Volume 3, Issue 6, June 2013 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Analytical Evaluation of Class Complexity Metric

Vinay Singh [1] , Kunj Bihari Jarika[3]
Department of IT,
Usha Martin Academy, Ranchi, India

Vandana Bhattacherjee[2]
Department of CS & E,
Birla Institute of Technology, Mesra, India

Abstract— Complexity measures at class level are always needed at every stage of the development life cycle. Since the nineties, the Object oriented metrics have been increasingly popular. Weighted Method per class (WMC), Class Method Complexity (CMC) and Class Complexity (CC) are the three popular metrics, which have been studied in this research. A complete class complexity metric (CCC) has been proposed. Analytical evaluation of these four metrics has been done by the Weyuker's property. An automated INDIGINOUS tool was developed for the empirical sample of these metrics.

Keywords— ***Cyclomatic Complexity, Object-Oriented Design, Software Metrics, Software Component, Weyuker's Property***

I. INTRODUCTION

Software Engineering metrics are unit of measurement, which are used to characterize software engineering products, processes and people. If used properly they can allow us to identify and quantify improvement and make meaningful estimates. The need for software metrics is indeed real. Just as any other engineering discipline, software engineering, to some extent, needs to be able to measure the quality of the product that is being produced and delivered to the customer. Measurement of software complexity and fault proneness has been conducted since seventies. Thomas McCabe proposes a measure of software called cyclomatic complexity [14]. Making use of graph theory, McCabe postulates that software with a large number of possible control path would be more difficult to understand, more difficult to maintain, and more difficult to test. However, the cyclomatic number presents only a partial view of complexity. One of the problems of using cyclomatic numbers as a measure of software complexity is that it produces just a single value to describe a module's complexity. Many programs have a large number of decisions but are easy to understand code and maintain. Thus relying only on the cyclomatic number to measure actual program complexity can be misleading. The metrics suite proposed by Chidamber and Kemerer is one of the best-known object-oriented metrics [13]. Weighted method per class defines the complexity of the class is the some of the total number of methods in which method complexities are considered to be unity. Li theoretically validated Chidamber and Kemerer metrics[15] using a metric evaluation framework proposed by Kitchenham et al[1] and discovered some of the deficiencies of metrics in the evaluation process and proposed a new set of object-Oriented metrics that overcome these deficiencies. Qingfeng et.al. [12]analyzed Software power(SP) and applied to measured the software complexity by expanding the information entropy theory. Kuljit et.al.[7] advocates that component users need not bother about the internal details of the components. They believed that complexity of the internal structure of the component could help estimating the effort related to evolution of the component. They have focused on quality of internal design of a software component and its relationship to the external quality attributes of the component. Barota[2] has discussed metrics to evaluate usability of software components. Changjun[3] Introduce a description technique for software architecture and separating the control flow from component based on the description technique, quality metric was proposed, which is exploited to preliminary evaluate the effects of component on the quality of software architecture.

A metric may be validated mathematically using measurement theory, or empirically by collecting data. Measurement theory attempts to describe fundamental properties of all measures. Weyuker [4] concentrated on finding desirable properties that these measures should satisfy. Weyuker proposed nine properties that partially characterize good software measures. Chidamber and Kemerer[13],Gurusaran and Roy[5] Chernavasky and Smith[6], Xhang and Xie[8] and Naveen and Joshi [10] have used Weyuker's property to analyze Object-Oriented metrics. There has been a mixed response from the community on these axioms. Chidamber and Kemerer have commented on unsuitability of some of the axioms to object oriented design metrics. Questions have also been raised about the consistency of these axioms. Cherniavsky and Smith [6] showed that the axioms are not sufficient. Property 9 has been the most controversial property with respect to object oriented metrics. The property tries to capture the complexity occurring due to interaction during composition. A more complete model of program complexity is needed with the increased spread of Object-Oriented programming for the need of a metric suite that could take into consideration the complexity of Object-Oriented structure. The objectives of this paper are twofold. First, we study the existing class complexity metrics viz. WMC, CMC, CC and proposed a new class complexity metric CCC. Second, an analytical evaluation of the complexity metric has been done. The rest of the paper is organized as follows. Section II considers the existing class complexity metrics. Section III proposes another complexity metric called complete class complexity metric for Object-Oriented design. Section IV is

the description of the program and their statistics. Section V is the description of the Weyuker's list of properties and an analytical evaluation of the complexity metric is presented, where as Concluding remarks are presented in Section VI.

II. EXISTING CLASS COMPLEXITY METRICS

Weighted Method per Class Metric of Chidamber and Kemerer

Definition

Consider a class C_i with methods M_1, M_2, $M_3 \ldots M_n$ that are defined in the class. Let c_1, c_2, $c_3 \ldots c_n$ be the complexity of the methods.

Then,

$$\text{WMC(Weighted Method per Class)} = \sum_{i=1}^{n} c_i$$

If all the method complexities are considered to be unity, then ,WMC = n the number of methods.

Theoretical Basis

WMC metric relates directly to complexity of an individual as defined by Bunge as the "numerosity of its composition" [9]. Thus, it can be said that the complexity of an object is the cardinality of its set of properties. In Object -Oriented terminology, the properties of an object include the instance variables and its methods. As mentioned in Chidamber and Kemerer, WMC relates directly to Bunge's definition of complexity of a thing, since methods are properties of object classes and complexity is determined by the cardinality of its set of properties. The number of methods is therefore, a measure of class definition as well as being an attribute of the class since attributes correspond to properties. They further mention that the number of instance variables has not been included in the definition of the metric since it was assumed that methods are more time consuming to design than instance variables.

Viewpoints

The number of methods and the complexity of methods involved is a predictor of how much time and effort is required to develop and maintain the class.

The large the number of methods in a class the greater the potential impact on children, since children will inherit all the methods defined in the class.

Classes with large number of methods are likely to be more application specific, limiting the possibility of reuse.

Class Method Complexity metric of Li

Definition

Li's [15]CMC (Class Method Complexity) is the summation of the internal structural complexity of all local methods, regardless of whether they are visible outside the class or not(e.g. all the private, protected and public methods in class). This definition is essentially the same as the first definition of the WMC metric in [13]. However, the CMC metric's theoretical basis and viewpoints are significantly different from the WMC metric.

Theoretical Basis

The CMC metric captures the complexity of information hiding in the local methods of a class. This attribute is important for the creation of the class in an Object-Oriented design (OOD) because the complexity of the information hiding gives an indication of the amount of effort needed to design, implement, test and maintain the class.

Viewpoints

The CMC metric is directly linked to the effort needed to design, implement, test and maintain a class. The more complex the class methods are, the more the effort needed to design, implement, test, and maintain the methods.

The more complex the class methods are, as measured by the internal complexity of the methods, the more the effort is needed to comprehend the realization of information hiding in a class.

Class Complexity Metric of Balasubramanian

Definition

Balasubramanian's CC (Class Complexity) metric, is calculated as the sum of the number of instance variable in a class and the sum of the weighted static complexity of local methods in class [11].

To measure the static complexity Balasubramanian uses McCabe's Cyclomatic Complexity [14] where the weighted result is the number of node subtracted from the sum of the number of edges in a program flow graph and the number of connected components.

III. PROPOSED METRIC

Definition

The CCC (Complete Class Complexity Metric) is proposed for measuring the complexity of class in Object-Oriented Design. To calculate the CCC metric the following nine metrics are designed at class and interface level.

Table 1 Metric Description

METRIC	NAME	DESCRIPTION
NOMT	Number of Methods	This metric is a count of the methods defined in a class.
AVCC	Average Cyclomatic Complexity	This metric uses McCabe's cyclomatic complexity to count the average cyclomatic complexity of methods defined in a class
MOA	Measure	This metric measure the extent of the part-whole relationship, realized by

	of aggregation	using attributes. The metric is a count of the number of data declarations whose types are user-defined classes.
EXT	External Method calls	This metric is a count of the total number of external method calls in a class.
NSUP	Number of Super Class	This metric is a count of the total number of ancestor classes of a given class
NSUB	Number of Sub Class	This metric is a count of the total number of immediate sub classes
INTR	Interface Implemented	This metric is a count of the number of interfaces implemented by a class.
PACK	Package Imported	This metric is a count of the number of packages imported in a class.
NQU	Number of Queries	This metric is a count of the number of return point of all the methods in a class.

Now the CCC metric is defined as follows:

$$CCC = NOMT + AVCC + MOA + EXT + NSUP + NSUB + INTR + PACK + NQU$$

Complexity Metric CCC is based upon the following assumptions.

The earlier class complexity metrics WMC defined by Chidamber-Kemerer, CMC metric by Li and CC metric by Balasubramanian have focused only on Number of methods, methods complexity, and method complexity along with the number of instance variables. It has been realized that in some cases these parameters are not sufficient to determine the complexity of the class. The focus was on each dimension of a class to measure the complexity. To measure the class complexity metric we have designed nine different metrics viz. NOMT, AVCC, MOA, EXT, NSUP, NSUB, INTR, PACK and NQU and finally get the sum of all these nine metrics to find the Complete Class Complexity Metric(CCC).

Theoretical Basis

In object-oriented design, classes are a combination of properties, behaviours and their relationships. The relationship between classes is usually measured by the number of aggregation and generalization. To measure the complexity of a class the focus should be given to all these properties. The CCC metric measures the classes at method level, attribute level and their relationships. Method complexity is measured by the number of methods, their internal complexity by average cyclomatic complexity, message sent to other methods and the number of return points. The properties in a class can be primitive type variable or a reference variable. The primitive type variable is used by the functionality of a class, the reference variable also termed as the measures of aggregation have been taken as an attribute for measuring the class complexity. The depth and height of the class should also be counted as it measures the strength of the reusability so the number of super classes, sub classes and the interface implemented has also been considered for computing class complexity.

Viewpoints

The CCC metric involved all the possible attribute of the class and is predictor of how much time and effort is required to design and maintain the class. The value of the CCC metric is directly linked with the understandability and testability of the class. The more the value of CCC metric are, the more the effort needed to maintain the class.

The example for illustration/calculation of existing and proposed complexity metric has been taken from /opensource/src/dlib which is shown below.

```
Package dlib;
import  java.util.*;// Package=1
public class NameDB extends NamedObject // Super class=1
{
        // constructors
        NameDB (String name) // Method=1
        {
                super(name); Number //External Method call =1
        }
        private Hashtable Names = new Hashtable();// Method of aggregation=1
        /**
         * find the name associated with a number, or return null
         */
        public Object FindName (int number)        //Method=1
        {
return( Names.get(new Integer(number)));//Queries=1;External method call =1
        }

        public Object FindName (Integer number)  //Method=1
```



```
        {
return( Names.get(number));  //Queries=1 ,External Method call =1
        }

        public Object FindNumber (String name) //Method=1
        {
                return( Names.get(name)); //Queries=1 ; External Method call =1
        }

        public void AddName (String s, int n) //Method=1
        {
                Integer i = new Integer(n);
                  Names.put(s,i); Number //External Method call=1
                  Names.put(i,s);
        }

        public void AddName (String s, Object n) //Method=1
        {
                Names.put(s,n);
                  Names.put(n,s);
        }
}
```

From the above class NameDb the following value of the design metrics are NOMT=4, AVCC=1, MOA=1,IV=0,EXT=4,NSUP=1,NSUB=0,NPI=1,NQU=3,CCC=4+1+1+0+4+1+0+3=15 WMC=4 CMC=4+1=5 and CC =4+1+0=5

IV. PROGRAM DESCRIPTION

The program Flow of INDIGINOUS tool is summarized in figure 1. The program flow receives Java source code as input, the source code is parsed and XML file is generated. The XML file is parsed for metric calculation and generating the Excel sheet. The required tokens are matched for the metrics WMC, CMC, CC, and the new proposed metric CCC. INDIGINOUS tool will compute the final metric value and display the result.

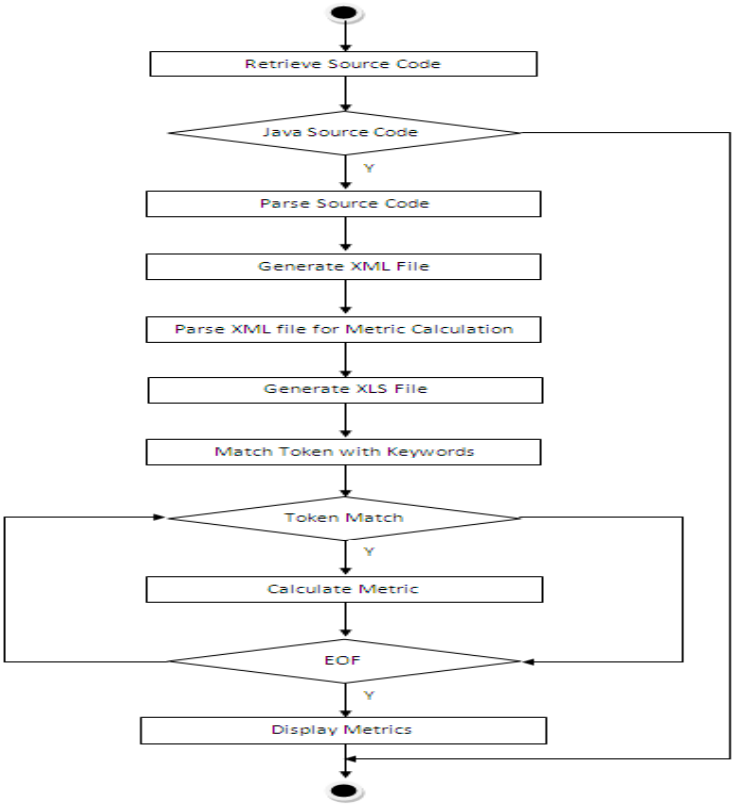

Figure 1 Program Flow chart of CCC Metric Tool

For analytical evaluation of Weyuker's property, the Metric values of test data is taken from the Post graduate computer Application student projects of Usha Martin Academy.

Seven different classes of linked list implemented in Java have been taken from two groups of six students. The class LinkedList_0 defines the functionality i.e. Create a node, get data, insert at beginning, insert at end and traverse the linked list. LinkedList_1 defines the functionality i.e. count the number of node, delete from beginning, delete at end, and the traversal of Linked List.LinkedList_2 defines the functionality i.e. create node, insert at beginning, insert at end, count the number of nodes and traversal of nodes. The class LinkedList_0and1 contains the union of the functionality of LinkedList_0 and LinkedList_1, where as LinkedList_2and1 contains the union of the functionality defines in LinkedList_1 and LinkedList_2

The statistics of all the seven classes are computed through an INDIGINOUS tool and listed in table 2.

Table 2 Program Statistics

Class Name	NOMT	AVCC	INST	MOA	EXT	NSUP	NSUB	INTR	PACK	NQU
LinkedList_0	5	2	1	1	5	0	0	0	1	1
LinkedList_1	4	2.75	1	1	1	0	0	0	1	0
LinkedList_2	5	2	1	1	5	0	0	0	1	1
LinkedList_01	4	2.25	1	1	5	0	0	0	1	0
LinkedList_0and1	8	2.25	1	1	5	0	0	0	1	1
LinkedList_01and1	7	2.428571	1	1	5	0	0	0	1	0
LinkedList_2and1	7	2.142857	1	1	5	0	0	0	1	1

V. WEYUKER PROPERTIES

The Weyuker properties are defined below. The notations used are as follows: P, Q and R denote classes, $P+$ denotes combinations of classes P and Q, μ denotes the chosen metric, μ(P) denotes the value of the metric for class P, and P≡Q (P is equivalent to Q) means that two class designs, P and Q, provide the same functionality. The definition of combination of two classes is taken here to be the same as suggested by [4], i.e., the combination of two classes results in another class whose properties (method and instance variables) are the union of the properties of the component classes. Also, "combination" stands for Weyuker's notion of "concatenation". Let μ be the metric of classes P and Q

Property 1: This property states that

$$(\exists P)(\exists Q)\ (\mu(P) \neq \mu(Q))$$

It ensures that no measure rates all classes to be of same metric value.

Property 2: Let c be a nonnegative number. Then there are finite numbers of classes with metric c. this property ensures that there is sufficient resolution in the measurement scale to be useful.

Property 3: There are distinct classes P and Q such that μ (p) = μ (Q).

Property 4: For object-oriented system, two classes having the same functionality could have different values, it is because of classes development are program dependent.

$$(\exists P)(\exists Q)(P = Q)\ \&\ (\mu(P) \neq \mu(Q))$$

Property 5: When two classes are concatenated, their metric should be greater than the metrics of each of the parts.

$$(\forall P)(\forall Q)(\mu(P) \leq \mu(P+Q)\ \&\ \mu(Q) \leq \mu(P+Q)$$

Property 6: This property suggests non-equivalence of interaction. If there are two classes bodies of equal metric value which, when separately concatenated to a same third class, yield program of different metric value.

For class P, Q and R

$$(\exists P)(\exists Q)(\exists R)(\mu(P) = \mu(Q)\ \&\ \mu(P+R) \neq \mu(Q+R)$$

Property 7: This property is not applicable for object-oriented metrics [13].

Property 8: It specifies that "if P is a renaming of Q; then μ(P) = μ (Q) "

Property 9: It suggests that metrics of a class formed by concatenating two class bodies, at least in some cases can be greater than the sum of individual metrics such that

$$(\exists P)(\exists Q)(\mu(P) + \mu(Q) < \mu(P+Q))$$

Analytical Evaluation of Complexity metric

Table 2 summarizes the metric evaluation results for CCC metrics for Weyuker's property.

Property 1. Let LinkedList_0 and LinkedList_1 are the two different classes with the different metric value .i.e. CCC(LinkedList_0) ≠ CCC(LinkedList_1) hence the property 1of Noncoarsencess is satisfied.

Property 2. There are many classes with different complexity value hence the property 2 of Granularity is satisfied.

Property 3. Let LinkedList_0 and LinkedList_2 are two different classes with same CCC value. CCC (LinkedList_0) = CCC (LinkedList_2) hence the property 3 of Non-uniqueness is satisfied.

Property 4. Let LinkedList_0 and LinkedList_01are the two classes with the same functionality, the choice of the number of methods, its internal complexity, number of external methods called, number of packages import, number of aggregations and number of queries may be different as it is a design decision and independent of the functionality. The CCC (LinkedList_0) =15 where as the CCC (LinkedList_01) =13.25, hence CCC (LinkedList_0) ≠ CCC (LinkedList_01) and thus property 4 that Design Details matter is satisfied.

Property 5. Let LinkedList_0 and LinkedList_1 are two different classes, the combined class is LinkedList_0and1 then CCC (LinkedList_0) =15, CCC (LinkedList_1) =9.75 and CCC (LinkedList_0and1)=18.25. Thus CCC (LinkedList_0) ≤ CCC (LinkedList_0 and1) and CCC (LinkedList_1) ≤ CCC(LinkedList_0and1) hence the property 5 of Monotonicity is satisfied.

Property 6. Let LinkedList_0 and LinkedList_2 are two different classes. CCC (LinkedList_0) and CCC (LinkedList_2) are same with the metric value 15. LinkedList_1 is the another class which is separately combined with LinkedList_0 and LinkedList_2 to get LinkedList_0and1 and LinkedList_2and1 the complexity value of CCC (LinkedList_0and1) = 18.25 and CCC (LinkedList_2and1)=17.14, so CCC (LinkedList_0) < CCC (LinkedList_0and1) and CCC (LinkedList_2) < CCC (LinkedList_2and1) hence the property 5 of Non Equivalence of Interaction is satisfied.

Property 8. The renaming of identifiers of a class could not affect the complexity value hence property 8 is satisfied.

Property 9. From property 5 of two classes LinkedList_0 and LinkedList_1, CCC (Linked List_0) = 15 and CCC (LinkedList_1) = 9.75 the summation of these two metric value are 24.75 which is lesser than the CCC value of LinkedList_0 and LinkedList_1 so the property 9 of Interaction can increase complexity is not satisfied as the complexity cannot be reduced it may increase.

Table 3 summarises the complexity metric result calculated from the metric values as presented in table1, where as table 4 shows the analytical evaluation result of different metrics.

Table 3 Complexity Metric Results

Class Name	**WMC**	**CMC**	**CC**	**CCC**
LinkedList_0	5	7	8	15
LinkedList_1	4	6.75	7.75	9.75
LinkedList_2	5	7	8	15
LinkedList_01	4	6.25	7.25	13.25
LinkedList_0and1	8	10.25	11.25	18.25
LinkedList_01and1	7	9.428571	10.42857	16.42857
LinkedList_2and1	7	9.142857	10.14286	17.14286

Table 4 Analytical Evaluation Results for Complexity Metrics

Weyuker's Property number	**WMC**	**CMC**	**CC**	**CCC**
1	√	√	√	√
2	√	√	√	√
3	√	√	√	√
4	√	√	√	√
5	√	√	√	√
6	√	√	√	√
7	Not Applicable	Not Applicable	Not Applicable	Not Applicable
8	√	√	√	√
9	X	X	X	X
√ Indicates that the metric satisfies the corresponding property. X Indicates that the metric does not satisfy the corresponding property.				

VI CONCLUSION

In this paper, an attempt has been made to define a complete class complexity metric (CCC) based on the definition of nine metrics defined at the class and interface level. An analytical evaluation of complete class complexity metric has been done based on Weyuker's properties. In this study we observe that all the metrics satisfy a majority of properties as described by Weyuker with exception to property 9 (interaction increases complexity). It has practical implications that it could not be satisfied in general. However, we could not found any practical example for metric that satisfied property 9.

The future work includes the empirical study of complexity metrics at system level. Validation of various versions of open source projects is being carried on as part of our ongoing work.

REFERENCES

[1] B. Kitechenham, SL. Pfleeger, NE. Fenton, "Towards a framework for software measurement validation", IEEE Trans. On Software Engineering 1995, 21(12), 929-944

[2] Barota, M. F., Troya, J.M., and Valleceillo, "Measuring the usability of Software Components, The Journal of System and Software, 2006

[3] Changjun Hu, Feng Jiao, Chongchong Zhao "An Architecture Quality Assessment for Domain-Specific Software". 2008 International Conference on Computer and Software Engineering.

[4] E.J. Weyuker, "Evaluating software complexity measures", IEEE Trans. On Software Engineering, 14, 1998, 1357-1365.

[5] Gurusaran and G. Roy, "On the Applicability of Weyuker property Nine to Object Oriented Structural Inheritance Complexity Metrics, "IEEE Trans. Software Eng., vol. 27, no 4 , pp. 361-364, Apr. 2001.

[6] J. Cherniavsky and C. Smith, "On Weyuker's Axioms for Software Complexity Measures , "IEEE Trans. Software Eng., vol. 17, no 6, pp. 636-638, June 1991.

[7] Kuljit Kaur Chachal, Hardeep Singh "A metric based approach to evaluating design of software component" 2008 IEEE International Conference on Global Software Engineering.

[8] L. Zhang and D. Xie, "Comments on the applicability of Weyuker's property 9 to Object-Oriented Structural Inheritance Complexity Metrics" IEEE Transaction on Software Engineering Vol 28, no 5, 2002, pp. 526-527

[9] M. Bunge, "Treatise on Basic philosophy Ontology 1: The Furniture of the World, Boston :Rediel, 1997.

[10] N.Sharma, P.Joshi and R.K. Joshi "Applicability of Weyuker's 9 to Object Oriented Metrics", IEEE Trans.on Software Engineering, Vol. 32, No. 3, March 2006.

[11] N.V. Balasubramanian, "Object-Oriented Metrics", Asian Pacific Software Engineering Conference (APSEC-96), December-1996, pp. 30-34

[12] Qingfeng Du and Fan Wnag "Software Power: A New Approach to Software Metrics" 2010 Second WRI World Congress on Software Engineering.

[13] S.R. Chidamber and C.F. Kemerer, "A Metric Suite for Object-Oriented Design", IEEE Trans. On Software Engineering, 20, 6(1994), 476-493.

[14] T.J.McCabe, "A Complexity Measure ", IEEE Transaction on Software Engineering, Vol. 2, 1976, pp. 308-320.

[15] W. Li, "Another metric suite for object-oriented programming", The Journal of Systems and Software 1998; 44(2):155-162.

