An Introduction to Object-Oriented Metrics

1.1 Object-Oriented Metrics

Increasingly, object-oriented measurements are being used to evaluate and predict the quality of software [16]. A growing body of empirical results supports the theoretical validity of these metrics [3, 5, 14, 19]. The validation of these metrics requires convincingly demonstrating that (1) the metric measures what it purports to measure (for example, a coupling metric really measures coupling) and (2) the metric is associated with an important external metric, such as reliability, maintainability and fault-proneness [11]. Often these metrics have been used as an early indicator of these externally visible attributes, because the externally visible attributes could not be measures until too late in the software development process.

1.2 Limitations of Object-Oriented Metrics

ISO/IEC international standard (14598) on software product quality states, "Internal metrics are of little value unless there is evidence that they are related to external quality." It need be noted that the validity of these metrics can sometimes be criticized [9]. Many things, including fatigue and mental and physical stress, can impact the performance of programmers with resultant impact on external metrics. "The only thing that can be reasonably stated is that the empirical relationship between software product metrics are not very likely to be strong because there are other effects that are not accounted for, but as has been demonstrated in a number of studies, they can still be useful in practice. [11]"

2. METRICS FOR ANALYSIS

When code is analyzed for object-oriented metrics, often two suites of metrics are used, the Chidamber-Kemerer (CK) [8] and MOOD [1, 2] suites. In this section, we enumerate and explain the specific measures that can be computed using this tool.

2.1 Coupling

In 1974, Stevens et al. first defined coupling in the context of structured development as "the measure of the strength of association established by a connection from one module to another [21]."Coupling is a measure of interdependence of two objects. For example, objects A and B are coupled if a method of object A calls a method or accesses a variable in object B. Classes are coupled when methods declared in one class use methods or attributes of the other classes.

The *Coupling Fact*or *(CF)* is evaluated as a fraction. The numerator represents the number of non-inheritance couplings. The denominator is the maximum number of couplings in a system. The maximum number of couplings includes both inheritance and non-inheritance related coupling. Inheritance-based couplings arise as derived classes (subclasses) inherit methods and attributes form its base class (superclass). The CF metric is included in the MOOD metric suite.

Empirical evidence supports the benefits of low coupling between objects [6, 7, 20]. The main arguments are that the stronger the coupling between software artifacts, (i) the more difficult it is to understand individual artifacts, and hence to correctly maintain or enhance them; (ii) the larger the sensitivity of (unexpected) change and defect propagation effects across artifacts; and (iii) consequently, the more testing required to achieve satisfactory reliability levels. Additionally, excessive coupling between objects is detrimental to modular design and prevents reuse. To summarize, low coupling is desirable.

2.2 Cohesion

Cohesion refers to how closely the operations in a class are related to each other. Cohesion of a class is the degree to which the local methods are related to the local instance variables in the class. The CK metrics suite examines the Lack of Cohesion (LOCOM), which is the number of disjoint/non-intersection sets of local methods [12].

There are at least two different ways of measuring cohesion:

1. Calculate for each data field in a class what percentage of the methods use that data field. Average the percentages then subtract from 100%. Lower percentages mean greater cohesion of data and methods in the class.
2. Methods are more similar if they operate on the same attributes. Count the number of disjoint sets produced from the intersection of the sets of attributes used by the methods.

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. This metric evaluates the design implementation as well as reusability. [source]

2.3 Encapsulation

Information hiding is a way of designing routines such that only a subset of the module's properties, its public interface, are known to users of the module. Information hiding gives rise to encapsulation in object-oriented languages. "Encapsulation means that all that is seen of an object is its interface, namely the operations we can perform on the object [17]."Information hiding is a theoretical technique that indisputably proven its value in practice. "Large programs that use information hiding have been found to be easier to modify -- by a factor of 4 -- than programs that don't [4, 18] ."

The following two encapsulation measures are contained in the MOOD metrics suite.

2.3.1 Attribute Hiding Factor (AHF)

The Attribute Hiding Factor measures the invisibilities of attributes in classes. The invisibility of an attribute is the percentage of the total classes from which the attribute is not visible. An attribute is called visible if it can be accessed by another class or object. Attributes should be "hidden" within a class. They can be kept from being accessed by other objects by being declared a private.

The Attribute Hiding Factor is a fraction. The numerator is the sum of the invisibilities of all attributes defined in all classes. The denominator is the total number of attributes defined in the project [10]. It is desirable for the Attribute Hiding Factor to have a large value.

2.3.2 Method Hiding Factor (MHF)

The Method Hiding Factor measures the invisibilities of methods in classes. The invisibility of a method is the percentage of the total classes from which the method is not visible.

The Method Hiding Factor is a fraction where the numerator is the sum of the invisibilities of all methods defined in all classes. The denominator is the total number of methods defined in the project.

Methods should be encapsulated (hidden) within a class and not available for use to other objects. Method hiding increases reusability in other applications and decreases complexity. If there is a need to change the functionality of a particular method, corrective actions will have to be taken in all the objects accessing that method, if the method is not hidden. Thus hiding methods also reduces modifications to the code [10]. The Method Hiding Factor should have a large value.

2.4 Inheritance

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.

2.4.1 Depth of Inheritance Tree (DIT)

The depth of a class within the inheritance hierarchy is defined as the maximum length from the class node to the root/parent of the class hierarchy tree and is measured by the number of ancestor classes. In cases involving multiple inheritance, the DIT is the maximum length from the node to the root of the tree [8].

Well structured OO systems have a forest of classes rather than one large inheritance lattice. The deeper the class is within the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior and, therefore, more fault-prone [15]. Deeper trees require greater design complexity, since more methods and classes are involved [8]. Indeed, deep hierarchies are also a conceptual integrity concern because it becomes difficult to determine which class to specialize from [3]. Additionally, interface changes within the tree must be reflected throughout the entire class tree and object instances. However, the deeper a particular tree is in a class, the greater potential reuse of inherited methods [8].

Applications can be considered to be "top heavy" if there are too many classes near the root, and indication that designers may not be taking advantage of reuse of methods through inheritance. Alternatively, applications can be considered to be "bottom heavy" whereby too many classes are near the bottom of the hierarchy, resulting in concerns related to design complexity and conceptual integrity.

2.4.2 Number of Children (NOC)

This metric is the number of direct descendants (subclasses) for each class. Classes with large number of children are considered to be 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 [3].

2.5 Complexity

2.5.1 Weighted Methods/Class (WMC)

WCM measures the complexity of an individual class. A class with more member functions than its peers is considered to be more complex and therefore more error prone [3]. 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 a class. Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse. This reasoning indicates that a smaller number of methods is good for usability and reusability.

However, more recently, software development trends support have more, smaller methods over fewer, larger methods for reduced complexity, increased readability, and improved understanding [13]. This reasoning contradicts the prior reasoning of the last paragraph. The most recent recommendation (of more, smaller methods) is most likely more prudent. However, if a method is in a large inheritance tree having a large number of methods may not be advisable.

Often, the WMC calculation considers complexity and the count of the number of methods applies a weighted complexity factor [12].

2.6 Additional Measures

2.6.1 Number of Classes

If a comparison is made between projects with identical functionality, those projects with more classes are better abstracted.

2.6.2 Lines of Code

If a comparison is made between projects with identical functionality, those projects with fewer lines of code has superior design and requires less maintenance.

Additionally, methods of large size will always pose a higher risk in attributes such as Understandability, Reusability, and Maintainability.

2.7 Summary of Metrics

The table below summarizes the metrics discussed above. It illustrates, in general, whether a high or low value is desired from a metric for better code quality. However, one still must exercise judgment when determining the best approach for the task at hand.

Table 1: Summary of Metrics

Metric	**Desirable Value**
Coupling Factor	Lower
Lack of Cohesion of Methods	Lower
Cyclomatic Complexity	Lower
Attribute Hiding Factor	Higher
Method Hiding Factor	Higher
Depth of Inheritance Tree	Low (tradeoff)
Number of Children	Low (tradeoff)
Weighted Methods Per Class	Low (tradeoff)
Number of Classes	Higher
Lines of Code	Lower

References

[1] Abreu, F. B. e., "The MOOD Metrics Set," presented at ECOOP '95 Workshop on Metrics, 1995.
[2] Abreu, F. B. e. and Melo, W., "Evaluating the Impact of OO Design on Software Quality," presented at Third International Software Metrics Symposium, Berlin, 1996.
[3] Basili, V. R., Briand, L. C., and Melo, W. L., "A Validation of Object Orient Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, vol. 21, pp. 751-761, 1996.
[4] Boehm, B. W., "Improving Software Productivity," *IEEE Computer*, pp. 43-57, September 1987.
[5] Briand, L., Emam, K. E., and Morasca, S., "Theoretical and Empirical Validation of Software Metrics," 1995.
[6] Briand, L., Ikonomovski, S., Lounis, H., and Wust, J., "Measuring the Quality of Structured Designs," *Journal of Systems and Software*, vol. 2, pp. 113-120, 1981.
[7] Briand, L. C., Daly, J. W., and Wust, J. K., "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Transactions on Software Engineering*, vol. 25, pp. 91-121, January/February 1999.
[8] Chidamber, S. R. and Kemerer, C. F., "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, 1994.
[9] Churcher, N. I. and Shepperd, M. J., "Comments on 'A Metrics Suite for Object-Oriented Design'," *IEEE Transactions on Software Engineering*, vol. 21, pp. 263-5, 1995.
[10] Coad, P., "TogetherSoft Control Center," pp. http://togethersoft.com.
[11] El Emam, K., "A Methodology for Validating Software Product Metrics," National Research Council of Canada, Ottawa, Ontario, Canada NCR/ERC-1076, June 2000 June 2000.
[12] Fenton, N. E. and Pfleeger, S. L., *Software Metrics: A Rigorous and Practical Approach*: Brooks/Cole Pub Co., 1998.
[13] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, d., *Refactoring: Improving the Design of Existing*

Code. Reading, Massachusetts: Addison Wesley, 1999.

[14] Glasberg, D., Emam, K. E., Melo, W., and Madhavji, N., "Validating Object-Oriented Design Metrics on a Commercial Java Application," National Research Council 44146, September 2000.

[15] Gustafson, D. A. and Prasad, B., "Properties of Software Measures," in *Formal Aspects of Measurement*, T. Denvir, Ed. New York: Springer-Verlag, 1991.

[16] Harrison, R., Counsell, S. J., and Nithi, R. V., "An Evaluation of the MOOD Set of Object-Oriented Software Metrics," *IEEE Transactions on Software Engineering*, vol. 24, pp. 491-496, June 1998.

[17] Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G., *Object-Oriented Software Engineering: A Use Case Driven Approach*. Wokingham, England: Addison-Wesley, 1992.

[18] Korson, T. D. and Vaishnavi, V. K., "An Empirical Study of Modularity on Program Modifiability," *Empirical Studies of Programmers*, pp. 168-86, 1986.

[19] Schneidewind, N. F., "Methodology for Validating Software Metrics," *IEEE Transactions on Software Engineering*, vol. 18, pp. 410-422, 1992.

[20] Selby, R. W. and Vasili, V. R., "Analyzing Error-Prone Systems Structure," *IEEE Transactions on Software Engineering*, vol. 17, pp. 141-152, 1991.

[21] Stevens, W., Myers, G., and Constantine, L., "Structured Design," *IBM Systems Journal*, vol. 13, pp. 60-73, 1974.