Chapter 5

Towards the Proposed Design Metrics

5.1 Introduction

Management through measurement is a basic principle of every engineering discipline including software engineering. Software 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 the features unique to object-oriented approach. In this 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 hierarchical relationship. So the in-built strength of the object oriented approach happens due to the data-hiding along with reusability of codes, but through them some level of complexity is induced in object-oriented design and codes. High complexity of classes/programs results in low quality and requires more effort in testing and maintenance. The analysis and design phases of object-oriented approach of software development need more focus as compared to others for more realistic-solution. Feedbacks from the measures applicable in design phase are of intensive use. Some metrics have already been proposed related to various object-oriented constructs e.g. encapsulation, coupling, cohesion, inheritance and polymorphism, are validated theoretically and empirically for optimal utilization (Chidamber and Kemerer, 1994; Balasubramanian 1996; Henderson, 1996; Briand et al., 1997 ; Li, 1998 ; Briand and Wust 2001; Bansiya et al., 2002; Counsell et al., 2002; Koru and Tian 2003; Aggarwal et al., 2006; Misra and Misra 2007; Chhabra and Gupta 2009). The previous researches,
analysis result of Chapter 4 and observations of review paper entitled ‘Identification and Implementation of existing object oriented software metrics for quality estimation’ (Singh & Nisha, 2009) shows the similar trends that the existing object-oriented metrics have fault prediction capability, but most of them are evaluated using source-codes i.e. later-stage information especially the metrics used to measure class size/complexity, coupling and cohesion, which are found more significant to predict the faults at class level. This justifies the need to introduce a new set of design metrics that use the static-information available the structural properties. This research proposes four metrics of class level for early and effective feedback to manage the quality in terms of reliability. These metrics measure the complexity induced by the use of various object-oriented design concepts like data hiding, aggregation, inheritance and cohesion which are capable to implement the dependencies among and with-in classes. The applicability of these metrics would strictly be restricted to the requirement and design phase. This chapter introduces new metrics having theoretical basis for their proposition explained using various constituents of cognitive theory inducing complexity, discussed in chapter 3. Metrics are illustrated with simple and appropriate examples. The proposed four metrics are theoretically evaluated using two successive frameworks given by Weyuker and Briand et al., as per research design and their results are also summarized in this chapter.

5.2 Newly Proposed Metrics

As discussed above four design metrics are introduced, early indicator of complexity and reusability at class level as a result of using the concepts; data-hiding, inheritance, aggregation and cohesion, during object-oriented design. In this section new metrics are defined with some theoretical basis and are illustrated with suitable examples. To explain theoretical background of each metrics development, constituents of basic cognitive theory for object-oriented software measures are used. The metrics are as follows:

5.2.1 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.

\[ \text{CMCM} = N_d + N_f \]

\[ N_d = \text{Number of public/protected data- members.} \]

\[ N_f = \text{Number of public/protected member-functions.} \]

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

**Theoretical basis of CMCM:**

Cognitive theory is considered as theoretical basis to propose the class-member complexity measure and their contribution in inducing the fault at class-level in design phase. The total number of public and protected data members and member-functions indicate the interface-size and access capability from outside the class. More the value of CMCM means higher will be the probability of interface effect and fan effect while dealing with class. It becomes difficult to recall and understand the class and more efforts are needed to comprehend the class. This means higher CMCM values are responsible for induction of more fault within the class-structure during the it’s design and will be an effective measure of design-complexity for early management of quality attributes.
Fig. 5.1 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()

From fig. 5.2

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
5.2.2 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 fig. 5.3) used, along with its depth.

![Fig. 5.3 Types of Inheritance](image)

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
Theoretical basis of (CICM):

As in the cognitive theory some effects; interference, fan, familiarity and memory span explain the association of any metrics with induced faults through recall and comprehension.

The induced fault in a class, is an effective indicator of product quality attributes. Higher value of CICM for a class means overall effect of level of class in inheritance hierarchy and it's immediate ancestors count contribute more complexity. The class deeper in the hierarchy are more familiar, hence it is easy to recall and comprehend. When a class is easy to comprehend, less number of faults are induced in it. The higher value of CICM is due to deeper class and/or more number of its immediate ancestors. The class complexity due to inheritance is directly associated with induced faults due to
interference and fan effect, although it may be a familiar class having less probability of inducing faults. Thus this metric is valuable indicator of external attributes through induced faults in the class on the basis of cognitive theory in early phases of developments.

From fig. 5.4

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

\[
CICM \text{ of class } V = \frac{2 + 1 + 0}{3}
\]

as immediate ancestors are class Q and class B.

From fig. 5.1:

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.

5.2.3 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.

\[ \text{CALM} = \frac{U_d}{N_d} \]

- \( U_d \) = 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.

**Theoretical basis of CALM:**

The proposition and utility of CALM is explained on the cognitive complexity induced in the class by the use of aggregation concept. The aggregation is one form of association and can be the reason for coupling of two or more classes. When one and more than one attribute of a class are declared as another class(s) type. The interference and fan effects due to the various links developed through aggregation, make the recall of class difficult. If it is difficult to recall a class means the class is difficult to understand.
and comprehend, therefore the class has more cognitive complexity. More the level of aggregation implemented in a class, more number of faults are induced due to higher cognitive complexity.

From fig5.2:

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

\[ \text{CALM} = \frac{0}{4} = 0. \text{ for basic_info class.} \]

5.2.4 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.

\[ \text{CCOM} = \frac{A_{A_{sc}}}{M_{A_{sc}}} \]

\[ A_{A_{sc}} = \text{actual attribute-sharing count} \]
\[ M_{A_{sc}} = \text{maximum attribute-sharing count} \]

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

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

So there are n such set \( \{I_1\} \{I_2\} \ldots \{I_n\} \)

and \( I_1 \cup I_2 \cup I_3 \ldots I_{n-1} \cup I_n = \{V_1, V_2, V_3, \ldots, V_m\} \).
N1= \{(Number of Methods using V_1)-1\}, N2= \{(Number of Methods using V_2)-1\} and so on.

Then

\[ AASC = \text{actual attribute-sharing count} = (N_1-1) + (N_2-1) + \ldots + (N_m-1) = N_1+N_2+\ldots+N_m \]

\[ MASC = \text{maximum attribute-sharing count} = (\text{Number of instance-variables}) \times (\text{number of methods}-1) = m \times (n-1) \]

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

\[ CCOM = \frac{N_1 + N_2 + \ldots + N_m}{m \times (n-1)} \]

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.

**Theoretical Basic of CCOM**

As defined above class cohesion measure is actual attributes sharing to total attributes sharing by all methods within a class. If all methods are interrelated through attribute sharing means higher value of CCOM within the class, then the class act as independent-module to perform the functions. The more cohesive class has least link with other classes and highly interrelated elements within it. Higher CCOM value of a class has low cognitive complexity due to least interference and fan effect. This class has less induced faults inspite of high CCOM value. As CCOM metrics can be theoretically connected to the cognitive complexity, cause of induced faults during class-design. Thus this can be effective measure of quality attribute of product and helpful in managing them in early phase of development.
From *fig. 5.5*

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

\[ I_1 = \{ \text{bookaccno, publisher, title, author, price} \} \]

\[ I_2 = \{ \text{bookaccno} \} \]

\[ I_3 = \{ \text{title, author} \} \]

So

\[ n = 3 \text{ and } m = 5 \]

\[ N_1 = (\text{Number of methods using bookaccno}) - 1 = 2 - 1 = 1 \]

\[ N_2 = ((\text{Number of methods using publisher}) - 1 = 1 - 1 = 0 \]

\[ N_3 = ((\text{Number of methods using title}) - 1) = 1 - 1 = 0 \]

\[ N_4 = ((\text{Number of methods using author}) - 1 = 2 - 1 = 1 \]

\[ N_5 = ((\text{Number of methods, using price}) - 1) = 1 - 1 = 0 \]

CCOM of class record
Similarly from fig. 5.2

In basic_info class:

Number of methods = \( n = 2 \)

Number of attributes/instance-variables = \( m = 4 \)

\( I_1 = \{ \text{rollno}, \text{name}, \text{fname}, \text{sex} \} \)

\( I_2 = \{ \text{rollno}, \text{name}, \text{fname}, \text{sex} \} \)

\( N_1 = 2 - 1 = 1 \)

\( N_2 = 2 - 1 = 1 \)

\( N_3 = 2 - 1 = 1 \)

\( N_4 = 2 - 1 = 1 \)

So CCOM of class basic_info

\[
\frac{N_1 + N_2 + N_3 + N_4}{m \times (n-1)} = \frac{1 + 1 + 1 + 1}{5 \times 2} = \frac{4}{10} = 0.4
\]

5.3 Theoretical Evaluation of Metrics

Theoretical evaluation is done to validate the metrics internally. This means to check that the metrics measure what it is suppose to measure using some mathematical properties. 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 (1988), used by various researchers (Chidamber & Kemerer, 1994; Balasubramanian, 1996; Aggarwal et al. 2006; Chhabra and Gupta 2009) for evaluating software metrics. The properties introduced by Weyuker are general to all good software metrics, still they are used to evaluate some metrics to measure complexity of object-oriented design. Metrics proposed by us are design metrics to measure complexity at class-level. Briand et al.
(1996) have provided a list of five properties for good complexity measure. Thus to provide more credibility to new design metrics, they are evaluated using two successive frameworks introduced by Weyuker and Briand et al. as discussed in chapter 3. Evaluation results are as follows:

5.3.1 Evaluation of Metrics using Weyuker's Framework

Weyuker suggests nine-properties; out of them first four check the sensitivity and discriminative power of metrics. The fifth property is used to check the monotonicity of metrics, sixth property is about interaction between two programs/classes. The seventh property is to check the effect of permutation of elements within the program/class. The eighth property states that renaming of variables does not affect the metric-value. Ninth property is to access the complexity change during interaction across classes within a program.

Comprehensive details of Weyuker's properties for evaluating object oriented metrics is given in Chapter 3.

5.3.1.1 Evaluation of CMCM: 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 public/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. Hence 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 property 3 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

\[ CMCM (P+R) = p + r - \alpha \]

\[ CMCM (Q+R) = p + r - \beta \]

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

so \( CMCM (P+R) \neq CMCM(Q+R) \) 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.

5.3.1.2 Evaluation of CICM: 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.
Fig. 5.6

CICM(P) = 1
CICM(Q) = 2

Fig. 5.7

CICM(P) = 2
CICM(Q) = 2

Fig. 5.8

CICM(P) = 1
CICM(Q) = 3
So, from fig 5.6 to fig 5.10:

\[ \text{CICM}(P+Q) \geq \text{CICM}(P) \]

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

Except when \( P \) & \( Q \) are in ancestor-descendent 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 \( \text{CICM}(P) = p \) & \( \text{CICM}(Q) = p \) and a class \( R \) combine with class \( P \) and class \( Q \) separately. Hence from fig. 10 \( \text{CICM}(P+R) \neq \text{CICM}(Q+R) \), thus property 6 is satisfied. Renaming of class does not affect CICM value. Thus property 7 is satisfied.
5.3.1.3 Evaluation of CALM: 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:

CALM (P+Q) ≥ CALM (P)

And CALM (P+Q) ≥ 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

CALM (P+R) ≠ 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.

5.3.1.4 Evaluation of CCOM: 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 - \alpha

where \alpha 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.

5.3.2 Evaluation of Metrics using Framework of Briand et al.

Briand et al have given specific properties to evaluate the complexity metrics. This framework includes five properties namely Nonnegative, Null value, Symmetry, Module monotonicity and disjoint module additivity (Briand et al. 1996). Before evaluating the design metrics against above properties I have to redefine the basic definition of system, module and complexity used in the framework. System: An object-oriented system is represented as a set of classes as its element and interrelationship between various classes. Module: It is taken as class of an object-oriented system. Complexity: The four design metrics are considered for estimation of complexity of an object-oriented system. Complete details of Briand et al. framework for evaluating object-oriented complexity metrics is discussed in chapter 3.

5.3.2.1 Evaluation of CMCM: The CMCM is defined as total sum of public and protected data members and methods, their minimum value will be zero, but not any negative number, so properly 1 is satisfied.
In an object-oriented system $S$, if there is no relationship of given class with any class, means the class has no interface with outside world and no public and protected class members here $CMCM=0$. Therefore property 2 is satisfied. Number of public and protected data members and member functions in a class is used to evaluate $CMCM$, no specific convention is used to show the interface of the class with other classes in the software. Hence property 3 is satisfied.

In an object-oriented system $S$, the complexity increases with the number of classes. Here property 4 is analogous to property 9 of Weyuker's framework. As discussed in previous section that $CMCM$ does not satisfy the Weyukers property 9 thus will not satisfy the property 4 also. $CMCM$ is devised for purely object-oriented environment.

If two classes are combined to form a single class, then the complexity of resultant class will be equal to the sum of $CMCM$ value of individual classes as two classes are disjoint. The disjoint classes have no common set of data member and member functions. Thus $CMCM$ satisfies the property 5.

### 5.3.2.2 Evaluation of CICM:

The number of immediate ancestors of a class never be a negative number. Similarly the depth or level of class in inheritance hierarchy may be zero and any positive whole number. Thus $CICM$ value will be always a non-negative number as per its definition discussed earlier. Therefore $CICM$ satisfies the property 1.

If a class does not inherit data and methods from any other class means the class has no relationship of inheritance type. The $CICM$ value of this class is zero, as it does not have any ancestor. So, there will be zero complexity due to inheritance, when no relationship exist of inheritance type. Therefore $CICM$ satisfies the property 2. When additional relationship are created across the classes. The value of $CICM$ are not expected to decrease since this class inherit more data and methods from more ancestors. Hence property 3 is satisfied. The property 4 i.e. module monotonicity is analogous to Weyuker's property 9. This property is not satisfied by $CICM$ as this metric does not satisfy the Weyuker's property 9, discussed in the earlier section.
When two disjoint classes are combined to form a resultant class, then the CICM
dvalue of resultant class will be the sum of CICM values of individual classes as sharing
factor is nil between two disjoint classes. Therefore property 5 is satisfied.

5.3.2.3 Evaluation of CALM: In a class there is a possibility that none of the
attributes is declared as another classes type. The minimum value of CALM is 0 and
maximum value is 1. So CALM values are always non-negative number. This satisfies
the property 1.

Aggregation is one form of relationship and if this type of relationship is zero
means none of the attributes of a given class is declared as another classes type. Thus
CALM value, which measures the complexity of class due to aggregation is equal to zero.
Thus property 2 is satisfied by CALM.

Additional relationship across various classes cannot expected to decrease the
value of CALM. Since more attributes out of the total attributes declared as instance-
variables in class to be of another classes type. Hence property 3 is also satisfied by this
metric. Property 9 introduced by Weyuker's is not satisfied by CALM, as explained
earlier. The property 4 of this framework is analogous of above and therefore not
satisfied by CALM.

When two disjoint classes are combined to form a resultant class none of the
attribute of any class can be declared as other classes type. This means no attribute is
common between two classes, declared as another classes type, thus CALM of resultant
class is sum of CALM values of individual classes only. Therefore property 5 is satisfied.

5.3.2.4 Evaluation of CCOM: The values of CCOM lies between 0 and 1, this indicates
that CCOM values are always non-negative number. Therefore satisfies the property 1.

If inter-relationship between different methods within a class on the basis of
attributes usage does not exist, then there is no actual attributes sharing between the
methods of a class. The CCOM value for that class is zero, indicates that property 2 is
satisfied.

Cohesiveness of class is not expected to decrease due to some additional
relationship between various methods of a class, since these methods are more
interrelated via attribute usage. Thus property 3 is satisfied. Similar to other three metrics, CCOM has not satisfied the Weyuker’s property 9, therefore CCOM dissatisfies the property 4 of this framework.

When two classes showing no relationship between them i.e. disjoint classes are merged to form a resultant class then sharing factor of cohesion in the resultant class is nil, as unrelated method from two classes are encapsulated. Thus the CCOM value of resultant class is equal to the sum of CCOM values of individual classes only and satisfies the property 5.

5.4 Results

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 measures. Reusability of codes is a key concept of object-oriented approach but induces complexity and hence we need to make balance between the two for a good design. If design flows are propagated to codes and then design complexity is evaluated from source codes, time, budget and efforts increase manyfold to overcome these flows. Four object-oriented metrics are devised for evaluating the complexity at class-level in design phase, induced through concept of data-hiding, inheritance, aggregation and cohesion. The classwise small values of metrics indicate that the developer have not made sufficient effort in reusing the data-members and member-functions of the classes within the program. The complexity is directly proportional to values of CMCM, CICM and CALM, but highest value of CCOM means least complexity. These are evaluated on Weyuker’s properties and Briand et al properties for complexity. All Weyuker’s properties (1 to7) except fifth are satisfied by four newly introduced metrics. The fifth property is also satisfied by one metric i.e. CMCM, but other three does not satisfy it. CMCM, CICM, CALM and CCOM satisfy the property 1 to property 5 of Briand et al. framework except property 4.

These metrics are empirically evaluated to predict the induced faults at design level in classes in the next chapter 6. The faults predicted by these metrics are further used to rank the classes according to number of faults by considering the class having
highest number of faults at high risk and rank 1, to acquired reliability at design level through fault forecasting technique.

5.5 Conclusion

As the name of the chapter suggests it is devoted to formulate the proposal for new metrics and named with the acronyms. These are as follows; Class Member Complexity Measure (CMCM), Class Inheritance Complexity Measure (CICM), Class Aggregation Level Measure (CALM), Class Cohesion Measure (CCOM). The basic objective is to propose these metrics rather models is to study the software at the design phase of its life cycle. It is to be highlighted that the usual norms and practice is to use metrics at post-code phase. With unavailability of metrics at pre-code phase it necessitates and found fruitful to put forward these above devised metrics.
References


