IJESRT
INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

Use Of Software Metrics To Measure And Improve The Quality Of The Software Design

Deepak Agrawal[1], Anurag Punde[2], Priyanka Pandey[3], Sonal Dubey[4], Mrinalika Ghosh[5], Vijit Jain[6]
agrawal.deepak22@gmail.com

Abstract

The software development process plays an important role in the field of Information & Telecommunication (ICT). The project managers are emphasis to improve the quality of the software process. To provide the good quality product the developers are prominence on new improved approaches. Many researchers have proposed many approaches, but, the most promising approach is object orientation. With the development of the improved software also demand for the software measurement and also enhance the quality of services (QoS). Such metrics are needed or useful when the organization is adopting the new technology with the new programming standards. The purpose of this study is to have qualitative analysis by elevating design complexity of the software. In this paper, we investigate the problems in software development which includes understand-ability, maintainability, complexity, efficiency, test-ability, reuse-ability, security and encapsulation. The metrics and the subset of metrics chosen are prevalent in practice. The anticipated outcomes acquired are beneficial to be used by software designers for orienting and aligning their design with small scale industry software development practices.

Index Terms— Software metrics, coupling, inheritance metrics, reusability, efficiency, complexity.

Introduction

There are lots of changes happening in today's technology. The OO approach has significance advantages over the traditional approach and structured programming approach. The characteristics of OO approach include data hiding, message passing, data abstraction, encapsulation etc. which differs with the other programming standards. The industries are now adopting OO approach very frequently. The rapid changes in the technology always require some methods that analyse the product developed by those technologies. The quality of any product will depend upon certain parameters such as encapsulation, data hiding, maintainability, efficiency, complexity, testability, usability etc. The programming standards and the efficient approach can develope the effective product. Effective software design requires designer to have experience and deep knowledge of object-oriented approach. Only a prudent use of OO mechanisms can result in reusable and maintainable software. To measure the software attributes of such technology, the use of software metrics which than measure it before the development of the product is required.Software metric is a quantitative as well as qualitative measurement. The software metric measures the efficiency of software quality attributes. The objective of having software metric is obtaining numerous valuable applications by measuring efficiency, testability, complexity, reusability, security and understandability. Software metrics can be categorized into product metrics and process metrics. Product metrics assess tracking risks or the security and discovering potential problem areas or efficiency, understandability or testability. Process metrics works on maintainability of the process of the team or organization. Now the software developers or the software managers need to identify the group of different metrics which measures the same aspect of software. The chosen of the software metrics are very necessary which provide some useful information, otherwise managers are unable to measure quality products and the purpose of metrics can be lost.

Many researchers have suggested many metrics, but, in this paper we are taking only few metrics as a base which evaluates the software quality by analyzing software quality attributes. In the following table we have shown the metrics with their description.

S. No	Name of Metrics	Attributes	Description	Source
1	Attribute Inheritance factor	Inheritance	A system level metrics, defined as the ratio of the sum of inherited attributes in all classes of the system under consideration to the total number of available attributes (locally defined plus inherited) for all classes.	CK
2	Method Inheritance Factor	Inheritance	System level metrics, defined as the ratio of the sum of the inherited methods in all classes of the system under consideration to the total number of available methods (locally defined plus inherited) for all classes.	MOOD
3	Number of Children	Inheritance	The measure of number of subclasses which will inherit the methods of the parent class. High value of NOC indicates more testing and improper abstraction which results in misuse of data.	CK
4	Depth of Inheritance	Inheritance	Measure of the ancestor classes that can potentially affect this class. Deeper a particular class in the hierarchy, greater the potential reuse of inherited methods and greater is the design complexity.	CK
5	Attribute Hiding Factor	Information Hiding	Measure of encapsulation. It is ratio of sum of invisibilities of all attributes defined in all classes to the total number of attributes defined in the system under consideration.	MOOD

6	Method Hiding Factor	Information Hiding	Measure of encapsulation. It is the ratio of sum of invisibilities of all methods defined in all classes to total number of methods defined in system under consideration. **Note**: inherited methods not considered.	MOOD
7	Coupling Factor	Coupling	Defined as ratio of maximum possible number of couplings in the system to actual number of couplings not attributable to inheritance.	MOOD
8	Coupling Between The Object	Coupling	Two classes are coupled when methods declared in one class use methods or instance variables defined by the other classes. To improve modularity and promote encapsulation, inter-object class couples should be kept minimum.	CK
9	Weighted Method per class	Class	Measures the complexity, predicts how much time and effort is required to develop and maintain the class .Higher value of WMC leads to a bigger value of complexity and decreases quality.	CK

The flow of this paper as follows: Section 2 contains the literature survey of different researchers, section 3 contains the analysis of the small industries project with their result section 4 contains the conclusion & future enhancement and section 5 contains the references.

SURVEY

Chidambaram and Kemmerer (CK) in 1991, proposed first version metrics known as [Chida91] metrics and in [Chida94], presented the definitions after some improvements. The metrics were defined to measure design complexity with their impact on external quality attributes such as maintainability, reusability, etc. CK'94 applied the metrics on real world projects and found that designers can keep inheritance hierarchy superficial by neglecting reusability which results in ease of understanding. The metrics also helped in detecting design flaws and gives testing resources.

Briand et al.'s metrics was proposed in 1997 as [Brian97]. The measurement of coupling between classes was the objective of this metrics. The metrics was when applied on real systems studies concluded that, if one intends to build quality models of OO designs, coupling was very likely be an important structural dimension to consider, which tend to be associated with fault-proneness.

MOOD metrics was originally proposed as [Bistro94] then was improved in [Bistro96a], and then was extended to MOOD2 metrics in 1998.The metrics defined were used for OO design mechanism which includes inheritance(MIF and AIF) , information hiding (MHF and AHF) and polymorphism(PF) metrics which result in consequent relation between software quality and development productivity.

Bansiya et al.'s metrics had its first version in 1999 and then after some up-gradations in 2002. The metrics were defined to assess design properties like encapsulation, coupling, cohesion, composition and inheritance .The software tool QMOOD++, allows the design assessment to be carried automatically by giving the parameters of interest for particular evaluation. The tool uses C++ as the target language.

Analysis of Small Industries Project

A.1 Method

A method is an operation upon an object and is defined in the class declaration.

• Metric 1: Weighted Methods per Class (WMC)

The WMC is a count of the methods implemented within a class or the sum of the complexities of the methods (method complexity is measured by Cyclomatic complexity). The second measurement is difficult to implement since not all methods are accessible within the class hierarchy due to inheritance. The number of methods and the complexity of the methods involved is a predictor of how much time and effort is required to develop and maintain the class. The larger the number of methods in a class, the greater the potential impact on children since children inherit all of the methods defined in a class. Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse. This metric measures Understandability, Maintainability, and Reusability.

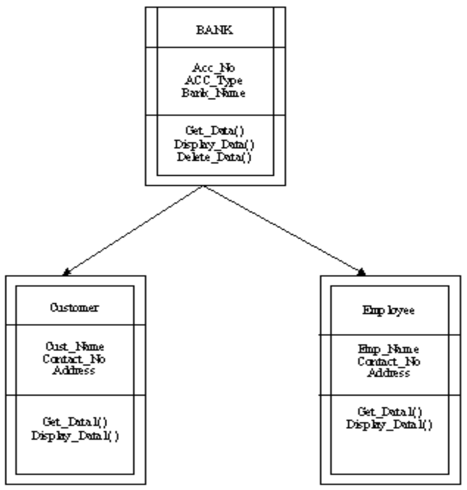

Figure-1: Class Diagram of BANK

To calculate the complexity of a class, the specific complexity metric that is chosen (e.g., cyclomatic complexity) should be normalized so that nominal complexity for a method takes on value 1.0. Consider a class K1, with methods M1… Mn that are defined in the class. Let C1 ….Cn be the complexity of the methods [Chidamber94].

$$\mathbf{WMC} = \sum_{i=1}^{n} C_i$$

If all method complexities are considered to be unity, then WMC = *n*, the number of methods in the class. **In Figure 1**, WMC for BANK is 3 (considering each method complexity to be unity).

1) A.2 Coupling

It is the degree to which components depend on one another. Classes (objects) are coupled three ways:

1. When a message is passed between objects, the objects are said to be coupled.

2. Classes are coupled when methods declared in one class use methods or attributes of the other classes.
3. Inheritance introduces significant tight coupling between super classes and their subclasses.
Since good object-oriented design requires a balance between coupling and inheritance, coupling measures focus on non-inheritance coupling. The next object-oriented metric measures coupling strength.

Metric 2: Coupling Between Object Classes (CBO)
CBO is a count of the number of other classes to which a class is coupled. It is measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends. Excessive coupling is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is reuse in another application. The 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 module is harder to understand, change or correct by itself if it is interrelated with other modules. Complexity can be reduced by designing systems with the weakest possible coupling between modules. This improves modularity and promotes encapsulation. CBO evaluates Efficiency and Reusability.

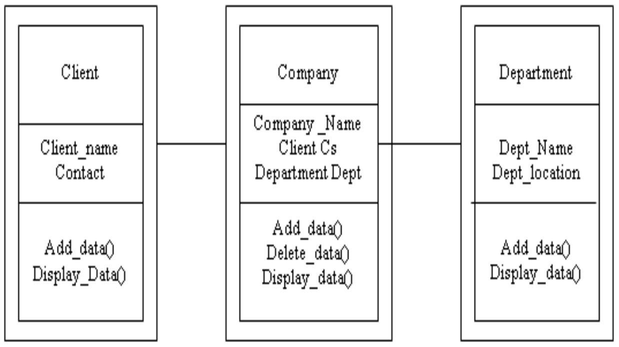

Figure-2: Class Diagram of Company

In Figure 2, Company class contains declarations of instances of the classes Client and Department. The Company class delegates its Client and Department issues to instances of the Client and Department classes. The value of metric CBO class Company is 2 and for class Client and Department is zero.

Metric 3: Coupling Factor
Coupling can be due to message passing (dynamic coupling) or due to semantic
Association links (static coupling) among class instances. It has been known that it is desirable that classes communicate with as few other classes and even when they communicate, they exchange as little information as possible.
Couplings due to the use of the inheritance are not included in CF, because a class is heavily coupled to its ancestors via inheritance. If no classes are coupled, CF = 0 % . If all classes are coupled with all other classes, CF = 100 %.

B Inheritance
Another design abstraction in object-oriented systems is the use of inheritance. Inheritance is a type of relationship among classes that enables programmers to reuse previously defined objects including variables and operators. Inheritance decreases complexity by reducing the number of operations and operators, but this abstraction of objects can make maintenance and design difficult. The two metrics used to measure the amount of inheritance are the depth and breadth of the inheritance hierarchy.

Metric 4: Depth of Inheritance Tree (DIT)
The depth of a class within the inheritance hierarchy is the maximum length from the class node to the root of the tree and is measured by the number of ancestor classes. The deeper a class is within the hierarchy, the greater the number methods it is likely to inherit making it more complex to predict its behavior. Deeper trees constitute greater design complexity, since more methods and classes are involved, but the greater the potential for reuse of

inherited methods. A support metric for DIT is the number of methods inherited (NMI). This metric primarily evaluates Efficiency and Reuse but also relates to Understandability and Testability.

In Figure 3, DIT for Total Emp class is 2 as it has 2 Ancestor classes Domestic/International and Company.
DIT for Domestic and International class is 1 as it has one ancestor class Company.

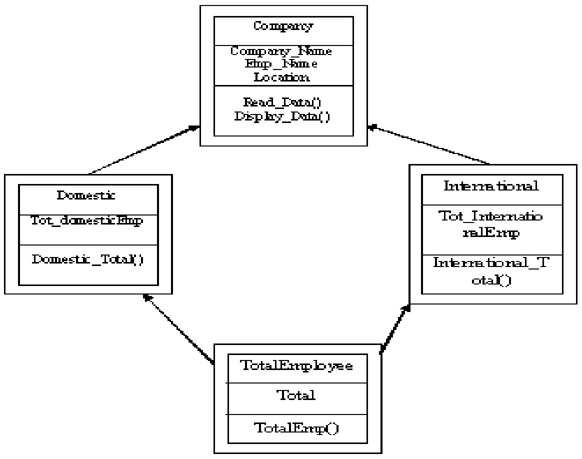

Figure-3: Class Diagram of Company

Metric 5: Number of Children (NOC)
The number of children is the number of immediate subclasses subordinate to a class in the hierarchy. It is an indicator of the potential influence a class can have on the design and on the system. The greater the number of children, the greater the likelihood of improper abstraction of the parent and may be a case of misuse of sub classing. But the greater the number of children, the greater the reusability since inheritance is a form of reuse. If a class has a large number of children, it may require more testing of the methods of that class, thus increase the testing time. NOC, therefore, primarily evaluates Efficiency, Reusability, and Testability.
In figure-2, NOC for Class Company is 2.

Metric 6: Method Inheritance Factor
Formula for calculation is written below**:**

$$MIF = \sum_{i=0}^{TC} \frac{Mi(Ci)}{Ma(Ci)}$$

Where, *Ma(Ci)* = *Mi(Ci)* + *Md(Ci)*
TC= total number of classes
Md(Ci) = the number of methods declared in a class
Mi(Ci) = the number of methods inherited in a class.
The MIF value is calculated from the project below by the above formula. The value of the number of methods inherited in Student is 0. Since, this is the base class. The classes : InternalExam and ExternalExam are inherited by the base class Student in which there are two private attributes and two protected attributes, and two operations as read() and display() . Both the operations are inherited by the derived classes. Hence, the number of methods inherited by the derived class InternalExam and ExternalExam is 2.Similarly, in the class Result; two classes are inherited i.e. the InternalExam and ExternalExam, so the inherited methods for this class is 2. Therefore, the numerator value of the analyzed project comes out to be 6. The denominator value can be easily calculated by summation of the numerator value along with the declared methods in each class.

Metric 7: Attribute Inheritance Factor
It is defined as follows:

$$AIF = \sum_{i=0}^{TC} \frac{Ad\ (Ci)}{Aa(Ci)}$$

where, $A_a(C_i)$ = *Ad (Ci)*+ *Ai (Ci)*

TC= total number of classes
Ad (Ci) = number of attribute declared in a class
Ai (Ci) = number of attribute inherited in a class
AIF is 0 % for class which lacks inheritance. As the MIF is calculated by calculating the number of methods declared and inherited, in the same manner

AIF is calculated by calculating the attributes which are inherited and declared. In our project shown in Figure 4, gives the value of AIF as 6/13.

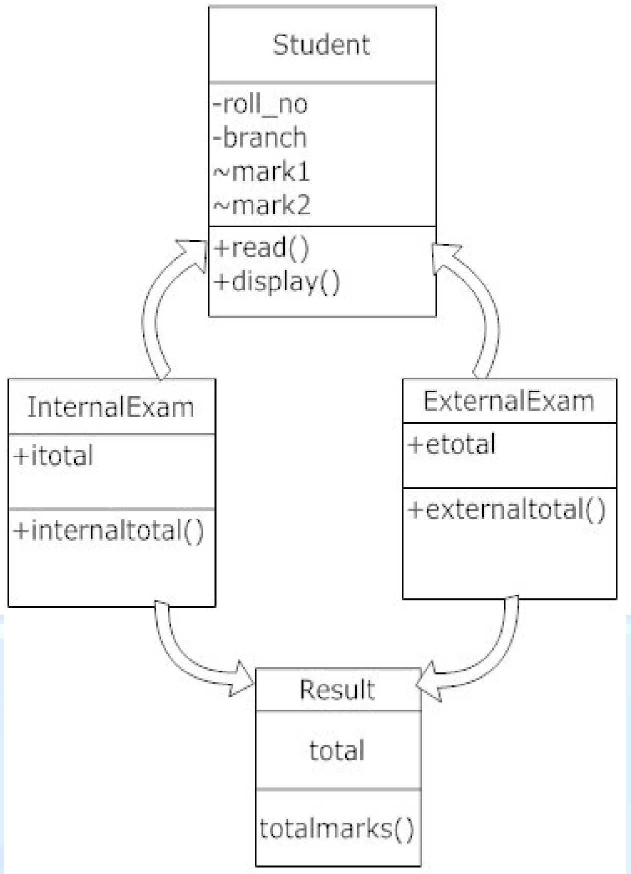

Figure 4 : Class diagram for student

Metric 8: Method Hiding Factor (MHF)
It is a measure of encapsulation defined as:
Where *Md(Ci)* is the number of methods declared in a class, and

$$MHF = \frac{\sum_{i=0}^{TC}\{\sum_{j=0}^{Md(Ci)} \frac{(1-V(Mmi))}{Md(Ci)}\}}{\sum_{i=1}^{TC} A_{d}(C_{i})}$$

Where *Md(Ci)* is the number of methods declared in a class, and

$$V(M_{mi}) = \sum_{j=0}^{TC} \frac{is\ visible(\mathbf{Mmi,Cj})}{TC-1}$$

From the figure 5, we have calculated the method hiding factor. But, there is no private visibilities of the methods are inside the class. So, the values of V (M_{mi}) of all the classes are 0. Hence, the MHF value is 3/8.

Metric 9: Attribute Hiding Factor
The value of AHF is calculated by the formula written as:

$$AHF = \frac{\sum_{i=0}^{TC}\{\sum_{a=1}^{Ad(Ci)} \frac{(1-V(Aai))}{Ad(Ci)}\}}{\sum_{i=1}^{TC} A_{d}(C_{i})}$$

Where *A (C) d i* is the number of methods declared in a class, and

$$V(A_{ai}) = \sum_{j=0}^{TC} \frac{is\ visible(\mathbf{Aai,Cj})}{TC-1}$$

From the figure 5 we have calculated the AHF, which is equal to 1/3.

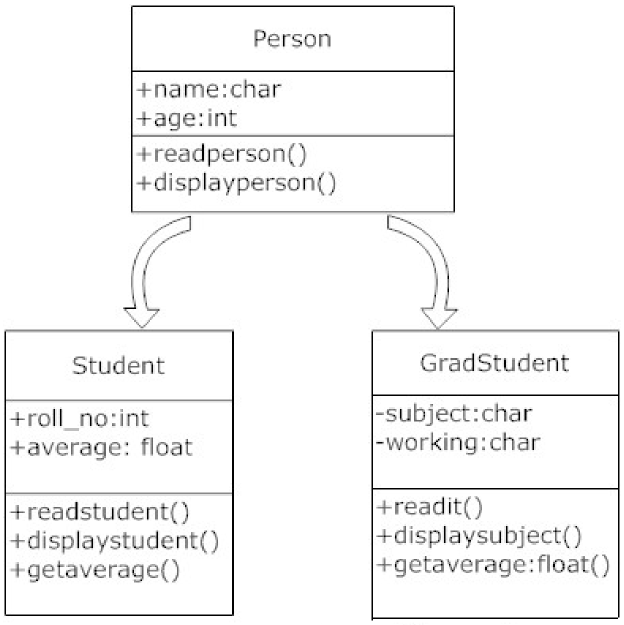

Figure 5: Class diagram for person

Conclusion & Future Improvement

The main concern to present this paper is to improve the quality of the software by providing systematic approach that can expand the efficiency of the software. The industries need to identify the overall approach from analysis to post implementations. If industries do not properly use the programming standards, proper use of data types, proper algorithms than those applications will become cumbersome and the required more maintenance. In future if we want to apply software re-engineering than we have so many problems arises in between this. So, here, we have suggested certain group of metrics which evaluate the quality of the software on the basis of software quality parameters. The product developed with these low standards effect the quality, efficiency of the product. Increase the complexity and ask for more maintenance and testing. The use of such metrics gives appropriate reimbursement in the development of software.

References

1. Fenton NE and Ohlsson N, Quantitative Analysis of Faults and Failures in a Complex Software System, IEEE Transactions on Software Engineering, to appear, 2000

2. S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. IEEE Trans. Software Eng., 20(6):476-493, 1994.

3. Jørgensen, M . A Review of Studies on Expert Estimation of Software Development Effort. 2002.

4. Chidamber S. and Kemerer C.: “Towards a Metrics Suite for Object Oriented Design”, Conference on Object-Oriented Programming: Systems, Languages and Applications (OOSPLA 91), Published in SIGPLAN Notices, vol. 26, no. 11, pp. 197-211, 1991.

5. Chidamber S. and Kemerer C.: “A Metrics Suite for Object Oriented Design”, IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476-493, 1994.

6. Warmer J. and Kleppe A.: The Object Constraint Language: Precise Modeling with UML, Addison Wesley Publishing Company, 1999.

7. Briand L., Morasca S. and Basili V.: “Property-Based Software Engineering Measurement”, IEEE Transactions on Software Engineering, vol. 22, no. 6, pp. 68-86, 1996.

8. Briand L., Devanbu W. and Melo W.: “An investigation into couplingmeasures for C++”, 19th International Conference on Software Engineering (ICSE 97), Boston, USA, pp. 412-421, 1997.

9. Brito e Abreu F. and Carapuça R.: “Object-Oriented Software Engineering: Measuring and

controlling the development process", 4th International Conference on Software Quality, Mc Lean, VA, USA, 1994

10. Bansiya J. and Davis C.: "A Hierarchical Model for Object-Oriented Design Quality Assessment", IEEE Transactions on Software Engineering, vol. 28, no. 1, pp. 4-17, 2002.