Analysis of Object Oriented Design Quality Metrics

Raman Maini[1], Suruchi Mehra[2]
[1]Associate Professor, [2]Research Scholar,
University College of Engineering, Punjabi University,
Patiala - 147002, Punjab, India
[1]research_raman@yahoo.com, [2]suruchipresent@gmail.com

ABSTRACT : *Metrics (quantitative estimates of product and project properties) can, if defined from sound engineering principles, be a precious tool for both project management and software development The purpose of this paper is to evaluate if software metrics can be used to determine the object oriented design quality of a software system. Encapsulation, inheritance, polymorphism, reusability, Data hiding and message-passing are the major attribute of an Object Oriented system. In order to evaluate the quality of an Object oriented system the above said attributes can act as indicators. The metrics are the well known quantifiable approach to express any attribute. Here in this paper we will go though different type of metrics.*

Keywords: *Object Oriented, Software metrics, Methods, Attributes, cohesion, coupling, Inheritance.*

1. Introduction

The use of object oriented software development techniques introduces new elements to software complexity both in software development process and in the final product. The backbone of any software system is its design. Object-oriented analysis and design are popular concepts in today's software development environment. They are often heralded as the silver bullet for solving software problems [1]. The concepts of software metrics are well established, and many metrics relating to product quality have been developed and used. The metrics were selected on the basis of their

ability to predict different aspects of object-oriented design (e.g. the lines of code metric predict a modules size).Programmers has measured their code from the day they started to write it. The size of a program can easily be described as the number of lines of code it consists of. But when it comes to measuring the object-oriented design more complex measures are required. The two first object-oriented languages that introduced the key concepts of object-oriented programming were SIMULA 1 (1962) and the Simula 67 (1967) Two of the pioneers in developing metrics for measuring an object-oriented design were Shyam R. Chidamber and Chris F. Kemerer . In 1991 they proposed a metric suite of six different measurements that together could be used as a design predictor for any object-oriented language. Their original suite has been a subject of discussion for many years and the authors themselves and other researchers has continued to improve or add to the "CK" metric suite. Other language dependent metrics (in this report the Java language is the only language considered) have been developed over the past few years e.g. in ; they are products of different programming principles that describes how to write well-designed code

A standardized metric set for OO does not yet exist for a metrics definition standard [4]. Therefore, it is necessary to define metrics and to analyze them. Once a set of metrics for any type of measurement is proposed, it is necessary to systematically validate them Validating a metric means providing convincing proof that:

A standardized metric [1]set for OO does not yet exist for a metrics definition standard. Therefore, it is necessary to define metrics and to analyze them. Once a set of metrics for any type of measurement is proposed, it is necessary to systematically validate them .Validating a metric means providing convincing proof that:

There are two types of relevant validation for this purposes theoretical or internal validation and empirical or external validation [3][4]

- Theoretical validation
- Empirical validation

Theoretical validation maps to point first point in the list above, and involves clarifying the properties of the attribute to be measured, and analytically proving that the metric satisfies those properties. Such attributes are termed as internal attributes. Theoretical Validation requires consensus among the research community regarding the properties of attributes and reaching such a consensus could potentially take many years. Empirical validation entails demonstrating points second and third above. Empirical validation requires correlating the metric to the external attribute by comparing the values of the metric with the values of the external attribute. [2][5]For example, a metric that measures the number of downloads for a KB may be related to external attributes of the metrics such as awareness within the organization.

2. Metrics, measures and metric theories:

One shouldn't confuse metrics with measures. A metric is a quantitative property of software products (product metrics) or processes (process metrics) whose values are numbers — either integer or real in our current framework). A measure is the value of a metric for a certain product or process.

For example, we can evaluate the metric "number of classes in the system", called just Classes, by counting the classes in our system. This yields a measure. In software we may distinguish between product metrics, which measure properties of the elements being turned out[7] (code, designs, documentation, bug reports…) and process metrics, which measure properties of the process whereby they are being turned out (salaries, expenses, time spent, delays…). To add product metrics requires interfacing with project management tool; this is a desirable development for the future.

Any metric should be relevant related to some interesting property of the processes or products being measured: cost, estimated number of bugs, ease of maintenance [6][8] A metric theory is a set of metric definitions accompanied with a set of convincing arguments to show that the metrics are relevant. Neither EiffelStudio nor this article provides a metric theory; our purpose is simply to provide the basic tools that enable the development and application of good metric theories.

2.1 Metric Types:

There are 3 metric types:

1. **Direct Measurement**
   – Eg length of source code or duration of testing process.
   (These measurement activities involve no other attributes or entities)
2. **Indirect/Derived Measurement**
   – Eg Module defect density = Number of defects/ Module Size.
   (These are combinations of other measurements)
3. **Predictive Measurement**

   – Eg predict effort required to develop software from the measure of its functionality – function point count[9]. (These measurements are systems consisting of mathematical models together with a set of prediction procedures for determining unknown parameters and interpreting results)[14]

2.2 Classification of Scales:

1. Nominal Scale

– This is the most primitive form of measurement. It is a scale that consists only of different classes that have no ordering

– Any distinct numbering or symbolic representation of the classes have no magnitude associated with them.

– Eg IEEE 802.1, 802.2, 802.3,…802.11

2. Ordinal Scale

– The classes are ordered with respect to the attribute. There is no quantitative comparison[10].

– Eg programmer skill (low, medium, high)

Santhan Perampalam

3. Interval Scale

– This scale captures information about the size of the intervals that separate classes. Thus we can understand the magnitude of the jump from one class to another.

– Addition and subtraction operations are permissible

– Eg programmer capability between: 60th and 80^{th} percentile of population

4. Ratio Scale

– A measurement mapping that the preserves ordering, the size of intervals between entities, and ratios between entities

– Eg project A took twice as long as project B

Santhan Perampalam5[2]

5. Absolute Scale

– States that there is only one way in which the measurement can be made. There is only one possible measurement mapping – the actual count.

– E.g. number of failures observed during integration testing can be measured only by counting the number of failures observed.

– For better understanding consider:

• Length of source code (of which LOC is a ratio scalar)

• Engineer's age (of which years is a ratio scalar)

• Number of lines of code (of which LOC is an absolute scalar) [4]

3. Metrics for Object Oriented Design:

3.1 Traditional Metrics:

In an object-oriented system, traditional metrics are generally applied to the methods that comprise the operations of a class. Methods reflect how a problem is broken into segments Traditional metrics have been applied for the measurement of software complexity of structured systems since 1976 [10]. The following discussion shows three popular traditional metrics.

McCabe Cyclomatic Complexity (CC)

Complexity metrics can be used to calculate essential information about constancy and maintainability of software system from source code. It also provides advice during the software project to help control the design. In the testing and maintain phase, complexity metrics provide detail information about software module to identify the areas of possible instability. Cyclomatic complexity (McCabe) can be used to evaluate the complexity of a method [3][6]. This metric measures the complexity of a control flow graph8 of a method or procedure. The idea is to draw the sequence a program may take as a graph with all possible paths. The complexity is calculated as "connections - nodes + 2" and will give a number denoting how complex the method is. See the following figure. Since complexity will increase the possibility of errors, a too high9 McCabe number should be avoided

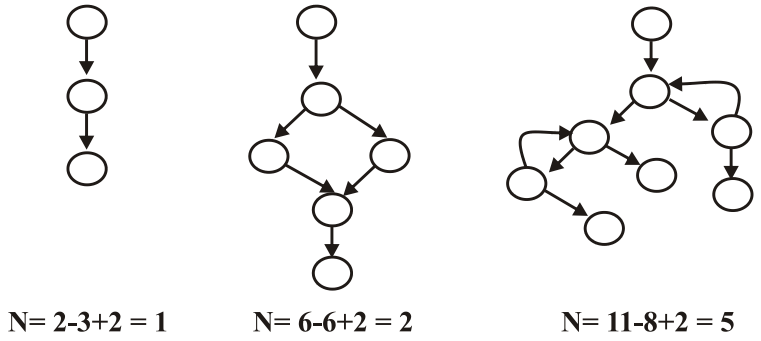

Figure 1

As described in Laing et al., McCabe et al. mention cyclomatic complexity is a measure of a module control flow complexity based on graph theory. Cyclomatic complexity cannot be used to measure the complexity of a class because of inheritance, but the cyclomatic complexity of individual methods can be combined with other measures to evaluate the complexity of the class . A high10 cyclomatic

complexity indicates that the code may be of low quality and difficult to test and maintain.

Source Lines of Code (SLOC)

SLOC is used to estimate the total effort that will be needed to develop a program, as well as to calculate approximate productivity. The SLOC metric measures the number of physical lines of active code, that is, no blank or commented lines code. Logical SLOC measures the number of statements, but their specific definitions are fixed to specific language for example, in C programming language logical SLOC measure the terminating semicolon. Since functionality is not as much interconnected with SLOC, expert developers may be capable to develop the same functionality with less code. So one program with less SLOC may show more functionalities than another similar program. Programs with larger SLOC values usually take more time to develop. Therefore, SLOC can be very effective in estimating effort. Thresholds for evaluating the SLOC measures vary depending on the coding language used and the complexity of the method

Comment Percentage (CP)

The CP metric is defined as the number of commented lines of code divided by the number of non-blank lines of code [11]. The comment percentage is calculated by the total number of comments divided by the total lines of code less the number of blank lines. The SATC11 has found a comment percentage of about 30% is most effective

3.2 C.K. Metrics:

Chidamber and Kemerer define the so called CK metric suite This metric suite offers informative insight into whether developers are following object oriented principles in their design . They claim that using several of their metrics collectively helps managers and designers to make better design decision. CK metrics have generated a significant amount of interest and are currently the most well known suite of measurements for OO software. Chidamber and Kemerer proposed six metrics; the following discussion shows their metrics.

Weighted Method per Class (WMC)

WMC measures the complexity of a class. Complexity of a class can for example be calculated by the cyclomatic complexities of its methods. High value of WMC indicates the class is more complex than that of low values. So class with less WMC is better. As WMC is complexity measurement metric, we can get an idea of required effort to maintain a particular class.[16]

Depth of Inheritance Tree (DIT)

DIT metric is the length of the maximum path from the node to the root of the tree. So this metric calculates how far down a class is declared in the inheritance hierarchy. The following figure shows the value of DIT for a simple class hierarchy. This metric also

Measures how many ancestor classes can potentially affect this class. DIT represents the complexity of the behavior of a class, the complexity of design of a class and potential Reuse[17].

Figure2

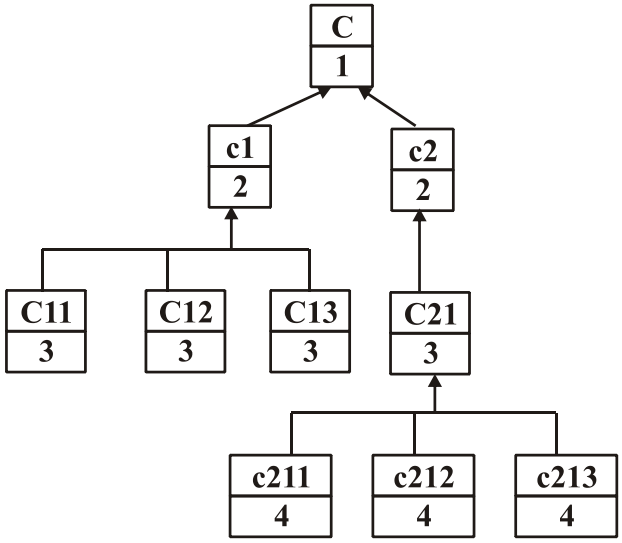

Figure 2: *The value of DIT for the class hierarchy*

If DIT increases, it means that more methods are to be expected to be inherited, which makes it more difficult to calculate a class's behavior. Thus it can be hard to understand a system with many inheritance layers. On the other hand, a large DIT value indicates that many methods might be reused.

Number of children (NOC)

This metric measures how many sub-classes are going to inherit the methods of the parent class. As shown in above figure, class C1 has three children, subclasses C11, C12, and C13. The size of NOC approximately indicates the level of reuse in an application. If NOC grows it means reuse increases. On the other hand, as NOC

increases, the amount of testing will also increase because more children in a class indicate more responsibility. So, NOC represents the effort required to test the class and reuse.

Coupling between objects (CBO)

The idea of this metrics is that an object is coupled to another object if two object act upon each other. A class is coupled with another if the methods of one class use the methods or attributes of the other class. An increase of CBO indicates the reusability of a class will decrease. Thus, the CBO values for each class should be kept as low as possible.CBO metric measure the required effort to test the class. An overview of object oriented design metrics

Response for a Class (RFC)

RFC is the number of methods that can be invoked in response to a message in a class. Pressman States, since RFC increases, the effort required for testing also increases because the test sequence grows. If RFC increases, the overall design complexity of the Class increases and becomes hard to understand. On the other hand lower values indicate greater polymorphism. The value of RFC can be from 0 to 50 for a class12, some cases the higher value can be 100- it depends on project to project [17]

Lack of Cohesion in Methods (LCOM)

This metric uses the notion of degree of similarity of methods. LCOM measures the amount of cohesiveness present, how well a system has been designed and how complex a class is . LCOM is a count of the number of method pairs whose similarity is zero, minus the count of method pairs whose similarity is not zero.

Raymond discussed for example, a class C with 3 methods M1, M2, and M3. Let I1={a, b, c, d, e}, I2= {a, b, e}, and I3= {x, y, z}, where I1 is the set of instance variables used by method M1. So two disjoint set can be found: I1 Ç I2 (= {a, b, e}) and I3. Here, one pair of methods who share at least one instance variable (I1 and I2). So LCOM = 2-1 =1. Riel13 states “Most of the methods defined on a class should be using most of the data members most of the time”. If LCOM is high, methods may be coupled to one another via attributes and then class design will be complex. So, designers should keep cohesion high, that is, keep LCOM low [18].

3.3 MOOD Metrics

The 6 MOOD metrics:

1. Attribute Hiding Factor (AHF)
2. Method Hiding Factor (MHF)
3. Method Inheritance (MIF)
4. Attribute Inheritance Factor (AIF)
5. Polymorphism Factor (PF)
6. Coupling Factor (CP)

Attribute Hiding Factor (AHF)

This is a clearly defined metric with no apparent inconsistencies. Its use is in determining the level of visibility of a class's data. The AHF metric shows the sum of the invisibilities of all attributes in all classes. The invisibility of an attribute is the percentage of the total classes from which this attribute is hidden. MHF and AHF represent the average amount of hiding among all classes in the system. The AHF metric is defined as follows. If the value of AHF is high (100%), it means all attributes are private. AHF with low (0%) value indicates all attributes are public.

Method Hiding Factor (MHF)

The MHF metric states the sum of the invisibilities of all methods in all classes. The invisibility of a method is the percentage of the total class from which the method is hidden. Abreu et al. [12] States, the MHF denominator is the total number of methods defined in the system under consideration. The MHF metric

Is defined as follows

$$\text{MHF} = \sum_{i=1}^{TC} M_k(C_i) \Big/ \sum_{i=1}^{TC} M_d(C_i)$$

Here $M_d(C_i) = M_v(C_i) + M_k(C_i)$

$M_d(C_i)$ = the number of methods defined in class Ci

$M_v(C_i)$ = the number of methods that visible in the class Ci

$M_k(C_i)$ = the number of methods hidden in Ci

Where the summation occurs over i=1 to TC. TC is defined as total number of classes. If the value of MHF is high (100%), it means all methods are private which indicates very little functionality. Thus it is not possible to reuse methods with high MHF. MHF with Low (0%) value indicates all methods are public that means most of the methods are unprotected.

Method Inheritance (MIF)

The MIF metric states the sum of inherited methods in all classes of the system under consideration. The degree to which the class architecture of an object oriented system makes use of inheritance for both methods and attributes MIF is defined as the ratio of the sum of the inherited methods in all classes of the system as follow.

$$\text{MIF} = \sum_{I=1}^{TC} M_i(C_i) \;/\; \sum_{i=1}^{TC} M_i(C_i)$$

Here $M_i(C_i) = M_d(C_i) + M_i(C_i)$

$M_i(C_i)$ = the number of methods defined in class Ci

$M_d(C_i)$ = the number of methods declared in the class Ci

$M_i(C_i)$ = the number of methods inherited in Ci

Where the summation occurs over i=1 to TC. TC is defined as total number of classes. If the value of MIF is low (0%), it means that there is no methods exists in the class as well as the class lacking an inheritance statement [13].

Attribute Inheritance Factor (AIF)

AIF is defined as the ratio of the sum of inherited attributes in all classes of the system. AIF denominator is the total number of available attributes for all classes. It is defined in an analogous manner and provides an indication of the impact of inheritance in the object oriented software [14]. AIF is defined as follows

$$\text{AIF} = \sum_{i=1}^{TC} A_i(C_i) \,/\, \sum_{i=1}^{TC} A_a(C_i)$$

Here $= A_a(C_i) = A_d(C_i) + A_i(C_i)$

$A_a(C_i)$ = the number of available attributes defined in class Ci

A_d (C_i) = the number of attributes that declared in the class Ci

A_i (C_i) = the number of methods inherited in Ci

Where the summation occurs over i=1 to TC. TC is defined as total number of classes. If the value of AIF is low (0%), it means that there is no attribute exists in the class as well as the class lacking an inheritance statement.

3.5. Polymorphism Factor (PF)

The POF represents the actual number of possible different polymorphic situation. It also represents the maximum number of possible distinct polymorphic situation for class Ci. The POF is defined as follows.

$$\text{POF} = \sum_{i=1}^{TC} M_o(C_i) / \sum_{i=1}^{TC} \left[M_n(C_i) * DC(C_i)\right]$$

Here $M_o(C_i) = M_n(C_i) + M_o(C_i)$

$M_n(C_i)$ = the number of new methods defined in class Ci

$M_o(C_i)$ = the number of overriding methods in the class Ci

DC (C_i) = the descendants in Ci

The numerator represents the actual number of possible different polymorphic situation and the denominator represents the maximum number of possible distinct polymorphic situation for class Ci [15]. The value of POF can be varies between 0% and 100%. If a project have 0% POF, it indicates the project uses no polymorphism and 100% POF indicates that all methods are overridden in all derived classes17.

Coupling Factor (CP)

The COF is defined as the ratio of the maximum possible number of couplings in the system to the actual number of coupling is not imputable to inheritance [31b].

The COF is defined as follows.

$$\text{COF} = \sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} is\_client(C_i, C_j)\right] / (TC2 - TC2)$$

4. Conclusion:

Measurement can help to improve the software process, assist in the tracking and control of a project and asses the quality of a product. By analyzing metrics, a developer can correct those areas of software process that are the cause of software defects. A wide variety of object oriented metrics have been proposed to assess the testability of an object oriented system. Most of the metrics focus on encapsulation, inheritance, class complexity and polymorphism. CK metrics suite is a set of six metrics which capture different aspects of an OO design; these metrics mainly focus on the class and the class hierarchy. It includes complexity, coupling and cohesion as well. On the other hand MOOD metrics focus on system level which includes encapsulation, inheritance, polymorphism, and massage passing.

5. References:

[1] B. Henderson-sellers, Object-Oriented Metrics, Measures of Complexity .Prentice Hall, 1996.

[2] F.B. Abreu and R. Carapuca, "Candidate Metrics for Object- Oriented Software within a Taxonomy Framework,"]. System and Software, vol. 26, no. l, pp. 87-96, Jan. 1994.

[3] L. Briand, S. Morasca, and V. Basili, De$ning and Vdidating High- Level Design Metrics, Techtucal Report CS-TR-3301, Univ. of Maryland, Dept. of Computer Science, College Park, Md., 1994.

[4] L. Briand, S. Morasca, and V. Basili, "Property Based Software Engineering Measurement," IEEE Trans. Software Eng., vol. 22, no. 1, p. 68-86, Jan. 1996.

[5] L.Briand , W.Daly and J. Wust, Unified Framework for Cohesion Measurement in Object-Oriented Systems. Empirical Software Engineering, 3 65-117, 1998.

[6] L.Briand , W.Daly and J. Wust, A Unified Framework for Coupling Measurement in Object-Oriented Systems. IEEE Transactions on software Engineering, 25, 91 121,1999.

[7] L.Briand , W.Daly and J. Wust, Exploring the relationships between design measures and software quality. Journal of Systems and Software, 5 245-273, 2000.

[8] Lorenz, Mark & Kidd Jeff, Object-Oriented Software Metrics, Prentice Hall, 1994.

[9] McCabe and Associates, Using McCabe QA 7.0, 1999, 9861 Broken Land Parkway 4th Floor Columbia, MD 21046.

[10] McCabe, T. J., "A Complexity Measure", IEEE Transactions on Software Engineering, SE-2(4), pages 308-320, December 1976.

[11] Moreau, D. R., "A Programming Environment Evaluation Methodology for Object-Oriented Systems", Ph.D. Dissertation, University of Southwestern Louisiana, 1987.

[12] Moreau, D. R., and Dominick, W. D., "Object-Oriented Graphical Information Systems: Research Plan and Evaluation", Journal of Systems and Software, vol. 10, pp. 23-28, 1989.

[13] Moreau, D. R., and Dominick, W. D., "A Programming Environment Evaluation Methodology for Object-Oriented Systems: Part I – The Methodology", Journal of Object-Oriented Programming, vol. 3, pp. 38-52, 1990.

[14] I.Brooks, "Object-Oriented Metrics Collection and Evaluation with a Software Process," Proc. OOPSLA '93 Workshop Processes and Metrics for Object-Oriented Software Development, Washington, D.C., 1993.

[15] R.Harrison, S.J.Counsell, and R.V.Nithi, An Evaluation of MOOD set of ObjectOriented Software Metrics. IEEE Trans. Software Engineering, vol.SE-24, no.6, pp. 491-496 June1998.

[16] S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object- Oriented Design," IEEE Trans. Software Eng., vol. 20, no. 6, pp. 476493, June 1994.

[17] S.R. Chidamber and C.F. Kemerer, "Authors Reply," lEEE Trans. Software Eng., vol. 21, no. 3, p. 265, Mar. 1995.

[18] S.R.Chidamber and C.F.Kamerer, A metrics Suite for Object-Oriented Design. IEEE Trans. Software Engineering, vol. SE-20, no.6, 476-493, 1994.