFC correspondente para cada método do código fonte analisado. Quando um nó do GFC é selecionado, suas correspondentes linhas de código fonte são destacadas para ajudar o desenvolvedor a compreender melhor o fluxo de execução desse código. Além disso, no GFC são marcados de azul os caminhos independentes identificados desse fluxo, sendo que cada caminho pode ser selecionado no menu suspenso (6) acima do grafo. Esses caminhos representam os casos de teste de unidade que devem ser criados para alcançar a cobertura das condições de um método. Para ajudar o desenvolvedor na construção desses casos de teste, a guia *Analysis* (2) mostra quais condições precisam ser verdadeiras ou falsas para satisfazer o caminho selecionado. Esta guia também mostra o valor da complexidade ciclomática do método analisado.

Em relação à eliminação da complexidade desnecessária, a guia *Optimized Graph* (5) mostra a versão otimizada do GFC original sem complexidade desnecessária, caso aplicável. Além disso, o desenvolvedor também pode verificar a sugestão refatorada do código fonte analisado na guia *Refactored Code* (3).

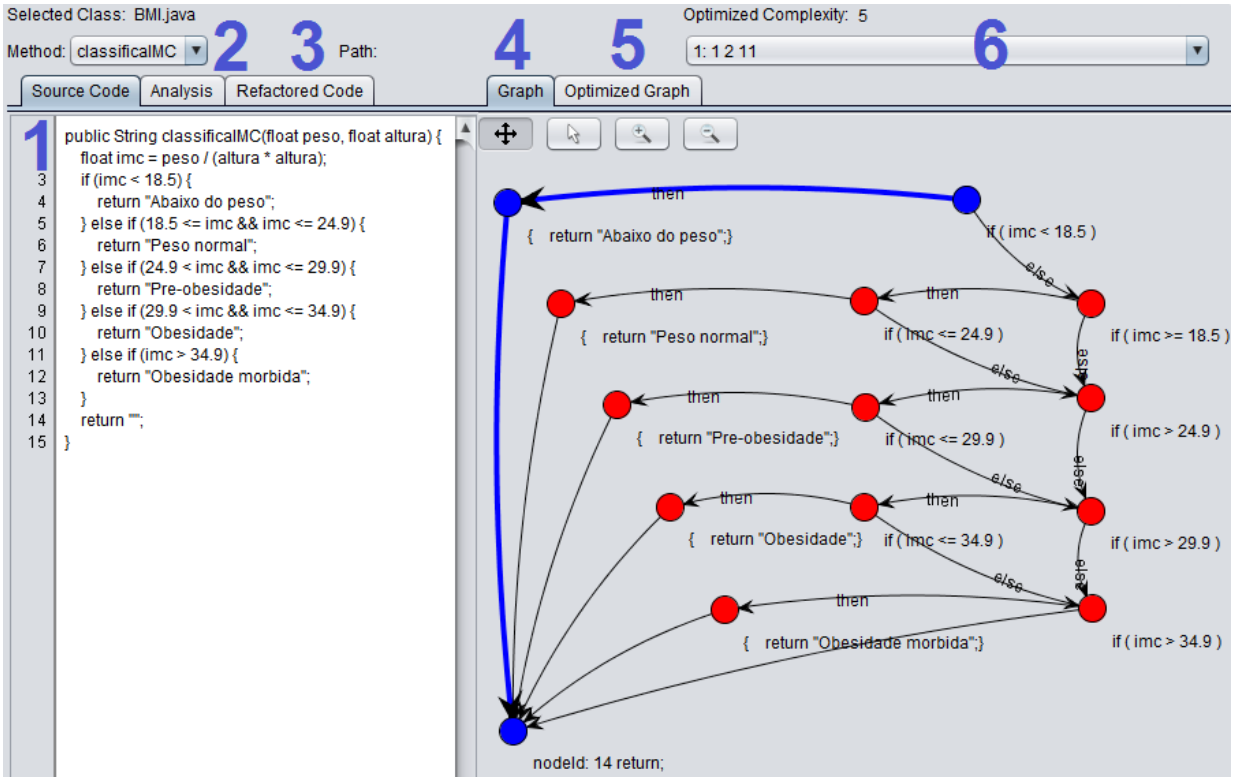

Figura 4. Ferramenta Complexity Tool desenvolvida.

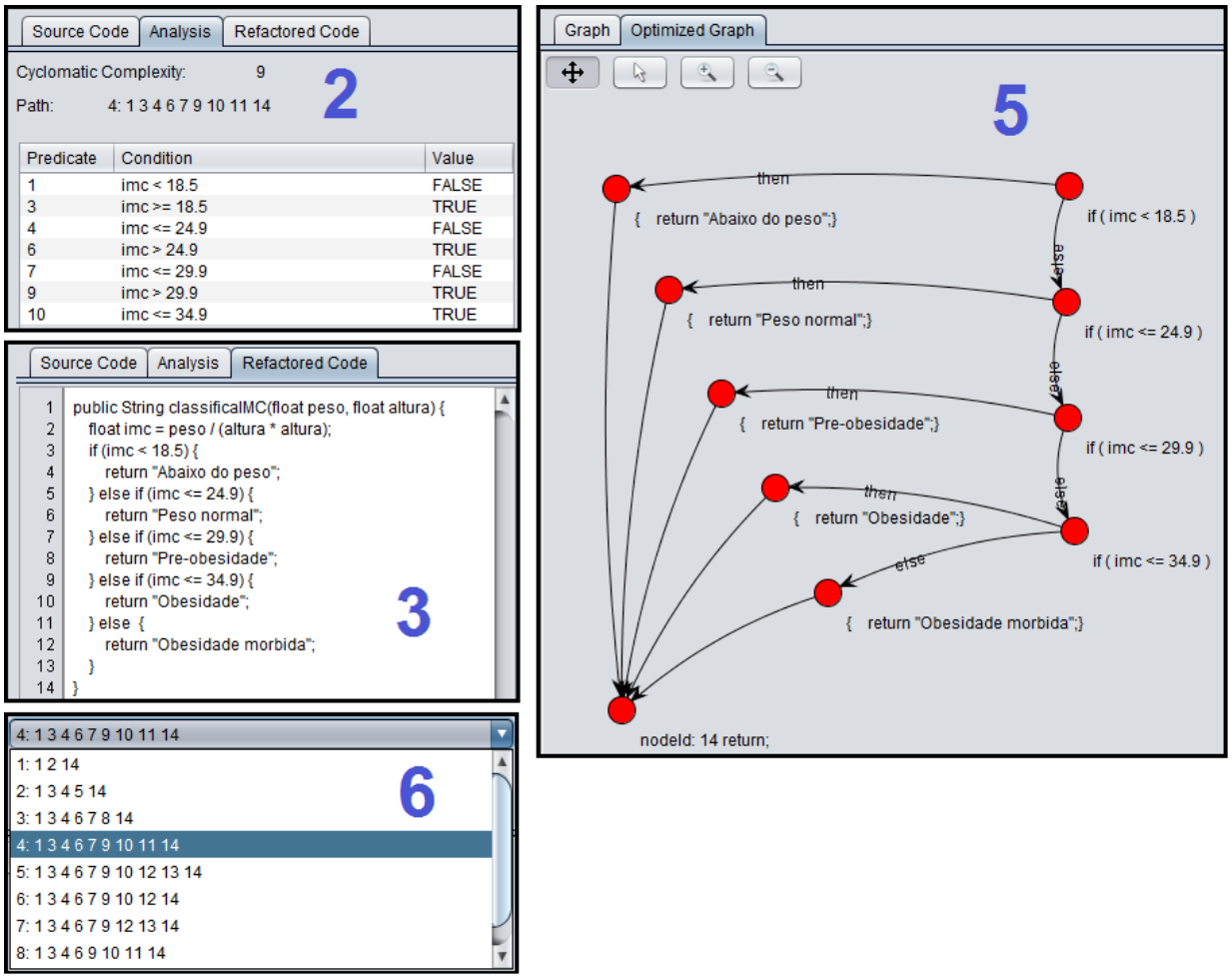

Figura 5. Detalhes de funcionalidades da ferramenta Complexity Tool.

Estudos experimentais

A ferramenta *Complexity Tool* foi avaliada em três estudos experimentais executados em cada etapa do seu desenvolvimento (**Tabela 3**). Os detalhes completos do primeiro, segundo e terceiro estudos se encontram disponíveis nos respectivos trabalhos CAMPOS JUNIOR et al. (2015, 2016b) e MAGALHÃES et al. (2017).

Tabela 3. Estudos experimentais realizados para avaliar as abordagens da ferramenta Complexity Tool.

Estudo	Categoria	Referência	Objetivo
#1	Apoio no planejamento de testes	CAMPOS JUNIOR et al. (2015)	Avaliar o apoio ao desenvolvedor no planejamento e criação dos testes de unidade.
#2	Eliminação da complexidade desnecessária	CAMPOS JUNIOR et al. (2016b)	Avaliar a identificação e eliminação da complexidade desnecessária.
#3	Diminuição do esforço na criação de testes	MAGALHÃES et al. (2017)	Avaliar se o esforço na criação dos testes é diminuído com a eliminação da complexidade desnecessária.

O primeiro estudo teve como objetivo avaliar a ferramenta no quesito de apoio ao desenvolvedor no planejamento de casos de teste de unidade. Neste estudo participaram alunos do curso de Bacharelado em Sistemas de Informação (BSI). Eles cursavam o sétimo período, possuíam conhecimentos na linguagem de programação Java, e já haviam estudado sobre complexidade ciclomática e testes de unidade. Foi proposta uma tarefa para os participantes construírem os casos de teste sobre um código fonte. Os participantes foram divididos aleatoriamente em dois grupos. Em um deles foi permitido o uso da ferramenta, enquanto que o outro serviu como grupo controle.

O segundo estudo experimental teve como objetivo avaliar a abordagem de identificação e eliminação do fenômeno da complexidade ciclomática desnecessária, implementado na ferramenta *Complexity Tool*. Neste estudo, os participantes receberam uma tarefa de programação dividida em duas etapas. Na primeira, o participante foi instruído a desenvolver um programa para calcular e classificar o Índice de Massa Corporal de uma pessoa, baseado em critérios fornecidos na tarefa. Na segunda etapa, foi apresentada aos participantes a ferramenta *Complexity Tool* e pedido que cada um deles executasse manualmente a refatoração do próprio código fonte desenvolvido, para diminuir a sua complexidade ciclomática com base nas informações fornecidas pelos GFCs. O estudo foi executado em duas turmas. Uma delas do curso de graduação em Engenharia Mecatrônica (EM) e matriculada na disciplina de Algoritmos. Seus participantes não possuíam conhecimentos na linguagem de programação Java, portanto receberam um treinamento prévio antes do início de cada atividade. A outra turma cursava o quarto período de graduação de BSI, e seus participantes já possuíam conhecimentos na linguagem Java. Ambas as turmas foram instruídas sobre o conceito de complexidade ciclomática.

O terceiro e último estudo foi executado em duas turmas de períodos diferentes do curso BSI. Os participantes da turma de quinto período não possuíam conhecimentos prévios sobre complexidade ciclomática

e testes de unidade, e por isso receberam um treinamento antes do experimento. A turma de sétimo período já possuía conhecimentos sobre complexidade ciclomática e testes de unidade. O estudo consistiu em reavaliar, na ferramenta *Complexity Tool*, a abordagem avaliada no segundo estudo com a funcionalidade adicional que sugere versão refatorada do código fonte para eliminação da complexidade desnecessária. Também consistiu em observar o esforço de criação dos casos de teste. Os participantes deste estudo receberam duas tarefas para executarem. A primeira delas consistiu no desenvolvimento de um método em linguagem Java, com enunciado semelhante ao utilizado no estudo anterior. Finalizada esta tarefa, os participantes foram instruídos a construir casos de teste de unidade para o método desenvolvido, até que fosse atingido 100% de cobertura. Os participantes foram divididos aleatoriamente em dois grupos. Em um deles, foi permitida a utilização da ferramenta para servir de auxílio na diminuição da complexidade ciclomática do código fonte e na geração dos casos de teste. O outro serviu como grupo de controle.

As turmas em que foram executados os estudos são compostas por alunos graduandos do IF Sudeste MG – Campus Juiz de Fora. Os resultados e a discussão desses estudos estão descritos nas próximas seções.

RESULTADOS

O primeiro estudo realizado sobre a ferramenta *Complexity Tool* tinha por objetivo verificar se a sua utilização apoiava o desenvolvedor na criação dos casos de teste de unidade. Para averiguar a hipótese de que o uso da ferramenta é positivo para com o objetivo, foi amostrada a cobertura obtida por cada participante na construção dos testes para o método apresentado. Ao analisar as amostras, foi constado que as mesmas possuem distribuições normais, pois através do método de *Shapiro-Wilk* apresentaram *p-value* > 0,100, maior que o nível de significância estabelecido (0,05). Foi também constatado que as amostras são homocedásticas por meio do teste de *Levene* que apresentou um *p-value* de 0,203, também maior que o nível de significância. Um método estatístico paramétrico chamado Test T foi utilizado para comparação dos valores de amostras em cada grupo. O *p-value* encontrado (0,0484) é menor que o nível de significância estabelecida, sugerindo que as médias de cobertura entre os grupos são diferentes de forma significativa. Com isso a média de cobertura das amostras foi calculada em 86,91% para o grupo que utilizou a ferramenta, e 62,1% para o grupo controle, evidenciando assim que a utilização da ferramenta foi positiva no apoio à criação de casos de teste para o método fornecido.

Tabela 4. Resultados do primeiro estudo experimental

Participantes	Teste	*P-value*	Resultado
12 da turma de 7º BSI	*Shapiro-Wilk*	Maior que 0,100	Distribuição Normal
	Levene	0,203	Amostras homocedásticas
	Test T	0,0484	Significativo

O segundo estudo tinha por objetivo avaliar a abordagem de eliminação da complexidade ciclomática desnecessária. Foi coletado dos participantes amostras do valor da complexidade ciclomática do método desenvolvido por cada um deles nas duas etapas. Através do teste de *Shapiro-Wilk*, os valores coletados não apresentavam uma distribuição normal, com um *p-value* de 0,048 para a primeira turma e 0,013 na segunda turma. Por esse motivo, um teste de hipóteses não paramétrico cha-

mado *Wilcoxon* pareado foi utilizado, pois cada amostra possui valores de complexidade ciclomática das duas etapas. Foi encontrado *p-value* igual a 0,002 e 0,009 (para cada turma respectivamente) em um nível de significância de 0,05, indicando que os dados coletados são estatisticamente significativos para as etapas nas duas turmas. Esses dados indicam que a abordagem proposta auxilia o usuário a desenvolver código fonte com uma complexidade ciclomática menor do que quando do não uso da abordagem, embora isso não possa ser generalizado para outros contextos que não sejam acadêmicos.

Tabela 5. Resultados do segundo estudo experimental

Participantes	Teste	*P-value*		Resultado
		EM	4º BSI	
19 da turma de EM 10 da turma de 4º BSI	*Shapiro-Wilk*	0,048	0,013	Distribuições não normais
	Wilcoxon pareado	0,002	0,009	Significativo

No último estudo experimental, duas hipóteses foram avaliadas, sendo a primeira delas, a de que o esforço na criação de testes de unidade é diminuído com a eliminação da complexidade desnecessária (H1), e a outra, consistiu em reavaliar a hipótese do estudo anterior para verificar se esta resulta em código fonte com menor complexidade ciclomática (H2). O esforço na criação de testes de unidade foi medido com o tempo de seu desenvolvimento até se atingir 100% de cobertura das instruções. Dados foram coletados no estudo realizado nas duas turmas de BSI. Para avaliar a primeira hipótese, a normalidade das distribuições dos dados coletados foi testada com o método de *Shapiro-Wilk*, cujo *p-value* foi maior que 0,100 para ambas, afirmando então que as amostras possuem distribuição normal. A homocedasticidade das amostras foi testada utilizando o método de *Levene*. Com *p-value* de 0,410 para a turma de sétimo período e 0,685 para a turma de quinto período, verificou-se que as amostras são homocedásticas. Sendo as amostras normais e homocedásticas, utilizou-se o teste de hipóteses paramétrico Test T. Com *p-values* de valor 0,610 (sétimo período) e 0,602 (quinto período) sendo maiores que o nível de significância estabelecida (0,05), os resultados sugerem que não existem indícios que apontem para diminuição do esforço na criação de testes de unidade. A segunda hipótese foi avaliada da mesma forma que a anterior, através do teste de *Shapiro-Wilk* indicando normalidade das amostras com p-value > 0,100, e o teste de Levene indicando a homocedasticidade das amostras com *p-values* de valor 0,262 (sétimo período) e 0,687 (quinto período). O teste paramétrico Test T foi utilizado para obter indícios que sugerem a aceitação da segunda hipótese na turma de sétimo período com *p-value* igual a 0,029 e menor que o nível de significância 0,05, e sua rejeição na turma de quinto período com *p-value* igual a 0,780 e maior que o nível de significância 0,05.

Tabela 6. Resultados do terceiro estudo experimental.

Participantes	Teste	*P-value*		Resultado
		5º BSI	7º BSI	
17 da turma de 5º BSI	*Shapiro-Wilk*	Maior que 0,100		Distribuição Normal
	Levene	0,685 (H1)	0,410 (H1)	Amostras são homocedásticas
		0,687 (H2)	0,262 (H2)	
10 da turma de 7º BSI	*Test T*	0,602 (H1)	0,610 (H1)	1ª hipótese não significativa. 2ª hipótese significativa somente para o 7º BSI.
		0,780 (H2)	0,029 (H2)	

DISCUSSÃO

Foram obtidas evidências que sugerem a validade da hipótese que diz respeito ao apoio no planejamento dos casos de testes de unidade. Os participantes que utilizaram a ferramenta obtiveram maior cobertura do código fonte em relação ao grupo que não utilizou. Caso o uso dessa solução em um meio profissional constar semelhante efeito em relação ao observado no meio acadêmico, pode contribuir com a melhoria da qualidade de sistemas desenvolvidos, uma vez que seus testes teriam maior cobertura e portanto, maior abrangência quanto a possíveis defeitos.

Em relação à diminuição da complexidade ciclomática quando da ocorrência de complexidade ciclomática desnecessária, foram obtidas evidências que sugerem a sua validade no segundo estudo experimental. Os participantes que utilizaram a ferramenta conseguiram realizar a tarefa apresentando menor complexidade ciclomática no código fonte, do que o do desenvolvido pelo grupo controle (participantes que não utilizaram a ferramenta). No último estudo experimental, os resultados também apontaram para sua validade na turma de sétimo período, quando os participantes utilizaram a ferramenta e obtiveram um código fonte com menor complexidade ciclomática. Entretanto, foi rejeitada para a turma de quinto período, pois seus participantes que utilizaram a ferramenta obtiveram códigos fonte de complexidade ciclomática semelhantes aos dos outros participantes que não utilizaram.

Quanto à hipótese sobre a diminuição do esforço dos testes com a eliminação da complexidade desnecessária, não foram encontradas evidências para sua aceitação, sendo necessários estudos mais abrangentes e complexos para avaliar a implicação do uso das abordagens desenvolvidas.

Ameaças à validade dos resultados

Um possível fator que pode ter influenciado o resultado para com a última e a penúltima hipótese avaliada, é a experiência dos participantes. Talvez, por não estarem habituados a criar casos de teste de unidade, tenham tido dificuldade na sua criação e isso pode ter impactado no tempo de desenvolvimento dos casos de teste. E também, por não estarem familiarizados com o conceito de complexidade ciclomática, apresentaram códigos fonte que podem ter impactado na abordagem de detecção da complexidade desnecessária, e assim, não puderam identificar condições redundantes em seus códigos fonte desenvolvidos.

CONCLUSÃO

Conclui-se que o uso de uma ferramenta para detectar ocorrência de complexidade ciclomática desnecessária em código fonte seja importante para estudantes de programação no meio acadêmico; visto que muitos deles, por estarem aprendendo a programar, podem escrever códigos fonte com instruções condicionais desnecessárias, conforme observado nos estudos experimentais. Sendo assim, se este problema não é tratado durante o aprendizado, quando entrarem no mercado de trabalho, podem levar o mau hábito de escrever código fonte complexo sem necessidade, gerando maiores custos de manutenção. Os resultados obtidos nos estudos experimentais realizados sugerem que a complexidade desnecessária pode ser tratada de forma automatizada e que existem ainda oportunidades de melhoria.

Em trabalhos futuros, pretende-se integrar a ferramenta *Complexity Tool* com Ambientes de Desenvolvimento Integrado (IDEs) através de *plug-ins*, visando facilitar a sua utilização por parte dos desenvolvedores. Ainda, pretende-se realizar experimentos e estudos de caso com usuários profissionais e com sistemas de software de maior porte, para que a abordagem possa ser avaliada em um contexto

ABSTRACT: *Programming apprentices may choose to prioritize the functioning of a source code by neglecting its quality, making it difficult to maintain and test. Based on that, a phenomenon called unnecessary structural complexity may occur when a method has a cyclomatic complexity value that can be reduced without changing its behavior. In previous works, different approaches were proposed, including the automated identification and removal of unnecessary cyclomatic complexity in source code through the use of control flow graphs and source code refactoring. Also, an approach to support the development of unit test cases, by specifying and displaying paths to be tested in a control flow graph. The goal of this paper is to aggregate empirical evidence obtained by experimentally evaluating the proposed approaches implemented in a tool named Complexity Tool. Evidence provided by studies performed in previous works, suggests that the approaches significantly impact on developed unit tests coverage increase and on the identification and removal of unnecessary cyclomatic complexity. No evidence was found regarding a possible effort decrease to develop unit tests.*

KEYWORDS: *Cyclomatic Complexity, Software Testing, Source Code Refactoring.*

BIBLIOGRAFIA

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

BALAZINSKA, M.; MERLO, E.; DAGENAIS, M.; LAGUE, B.; KONTOGIANNIS, K. "Advanced Clone-Analysis to Support Object Oriented System Refactoring.", **Proceedings of Seventh Working Conference on Reverse Engineering (WCRE'00)**. IEEE, p. 98-107, Nov. 2000.

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.

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, 2016a.

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, 2016b.

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.

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.

IEEE standard glossary of software engineering terminology. 1990.

LEHMAN, M. M. "Programs, Life Cycles and Laws of Software Evolution", **Proceedings of the IEEE**, vol. 68, no. 9, p.1060–1076, Set. 1980.

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.

MCCABE, T. J. "A complexity measure". In: **IEEE Trans. Software Eng**. Vol. SE-2, N. 4, p. 308-320, 1976.

YU, S.; 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.

ZHANG, M; BADDOO, N. "Performance Comparison of Software Complexity Metrics in an Open Source Project." **Software Process Improvement**, p. 160-174, 2007.

Submetido em: 31/07/2017 Aceito em: 15/03/2018