OBJECT ORIENTED METRICS MEASUREMENT PARADIGM

Dr. K.P. Yadav*

Ashwini Kumar**

Sanjeev Kumar***

Abstract:

The increasing importance of software measurement has led to development of new software measures. Many metrics have been proposed related to various constructs like class, coupling, cohesion, inheritance, information hiding and polymorphism.

The central role that software development plays in the delivery and application of information technology, managers are increasingly focusing on process improvement in the software development area. it is very difficult for project managers and practitioners to select measures for object-oriented systems. This demand has spurred the provision of a number of new and/or improved approaches to software development, with perhaps the most prominent being object-orientation (OO).Tthe focus on process improvement has increased the demand for software measures, or metrics with which to manage the process. The need for such metrics is particularly acute when an organization is adopting a new technology for which established practices have yet to be developed. This research addresses these needs through the development and implementations of a suite of metrics for OO design.Object-oriented metrics require the use of classes. In VB classic, a class is a .cls or a .ctl file (class and usercontrol). In VB.NET, a class is defined by a Class block. VB.NET Structures and Interfaces are not regarded to as classes, whereas abstract classes are.

Key Words: Object Oriented, Design, Development, Metric, Measure, Coupling, Cohesion, Complexity, Size.

* SIET, Ghaziabad, UP.

** Shrivas, DKES-SCS/ GGS IP University, New Delhi.

*** Bareilly College, Bareilly, UP.

1. Introduction:

Object-Oriented Analysis and Design of software provide many benefits such as reusability, decomposition of problem into easily understood object and the aiding of future modifications. But the OOAD software development life cycle is not easier than the typical procedural approach. Therefore, it is necessary to provide dependable guidelines that one may follow to help ensure good OO programming practices and write reliable code. Object-Oriented programming metrics is an aspect to be considered. Metrics to be a set of standards against which one can measure the effectiveness of Object-Oriented Analysis techniques in the design of a system.

Five characteristics of Object Oriented Metrics are as following:

1. Localization operations used in many classes
2. Encapsulation metrics for classes, not modules
3. Information Hiding should be measured & improved
4. Inheritance adds complexity, should be measured
5. Object Abstraction metrics represent level of abstraction

We can signify nine classes of Object Oriented Metrics. In each of then an aspect of the software would be measured:

_ Size
_ Population (# of classes, operations)
_ Volume (dynamic object count)
_ Length (e.g., depth of inheritance)
_ Functionality (# of user functions)
_ Complexity

2. Chidamber & Kemerer's Metrics Suite:

Chidamber and Kemerer's metrics suite for OO Design is the deepest research in OO metrics investigation. They have defined six metrics for the OO design.

a) Weighted Methods per Class

Consider a Class C1, with methods M1... Mn that are defined in the class. Let c1... cn be the complexity of the methods.

If all method complexities are considered to be unity, then WMC = n, the number of methods.

Theoretical basis: WMC relates directly to Bunge's1 definition of complexity of a thing, since methods are properties of object classes and complexity is determined by the cardinality of its set of properties. The number of methods is, therefore, a measure of class definition as well as being attributes of a class, since attributes correspond to properties.

b) Depth of Inheritance Tree (DIT)

Depth of inheritance of the class is the DIT metric for the class. In cases involving multiple inheritance, the DIT will be the maximum length from the node to the root of the tree.

Theoretical basis: DIT relates to Bunge's notion of the scope of properties. DIT is a measure of how many ancestor classes can potentially affect this class.

Viewpoints:

• The deeper a class is in the hierarchy, the greater the number of 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.

• The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods.

The ontological principles proposed by Bunge in his “Treatise on Basic Philosophy” form the basis of the concept of objects.

While Bunge did not provide specific ontological definitions for object oriented concepts, several recent researchers have employed his generalized concepts to the object oriented domain.

c) Number of children (NOC)

NOC = number of immediate sub-classes subordinated to a class in the class hierarchy.

Theoretical basis: NOC relates to the notion of scope of properties. It is a measure of how many subclasses are going to inherit the methods of the parent class.

Viewpoints:

• Greater the number of children, greater the reuse, since inheritance is a form of reuse.

• Greater the number of children, the greater the likelihood of improper abstraction of the parent class. If a class has a large number of children, it may be a case of misuse of sub-classing.

• The number of children gives an idea of the potential influence a class has on the design. If a class has a large number of children, it may require more testing of the methods in that class.

d) Coupling between object classes

CBO for a class is a count of the number of other classes to which it is coupled.

Theoretical basis: CBO relates to the notion that an object is coupled to another object if one of them acts on the other, i.e., methods of one use methods or instance variables of another. As stated earlier, since objects of the same class have the same properties, two classes are coupled when methods declared in one class use methods or instance variables defined by the other class.

e) Lack of Cohesion in Methods

Lack of Cohesion (LCOM) measures the dissimilarity of methods in a class by instance variable or attributes. A highly cohesive module should stand alone; high cohesion indicates good class subdivision. Lack of cohesion or low cohesion increases complexity, thereby increasing the likelihood of errors during the development process. High cohesion implies simplicity and high reusability. High cohesion indicates good class subdivision. Lack of cohesion or low cohesion increases complexity, thereby increasing the likelihood of errors during the development process. Classes with low cohesion could probably be subdivided into two or more subclasses with increased cohesion.

3. Metrics for Object Oriented Design:

The MOOD metrics set refers to a basic structural mechanism of the OO paradigm as encapsulation (MHF and AHF), inheritance (MIF and AIF), polymorphisms (PF) , message-passing (CF) and are expressed as quotients. The set includes the following metrics:

Method Hiding Factor (MHF)

MHF is defined as the ratio of the sum of the invisibilities of all methods defined in all classes to the total number of methods defined in the system under consideration.

The invisibility of a method is the percentage of the total classes from which this method is not visible.

Attribute Hiding Factor (AHF)

AHF is defined as the ratio of the sum of the invisibilities of all attributes defined in all classes to the total number of attributes defined in the system under consideration.

Method Inheritance Factor (MIF)

MIF is 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.

Attribute Inheritance Factor (AIF)

AIF is 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.

Polymorphism Factor (PF)

PF is defined as the ratio of the actual number of possible different polymorphic situation for class Ci to the maximum number of possible distinct polymorphic situations for class Ci.

Coupling Factor (CF)

CF is defined as the ratio of the maximum possible number of couplings in the system to the actual number of couplings not imputable to inheritance.

4. Complexity Metrics and Models:

4.1 Halstead's Software Science

The Software Science developed by M.H.Halstead principally attempts to estimate the programming effort.

The measurable and countable properties are:

n1 = number of unique or distinct operators appearing in that implementation

n2 = number of unique or distinct operands appearing in that implementation

N1 = total usage of all of the operators appearing in that implementation

N2 = total usage of all of the operands appearing in that implementation

From these metrics Halstead defines:

I. the vocabulary n as n = n1 + n2

II. the implementation length N as N = N1 + N2

Operators can be "+" and "*" but also an index "[...]" or a statement separation " ; ". The number of operands consists of the numbers of literal expressions, constants and variables.

4.2 Length Equation

It may be necessary to know about the relationship between length N and vocabulary n.

Length Equation is as follows. " ' " on N means it is calculated rather than counted :

N ' = n1log2n1 + n2log2n2

It is experimentally observed that N ' gives a rather close agreement to program length.

4.3 Quantification of Intelligence Content

The same algorithm needs more consideration in a low level programming language. It is easier to program in Pascal rather than in assembly. The intelligence Content determines how much is said in a program. In order to find Quantification of Intelligence Content we need some other metrics and formulas:

Program Volume: This metric is for the size of any implementation of any algorithm.

V = Nlog2n

Program Level: It is the relationship between Program Volume and Potential Volume. Only the most clear algorithm can have a level of unity.

L = V* / V

Program Level Equation: is an approximation of the equation of the Program Level. It is used when the value of Potential Volume is not known because it is possible to measure it from an implementation directly.

L ' = n* 1n2 / n1N2

Intelligence Content

I = L ' x V = (2n2 / n1N2) x (N1 + N2)log2(n1 + n2)

In this equation all terms on the right-hand side are directly measurable from any expression of an algorithm. The intelligence content is correlated highly with the potential volume. Consequently, because potential volume is independent of the language, the intelligence content should also be independent.

4.4 Programming Effort

The programming effort is restricted to the mental activity required to convert an existing algorithm to an actual implementation in a programming language.

In order to find Programming effort we need some metrics and formulas:

Potential Volume: is a metric for denoting the corresponding parameters in an algorithm's shortest possible form. Neither operators nor operands can require repetition.

V ' = (n* 1 + n*2) log2 (n* 1 + n*2)

Effort Equation

The total number of elementary mental discriminations is:

E = V / L = V2 / V'

The implementation of any algorithm consists of N selections of a vocabulary n. a program is generated by making as many mental comparisons as the program volume equation determines, because the program volume V is a measure of it. Another aspect that influences the effort equation is the program difficulty. Each mental comparison consists of a number of elementary mental discriminations. This number is a measure for the program difficulty.

4.5 McCabe's Cyclomatic number

A measure of the complexity of a program was developed by McCabe. He developed a system which he called the cyclomatic complexity of a program. This system measures the number of independent paths in a program, thereby placing a numerical value on the complexity. In practice it is a count of the number of test conditions in a program.

The cyclomatic complexity (CC) of a graph (G) may be computed according to the following formula:

CC(G) = Number (edges) - Number (nodes) + 1

The results of multiple experiments (G.A. Miller) suggest that modules approach zero defects when McCabe's Cyclomatic Complexity is within 7 ± 2.

Complexity between 10 and 15 minimized the number of module changes.

4.6 Fan-In Fan-Out Complexity - Henry's and Kafura's

Henry and Kafura (1981) identified a form of the fan in - fan out complexity which maintains a count of the number of data flows from a component plus the number of global data structures that the program updates. The data flow count includes updated procedure parameters and procedures called from within a module.

Complexity = Length x (Fan-in x Fan-out)2

Length is any measure of length such as lines of code or alternatively McCabe's cyclomatic complexity is sometimes substituted.

Henry and Kafura validated their metric using the UNIX system and suggested that the measured complexity of a component allowed potentially faulty system components to be identified. They

found that high values of this metric were often measured in components where there had historically been a high number of problems.

5. Difference between an Interface and an Abstract class:

There are quite a big difference between an *interface* and an *abstract class*, even though both look similar.

- Interface definition begins with a keyword interface so it is of type interface
- Abstract classes are declared with the abstract keyword so it is of type class
- Interface has no implementation, but they have to be implemented.
- Abstract class's methods can have implementations and they have to be extended.
- Interfaces can only have method declaration (implicitly public and abstract) and fields (implicitly public static)
- Abstract class's methods can't have implementation only when declared abstract.
- Interface can inherit more than one interfaces
- Abstract class can implement more than one interfaces, but can inherit only one class
- Abstract class must override all abstract method and may override virtual methods
- Interface can be used when the implementation is changing
- Abstract class can be used to provide some default behavior for a base class.
- Interface makes implementation interchangeable
- Interface increase security by hiding the implementation
- Abstract class can be used when implementing framework

However, in practice when you come across with some application-specific functionality that only your application can perform, such as startup and shutdown tasks etc. The abstract base class can declare virtual shutdown and startup methods. The base class knows that it needs those methods, but an abstract class lets your class admit that it doesn't know how to perform those actions; it only knows that it must initiate the actions. When it is time to start up, the abstract class can call the startup method. When the base class calls this method, it can execute the method defined by the child class.

5. Future Work & Conclusion:

A metric for software model complexity which is a combination of some of the metrics mentioned above with a new approach. With this metric we can measure software's overall complexity. Also there are metrics for measuring software's run-time properties and would be worth studying more.

The metric suite is not adoptable as such and according to some other researches it is still premature to begin applying such metrics while there remains uncertainty about the precise definitions of many of the quantities to be observed and their impact upon subsequent indirect metrics. Analyzing and collecting the data can predict design quality. If appropriately used, it can lead to a significant reduction in costs of the overall implementation and improvements in quality of the final product. Using early quality indicators based on objective empirical evidence is therefore a realistic objective. According to my opinion it's motivating for the developer to get early and continuous feedback about the quality in design and implementation of the product they develop and thus get a possibility to improve the quality of the product as early as possible. It could be a pleasant challenge to improve own design practices based on measurable data. It is unlikely that universally valid object-oriented quality measures and models could be devised, so that they would suit for all languages in all development environments and for different kind of application domains. Therefore measures and models should be investigated and validated locally in each studied environment. It should be also kept in mind that metrics are only guidelines and perhaps not rules.

6. References:

- Shyam R. Chidamber, Chris F. Kemerer, A METRICS SUITE FOR OBJECT ORIENTED DESIGN, 1993
- Carnegie Mellon School of Computer Science, Object-Oriented Testing & Technical Metrics, PowerPoint Presentation , 2000
- Sencer Sultanoðlu, Ümit Karakaþ, Object Oriented Metrics, Web Document, 1998
- Linda H. Rosenberg, Applying and Interpreting Object Oriented Metrics
- Sencer Sultanoðlu, Ümit Karakaþ, Complexity Metrics and Models, Web Document, 1998
- Inheritance and Polymorphism—Specialization and Generalization: http://en.csharp-online.net/Inheritance_and_Polymorphism%E2%80%94Specialization_and_Generalization

- Jaana Lindroos, Code and Design Metrics for Object-Oriented Systems, 2004
- Ralf Reißing, Towards a Model for Object-Oriented Design Measurement
- Magiel Bruntink, Testability of Object-Oriented Systems: a Metrics-based Approach, 2003
- Aine Mitchell, James F. Power, Toward a definition of run-time object-oriented metrics, 2003
- Sencer Sultanoðlu, Ümit Karakaþ, Software Size Estimating, Web Document, 1998
- David N. Card, Khaled El Emam, Betsy Scalzo, Measurement of Object-Oriented Software Development Projects, 2001
- MSDN Library: http://msdn2.microsoft.com/en-us/library/default.aspx
- Practical Approach to Computer Systems Design and Architecture: http:// www.codeproject.com/ useritems/System_Design.asp
- Introduction: What is Object-Oriented Programming?: http://www.inf.ufsc.br /poo/smalltalk/ibm/tutorial/oop.html
- Abstraction and Generalization: http:// cs.wwc.edu/~aabyan/ PLBook/HTML/ AbsGen.html