Comparison of Software Complexity Metrics

Ali Athar Khan, Amjad Mahmood, Sajeda M. Amralla and Tahera H. Mirza

Department of Computer Science, University of Bahrain, Sakhir, Bahrain

Received: 25 Sept. 2015, Revised: 1 Dec. 2015, Accepted: 12 Dec. 2015, Published: 1 (January) 2016

Abstract: One of the main problems in software engineering is the inherent complexity. Complexity metric is used to estimate various parameters such as software development cost, amount of time needed for implementation and number of tests required. In this paper, different software complexity models are critically studied and compared. For application, quick sort algorithm is considered. The programs are written in three object oriented languages: C++, Visual Basic and Java. Software complexity for each program is found using the four popular LOC, McCabe, Halstead and Cognitive models. The results are compared.

Keywords: Software Complexity Metric, Line of Code, Cyclomatic Number, Cognitive Complexity.

1. INTRODUCTION

A software metric is the measurement, usually using numerical ratings, to quantify some characteristics or attributes of a software entity [1]. Typical measurements include quality of the source codes, development process and the accomplished applications. Software complexity is a major feature of computer software and is difficult to be measured accurately. It is defined as “the degree to which a system or component has a design or implementation that is difficult to understand and verify” [2]. High complexity may result in more errors and difficulties in maintenance, understandability, modifiability and testing effort [3,4]. Therefore, there has been a great deal of interest in defining appropriate metrics to measure the complexity of the software. Although some useful metrics have been proposed to measure the software complexity [2-9], the current solutions are not enough to settle down this rigorous problem. Both computer researchers and the software engineers are looking for more powerful and effective metric of software complexity [10].

A study was done using 71,917 C/C++ programs to find relations between internal software metrics and metrics of software dependability. It was found that there is a very strong correlation between Lines of Code and Halstead Volume; there is an even stronger correlation between Lines of Code and McCabe’s Cyclomatic Complexity; and none of the internal software metrics makes it possible to discern correct programs from incorrect ones [11].

The aim of this paper is to critically study and compare four commonly used complexity metrics: LOC complexity, McCabe Cyclomatic Complexity, Halstead and Cognitive metric. It is demonstrated through an example that these metrics provide different complexity measures for the same piece of code, thus making it difficult for the software engineer to choose a suitable complexity metric. Therefore, there is a need to develop a unified complexity metric which is more powerful and effective in measuring the software complexity.

The rest of the paper is organized as follow: section 2 presents a review of the software complexity metrics. Section 3 presents complexities of programs written in three object-oriented programming languages using various complexities metrics. A comparison of complexity metrics is presented in section 4 followed by conclusion in section 5.

2. REVIEW OF SOFTWARE COMPLEXITY METRICS

Several methods have been proposed to measure the software complexity. Among the most frequently cited measures are the line of code (LOC), McCabe’s cyclomatic complexity, Halstead’s software metric and Cognitive weights model. We briefly discuss these metrics in this section.

A. Line of Code (LOC) Complexity

The simplest way to measure the complexity of a program is to count the lines of executable code. There is a strong relationship and connection between complexity and size of code which influences the testability and increases the implementation and running time [4]. A
program with larger LOC value takes more time to be developed. Generally, logical lines of code (LLOC) are more useful as compared to physical lines of code. LOC is a good estimate of the complexity of a program, is easy to implement, and does not require the complex operations and calculations [6]. Moreover, counting lines of code can be transformed from a manual operation to an automated operation. However, it is programmer and language dependent and it does not take into consideration the code functionality [12].

B. McCabe’s Cyclomatic Complexity

McCabe defined the cyclomatic number as program complexity [3]. This counts the number of linearly independent paths through a program [11]. First the flow graph of the program is drawn and then the cyclomatic number is found using the following formula [5]:

\[ M = V(G) = e - n + 2p \]  

Where, \( e \) is the number of edges in the graph, \( n \) is the number of nodes and \( p \) is the number of unconnected parts in the graph. It is recommended that no single module has a value of \( M \) greater than 10. Modules which have a value of greater than 10 are considered as complex modules and require much more testing effort. Those modules should be redesigned to reduce value of \( M \) [3]. Cyclomatic number can be easily computed in the development lifecycle during all phases. It improves the testing process, highlights the best areas of concentration for testing and gives the number of recommended tests for software [3,5]. However, the cyclomatic number presents only a partial view of complexity and can be misleading [4].

C. Halstead’s Software Metric

Halstead model defines a program as a collection of tokens, classified as either operators or operands. He proposed the following formulas to find Program Length, Program Vocabulary, Volume, Difficulty, and Effort [4,8]:

Program Vocabulary \((h) = h_1 + h_2\)  
Program Length \((N) = N_1 + N_2\)  
Volume \((V) = N \log_2 h\)  
Potential Volume \((V^*) = (2 + h_2) \log_2(2 + h_2)\)  
Program Level \((L) = V^* / V\)  
Difficulty \((D) = V / V^*\)  

Effort \((E) = V / L\)  

Faults \((B) = V / S^*\)  

Where \(N_i\) is the number of all operators in the code, \(N_2\) is the number of all operands in the code, \(h_j\) is the number of distinct operators in the code, \(h_2\) is the number of distinct operands in the code, \(V^*\) is the minimum volume represented by a built-in function that can perform the task of the entire program and \(S^*\) is the mean number of mental discriminations or decisions between errors - a value estimated as 3,000 by Halstead.

Halsted found that complexity increases as vocabulary and length increase. Moreover, complexity increases as volume increases and program level decreases. Modules which do not have program levels close to 1 are too complex.

Halstead method is easy to implement, simple to calculate, can be used for any programming language, minimizes rate of errors and maintenance effort. However, there are many shortcomings of this model. It has little or no use as a predictive estimating model. It is based on some unreal and imaginary assumptions which cannot be proven easily. For the large programs it is difficult to count the distinct operators and operands.

D. Cognitive Weights Model

Cognitive Weights Model proposed Cognitive Functional Size (CFS) to measure the complexity. CFS is based on cognitive weights. For this, every Basic Control Structure (BC) is assigned a cognitive weight. Either all the BCS’s are in a linear layout or some BCS’s are embedded in others. For the former, sum of the weights of all n BCS’s are added and for the later, cognitive weights of inner BCS’s are multiplied by the weights of external BCS’s [2,7,9]. Figure 1 shows different types of BCSs, the corresponding dedicated weight and Real Time Process Algebra (RTPA) for each one [7].

The total cognitive weight of the software, \(W_c\) is defined as the sum of cognitive weights \(W_c\) of its \(q\) linear blocks composed of individual BCSs. Since each block may consist of \(m\) layers of nesting BCSs, and each layer of \(n\) linear BCSs, \(W_c\) is given as:

\[ W_c = \sum_{j=1}^{q} \prod_{k=1}^{m} \sum_{i=1}^{n} w_c(j,k,i) \]  

If there is no embedded BCS in any of the \(q\) blocks, then \(m=1\) and \(W_c\) is simplified as:

\[ W_c = \sum_{j=1}^{q} \sum_{i=1}^{n} w_c(j,k,i) \]
Cognitive Functional Size is defined as:

\[ CFS = (N_i + N_o) \times W_c \]  \hspace{1cm} (12)

Where \( N_i \) is the number of inputs and \( N_o \) is the number of outputs.

\[ I_k = (\text{Identifiers} + \text{Operands})_k = (ID_k + OP_k) \times IU \]  \hspace{1cm} (13)

Where \( OP_k \) is total number of operators in the \( k^{th} \) LOC of software, \( IU \) is the Information Unit to represent that at least any identifier or operator has one information unit in them.

2. Total Information contained in a software (ICS) is sum of information contained in each line of code:

\[ ICS = \sum_{k=1}^{LOCs} I_k \]  \hspace{1cm} (14)

3. The weighted Information Count of a line of code (WICL) is defined as:

\[ WICL_k = \frac{ICS_k}{LOCs - k} \]  \hspace{1cm} (15)

4. The Weighted Information Count of the Software (WICS) is defined as:

\[ WICS = \sum_{k=1}^{LOCs} WICL_k \]  \hspace{1cm} (16)

5. Cognitive Information Complexity Measure (CICM) is defined as:

\[ CICM = WICS \times W_c \]  \hspace{1cm} (17)

6. Information Coding Efficiency \( (E_i) \) of a software is defined as:

\[ (E_i) = \frac{ICS}{LOCS} \]  \hspace{1cm} (18)

Cognitive weight is a good method since it is easy to understand and calculate, represents the complexity value in terms of small number, and is almost independent of language programmer's experience. Table 1 compares the four metrics with reference to various parameter attributes.

3. Finding Complexity of Programs

In this paper, quick sort algorithm is written in three object oriented languages: C++, Visual Basic and Java (program codes are given in Figure 2, 3 and 4). For each program, all four metrics are found and compared. The metrics which are obtained here are both ‘pure’ object-oriented metrics and metrics proposed for structural programming that could also be applied to object-oriented programming.
TABLE 1: COMPARISON BETWEEN COMPLEXITY METRICS

<table>
<thead>
<tr>
<th>Attribute</th>
<th>LOC</th>
<th>McCabe's</th>
<th>Halsted</th>
<th>Cognitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is it language dependent?</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Is it sensitive to cosmetic changes?</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Is easy to be computed?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Is it predictive estimating model?</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

1. `#include <iostream>
2. using namespace std;
3. int main()
4. {
5.    int A[] = {1, 9, 0, 5, 6, 7, 8, 2, 4, 3};
6.    int length=10;
7.    quickSort(A,0,length-1);
8. }
9. int quickSort(int A[], int F, int L)
10. {
11.    if (F < L)
12.       {
13.           int pivotIndex;
14.           Partition(A,F,L, pivotIndex);
15.           quickSort(A,F,pivotIndex-1);
16.           quickSort(A , pivotIndex+1, L);
17.       }
18.    void partition(int A[], int F, int L, int &pivotIndex)
19.    {
20.        int pivot = A[F];
21.        int lastS1 = F ;
22.        int firstUnknown = F + 1 ;
23.        for (; firstUnknown <= L ;
24.            ++firstUnknown)
25.            {
26.                if ( A[firstUnknown] < pivot)
27.                    ++lastS1;
28.                Swap(A[firstUnknown], A[lastS1]);
29.            }
30.        Swap(A[F], A[lastS1]);
31. }
32. }
33. void Swap(int & x, int & y)
34. {
35.    Int temp = x;
36.    x=y;
37.    y=temp;
38. }
39. Figure 2. Quick sort implementation code in C++ [13]
40. Figure 3. Quick sort implementation code in Java [14]

To find the complexity, we used the cyclomatic and flow graphs for each program. These graph for C++ code are given in Figure 5 and Figure 6 respectively. Similar graphs were also prepared for visual basic and java codes which are not included because of the lack of space. Software complexity metrics are calculated and results are shown in Table 2.
Public Class Form1
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
Dim length as Integer
Dim arr (10) as integer
arr(0)=1
arr(1)=9
arr(2)=0
arr(3)=5
arr(4)=6
arr(5)=7
arr(6)=8
arr(7)=2
arr(8)=4
arr(9)=3
length = 10
Quicksort(arr, 0, length - 1)
End Sub
Public Sub Quicksort(list()As Integer, ByVal min As Long, ByVal max As Long)
Dim med_value As Long
Dim hi As Long
Dim lo As Long
Dim i As Long
If min >= max Then Exit Sub
i = Int((max - min + 1) * Rnd + min)
med_value = list(i)
list(i) = list(min)
lo = min
hi = max
Do
Do While list(hi) >= med_value
hi = hi - 1
If hi <= lo Then Exit Do
Loop
If hi <= lo Then
list(lo) = med_value
End If
list(lo) = list(hi)
Exit Do
Loop
Quicksort(list, min, lo – 1)
Quicksort(list, lo + 1, max)
End Sub
End Class
Figure 4: Quick sort implementation code in VB.Net [15]

Figure 5: Cyclomatic graph of C++ code
4. COMPLEXITY COMPARISON OF PROGRAMS

The complexity of the three programs are summarized in Table 3. It can be seen that the VB program has the highest LOC and LLOC values which are 75 and 24, respectively. Hence it is more complex and takes more time to be developed. The McCabe Cyclomatic numbers of all the three programs are greater than 10. These are complex modules and require much more testing time.

Here also VB program is the worst. It is observed that one of our calculated Halstead values which is Program Level, L of all the three programs are not close to 1 and are very low. This indicates that all the programs are too complex. The Halstead difficulty level D for C++ program has the highest value. This shows that C++ is the worst and hence more complex compared to other two programs. It is found that VB program has the highest Halstead effort value. VB program has also the highest Halstead Faults number which is 0.508. The Cognitive CICM value of C++ program is 78.009 which is the highest. This means that the amount of information contained in the software is more as compared to others.

<table>
<thead>
<tr>
<th>Complexity Method</th>
<th>C++ Program</th>
<th>VB Program</th>
<th>Java Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>48</td>
<td>75</td>
<td>52</td>
</tr>
<tr>
<td>LLOC</td>
<td>17</td>
<td>24</td>
<td>18</td>
</tr>
<tr>
<td>McCabe method</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M = e-n+2p</td>
<td>M=28-23+2*3 = 11</td>
<td>M=60-51+2*5 = 19</td>
<td>M=30-24+2*3 =12</td>
</tr>
<tr>
<td>Halstead method</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h = h₁ + h₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N = N₁ + N₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V = N log₂ h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V’ = (2 + h₁) log₂ (2 + h₂)</td>
<td>V = 1100.47</td>
<td>V’ = 1525</td>
<td>V’ = 1075.08</td>
</tr>
<tr>
<td>L = V’ / V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D = V/N’</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E = V/L</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B = V / S’</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cognitive method</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOC</td>
<td>48</td>
<td>75</td>
<td>52</td>
</tr>
<tr>
<td>Total # of identifiers</td>
<td>66</td>
<td>86</td>
<td>58</td>
</tr>
<tr>
<td>Total # of operators</td>
<td>9</td>
<td>17</td>
<td>10</td>
</tr>
<tr>
<td>Wc</td>
<td>28</td>
<td>23</td>
<td>39</td>
</tr>
<tr>
<td>WICS</td>
<td>2.786</td>
<td>2.2682</td>
<td>1.9934</td>
</tr>
<tr>
<td>CICM</td>
<td>78.0094</td>
<td>52.170</td>
<td>77.743</td>
</tr>
</tbody>
</table>

Figure 6: Flow graph of C++ code
TABLE 3: SOFTWARE COMPLEXITY OF PROGRAMS

<table>
<thead>
<tr>
<th>Complexity Method</th>
<th>C++ Program</th>
<th>VB Program</th>
<th>Java Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC / LLOC</td>
<td>48</td>
<td>75</td>
<td>52</td>
</tr>
<tr>
<td>McCabe Cyclomatic number</td>
<td>17</td>
<td>24</td>
<td>18</td>
</tr>
<tr>
<td>Halstead method</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program level</td>
<td>0.094</td>
<td>0.126</td>
<td>0.102</td>
</tr>
<tr>
<td>Difficulty</td>
<td>10.56</td>
<td>7.91</td>
<td>9.78</td>
</tr>
<tr>
<td>Effort</td>
<td>11707.13</td>
<td>12103.17</td>
<td>10540</td>
</tr>
<tr>
<td>Faults</td>
<td>0.37</td>
<td>0.508</td>
<td>0.358</td>
</tr>
<tr>
<td>Cognitive method</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CICM</td>
<td>78.0094</td>
<td>52.170</td>
<td>77.743</td>
</tr>
</tbody>
</table>

5. CONCLUSION

The software complexity plays a vital role to reduce the effort to build and maintain software, and to enhance the effectiveness of testing and software quality. The more complex the software solution the more errors it generates. In this paper, four software metrics, their importance, weaknesses and strengths are studied. Then the LOC, McCabe’s cyclomatic, Halstead and Cognitive Weight metrics were calculated for three programs written in object oriented languages to implement the quick sort algorithm.

According to LOC metric and McCabe’s number $M$, programs in C++, Java and VB are in increasing order of complexity. All the three are too complex as their $M$ values are greater than 10. According to Halstead and cognitive weight metrics, program in C++ has complexity higher than that of program in Java and program in Java has complexity higher than that of program in VB. Thus it is not possible to say which program is more complex because different software metric gives different result. However, C++ is better in aspect of size, testing time, VB is better in aspect of difficulty and Java is better in aspect of effort.

The reason that not all metrics are giving the same results is that each method covers a part and considers some parameters while leaving some others. Therefore, a combination of metrics is recommended to be used to measure the complexity.

REFERENCES

Biographical notes

Dr. Ali Athar Khan received a B. Tech (Hons) EE from Indian Institute of Technology, Khargpur, India; his M.E.ECE from Indian Institute of Science, Bangalore, India; and PhD in Computer Science from Indian Institute of Technology, Roorkee, India. He is an Associate Professor in the Department of Computer Science at the University of Bahrain. He has published His research interests include Petri net, parallel processing and software Engineering.

Tahera H. Mirza got her B.Sc. in computer science from the department of computer science, university of Bahrain, Bahrain.

Dr. Amjad Mahmood is an associated professor in the department of computer science in the University of Bahrain, Bahrain. He received his M. Sc. in computer science from QAU, Pakistan in 1989 and a Ph.D., also in computer science, from the University of London, UK in 1994. Before joining the University of Bahrain in 2000, he worked with King Saud University, Saudi Arabia (1999–2000), Philadelphia University, Jordan (1997–1999) and National University of Science and Technology, Pakistan (1995–1997). He has published over 65 research papers in international journals and conferences. His research interests include distributed computing, real-time systems, meta-heuristics, World Wide Web, and software engineering.

Sajeda M. Amralla got her B.Sc. in computer science from the department of computer science, university of Bahrain, Bahrain.