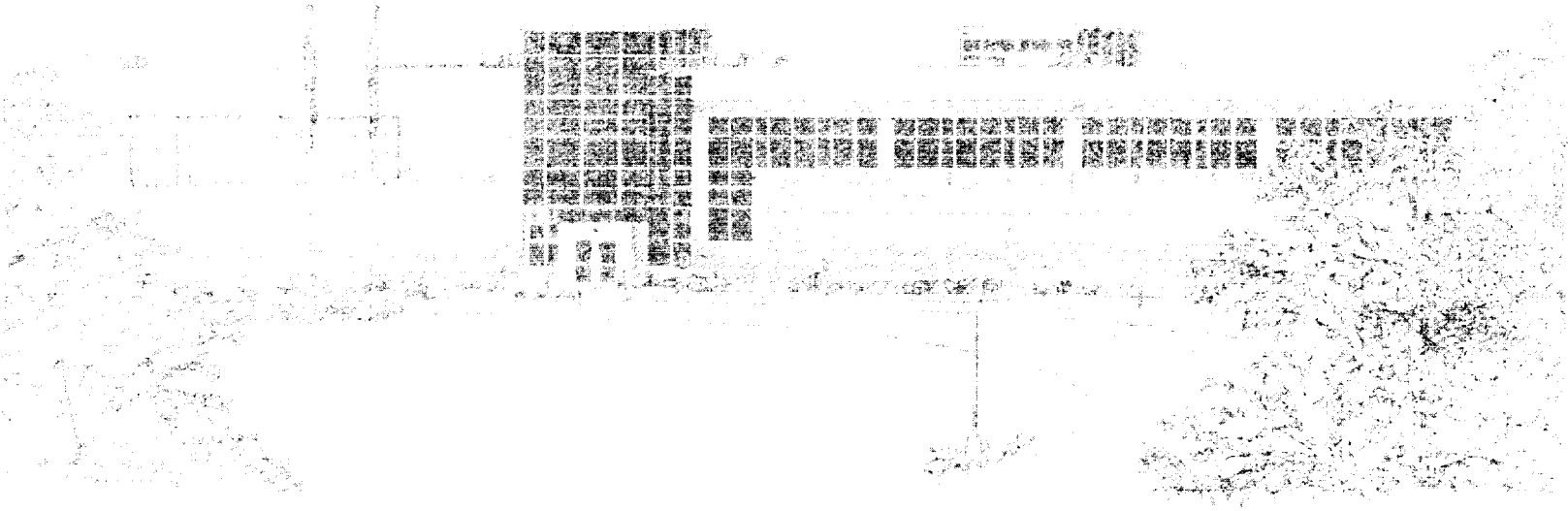
NASA/WVU Software IV & V Facility
Software Research Laboratory
Technical Report Series

NASA-IVV-97-003
WVU-IVV-97-003
WVU-CS-TR-97-006

An Improved Suite of Object Oriented Software Measures

By Ralph D. Neal, Roland Weistroffer, and Richard J. Coppins



National Aeronautics and Space Administration

West Virginia University

NASA-IVV-97-003

NASA IV&V Facility, Fairmont, West Virginia

An Improved Suite of Object Oriented Software Measures

Ralph D. Neal, H. Roland Weistroffer, and Richard J. Coppins

June 27, 1997

This technical report is a product of the National Aeronautics and Space Administration (NASA) Software Program, an agency wide program to promote continual improvement of software engineering within NASA. The goals and strategies of this program are documented in the NASA software strategic plan, July 13, 1995.

Additional information is available from the NASA Software IV&V Facility on the World Wide Web site http://www.ivv.nasa.gov/

This research was funded under cooperative Agreement #NCC 2-979 at the NASA/WVU Software Research Laboratory.

An Improved Suite of Object Oriented Software Measures

Ralph D. Neal
West Virginia University

H. Roland Weistroffer
Virginia Commonwealth University
School of Business, Richmond, VA 23284-4000

Richard J. Coppins
Virginia Commonwealth University
School of Business, Richmond, VA 23284-4000

Abstract

In the pursuit of ever increasing productivity, the need to be able to measure specific aspects of software is generally agreed upon. As object oriented programming languages are becoming more and more widely used, metrics specifically designed for object oriented software are required. In recent years there has been an explosion of new, object oriented software metrics proposed in the literature. Unfortunately, many or most of these proposed metrics have not been validated to measure what they claim to measure. In fact, an analysis of many of these metrics shows that they do not satisfy basic properties of measurement theory, and thus their application has to be suspect. In this paper ten improved metrics are proposed and are validated using measurement theory.

Introduction

In the early days of computer applications, software development was best described as art, rather than science. The absence of design guidelines often resulted in spaghetti code that was unintelligible to those charged with maintaining the software. In the pursuit of greater productivity, software development eventually became more structured and evolved into the field of software engineering. Part of the engineering concept is that the characteristics of the product must be controllable, and DeMarco [4] reminds us that what is to be controlled must be measured.

Measurement is the process whereby numbers or symbols are assigned to dimensions of entities in such a manner as to describe the dimension in a meaningful way. For example, inches or centimeters are meaningful in measuring the dimension height of the entity person. They are not meaningful however, to measure the age of a person. A requirement of meaningful measurement is that intuitive and empirical assessments of the entities and dimensions are preserved. For

example, when measuring the height of two people, the taller person should be assigned the larger value. A further requirement for meaningfulness is some model that defines how measurements are to be taken. For example, should posture be taken into consideration when measuring human height? Should shoes be allowed? Should the height be measured to the top of the head or the top of the hair? The model is necessary because a reasonable consensus by the measurers is needed [6].

Software development continues to evolve. In recent years, *object oriented* programming languages have become widely accepted, and to some extend the concepts of *structured programming* are being replaced or augmented by object oriented concepts. Object oriented software is organized as a collection of *objects*, where objects are entities that combine both data structure and behavior. By contrast, data structures and behavior are only loosely connected in traditional, structured programming (see for example [13]). Though there is no general agreement among authors on all the characteristics and on the exact terminology that define the object oriented paradigm, object oriented models can be summarized by the three properties of *encapsulation*, *abstraction*, and *polymorphism* [7]. Encapsulation, sometimes also called *information hiding*, refers to the concept of combining data and functions into objects, thus hiding the specifics from the user, who is given a more conceptual view of the objects, independent of implementation details. Abstraction refers to the grouping of objects with similar properties into classes. Common properties are described at the class level. A class may possess subclasses that describe further properties. Multiple levels of subclasses are allowed. Polymorphism is the ability of an object to interpret a message according to the properties of its class. The same message may result in different actions, depending on the class or subclass that contains the object. Subclasses inherit all the properties of their super classes, but may have additional properties not defined in the super class.

Because object oriented software has distinctly different characteristics from traditional, structured programs, different metrics are needed for object oriented software. Few such metrics were available until just a few years ago, but recently there has been an avalanche of newly proposed metrics [3] [9] [10]. Unfortunately, few of these metrics have been validated beyond some regression analysis of observed behavior.

Validation of a software metric is showing that the metric is a proper numerical characterization of the claimed dimension [1] [5]. Zuse [16] did an extensive validation and classification of conventional software metrics, using measurement theory. Neal [11] did a similar study for object oriented metrics and found that many of the proposed metrics cannot be considered valid measures of the dimension they claim to measure. The current paper proposes a suite of ten new metrics which have been validated using measurement theory, and which may replace some of those earlier published metrics that were not found to be valid measures.

The rest of this paper is organized as follows: In the next section, a model based on measurement theory and modified from Zuse's model [16] [17] [19] is described, for validating object oriented software metrics. Following that, ten new metrics are introduced and classified, based on this model.

The Validation Model

There are two fundamental considerations in measurement theory: *representation* and *uniqueness*. The representation problem is to find sufficient conditions for the existence of a mapping from an observed system to a given mathematical system. The sufficient conditions, referred to as representation axioms, specify conditions under which measurement can be performed. The measurement is then stated as a representation theorem. Uniqueness theorems define the properties and valid processes of different measurement systems and tell us what type of scale results from the measurement system. The scale used dictates the meaningfulness of measures or metrics [8] [12] [14].

To explain the representation problem, suppose we observe that Tom is the tallest of a group of three people, Dick is the shortest of the three, and Harry is taller than Dick and shorter than Tom. A *taller than* relationship among the three people has been empirically established [5]. Any measurement taken of the height of these three people must result in numbers or symbols that preserve this relationship. If it is further observed that Tom is *much taller th*an Dick, then this relationship must also be preserved by any measurement taken. That is, the numbers or symbols used to represent the heights of Tom and Dick must convey to the observer the fact that Tom is indeed *much* taller than Dick. If it is further observed that Dick towers over Tom when seated on Harry's shoulders, then another relationship has been established which must also be preserved by any measurement taken. This relationship might be represented in the real number system by .7 Dick + .8 Harry > Tom. Any numbers that resulted from measuring the height of Tom, Dick, and Harry would have to satisfy the observation represented by our formula.

To illustrate the uniqueness problem, let us consider two statements: 1) This rock weighs twice as much as that rock; 2) This rock is twice as hot as that rock. The first statement seems to make sense but the second statement does not. The ratio of weights is the same regardless of the unit of measurement while the ratio of temperature depends on the unit of measurement. Weight is a ratio scale, therefore, regardless of whether the weights of the rocks are measured in grams or ounces the ratio of the two is a constant. Fahrenheit and Celsius temperatures are interval scales, i.e., they exhibit uniform distance between integer points but have no natural origin. Because Fahrenheit and Celsius are interval scales, the ratio of the temperatures of the rocks measured on the Fahrenheit scale is different from the ratio when the temperatures are measured on the Celsius scale. Statements, such as those above, are meaningful only if they are unique, i.e. if their truth is maintained even when the scale involved is replaced by another admissible scale.

Metrics may be valid with respect to ordinal scales only, or they may be valid with respect to interval or ratio scales. The *ordinal scale* allows comparisons of the type "entity A is greater than entity B" or "entity A is at least as great as entity B." Rank order statistics and non-parametric statistics may be used with entities measured on an ordinal scale. The median is the most meaningful measure of centrality on an ordinal scale.

The *interval scale* allows the differences between measurements to be meaningful. Statements such as "the difference between A and B is greater than the difference between B and C" can be made about entities measured on an interval scale. When groups of entities are measured on the interval scale, parametric statistics as well as all statistics that apply to ordinal scales can be used, provided that the necessary probability distribution assumptions are being met. The arithmetic mean is the most powerful and meaningful measure of centrality.

Use of the *ratio scale* implies that the ratios of measurements are meaningful, as for example in measuring the density or volume of something. Statements such as "A is twice as complex as B" can be made about entities measured on the ratio scale. When groups of entities are measured on the ratio scale, percentage calculations as well as all statistics that apply to the interval scale can be used. The arithmetic mean is the most powerful and meaningful measure of centrality.

In order for a metric to be valid on an ordinal scale, it must be representative of the dimension under consideration (e.g. complexity or size) and it must satisfy the axioms of the weak order, i.e. completeness, transitivity, and reflexivity. In order for a metric to be valid on an interval or ratio scale, it must be valid on an ordinal scale and satisfy additional axioms. There are certain desirable properties that contribute toward the degree that a metric may be considered representative [15]. For example, a metric should satisfy intuition, i.e. it should make sense based upon the professional experience of the measurer. Entities that appear better in the dimension being measured (based on the observer's experience) should score higher on the metric being used. Entities that appear similar should score roughly the same. Consistency is another important feature. The measurement must be such that very nearly the same score is achieved regardless of the measurer, and the order in which the entities appear, in relation to each other, must be consistent from measurement to measurement. Further, in order for the metric to be useful, there must be sufficient variation in the measurement of different entities.

Zuse [16] [17] evaluated metrics of software code using flowgraphs to describe the possible structures being measured. Zuse defined modifications to the flowgraphs to describe the properties of the metric. The value of each metric increased, decreased, or stayed the same for each modification. The relation between the value of the metric taken before the modification to the flowgraph and the value of the same metric taken after the modification to the flowgraph is the *partial property* of the modification for this metric. Before a metric can be considered valid with respect to a specific scale, sufficient *atomic modifications* must be defined to describe the changes that can affect the metric, the partial properties of the metric must be established, and these partial properties must satisfy common intuition. An atomic modification is defined as the smallest change that can be made to an entity being measured, which will change the result of the measurement. There may be multiple possible atomic modifications for a metric.

Determining what the relevant atomic modifications are for each metric is itself a task based on intuition. Validation here is not a mathematical proof, but rather comparable to validation of scientific theories. Once sufficient evidence is found to support a theory, and as long as no contrary evidence is found, a theory is accepted as valid. The possibility of later refutal is always a reality.

If no atomic modification is found to contradict the premise of the measure, the measure may be accepted as valid on the ordinal scale. In order to be accepted valid on the ratio scale, the measure must preserve size relationships under concatenation of entities, i.e. the concatenated entities must have a measure equal to the sum (or the union, depending on the type of entities) of the measures of the original entities [18].

Newly Proposed Object-Oriented Software Metrics

1. Potential Methods Inherited (PMI)

PMI is defined as the maximum count of methods that could potentially be invoked by a class. PMI is offered as an improvement to the *Depth of the Inheritance Tree (DIT)* metric of Chidamber and Kemerer [3], and the *Number of Methods Inherited by a Subclass (NMI)* metric of Lorenz and Kidd [10]. Both are proposed as measures of complexity. DIT is defined as zero for a class that has no super class, i.e. the root node of the inheritance tree, one for each of the root node's immediate subclasses, two for the subclasses of these subclasses, etc. NMI is defined as the number of methods inherited by a class, i.e. the count of methods in all super classes. PMI differs from NMI in that PMI also counts the methods that are defined in the class itself, i.e. are not inherited.

As an example, assume in Figure 1 below that class A has five methods, class B has four methods, and class C has six methods. The NMI for classes B and C would be 5, the PMI for class B would be 9, and the PMI for class C would be 11. NMI for the combined class B+C in Figure 2 would still be 5. The PMI for the combined class B+C would be 15.

To see problems with the validity of the DIT metric, consider the case where a class with subclasses is combined with a sibling class (Figures 1 and 2 below). No change in DIT takes place indicating equal complexity, even though the number of methods inherited has increased. Alternatively, consider the case where a class is combined with its super class, thereby reducing the DIT for all of its subclasses (Figures 3 and 4 below). The new DIT would indicate that complexity has been reduced, even though no change in the number of methods has taken place.

To see problems with the validity of the NMI metric, consider the case where two sibling classes are combined (Figures 1 and 2 below). The NMI for the new, combined class is the same as the NMI was for the sibling classes before the combination, indicating equal complexity, even though the new class may contain many more methods. This problem does not occur with the proposed PMI metric, as it counts the local methods as well.

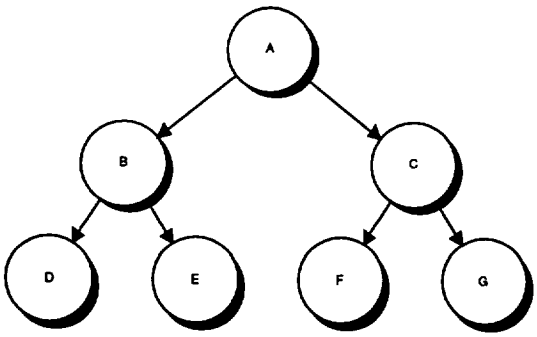


Figure 1

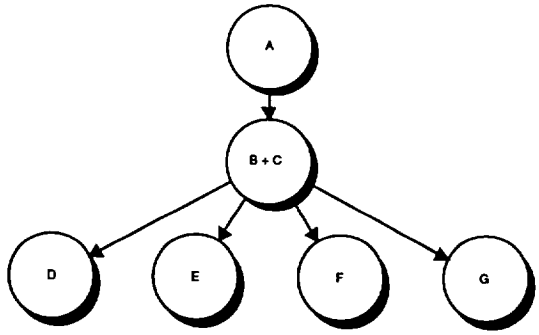


Figure 2

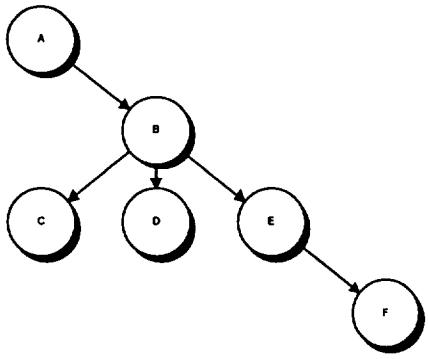


Figure 3

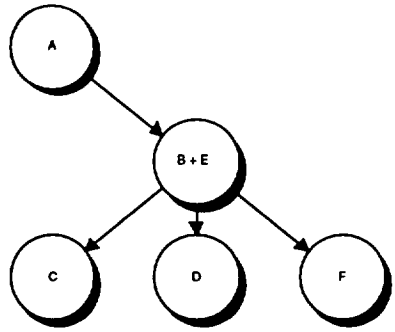


Figure 4

To further validate PMI as a measure of complexity, consider Figure 1 and the atomic modification of adding or deleting a class at some lower level of the inheritance tree. If for example, a new class is added as a subclass of D, E, F, or G, it is clear that the PMI of the added class is higher than that of the parent class. This supports our intuitive assumption that complexity increases (decreases) as methods are added to (deleted from) an existing inheritance tree or hierarchy chart. Irrespective of where the new class is added (deleted), the PMI of the classes effected, i.e. those classes that are at some lower level of the path on which the added or deleted class lies, will increase (decrease).

Consider the atomic modification of combining two classes that are not children and not siblings of each other, e.g. combine B and G in Figure 1, to get Figure 5. The PMI for class F remains unchanged. The PMI of classes D, and E increases, as does the PMI of the combined class.

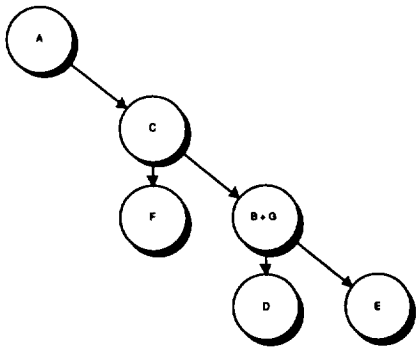


Figure 5

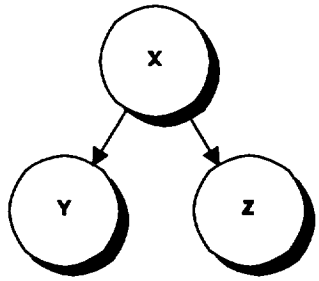


Figure 6

Finally, consider the atomic modification of combining a parent with its child class, as shown for example in Figures 3 and 4. The PMI of class F remains unchanged. The PMI of classes C and D increases.

As stated earlier, a measure can be used as an ordinal scale if the partial properties of the atomic modifications defined for that measure are acceptable and the axioms of the weak order hold. Thus, if the above examples pass the test of common sense and intuition, PMI can be accepted on the ordinal scale as a complexity measure of classes.

In order to validate PMI on the ratio scale, we need to show that PMI is additive under the concatenation patterns of object-oriented classes. Consider Figure 1 as a structure to which we wish to add Figure 6. Assume XYZ is inserted between A, B and C such that X is a subclass of A, B is a subclass of Y, and C is a subclass of Z. Let -old signify measurements taken before the merge, and let -new signify measurements taken after the merge. Since all methods in all classes along the path to the root node are counted, it is always the case that

$$PMI_{D\text{-}new} = PMI_{D\text{-}old} + PMI_{Y\text{-}old}$$
$$PMI_{E\text{-}new} = PMI_{E\text{-}old} + PMI_{Y\text{-}old}$$
$$PMI_{F\text{-}new} = PMI_{F\text{-}old} + PMI_{Z\text{-}old}$$
and $$PMI_{G\text{-}new} = PMI_{G\text{-}old} + PMI_{Z\text{-}old}.$$

In general, whenever concatenation takes place, the PMI of classes affected will increase by the number of methods in the classes that are being inserted within the path to the root node. Thus the PMI metric meets the measurement theory properties of the ratio scale.

Metrics should be easy to calculate, easy to capture, and be cost effective [2] [10] [17]. Actual methods inherited, as opposed to potential methods inherited, would be a more powerful measure of inheritance. However, calculating actual inheritance may not be possible in the early stages of program development, and when possible, would likely be extremely costly. PMI captures more information than DIT and NMI, meets the criteria of the ratio scale, and can be calculated at reasonable expense.

2. Proportion of Methods Inherited by a Subclass (PMIS)

PMIS is offered as an improvement to the NMI metric of Lorenz and Kidd [10]. PMIS is a measure of the strength of subclassing by inheritance. PMIS ranges along the closed interval [0,1], and is calculated by dividing the NMI of the subclass by PMI of the subclass, i.e. the total number of methods in the path including the subclass (see previous section). Thus,

PMIS = NMI/ PMI.

The partial properties of the metric are described by the atomic modifications of adding (subtracting) a method to the subclass being measured, and adding (subtracting) a method to a class along the path to the subclass being measured. As methods are added to (removed from) the subclass being measured, PMIS increases (decreases). As methods are added to (removed from) a super class of the subclass being measured, PMIS decreases (increases). If we accept that PMIS measures subclassing by specialization, then we can accept PMIS as an ordinal scale.

The measure PMIS can be parsed into two separate components. The dividend is the number of methods inherited by the subclass, and the divisor is the total number of methods available to the subclass. Both the dividend and the divisor are counts, which put them on the absolute scale. PMIS is the proportion of methods available to a subclass which are available through inheritance. Proportions are ratio scales [12]. Therefore, if we accept PMIS as a measure of specialization, PMIS can be accepted as a ratio scale.

3. Density of Methodological Cohesiveness (DMC)

DMC is proposed as an alternative to the *Lack of Cohesion in Methods (LCOM)* metric of Chidamber and Kemerer [3]. LCOM is calculated by looking at all possible pairs of methods within a class, and by counting those pairs with some common instance variables, and those pairs having zero common instance variables. LCOM is defined as the number of pairs with zero common instance variables, minus the number of pairs with some common instance variables, as long as this difference is non-negative, otherwise LCOM is taken to be zero. One problem with the LCOM metric is that a value of zero can mean anything between equal numbers of pairs with and without common instance variables, to all pairs having some common instance variables. The metric does not discriminate in these cases, and thus may be of little value.

DMC is defined as the number of pairs of methods that have some common instance variables within an object class, divided by the total number of pairs of methods in the class. DMC ranges along a continuum in the closed interval [0,1]. If S = the count of pairs of methods with some similarities, and if N = the total number of methods, then the total number of pairs of methods = (N*(N-1))/2, and DMC = S / ((N*(N-1))/2) = 2S/(N*(N-1)).

In order to describe the partial properties of DMC we apply the atomic modification of adding or deleting a method to the measured class. If the added method uses instance variables already being used in the class, DMC will increase in proportion to the number of methods already using the instance variables. If added methods use instance variables unique in the class, DMC will decrease in proportion to the number of methods added. This seems congruent to most users' understanding of inter-relatedness and thus their understanding of cohesiveness of methods. A measure can be used as an ordinal scale if the user accepts the partial properties of the atomic modifications defined for that measure and the axioms of the weak order hold. The axioms of the weak order hold and DMC can be accepted as an ordinal scale.

The measure DMC is the proportion of two counts, and proportions are ratio scales [12]. Therefore, if we can accept that the ratio of pairs of methods which have some common instance variables defines a measure of cohesiveness of methods within a class, DMC may be assigned to the ratio scale. Counting common instance variables within the method pairs would likely be an even better metric, but it would be harder and more costly to determine.

4. Messages and Arguments (MAA)

MAA undertakes to quantify the communication complexity of a method and is offered as an improvement to the *Number of Message-sends (NMS)* metric of Lorenz and Kidd [10]. Some messages require no arguments. These messages nevertheless add complexity to the method and must be accounted for in any metric which proposes to measure method complexity. Each argument required by a message adds additional complexity to the method and must likewise be accounted for in any metric which proposes to measure method complexity. MAA is a count of the number of message-sends in a method plus a count of the number of arguments in the messages. MAA ranges from 0 to *N* where *N* is a positive integer.

Let $|m|$ be the number of message-sends in a method, and let $|a_i|$ be the number of arguments in the i^{th} message-send. Then

$$MAA = \Sigma_i |a_i| + |m|.$$

The metric is designed to measure a method. Methods are designed to fulfill certain purposes and certain functions must be called in order to meet this design. These message-sends are not likely to be moved from one method to another. The partial property of interest is the addition (deletion) of message-sends or function calls to a method. As function calls are added (deleted), the value of MAA increases (decreases). The user would agree that a method grows more (less) complex as function calls are added to (deleted from) the method. The user would also agree that function calls with arguments add more complexity than function calls without arguments and that the more arguments a function call has, the more complexity it adds. If that is the case, MAA can be assigned to the ordinal scale.

The formation of a class is accomplished through the concatenation of methods. Concatenation can be either sequential or alternative. The alternative form of concatenation would involve the inclusion of an "IF" statement in the class to determine whether a method would be instantiated at run time. The sequential form is used whenever all methods are to be instantiated without exception. Let {M} be the set of methods concatenated to form a class and let M_i be the i^{th} method. Further, let MAA_i be the MAA of method M_i and let MAA_M be the MAA of the set {M}. Since there are no message-sends included in the "IF" statements added in the alternate form of concatenation and because message-sends are not merged in the sequential form of concatenation, then, $MAA_M = \Sigma_i MAA_i$. This being the case, it follows that if MAA is a valid measure of method complexity on an ordinal scale, then MAA is valid on a ratio scale.

5. Density of Abstract Classes (DAC)

DAC is offered as a complement to the *Number of Abstract Classes (NAC)* metric of Lorenz and Kidd [1]. Abstract classes facilitate the reuse of methods and state data. Methods in abstract classes are not instantiated but rather are passed on to subclasses through inheritance. DAC is proposed as a measure of reuse through inheritance. DAC is the proportion of abstract classes in a project and ranges along the closed interval [0,1].

Consider the hierarchy chart in Figure 1, where the nodes represent classes and subclasses, and the arcs represent inheritance. Assume that the main purpose of class C is to define a common

interface for classes F and G. Then class C cannot be instantiated and is known as an abstract class. Assuming that class C is the only abstract class in this program, then the DAC of this program is 1/7.

Since DAC is calculated solely from the classes within a program or project, the partial properties of the metric are described by (1) adding or subtracting an abstract class to the program or project, and by (2) adding or subtracting a class that is not an abstract class to the program or project. As abstract classes are added (subtracted), DAC increases (decreases). As classes that are not abstract classes are added (subtracted), DAC decreases (increases). If we accept the density of abstract classes as a measure of reusability through inheritance then DAC is a valid measure on an ordinal scale.

The measure DAC is a quotient where the dividend is the count of abstract classes and the divisor is the count of all classes. As counts, both the dividend and the divisor are absolute scales, and proportions of absolute scales are ratio scales [12]. Therefore, if we can accept that the count of abstract classes defines a measure of reuse through inheritance, DAC may be assigned to the ratio scale.

The advantage of DAC over NAC is that DAC allows us to compare programs of different sizes, as it measures a proportion rather than an absolute count.

6. Proportion of Overriding Methods in a Subclass (POM)

POM is the proportion of methods in a subclass that override methods from a superclass. POM ranges along the closed interval [0,1] and is proposed as an improvement to the *Number of Methods Overwritten (NMO)* metric of Lorenz and Kidd [10]. Lorenz and Kidd argue that subclasses should extend their superclasses, be specializations of the superclass, instead of overriding the methods of the superclasses. POM is offered as an inverse measure of specialization. A large POM may indicate a design problem.

Overriding methods in a subclass are those that have the same name as some methods in the superclass. Let |M| be the total number of methods in the subclass, and let |O| be the number of methods in the subclass that override methods in a superclass. Then

POM = |O|/|M|.

The partial properties of POM are described by adding or subtracting overriding methods to or from the subclass. As overriding methods are added (subtracted), POM increases (decreases). If we accept the proportion of overriding methods as an inverse measure of specialization then POM is a valid measure on an ordinal scale. The metric POM is a quotient where the dividend is the count of methods in a subclass that override methods from a superclass, and the divisor is the count of all methods in the subclass. As counts, both the dividend and the divisor are absolute scales, and proportions of absolute scales are ratio scales [12]. Thus DAC is a valid measure on the ratio scale.

The advantage of POM over NMO is that POM allows us to compare programs of different sizes, as it measures a proportion rather than an absolute count.

7. Unnecessary Coupling through Global Usage (UCGU)

UCGU is offered as an improvement to the *Global Usage (GUS)* metric of Lorenz and Kidd [10]. Whereas GUS counts the number of global variables, including system variables that are global to the entire system, class variables that are global to the class, and pool directories which are global to any classes that include them, UCGU counts the number of times global variables are invoked. Assume C_i is the number of class global variables in class i, G_i is the number of system variables invoked in class i, and P_i is the number of pool directories included in class i. Then

$$UCGU = \Sigma_i G_i + \Sigma_i C_i + \Sigma_i P_i.$$

The partial properties that define the measure are the addition or deletion of global variables, in their various guises, to the system. Since the instances of global variable usage are counted, adding or removing system global variables or pool directories without invoking them will not cause UCGU to change (though GUS would change). As class global variables are added (subtracted), UCGU increases (decreases). As system global variable or pool directory usage is added (subtracted), UCGU increases (decreases). If we accept UCGU as a measure of poor design and we also accept that the design deteriorates as global variables are added, then we can accept UCGU as a valid measure on an ordinal scale.

In order to establish UCGU as a ratio scale, we need to show that the measure UCGU is additive under the concatenation patterns of object-oriented classes. As an example, consider the hierarchy chart of Figure 1 as a structure to which we wish to add Figure 6. Clearly, if Figure 6 is inserted into Figure 1, the global variables in the two programs represented by the hierarchy charts are not affected. The instances of global variable usage are not affected by the merger of the two programs. UCGU of the merged program is equal to the addition of UCGU of the initial program and UCGU of the added program. Therefore UCGU can be accepted as a valid measure on the ratio scale.

8. Degree of Coupling between Classes (DCBO)

DCBO is offered as an improvement to the *Coupling between Object Classes (CBO)* metric of Chidamber and Kemerer [3]. According to Chidamber and Kemerer, excessive coupling among object classes can hinder reuse through the deterioration of modular design, and the greater the degree of coupling the more sensitivity to changes in other parts of the program. Whereas CBO is a count of other classes with which a specific class shares methods or instance variables, DCBO for a class is the count of methods utilized from other classes. DCBO ranges from 0 to N, where N is a positive integer. DCBO represents the outside-the-class methods utilization for the class being measured. If a class is entirely self-contained, the DCBO for that class is zero.

In order to describe the partial properties of DCBO we apply the following atomic modifications:

a) addition (deletion) of a method to the measured class which calls a method which resides in a different class,
b) addition (deletion) of a method to another class and the addition (deletion) of a call to the method from the measured class,

c) movement of a method from the class where it resides to the measured class which uses the method.

Consider Figure 1 and atomic modification a. If a new call to a method which resides in class E is added to class D, it is clear that DCBO of class D increases. The user would generally agree that inter-object complexity increases (decreases) as inter-object coupling is added to (deleted from) an existing hierarchy chart. DCBO would seem to meet that criterion.

If atomic modification b is applied to Figure 1 by adding a new method to class E and calling the new method from class D, it is clear that DCBO of class D increases. The user would generally agree that inter-object complexity increases (decreases) as coupling is added to (deleted from) an existing hierarchy chart. Again, DCBO would seem to meet that criterion.

If a method which resides in class E is moved to class D, i.e., the application of atomic modification c, and it is assumed that both classes need access to the method, it is clear that the DCBO of class E increases while the DCBO of class D decreases. The user would generally agree that while the over-all complexity has neither increased nor decreased the complexity of both D and E has changed.

Thus DCBO appears to be a valid measure on the ordinal scale. In order to establish DCBO as a ratio scale, we need to show that DCBO is additive under the concatenation patterns of object-oriented classes. Since DCBO is a measure of one aspect of a class's complexity, the concatenation pattern of interest is the merger of two classes. DCBO of class D+E is equal to $DCBO_D+DCBO_E-\Lambda_{DE}-\Lambda_{ED}$, where Λ_{DE} is the number of methods in D called by E, and Λ_{ED} is the number of methods in E called by D.

Because we are measuring the interaction between two classes, the merging of these classes changes the fundamental relationship of the methods within each to the methods in the other, i.e., what was interclass communication before the merge becomes intraclass communication after the merge. Total communication has remained the same but the fundamental relationships have changed. Let us define a *conservation of communication*: The merging of two classes does not change the amount of communication taking place, i.e., the number of methods calling other methods. However, the perspective of the communication in relation to object boundaries changes from interclass communication to intraclass communication whenever the merged classes utilize the methods of each other. The exact opposite effect takes place when a class is split into multiple classes.

Let us further define *DCWO* as the *Degree of Coupling within a Class*, i.e., the number of methods utilized within the class. Then, total communication for some predefined system is equal to DCBO + DCWO = κ, where κ is a constant for the system in question. Thus, if two classes are merged (or a class is split into two), $\kappa = DCBO_{old} + DCWO_{old} = DCBO_{new} + DCWO_{new}$. DCBO/$\kappa$ satisfies the requirements of the ratio scale.

9. Number of Private Instance Methods (PrIM)

PrIM is offered as a measure of information hiding by a class. The PrIM for a class is the count of the number of methods within the measured class which are declared to be private. These are the methods that cannot be called by other classes. Because the private methods are hidden to

other classes, this count is said to represent the amount of information hidden from the calling classes. PrIM ranges from 0 to N, where N is a positive integer.

In order to describe the partial properties of PrIM consider the addition (deletion) of a method to the measured class, which is declared as private and therefore cannot be called by another class. The PrIM of the measured class increases (decreases). It seems reasonable that the amount of information a class hides from other classes increases (decreases) as private methods are added to (deleted from) the class. A measure can be used as an ordinal scale if the measurer accepts the partial properties of the atomic modifications defined for that measure and the axioms of the weak order hold. The axioms of the weak order hold and the acceptance of PrIM as an ordinal scale seems clear.

In order to establish PrIM as a ratio scale, we need to show that PrIM is additive under the concatenation patterns of object-oriented classes. Since PrIM is a measure of the information hidden by a class, the required concatenation pattern would be to merge two classes and recalculate PrIM. Private methods are those methods that cannot be called by other classes. Consider Figure 1. The PrIM of class B+E is equal to $PrIM_B + PrIM_E - \partial$, where ∂ is the number of private methods that B and C hold in common, i.e., ∂ is the intersection of the methods of classes B and C. This is the equivalent to the union of two sets. Thus PrIM can be used as a ratio scale.

10. Strings of Message Links (SML)

SML is proposed as a measure of error-detection complexity. It is offered as an improvement to the *Strings of Message-sends (SMS)* metric of Lorenz and Kidd [10]. Whereas SMS is defined as the number of linked messages, SML is defined as the count of intermediate results that linked messages generate to feed to each succeeding message. The message linking form of coding makes intelligent error recovery more difficult. SML ranges from 0 to N, where N is a positive integer.

As an example, consider the Smalltalk command "*myAccount balance print*", which causes four messages to be strung together with no chance of detecting invalid intermediate results. SML is calculated by counting the potential nil and false conditions that are not accounted for in the code. SML for this example is three.

The expanded code is explained in the following diagram:

self account balance printToTranscript

First, the message *account* is passed to *self*. This message states that the portion of *self* that represents an account is to be used. *anAccount* is returned from this operation, or nil if the account is nonexistent. A nil value results in a run-time error.

anAccount balance (or nil)

Second, the message *balance* is sent to *anAccount* (the result of the previous message). The result of this operation is *aFloat*, or the account balance may be nil. A nil value again results in a run-time error.

aFloat printToTranscript (or nil)

Third, the message *printToTranscript* is sent to the object *aFloat* (the result of the previous message). This results in the balance portion of the account being formatted as a string, or false if the balance came back as another format, say as integer.

aString (or false)

Fourth, is the resulting string from the *printToTranscript* operation.

The partial property description of this measure is the addition (deletion) of message-sends to (from) the nested message structure. As statements are added to (deleted from) the nested structure, the value of SML increases (decreases), which is in compliance with the measurer's reasonable expectation. Thus, SML can be assigned to the ordinal scale.

Consider the merging or splitting of nested message-sends. If the above example of a Smalltalk command were split into two message-send statements, SML for each string would be one, and the total SML would be two. In general, merging two nested strings will result in an SML measure equal to the sum of the SMLs of the two nested strings, plus one. Further, in the above example, if we split the original command into four one message-send statements, the SML for each string becomes zero, and thus the total SML becomes Zero. Because SML is a count, possesses a natural zero, and has no transformations, SML appears to be a valid measure not only on the ratio scale, but on an absolute scale.

Conclusion

Validity may depend on the use to which the measure is to be applied. If one is looking for "red light" indicators that something may be wrong or that something may require extra attention to assure that nothing does go wrong, then an ordinal scale may be all that is required. These "red light" indicators help find abnormal conditions. Finding outliers seems to be the state-of-the-art at this time. However, to truly understand software and the software development process, we need to get a better grip on software measurement.

At a minimum, measures must be validated before they are placed in use. Not every metric that has been proposed in the literature states the dimension that it proposes to measure. However, validation of a metric requires the identification of this dimension. Thus, the very act of validating the measures will help define the many dimensions of object-oriented software.

References

[1] L. Baker, J. M. Bieman, N. Fenton, D. A. Gustafson, A. Melton, and R. Whitty, "A philosophy of software measurement", *The Journal of Systems and Software*, **12**, 277-281, 1990.

[2] W. Boehm, *Software Engineering Economics*. Englewood Cliffs: Prentice-Hall, 1981.

[3] R. Chidamber and C. F. Kemerer, "A metric suite for object oriented design", *IEEE Transactions on Software Engineering*, **20**(6), June 1994.

[4] DeMarco, *Controlling Software Projects*. New York: Yourdon Press, 1982.

[5] Fenton, *Software Metrics: A Rigorous Approach*. London: Chapman & Hall, 1991.

[6] Fenton, "Software measurement: A necessary scientific basis", *IEEE Transactions on Software Engineering*, **20**(3), March 1994.

[7] Henderson-Sellers, *A Book of Object-Oriented Knowledge*. New York: Prentice Hall, 1992.

[8] N. Hong, M. V. Mannino, and B. Greenberg, "Measurement theoretic representation of large, diverse model bases", *Decision Support Systems*, **10**, 1993.

[9] Li and S. Henry, "Object-oriented metrics that predict maintainability", *Journal of Systems and Software*, **23**, 111-122, 1993.

[10] Lorenz and J. Kidd, *Object-Oriented Software Metrics*. Englewood Cliffs: Prentice Hall, 1994.

[11] D. Neal, The measurement theory validation of proposed object-oriented software metrics, unpublished dissertation, Virginia Commonwealth University, 1996.

[12] S. Roberts, *Measurement Theory with Applications to Decisionmaking, Utility, and the Social Sciences*. Reading, Massachusetts: Addison-Wesley, 1979.

[13] Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Englewood Cliffs: Prentice Hall, 1991.

[14] W. Savage and P. Ehrlich, "A brief introduction to measurement theory and to the essays", in *Philosophical and Foundational Issues in Measurement Theory*. New Jersey: Lawrence Erlbaum Associates, 1992.

[15] J. Weyuker, "Evaluating software complexity measures", *IEEE Transactions on Software Engineering*, **14**(9), September 1988.

[16] Zuse, Me□theoretische Analyse von statischen Softwarekomplexit□tsma□en (in German), unpublished dissertation, Technical University Berlin, 1985.

[17] Zuse, *Software Complexity: Measures and Methods*. Berlin: Walter de Gruyter, 1990.

[18] Zuse, "Foundations of object-oriented software measures", *Proceedings of the Third IEEE International Software Metrics Symposium*, March 1995.

[19] Zuse and P. Bollmann, "Software metrics: Using measurement theory to describe the properties and scales of static software complexity metrics", *Sigplan Notices*, **24**(8), August, 1989.