Analyzing Theoretical Basis and Inconsistencies of Object Oriented Metrics

Anisha Gupta [1], Gunjan Batra[2] & Vijaylaxmi[3]

Assistant Professor (Dept of Computer Science)
Bharati Vidyapeeth College of Engineering and technology
New Delhi, India
(anishagupta2@gmail.com [1], gunjan_batra27@yahoo.co.in [2],
tyagi.vijaylaxmi@gmail.com [3])

ABSTRACT
Metrics help in identifying potential problem areas and finding these problems in the phase they are developed decreases the cost and avoids major ripple effects from these in later development stages. These days, Object Oriented Paradigm is mainly used for all practical purposes, hence accessing Object Oriented Systems is a major research area in Software Engineering. As proved by researchers Procedural metrics are unfit for measuring various OO characteristics. Many researchers have proposed different OO metrics suite. This paper discusses the most commonly used OO metrics suite (CK, MOOD & LI) on the basis of characteristic they measure. An advanced metric for inheritance is also discussed. The strengths and weaknesses of all these are identified. The paper concludes by identifying that none of the metrics suite is full proof and there are flaws in almost all of them. Moreover, there is no single metric that can measure all the aspects of an OO System. Rather some of the suites have been found to be complimentary. Also, the paper stresses on the fact that metrics should not be treated as rules. Rather appropriate metrics identified using GQM approach act as indicators of the progress that a project has made.

General Terms
Design based Metrics,Object Oriented Metrics, Object Oriented Paradigm
Keywords
Object Oriented Metrics, Object Oriented Paradigm, Inheritance, Coupling, Cohesion, Encapsulation.

1. INTRODUCTION

A key element of any engineering process is measurement. Measurements are used to access the quality of engineered product or process used to build it. Work on software metrics started as early as 1970's with the belief of improving software estimation practices. These included metrics for procedural programming. However since 90's several designers started shifting from Procedural Paradigm to Object Oriented Paradigm because it is a faster development process having module based architecture, highly reusable features which increase design quality and so on. This trend created a new challenge especially to the management team as the conventional metrics invented for classical paradigm seemed no longer valid in supporting their project planning and resource allocation.

There are several characteristics of an object oriented design which include inheritance, cohesion, coupling, encapsulation, message passing etc. The traditional complexity metrics cannot measure these characteristics and are thus not suitable for measuring complexity in OO systems. To ensure the quality of

OO systems, many researchers like Chidamber & Kemerer (C&K Metrics), Henry Li (Li Metrics), Abreau Brito (MOOD Metrics) etc proposed metrics for OO characteristics.

The fig 1 shows the various design based metrics which are categorized as Procedural and OO Metrics. The paper focuses on 3 main types of OO metrics i.e. CK, MOOD & Li.

Section 2 discusses the objectives of metrics. Section 3, 4, 5 and 6 discuss the OO metrics for Inheritance,

Coupling, Cohesion and Encapsulation respectively. The flaws of these metrics are also discussed in the respective sections.

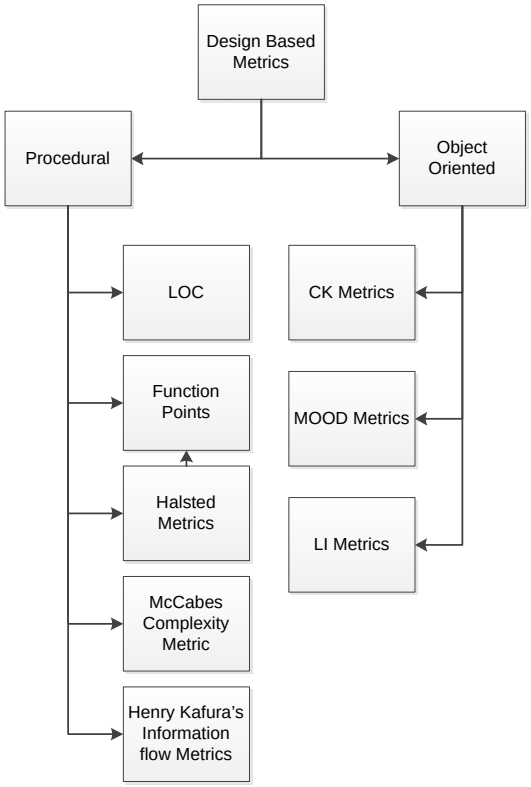

Fig 1 Classification of Metrics

2. 5-FOLD OBJECTIVE OF METRICS

Software metrics which have become an integral part of software development have five major objectives [2].

1. Perception: A metric provide us appropriate information to realize or recognize a software process or product more efficiently.

2. Software Inspection: Metric helps us to examine our product with different input test cases.

3. Planning: software metric also help in various types of planning like budget planning and job scheduling.

4. Optimization: With the help of software metric we can optimize the software by removing dead code, correcting the memory allocation etc.

5. Quality Enrichment: With the help of software metric we are able to improve the quality of software or part of software up to significant amount

3. INHERITANCE

Inheritance is a powerful mechanism in an Object-Oriented (OO) programming. This mechanism supports the class hierarchy design and captures the IS-A relationship between a super class and its subclass. The various metrics proposed for measuring inheritance have been discussed below.

3.1 C& K Metrics

3.1.1 Depth of Inheritance tree (DIT)

It is "the maximum length from the node to the root of the tree".

Theoretical basis it gives a measure of ancestor classes that can potentially affect this class. Higher value of DIT shows a higher potential for reuse but increased complexity.

***Flaws*:** Li found two ambiguous points in this definition [3]

- maximum length from node to root becomes unclear with multiple roots
- Conflicting goals stated in the definition and theoretical basis for the DIT metric. As per the theoretical basis the DIT metric measures the number of ancestor classes of a class, but the definition of DIT stated that it should measure the length of the path in the inheritance tree, which is the distance between two nodes in a graph. This would result in different results in case of multiple inheritances.

3.1.2 Number of children (NOC)

It is the number of immediate sub-classes subordinated to a class in the class hierarchy.

Theoretical basis It is a measure of the subclasses that are going to inherit the methods of the parent class. High NOC value indicates high potential for reuse and likelihood of improper abstraction.

Flaws: The definition of NOC metric counts only the immediate sub-classes instead of all the descendants of the class and thus gives the distorted view of the system [1]. It is not clear as o why only the immediate sub classes have been considered.

3.2 Mood Metrics

3.2.1 Method Inheritance Factor (MIF)

It is the ratio of the sum of number of inherited methods of all classes in system to the sum of number of available methods which may be local or inherited for all classes in system.

3.2.2 Attribute Inheritance Factor (AIF)

It is the ratio of the sum of number of inherited attributes of all classes in system to the sum of number of available attributes which may be local or inherited for all classes in a system.

Theoretical basis High values of MIF & AIF indicate superfluous inheritance thereby leading to greater coupling and reducing the possibility of reuse and low values indicate heavy use of overrides or lack of inheritance [4].

Flaws: Definitions for MIF is inconsistent with the 0-1 scale [5]. The methods available in a leaf class are not inheritable (within the system). Therefore the MIF denominator, by including leaf classes does not represent the maximum possible inheritance, rather it represents a value *greater* than the maximum; hence the value for MIF, for any system, can never be 1 [5].Examples have been given in [1] where MIF values even for the maximum possible inheritance comes out to be 66.6% whereas it should have been 100%.Generally all the attributes are private so AIF does not have much significance.

3.2.3 Extended AIF & MIF

Problem with the AIF/MIF formula is that it considers the count of members a class can reference in a system or a package. But, when we calculate AIF/MIF for each class, members outside the class (except for the members that are inherited) need not considered. It is because denominator in AIF and MIF requires "Total no. of members that a class Ci can reference" and all the members that are public can be referenced by a class, no matter whether it is in its same package or outside the package. Even protected members act as public members in their own package.

Thus, in the extension to MIF & AIF the denominator considers the members of ancestor classes of class Ci and the members defined inside class Ci only. If a class "x" is present in same package as that of class Ci and has public members, but has no interaction with the class Ci, then members of class "x" are not considered. As shown with examples in [4] values for Extended AIF & MIF give more accurate results as compared to simple AIF & MIF. Their results confirm to the assumption that 0.5 can be taken as the threshold value and Values greater than 0.5 indicate unacceptable levels of inheritance.

3.3 Li Metrics

3.3.1 Number of Descendant Classes (NDC)

It is total number of descendent classes (subclasses) of a class. Li proposed NDC as an alternative to NOC and claimed that the NDC metric captures the classes attribute better than NOC by overcoming the limitation of distorted view in NOC [3].

Theoretical basis Higher the values of NDC greater would be the impact of any change made in that class. Thus classes with higher values of NDC would add to complexity of design.

3.3.2 Number of Ancestor Classes (NAC)

It measures the total number of ancestor classes from which a class inherits in the class inheritance hierarchy. It has been proposed by Li as an alternative to DIT to remove the conflict between the definition and Theoretical basis of DIT[3].

Theoretical basis Classes with very high values of NAC are more complex because they get affected by change in any of its superclass. High NAC values indicate higher maintainability efforts because such classes are difficult to understand and modify.

Flaws in NAC & NDC: These metric can be modified to include whether data or methods are being inherited. This would increase the accuracy of complexity measure caused by inheritance relations amongst classes.

4. COUPLING

It refers to the degree of interdependence among the components of a software system. Good software design should obey the principle of low coupling. Strong coupling makes a system more complex because highly interrelated modules are harder to understand, change or correct. A class is said to be coupled to another class if it uses variables or methods of another class.

4.1 C&K METRICS

4.1.1 Response for Class (RFC)

It is number of methods that can be invoked in response to a message sent to an object of a class

Theoretical basis is that the larger the number of methods that can be invoked from a class through messages, the greater the complexity of the class. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes complicated since it requires a greater level of understanding on the part of the tester.

Flaws: C&K recommended only one level of nesting during the collection of data for calculating RFC. This gives incomplete and ambiguous approach as in real programming practice there exists "Deeply nested call-backs" that are not considered here [1].

4.1.2 Coupling Between Objects (CBO)

It is a count of the number of other classes to which a class is coupled. .CBO counts the non-inheritance related calls

Theoretical basis is that larger the number of couples, the higher the sensitivity to changes in other parts of the design and therefore maintenance is more difficult. Strong coupling complicates a system since a class is harder to understand, and modify.

Flaws The unit "class" used in this metric is difficult to justify as it includes a large variety of reference types that occur through method calls, method parameters, return types, thrown exceptions and abstract data types [3]. It can be refined to give a better view of different types of couplings.

4.2 LI METRICS

4.2.1 Coupling Through Message Passing (CTM)

It measures the number of different messages sent out from a class to other classes excluding the messages sent to the objects created as local objects in the local methods of the class [3].

Theoretical basis it relates to the notion of message passing in object-oriented programming. It gives an indication of the number of methods of other classes that are needed to fulfill the class' own functionality.

4.2.2 Coupling Through Abstract Data type (CTA)

It gives a measure of the total number of classes that are used as abstract data types in the data-attribute declaration of a class [3].

Theoretical basis CTA metric relates to the notion of class coupling through the use of abstract data types and thus gives the scope of how many other classes' services a class needs in order to provide its own service to others.

Flaws for CTM & CTA Li proposed CTA and CTM to strengthen and complement the CBO metrics. These two metrics however capture the count of couples only through Abstract Data Type and Message passing. Other couples through references like parameter passing return types etc. have not been accounted for. As a result these two metrics can not be independently used for measuring coupling. Also if they are used with CBO these would result in double counting of references through ADT and message passing. The solution to this lies in modifying the definition of CBO such that it would exclude the couples through Abstract Data Type, Message Passing and inheritance.

4.3 MOOD Metrics

4.3.1 Coupling Factor (COF)

It measures the coupling between classes excluding coupling due to inheritance. It is the ratio between the number of actually coupled pairs of classes in a scope (e.g., package) and the possible number of coupled pairs of classes.

***Theoretical basis*:** It counts all the client supplier relationships in a system e.g. one class's use of another class as an instance variable [6]. It thus gives the degree of inter-relation of a class with another class.

Flaws: There are many cases where the definition of COF becomes ambiguous because the relationship between any two classes in a system is not constrained to just one or the other of these relationship types. Example: Consider two classes Class A & Class B. Class B inherit Class A. Also, each of these contain object of the other class as attribute. Here, Class B's use of a set of A's objects has nothing to do with the fact that class A is its super class. There is probably no 'correct' way of dealing with this situation is terms of the COF metric but a decision needs to be made one way or the other and it needs to be explicit in the metric's definition[1].

5. COHESION

It is a measure of how strongly-related or focused the responsibilities of a single module are. As applied to object oriented programming, if the methods that serve the given class tend to be similar in many aspects, then the class is said to have high cohesion. Cohesion is commonly categorized into seven levels (ranging from low cohesion to high cohesion): coincidental, logical, temporal, procedural, communicational, sequential and functional. Low cohesion modules for example are modules with coincidental cohesion, are indicative of a module that performs two or more basic functions. High cohesion modules have functional cohesion which indicates that modules perform only one basic function.

5.1 C&K Metrics

5.1.1 Lack of Cohesion in Methods (LCOM)

It is a measure for the number of not connected method pairs in a class representing independent parts having no cohesion. It represents the difference between the number of method pairs not having instance variables in common, and the number of method pairs having common instance variables.

Theoretical basis: A high LCOM value would mean low cohesion which indicates the disparateness among methods of a class [7]. So it can be used to identify flaws in the class design. The class should probably be split into two or more classes.

Flaws: In [8, 9] they showed that LCOM metric is inconsistent as it gives the same value of LCOM to both less cohesive and highly cohesive class.

Also, LCOM metric is not able to distinguish a value of cohesion between 0 and any other non-zero value or between a class with high cohesion and a class with medium cohesion. They do not provide precise value of cohesiveness.

6. INFORMATION HIDING AND ENCAPSULATION

In computer science, **information hiding** is the principle of segregation of the *design decisions* in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed. The protection involves providing a stable interface which protects the remainder of the program from the implementation (the details that are most likely to change).

6.1 MOOD Metrics

The metrics, 'Method Hiding Factor' (MHF) and 'Attribute Hiding Factor' (AHF) are considered by the MOOD team to be measures of encapsulation [10, 11]. On many occasions, the term encapsulation is used interchangeably with information hiding. However, many researchers argue that this indicates poor understanding of the concept of encapsulation. Information hiding and encapsulation are not synonymous. Information hiding is only one part of the concept. Encapsulation can be thought of as being an aggregate of two different but related attributes, namely *privacy* and *unity*. Encapsulation can be quantified by measuring both the unity and privacy (i.e. data privacy) of a class. A class that has only private data members will not necessarily be unified. Equally, a fully unified class may contain only visible (public) data.MHF and AHF are measures of the visibility of a class's properties. They are not measures of encapsulation [5].

6.1.1 Method Hiding Factor (MHF)

It explain**s** how methods are encapsulated in a class. The MHF is computed by dividing the number of all visible methods in all classes by the number of all methods in the classes.

Theoretical basis The number of visible methods is a measure of the class functionality. Increasing the overall functionality will reduce MHF. A low MHF indicates insufficiently abstracted implementation. A large proportion of methods are unprotected and the probability of errors is high. A high MHF indicates very little functionality. It may also indicate that the design includes a high proportion of specialized methods that are not available for reuse.

Flaws: Visibility of a member is calculated depending on whether it is public, private or protected. However, we must take into account the effect of a friend construct on the visibility of that class. The friend construct is a C++ mechanism which grants a class or function access to the internal parts of other classes. Existing work in this area has focused on the interplay between inheritance and the friend mechanism, [14], and on the impact the use of the friend construct may have on external attributes of software, [15]. Coarse grained measures of the friend mechanism, [15] focus friend declarations as opposed to actual friend usage. There is a need to make use of fine grained metrics as given in [13].

Moreover, there is no well defined lower & upper limit for the value of MHF.

6.1.2. Attribute Hiding Factor (AHF): is the percentage of invisibilities of attributes in a class. The AHF is computed by dividing the number of visible attributes in a class by the number of all attributes in a class.

Theoretical basis ideally, all attributes should be hidden, and thus AHF=100% is the ideal value.

Flaws: There are no apparent inconsistencies in this metric.

7. RECOMMENDATIONS

Though a number of OO software metrics have been proposed but there is insufficient statistical data to prove the goodness or badness of a metric. For example value of 8 for one metric does not imply that it is twice as complex or twice as “bad” as a value of 4. Thus metrics should not be used as rules but only as guidelines to give an indication of the progress that a project has made. The Software Assurance Technology Center therefore, proposed interpretation guidelines for the usage of metrics based on a comparison of the values [10].

Moreover, with this broad variety of available metrics it often becomes difficult for the software engineer to decide on what to use. GQM[12] proposed by Basili is a useful approach in these situations. It creates a hierarchy of goals; questions that should be answered in order to know if the goals satisfy; and metrics that must be made in order to answer the question. Thus, the GQM approach provides guidelines to find out metrics to be used.

8. CONCLUSION

Detailed study of OO metrics has led us to conclude that MOOD & CK metrics cover every aspect of OOP but many flaws have been observed in them. The paper demonstrates the need for refining these metrics.

MOOD metrics operate at the systems level. Comparing them with those of Chidamber and Kemerer, it is observed that the two sets are complementary, offering different assessments of a system. The CKmetrics are useful to designers and developers of systems, giving them an evaluation of a system at the class level. The MOOD metrics, on the other hand, is of use to project managers, as the metrics operate at a systems level, providing an overall assessment of a system. Li Metrics on the other hand were proposed to remove certain inconsistencies in CK Metrics.

A single measure cannot measure all the aspects of an Object Oriented product. Thus we can use a combination of these metrics which might belong to different researchers to suit the situation appropriately. GQM approach helps in identifying these appropriate metrics.

The results of the metrics must be accessed by the designers keeping in view the trade off between different attributes. For example: deeper the hierarchy of an inheritance tree the better it is for reusability but worse for maintenance. So we select a threshold value that would keep a balance for both the attributes.

9. FUTURE WORK

Most of the research in OO metrics focuses on static metrics which are measures of certain statistics. But many software quality attributes such as execution time, performance and reliability depend on the dynamic behavior of the software. So there is a need to focus on dynamic metrics based on the running environment in future.

10. REFERENCES

[1] Amandeep Kaur, Satwinder Singh, Dr. K.S. Kahlon and Dr Parvinder Singh, "Empirical Analysis of CK & MOOD Metric Suit", December 2010.

[2] Parvinder Singh, "Dynamic metric for polymorphism in object oriented systems", world academy of sciences, engineering and technology, 39, 2008

[3] Dr. Rakesh Kumar and Gurvinder Kaur, " Comparing Complexity in Accordance with Object Oriented Metrics" , IJCA, February,2011

[4] Shreya Gupta, Ratna Sanyal, "Advanced Object Oriented Metrics for Process Measurement", proceedings of sixth International Conference on Software Engineering Advances,2011

[5] Tobias Mayer & Tracy Hall, "Measuring OO Systems: A Critical Analysis of MOOD Metrics"

[6] J.S.V.R.S.SASTRY , Dr. K.V.RAMESH, M.PADMAJA, "Measuring Object Oriented Systems based on Experimental Analysis of Complexity Metrics", May 2011

[7] Ahmed M. Salem, Abrar A.Qureshi, "Analysis Of Inconsistencies in Object Oriented Metrics",February 2011

[8] S. Sunnit and R. S. Salaria, "Critical Analysis of Inconsistencies in CK and MOOD Metrics," 2nd ACIS International Conference on Software Engineering Research, Management and Applications, Los Angeles, 5-7 May 2004

[9] S. Sunnit and R. S. Salaria, "Analysis of CK and Mood Metrics for Inconsistencies and Prediction of Quality Attributes," November 2007.

[10] F.B. eAbreau, M. Goulão, R. Esteves, "Toward the design quality evaluation of object-orientated software systems", Proc. 5th Int. Conf. On Software Quality, 1995

[11] F.B. eAbreau, W. Melo, "Evaluating the impact of object-orientated design on software Quality", March 1996.

[12] Victor R. Basili1. Gianluigi Caldiera1. H. Dieter Rombach2, "THE GOAL QUESTION METRIC APPROACH".

[13] Michael English, Jim Buckley and Tony Cahill, "Fine Grained Software Metrics In Practice", First International Symposium on Empirical Software Engineering and Measurement,2007.

[14] S. Counsell and P. Newson. Use of Friends in C++ Software: An Empirical Investigation. Journal of Systems and Software, 53(1):15–21, 2000.

[15] L. Briand, P. Devanbu, and W. Melo." An Investigation into Coupling Measures for C++". In International Conference on Software Engineering, pages 412–421, 1997