# Enhancements to an OO Metric: CB Measure

D. I. De Silva[1*], S. R. Kodituwakku[2], A. J. Pinidiyaarachchi[2], N. Kodagoda[1]

[1] Faculty of Computing, Sri Lanka Institute of Information Technology, Malabe, Sri Lanka.
[2] Faculty of Science, University of Peradeniya, Peradeniya, Sri Lanka.

* Corresponding author. Tel.: +94716512719; email: dilshan.i@sliit.k
Manuscript submitted August 30, 2017; accepted October 20, 2017.
10.17706/jsw.13.1.72-81

**Abstract:** Due to the wide usage of the object-oriented paradigm as a development paradigm many researches have proposed metrics to measure the complexity of object-oriented programs. The proposed object-oriented metrics can be divided into two categories based on the main aspect they have considered: metrics based on object-oriented aspects and metrics based on the cognitive aspects. Majority of the metrics which belong to the latter category have relied on a maximum of three complexity factors to derive the complexity of a program. CB measure is one of the few metrics that has considered four or more complexity factors to measure the complexity associated with a software program. However, there exists some other factors that could be considered by the CB measure to make it a more practically applicable measure. Such factors were proposed by the authors in a previous study. This paper demonstrates how those factors can be incorporated to the CB measure. In addition, it validates the practical applicability of the modified CB measure.

**Key words:** Object-oriented paradigm, software complexity, complexity metrics, CB measure

## 1. Introduction

Towards the later part of the 1980s, the introduction of object-oriented (OO) paradigm brought about a different means of looking at programming with the use of objects. Objects are the instances of classes. They can consist of fields and methods. Fields are used to store data of objects and methods are used to perform the actions of them.

The wide usage of the object-oriented paradigm as a development paradigm led the way for researches to propose metrics to measure the complexity of object-oriented programs. The proposed object-oriented metrics can be mainly divided into two categories based on the aspects they have considered: metrics based on object-oriented aspects and metrics based on the cognitive aspects. Chidamber and Kemerers' suite of six metrics [1], Chen and Luis' suite of eight metrics [2], MOOD metrics [3], and Li's suite of six metrics [4] are some of the OO metrics which were proposed based on the OO aspect. Class complexity measure [5], weighted class complexity measure [6], cognitive code complexity measure [7], CB measure [8] are some of the OO metrics which were proposed based on the cognitive aspect.

Majority of the cognitive based complexity metrics have relied on a maximum of three complexity factors to compute the complexity of an entire program. CB measure is one of the few metrics that has considered four or more complexity factors to measure the complexity associated with a software program. However, there exists some other factors that could be considered by the CB measure to make it a more practically applicable measure. A previous study proposed some of such factors that could be incorporated to the CB

measure [9]. This study demonstrates how those factors can be incorporated to the CB measure. Additionally, it validates the practical applicability of the modified CB measure.

The next section discusses about the CB measure. Section 3 discusses how the previously proposed factors are accommodated in the modified CB measure. Section 4 discusses about the practical applicability of the modified CB measure. Section 5 concludes the paper.

## 2. CB Measure

Chhillar and Bhasin believed that software complexity is a multidimensional attribute of software and thereby it cannot be measured by considering a single factor. With this in mind, they proposed the CB measure in 2011. It was based on four significant factors that contribute to the complexity of software:

*Inheritance level of classes (Wi):* The degree of understanding a statement increases with the level of inheritance of classes. Taking this into account Chhillar and Bhasin assigned a weight of zero for executable statements in the bass class, a weight of one for the executable statements which are at the first derived class, 2 for the statements at the next derived class. Similarly, the weight allocated for the statements increased by one for each derived class.

*Type of control structures in classes (Wc):* Chhillar and Bhasin believed that the complexity added to a class/program by a control structure differs depending on its type. Thus, a weight of zero was assigned to sequential statements, one for conditional control structures such as if-else conditions, two for iterative control structures such as for, while, and do-while loops, and n for switch-case statements with n cases.

*Nesting level of control Structures (Wn):* The understandability of a program decreases with the number of nesting levels of control structures. As a result, the complexity of that program will increase. Taking this into consideration Chhillar and Bhasin assigned a weight of zero for sequential statements, one for statements which are at the outer most level of nesting, two for statements which are at the next inner level of nesting. Similarly, the weight allocated for the statements was increased by one for each level of nesting.

*Size of a class in terms of token count:* With the belief that the complexity of a class or program increases along with the size of it, Chhillar and Bhasin considered the size of a class or program to be the final factor of their measure. The size of an executable statement was calculated in terms of the operators, operands, methods/functions, and strings in that statement.

Considering the above-mentioned factors, a weighted complexity measure for an object-oriented program P was suggested as:

$$C_w(P) = \sum_{j=1}^{n} (S_j) * (W_t)_j$$

where

$C_w$(P) = Weighted complexity measure of program P

$S_j$ = Size of $j^{th}$ executable statement in terms of tokens count

$n$ = Total number of executable statements in program P

$j$ = Index variable

$W_t = W_n + W_i + W_c$

## 3. Accommodating the New Factors

In a previous work [9], the authors presented additional factors that could be considered by the CB measure to make its complexity calculation more accurate. The present paper demonstrates how the CB measure can be modified to incorporate those factors. Thus, in addition to considering the complexities that arises due to inheritance level, type and nesting level of control structures and size, modified CB measure

(MCB) considers the complexity that arises due to dynamic memory access, recursive methods, threads, pointers and references, exceptions, compound conditional statements.

As there is a minor difference between dynamic memory access and pointers and references, with regards to regular data allocation and variable retrieval, they can be considered in an equivalent manner in the MCB formula by adding a constant value of two for the *, & and new operators. These can be considered by the size (S) attribute. In languages such as C++ when the memory is dynamically allocated using the "new" operator it should be deallocated using the "delete" operator. Thus, when a "delete" operator is present in a statement, like the other operators mentioned above, a value of two is added when computing values for the size (S) attribute. Fig. 1 shows a sample program which was written to demonstrate the effect that dynamic memory access and pointers and references have on the complexity of a program. Table 1 and Table 2 clearly demonstrates how complexity is calculated for the program given in Fig. 1, using the CB and MCB measures.

```
#include <iostream>
using namespace std;
void allocate() ;

int * p1 ;
int * p2;

void allocate() {

 int number = 88;
 int *p1 = &number;

  p2 = new int(99);
}

int main() {
  allocate();
  cout <<"p1 value :"<< p1;
  cout << "p2 value :"<<*p2;
  delete p1;
  delete p2;

  return 0;
}
```

Fig. 1. Sample program to demonstrate the effect of dynamic memory access and pointers and references.

Table 1. Complexity Caculation of CB for Allocate Program

| Executable Statement | S | Wn | Wi | Wc | W | S*W |
|---|---|---|---|---|---|---|
| void allocate() | 2 | 0 | 1 | 0 | 1 | 2 |
| int number = 88 | 4 | 0 | 1 | 0 | 1 | 4 |
| int *p1 = &number | 6 | 0 | 1 | 0 | 1 | 6 |
| p2 = new int(99) | 4 | 0 | 1 | 0 | 1 | 4 |
| int main() | 2 | 0 | 1 | 0 | 1 | 2 |
| allocate() | 1 | 0 | 1 | 0 | 1 | 1 |
| cout<<"p1 value :"<< p1 | 5 | 0 | 1 | 0 | 1 | 5 |
| cout<< "p2 value :"<<*p2 | 6 | 0 | 1 | 0 | 1 | 6 |
| delete p1 | 2 | 0 | 1 | 0 | 1 | 2 |
| delete p2 | 2 | 0 | 1 | 0 | 1 | 2 |
| return 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| **CB value** | | | | | | **35** |

Table 2. Complexity Caculation of MCB for Allocate Program

| Executable Statement | S | Wn | Wi | Wc | W | S*W |
|---|---|---|---|---|---|---|
| void allocate() | 2 | 0 | 1 | 0 | 1 | 2 |
| int number = 88 | 4 | 0 | 1 | 0 | 1 | 4 |
| int *p1 = &number | 8 | 0 | 1 | 0 | 1 | 8 |
| p2 = new int(99) | 5 | 0 | 1 | 0 | 1 | 5 |
| int main() | 2 | 0 | 1 | 0 | 1 | 2 |
| allocate() | 1 | 0 | 1 | 0 | 1 | 1 |
| cout<<"p1 value :"<< p1 | 5 | 0 | 1 | 0 | 1 | 5 |
| cout<< "p2 value :"<<*p2 | 7 | 0 | 1 | 0 | 1 | 7 |
| delete p1 | 3 | 0 | 1 | 0 | 1 | 3 |
| delete p2 | 3 | 0 | 1 | 0 | 1 | 3 |
| return 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| **MCB value** | | | | | | **41** |

The complexity that arises due to the usage of threads is considered in the MCB measure by adding a constant value of two to the total size (S) value, for statements with a thread invocation. Fig. 2 shows a sample program which was written to demonstrate the effect that threads has on the complexity of a program. Table 3 and Table 4 clearly demonstrates how complexity is calculated for the program given in Fig. 2, using the CB and MCB measures.

```
public class MyThread implements Runnable{
  public void run( ){
   System.out.println("Thread running");
  }
  public static void main(String args[]){
      MyThread mythread = new MyThread();

      Thread thread = new Thread(mythread);

      thread.start();
 }
}
```

Fig. 2. Sample program to demonstrate the effect of threads.

Table 3. Complexity Caculation of CB for MyThread Program

| Executable Statement | S | Wn | Wi | Wc | W | S*W |
|---|---|---|---|---|---|---|
| public void run( ) | 2 | 0 | 1 | 0 | 1 | 2 |
| System.out.println("Thread running") | 6 | 0 | 1 | 0 | 1 | 6 |
| public static void main (String args[]) | 4 | 0 | 1 | 0 | 1 | 4 |
| MyThread mythread = new MyThread() | 5 | 0 | 1 | 0 | 1 | 5 |
| Thread thread = new Thread(mythread) | 5 | 0 | 1 | 0 | 1 | 5 |
| thread.start() | 3 | 0 | 1 | 0 | 1 | 3 |
| **CB Value** | | | | | | **25** |

Table 4. Complexity Caculation of MCB for MyThread Program

| Executable Statement | S | Wn | Wi | Wc | W | S*W |
|---|---|---|---|---|---|---|
| public void run( ) | 2 | 0 | 1 | 0 | 1 | 2 |
| System.out.println("Thread running") | 6 | 0 | 1 | 0 | 1 | 6 |
| public static void main (String args[]) | 4 | 0 | 1 | 0 | 1 | 4 |
| MyThread mythread = new MyThread() | 6 | 0 | 1 | 0 | 1 | 6 |
| Thread thread = new Thread(mythread) | 8 | 0 | 1 | 0 | 1 | 8 |
| thread.start() | 5 | 0 | 1 | 0 | 1 | 5 |
| **MCB Value** | | | | | | **31** |

During exception handling the complexity of handling all the predicates of a try-catch() statement and throws have to be considered. Since throwing of exceptions without try-catch statements is often in line with class declarations, it will be considered with the use of the size (S) attribute by adding a value of one for the "throw" keyword. However, when try-catch() statements are involved, the catch blocks will act as the predicates for that type of exception. Therefore, each catch block can be considered as a branch in a conditional statement. Thus, this can be considered with the use of the weight due to type of control structure (Wc) attribute by assigning a weight of one for each catch block. Any exception that is thrown by a function can only be thrown once by a program for a given instance. Therefore, if the same exception is caught in several statements, only the first statement which catches that exception is considered. Fig. 3 shows a sample program which was written to demonstrate the effect that exceptions has on the complexity of a program. Table 5 and Table 6 clearly demonstrates how complexity is calculated for the program given in Fig. 3, using the CB and MCB measures.

```
public class Test{
   int a, b, c;

   public void output1( ){
      System.out.println("Output1");
   }

   public void output2( ){
      System.out.println("Output2");
   }


   public static void main(String args[]){
      Test t1 = new Test();
      Test t2 = new Test();

      try{
         t1.output1();
      }

      catch (Exception e){
         System.out.println("Error");
      }
      t1.output2();
   }
}
```

Fig. 3. Sample program to demonstrate the effect of exceptions.

Table 5. Complexity Caculation of CB for Test Program

| **Executable Statement** | **S** | **Wn** | **Wi** | **Wc** | **W** | **S*W** |
|---|---|---|---|---|---|---|
| public void output1( ) | 2 | 0 | 1 | 0 | 1 | 2 |
| System.out.println("Output1") | 6 | 0 | 1 | 0 | 1 | 6 |
| public void output2( ) | 2 | 0 | 1 | 0 | 1 | 2 |
| System.out.println("Output2") | 6 | 0 | 1 | 0 | 1 | 6 |
| public static void main(String args[]) | 4 | 0 | 1 | 0 | 1 | 4 |
| Test t1 = new Test() | 5 | 0 | 1 | 0 | 1 | 5 |
| t1.output1() | 3 | 0 | 1 | 0 | 1 | 3 |
| catch (Exception e) | 1 | 0 | 1 | 0 | 1 | 1 |
| System.out.println("Error") | 6 | 0 | 1 | 0 | 1 | 6 |
| t1.output2() | 3 | 0 | 1 | 0 | 1 | 3 |
| **CB Value** | | | | | | **38** |

Table 6. Complexity Caculation of MCB for Test Program

| **Executable Statement** | **S** | **Wn** | **Wi** | **Wc** | **W** | **S*W** |
|---|---|---|---|---|---|---|
| public void output1( ) | 2 | 0 | 1 | 0 | 1 | 2 |
| System.out.println("Output1") | 6 | 0 | 1 | 0 | 1 | 6 |

| | | | | | | |
|---|---|---|---|---|---|---|
| public void output2( ) | 2 | 0 | 1 | 0 | 1 | 2 |
| System.out.println("Output2") | 6 | 0 | 1 | 0 | 1 | 6 |
| public static void main(String args[]) | 4 | 0 | 1 | 0 | 1 | 4 |
| Test t1 = new Test() | 6 | 0 | 1 | 0 | 1 | 6 |
| t1.output1() | 3 | 0 | 1 | 0 | 1 | 3 |
| catch (Exception e) | 1 | 0 | 1 | 1 | 2 | 2 |
| System.out.println("Error") | 6 | 0 | 1 | 0 | 1 | 6 |
| t1.output2() | 3 | 0 | 1 | 0 | 1 | 3 |
| **MCB Value** | | | | | | **40** |

To accommodate the impact of recursive methods in the MCB formula, initially the total CB value for the entire program is calculated in the usual manner. Next, the addition of S*W value of the statements belonging to the recursive function is added to the computed total CB value. A calculation approach such as this was suggested to differentiate the complexity added by recursive functions, based on their content. Fig. 4 shows a sample program which was written to demonstrate the effect that recursive methods has on the complexity of a program. Table 7 and Table 8 clearly demonstrates how complexity is calculated for the program given in Fig. 4, using the CB and MCB measures.

```
public class Factorial{
 int a,b,c;

 public int fact(int n){
   if (n == 0)
       return 1;
   else
       return (n * fact(n-1));
 }
 public static void main(String args[]){
       Factorial t1 = new Factorial();
       int f = 0;
       f = t1.fact(5);
       System.out.println(f);
 }
}
```

Fig. 4. Sample program to demonstrate the effect of recursion.

Table 7. Complexity Caculation of CB for Factorial Program

| **Executable Statement** | **S** | **Wn** | **Wi** | **Wc** | **W** | **S*W** |
|---|---|---|---|---|---|---|
| public int fact(int n) | 2 | 0 | 1 | 0 | 1 | 2 |
| if (n==0) | 4 | 1 | 1 | 1 | 3 | 12 |
| return 1 | 1 | 1 | 1 | 0 | 2 | 2 |
| return (n * fact(n-1)) | 3 | 1 | 1 | 0 | 2 | 6 |
| public static void main() | 4 | 0 | 1 | 0 | 1 | 4 |
| Factorial t1 = new Factorial() | 5 | 0 | 1 | 0 | 1 | 5 |
| int f = 0 | 4 | 0 | 1 | 0 | 1 | 4 |
| f = t1.fact(10) | 5 | 0 | 1 | 0 | 1 | 5 |
| System.out.println(f) | 5 | 0 | 1 | 0 | 1 | 5 |
| **CB Value** | | | | | | **45** |

Table 8. Complexity Caculation of MCB for Factorial Program

| **Executable Statement** | **S** | **Wn** | **Wi** | **Wc** | **W** | **S*W** |
|---|---|---|---|---|---|---|
| public int fact(int n) | 2 | 0 | 1 | 0 | 1 | 2 |
| if (n==0) | 4 | 1 | 1 | 1 | 3 | 12 |

| return 1 | 1 | 1 | 1 | 0 | 2 | 2 |
|---|---|---|---|---|---|---|
| return (n * fact(n-1)) | 3 | 1 | 1 | 0 | 2 | 6 |
| public static void main() | 4 | 0 | 1 | 0 | 1 | 4 |
| Factorial t1 = new Factorial() | 5 | 0 | 1 | 0 | 1 | 5 |
| int f = 0 | 4 | 0 | 1 | 0 | 1 | 4 |
| f = t1.fact(10) | 5 | 0 | 1 | 0 | 1 | 5 |
| System.out.println(f) | 5 | 0 | 1 | 0 | 1 | 5 |
| **MCB Value** | | | | | | **65** |

To differentiate the complexity that arises due to the usage of simple and compound conditions, a weight of one was assigned to the weight due to type of control structure attribute (Wc), whenever the "&&" or "||" operator was used in a decisional statement to concatenate two or more conditions. Fig. 5 shows a sample program which was written to demonstrate the effect that compound conditions has on the complexity of a program. Table 9 and Table 10 clearly demonstrates how complexity is calculated for the program given in Fig. 5, using the CB and MCB measures.

```
public class Result{

  public void res(int marks){
   if(marks > 0 && marks < 50)
        System.out.println("Fail");
   else
        System.out.println("Pass");
  }

  public static void main(String args[]){
     Result r = new Result();
     r.res(50);
  }
}
```

Fig. 5. Sample program to demonstrate the effect of compound conditions.

Table 9. Complexity Caculation of CB for Result Program

| **Executable Statement** | **S** | **Wn** | **Wi** | **Wc** | **W** | **S*W** |
|---|---|---|---|---|---|---|
| public void res (int marks) | 2 | 0 | 1 | 0 | 1 | 2 |
| if (marks >0 && marks <50) | 8 | 1 | 1 | 1 | 3 | 24 |
| System.out.println("Fail") | 6 | 1 | 1 | 0 | 2 | 12 |
| System.out.println("Pass") | 6 | 1 | 1 | 0 | 2 | 12 |
| public static void main (String args[ ]) | 4 | 0 | 1 | 0 | 1 | 4 |
| Result r = new Result() | 5 | 0 | 1 | 0 | 1 | 5 |
| r.res(50) | 3 | 0 | 1 | 0 | 1 | 3 |
| **CB Value** | | | | | | **62** |

Tabel 10. Complexity Caculation of MCB for Result Program

| **Executable Statement** | **S** | **Wn** | **Wi** | **Wc** | **W** | **S*W** |
|---|---|---|---|---|---|---|
| public void res(int marks) | 2 | 0 | 1 | 0 | 1 | 2 |
| if (marks >0 && marks <50 ) | 8 | 1 | 1 | 2 | 4 | 32 |
| System.out.println("Fail") | 6 | 1 | 1 | 0 | 2 | 12 |
| System.out.println("Pass") | 6 | 1 | 1 | 0 | 2 | 12 |
| public static void main(String args[ ]) | 4 | 0 | 1 | 0 | 1 | 4 |
| Result r = new Result() | 5 | 0 | 1 | 0 | 1 | 5 |

| r.res(50) | 3 | 0 | 1 | 0 | 1 | 3 |
|---|---|---|---|---|---|---|
| **MCB Value** | | | | | | **70** |

## 4. Experiments and Results

To find the practical applicability of the CB measure and the MCB measure, as explained in [10], an empirical study was conducted using thirty software engineers with two or more years of working experience and five open source java programs. Initially, the software engineers were asked to rank five programs that were given to them on scale of 1-5 by giving one to the program with the least complexity and five to the program with the highest complexity. From the obtained ranks, an average value was calculated for each program. Table 13 shows the ranking of the software engineers for the five programs.

Next, complexities of the five programs were calculated using the CB and MCB measures. Table 11 and Table 12 shows the CB and MCB values derived for the five programs. Subsequently, the five programs were ranked based on the complexity values derived for the two measures. Table 13 shows the ranking of the five programs for the two measures.

Finally, Spearman's rank correlation coefficient between the two measures and the ranking of the software engineers was used to find out the most practically applicable measure.

Table. 11. CB Values Of The Five Java Programs

| **Program** | **CB Value** |
|---|---|
| Class2 | 142 |
| FactMain | 84 |
| FibonacciMain | 96 |
| ListStack | 95 |
| BubbleSort | 257 |

Table. 12. MCB Values of the Five Java Programs

| **Program** | **MCB Value** |
|---|---|
| Class2 | 142 |
| FactMain | 108 |
| FibonacciMain | 148 |
| ListStack | 154 |
| BubbleSort | 259 |

Table. 13. Comparison of Ranks Obtained

| **Programs** | **Ranks of Software** | **Ranks of CB** | **Ranks of MCB** |
|---|---|---|---|
| Class2 | 1 | 4 | 2 |
| FactMain | 2 | 1 | 1 |
| FibonacciMain | 3 | 3 | 3 |
| ListStack | 4 | 2 | 4 |
| BubbleSort | 5 | 5 | 5 |

As demonstrated in Table 14, it can be observed that, from the two measures, the MCB measure has the highest correlation coefficient with the rankings of the software engineers. CB measure has a correlation coefficient of 0.300 and MCB measure has a correlation coefficient of 0.900, with the rankings of the software engineers. Thus, it can be concluded that MCB measure is practically more applicable than the CB measure.

Table. 14. Spearman's Correlation Coefficient between Complexity Measures and Expert Rankings

| | | | Ranking of Software Engineers |
|---|---|---|---|
| Spearman's rho | Ranking of Software Engineers | Correlation Coefficient | 1.000 |
| | | Sig. (2-tailed) | . |
| | | N | 5 |
| | CB Rankings | Correlation Coefficient | .300 |
| | | Sig. (2-tailed) | .624 |
| | | N | 5 |
| | MCB Rankings | Correlation Coefficient | .900* |
| | | Sig. (2-tailed) | .037 |
| | | N | 5 |
| *. Correlation is significant at the 0.05 level (2-tailed). | | | |

## 5. Conclusion

Majority of the OO complexity metrics that has been proposed based on the cognitive aspect have relied on a maximum of three complexity factors to derive the complexity of a program. CB measure is one of the few metrics that has considered four or more complexity factors to measure the complexity associated with a software program. It considers the complexity that arises due to inheritance level of statements, type and nesting level of control structures and size of the program. In a previous study [9] authors propose some other factors that could be considered by the CB measure to make its complexity calculation more accurate. This paper demonstrates how those factors can be accommodated to the CB measure. It also validates the practical applicability of the MCB measure.

## Acknowledgment

The first author would like to thank Ms. R. W. D. N. K. Rajapakse for her kind contribution in checking the manuscript for the correct use of language.

## References

[1] Chidamber S. R., & Kemerer C. F. (1994, June). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, *20*, 476-493.

[2] Chen, J. Y., & Lu, J. F. (1993). A new metrics for object-oriented design. *Information of Software Technology*, *35(4)*, 232-240.

[3] Abreu F. B., & Carapuça R. (1994). Object-oriented software engineering: Measuring and controlling the development process. *Proceedings of the 4th International Conference on Software Quality*, ASQC, McLean, VA, USA.

[4] Li, W. (1998). Another metric suite for object-oriented programming. *The Journal of System and Software*, *44(2)*, 155-162.

[5] Misra S. (2007). An object-oriented complexity metric based on cognitive weights. *Proceedings of the 6th IEEE International Conference on Cognitive Informatics*.

[6] Misra S., & Akman K. I. (2008). Weighted class complexity: A measure of complexity for object oriented system. *Journal of Information Science and Engineering*, *24(6)*, 1689-1708.

[7] Misra S., Akman I., & Koyuncu M. (2011). An inheritance complexity metric for object-oriented code: A cognitive approach. *Sadhana Academy Proceedings in Engineering Sciences*, *36(3)*, 317 - 377.

[8] Chhillar U., & Bhasin S. (2011, July). A new weighted composite complexity for object-oriented systems. *International Journal of Information and Communication Technology Research*, *1(3)*, 101-108.

[9] De Silva, D. I., Kodituwakku, S. R., Pinidiyaarachchi, A. J., & Kodagoda, N. (2015). Improvements to a

complexity metric: CB measure. *Proceedings of the IEEE 10th International Conference on Industrial and Information Systems (ICIIS)*, Peradeniya, Sri Lanka, pp. 401-406.

[10] De Silva, D. I., Weerawarna, N., Kuruppu, K., Ellepola, N., & Kodagoda, N. (2013). Applicability of three cognitive complexity metrics. *Proceedings of the 8th International Conference on Computer Science &Education*, Colombo, Sri Lanka, 573-578.

**D. I. De Silva** graduated from Sri Lanka Institute of Information Technology (SLIIT) with a BSc special honors in information technology, in 2009. He completed his MSc in information technology in 2012. Currently he is reading for an MPhil at University of Peradeniya, Sri Lanka. He joined the academic staff of SLIIT in 2009 and currently works as a senior lecturer attached to the Department of Software Engineering of Faculty of Computing. His primary research interests are software complexity, software metrics, and machine translation.

**S. R. Kodituwakku** joined University of Peradeniya as an academic staff member in 1993. He received his PhD in 2004. Currently he is the dean, Faculty of Science at University of Peradeniya, Sri Lanka. In 2008, he was promoted as an associate professor and subsequently, he was appointed as a professor of Statistics and Computer Science in 2013. He served as the head, Department of Statistics and Computer Science at University of Peradeniya from 2015 to 2016. Also, he has served as the chairman, Board of Study in Statistics and Computer Science and Secretary, Postgraduate Institute of Science at Peradeniya University, from 2017 to 2016 and 2007 to 2010 respectively..

**A. J. Pinidiyaarachchi** graduated from University of Peradeniya, Sri Lanka in 2001. She received her PhD in 2009 from Uppsala University, Sweden. Currently she serves as a senior lecturer and as the Coordinator/Postgraduate Diploma in IT, Postgraduate Institute of Science (PGIS) at University of Peradeniya. She is a member of BoS/Statistics and Computer Science, PGIS and Editorial Board Statistical Handbook at University of Peradeniya.

**N. Kodagoda** graduated from University of Moratuwa, Sri Lanka in 1995. He is a PhD student at Sheffield Hallam University, England. He received his BSc and MPhil from University Moratuwa in 1995 and 2004 respectively. Currently he serves as the head, Department of Software Engineering, Faculty of Computing at Sri Lanka Institute of Information (SLIIT), Sri Lanka. He served as the head, Department of Information Technology and Chairman, Curriculum and Academic Quality of Faculty of Computing, SLIIT, from 2008 to 2015 and 2010 to 2013. Also, he served as a systems analyst cum programmer and research assistant at University of Moratuwa, from 1996 to 2001 and 1995 to 1996 respectively.