IJCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Volume-3, Issue-5 E-ISSN: 2347-2693

Empirical Study of Object Oriented Software Metrics for Evaluating Software Defects: CKMOOD Suits

Aarti Sharma[1*] and Manoj Wadhwa[2]

[1*,2] *Dept. of Computer Science & Engineering, Echelon Institute of Technology, Faridabad, India*
www.ijcseonline.org

Received: May/04/2015 Revised: May/10//2015 Accepted: May/24/2015 Published: May/30/ 2015

Abstract—Today, Object Oriented Design metrics plays a vital role in software development. Object Oriented Metrics are used for evaluating and predicting the quality and productivity of software. To produce high quality object oriented software, we need a strong design especially during earlier phases of software development. Many object oriented software metrics were proposed for increasing the quality of software design such as fault proneness and the maintainability of classes and methods. In this paper, we provide an empirical evidence for object oriented design complexity metrics with the help of CK suite and MOOD metric suite for determining software defects. In this paper, we find that the effect of these metrics on defects vary across object oriented programming languages like C++, ASP and Java. We apply these metrics on java and C++ programs and find the defects and design high quality software products.

Keywords— Object Oriented Software Metrics, Software Defects, CK Suite, MOOD Suite.

I. Introduction

Software development process and demand of software is increasing day by day. Software metric plays an important role in developing software products or for making them more effective. Software metrics are used to measure software products from analysis phase to testing phase (like analysis, design, coding, testing and maintenance.) and also for improving efficiency and productivity of a software. The Object oriented design metric is important part of software development. The main objective of OODM is to improve the quality and efficiency of software after analyzing the defects. The defects of software can be identified with the help of object oriented design metric at the design phase of software. There are two metrics CK and MOOD metric suits which affect the fault proneness of software. The defects are depends on two factors size and complexity of software. Software defects are some errors or bugs in a program which can be measured in two ways: defect density and failure density [10]. Defect density can be calculated by total number of defects found in every thousand line of program source code. Failure density can be calculated by total number of detected failures per thousand line of code.
In this paper, there is a description about object oriented software metrics. In 1st section, there is an introduction about object oriented metrics CK and MOOD suits, which defines the defects density of software. 2nd section describes the literature review of object oriented design metrics. 3rd section describes the empirical analysis on software defects based on object oriented design. 4th section describes the result after applying these metrics. 5th section describes the conclusion and future scope of the paper and last is acknowledgement and references.

Corresponding Author: *AARTI SHARMA*, sharmaaarti05@gmail.com
Department of Computer Science, EIT, Faridabad, India

II. Literature Review

A. *Metrics for Object Oriented Design*

Object oriented Software metrics primarily focused on understanding software system in terms of objects, classes and their properties. Object and properties of a class describes the structure and behavior of a system. Chidamber and Kemerer proposed the first set of design complexity metrics using Bunge's Ontology as the theoretical basis [] for clear understanding of a system. CK suite work Object Oriented Design Metric by using inheritance , coupling and cohesion between classes and objects via metrics such as Weighted Method per Class (WMC), Coupling between Objects(CBO), Depth of Inheritance(DIT), Number of Children(NOC), Response for a Class(RFC), and Lack of Cohesion(LCOM), which denotes complexity of classes and coupling , cohesion, inheritance of classes. After that Abreu proposed MOOD metrics to measure encapsulation and polymorphism factor via metrics such as Method Hiding Factor(MHF), Attribute Hiding Factor(AHF),Method Inheritance Factor(MIF), Attribute Inheritance Factor(AIF) , Polymorphism factor(PF), Coupling Factor (CF), which denotes the hiding aspect and taking different forms of a class on their usage context.

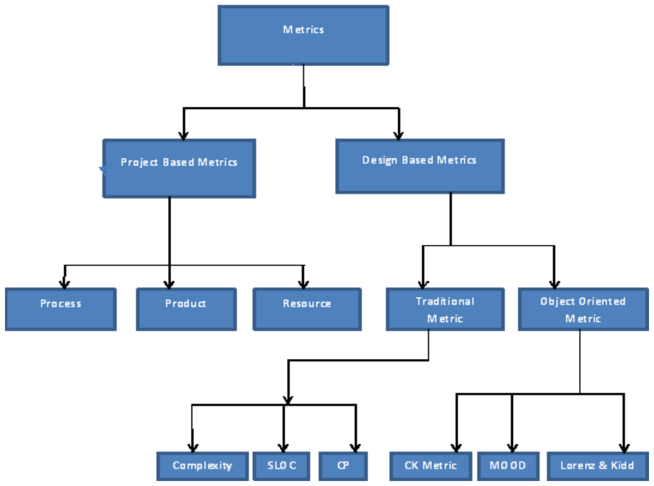

Fig. 1: Metrics Hierarchy

The template is used to format your paper and style the text. All margins, column widths, line spaces, and text fonts are prescribed; please do not alter them. You may note peculiarities. For example, the head margin in this template measures proportionately more than is customary. This measurement and others are deliberate, using specifications that anticipate your paper as one part of the entire proceedings, and not as an independent document. Please do not revise any of the current designations.

1) Chidamber and Kemerer(CK) metrics suite

Chidamber and Kemerer invented six metrics for measuring object oriented programs. These metrics discussed as follow:

(a) Weighted Method per Class (WMC)

WMC is defined as the sum of the complexity of the methods of the class. It is equal to the number of methods when all methods are of the complexity equal to UNITY [3][4].

If a Class C has n methods and c1, c2 …cn be the complexity, then WMC(C) = c1 + c2 +… + cn.

WMC is the predictor of how much Time and Effort is required to develop and to maintain the class. Greater the number of methods more is the impact on the children.

Classes with large WMC are likely to have more faults, limiting the possibility of re-use and making the effort expended one-shot investment. Large WMC increases the density of bugs and decreases the quality of software. A class with a low WMC usually points to greater polymorphism [8].

(b) Depth of Inheritance Tree (DIT)

DIT is defined as the maximum length inheritance path from the class to the root class. Classes with large DIT are likely to inherit, making more complex to predict its behavior. Greater value of DIT leads to greater the potential re-use of inherited methods. Large DIT increases density of bugs and decreases the quality of software. Small values of DIT in most of the system's classes may be an indicator that designers are forsaking re-usability for simplicity of understanding. More is the depth of the inheritance tree greater will be the reusability of class and reduces coding, testing and documentation time. A class situated too deeply in the inheritance tree will be relatively complex to develop, debug and maintain. If DIT is large then testing will be more expensive. As a positive factor, deep trees promote reuse because of method inheritance. Although, inheritance decreases complexity by reducing the number of operations and operators, but this abstraction of objects can make maintenance and design difficult. The depth and breadth of the inheritance hierarchy are used to measure the amount of inheritance.

As depth of the inheritance tree increases, the number of faults also increases. However, it's not necessarily the classes deepest in the class hierarchy that have the most faults. Most fault-prone classes are the ones in the middle of the tree. The root and deepest classes are consulted often, and due to familiarity, they have low fault-proneness compared to classes in the middle [8].

(c) Number of Children (NOC)

NOC is defined as the number of immediate subclasses subordinated to a class in the class hierarchy [].

Small values of NOC may be an indicator of lack of communication between different class designers. A class with a high NOC and a high WMC indicates complexity at the top of the class hierarchy. The class is potentially influencing a large number of descendant classes. This can be a sign of poor design. A redesign may be required.

Greater is the value of NOC greater will be the reusability which in turn enhances productivity. This metric gives an indication of the number of direct descendants (subclasses) for each class. Classes with large number of children are considered to be hard to maintain and thus, difficult to modify and usually require more testing because of the effects on changes on all the children. They are also considered more complex and fault-prone because a class with numerous children may have to provide services in a larger number of contexts and therefore must be more flexible.

A large number of child classes, can indicate that base class may require more testing and there is improper abstraction of the parent class. Not all classes should have the same number of sub-classes. Classes higher up in the hierarchy should have more sub-classes then those lower down. High NOC has been found to indicate fewer faults. This may be due to high reuse, which is desirable [3].

(d) Coupling between Objects (CBO)

Coupling between Object Classes (CBO) for a class is a count of the number of other classes to which it is coupled

[8]. A class that is coupled to other classes is sensitive to changes in those classes and as a result it becomes more difficult to maintain and gets more error prone. As Coupling between Object classes increases, reusability decreases and it becomes harder to modify and test the software system. So there is the need to set some maximum value of coupling level for its reusability.

Two classes are coupled when methods declared in one class use methods or instance variables defined by the other class. Excessive coupling between object classes is detrimental to modular design and prevents reuse. So, high value of CBO is undesirable. Therefore, more independent a class is, the easier it is to reuse it in another application. In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult. A high coupling has also been found to indicate fault proneness. Excessive coupled classes prevent reuse of existing components and they are damaging for a modular, encapsulated software design. To improve the modularity of a software the inter coupling between different classes should be kept to a minimum [3].

(e) Response for a Class (RFC)

RFC is the no. of methods in the response set i.e. the number of methods of the class plus the number of methods called by any of those methods [9] As RFC increases, the effort required for testing also increases because the test sequence grows. It also follows that RFC increases, the overall design complexity of the class increases.

Since RFC specifically includes methods called from outside the class, it is also a measure of the potential communication between the class and other classes. A large RFC has been found to indicate more faults. Classes with a high RFC are more complex and harder to understand. Testing and debugging is complicated as there is more number of test sequences. The Response for a class is high thus increasing the testing effort, test sequence and the overall design complexity of the class. Therefore, reduce the number of operations that maybe execute in response to a message received [8].

(f) Lack of Cohesion of Methods (LCOM)

LCOM is defined as the measurement of the dissimilarity of methods in a class via instanced variables [21].

Each method within a class, C accesses one or more attributes. LCOM is the number of methods that access one or more of the same attributes. If no methods access the same attributes, then LCOM=0. If LCOM is high, methods may be coupled to one another via attributes. This increases the complexity of the class design. As coupling increases, reusability decreases and testing and debugging are also complicated and expensive. Although there are cases in which high value for LCOM is justifiable, it is desirable to keep cohesion high; i.e. keep LCOM low. Higher value of Lack of Cohesion in Methods increases the complexity of class design. Therefore, reduce the lack of cohesion in methods by breaking down the class into two or more separate classes. High cohesion indicates good class subdivision. Lack of Cohesion or low cohesion increases complexity, thereby increasing the likelihood of errors during the development process. It does not promote encapsulation and implies classes should probably be split into two or more subclasses. High LCOM indicates the low quality design of the software [3].

Consider a Class C1 with methods M1, M2. . . Mn. Let {Ii} = set of instance variables used by the method Mi. There are n such sets I1 . . . In.

LCOM = ‖the number of disjoint sets formed by the intersections of the n sets.‖

N = number of different possible pairs of methods (N = n (n−1)/2).

P = |{(mi,mj) : i < j and Ii ∩ Ij = null|

Q = |{(mi,mj) : i < j and Ii ∩ Ij = not null|.

N= P+Q and LCOM = P

B. Metrics for Object Oriented Design (MOOD)

MOOD metrics suits were proposed by Fernando Brito and Rogerio Carpuca in 1994for identification of quality, abstraction and quantitative measurement of object oriented programs. It includes six metrics which measure the presence of OOD (object oriented design) attribute. These metrics values lie between 0 and 1. The MOOD metrics are:

1) Method Hiding Factor (MHF)

It defines the ratio of sum of the invisibilities of all the methods in all classes to the total number of methods defined in a system . the invisibilities of method can be the percentage of all the classes in a system from which this method is not visible. If methods are private then MHF=100%.

MHF is defined as:

$$MHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{M_d(C_i)} (1 - V(M_{mi}))}{\sum_{i=1}^{TC} M_d(C_i)}$$

$$V(M_{mi}) = \frac{\sum_{j=1}^{TC} is\_visible(M_{mi}, C_j)}{TC - 1}$$

$$is\_visible(M_{mi}, C_j) = \begin{cases} 1 \Leftrightarrow J \neq i \wedge C_j \text{ may call } M_{mi} \\ 0 \quad \text{otherwise} \end{cases} \tag{1}$$

M_d (C_i) is total number of methods, TC is the total number of classes in a program, V (M_{mi}) is the visibility of methods M_{mi}.

In object oriented programming, an interface of an object is created to include a group of methods without implementing the behavior f methods. The interface is visible to the whole program. The implementation of the interface is hidden to itself. MHF is 0 when all methods are public. MHF was found to be moderately and negatively correlated with defect density [4]. Defect density would decrease when MHF increase.

2) *Attribute Hiding Factor(AHF)*

AHF is very similar to MHF. It defines the ratio of sum of the invisibilities of all the attributes in all classes to the total number of attributes defined in a system. The invisibilities of attributes can be the percentage of all the classes in a system from which this attribute is not visible. If methods are private then MHF=100%. AHF can be defined as:

$$AHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{A_d(Ci)} (1 - V(A_{mi}))}{\sum_{i=1}^{TC} A_d(C_i)}$$

$$V(A_{mi}) = \frac{\sum_{j=1}^{TC} is\_visible(A_{mi}, C_j)}{TC - 1}$$

$$is\_visible(A_{mi}, C_j) = \begin{cases} 1 \Leftrightarrow J \neq i \wedge C_j \text{ may reference } A_{mi} \\ 0 \quad \text{otherwise} \end{cases} \tag{2}$$

Here $A_d(C_i)$ is the total number of attribute defined , TC is the total number of classes in the program .

3) *Method Inheritance Factor(MIF)*

It is the ratio of sum of inherited methods to the total number of methods in all classes for the system. If no re-usability of methods then MIF=0.It is defined as :

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)} \tag{3}$$

Here $M_a(Ci) = M_d(C_i) + M_i(C_i)$, where $M_d(C_i)$ is the total number of inherited methods defined in a class C_i and $M_i(C_i)$ is total number of inherited method in class C_i. When a class defines more of its own methods, the MIF is getting lower. It is suggested to take the value of MIF between 0.25 and 0.37[1].

4) *Attribute Inheritance Factor (AIF)*

AIF is very similar to MIF. It is the ratio of sum of inherited attributes to the total number of attributes in all classes for the system. If no reusability of attributes then AIF=0. AIF can be defined as:

AIF = inherited attributes/total attributes available in classes

5) *Polymorphism Factor (MIF)*

It is the ratio of actual no. of methods override to the maximum number of methods override in all classes for the system. If all the methods are overridden in all derived classes then PF=100%. In object oriented programming, polymorphism allow message passing with different implementations. PF value should be lower than 0.1.

PF can be defined as:

$$PF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} [M_n(C_i \times DC(C_i))]} \tag{4}$$

6) *Coupling Factor (CF)*

CF is similar to the CBO in CK metric suite. It measure coupling of classes. It is the ratio of actual coupling among classes to maximum number of coupling possible in all the classes. If all the classes are coupled then, CF=100%. Abreuand Carapuca suggested that CF should be below 0.52. CF is highly correlated with software defects due to coupling between objects.

III. EMPIRICAL STUDY

This paper shows an association between object oriented design and fault proneness .The work is done on publically available data set and it shows the result and validations for fault proneness classes and objects. Several metrics like cohesion, coupling, encapsulation, inheritance and size of data are used in this study. The OO Metrics mainly work on independent variables which are used in software development process. Following are the different hypotheticals examples with their CK metric value and MOOD metric value.

A. *Figures and Tables*

Example 1: Object Oriented Design for multiple inheritance

Figure II shows the Object-Oriented design for multiple inheritance and Table I shows the CK metrics values for each class.

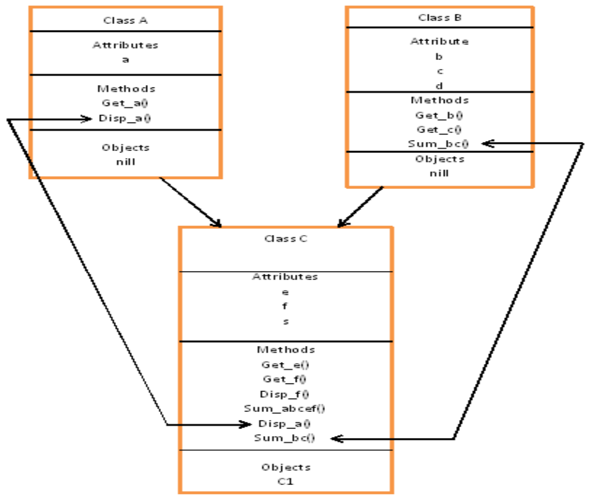

Fig. 2*:Object-Oriented Design For Multiple Inheritance*

Table 1.CK Metrics Values for Multiple Inheritances

	WMC	DIT	NOC	CBO	RFC	LCOM
CLASS A	2	0	1	1	2	0
CLASS B	3	0	1	1	3	1
CLASS C	6	2	0	2	11	2
AVERGE	3.7	0.7	0.7	1.3	5.3	1

Example 2: Object-Oriented design for shapes drawing program

Figure III shows the Object-Oriented design for shapes drawing program and Table II shows the values of CK metrics for each class.

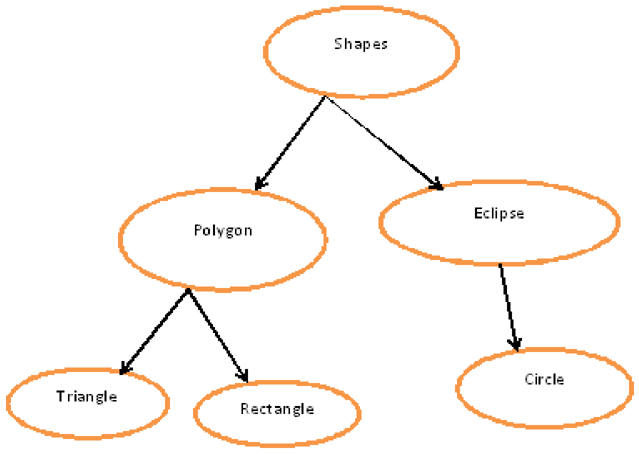

Fig. 3: Object-Oriented Design For Shapes Drawing Program

Table: II
Hierarchy For The Hypothetical Shapes Drawing Program

Classes	WMC	DIT	NOC	CBO	RFC	LCOM
Shapes	2	0	2	1	3	1
Polygon	1	2	1	0	1	0
Eclipse	0	1		0	0	0
Triangle	1	0	2	3	4	0
Rectangle	1	0	2	3	4	0
Circle	1	0	2	0	1	0
Average	2	1	3	2.3	4.3	0.3

IV. Results

To perform the empirical analysis on object oriented design metrics, we design a project model based on CK and MOOD metrics. After applying these metrics on the software design metrics we conclude some values based on fault proneness. Some design metrics have some bugs due to large classes and methods. So this model calculates the value of each metrics and defects in programs and designs. So we can easily increase the performance and quality of software design with this model.

Table: III
Values of the output metric (Defect Index from CKMOOD metric)

METRICS	SOURCE CODE1	SOURCE CODE2	SOURCE CODE3	SOURCE CODE4	SOURCE CODE5	SOURCE CODE6	SOURCE CODE7	SOURCE CODE8	SOURCE CODE9
WMC	20	25	30	35	15	10	40	45	50
DIT	2	3	5	7	2	1	8	10	12
CBO	4	5	7	8	2	1	10	15	20
NOC	2	3	4	6	3	1	7	5	10
LCOM	2	3	5	7	2	1	8	10	12
RFC	20	45	55	65	10	5	70	75	80
MHF	0.305	0.756	0.897	0.867	0	0	0.456	0.898	0.912
AHF	0.375	0.667	0.16	0.94	0.667	0.12	0.97	0.86	1.0
MIF	0.491	1	1	0.4	1	0.2	0.498	0.13	0.26
AIF	0.676	1	1	0.5	1	0	0.6	0.5	0.9
PF	0	0.2	0.5	0.6	0	0	0.8	0.5	0.87
CF	0.78	0.25	0.3	0.3	0.28	0.2	0.5	0.3	0.8
DEFECT OF CK METRIC	1.0630	1.2610	1.3739	1.4405	1.0233	0.9253	1.5643	1.6751	1.8765
DEFECT OF MOOD	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000
HYBRID DEFECTS	1.5630	1.7610	1.8739	1.9405	1.5233	1.4253	2.643	2.1751	2.3765

Here are some figures which shows the relationship between all the object oriented software metrics and defects in software .

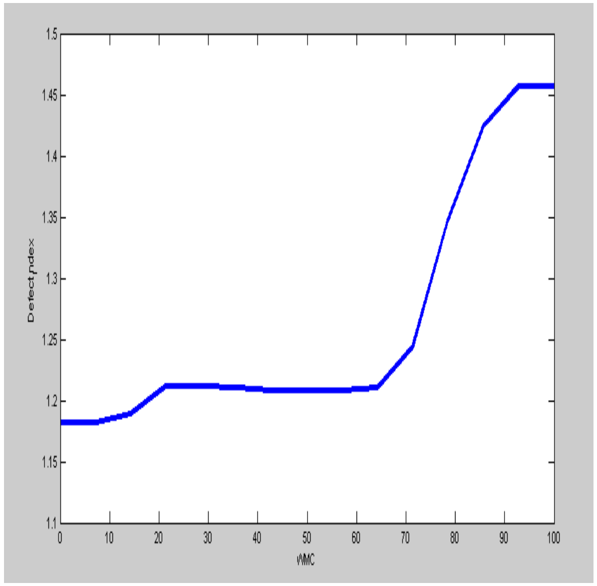

Fig. 4: Graph for Defect Proneness vs. WMC

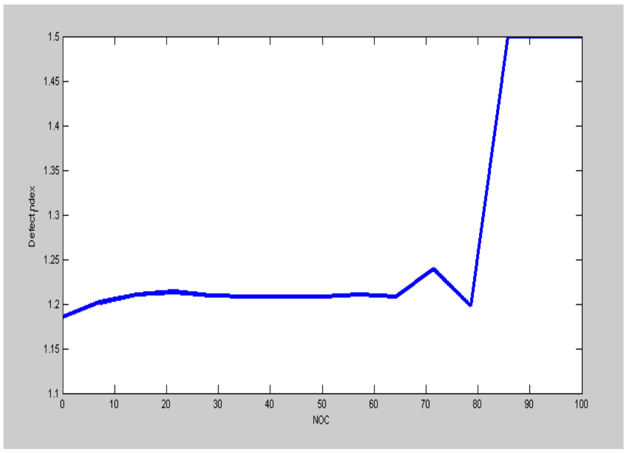

Fig. 5: Graph for Defect Proneness vs. NOC.

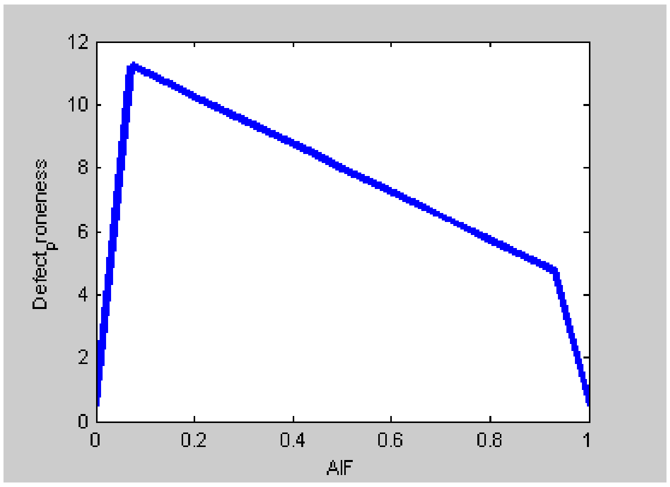

Fig. 8: Graph for Defect Proneness vs. AIF.

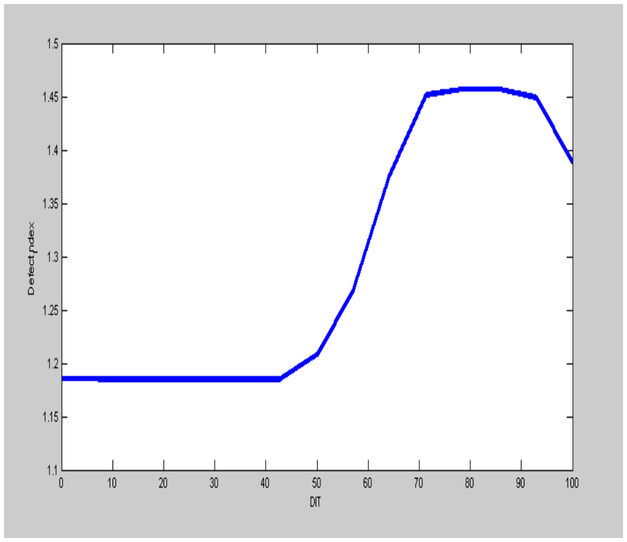

Fig. 6: Graph for Defect Proneness vs. DIT

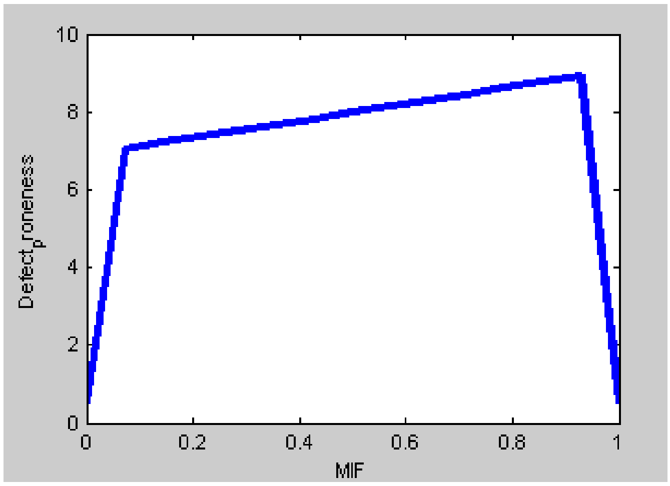

Fig. 9: Graph for Defect Proneness vs. POF

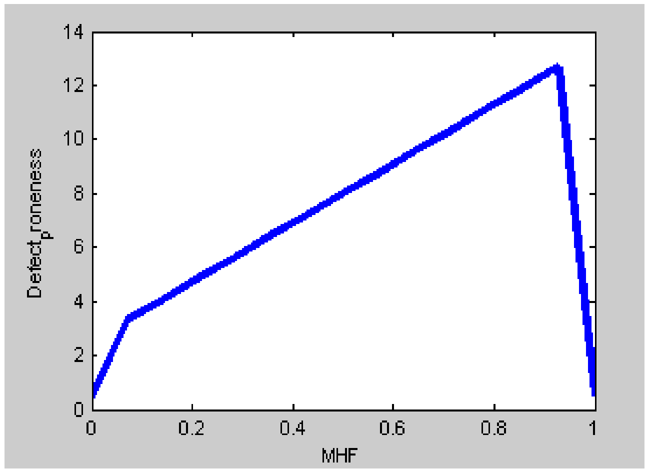

Fig. 7: Graph for Defect Proneness vs. MHF

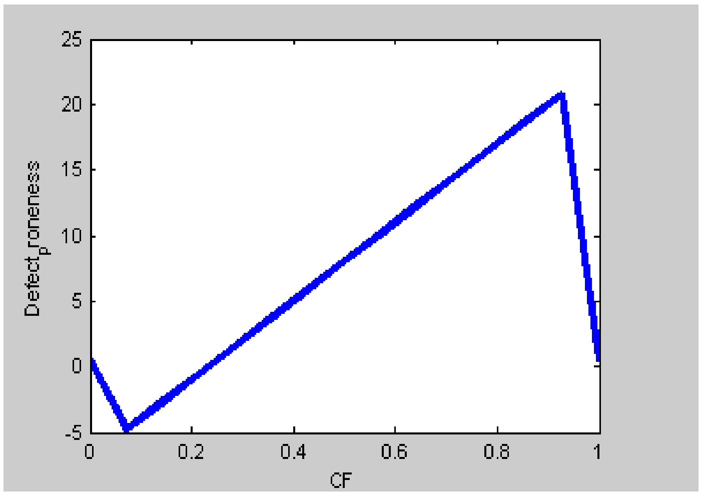

Fig. 10: Graph for Defect Proneness vs. COF.

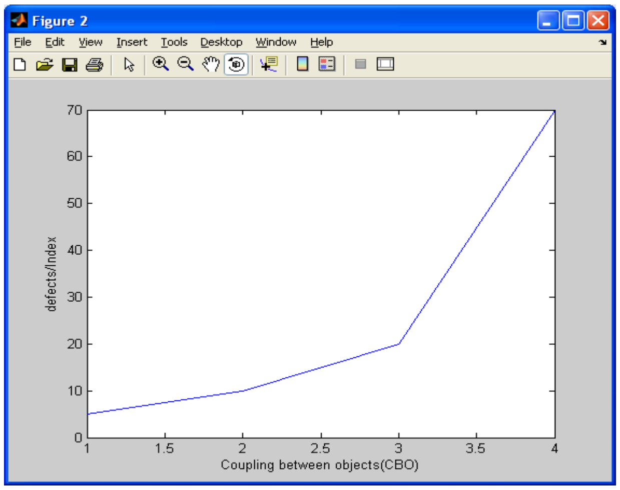

Fig. 11: Graph for Defect Proneness vs. CBO.

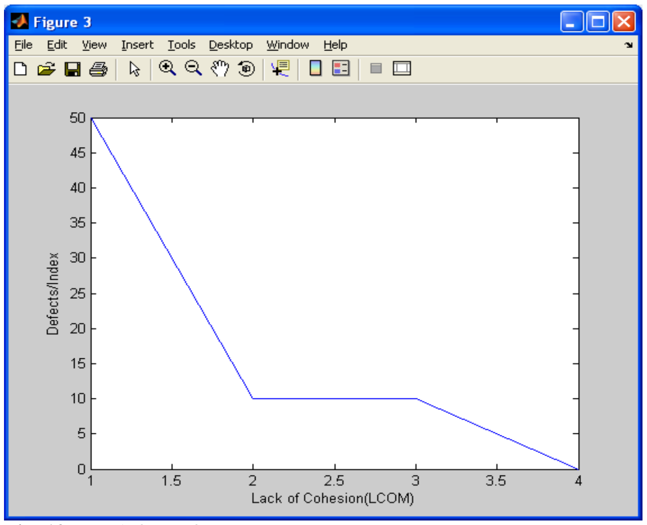

Fig. 12: Graph for Defect Proneness vs. LCOM.

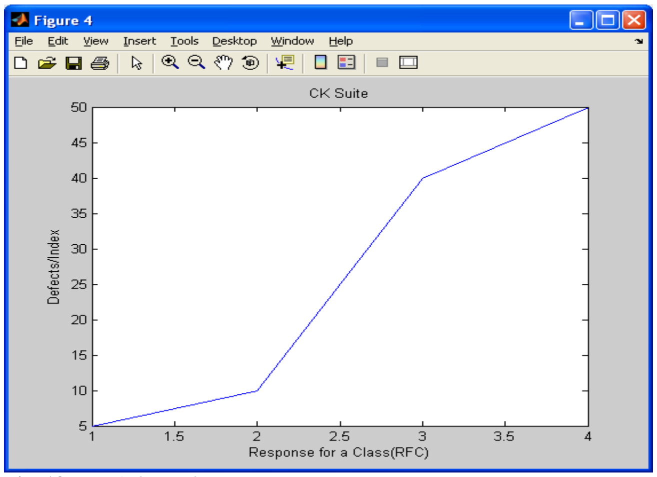

Fig. 13: Graph for Defect Proneness vs. RFC

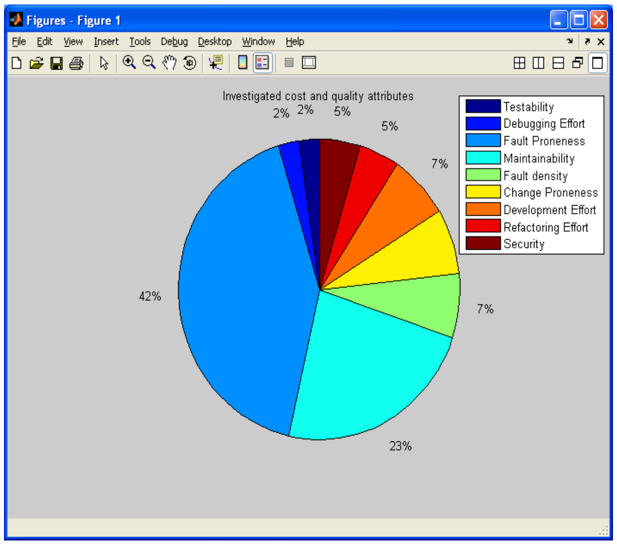

Fig. 14: Pie Chart for Investigated Cost and Quality Attribute.

From the values, it is clear that classes with lesser value of defect index are less prone to faults as compared to classes with higher value of defect index and hence, they need to be reconsidered. In the last figure 14 there is analysis on cost and quality attribute on the basis of faults proneness.

V. CONCLUSION & FUTURE SCOPE

In this paper, we describe 2 metrics CK and MOOD, which are used for measuring defects in any object oriented software design. For better performance of software we need a good design without fault proneness. So, we proposed a model for making software fault free. It analyzed the performance of proposed model using the fuzzy logic approach. The proposed model includes the metrics given by Chidamber and Abreu (1994). The model can be effectively used for predicting the faulty classes in the early phases of SDLC which in result minimize the effort of the software developers. Hence, the model can help in improving the quality and reducing faulty classes in the OOD early. The study can be extended to deal with object oriented design specifications. More combinations of the different available metrics can be integrated depending upon the requirements of the user. We used 6 metrics of CK and 6 metrics of MOOD metric suite, correlation of other metrics can also be examined and they can also be used to estimate the prediction of fault proneness. We used fuzzy logic approach another approaches like neural networks, case based systems can also be used to make the system more effective. We can also find the solution to other inconsistencies to which the solution has not been proposed yet.
Due to the inconsistent findings of some metrics relating to software defect, future studies could systematically validate

these metrics using different projects in different scales. Different programming languages may have different impacts on the use of metrics. Future studies can compare and contrast the same projects written in different languages or for different platforms. Mobile applications could be one of the best candidates since the same application may be prepared using different programming languages and target to different platforms such as iOS, Android, Microsoft Windows Phone, etc.

ACKNOWLEDGMENT

I would like to thank Dr. (Prof.) Manoj Wadhwa , who help me in all the activities and his support make me able to complete my work and also thankful to Echelon Institute of Technology for providing me all the services and facilities during my studies.

REFERENCES

[1] Abreu, F.B. and Carapuca, R. 1994. Candidate metric for OOS within taxonomy framework. *J. Syst. Software*. 26:

[2] Chandra, E. and Linda, P.E. 2010. Assessment of software quality through object oriented metrics. *CIIT Int. J. Software Engg*. 2: 2.

[3] Dubey, S.K. and Rana, A.2010. A comprehensive assessment of object-oriented software systems using metrics approach. *IJCSE*. 2: 2726-2730.

[4] J. Capers, Software Quality: Analysis and Guidelines for success, International Thomson Computer Press, USA, 2000.

[5] Khalsa, S.K. 2009. A Fuzzified approach for the prediction of fault proneness and defect density. Proc. of the World Congress on Engineering. Vol. I. WCE 2009. July 1-3, London, U.K.

[6] Pressman, R.S. 2001. Software engineering-A practitioner's approach. McGraw-Hill international edition. 5th edition.

[7] Shyam R. Chidamber, Chris F. Kemerer, ―A metrics suite for object oriented design‖, IEEE transactions on software engineering, Vol. 20, No. 6, pp. 476-493, June 1984.

[8] Subramanyam, R., Krishnan, M.S., ―Empirical analysis of CK metrics for object -oriented design complexity: Implications for software defects‖, IEEE Transactions on Software Engineering, Vol. 29, No. 4, pp. 297-310, April 2003.

[9] Subramanyam, R. and Krishnan, M.S. 2003. Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Software Engg*. 29(4): 297-310.

[10] Wahyudin D., Ramler R. and Biffle S., *A framework for Defect Prediction in Specific Software Project Contexts.* Proc. of the 3rd IFIP CEE-SET, 2008, 295-308.

[11] Zhou Y. and Leung H., *Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults.* IEEE Trans. on Software Engineering, 32(10), 2006, 771-789

AUTHORS PROFILE

Aarti Sharma received the B.Tech degrees in Information Technology from Lingaya's Institute of Technology in 2012 and pursuing M.Tech in Computer Science & Engineering from Echelon Institute of Technology, Faridabad, India.

Dr.(Prof.) Manoj Wadhwa is working as a Professor and Head Department Of Computer Science and Engineering at Echelon Institute of Technology(EIT), Faridabad, India. He has received M.Tech (Computer Science & Technology) from KU Kurukshetra. He has been published and presented more than 30 Research and Technical paper in International Journals, International Conferences and National Conferences. His main research interests are Software Engineering, Software Metrics, Software Testing, Software Quality and Object Oriented Design. He is the member of IEEE, CSI and ISTE.