COMPLEXITY TOOL: UMA FERRAMENTA PARA MEDIR COMPLEXIDADE CICLOMÁTICA DE MÉTODOS JAVA

COMPLEXITY TOOL: A TOOL FOR MEASURING CYCLOMATIC COMPLEXITY IN JAVA METHODS

Heleno de Souza Campos Junior[1], Alisson Fernandes do Prado[2], Marco Antônio Pereira Araujo[3]

Resumo: Este trabalho tem como objetivo apresentar uma ferramenta capaz de gerar grafos da complexidade ciclomática a partir de métodos escritos em linguagem Java. A partir da leitura do código fonte do método, a ferramenta executa um *parsing* com a finalidade de criar uma estrutura de grafos que reflete o fluxo do código fonte analisado. Após o *parsing*, é feita a renderização do grafo para que o usuário possa visualizar os caminhos que a execução do código fonte pode seguir. Através da execução de testes sobre os grafos gerados, a ferramenta demonstrou apresentar corretamente os grafos da complexidade ciclomática.

Palavras-chave: Engenharia de software, grafo de controle de fluxo, qualidade de software, teste de software, métricas de software.

Abstract: *The purpose of this paper is to present a tool that is capable of generating cyclomatic complexity graphs of methods written in Java language. By the scanning of Java code, the tool executes a parsing technique to create a graph structure that reflects the scanned code's control flow. After the parsing, the rendering of the graph is made so the user can visualize the different paths that the code execution can take. It was possible through tests to check that the tool can correctly generate cyclomatic complexity graphs.*

Keywords: *Software engineering, control flow graph, software quality, software testing, software metrics.*

[1] Bolsista IFSUDESTEMG, Bacharelado em Sistemas de Informação, heleno_scj@hotmail.com;
[2] Bolsista IFSUDESTEMG, Bacharelado em Sistemas de Informação, alissonpradodev@gmail.com;
[3] Núcleo de Infórmatica, Campus Juiz de Fora, marco.araujo@ifsudestemg.edu.br.

INTRODUÇÃO

Com o avanço cada vez maior na área de desenvolvimento de software, torna-se essencial para desenvolvedores voltarem sua atenção para a qualidade do que é produzido. Segundo CAVANO e McCALL (1978), para garantir a qualidade de software, vários aspectos individuais devem ser levados em conta, como manutenibilidade, confiabilidade, flexibilidade, corretude, testabilidade, entre outros. Embora esse conceito tenha sido proposto há algum tempo, atualmente ainda é considerado correto, segundo BARNEY et al. (2012), que complementam dizendo que garantir níveis altos de qualidade em aspectos individuais não é suficiente para garantir que um software de maneira geral será de qualidade. Este trabalho busca melhorar a qualidade de software através da apresentação de uma ferramenta cujo objetivo é auxiliar a tarefa de testes de software. De acordo com ORSO e ROTHERMEL (2014) teste de software é uma das abordagens mais usadas para medir e, de certa forma, melhorar a qualidade de software.

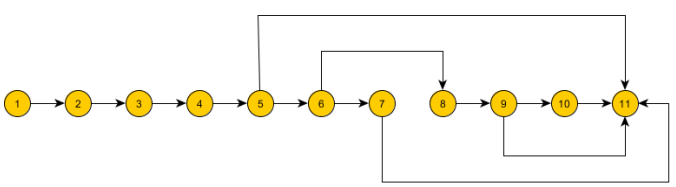
BINDER (1999) e YOUNESSI (2002) destacam que o principal objetivo do teste de software é revelar defeitos no funcionamento de sistemas usando uma quantidade mínima de tempo e esforço. Entretanto, não é viável testar todos os valores possíveis para um código fonte, por questões de tempo. Por conta disso, uma técnica para cobrir todos os caminhos que a execução de um código pode seguir, sem que seja necessário utilizar todas as combinações de valores possíveis, é discutida a seguir.

McCABE (1976) desenvolveu uma métrica para medir a complexidade de módulos, ou seja, unidades de código fonte de software. Tal métrica foi nomeada complexidade ciclomática e é amplamente utilizada, de acordo com MICHURA e CAPRETZ (2005). Uma alta complexidade ciclomática significa que determinada unidade de software é difícil de se entender e dar manutenção. A métrica é baseada no conceito de caminhos que um código fonte pode seguir. Quanto mais caminhos existirem, maior é a dificuldade de um desenvolvedor conseguir abstrair a execução em seu pensamento.

Para efeitos de visualização, McCabe utilizou o conceito de grafos, mais especificamente, grafos de controle de fluxo. Os grafos de controle de fluxo foram propostos por ALLEN (1970), e seu principal objetivo é prover uma representação gráfica de um módulo de um programa em relação a seu código fonte. Trata-se de um grafo direcionado, onde cada nó representa um bloco de comando do programa em análise, e as arestas que os conectam representam os caminhos que os fluxos de execução seguem.

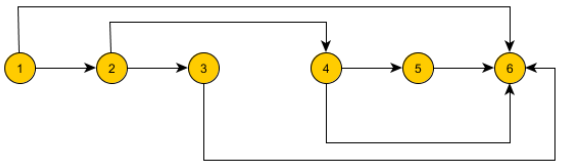
De forma a simplificar sua visualização, os grafos de controle de fluxo podem ser alterados quando for necessário representar a complexidade ciclomática de um código, de

modo a somente apresentar nós de blocos de comando que alterem o fluxo de execução. Tomando por exemplo o grafo de controle de fluxo da Figura 1, que representa o código à sua direita, inicialmente com 11 nós, retirando os redundantes do ponto de vista da complexidade ciclomática e renumerando os blocos, tem-se um grafo da complexidade ciclomática com 6 nós apenas, apresentado na Figura 2.




```
1  public int teste(){
2      int a, b, c;                //1
3      a = 2;                      //2
4      b = 4;                      //3
5      c = 6;                      //4
6      if(a==2){                   //5
7          if(b==5){               //6
8              return 5;           //7
9          } else {                //8
10             if(c==6){           //9
11                 return 3;       //10
12             }
13         }
14     }
15
16 }  //11
```

Figura 1 – Grafo de controle de fluxo.




```
1  public int teste(){
2      int a, b, c;
3      a = 2;
4      b = 4;
5      c = 6;
6      if(a==2){                   //1
7          if(b==5){               //2
8              return 5;           //3
9          } else {
10             if(c==6){           //4
11                 return 3;       //5
12             }
13         }
14     }
15
16 }  //6
```

Figura 2 – Grafo de controle de fluxo simplificado para o cálculo da complexidade ciclomática.

Para se calcular a complexidade ciclomática, McCabe constatou que pode-se contar o número de caminhos independentes presentes no grafo.

Segundo McCABE e WATSON (1996), caminhos indendepentes são caminhos que o fluxo de execução deve seguir para que todas as instruções em uma unidade de código fonte sejam executadas pelo menos uma vez. Cada caminho independente inclui pelo menos uma aresta que não tenha sido percorrida antes do caminho ser definido. No exemplo da Figura 2, ao se observar o grafo de controle de fluxo, é possível obter os seguintes caminhos independentes:

- 1 – 2 – 3 – 4 – 5 – 6 – 8 – 9 – 10 – 11
- 1 – 2 – 3 – 4 – 5 – 6 – 8 – 9 – 11
- 1 – 2 – 3 – 4 – 5 – 6 – 7 – 11
- 1 – 2 – 3 – 4 – 5 – 11

Portanto, 4 caminhos, equivale à complexidade ciclomática de valor 4. A mesma quantidade pode ser obtida ao analisar os caminhos independentes do segundo grafo, que é equivalente ao primeiro, porém simplificado.

O valor da complexidade ciclomática pode ainda ser obtida através de uma fórmula matemática, de acordo com McCabe (1976), representada por: CC = E – N + 2P, onde CC = complexidade ciclomática, E = quantidade de arestas do grafo (*edges*), N = quantidade de nós do grafo (*nodes*), P = quantidade de módulos conectados (neste estudo, P será sempre 1, pois o objetivo deste trabalho é analisar a complexidade ciclomática de um único módulo ou método por vez).

Aplicando a fórmula para os dois grafos do exemplo, tem-se:

- Primeiro grafo (Figura 1):

  CC = 13 – 11 + 2 * 1

  CC = 4

- Segundo grafo (Figura 2):

  CC = 8 – 6 + 2 * 1

  CC = 4

Outros métodos para o cálculo da métrica, ainda segundo McCabe (1976), incluem a quantidade de área fechadas presentes no grafo que representa um método ou módulo somado de 1, sendo o total igual à complexidade ciclomática, ou a quantidade de nós predicados no grafo (nós predicados são nós que representam uma condição no código fonte), somado de um.

Uma vez que se possui o conhecimento dos caminhos independentes de um módulo, é possível saber o que deve ser testado, de modo que se possa obter uma cobertura completa das condições do código fonte. Isso se deve ao fato de a quantidade de caminhos independentes e, por consequência, o valor da complexidade ciclomática equivale no máximo ao total de casos que devem ser testados, exceto quando existem caminhos independentes inatingíveis.

Aplicando esse conceito ao código fonte apresentado nas Figuras 1 e 2, para se obter uma cobertura completa das condições em relação a seus testes, são necessários 4 casos de teste, onde cada caso é configurado para seguir um caminho independente.

O objetivo deste artigo é apresentar uma ferramenta para auxiliar o usuário na criação de casos de testes, através da construção do grafo da complexidade ciclomática de um dado método. Através de uma revisão sistemática na literatura, de acordo com os procedimentos propostos por KITCHENHAM (2004), buscou-se trabalhos relacionados em bases digitais como ACM Digital Library, EI Compendex, IEEE Digital Library, ISI Web of Science, Science Direct e Scopus, resultando em 19 estudos, sendo os 2 principais apresentados a seguir.

SHUKLA e RANJAN (2012) propõem uma nova métrica de complexidade baseada na métrica de McCabe e, para fim de avaliar essa métrica, foi desenvolvida uma ferramenta que calcula os valores para métricas como LOC, Halstead (HALSTEAD, 1977), Complexidade Ciclomática e a própria métrica abordada, sendo que, além do cálculo da Complexidade Ciclomática, é gerado o grafo de controle de fluxo. A ferramenta foi desenvolvida em C# sob o *framework* .NET 2005 e é capaz de analisar módulos na linguagem C, entretanto os autores não citam onde encontrar a ferramenta e mesmo através de buscas, não foram encontrados resultados relevantes.

BOGHDADY et al. (2011) desenvolvem uma técnica de geração de testes através de diagramas de atividade UML (*Unified Modeling Language*), gerando um grafo chamado ADG (Grafo de Diagrama de Atividades) a partir de uma ADT (Tabela de Diagrama de Atividades) e, então, com o uso da Complexidade Ciclomática, trata-se valores máximos e mínimos de casos de testes, e são gerados os testes nos caminhos independentes do ADG.

Através da revisão sistemática, não foi encontrada uma ferramenta que atendesse de forma satisfatória à necessidade apresentada, justificando o desenvolvimento de uma solução própria.

MÉTODOS

A ideia inicial do software desenvolvido neste trabalho era auxiliar o desenvolvedor a visualizar o código fonte escrito em linguagem Java através da exibição do grafo da complexidade ciclomática, possibilitando verificar sua complexidade. Após o início do desenvolvimento, resolveu-se alterar o objetivo para o auxílio da identificação de casos de teste através da exibição do grafo da complexidade ciclomática e seus caminhos independentes. Entretanto, o foco deste trabalho é apresentar a metodologia utilizada para o desenvolvimento da ferramenta.

Sendo o objetivo gerar um grafo a partir da análise de código fonte escrito em linguagem Java, o problema foi dividido em 3 etapas essenciais, *scanner*, onde a ferramenta deve ler uma entrada de dados, no caso, um método escrito na linguagem Java; *parser*, o programa deve processar e entender a estrutura do método lido, gerando uma estrutura de grafo que o represente; *rendering*, a partir da estrutura de grafos, o programa deve renderizar o grafo na tela do usuário para visualização.

Scanner

Para cumprir o que esta etapa requer, foi utilizada uma biblioteca de código aberto chamada Javaparser (2015). Trata-se de um projeto de código aberto no repositório GitHub (2015) e conta com vários colaboradores. A biblioteca possibilita a leitura de arquivos de código fonte escritos na linguagem Java e disponibiliza uma *Abstract Syntax Tree* (AST) representando a classe lida. Com isso, é possível coletar informações cruciais e ainda navegar pelo código a ser analisado.

Parser

Nessa etapa é necessário processar a AST para gerar uma estrutura de grafos que represente o código fonte. Para isso, criou-se um *Recursive Descent Parser* para processar a AST a partir de regras com os seguintes blocos de: *If / Else, Switch / Case, For-each, For, While, Do-While, Return, Break*.

Exemplificando uma dessas regras, para o *parsing* de um comando *If*, deve-se ler o nó da AST correspondente ao *If* e verificar se a condição é simples, por exemplo, “a > 2”, caso não seja, é necessário fazer uma refatoração das condições e transformar o *If* em comandos *If* de condição simples. Quando a condição for simples, deve-se criar um novo nó para o *If* ser representado no grafo da complexidade ciclomática e fazer o *parsing* do *then* deste *If*. Todos os nós que forem criados para o *then* devem estar à esquerda do nó do *If*. Caso o *If* tenha *else*, todos os nós criados devem estar à direita do nó do *If*. Ao final, o nó do *If* é conectado ao seu pai.

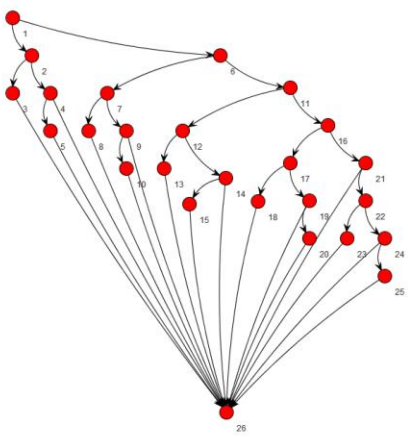
Adotou-se a convenção de direcionamento dos nós para facilitar o entendimento do grafo. Caso a condição do nó atual seja satisfeita, os nós derivados desta condição ficarão à esquerda do atual, caso contrário, à direita.

Renderização

A partir do momento que se tem a estrutura do grafo da complexidade ciclomática, deve-se renderizá-lo na tela para o usuário. Para isso, foi utilizada a biblioteca JUNG (*Java Universal Network/Graph Framework*), também de código aberto, disponível no repositório sourceforge. A biblioteca possui alguns *layouts* prontos para representação de grafos, porém não possui a representação que foi planejada (utilizando filhos à direita e esquerda). A implementação de um algoritmo para ordenação e *layout* do grafo da complexidade ciclomática é um dos trabalhos futuros previstos.

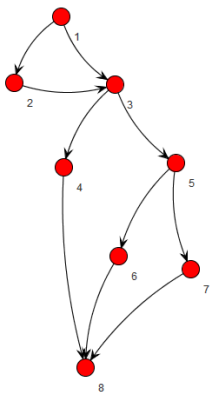
RESULTADOS E DISCUSSÃO

A ferramenta possibilita a geração de grafos completos de códigos fonte escritos na linguagem Java para blocos de comando até sua versão 1.7. As Figuras 3 a 5 apresentam algumas saídas geradas pela ferramenta para métodos com complexidades e características diversas como *if/else* aninhados (Figura 3), presença de *return* no meio do método (Figura 4) e laços de repetição *for* (Figura 5).



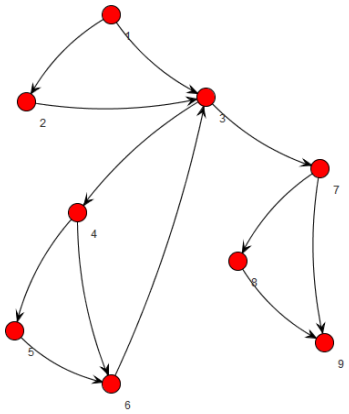
```
1  public float getValorPassagem(){
2          milhas = 300;
3          if (this.milhas <= 400) {   //1
4              if (this.tipoPessoa.equalsIgnoreCase("criança")) {  //2
5                  valorPassagem = 1.5F;       //3
6              } else if (this.tipoPessoa.equalsIgnoreCase("adulto")) {        //4
7                  valorPassagem = 1.8F;           //5
8              }
9          } else if (this.milhas <= 800) {            //6
10             if (this.tipoPessoa.equalsIgnoreCase("criança")) {      //7
11                 valorPassagem = 1.6F;           //8
12             } else if (this.tipoPessoa.equalsIgnoreCase("adulto")) {        //9
13                 valorPassagem = 1.9F;               //10
14             }
15         } else if (this.milhas <= 1500) {           //11
16             if (this.tipoPessoa.equalsIgnoreCase("criança")) {          //12
17                 valorPassagem = 1.7F;               //13
18             } else if (this.tipoPessoa.equalsIgnoreCase("adulto")) {            //14
19                 valorPassagem = 2F;         //15
20             }
21         } else if (this.milhas <= 2500) {               //16
22             if (this.tipoPessoa.equalsIgnoreCase("criança")) {          //17
23                 valorPassagem = 1.9F;               //18
24             } else if (this.tipoPessoa.equalsIgnoreCase("adulto")) {            //19
25                 valorPassagem = 2.15F;              //20
26             }
27         } else if (this.milhas > 2500) {            //21
28             if (this.tipoPessoa.equalsIgnoreCase("criança")) {          //22
29                 valorPassagem = 2F;         //23
30             } else if (this.tipoPessoa.equalsIgnoreCase("adulto")) {            //24
31                 valorPassagem = 2.5F;           //25
32             }
33         }
34         return valorPassagem * this.milhas;  //26
35 }
```

Figura 3 – Saída da ferramenta para o método getValorPassagem.



```
1  public boolean obterAprovacao(int numFaltas, int qtdCreditos) {
2          Float porcentagemPresenca = null;
3          if(numFaltas!=null ){     //1
4              porcentagemPresenca = numFaltas*100/qtdCreditos*36;      //2
5          }
6          if (this.notaTotal >= 60)  {     //3
7              return true;        //4
8          } else {
9              if (this.notaProvaFinal >= 60) {            //5
10                 return true;            //6
11             } else {
12                 return false;       //7
13             }
14         }
15     }
16 }//8
17
18
```

Figura 4 – Saída da ferramenta para o método obterAprovacao.




```
1  public void foo(int x, int y){
2
3      if(x==0){       //1
4          System.out.println("0");        //2
5      }
6
7      for(int i=0; i<10; i++){ //3
8          if(i!=3){  //4
9              System.out.println("Not 3");  //5
10         }
11         System.out.println(i);
12     }  //6
13
14     if(y==2){           //7
15         System.out.println("1");        //8
16     }
17
18 }       //9
```

Figura 5 – Saída da ferramenta para o método foo.

Apesar de funcional, algumas limitações podem ser observadas. A ferramenta só é capaz de analisar código fonte escrito na linguagem Java até a versão 1.7, e condições compostas presentes em blocos de comando que não são *If* (como *for* e *while*) ainda não são suportadas.

Durante o desenvolvimento da ferramenta, alguns desafios de implementação tomaram grande quantidade de tempo para serem solucionados. Um deles foi a refatoração de condições compostas (possuem conectores lógicos) de um *If*, como exemplificado na Figura 6. Várias regras precisaram ser criadas para transformar uma condição composta em condições simples, para poder-se aplicar as regras do *parsing*.

```
1 if(a==1 && b==2){
2     System.out.println("Primeiro");
3 }else{
4     if(a==4 || b==5){
5         System.out.println("Primeiro");
6     }
7 }
8
```

Figura 6 – Código fonte que apresenta *If* com condição composta.

Para fazer o *parsing* do código fonte da Figura 6, é preciso transformá-lo em um algoritmo equivalente, contendo somente condições simples, como apresentado na Figura 7.

```
1  if(a==1){
2      if(b==2){
3          System.out.println("Primeiro");
4      }else if(a==4){
5          System.out.println("Segundo");
6      }else if(b==5){
7          System.out.println("Segundo");
8      }
9  }else{
10     if(a==4){
11         System.out.println("Segundo");
12     }else if(b==5){
13         System.out.println("Segundo");
14     }
15 }
16
```

Figura 7 – Código fonte da condição composta simplificado.

Executar tal transformação não é uma tarefa trivial. Existem muitas possibilidades de derivação. Foram criadas regras com o objetivo de abranger todas as derivações possíveis para condições compostas utilizando conectores lógicos *AND* e *OR* em condições *If*, entretanto o detalhamento dessas regras não estão no escopo deste artigo.

CONCLUSÃO

A ferramenta produz grafos da complexidade ciclomática de forma correta para todos os casos testados durante seu desenvolvimento. Pode auxiliar na identificação de caminhos independentes e consequentemente na geração de casos de teste. Além de propiciar ao usuário desenvolvedor uma visualização gráfica de seu código fonte.

Como trabalhos futuros, visa-se a sugestão de melhoria de código fonte ao desenvolvedor através da análise estática de grafos de controle de fluxo equivalentes; geração automática de casos de teste baseado em caminhos independentes do grafo da complexidade ciclomática; integração com a IDE de desenvolvimento NetBeans e algoritmos para ordenação e visualização de grafos.

Agradecimentos

Os autores agradecem à FAPEMIG e ao IF Sudeste MG pelo apoio financeiro e incentivo à pesquisa intitulada “Desafios em Manutenção de Software Evolutiva: avaliação, impactos e oportunidades de pesquisa”, na qual este trabalho está inserido.

BIBLIOGRAFIA

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

BARNEY, S., PETERSEN, K., SVAHNBERG, M., AURUM, A., & BARNEY, H. Software quality trade-offs: A systematic map. **Information and Software Technology.** Vol. 54, Issue 7, p. 651-662, 2012.

BINDER, R. V. **Testing Object-Oriented Systems**: Models, Patterns, and Tools. Addison-Wesley, 1999.

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.

CAVANO, J., P., & MCCALL, J. A. A framework for the measurement of software quality. **Proceedings of the software quality assurance workshop on Functional and performance issues.** Nova Iorque, ACM, 1978, p. 133-139.

JAVAPARSER, disponível em <http://javaparser.github.io/javaparser/>, acesso em Agosto, 2015.

GITHUB, disponível em <http://github.com/>, acesso em Agosto, 2015.

KITCHENHAM, B. (2004). **Procedures for performing systematic reviews**. Keele, UK: 2004. Relatório Técnico disponível em: http://people.ucalgary.ca/~medlibr/kitchenham_2004.pdf. Acesso em 7 ago 2015.

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

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

MICHURA, J., & CAPRETZ, M. A. Metrics suite for class complexity. In: International Conference on Information Technology, 2005, Las Vegas. **Proceedings ITCC 2005 International Conference on Information Technology: Coding and Computing**, Las Vegas, IEEE, 2005, Vol. 2, p. 404-409.

ORSO, A., & ROTHERMEL, G. Software testing: a research travelogue (2000-2014). In: 36th International Conference on Software Engineering, 2014, Hyderabad. **Proceedings of the on Future of Software Engineering.** Nova Iorque, ACM, 2014, p. 117-132.

SHUKLA, A., & RANJAN, P. A generalized approach for control structure based complexity measure. **Recent Advances in Information Technology (RAIT), 2012 1st International Conference on.** IEEE, 2012 p. 916-921.

YOUNESSI, H.. **Object-Oriented Defect Management of Software**. Prentice-Hall, 2002.