New Design Metrics for Complexity Estimation in Object Oriented Systems

Nisha Malik
Asstt. Professor, Deptt. of Computer Science, Govt. College for Women
Rohtak, Haryana (India)
nisha196@gmail.com

Rajender Singh Chhillar
Professor, Deptt. of Computer Science and Applications,
M.D. University, Rohtak, Haryana (India)

Abstract

This paper proposes four design metrics of class level for early and effective feedback to manage the quality of object-oriented software. These metrics measure the complexity induced by the use of various object-oriented design concepts like data hiding, aggregation, inheritance and cohesion. These four metrics are evaluated from the output of design phase before the starting of the coding to reduce the overall efforts in term of time, cost, and manpower. Although various set of metrics already exist to measure the design complexity of class and object-oriented system but maximum of them are evaluated from the source code not before actual coding. These measures are analytically evaluated against Weyuker's properties for object-oriented metrics. Metrics data is collected from two C++ projects as empirical sample for analysis.

Keywords- Object-oriented design; Design metrics; Complexity measurement

I. INTRODUCTION

The quantitative information forms the basis of every engineering discipline; software engineering is not an exception. Software measures/metrics are essential to manage the various processes of software development to produce a quality product. Some of the traditional metrics have been used to analyse the object-oriented codes, but are certainly not sufficient to characterize the all features unique to object-oriented approach. Metrics have already been proposed by various authors related to various object-oriented constructs e.g. encapsulation, coupling, cohesion, inheritance and polymorphism. Among the existing object-oriented metrics most are evaluated using source-codes i.e. later-stage information.

In object-oriented approach, each category of data and their corresponding operations are collected into a single unit using encapsulation. The concept of identification, classification and abstraction are used to format a class-structure. Polymorphism and inheritance are unique to object-oriented approach regarding the reusability of codes. Polymorphism means the same operator or method is used to perform different functions. Inheritance is the mechanism that permits the sharing of data and methods among classes based on a hieratical relationship. The in-built strength of the object-oriented approach happens due to the data-hiding along with reusability of codes but these concepts induce some level of complexity in design and codes. High complexity of classes/programs requires more effort in testing and maintaining the software. The quality of product is also affected by its complexity. The analysis and design phases of object-oriented approach of software development need more focus as compared to others. Feedbacks from the measures applicable in design phase are of intensive use. This need instigates the authors to propose new set of design metrics. The applicability of these metrics would strictly be restricted to the requirement and design phase. Metrics are illustrated with simple and suitable examples. The proposed set of four metrics are analytically evaluated against Weyuker's set of measurement principles. The data is collected from two C++ projects to analyse metrics empirically.

II. RELATED WORKS

Management through measurement is not a new concept, but basic principle of every engineering discipline. The use of measures/metrics in software engineering is started nearly in 1970's. Many researchers contributed to this area by introducing the measures for traditional approach. As industry grew, the improved and new approach of software development (like object-oriented approach) was gradually introduced. The main focus of object-oriented approach is on modeling the real world in terms of basic-entities i.e. objects and classes. The traditional metrics are observed to be inadequate to cover all the new and unique aspects of object-oriented approach. In this approach emphasis is given on design phase for more realistic-solution. Number of metrics have been used for object-oriented software for estimating design complexity and provides feedback to manage the quality of deliverable. R. Morris introduced few metrics, but unable to provide formal and testable definitions [1]. Similarly Moreau and Dominick [2] and some others [3,4] have done sound work by providing formal definition to the subject. Later Chidamber and Kemerer [5] developed and implemented a new set of metrics i.e. A Metrics Suite for Object-Oriented Design. Balasubramanian [6] propose three metrics and compared them with WMC, LCOM and RFC, introduced in Chidamber and Kemerer [5]. Some authors provided a sound contribution in this area, by introducing and evaluating the measures, that are capable of providing feedback in the early phases of object-oriented development cycle [7, 8, 9,10,11,12,13,14,15,16]. This study is also one step ahead in this direction.

III. NEWLY PROPOSED METRICS

As discussed above we introduce four design metrics, indicator of complexity and reusability at class level as a result of using the concepts; data-hiding, inheritance, coupling and cohesion, during object-oriented design. These metrics are as follows:

A. *Class Member Complexity Measure (CMCM)*

It is a measure of complexity through access-capabilities or data-hiding at class-level by public/protected data-members and member- functions. In a class all members are private by default. They are declared as private, public and/or protected depending upon the level of hiding the data and functions internal to the class via encapsulation. Hence public/protected members provide interface to class and are responsible for data-access from outside world through inheritance and message passing, complexity and reusability of data and/or functions.

CMCM of a class counts the number of data-member and member-function declared as public/protected.

$CMCM = N_d + N_f$

N_d = Number of public/protected data- members.

N_f = Number of public/protected member-functions.

Higher value of CMCM means high capacity of codes to be reused, lacking in hiding the data and more complexity of class.

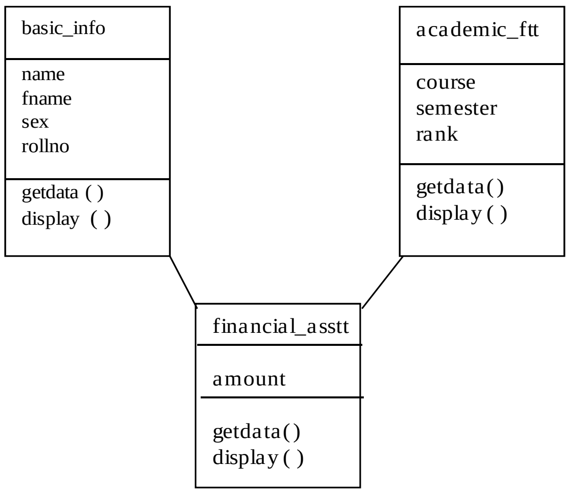

Figure I. Class-diagram of fee-management system

Class basic_info
Private: Long int rollno
Protected: char name, char fname, char sex
Public: void getdata( ), void display ( )

Figure 2. Details of basic_info class

From figure 2

Number of protected data-member = 3

Number of public data-member = 0

Number of protected member-function = 0

Number of Public member-function= 2

Hence CNCN= 3+0+0+2 = 5

B. Class Inheritance Complexity Measure (CICM)

It is a measure of class complexity induced by sharing the data/methods from other class(s) i.e. ancestor(s) through inheritance. As the depth of a class within its class-hierarchy increase, means more the use of inheritance or more number of ancestors. When a given class inherent properties of its ancestors, codes are reused. The implementation of inheritance permutes reusability and hence complexity. But complexity of a class also depends upon the type of inheritance (i.e. single/multilevel and multiple, shown in figure 3) used, along with its depth.

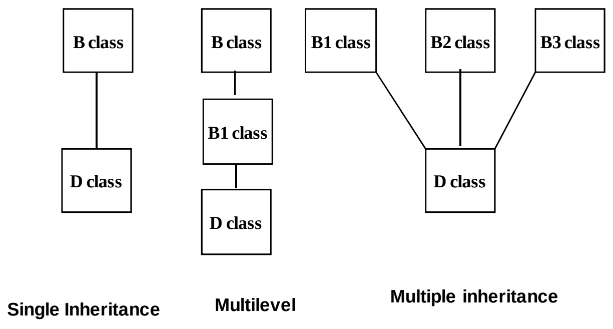

Figure 3. Types of inheritance

In single inheritance, a derived class shares the data/methods of only one root class. Multilevel inheritance means successive single inheritance i.e. certainly one or more than one class must play a dual role of derived and root class simultaneously. In multiple inheritances there is only one derived class and more than one root classes. In large size software, hybrid inheritance is used for proper code reusability. The hybrid inheritance means combination of any two or more than two type of inheritance like single + multiple, multiple + multilevel and single + multiple + multilevel inheritance.

The CICM calculates the individual complexity of a class due to the properties inherited from ancestors in inheritance hierarchy of system/program, using a general formula

$$CICM\begin{cases} \dfrac{0\ for\, i = 0}{n + \sum_{i=1}^{n} R_i\ \ for\ \ i \geq 1} \end{cases}$$

Where n= number of immediate ancestors of a class

R_i= CICM value of ith root-class .

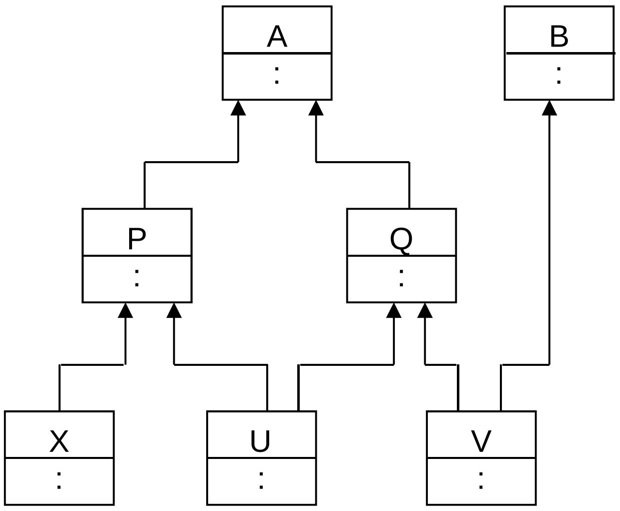

Figure 4. Inheritance hierarchy of system

From figure 4

CICM of class A= 0 as it has no ancestor

CICM of class B = 0 as it has no ancestor

CICM of class P= 1+0=1 as n=1 means it has only one immediate ancestor (i.e. class A) and its CICM value is 0.

Similarity

CICM of Class Q =1+0 =1} as immediate ancestor is class A

CICM of class X =1+1 =2} as immediate ancestors is class P

CICM of class U =2+1+1 =4} as immediate ancestors are class Q and class P

$$CICM\ of\ classV = \left.\begin{matrix} 2+1+0 \\ 3 \end{matrix}\right\}$$ as immediate ancestors are class Q and class B.

From figure 1:----

CICM of basic_info = 0 as it has no ancestor class.

CICM of academic_fit=0 as it has also no ancestor class.

CICM of financial_assit = $2 + \sum_{i=1}^{2} R_i$

= 2 + [R_1+R_2]

= 2+[0+0]=2

As financial_assit class has two immediate ancestor classes i.e. basic_info and academic_fit. Then R_1 and R_2 are the CICM values of basic_info and academic_fit classes respectively.

So, the value of CICM for a class depends upon following factors.

Depth of the class in the inheritance hierarchy.

Number of immediate ancestors of the class and their depth in the inheritance hierarchy.

C. Class Aggregation Level Measure (CALM)

It is a measure of complexity in a class through coupling that means actual usage relationship between the classes where one class uses the other. As aggregation is a tightly coupled form of instances association and is the "past-whole" or 'a-past-of" relationship. Thus aggregation is one of the mechanisms that constitute coupling in various frameworks for coupling measurement. Attributes or data-members declared as user-defined data-types (i.e. classes) establish "part-whole" relationship or aggregation.

CALM is the ratio of number of attributes declared as user-defined classes type to the total number of attributes declared in the class.

$$CALM = \frac{U_d}{N_d}$$

U_d= Number of attributes in the class declared as user defined type or another classes type.

N_d= Total number of attributes declared in the class.

If none of the attribute declared as another class type, then CALM=0. If all attributes of a class are declared as another class type, CALM=1.

From figure 2:-----

The basic_info class has four attributes, but none is user defined type (i.e. another class type). Thus

$$CALM = \frac{0}{4} = 0.$$ for basic_info class.

D. Class Cohesion measure (CCOM

It is a measure of complexity through class cohesiveness in terms of degree of similarity among methods in attribute usage. The class is said to be cohesive, if a class has different methods to perform different functions on the same set of data/instance-variables. Thus cohesiveness can be calculated in terms of attributes usage by various methods within the class i.e. more attributes/instance-variables are used by more number of methods of a class, more will be the cohesiveness of class. Parameter-list of methods in class is used to identify the attribute/instance-variable used by methods. A less cohesive class may split into number of classes.

CCOM is the ratio of actual-attributes sharing to the maximum attributes-sharing by all methods in the class.

$$CCOM = \frac{AA_{sc}}{MA_{SC}}$$

AA_{sc} = actual attribute-sharing count

MA_{sc} = maximum attribute-sharing count

Let a class C has n methods M_1, M_2……. M_n and m instance-variables V_1,V_2….. V_m.

Let I_i = set of all instance-variables used by M_i

So there are n such set {I_1} {I_2}…. {I_n}

and I_1UI_2 UI_3….I_{n-1}UI_n={V_1 V_2'V_3..V_m}.

N1= {(Number of Methods using V_1)-1}, N2= {(Numbers of Methods using V_2)-1} and so on.

Then

AA_{sc} = actual attribute-sharing count

= {(Number of Methods using V_1)-1} + {(Number of methods using V_2)-1}……… + {(No. of methods using Vm)-1}

= N1+N2+…..+N_M

MA_{sc} = maximum attribute-sharing count

= (Number of instance-variables) x {(number of methods)-1} = m × (n-1)

$$CCOM = \frac{N_1 + N_2 + ... + N_m}{m \times (n-1)}$$

As sharing of attribute means when more than one method use that attribute. Therefore

If none of the attribute is used by more than one method then CCOM is 0 and if all attributes are used by all methods, CCOM=1.

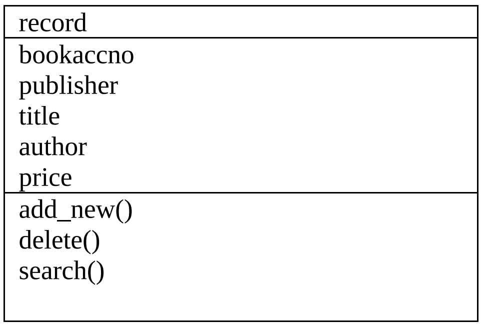

Figure 5

From figure 5

Class record has three methods i.e. M_1, M_2 and M_3 and have following set of instance variables:-

I_1, = {bookaccno, publisher, title, author, price}

I_2= {bookaccno}

I_3 = {title, author}

So

n=3 and m=5

N_1= {(Number of methods using bookaccno)} = 2-1=1

N_2= {(Number of methods using publisher) - 1} = 1-1 = 0

N_3= {(Number of methods using title)-1}

= 1-1 = 0

N_4= {(Number of methods using author)-1} = 2-1=1

N_5= {(Number of methods, using price)-1}

= 1-1 = 0

CCOM of class record

$$= \frac{N_1 + N_2 + N_3 + N_4 + N_5}{m \times (n-1)} = \frac{1+0+0+1+0}{5 \times 2} = 0.2$$

Similarly from figure 2

In basic_info class:-

Number of methods = n = 2

Number of attributes/instance-variables = m = 4

I_1= {rollno, name, fname, sex}

I_2={rollno, name, fname, sex} N_1=2-1=1

N_2=2-1=1

N_3=2-1=1

N_4=2-1=1

So CCOM of class basic_info

$$= \frac{N1 + N2 + N3 + N4}{m \times (n-1)} = \frac{1+1+1+1}{4 \times (2-1)} = \frac{4}{4} = 1$$

IV. EVALUATION OF METRICS

Four object-oriented metrics introduced in this paper are evaluated using a criteria suggested by Weyuker [17]. Several researchers have recommended some mathematical properties that should be possessed by a software metric, but mostly proved to be informal in evaluating metrics. Hence selected the formal list of properties recommended by Weyuker, used by various researchers [5, 6, 14, 16] for evaluating measures.

Weyuker suggests nine-properties; out of them first four check the sensitivity and discriminative power of metrics. The fifth property indicates that if two classes are combined, then their metric value should be greater than metric-value of each individual class. The sixth property is about interaction between two programs/classes. The seventh property states that a measure be sensitive to statement order within a program/class. The eighth property states that renaming of variables does not affect the metric-value. Ninth property states that sum of the metric values of programs/classes could be less than the metric value of program/class, when they are considered as single system. But the order of statement in the class definition has no impact on functionality, thus seventh property is irrelevant for object oriented metrics evaluated at class level. Increase in the number of classes, increases the design complexity of object oriented software. Thus ninth property is also not an essential feature for object oriented measures and these two properties are not needed to be satisfied [5].

Therefore the list of properties (developed by Weyuker) for evaluating object oriented metrics includes following:

Let *u* be metric of program/class *P* and *Q*.

Property 1: This property states that

(P),(Q)($u(P) \neq u(Q)$)

It ensures that no measure rates all program/class to be of same metric value.

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

Property 3: There are distinct program/class *P* and *Q* such that u(*P*) = *u*(*Q*) .

Property 4: For object-oriented system, two program/class having the same functionality

could have different values.

(P)(Q)($P \equiv Q$ and $u(P) \neq u(Q)$)

Property 5: When two program/class are concatenated, their metric should be greater

than the metrics of each of the parts.

(P)(Q) ($u(P) \leq u(P + Q)$ and $u(Q) \leq u(P + Q)$)

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

For program/class *P*, *Q*, *R*

(P)(Q)(R) ($u(P) = u(Q)$ and $u(P + R) \neq u(Q + R)$)

Property 7: It specifies that "if *P* is a renaming of *Q*; then *u*(*P*) = *u*(*Q*) "

A. Evaluation of CMCM on Weyuker's Properties

The two object-oriented classes P and Q, vary in their Public/Protected data-members and member-functions, and hence CMCM value. Thus property1 is satisfied. There are finite number of classes having either same number of pubic/protected data-members and member- functions or different number of public/protected data-members and member- functions but their sum is equal like 5+4=9 and 6+3=9 The CMCM value may be same for finite number of classes. Therefore property 2 is satisfied. There is a possibility that for two different classes P and Q, the CMCM value is same i.e. CMCM (P) = CMCM (Q). Thus property3 is satisfied.

The decision of declaring the data-member and member-function as public/protected is totally independent of functionality. It satisfies property 4.

Let CMCM (P) =p and CMC (Q) =q and CMCM (P+Q) =p + q - a

Where a is a function of the common public/protected member-functions and data-members between class P and class Q. But maximum value of a is min (p, q). Hence

$$CMCM\ (P+Q) \geq CMCM\ (P)$$

and $CMCM\ (P+Q) \geq CMCM\ (Q)$

Therefore satisfies the property 5.

Let CMCM (P) = p, CMCM (Q) = p and CMCM I = r

Then

CMCM (P+R) = p + r- α

CMCM (Q+R) = p+ r- β

Where α is the common public/protected data- member and member-function between P and R and β is common between Q and R. As $\alpha \neq \beta$

therefore CMCM (P+R) $\neq$ CMCM(Q+R) and property 6 is satisfied .Renaming of class does not affect the value of metric, as CMCM is measured at class level, thus satisfy property 7.

B. Evaluation of CICM on Weyuker's properties

The two object oriented classes P and Q vary in their depth in inheritance hierarchy and the number of immediate ancestors. As depth of a derived class is always greater than root class and number of ancestors (immediate) may be different. Thus property 1 is satisfied. There is finite number of classes having same value of CICM as siblings exist in the hierarchy. Thus property 2 is satisfied. There is a possibility for two class P and Q such that CICM (P) =CICM (Q). It satisfies property 3. Depth of inheritance hierarchy and number of immediate ancestors of a class are independent of functionality, but dependent on design implementation. Therefore property 4 is satisfied.

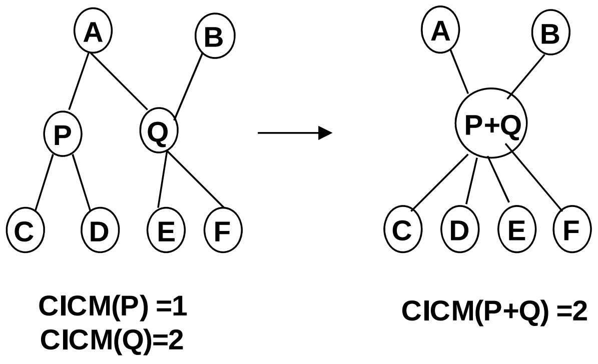

Figure 6

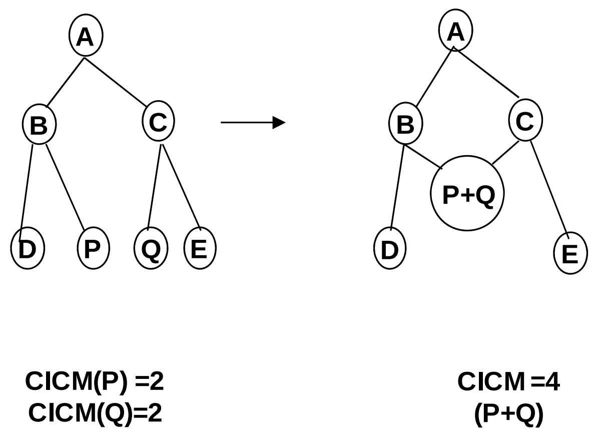

Figure 7

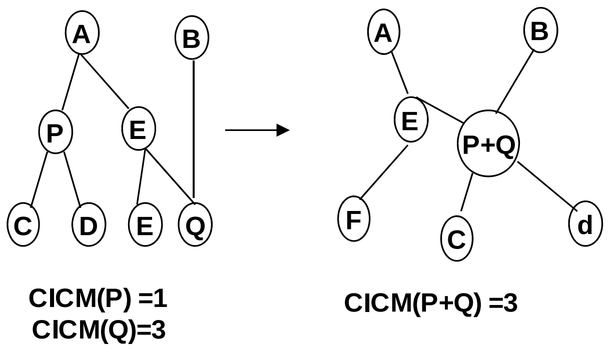

Figure 8

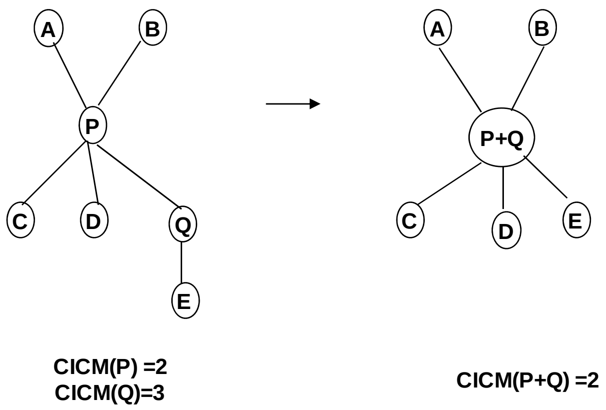

Figure 9

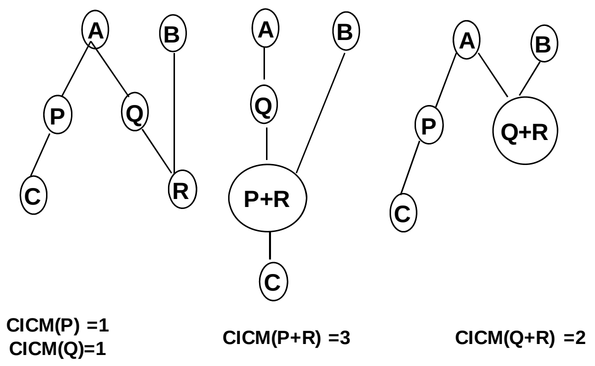

Figure 10

So, from figure 6 to figure 10 :----

$\text{CICM (P+Q)} \geq \text{CICM (P)}$

And $\text{CICM (P+Q)} \geq \text{CICM (Q)}$

Except when P & Q are in ancestor-decendent relationship (i.e. one is child of other). Thus property 5 is satisfied for classes P & Q except when one is child of other. If CICM (P) =p & CICM (Q) =p and a class R. combine with class P and class Q separately. Hence from fig. X CICM (P+R) $\neq$ CICM (Q+R), thus property 6 is satisfied. Renaming of class does not affect CICM value. Thus property 7 is satisfied.

C. Evaluation of CALM on Weyuker's Properties

Any two object oriented classes P and Q can have different number of attribute declared as user-defined classes type and total attributes. Hence property 1 is satisfied. The two object oriented classes P and Q can have same value of CALM i.e. CALM (P) =CALM (Q) =n. So there is finite number of object oriented classes having same metric value. Therefore property 2 is satisfied. There is non-zero probability that CALM (P) =CALM (Q). It satisfies property 3. The decision of declaring the data and their type is dependent on design criteria not on functionality, thus CALM satisfies property 4.

If CALM (P) =p and CALM (Q) =q

and CALM (P+Q) =p+q-α

Where α is function of number of common attributes of P and Q classes. So only for some cases:-

$\text{CALM (P+Q)} \geq \text{CALM (P)}$

And $\text{CALM (P+Q)} \geq \text{CALM (Q)}$

Thus does not satisfy property 5.

If CALM (P) =p, CALM (Q) =p and a class R combine with class P and class Q separately, then

$\text{CALM (P+R)} \neq \text{CALM (Q+R)}$

As class R shares unequal attributes with class P and class Q. Therefore property 6 is satisfied. The remaining of class does not change the value of CALM, hence property 7 is satisfied.

D. Evaluation of CCOM on Weyuker's Properties

The two object-oriented classes have different number of attributes, methods and methods sharing on attributes. Hence Property 1 is satisfied. Let COM (P) = n then there are two and more classes must exist, having CCOM value as n. It satisfies property 2. There is a possibility that two object-oriented classes P and Q have same number of attributes, methods and methods sharing on attributes. Then CCOM (P) =CCOM (Q), thus property 3 is satisfied. The decision about number of attribute, their usage and sharing is design dependent means independent of functionality. Therefore Property 4 is satisfied.

If CCOM (P) = p and CCOM (Q) =q

and CCOM (P+Q) = p + q-α

where α is the function of number of methods, number of attributes and methods sharing on attributes within the class and CCOM is a ratio metric, so in few cases—

CCOM (P) $\geq$ CCOM (P+Q)

And CCOM (Q) $\geq$ CCOM (P+Q)

Thus does not satisfy property 5. If CCOM(P)=p and CCOM(Q)=p and class R, combines with class P and class Q, then CCOM(P+R) $\neq$ COM(Q+R), because class R does not have identical methods and attributes common with class P and Q. Thus satisfies property 6. As renaming of class/entity, does not alter the CCOM-value. Thus satisfies the Property 7.

Thus all Weyuker's properties (1 to7) except fifth given above are satisfied by four newly introduced measures. The fifth property is also satisfied by one measures i.e. CMCM, but other three does not satisfy it.

V. EMPERICAL ANALYSIS OF METRICS

After evaluating the newly introduced metrics on Weyuker's properties, their values are calculated for commercial C++ projects from a telecom service provider, considering the privacy matters of company, they are referred as project A1 and Project A2 in this study, they contains 20 and 26 classes respectively. The data needed for the evaluation of metrics was collected manually.

A. Analysis of CMCM on collected data

The summary statistics of CMCM form both project is shown by Table I. Histograms in figure 11 and figure 12 show distribution of CMCM values for Project A1 and Project A2 by taking percentage of classes along Y-axis.

Table I: Descriptive statistics for CMCM

	Metric	Min.	Max.	Mean	Median	Mode	Std. dev.
Project A1	CMCM	2.00	10.00	5.65	5.00	4.00	2.37
ProjectA2	CMCM	0.00	36.00	9.85	7.00	2.00	8.57

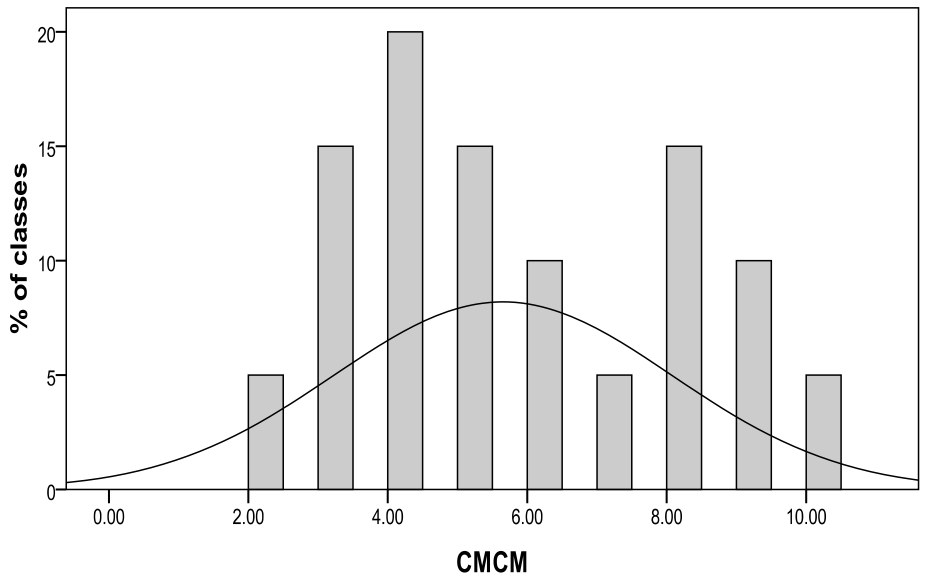

Figure 11. Graphical representation of CMCM values for project A1

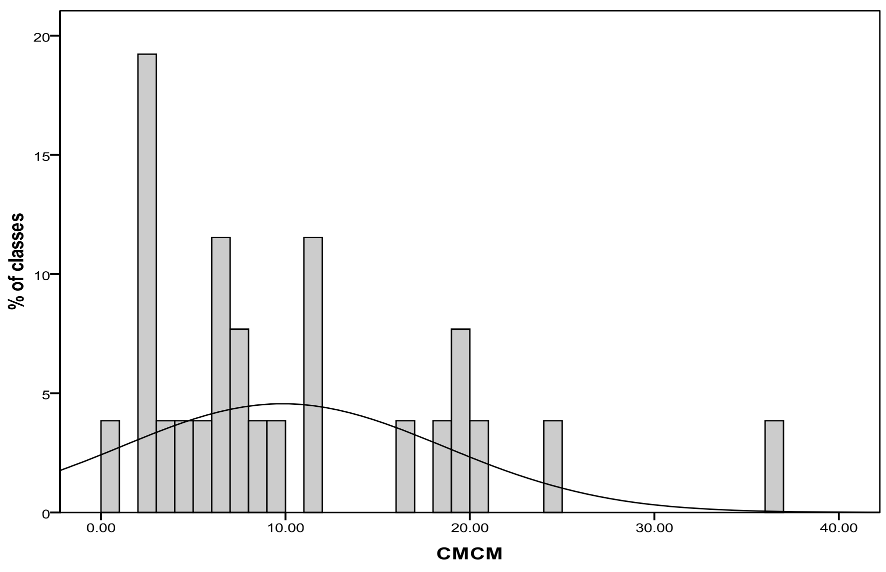

Figure 12. Graphical representation of CMCM values for project A2

From table I, difference between Minimum and Maximum values, shows that CMCM values are more spreader for project A2 as compared to project A1 and standard deviation-values also indicates the same. Higher mean value of CMCM for project A2 indicates that programmers are liberal in declaring the public/projected data members and member functions inside class i.e. lack of data hiding, but more reusable capability. Thus in general project A2 classes are more complex as compared to the classes of project A1.

B. *Analysis of CICM on collected data*

The histograms are shown in figure 13 and figure 14 for projects A1 and A2, and descriptive statistics

Table 2. Descriptive statistics for CICM

	Metric	Min.	Max.	Mean	Median	Mode	Std.dev.
Project A1	CICM	0	4	1.80	2.0	1.0	1.36
ProjectA2	CICM	0	1	0.23	0.00	0.00	0.43

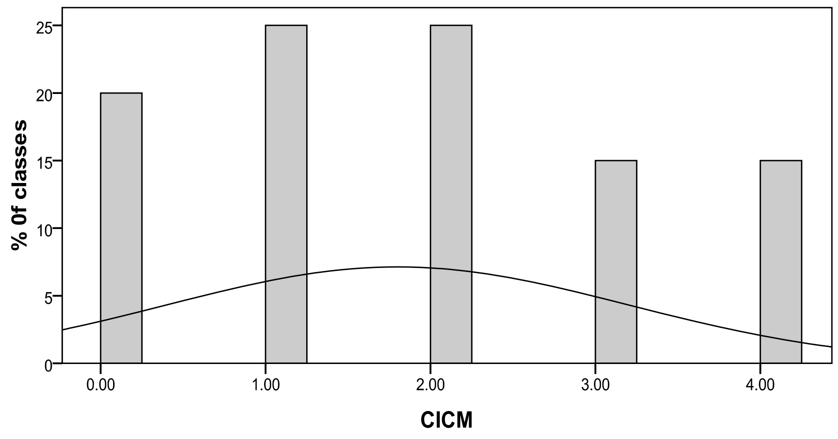

Figure 13. Graphical representation of CICM values for project A1

are in table 2. Both histograms and statistical results provide clear indication of low level of complexity induced by inheritance in classes of project A2 and nearly 78% of classes are root classes having 0 value of CICM and design is heavy at top. In project A1 nearly 20% of classes have zero value means top is not so heavy. As maximum values are 4 and 1 for projects A1 and A2 indicates that the programmer of project A1 has makes use of sharing the methods/attributes through inheritance, but up to few levels in the manageable way. Hence the classes of project A2 are making very little use of inheritance and therefore fewer complexes as compared to the classes of project A1.

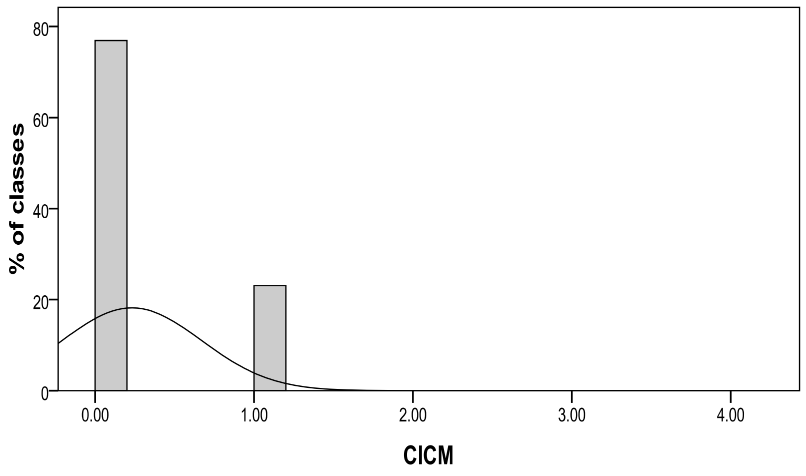

Fig. XIV Graphical representation of CICM values for project A2

C. Analysis of CALM on collected data

The histograms (figure 15 and figure 16) and descriptive statistics (table 3) for both projects is given below. The CALM data, their descriptive statistics and histograms for both projects indicate that concept of aggregation is used more intensively in project A1. In project A2 most of the classes do not use aggregation and others use at very low level. High is the level of aggregation, more capable will be the class to be coupled with other classes and constitute more complexity. Therefore, in general, classes of project A1 have more complexity as compared to project A2.

Table 3 Descriptive statistics for CALM

	Metric	Min.	Max.	Mean	Median	Mode	Std. dev.
Project A1	CALM	0	1	0.34	0.29	0.00	0.31
ProjectA2	CALM	0	0.5	0.06	0.00	0.00	0.13

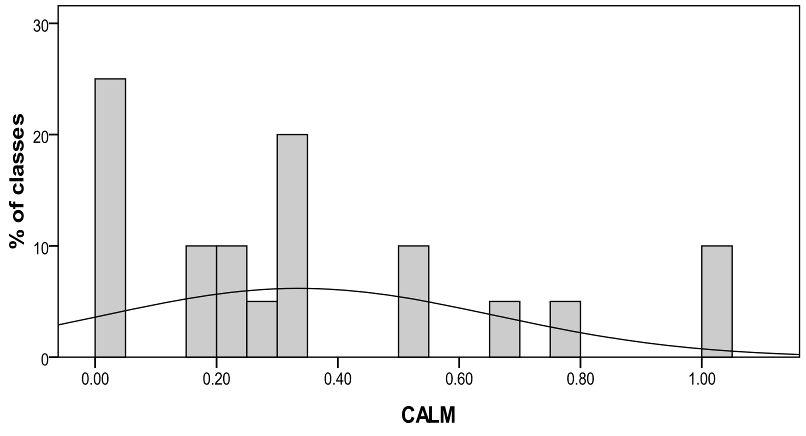

Figure 15. Graphical representation of CALM values for project A1

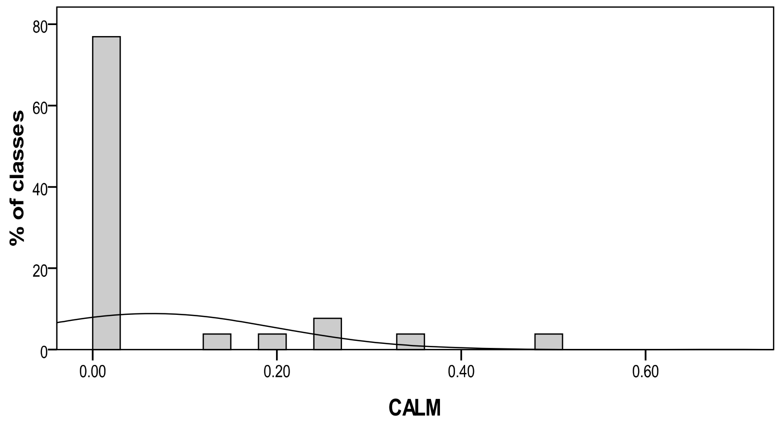

Figure 16. Graphical representation of CALM values for project A2

D. *Analysis of CCOM on collected data*

The class-wise CCOM data from projects A1 and A2 is graphically represented by histograms in figure 17 and figure 18. They depict that range of data for project A2 is 0 to 1 and for project A1 is 0.50 to o.92. This is also indicated by minimum, maximum and standard deviation values shown by table 4. The mean values for projects A1 and A2 are 0.74 and 0.52 respectively and it means less stress on designing cohesive classes in project A2. In general the classes of project A1 have more attribute sharing among methods within the class and are more cohesive and have few complexes.

Table 4. Descriptive Statistics for CCOM

	Metric	Min.	Max.	Mean	Median	Mode	Std. dev.
Project A	CCOM	0.50	0.92	0.74	0.75	0.80	0.12
ProjectA2	CCOM	0.00	1.00	0.52	0.59	0.00	0.35

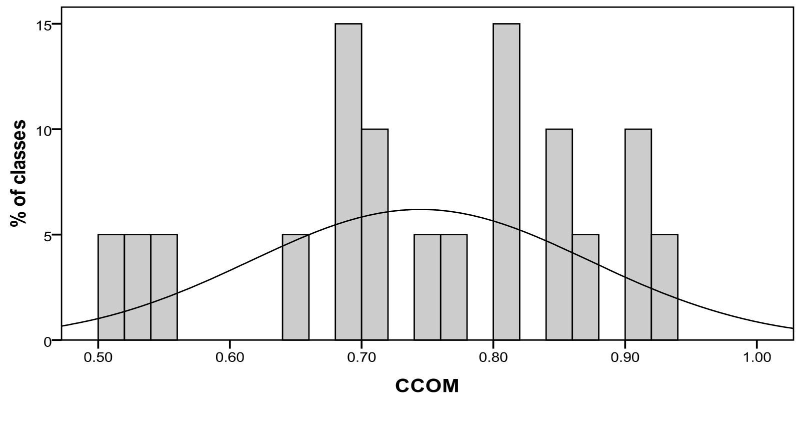

Figure 17. Graphical representation of CCOM values for project A1

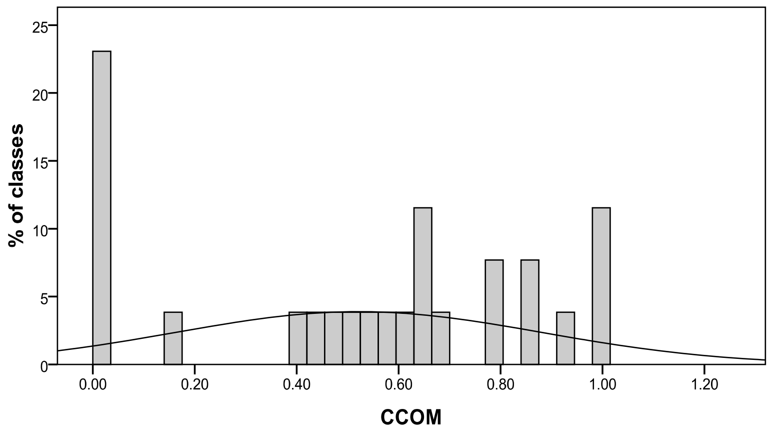

Figure 18. Graphical representation of CCOM values for project A2

Therefore analyzing the four measures using data collected from two C++ projects, it is observed that on the basis of two measures (i.e. CMCM and CCOM) classes of project A1 are less complex. But on the basis of other two measures (i.e. CICM and CALM) the classes of project A2 are less complex.

VI. CONCLUDING REMARKS AND FUTURE WORK

Software has its own importance in everyone’s life, their cost and reliability is a prime issue for software industry. This can be resolved by using improved and efficient approach of software development i.e. object-oriented approach and getting effective and timely feedback from software metrics. Reusability of codes are key concept of object-oriented approach but induce complexity and hence we need to make balance between the two for a good design. The study devised four object-oriented metrics for evaluating the complexity at class-level in design phase, induced through concept of data-hiding, inheritance, aggregation and cohesion. The small values of metrics indicate that the developer have not made sufficient effort in reusing the data-members and member-functions of a class. The complexity is directly proportional to values of CMCM, CICM and CALM, but highest value of CCOM means least complexity. The complexity management at time of class declaration/construction in design phase is able to control the cost, time and quality of software with the use of proposed metrics by reducing testing and maintenance efforts. These are evaluated on Weyuker’s properties and data from C++ projects. To further extend, these metrics can be implemented on other projects of different object-oriented languages and be associated with external attributes of product like reliability.

References

[1] R. Morris, “*Metrics for object oriented software development*”. Master thesis M.I.T, Sloan School of Management, Cambridge, M A.,1988.
[2] R. Moreau and W. D. Dominick, “Object oriented graphical information systems: research plan and evaluation metrics”. *Journal of Systems and Software,* vol 10, pp. 23-28, 1989.
[3] S. R. Chidamber and C. F. Kamerer, “Towards a metrics suite for object oriented design”. In Proc. 6th ACM conf. object oriented programming. Syst. Lang. and application.(OOPSLA), Phoenix, AZ,197-211, 1991.
[4] S. Whitmire, "Measuring complexity in object-oriented software," presented at theThird Int. Conf. Applicat. Software Meas., La Jolla, CA, 1992.
[5] S. R. Chidamber and C. F. Kamerer, “A metrics suite for object-oriented design”. *IEEE Transactions on Software Engineering,*vol 20, no. 6, pp. 476-493,1994.
[6] N. V. Balasubramnian, “*Object-oriented metrics*”. In proceeding of the 3rd Asia-Pacific Software Engineering (pp30-34), Washington, DC: IEEE Computer Society, 1996.
[7] Henderson-sellers, *Object-Oriented Metrics, Measures of Complexity*. New Delhi: Prentice Hall, 1996.
[8] L. C. Briand, P. Devanbu and W. Melo, “ *An investigation into coupling measures for C++*”. In proceedings of the 19th International Conference on Software Engineering, 18-23 May, Boston, MA, 1997.
[9] W. Li, “Another metrics suite for object oriented programming”. *Journal of Systems and Software*, vol 44, no.*2*, pp. 155-162, 1998.
[10] L. C. Briand and, J. Wust, “Modeling development effort in object-oriented systems using design properties”, *IEEE Transactions on Software Engineering,* vol 27, pp. 963–986, 2001.
[11] Jagdish Bansiya and Carl G. Davis, “A hierarchical model for object oriented design quality assessment”, *IEEE Transactions on Software Engineering,* vol 28, no 1, pp 4-18, 2002.
[12] S. Counsell, E. Mendes, S. Swift and A. Tucker, “Evaluation of an object-oriented cohesion metrics through Hamming distances”. Tech. Rep. BBKCS 2-10, Brikbeck College, University of London, UK, 2002.
[13] Koru, A.G. and Tian, J. 2003. An empirical comparison and characterization of high defect and high complexity modules. *Journal of Systems and Software,* vol. 67, pp. 153–163.

[14] K. K. Aggarwal, Yogesh Singh, Arvind Kaur and Ruchika, Malhotra, “Empirical study of object-oriented metrics”, *Journal of Object Technology*, vol *5*, no. 8, pp. 149-173, 2006.
[15] S. Misra and A. K. Misra, “Evaluation and comparison of cognitive complexity measure”. *ACM SIGSOFT Software Engineering Notes*, vol *32*, no. 2, pp. 1-5, 2007.
[16] Jitender Kumar Chhabra and Varun Gupta, “ Evaluation of object-oriented spatial complexity measures”, vol 34, no.3, pp. 1-5, 2009. dio:10.1145/1527202.1527208
[17] Weyuker, “Evaluating software complexity measures”. *IEEE Transactions on Software Engineering*, vol 14, no.9, pp. 1357- 1365, 1988