Object Oriented Metrics
(A Survey Approach)

Seyyed Mohsen Jamali
Department of Computer Engineering
Sharif University of Technology
January 2006
Tehran Iran

Abstract
Given 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. 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). In addition, the 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 implementation of a suite of metrics for OO design.

Index Terms
Object Oriented, Design, Development, Metric, Measure, Coupling, Cohesion, Complexity, Size

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:

- Localization — *operations used in many classes*
- Encapsulation — *metrics for classes, not modules*
- Information Hiding — *should be measured & improved*
- Inheritance — *adds complexity, should be measured*
- 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
  How classes are interrelated
- Coupling
  # of collaborations between classes, number of method calls, etc.
- Sufficiency

Does a class reflect the necessary properties of the problem domain?

- Completeness
  Does a class reflect all the properties of the problem domain? (for reuse)
- Cohesion
  Do the attributes and operations in a class achieve a single, well-defined purpose in the problem domain?
- Primitiveness (Simplicity)
  Degree to which class operations can't be composed from other operations
- Similarity
  Comparison of structure, function, behavior of two or more classes
- Volatility
  The likelihood that a change will occur in the design or implementation of a class

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.
In this section we'll have a complete description of their metrics:

Metric 1: Weighted Methods per Class (WMC)
Definition: Consider a Class C1, with methods $M_1 ... M_n$ *that are defined in the class*. Let $c_1 ... c_n$ be the complexity of the methods. Then:

$$\text{WMC} = \sum_{i=1}^{n} ci$$

If all method complexities are considered to be unity, then WMC = n, the number of methods.
Theoretical basis: WMC relates directly to Bunge's[1] 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.
Viewpoints
• The number of methods and the complexity of 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 will inherit all the methods defined in the class.
• Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse.

Metric 2: Depth of Inheritance Tree (DIT)
Definition: 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.

---

[1] 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.

Metric 3: Number of children (NOC)
Definition: 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.

Metric 4: Coupling between object classes (CBO)
Definition: 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.
Viewpoints:
• Excessive coupling between object classes is detrimental to modular design and prevents reuse. The 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 measure of coupling is useful to determine how complex the testings of various parts of a design are likely to be. The higher the inter-object class coupling, the more rigorous the testing needs to be.

Metric 5: Response for a Class (RFC)
Definition: RFC = | RS | where RS is the response set for the class.
Theoretical basis: The response set for the class can be expressed as:
RS = { M } $\cup$ all i { Ri }
where { Ri } = set of methods called by method *i* and { M } = set of all methods in the class The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class[26]. The cardinality of this set is a measure of the attributes of objects in the class. Since it specifically includes methods called from outside the class, it is also a measure of the potential communication between the class and other classes.
Viewpoints:
• If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding required on the part of the tester.
• The larger the number of methods that can be invoked from a class, the greater the complexity of the class.
• A worst case value for possible responses will assist in appropriate allocation of testing time.

Metric 6: Lack of Cohesion in Methods (LCOM)
Definition: Consider a Class C1 with *n* methods M1, M2..., Mn. Let {Ij} = set of instance variables used by method Mi. There are *n* such sets {I1},... {In}. Let P = { (Ii,Ij) | Ii $\cap$ Ij = $\varnothing$ } and Q = { (Ii,Ij) | Ii $\cap$ Ij _ $\varnothing$ }. If all n sets {I1},... {In} are $\varnothing$ then let P = $\varnothing$.
LCOM = |P| - |Q|, if |P| > |Q|
= 0 otherwise[28]
Example: Consider a class C with three methods M1, M2 and M3. Let {I1} = {a,b,c,d,e} and {I2} = {a,b,e} and {I3} = {x,y,z}. {I1} $\cap$ {I2} is non-empty, but {I1} $\cap$ {I3} and {I2} $\cap$ {I3} are null sets. LCOM is the (number of null-intersections - number of non-empty intersections), which in this case is 1.
Theoretical basis: This uses the notion of degree of similarity of methods. The degree of similarity for two methods M1 and M2 in class C1 is given by:
$\sigma()$ = {I1} $\cap$ {I2} where {I1} and {I2} are the sets of instance variables used by M1 and M2

The LCOM is a count of the number of method pairs whose similarity is 0 (i.e. $\sigma()$ is a null set) minus the count of method pairs whose similarity is not zero. The larger the number of similar methods, the more cohesive the class, which is consistent with traditional notions of cohesion that measure the inter-relatedness between portions of a program. If none of the methods of a class display any instance behavior, i.e. do not use any instance variables, they have no similarity and the LCOM value for the class will be zero. The LCOM value provides a measure of the relative disparate nature of methods in the class. A smaller number of disjoint pairs (elements of set P) implies greater similarity of methods. LCOM is intimately tied to the instance variables and methods of a class, and therefore is a measure of the attributes of an object class.

Viewpoints:

• Cohesiveness of methods within a class is desirable, since it promotes encapsulation.
• Lack of cohesion implies classes should probably be split into two or more sub-classes.
• Any measure of disparateness of methods helps identify flaws in the design of classes.
• Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

3. MOOD (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.
note : inherited methods not considered.

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 C_i to the maximum number of possible distinct polymorphic situations for class C_i.

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. Some Traditional Metrics

There are many metrics that are applied to traditional functional development. The SATC[2], from experience, has identified three of these metrics that are applicable to object oriented development: Complexity, Size, and Readability. To measure the complexity, the cyclomatic complexity is used.

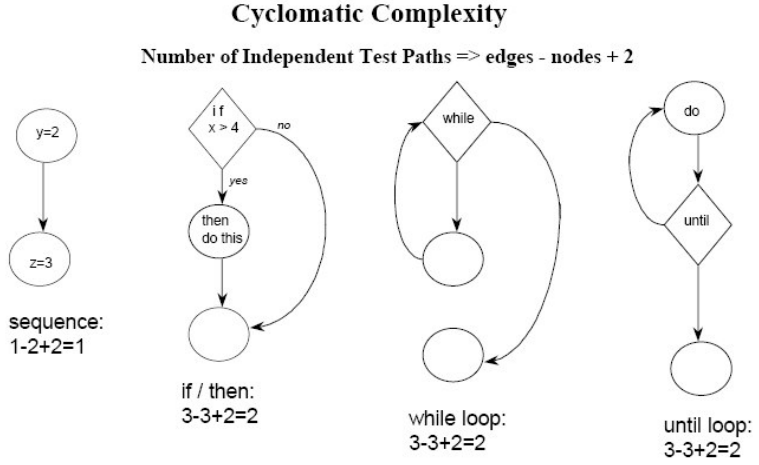
---

[2] Software Assurance Technology Center (SATC) at NASA Goddard Space Flight Center

Metric 1: Cyclomatic Complexity (CC)

Cyclomatic complexity (McCabe) is used to evaluate the complexity of an algorithm in a method. It is a count of the number of test cases that are needed to test the method comprehensively. The formula for calculating the cyclomatic complexity is the number of edges minus the number of nodes plus 2. For a sequence where there is only one path, no choices or option, only one test case is needed. An *IF* loop however, has two choices, if the condition is true, one path is tested; if the condition is false, an alternative path is tested.

Figure 1 shows a method with a low cyclomatic complexity is generally better. This may imply decreased testing and increased understandability or that decisions are deferred through message passing, not that the method is not complex. 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. Although this metric is specifically applicable to the evaluation of Complexity, it also is related to all of the other attributes



examples of calculations for the cyclomatic complexity for four basic programming structures.

Figure 1

Metric 2: Size

Size of a class is used to evaluate the ease of understanding of code by developers and maintainers. Size can be measured in a variety of ways. These include counting all physical lines of code, the number of statements, the number of blank lines, and the number of comment lines. Lines of Code(LOC) counts all lines. Non-comment Non-blank (NCNB) is sometimes referred to as Source Lines of Code and counts all lines that are not comments and not blanks. Executable Statements (EXEC) is a count of executable statements regardless of number of physical lines of code. For example, in FORTRAN and *IF* statement may be written:

IF X=3 THEN Y=0

This example would be 3 LOC, 3 NCNB, and 1 EXEC.

Executable statements is the measure least influenced by programmer or language style. Therefore, since NASA programs are frequently written using multiple languages, the SATC uses executable statements to evaluate project size. Thresholds for evaluating the meaning of size measures vary depending on the coding language used and the complexity of the method. However, since size affects ease of understanding by the developers and maintainers, classes and methods of large size will always pose a higher risk.

Metric 3: Comment Percentage

The line counts done to compute the Size metric can be expanded to include a count of the number of comments, both on-line (with code) and stand-alone. The comment percentage is calculated by the total number of comments divided by the total lines of code less the number of blank lines. Since comments assist developers and maintainers, higher comment percentages increase understandability and maintainability.

5. Complexity Metrics and Models

5.1 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:

- n_1 = number of unique or distinct operators appearing in that implementation
- n_2 = number of unique or distinct operands appearing in that implementation
- N_1 = total usage of all of the operators appearing in that implementation
- N_2 = total usage of all of the operands appearing in that implementation

From these metrics Halstead defines:

I. the vocabulary n as $n = n_1 + n_2$
II. the implementation length N as $N = N_1 + N_2$

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.

5.2 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 :
$N' = n_1 log_2 n_1 + n_2 log_2 n_2$
It is experimentally observed that N ' gives a rather close agreement to program length.

5.3 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.
$V = N\log_2 n$

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.
$L' = n^*_1 n_2 / n_1 N_2$

Intelligence Content
$I = L' \times V = (2n_2 / n_1 N_2) \times (N_1 + N_2)\log_2(n_1 + n_2)$
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.

5.4 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.
$V' = (n^*_1 + n^*_2) \log_2 (n^*_1 + n^*_2)$

Effort Equation
The total number of elementary mental discriminations is:
$E = V / L = V^2 / V'$
If we express it: The implementation of any algorithm consists of N *selections* (nonrandom > 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.

Time Equation
A concept concerning the processing rate of the human brain, developed by the psychologist John Stroud, can be used. Stroud defined a moment as the time required by the human brain to perform the most elementary discrimination. The Stroud number S is then Stroud's moments per second with 5 <= S <= 20. Thus we can derive the time equation where, except for the Stroud number S, all of the parameters on the right are directly measurable:
$T' = (n_1 N_2 (n_1 log_2 n_1 + n_2 log_2 n_2) log_2 n) / 2n_2 S$

5.5 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.
A study of PASCAL and FORTRAN programs (Lind and Vairavan 1989) found that a Cyclomatic Complexity between 10 and 15 minimized the number of module changes.

5.6 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.

$$\text{Complexity} = \text{Length} \times (\text{Fan-in} \times \text{Fan-out})^2$$

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.

6. Future Works
I've defined 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 (including all its components and classes). I'm trying to have sufficient experimental results to prepare a new paper (other than this survey). Also there are metrics for measuring software's run-time properties and would be worth studying more.

7. Conclusion

This paper introduces the basic metric suite for object-oriented design. 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. 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. For example the usefulness of the proposed metrics, and others, would be greatly enhanced if clearer guidance concerning their application to specific languages were to be provided.

Metric data provides quick feedback for software designers and managers. 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. The improved quality, in turn reduces future maintenance efforts. 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 not rules. They are guidelines that give an indication of the progress that a project has made and the quality of design.

8. References:

[1] Shyam R. Chidamber, Chris F. Kemerer, A METRICS SUITE FOR OBJECT ORIENTED DESIGN, 1993

[2] Carnegie Mellon School of Computer Science, Object-Oriented Testing & Technical Metrics, PowerPoint Presentation , 2000

[3] Sencer Sultanoðlu, Ümit Karakaþ, Object Oriented Metrics, Web Document, 1998

[4] Linda H. Rosenberg, Applying and Interpreting Object Oriented Metrics

[5] Sencer Sultanoðlu, Ümit Karakaþ, Complexity Metrics and Models, Web Document, 1998

[6] Jaana Lindroos, Code and Design Metrics for Object-Oriented Systems, 2004

[7] Ralf Reißing, Towards a Model for Object-Oriented Design Measurement

[8] Magiel Bruntink, Testability of Object-Oriented Systems: a Metrics-based Approach, 2003

[9] Aine Mitchell, James F. Power, Toward a definition of run-time object-oriented metrics, 2003

[10] Sencer Sultanoðlu, Ümit Karakaþ, Software Size Estimating, Web Document, 1998

[11] David N. Card, Khaled El Emam, Betsy Scalzo, Measurement of Object-Oriented Software Development Projects, 2001