www.ijecs.in
International Journal Of Engineering And Computer Science ISSN: 2319-7242
Volume 4 Issue 11 Nov 2015, Page No. 14973-14979

Cognitive Perspective Of Attribute Hiding Factor Complexity Metric

Francis Thamburaj[1], A. Aloysius[2]
[1]Computer Science Department, Bharathidasan University, St. Joseph's College
Tiruchchirappalli, Tamil Nadu 620002, India
francisthamburaj@gmail.com
[2]Computer Science Department, Bharathidasan University, St. Joseph's College
Tiruchchirappalli, Tamil Nadu 620002, India
aloysius1972@gmail.com

Abstract: ***Information hiding is one of the key features and a powerful mechanism in Object-Oriented programming. It is critical to build large complex software that can be maintained economically and extended with ease. As information hiding improves the software productivity and promotes the software quality, it is essential to measure it. Further, the data or attribute value safety plays the vital role in the reliability of the software, which is the key factor determining the success of software. Data safety can be achieved by hiding the attribute. Hence, it is necessary and vital to measure the attribute hiding factor more accurately. This article introduces a new complexity metric called Cognitive Weighted Attribute Hiding Factor. It is defined and mathematically formulated to yield better results than the original Attribute Hiding Factor complexity metric. It is statistically proved by comparative study. Further, the new complexity metric is tested for empirical validity and applicability with a case study. The results show that the new complexity metric index due to the combination of encapsulation and attribute scoping is better, broader and truer to reality.***

Keywords: Attribute Hiding Factor, Information Hiding, Encapsulation Metric, Cognitive Software Complexity Metrics, Object-Oriented Software Metrics, Software Engineering.

1. Introduction

All the software metrics are mainly aimed at quality software production in terms of higher reliability at lower cost. The reliability is closely connected with the information or data or attribute value safety. Here, the term 'data safety' refers to safeguarding the data from accidental change or unintentional modification of the attribute values, or illegal access of the attribute values. This is very crucial to the overall success of the software even at the expense of higher cost of the software production. In order to achieve this safety, data members are encapsulated in the object-oriented paradigm. The usual encapsulation adopted is the '*class*' enclosure in which both the data members and the operations on data members are placed.

This encapsulation mechanism, forming the core concept in software engineering and fundamental design principle, making it the most important semantic characteristic of object-oriented programming, is the root cause of the popularity of object-oriented programs [1]. It gives raise to many valuable features like clear program structure, easier comprehension by hiding unnecessary internal complexities, higher reusability with inheritance mechanism, contextual processing via static or compile time polymorphism and dynamic or run-time polymorphism techniques, multi-granular testability due to class unity, greater extensibility, cheaper maintainability, finer modifiability without much side-effects, and so on.

But, the encapsulation can't ensure the complete safety of the data, even though it revolves around hiding the implementation details of a specific component, because encapsulation means only grouping of properties and that hiding is an orthogonal concept [2]. The process of encapsulation ensures only that the design decisions that are likely to change are localized [3]. So, encapsulation and information hiding are not the same. The instance variables and instance methods may be encapsulated but may still be totally or partially visible to other classes and packages [4]. By declaring the class as well as the variable as 'public', the variable can be accessed by any class like global variable and the global variables are evils because they create tight coupling which leads to lower modifiability, difficult testability, lesser extensibility etc. [5]. Therefore, the data hiding is actually implemented by the combination of class encapsulation and scoping of the member attributes and methods within the class.

On the one hand, the class encapsulation binds the instance variables and the instance methods that manipulate the values of instance variables as a single unit in order to hide internal complexity. The objects are created out of this class blue-print. So, the user of an object can view the object as a black box that provides services such as accessing or modifying the data etc.

On the other hand, the scoping mechanism controls the visibility of attributes and methods from other classes within the package or outside the package. The scoping mechanism varies from one language to another. This article focuses only on Java language, although it can be extended and applied to other languages with their own binding as Abreu et al has done for C++ and Eiffel [6] [7]. In Java language, there are four different scopes that can be used with the attributes, methods, and classes. They are implemented using three key words 'private', 'protected', and 'public'. The default scope is the package private scope and it does not have any special keyword. It makes the instance attributes, instance methods, and classes visible to all the other classes in the package in which the class is defined. The visibility of the 'private' scope is within the class encapsulation only. The 'protected' scope cuts across the boundaries of different packages and broadens the visibility to all the classes in the hierarchical inheritance

tree spread over multiple packages. Thus, information hiding is basically concealing the data and the connected interfaces that help to access or modify data or do any other manipulation of the data.

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 [8]. Thirdly, as it hides the implementation details of the software unit from its clients, the subsequent changes can be done with ease [9]. 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, leading to faster production of the software system [10]. 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 [10]. Sixthly, it increases the software product flexibility, which means the possibility of drastically changing or improving one class or module without changing the other. This paves way for excellent modifiability without side-effects. Seventhly, it decreases the complexity and increases the reliability due to information hiding [11]

The plethora of benefits shows the importance of information hiding and highlights the necessity of measuring the information hiding factor of the software system and especially data hiding, in order to maximize the reliability and minimize the production cost of the software. This article defines a new object-oriented software complexity metric called Cognitive Weighted Attribute Hiding Factor (CWAHF).

The following section 2 gives a short survey of literature indicating the need for new metric for data hiding. The section 3 proposes and defines the new complexity metric CWAHF. The section 4 depicts the calibration of the cognitive weights. The section 5 deals with the validation of new complexity metric through the comparative study of CWAHF and AHF. The section 6 does the experimentation and case study of the proposed complexity metric. The section 7 presents the conclusion and the possible future works.

2. Survey of Literature

Information hiding was first described by Parnas in his seminal article [10]. The software metrics that measure the amount of visibility of attributes and methods is called information hiding factor complexity metrics. 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. These metrics are part of the object-oriented metric suite called Metrics for Object Oriented Design (MOOD) proposed by Fernando Britto Abreu and Rogério Carapuça in 1994 [8]. 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 [8]. 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 [12]. 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 [12].

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 [13]. 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 [14]. Bansiya et al proposes Data Access Metric (DAM), which is the ratio of the number of private and protected attributes to the total number of attributes declared in the class. The range of DAM metric is from 0 to 1 and a high value is desired [15]. Saini et al proposed Encapsulation Factor (EF) based on the privacy and unity of attributes [16]. Tempero et al studied empirically 100 open-source Java applications to determine to what degree non-private fields are declared, and to what extent they are used. [17]. Agrawal et al proposes 'Vulnerability Confinement Capacity' metric to assess and improve encapsulation for minimizing vulnerability of an object oriented design [18]. Singh et al studied the effectiveness of encapsulation metric to refactor code and identify error prone classes [19]. Zoller et al developed two software metrics for Java, Inappropriate Generosity with Accessibility of Types (IGAT) and Inappropriate Generosity with Accessibility of Methods (IGAM) to measure the amount of types and methods with an unnecessarily generous access modifier [9]. Yadav et al proposed Encapsulated Class Complexity Metric to measure the complexity of class design [20]. Srinivasan and Devi have defined, among other metrics in their suite, the Attributes-Per-Class Factor (APCF) as the ratio of the number of private and protected attributes to all the attributes defined in the class. The metric is used to measure the amount of object properties, potential impact on children, the time and effort needs for the construction of a class [21].

Snyder examined the relationship between encapsulation and inheritance, since the inheritance mechanism severely compromises the benefits of encapsulation [22]. For example, permitting access to instance variables defined by the ancestor classes takes away the freedom of the designer to change the name, remove, or reinterpret an instance variable without the risk of adversely affecting descendant classes that depend on that instance variable. In Java, the scope of inheriting class limits the visibility of the attributes [23]. Chhillar et al, based on the class hierarchy, defines a suite of Member Access

Control Metrics such as Member Function Access Control Metrics, Data Member Access Control Metrics, and Member Access Control Factor Metrics, in order to estimate the time, cost, and effort for object-oriented software development [24].

None of the above list of metrics and studies is concerned with the cognitive aspect of the encapsulation and scopes. The only information hiding complexity metric based on cognitive aspect is the Cognitive Weighted Method Hiding Factor complexity metric proposed by the author earlier [25]. Therefore, there is a need to propose a new Cognitive Weighted Attribute Hiding Factor complexity metric to complement the previously proposed complexity metric for instance methods.

3. Cognitive Weighted Attribute Hiding Factor

The cognitive weighted attribute hiding factor complexity metric is based on the Abreu's attribute hiding factor. The AHF complexity metric is well defined based on the author's seven criteria for robust object-oriented metric, such as formal metric definition, system size independence, dimensionless of metric, early obtainability, down scalable, easy computability, and language independence [8]. It is also empirically validated both by the author and others [6] [7]. The AHF is formally defined as,

$$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)} \quad (1)$$

where,

$A_d(C_i) = A_v(C_i) + A_h(C_i)$
$A_d(C_i)$ = Total number of defined attributes in class C_i
$A_v(C_i)$ = Number of visible attributes in class C_i
$A_h(C_i)$ = Number of hidden attributes in class C_i
TC = Total number of Classes in the whole system

The complexity value of AHF ranges from 0% to 100%. If the value of AHF is 100%, it means all attributes are private, which is the ideal and desirable situation. When AHF value is 0% it indicates that all attributes are public, which is against the very spirit of the object-oriented paradigm. For better software quality and reliability, the complexity value should be kept high, preferably above 70% [26]. Very low values for AHF should trigger the designers' attention.

The AHF complexity metric captures only the architectural complexity of the software and does not bother about the cognitive complexity. But, 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 [27]. 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 [28].

Therefore, the new CWAHF is put forward to include the cognitive complexity. It augments the cognitive complexity based on the different types of visibility of the attributes. The visibility can range from fully invisible, partially visible, and fully visible. In Java language this range of visibility is implemented using different attribute scopes such as 'private', 'protected', 'public', and the default as mentioned in the introduction section. Based on these four types of attribute visibility, the new CWAHF can be mathematically defined as

$$CWAHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_h(C_i) + \sum_{i=1}^{TC} A_v(C_i)} \quad (2)$$

where,

$$\sum_{i=1}^{TC} A_h(C_i) = \sum_{i=1}^{TC} A_p(C_i) * CW_{pa} + A_d(C_i) * CW_{da} + A_t(C_i) * CW_{ta}$$

$$\sum_{i=1}^{TC} A_v(C_i) = \sum_{i=1}^{TC} A_u(C_i) * CW_{ua}$$

$A_p(C_i)$ = Number of private attributes in class C_i
$A_d(C_i)$ = Number of default attributes in class C_i
$A_t(C_i)$ = Number of protected attributes in class C_i
$A_u(C_i)$ = Number of public attributes in class C_i
CW_{pa} = Cognitive Weight of private attribute
CW_{da} = Cognitive Weight of default attribute
CW_{ta} = Cognitive Weight of protected attribute
CW_{ua} = Cognitive Weight of public attribute
TC = Total number of Classes in the whole system

In the Eq. (2), the denominator represents the cognitive complexity of all the attributes including non-public or hidden and public or visible attributes. The cognitive weight of public or fully visible attributes CW_{ua} is assumed to be 1. According to Abreu, the denominator represents the maximum number of possible distinct usage of the attribute hiding factor and the purpose of the denominator is to act as normalizer for the complexity metric AHF [6]. So, it will be more apt and meaningful to multiply the public or visible attributes by the cognitive weight value of 1 and sum up with the invisible or hidden complexity metric value in the denominator of the complexity metric CWAHF in order to act as normalizer as far as the cognitive complexity metric is concerned. This makes the range of complexity metric values of CWAHF to align with that of Abreu's range of complexity metric values due to different types of attribute invisibilities. In other words, the range of complexity metric values will be from 0% to 100%. Further, the normalized complexity metric CWAHF becomes dimensionless satisfying one of the seven criteria for robust object-oriented metric proposed by Abreu et al [8].

The numerator of the Eq. (2) represents the summation of the number of non-public or hidden attributes in each class of the whole software system. Here, the attributes belonging to each type of non-public scope is multiplied by the corresponding cognitive weights CW_{pa}, CW_{da}, CW_{ta}, of private, default and protected attributes respectively. Note that this complexity metric value also appears as one of the terms in the denominator of the Eq. (2). The cognitive weights CW_{pa}, CW_{da}, CW_{ta} are calibrated in the following section.

4. Calibration of Cognitive Weights

The cognitive weights for different types of visibility of attributes are calibrated in this section. A comprehension test

was conducted in order to find the cognitive weight factor for private attribute CW_{pa}, default attribute CW_{da}, and protected attribute CW_{ta}. Three different group of students were selected to undergo the test to find out the time taken to understand the complexity of different types of visibility of the attributes in the given program. These groups of students had sufficient exposure to Java programming and especially, in understanding various types of scope usages and attribute 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 with 9 different programs namely, P1 to P9, three for each type of attribute 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.

For each group of students, the average time taken to comprehend each individual program from P1 to P9 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 1, under the column CMT. The tested programs are grouped into private attribute scope testing programs, default attribute scope testing programs, and protected attribute scope testing programs. The corresponding CMT values are also grouped into three categories, namely, Private Attribute (PA) values, Default Attribute (DA) values, and proTected Attribute (TA) values. Then the average of each of these categories is calculated and displayed in the last but one column of Table 1 as the Average Comprehension Mean Time (ACMT) in seconds. The rounded ACMT values in the last column of the Table 1 represents the Cognitive Weight (CW) for different type of invisibility of the attributes.

Table 1: Calibration of Cognitive Weights

Category	*Program #*	*CMT (Secs)*	*ACMT (Secs)*	*CW (Rounded)*
Private Attribute (PA)	P1	205.233	205.071	2
	P2	211.567		
	P3	198.412		
Default Attribute (DA)	P4	339.833	323.029	3
	P5	324.433		
	P6	304.822		
Protected Attribute (TA)	P7	437.833	427.317	4
	P8	418.800		
	P9	425.317		

The Table 1 is graphically represented in Figure 1. The comprehension mean time for three different attribute scopes are grouped under the heading private attribute, default attribute, protected attribute denoted by PA, DA, TA. In the bar chart, the CMT for each program is posted over the corresponding bar. The first three programs test the comprehensibility of private attributes and their average CMT is 205.0707 which is rounded to yield 2 as the cognitive weight for PA. The programs 4, 5, and 6 test the comprehensibility of the default attributes and their average CMT is 323.0294 which is rounded to yield 3 as the cognitive weight for DA. The last three programs test the comprehensibility of protected attributes and their average CMT is 427.3167 which is rounded to yield 4 as the cognitive weight for TA. All the three rounded cognitive values are given under the column Cognitive Weights (CE) in Table 1.

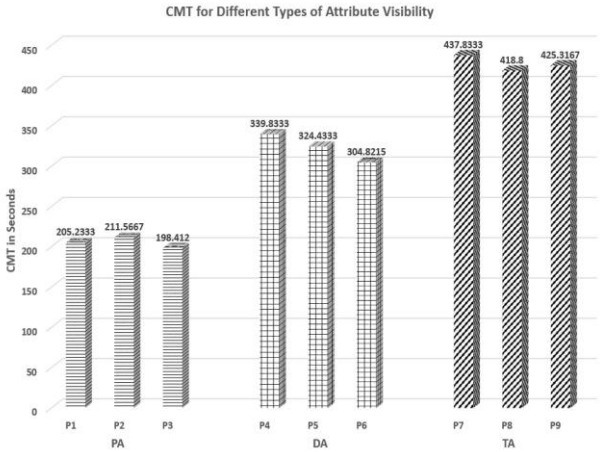

Figure 1: Categorized Cognitive Weights

Thus the calibration of difficulty in understanding the different types or shades of invisibility of attributes in the program has brought out distinct index as the cognitive weight value. The calibration is done by measuring the time and effort needed to comprehend the program as per Wang's methodology [27]. The ratio of these cognitive weights correspond to our natural intuitive understanding of difficulties and hence more meaningful and truthful [29].

5. Validation of CWAHF Complexity Metric

The proposed and formally defined complexity metric CWAHF is validated by the comparative study, as it is done in earlier cases of newly proposed complexity metrics [25] [30]. The comparative study is performed against the complexity metric AHF 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 was 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 2 under the column head CMT. The complexity values of AHF and CWAHF 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 2 under the column AHF and CWAHF.

Table 2: Complexity Metric Values and CMT Values

Program #	*AHF*	*CWAHF*	*CMT*
P1	0.923	0.973	345.47
P2	0.88	0.9565	325.68
P3	0.8	0.9167	357.92
P4	0.83	0.9333	295.91
P5	0.75	0.875	199.89

Pearson Correlation test, based on the Table 2 values, was conducted between the AHF and CMT. The correlation value r(AHF, CMT) is 0.6804. Again the Pearson Correlation with CWAHF and CMT was calculated and the value r(CWAHF, CMT) is 0.7642. Both the correlations were found to be positive, implying that both AHF and CWAHF 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 CWAHF than the AHF concludes that CWAHF is a better indicator of complexity of the classes with various scopes of attributes. This fact is further clarified clearly in the correlation chart given in Figure 2.

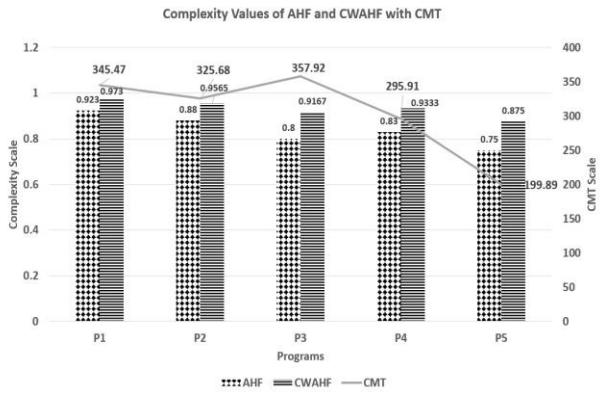

Figure 2: Correlation of AHF and CWAHF with CMT

In Figure 2, the CWAHF values are closer to the actual comprehension mean time taken by the students to understand the complexity of different attribute scopes in the given programs than the values of AHF. Thus the proposed CWAHF complexity metric, as it includes the cognitive complexity, is proved to be more robust and more realistic complexity metric than AHF complexity metric which considers only the architectural complexity.

6. Experimentation and Case Study

The newly proposed, defined, and validated complexity metric CWAHF given by Eq. (2) is evaluated here for applicability of the metric with the following case study program.

```
1:  /***** Case Study Program 1 *****/
2:  class C1{
3:     private int i1 = 10;
4:     public double d1 = 44;
5:     protected float f1 = 3.3f;
6:     public void getInput() {
7:        ….
8:     }
9:     public void putOutput() {
10:       ….
11:    }
12: }
13: class C2 extends C1 {
14:    private int i2 = 20;
15:    protected float f2 = 7.7f;
16:    public String str1 = "Francis";
17:    double d2=3.0;
18:    double d3=2.0;
19:    public void getInput() {
20:       ….
21:    }
22:    void processData() {
23:       ….
24:    }
25:    public void putOutput() {
26:       ….
27:    }
28: }
29: public class C3 extends C1{
30:    protected short s1=2;
31:    short s2=3;
32:    protected int i3=30;
33:    public void calcAndDisplay() {
34:       ….
35:    }
36:    public void getNewValues() {
37:       ….
38:    }
39: }
```

The program has three classes, namely, C1, C2, and C3. It is a multi-level hierarchical inheritance tree. The root class C1 has one private variable ‘int i1’, one public double variable ‘d1’, and one protected float variable ‘f1’. The class C2 has one private variable ‘int i2’, one protected float variable ‘f2’, one public string variable ‘str1’, and two default double variables ‘d2’, ‘d3’. The class C3 has one protected short variable ‘s1’, one default short variable ‘s2’ and one protected variable ‘int i3’. The Unified Modeling Language (UML) diagram of the program is given in Figure 3. It gives a clear picture of all the attributes with their scopes in different classes of the system including the available methods in each class of the software system.

In calculating the AHF complexity metric value, Abreu considers all non-public methods as hidden methods [6]. Further, this complexity metric value considers only the structural aspect of the program. Applying the Abreu’s complexity metric AHF as given in Eq. (1)

$$AHF = (2+4+3) / (3+5+3)$$
$$= 9 / 11 = 0.8181 \text{ or } 82\%$$

Similarly, applying to the proposed complexity metric CWAHF, 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 CWAHF, according to the type of scope, each attribute is multiplied by the corresponding visibility type based attribute hiding cognitive weight in both the numerator and the denominator. The public scope attributes, which can occur only in the denominator, are multiplied by the unit attribute hiding cognitive weight.

CWAHF = ((2+4) + (2+4+6) + (8+3)) / (29 + (2*1))
= (6+12+11) / (29+2)
= 29 / 31 = 0.9355 or 94%

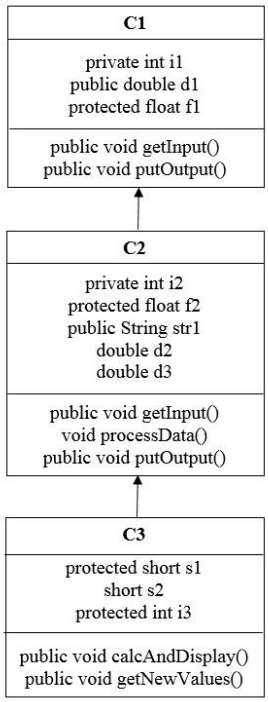

Figure 3: UML Diagram

Thus the case study proves the applicability of the newly proposed and defined CWAHF. Here the complexity value of CWAHF is greater than the complexity value of AHF, because CWAHF is based on the combined complexity of both structural and cognitive aspects of the program. Though the complexity has increased, the range of complexity value is fixed as that of AHF. That is, from 0% to 100%. This is due to effect of normalization of the quotient by multiplication of the denominator by the cognitive weight value of 1. This is in line with the spirit of the formulation of AHF by Abreu [6]. Hence, the complexity value of CWAHF becomes larger than the complexity value of AHF, but always within the range of 0% to 100%. The higher complexity metric values are preferred, especially above 60%. Very low values are calls for redesign of the software system.

7. Conclusion

In this article a new complexity metric called Cognitive Weighted Attribute Hiding Factor has been proposed and mathematically defined for measuring the class level complexity. The attribute hiding factor given by Abreu measures only the structural complexity. The cognitive weighted attribute 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 attribute scopes used to hide the visibility of the attribute in other classes differ in the increasing order from private, default, and protected attribute scopes. The proposed CWAHF 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 CWAHF is a better indicator of class complexity, due to the encapsulation and attribute scopes, than the AHF.

Regarding the future works, the empirical studies can be done with the software industry groups. The new metric can also be empirically experimented with large number of open source software programs. For this purpose, a software tool can be developed for automatically calculating the CWAHF values to compare it with other related attribute hiding complexity metrics. Also, the CWAHF can be applied and studied for the other object-oriented languages like C++, ADA etc.

References

[1] Voigt, Janina, Warwick Irwin, and Neville Churcher, “Class encapsulation and object encapsulation: An empirical study,” Computer Science and Software Engineering, University of Canterbury, 2010.

[2] P. Rogers, "Encapsulation is not information hiding,” Java World, http://www.javaworld/javaworld/jw-05-2001/jw-0518-encapsulation.html, pp. 1-3, 2001.

[3] Booch, G., Maksimchuk, R., Engle, M., Young, B., Conallen, J., Houston, K., “Object oriented design with applications,” Addison Wesley Professional, 2007.

[4] 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.

[5] Bain Scott L., “Encapsulation as a first principle of object-oriented design,” http://www.netobjectives.com/ resources/articles/, first-principle-object-oriented-design, pp. 1-15, April 2004.

[6] 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.

[7] Abreu, Fernando Brito, Rita Esteves, and Miguel Goulão, “The design of Eiffel programs: Quantitative evaluation using the mood metrics,” In Proceedings of TOOLS’96. California, July, 1996.

[8] 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.

[9] 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.
[10] 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.
[11] 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.
[12] 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.
[13] Cao, Yong, and Qingxin Zhu, "Improved metrics for encapsulation based on information hiding," Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for. IEEE, 2008.
[14] 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.
[15] 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.
[16] Saini Sunint and Mehak Aggarwal, "Enhancing mood metrics using encapsulation," In ICAI, in Proceedings of the 8th WSEAS International Conference on Automation and Information, Vancouver, Canada, June 19-21, vol. 7, pp. 252-257, 2007.
[17] Tempero, Ewan, "How fields are used in Java: An empirical study," In Australian Software Engineering Conference, IEEE, ASWEC'09, pp. 91-100, 2009.
[18] 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.
[19] Singh, Satwinder, and K. S. Kahlon, "Effectiveness of encapsulation and object-oriented metrics to refactor code and identify error prone classes using bad smells," ACM SIGSOFT Software Engineering Notes vol. 36, no. 5, pp. 1-10, 2011.
[20] Yadav, A., and R. A. Khan., "Development of Encapsulated Class Complexity Metric," Procedia Technology, vol. 4, pp. 754-760, 2012.
[21] K.P. Srinivasan, T. Devi, "A complete and comprehensive metrics suite for object-oriented design quality assessment," International Journal of Software Engineering and Its Applications, vol. 8, no. 2, pp. 173-188, 2014.
[22] A. Snyder, "Encapsulation and inheritance in object-oriented programming languages," in Conference proc. on Objectoriented programming systems, languages and applications (OOPSLA '86), Portland, OR, USA, Sep. 29–Oct. 2, pp. 38–45, 1986.
[23] Ankita Mann, Sandeep Dalal and Neetu Dabas., "Measurement of design complexity of different types of inheritance using cohesion metrics", International Journal of Computer Applications, vol. 77, no. 3, pp. 26-32, September 2013.
[24] Chhillar et al., "An access control metric suite for class hierarchy of object-oriented software systems", International Journal of Computer and Communication Engineering, vol. 4, no. 1, January 2015.
[25] Thamburaj Francis and A. Aloysius, "Cognitive weighted method hiding factor complexity metric," International Journal of Computer Science and Software Engineering (IJCSSE), vol. 4, no. 10, October 2015.
[26] B. F. Abreu, "Design metrics for object oriented software system," ECOOP'95, Quantitative Methods Workshop, Portugal, 1995.
[27] 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.
[28] Aloysius A., "A Cognitive Complexity Metrics Suite for Object Oriented Design," PhD Thesis, Bharathidasan University, Tiruchirappalli, India, 2012.
[29] N. E. Fenton and J. Bieman, "Software metrics: A rigorous and practical approach," 3rd edition. CRC Press, ISBN: 9781439838228, pp. 54, November 2014.
[30] 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, 2015.

Author Profile

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. He is the founder of Computer Science Department in Loyola College, Chennai, in 1993, and Information Technology Department in St. Joseph's College, Trichy, in 2006. 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 research areas are Cognitive Aspects in Software Design, Big Data, and Cloud Computing.