International Journal of Computer Science and Software Engineering (IJCSSE), Volume 4, Issue 10, October 2015
ISSN (Online): 2409-4285 www.IJCSSE.org Page: 272-279

Cognitive Weighted Method Hiding Factor Complexity Metric

Francis Thamburaj[1] and A. Aloysius[2]

[1, 2] Computer Science Department, Bharathidasan University, St. Joseph's College, Tiruchirappalli, Tamil Nadu 620002, India

[1]*francisthamburaj@gmail.com*, [2]*aloysius1972@gmail.com*

ABSTRACT

Method hiding plays a key role by serving the foundation for several guidelines in object-oriented software design. It brings about plethora of benefits such as easy comprehension, secured accessibility, modifiability without side-effects, modular testing, higher reliability etc. But, it has to be handled carefully since very low method hiding may result in insufficiently abstracted implementation and very high method hiding will lead to very little functionality. Hence, the method hiding factor has to be measured accurately in order to produce quality software. This article, proposes a new Cognitive Weighted Method Hiding Factor (CWMHF) complexity metric. It measures not only the software structural complexity, but also the cognitive complexity on the basis of type. The cognitive weights are calibrated based on 27 empirical studies with 120 persons. A case study and experimentation of the new software metric shows encouraging results. Further, a comparative study is made and the correlation test has proved that CWMHF complexity metric is a better, more realistic, and more comprehensive indicator of the software complexity than the existing Abreu's Method Hiding Factor complexity metric.

Keywords: *Cognitive Method Hiding Factor, Cognitive Metrics, Information Hiding, Software Complexity Metrics, Software Engineering.*

1. INTRODUCTION

The object-oriented paradigm plays an important role in the modern programming world. The ability to capture the real world easily is the main source of its attraction. Though the popularity of object-oriented programs is based on many reasons like clarity of structure, easy to comprehend, safe-guarded data values through encapsulation, reusability due to inheritance, contextual processing due to polymorphism and so on, information hiding plays the key role by serving the foundation for several guidelines in object-oriented software design [1].

Information hiding was first described by Parnas in his seminal article [2]. It is the fundamental concept in software engineering and rudimental principle of object-oriented programming. The objects are copies created out of the blue-print called class. The class is an encapsulation and contains the instance variables and instance methods. Because objects encapsulate data and implementation, the user of an object can view the object as a black box that provides services. Thus, information hiding is basically concealing the data and the methods that allows to access or modify the data.

In object-oriented programming languages, information hiding is achieved by encapsulating or packaging the instance variables that hold the data values or information, and the instance methods or the operations over the information. The extensively used encapsulation in the object-oriented programs is the 'class' block. Though encapsulation technique is used to hide information, encapsulation and information hiding are not the same [3]. The instance variables and instance methods may be encapsulated but may still be totally or partially visible to other classes and packages. The information hiding is actually implemented by the combination of class encapsulation and scoping of the attributes and methods within the class.

There are many benefits of information hiding. First and foremost, the data is safe-guarded, as opposed to the structured way of programming, by the encapsulation. Any modification or even access of the data is possible only through the related methods. Secondly, the encapsulation yields easy comprehension and helps to cope with complexity by bringing a better perspective on how to use the services of the class [4]. Thirdly, as it hides the implementation details of the software unit from its clients, the subsequent changes can be done with ease [1]. Instance variables and methods can be added, deleted, or changed, but as long as the services provided by the object remain the same, code that uses the object can continue to use it without being rewritten. Fourthly, the encapsulated classes and packages can be written without the detailed knowledge of other classes and packages. This helps to write different classes simultaneously by many programmers leading to faster

production of the software system [2]. Fifthly, it allows encapsulated modules to be reassembled and replaced without reassembling the entire software system. So, the testing of different classes can be done in parallel, speeding up the software production [2]. Sixthly, it increases software product flexibility, which means the possibility of drastically changing or improving one class without changing the other. This paves way for excellent modifiability without side-effects. Seventhly, it decreases the complexity and increases reliability [5]
In order to reap these benefits, it is important to measure the information hiding factor of the software in order to raise the quality of the software and to reduce the cost of software production. The method hiding factor, which is part of the information hiding, has to be handled carefully since very low method hiding may result in insufficiently abstracted implementation and very high method hiding will lead to very little functionality. Hence, the method hiding factor has to be measured more accurately in order to maintain the balance between hiddenness and visibility of methods in the system. This article, proposes and defines a new Cognitive Weighted Method Hiding Factor (CWMHF) complexity metric.

2. SURVEY OF LITERATURE

Multitude of object-oriented complexity metrics have been proposed by researchers. The aim of these software metrics is to produce quality object-oriented programs in terms of less complex designing, robust coding, better testing, easy comprehension, and economical maintenance. Therefore all the software complexity metrics are focused to measure the quality of the software being developed. Some of the popular and the most referenced object-oriented complexity metric suites are Metrics for Object Oriented Software Engineering (MOOSE) proposed by Chidamber and Kemerer in 1991 [6], Metrics for Object Oriented Design (MOOD) proposed by Fernando Britto Abreu and Rogério Carapuça in 1994 [4], the second set of MOOD metrics called the MOOD2 in 2001 [7] and the Quality Metrics for Object Oriented Design (QMOOD) proposed by Jagdish Bansiya in 2002 [8].
Although many object-oriented software complexity metrics have been suggested by several authors, only a very few object-oriented complexity metrics are proposed based on information hiding principle. Among these complexity metrics, Abreu's Attribute Hiding Factor (AHF) and Method Hiding Factor (MHF) are frequently referenced. The AHF is the ratio of all the hidden attributes to the total number of attributes defined in all the classes. The MHF is defined as the division of the addition of all the invisible methods defined in all classes with all the methods under consideration [4]. Also, Abreu *et al* proposed the Attribute Hiding Effective Factor (AHEF), and Operation Hiding Effective Factor (OHEF) in the second set of MOOD metrics called MOOD2. The AHEF is defined as the quotient between the cumulative number of the specification classes that do access the specification attributes and the cumulative number of the specification classes that can access the specification attributes [7]. Similarly, the OHEF is defined as the quotient between the cumulative number of the specification classes that do access the specification operations and the cumulative number of the specification classes that can access the specification operations [7].
The Abreu's information hiding metrics are not sufficient, because they are method and attribute level that are only finely granular and they are incomplete. So, Cao et al proposes information hiding metrics of the class and the system which are coarsely granular and medium granular [9]. Agrawal et al proposes Vulnerability Confinement Capacity metric to assess and improve encapsulation for minimizing vulnerability of an object oriented design [10]. Chen et al proposed Operating Complexity Metric (OXM), Operating Argument Complexity Metric (OACM), and Attribute Complexity Metric (ACM). These metrics are very subjective in nature [11]. Yadav et al proposed Encapsulated Class Complexity Metric to measure the complexity of class design [12].
None of the proposed method hiding factor complexity metrics considered the cognitive complexity aspect. They have dealt only with structural complexity of the software system. Wang observed that the traditional measurements cannot actually reflect the real complexity of software systems in a software design, representation, cognition, comprehension and maintenance. Instead the cognitive complexity metrics is an ideal measure of software functional complexities and sizes, as it represents the real semantic complexity by integrating both the operational and architectural complexities [13]. The cognitive complexity is defined as the mental burden on the user who deals with the code as developer, tester, maintainer etc. It is measured in terms of cognitive weights. Cognitive weights are defined as the extent of difficulty or relative time and effort required for comprehending given software, and measure the complexity of logical structure of software [14].
Regarding the cognitive complexity metrics, Wang et al proposed Cognitive Functional Complexity of Software [13]. Misra Sanjay proposed the Cognitive Weight

Complexity Measure [15], and Weighted Class Complexity [16]. Mishra Deepti has proposed the Class Complexity due to Inheritance [17]. Arockiam et al proposed the Extended Weighted Class Complexity [18]. Aloysius et al proposed Attribute Weighted Class Complexity [19], Cognitive Weighted Response For a Class [20], and Cognitive Weighted Coupling Between Object [21]. Francis Thamburaj et al has proposed Cognitive Weighted Polymorphism Factor [22]. But, there is no cognitive complexity metric for information hiding. Hence, there is a need to propose method hiding factor complexity metric based on cognitive perspective

The following section 3 defines and explains the proposed metric CWMHF. The section 4 discusses the calibration of the cognitive weights. The section 5 deals with the experimentation and the case study of the new complexity metric. The section 6 does the comparative study of CWMHF with MHF. The section 7 presents the conclusion and the possible future works.

3. COGNITIVE WEIGHTED METHOD HIDING FACTOR

Cognitive Weighted Method Hiding Factor (CWMHF) complexity metric is based on the Method Hiding Factor (MHF) complexity metric proposed by Abreu et al in MOOD 1 suite in 1994 to measure the visibility of the methods in different classes of the software system. It is defined as a quotient, where the numerator represents the number of invisible methods and the denominator represents the total number of methods defined in all the class in the system. It is formally defined as follows:

$$MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)} \quad (1)$$

where,

$M_d(C_i) = M_v(C_i) + M_h(C_i)$
$M_d(C_i)$ = Total number of methods defined in class C_i
$M_v(C_i)$ = Number of visible methods in class C_i
$M_h(C_i)$ = Number of hidden methods in class C_i
TC = Total number of Classes in the whole system.

If the value of MHF is high (100%), it means all methods are private which indicates very little functionality. Thus it is not possible to reuse methods with high MHF. MHF with low (0%) value indicate all methods are public that means most of the methods are unprotected which is against the very spirit of the object-oriented paradigm. This complexity metric measures only the structural complexity and does not bother about the cognitive aspect of the hidden methods. Therefore, the new CWMHF is put forward to include the cognitive complexity.

The CWMHF complexity metric augments the cognitive complexity based on the different types of visibility of the methods. The visibility can range from fully invisible, partially visible, and fully visible. The scope of this article is Java only and in this the range of visibility is implemented using different method scopes such as 'private', 'protected', 'public', and the default. The private scope makes the method visible only within the class, whereas the public scope makes the method visible to the whole system. The protected scope makes the method visible both within the class and the inheriting classes both within the package and other packages. Based on these four types of method visibilities, the new CWMHF can be mathematically defined as

$$CWMHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_h(C_i) + \sum_{i=1}^{TC} M_v(C_i)} \quad (2)$$

where,

$$\sum_{i=1}^{TC} M_h(C_i) = \sum_{i=1}^{TC} M_p(C_i) * CW_{pm} + M_d(C_i) * CW_{dm} + M_t(C_i) * CW_{tm}$$

$$\sum_{i=1}^{TC} M_v(C_i) = \sum_{i=1}^{TC} M_u(C_i) * CW_{um}$$

$M_p(C_i)$ = Number of private methods in class C_i
$M_d(C_i)$ = Number of default methods in class C_i
$M_t(C_i)$ = Number of protected methods in class C_i
$M_u(C_i)$ = Number of public methods in class C_i
CW_{pm} = Cognitive Weight of private methods
CW_{dm} = Cognitive Weight of default methods
CW_{tm} = Cognitive Weight of protected methods
CW_{um} = Cognitive Weight of public methods
TC = Total number of Classes in the whole system

In Eq. (2), the cognitive weights CW_{pm}, CW_{dm}, CW_{tm}, of the private, default, and protected methods respectively are calibrated in the following section. The cognitive weight of public methods CW_{um} is assumed to be 1. The range aligns the scale of complexity values of CWMHF with that of Abreu's complexity metric due to method invisibilities. According to Abreu, the denominator represents the maximum number of possible distinct usage of the method hiding factor and purpose of the denominator is to act as normalizer for the complexity metric MHF [23]. So, it will be more apt and meaningful to multiply the public or visible methods by cognitive weight 1 in the denominator of the complexity metric CWMHF in order to act as normalizer as far as the cognitive complexity metric is concerned. Further, the normalized complexity metric CWMHF becomes

dimensionless satisfying one of the seven criteria for robust object-oriented metric proposed by Abreu et al [4].

4. CALIBRATION OF COGNITIVE WEIGHTS

Cognitive weights for different shades of invisibilities of methods are calibrated in this section. In order to find the cognitive weight factor for private method (CW_{pm}), default method (CW_{dm}), and protected method (CW_{tm}), a comprehension test was conducted for three different groups of students to find out the time taken to understand the complexity of different types of invisibilities of methods. These groups of students had sufficient exposure to Java programming and especially, in understanding various types of scope usages and method hiding techniques. Around 40 students, who have scored 65% and above marks in Semester examination, were selected in each group. One undergraduate group and two postgraduate groups are called for the comprehension test and supplied 9 different programs namely, P1 to P9, three for each type of method hiding with multiple choice answers. The time taken by each student to understand the program and to choose the best answer was recorded after the completion of each program. This process is repeated for each group of students. To be accurate, these program comprehension tests were conducted online and the comprehension timings were registered automatically by the computer in seconds.

The average time taken to comprehend each individual program from P1 to P9 by each group was calculated, so as to get 27 different Comprehension Mean Times (CMT). Since 3 different groups of students have done the comprehension test for the same program, their values are averaged to obtain the 9 different values. These values are tabulated in Table I, under the column CMT. The tested programs are grouped into private method scope testing programs, default method scope testing programs, and protected method scope testing programs. The corresponding CMT values are also grouped into three categories, namely, Private Method (PM) values, Default Method (DM) values, and protected Method (TM) values. Then the average of each of these categories is calculated and displayed in the last column of Table I as the Average Comprehension Mean Time (ACMT) in seconds. The rounded Cognitive Weights (CW) are given in the last column of the Table I.

Table 1: Calibration of Cognitive Weights

Category	*Program #*	*CMT (Secs)*	*ACMT (Secs)*	*CW (Rounded)*
Private Method (PM)	P1	393.9231	418.8114	4
	P2	459.75		
	P3	402.761		
Default Method (DM)	P4	498.0909	496.7121	5
	P5	488.0		
	P6	504.0455		
Protected Method (TM)	P7	613.853	620.5196	6
	P8	590.7059		
	P9	657.0		

The Fig. 1 is the pictorial representation of Table 1. The comprehension mean time for three different method scopes are grouped under the heading private method, default method, protected method denoted by PM, DM, TM. In the bar chart, the CMT for each program is posted over the corresponding bar. The first three programs test the comprehensibility of private methods and their average CMT is 418.8114 which is rounded to yield 4 as the cognitive weight for PM. The programs 4, 5, and 6 test the comprehensibility of the default methods and their average CMT is 496.7121 which is rounded to yield 5 as the cognitive weight for DM. The last three programs test the comprehensibility of protected methods and their average CMT is 620.5196 which is rounded to yield 6 as the cognitive weight for TM.

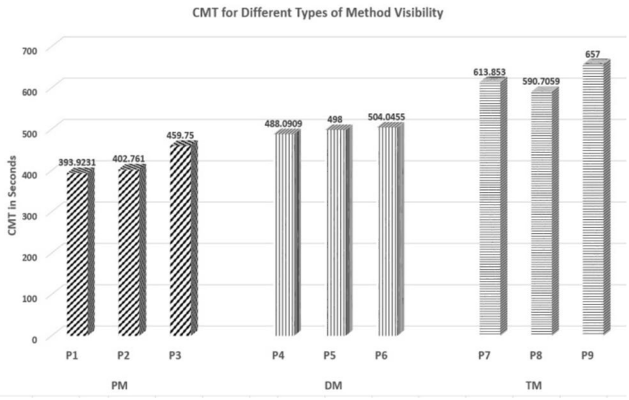


Fig. 1. Categorized Cognitive Weights

Thus the calibration of difficulty in understanding the different types or shades of invisibilities of methods in the program has brought out distinct index as the cognitive metric value. The calibration is done by measuring time and effort needed to comprehend programs as per Wang's methodology [13]. The ratio of these cognitive weights correspond to our natural intuitive understanding of difficulties and hence more meaningful and truthful [24].

5. EXPERIMENTATION AND CASE STUDY

The proposed CWMHF complexity metric given by Eq. (2) is evaluated with the following case study program. The program has four classes, namely, C1, C2, C3, and C4. It is a multi-level hierarchical inheritance tree. The root class C1 has two protected one-argument methods 'getItem()', 'getBrand()' and two private no-argument methods 'putItem()', 'putBrand()'. The class C2 has two protected single argument methods 'getNum()', 'getPrice()', two private no-argument methods 'putNum()', 'putPrice()', and one default no-argument method 'putAmount()'. The class C3 has two public no-argument methods 'putNum()', 'putPrice()'. The class C4 has two public no-argument methods 'putItem()', and 'putBrand()'.

```
1:  /***** Case Sudy Program *****/
2:  class C1{
3:    protected String s1, s2;
4:    protected String getItem(String s1){
5:      this.s1 = s1;
6:      return s1;
7:    }
8:    protected String getBrand(String s2){
9:      this.s2 = s2;
10:     return s2;
11:   }
12:   private void putItem() {
13:     System.out.print( s1 +" ");
14:   }
15:   private void putBrand() {
16:     System.out.print( s2 +" ");
17:   }
18: }
19:
20: class C2 extends C1{
21:   private int num;
22:   private float price;
23:   protected void getNum(int i) {
24:     this.num = I;
25:   }
26:   protected void getPrice(float f) {
27:     this.price = f;
28:   }
29:   private int putNum() {
30:     return num;
31:   }
32:   private float putPrice() {
33:     return price;
34:   }
35:   void putAmount(){
36:     System.out.print(num*price+" ");
37:   }
38: }
39:
40: class C3 extends C2{
41:   private int num;
42:   private float price;
43:   public int putNum() {
44:     return num;
45:   }
46:   public float putPrice() {
47:     return price;
48:   }
49: }
50:
51: class C4 extends C3{
52:   private String item;
53:   private String brand;
54:   public String putItem() {
55:     item  = s1;
56:     return item;
57:   }
58:   public String putBrand() {
59:     brand = s2;
60:     return brand;
61:   }
62: }
```

The UML diagram of the program is given in Fig. 2. It gives a clear picture of all the methods with their scopes in different classes of the software system. In calculating the MHF complexity value, Abreu considers all non-public methods as hidden methods [23]. Further, this complexity metric value considers only the structural aspect of the program. Applying the Abreu's complexity metric MHF as given in Eq. (1)

$$MHF = \frac{M_h(C_1) + M_h(C_2) + M_h(C_3) + M_h(C_4)}{M_d(C_1) + M_d(C_2) + M_d(C_3) + M_d(C_4)}$$

$$= (4+5+0+0) / (4+5+2+2)$$
$$= 9 / 13 = 0.6923 \text{ or } 69.23\%$$

Similarly, applying to the newly proposed complexity metric CWMHF, the complexity value can be calculated.

This complexity metric includes both the structural complexity as well as the cognitive complexity of the program. Hence, in the calculation of CWMHF, each method is multiplied by the corresponding method hiding cognitive weight in the numerator and in the denominator the public or visible methods are multiplied by the cognitive weight value of 1.

CWMHF = ((12+8)+(12+8+5)+0+0) / ((20+25)+(4*1))
= 45 / 49 = 0.9184 or 91.84%

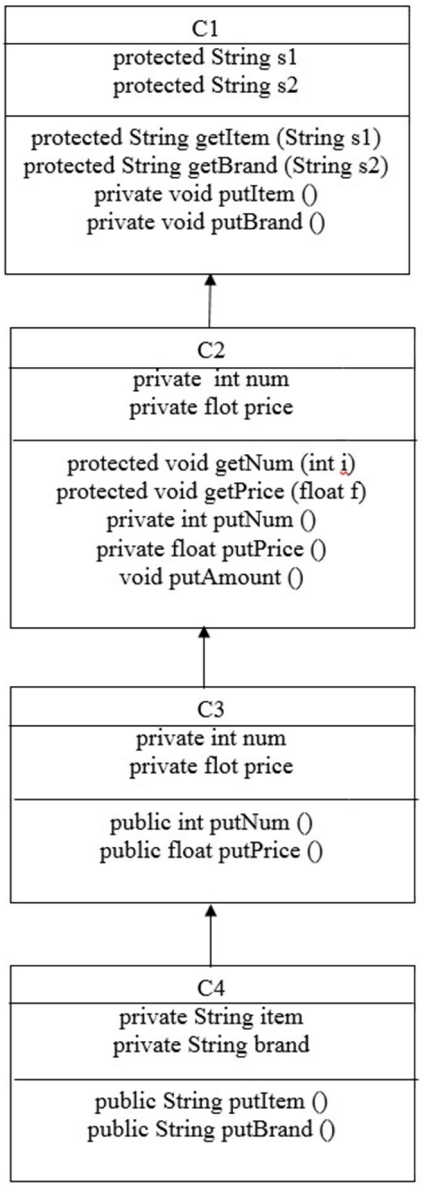


Fig. 2. UML Diagram of Case Study Program

Thus the case study proves the applicability of the newly proposed and defined CWMHF. Here the complexity value of CWMHF is greater than the complexity value of MHF, because CWMHF is based on the combined complexity of both the architectural and the cognitive aspects of the program. Though the complexity has increased, the range of complexity value is fixed as that of MHF. That is, from 0% to 100%. This is due to the effect of normalization of the quotient by multiplication of the public methods in the denominator by the cognitive weight value of 1. This is in line with the spirit of the formulation of MHF by Abreu [23]. Hence, the complexity value of CWMHF becomes larger than the complexity value of MHF.

6. COMPARATIVE STUDY

In this section, the comparative study is done to validate the CWMHF complexity metric, as it is done in other cases of newly proposed complexity metric [22]. The comparative study is performed against the complexity metric MHF which is part of the most widely accepted and empirically verified MOOD metric suite.

In order to do the comparative study, a comprehension test was conducted to a group of students who are doing their master's degree. There were forty students in the group who participated in the test. The students were given five different programs, P1 to P5, in Java for the comprehension test. The time taken to complete the test in seconds is captured in the online style, in order to maintain the accuracy. The average time taken to comprehend each program by all students is calculated and placed in Table II under the column head CMT. The MHF and CWMHF values are calculated manually for each of the five programs as demonstrated in the case study section of this article. Their values are also tabulated in Table II under the column MHF and CWMHF.

Table 2: Complexity Metric Values and CMT Values

Program #	***MHF***	***CWMHF***	***CMT***
P1	0.333	0.7148	501.08
P2	0.285	0.7059	598.13
P3	0.2	0.5714	435.21
P4	0.154	0.4286	249.39
P5	0.1	0.3077	197.82

Based on the Table 2 values, Pearson Correlation test was conducted between the MHF and CMT. The correlation value r(MHF, CMT) is 0.8999. Again the Pearson Correlation with CWMHF and CMT was calculated and the value r(CWMHF, CMT) is 0.9658. Both the correlations were found to be positive, implying that both MHF and CWMHF correlates well

with CMT values captured in the empirical test conducted. This shows that the CMT values are truthful and meaningful. The bigger correlation value for CWMHF than the MHF concludes that CWMHF is a better indicator of complexity of the classes with various scopes of methods. This fact is further clarified clearly in the correlation chart given in Fig. 3.

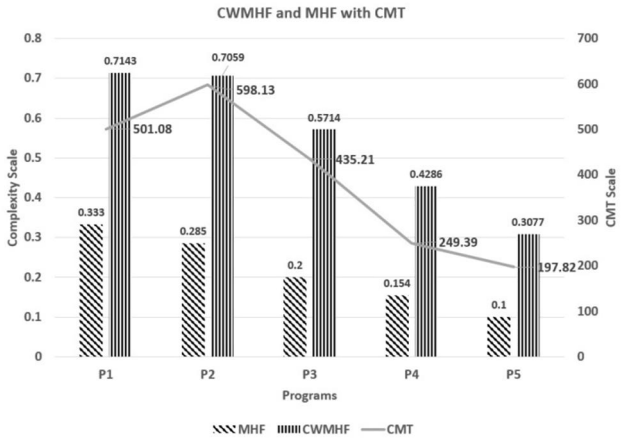


Fig. 3. Correlation of MHF and CWMHF with CMT

In Fig. 3 the CWMHF values are closer to the actual comprehension mean time taken by the students to understand the complexity of different method scopes in the given programs than the values of MHF. Thus the proposed CWMHF complexity metric, as it includes the cognitive complexity, is proved to be more robust and more realistic complexity metric than MHF complexity metric which considers only the architectural complexity.

7. CONCLUSIONS

In this article a new complexity metric called Cognitive Weighted Method Hiding Factor has been proposed and mathematically defined for measuring the class level complexity. The method hiding factor given by Abreu measures only the structural complexity. The cognitive weighted method hiding factor captures not only the structural complexity, but also the cognitive complexity that arises due to time and effort needed to comprehend the software. The cognitive weights are calibrated using series of comprehension tests and found that the cognitive load for different method scopes used to hide the visibility of the method in other classes differ in the decreasing order from protected, default, and private method scopes. The new CWMHF complexity metric is more comprehensive in nature and more true to reality. This is proved empirically by conducting a set of comprehension tests. Further, the applicability of the complexity metric is verified by case study. It is again confirmed by performing the correlation analysis that concluded saying that CWMHF is a better indicator of class complexity due to the encapsulation and method scopes than the MHF.

Regarding the future works, the CWMHF can be applied and studied for the other object oriented languages such as C++, ADA etc. Further, the empirical studies can be done with software industry groups. Also, a tool has to be developed for calculating the CWMHF values to compare it with other related method hiding complexity

REFERENCES

[1] Zoller Christian, and Axel Schmolitzky, "Measuring inappropriate generosity with access modifiers in Java systems," Software Measurement and the 2012 Seventh International Conference on Software Process and Product Measurement (IWSM-MENSURA), 2012 Joint Conference of the 22nd International Workshop on. IEEE, 2012.

[2] D. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM, vol. 15, no. 12, 1972, pp. 1053–1058, Dec. 1972.

[3] Tahvildari, Ladan, and Ashutosh Singh, "Categorization of object-oriented software metrics," Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering, Halifax, Nova Scotia, pp. 235–239, May 2000.

[4] F. B. Abreu, and R. Carapuça., "Object-oriented software engineering: Measuring and controlling the development process," Proceedings of the 4th international conference on software quality. vol. 186, pp. 1-8, 1994.

[5] Gupta Nidhi, and Rahul Kumar, "Reliability Measurement of Object Oriented Design: Complexity Perspective," International Advanced Research Journal in Science, Engineering and Technology, vol. 2, no. 4, pp. 33-44, April 2015.

[6] S. R. Chidamber, C. F. Kemerer, "Towards a metrics suite for object-oriented design," Object-Oriented Programming Systems, Languages and Applications (OOPSLA), vol. 26, pp. 197–211, 1991.

[7] Abreu, Fernando Brito. "Using OCL to formalize object oriented metrics definitions." In Tutorial in 5th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2001). 2001.

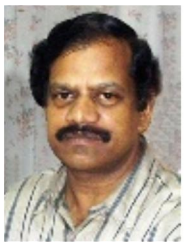
[8] J. Bansiya, and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," IEEE Transactions on Software Engineering," vol. 28, no. 1, pp. 4-17, 2002.

[9] Cao Yong, Qingxin Zhu, "Improved Metrics for Encapsulation Based on Information Hiding," Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for, pp. 742-747, 18-21 Nov. 2008.

[10] Agrawal, A., and R. A. Khan., "Assessing and Improving Encapsulation for Minimizing Vulnerability of an Object Oriented Design", Computational

Intelligence and Information Technology, Springer Berlin Heidelberg, pp. 531-533, 2011.
[11] Chen, J. Y., and J. F. Lu. "A new metric for object-oriented design," Information and Software Technology, vol. 35, no. 4, pp. 232-240, April 1993.
[12] Yadav, A., and R. A. Khan., "Development of Encapsulated Class Complexity Metric," Procedia Technology vol. 4, pp. 754-760, 2012.
[13] Y. Wang, and J. Shao, "Measurement of the cognitive functional complexity of software," Proc. Second IEEE Int. Conf. Cognitive Informatics (ICCI'03), pp. 1-6, 2003.
[14] Aloysius A., "A Cognitive Complexity Metrics Suite for Object Oriented Design," PhD Thesis, Bharathidasan University, Tiruchirappalli, India, 2012.
[15] Misra Sanjay, "A Complexity Measure Based on Cognitive Weights", Int. Jr. Theoretical and Applied Computer Sciences, vol. 1, no. 1, pp. 1-10, 2006.
[16] Misra sanjay, Akman K. Ibrahim, "Weighted Class Complexity: A Measure of Complexity for Object Oriented System", Jr. Information Science and Engineering, vol. 24, pp. 1689-1708, 2008.
[17] Deepti Mishra and Alok Mishra, "Object-Oriented Inheritance Metrics: Cognitive Complexity Perspective", Proceedings of the 4th International Conference on Rough Sets and Knowledge Technology, pp. 452–460, 2009.
[18] L. Arockiam, A. Aloysius and J. Charles selvaraj, "Extended Weighted Class Complexity: A new software complexity for objected oriented systems", Proceedings of International Conference on Semantic Ebusiness and Enterprise computing (SEEC), pp. 77-80, 2009.
[19] Arockiam L, and Aloysius A., "Attribute Weighted Class Complexity: A New Metric for Measuring Cognitive Complexity of OO Systems", WASET, vol. 5, pp. 10-21, 2011.
[20] Aloysius. A, Arockiam. L., "Cognitive Weighted Response For a Class: A New Metric for Measuring Cognitive Complexity of OO Systems," International Journal of Advanced Research in Computer Science, vol. 3, no. 4, 2012.
[21] A. Aloysisus, and L. Arockiam L, "Coupling Complexity Metric: A Cognitive Approach," International Journal of Information Technology and Computer Science, vol. 4, no. 9, pp. 29-35, 2012. ISSN: 2074-9007 (Print), ISSN: 2074-9015 (Online), DOI: 10.5815/ijitcs.2012.09.04.
[22] Thamburaj Francis, A. Aloysius, "Cognitive weighted polymorphism factor: A new cognitive complexity metric," World Academy of Science, Engineering and Technology, International Science Index, Computer and Information Engineering, vol. 2, no. 11, pp. 1604-1609, 2015, ISSN 1307-6892, in press.
[23] F. B. Abreu, M. Goulao, and R. Estevers, "Toward the design quality evaluation of object-oriented software systems," Proceedings of the 5th International Conference on Software Quality, Austin, Texas, USA, pp. 44-57, 1995.
[24] N. E. Fenton, and J. Bieman, "Software metrics: A rigorous and practical approach," 3rd edition. CRC Pr

AUTHOR PROFILES:



T. Francis Thamburaj is working as Assistant Professor in Department of Computer Science, St. Joseph's College, Trichy, Tamil Nadu, India. He has obtained the Master of Computer Applications degree in 1987 and Master of Philosophy degree in 2001 from Bharathidasan University, Trichy. He has 25 years of experience in teaching Computer Science. His research areas are Artificial Neural Networks and Software Metrics. He has published many research articles in the National / International conferences, and journals. Notably, he has presented, in 2011, a research paper in the World Congress in Computer Science, Computer Engineering, and Applied Computing (WORLDCOMP'11), Las Vegas, USA. A list of his research articles can be found in Google Scholar website. He is currently pursuing Doctor of Philosophy program and his current area of research is the Cognitive Complexity of Object Oriented Software Metrics.

A. Aloysius is working as Assistant Professor in Department of Computer Science, St. Joseph's College, Trichy, Tamil Nadu, India. He has got the Master of Computer Science degree in 1996, Master of Philosophy degree in 2004, and Doctor of Philosophy in Computer Science degree in 2013 from Bharathidasan University, Trichy. He has 15 years of experience in teaching and research. He has published many research articles in the National/ International conferences and journals. He has also presented 2 research articles in the International Conferences on Computational Intelligence and Cognitive Informatics in Indonesia. He has acted as a chair person for many national and international conferences. His current area of research is Cognitive Aspects in Software Design, Big Data, and Cloud Computing.