



# 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, modification 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 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]:

$$M = V(G) = e - n + 2p \quad (1)$$

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  $M$  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]:

$$\text{Program Vocabulary } (h) = h_1 + h_2 \quad (2)$$

$$\text{Program Length } (N) = N_1 + N_2 \quad (3)$$

$$\text{Volume } (V) = N \log_2 h \quad (4)$$

$$\text{Potential Volume } (V^*) = (2 + h_2) \log_2 (2 + h_2) \quad (5)$$

$$\text{Program Level } (L) = V^* / V \quad (6)$$

$$\text{Difficulty } (D) = V / V^* \quad (7)$$

$$\text{Effort } (E) = V / L \quad (8)$$

$$\text{Faults } (B) = V / S^* \quad (9)$$

Where  $N_1$  is the number of all operators in the code,  $N_2$  is the number of all operands in the code,  $h_1$  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 (BCS) 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:

$$W_c = \sum_{j=1}^q \prod_{k=1}^m \sum_{i=1}^n w_c(j, k, i) \quad (10)$$

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

$$W_c = \sum_{j=1}^q \sum_{i=1}^n w_c(j, k, i) \quad (11)$$

Cognitive Functional Size is defined as:

$$CFS = (N_i + N_o) \times W_c \quad (12)$$

Where  $N_i$  is the number of inputs and  $N_o$  is the number of outputs.











Definition of BCSs and their equivalent cognitive weights ( $W_i$ )			
Category	BCS	Structure	$W_i$ RTPA notation
Sequence	Sequence (SEQ)		1 P→Q Note: Consider only one sequential structure in a component
Branch	If-then-else (ITE)		2 (?exp BL = T)→P   (?- )→Q
Case	(CASE)		3 ? exp RT = 0→P <sub>0</sub>   1→P <sub>1</sub>   ...   n-1→P <sub>n-1</sub>   else→O
Iteration	For-do (R <sub>i</sub> )		3 R <sub>i-1</sub> <sup>+</sup> (P(i))
	Repeat-until (R <sub>u</sub> )		3 R <sub>i-1</sub> <sup>exp BL = T</sup> (P)
	While-do (R <sub>w</sub> )		3 R <sub>i-0</sub> <sup>exp BL = T</sup> (P)
Embedded component	Function call (FC)		2 P <sub>i</sub> F Note: Consider only user-defined functions
	Recursion (REC)		3 P <sub>i</sub> P
Concurrency	Parallel (PAR)		4 P    Q
	Interrupt (INT)		4 P    @ (eS ↗ Q ↘ @)

Figure 1: BCSs and their cognitive weights [7]

It has been established that the larger is the cognitive complexity, the larger the amount of information contained in the software. Programs having higher code cognitive efficiency use fewer lines of code to implement more complex software. However, the cognitive weight method was modified to become more efficient and easier to calculate [2]. The following concepts were added.

1. Information contained in one line of code is the number of all operators and operands in that line of code. Thus Information contained in  $k^{th}$  line of code is given by:

$$I_k = (\text{Identifiers} + \text{Operands})_k = (ID_k + OP_k) IU \quad (13)$$

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:

$$ICS = \sum_{k=1}^{LOCS} I_k \quad (14)$$

Where,  $I_k$  is Information contained in  $k^{th}$  line of code and LOCs is total lines of code in the software.

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

$$WICL_k = ICS_k / [LOCs - k] \quad (15)$$

Where,  $WICL_k$  is Weighted Information Count for the  $k^{th}$  line and  $ICS_k$  is information contained in a software for the  $k^{th}$  line.

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

$$WICS = \sum_{k=1}^{LOCS} WICL_k \quad (16)$$

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

$$CICM = WICS * W_c \quad (17)$$

6. Information Coding Efficiency ( $E_I$ ) of a software is defined as:

$$(E_I) = ICS / LOCS \quad (18)$$

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

Attribute	LOC	McCabe's	Halsted	Cognitive
Is it language dependent?	Yes	No	Yes	No
Is it sensitive to cosmetic changes?	Yes	No	No	No
Is easy to be computed?	Yes	Yes	No	Yes
Is it predictive estimating model?	No	Yes	No	Yes

```

1  #include<iostream>
2  using namespace std;
3
4  int main()
5  {
6
7
8      int A[] = {1, 9, 0, 5, 6, 7, 8, 2, 4, 3};
9      int length=10;
10     quickSort(A,0,length-1);
11 }
12 void quicksort(int A[], int F, int L)
13 {
14     int pivotIndex;
15
16     if (F < L)
17     {
18         Partition(A,F,L, pivotIndex);
19         quicksort( A,F,pivotIndex-1);
20         quicksort(A , pivotIndex+1, L);
21     }
22 }
23
24 void partition(int A[] , int F, int L, int &
25                pivotIndex)
26 {
27     int pivot = A[F];
28     int lastS1 = F ;
29     int firstUnknown = F + 1 ;
30
31     for ( ; firstUnknown <= L ;
32          ++ firstUnknown)
33     {
34         if ( A[firstUnknown] < pivot)
35             { ++lastS1;
36               Swap(A[firstUnknown], A[lastS1]);
37             }
38     }
39     Swap(A[F], A[lastS1]);
40     pivotIndex=lastS1;
41 }
42
43 void Swap(int & x, int & y)
44 {
45     Int temp = x;
46     X=y;
47     Y=temp;
48 }

```

Figure 2. Quick sort implementation code in C++ [13]

```

1  Public Class QS
2  {
3
4      Public static void main (String[] args)
5
6      {
7          int arr[10] = {1, 9, 0, 5, 6, 7, 8, 2, 4, 3};
8          int length = 10;
9          int result[]= new int (10);
10         result =QuickSort (arr, 0, length-1);
11     }
12
13     Static int [] QuickSort ( int [] a, int l, int r )
14     {
15         if ( l < r )
16
17         {
18             int i=;
19             int j=r;
20             int k = (int) (( l+r ) / 2);
21             int pivot = a[k];
22
23             do
24             {
25                 while (a[i].less (pivot))
26                     i++;
27                 while (pivot.less (a[j]))
28                     j--;
29
30                 if ( i <= j )
31                 {
32                     int t = a[i] ;
33                     a[i] = a[j] ;
34                     a[j]=t;
35
36                     i++;
37                     j++;
38                 }
39             }
40             while (i < j)
41
42             a= QuickSort (a, l, j );
43             a= QuickSort (a, i, r );
44         }
45     }
46     return a;
47 } // end of QuickSort
48
49 } // end of class QS
50
51
52 } // end of class QS

```

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.



```

1 Public Class Form1
2
3 Private Sub Form1_Load(ByVal sender As System.Object, ByVal
   e As System.EventArgs) Handles MyBase.Load
4
5 Dim length as integer
6 Dim arr (10) as integer
7
8 arr(0)=1
9 arr(1)=9
10 arr(2)=0
11 arr(3)=5
12 arr(4)=6
13 arr(5)=7
14 arr(6)=8
15 arr(7)=2
16 arr(8)=4
17 arr(9)=3
18
19 length = 10
20
21 Quicksort(arr, 0, length-1)
22
23 End Sub
24
25
26 Public Sub Quicksort(list()As Integer,ByVal min As Long, ByVal
   max As Long)
27 Dim med_value As Long
28 Dim hi As Long
29 Dim lo As Long
30 Dim i As Long
31
32 If min >= max Then Exit Sub
33 i = Int((max - min + 1) * Rnd + min)
34 med_value = list(i)
35 list(i) = list(min)
36 lo = min
37 hi = max
38
39
40 Do
41 Do While list(hi) >= med_value
42 hi = hi - 1
43 If hi <= lo Then Exit Do
44 Loop
45
46 If hi <= lo Then
47 list(lo) = med_value
48 End If
49
50 list(lo) = list(hi)
51 lo = lo + 1
52
53
54 Do While list(lo) < med_value
55 lo = lo + 1
56 If lo >= hi Then Exit Do
57 Loop
58
59 If lo >= hi Then
60 lo = hi
61 list(hi) = med_value
62 End If
63
64 End If
65
66 list(hi) = list(lo)
67
68 Exit Do
69 Loop
    
```

```

70
71 Quicksort (list, min, lo - 1)
72 Quicksort (list, lo + 1, max)
73 End Sub
74
75 End Class
    
```

Figure 4: Quick sort implementation code in VB.Net [15]

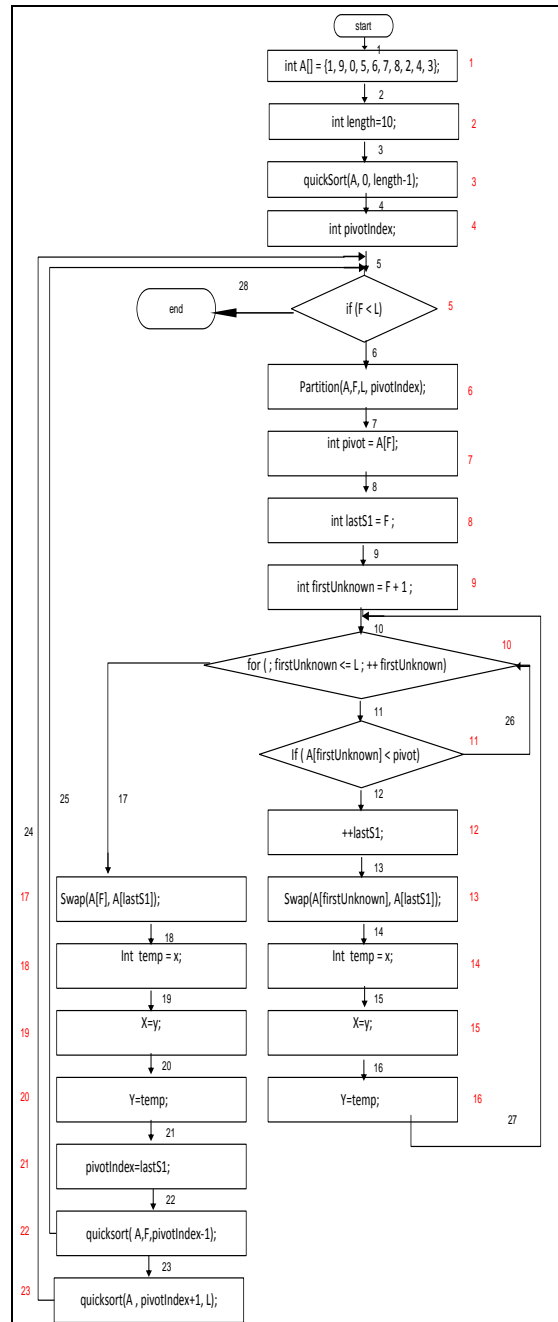


Figure 5: Cyclomatic graph of C++ code

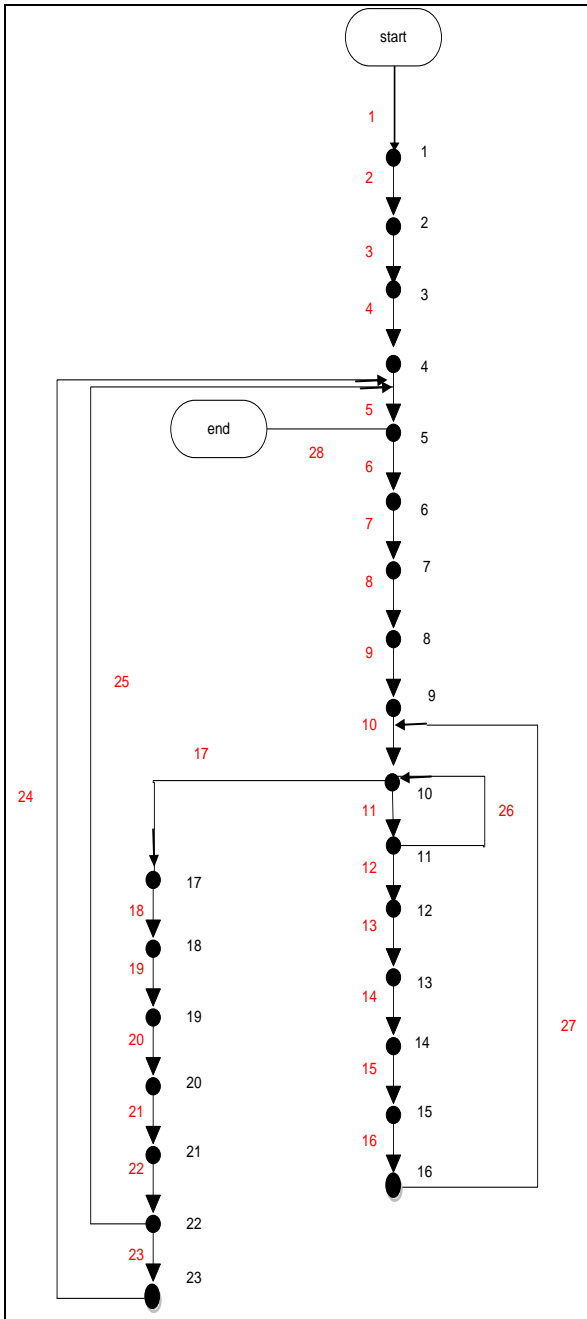


Figure 6: Flow graph of C++ code

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 2: COMPLEXITY CALCULATIONS

Complexity Method	C++ Program	VB Program	Java Program
<b>LOC</b>	48	75	52
<b>LLOC</b>	17	24	18
<b>McCabe method</b> $M = e - n + 2p$	$M = 28 - 23 + 2 * 3 = 11$	$M = 60 - 51 + 2 * 5 = 19$	$M = 30 - 24 + 2 * 3 = 12$
<b>Halstead method</b>	$N_1 = 142$ $N_2 = 57$ $h_1 = 25$ $h_2 = 21$ $h = 46$ $N = 199$ $V = 1100.47$ $V^* = 104.19$ $L = 0.094$ $E = 11707.13$ $B = 0.37$	$N_1 = 155$ $N_2 = 89$ $h_1 = 41$ $h_2 = 35$ $h = 76$ $N = 244$ $V = 1525$ $V^* = 192.77$ $L = 0.126$ $E = 12103.17$ $B = 0.508$	$N_1 = 133$ $N_2 = 53$ $h_1 = 33$ $h_2 = 22$ $h = 55$ $N = 186$ $V = 1075.08$ $V^* = 109.92$ $L = 0.102$ $E = 10540$ $B = 0.358$
<b>Cognitive method</b>			
LOC	48	75	52
Total # of identifiers	66	86	58
Total #r of operators	9	17	10
Wc	28	23	39
WICS	2.786	2.2682	1.9934
CICM = WICS * Wc	78.0094	52.170	77.743



TABLE 3: SOFTWARE COMPLEXITY OF PROGRAMS

Complexity Method	C++ Program	VB Program	Java Program
<b>LOC</b>	48	75	52
<b>LLOC</b>	17	24	18
McCabe Cyclomatic number	11	19	12
<b>Halstead method</b>			
Program level	0.094	0.126	0.102
Difficulty	10.56	7.91	9.78
Effort	11707.13	12103.17	10540
Faults	0.37	0.508	0.358
<b>Cognitive method</b>			
CICM	78.0094	52.170	77.743

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

- [1] B. Jayanthi, and K. KrishnaKumari, "Brief study on Software quality metrics and software complexity metrics in Web application," International Journal of Engineering Sciences & Research Technology, vol. 3, no. 12, 2014, pp. 441,444.
- [2] D. S. Kushwaha, and A. K. Misra, "A modified cognitive information complexity measure of software," ACM SIGSOFT Software Engineering Notes, vol 31, no. 5, 2006, pp. 1-4.
- [3] A. H. Watson, and T. J. McCabe, "Structured testing: a testing methodology using the cyclomatic complexity metric," Computer Systems Laboratory, National Institute of Standards and Technology, September 1996.
- [4] S. Nystedt, and C. Sandros, "Software complexity and project performance," School of Economics and Commercial Law at the University of Gothenburg, 1999.
- [5] A. and Sharma, D.S. Kushwaha, "A Complexity measure based on requirement engineering document," Journal of Computer Science and Engineering, vol 1, no. 1, 2010, pp. 112-117.
- [6] C. Jones, "Strength and weaknesses of software metrics," Version 5, 2006.
- [7] J. Shao, and Y. Wang, "A new measure of software complexity based on cognitive weights," Can. J. Elect. Comput. Eng., vol. 28, no. 2, 2003, pp. 69-74.
- [8] M. H. Halstead, "Elements of Software Science," New York : Elsevier North, 1976.
- [9] S. Misra, "A complexity measure based on cognitive weights," International Journal of Theoretical and Applied Computer Sciences, vol. 1, no. 1, 2006, pp. 1-10.
- [10] S. Yu, and S. Zhou, "A survey on metric of software complexity," In The 2nd IEEE International Conference on Information Management and Engineering (ICIME), 2010, pp. 352-356.
- [11] M. J. P. van der Meulen, "Correlations between internal software metrics and software dependability in a large population of small C/C++ programs," 18th IEEE International Symposium on Software Reliability Engineering, 2007, pp. 203-206
- [12] E. E. Mills, "Software metrics," SEI Curriculum Module SEI-CM-12-1.1, Seattle University, December 1988.
- [13] F. M. Carrano, "Data abstraction and problem solving with C++," New Jersey: Pearson, 5th Edition, 2006.
- [14] J. Bishop, "Java gently, Programming Principles Explained," England: Addison-Wesley, 3rd Edition, 2001.
- [15] "http://www.bigresource.com," Oct 2010.



### 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.