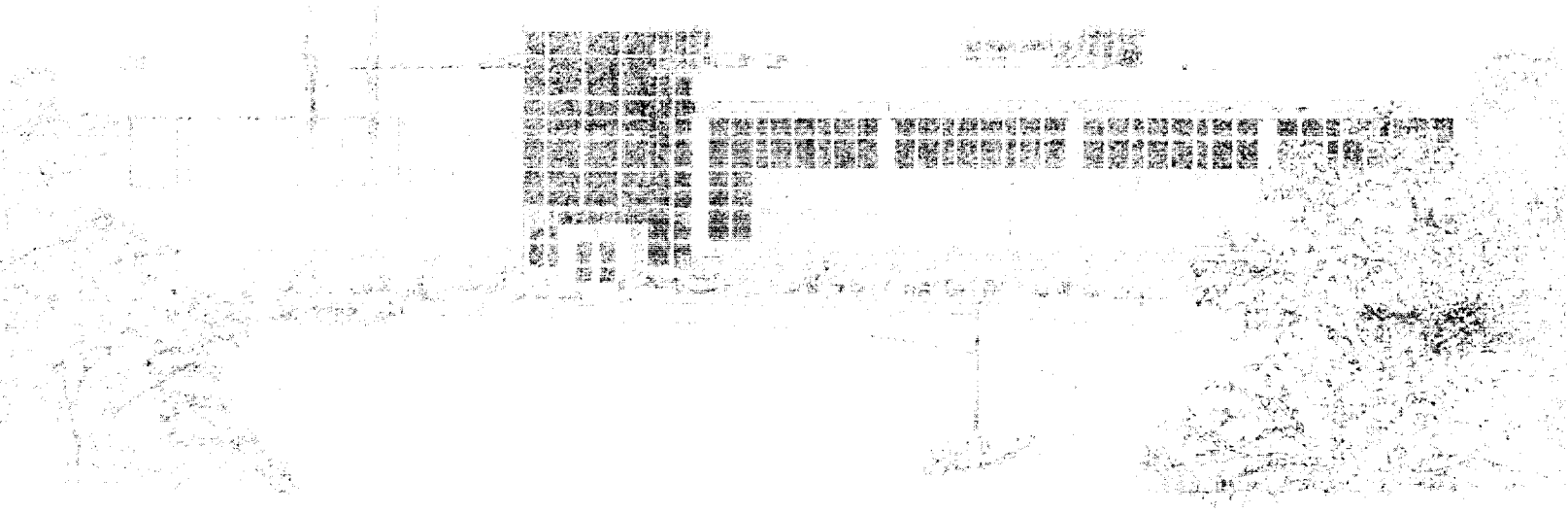
NASA-IVV-97-004
WVU-IVV-97-004
WVU-CS-TR-97-007

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

The Assignment of Scale to Object-Oriented Software Measures

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



National Aeronautics and Space Administration

West Virginia University

NASA-IVV-97-004

NASA IV&V Facility, Fairmont, West Virginia

The Assignment of Scale to 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.

The Assignment of Scale to Object-Oriented Software Measures

Ralph D. Neal
West Virginia University

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

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

Abstract

In order to improve productivity (and quality), measurement of specific aspects of software has become imperative. As object oriented programming languages have become more widely used, metrics designed specifically for object-oriented software are required. Recently a large number of new metrics for object-oriented software has appeared in the literature. Unfortunately, many of these proposed metrics have not been validated to measure what they purport to measure. In this paper fifty (50) of these metrics are analyzed.[1]

1. Introduction

Software development historically has been the arena of the artist. Artistically developed code often resulted in arcane algorithms or spaghetti code that was unintelligible to those who had to perform maintenance (or make it work in the first place). Initially only very primitive measures such as lines of code (LOC) and development time per stage of the development life cycle were collected. Projects often exceeded estimated time and budget. In the pursuit of greater productivity, software development evolved into software engineering. Part of the software engineering concept is the idea that the product should be controllable. DeMarco [8] reminds us that what is not measured cannot be controlled.

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. An entity may be a thing or an event, i.e., a person, a play, a developed program or the development process. A dimension is a trait of the entity, such as the height of a person, the cost of a play, or the length of the development process. Obviously, the entity and the dimension to be measured must be specified in advance. Measurements cannot be taken and then applied to just any dimensions. Unfortunately this is exactly what the software development community has been doing, e.g., lines-of-code, being a valid measurement of size, has been used to "measure" the complexity of programs [10] [24].

[1] Funded in part by NASA Cooperative Agreement NCCW-0040

Fenton [10] argued that much of the software measurement work published to date is scientifically flawed. ~~This is not a revelation.~~ Software metrics usually have been taken at face value. Because many people believe that any quantification is better than no quantification at all, just counting the lines of code (for example) was enough to give management the feeling of doing something to try to gain control of the software development process. After obtaining the quantification, management had to try to decide just what was described and how the development process was influenced. Fenton [9] stated that it is often the case that the general lack of validation of software metrics is the reason that managers do not know what to do with the numbers with which they are presented.

Fenton is not the only author who has observed this lack of scientific precision. Baker, et al., [1] said as much when they wrote that research in software metrics often is suspect because of a lack of theoretical rigor. Li and Henry [14] argued that validation is necessary for the effective use of software metrics. Schneidewind [20] stated that metrics must be validated to determine whether they measure what it is they are alleged to measure. Weyuker [22] stated that existing and proposed software measures must be subjected to an explicit and formal analysis to define the soundness of their properties.

In recent years, *object oriented* programming languages have become widely accepted. Object oriented software is recognized 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, Rumbaugh et. al., [18]). Though there is no general agreement among authors on all of the characteristics and on the exact terminology that define the object oriented paradigm, such models can be summarized by the three properties of *encapsulation*, *abstraction*, and *polymorphism*. Encapsulation, also called *information hiding*, refers to the concept of encapsulating data and functions (algorithms) into objects, thus hiding the specifics from the user, who sees only a conceptual view of the objects. Abstraction refers to the grouping of objects with similar properties into *classes*. Common properties are described at the level of the class, and a class may contain subclasses with further properties shared by subsets of objects of the original class. 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 upon the class or subclass that contains the object. Subclasses inherit all the properties of their superclasses, but may have additional properties not defined in the superclass.

Because object oriented software has distinctly different characteristics than traditional structured software, different metrics are needed for object oriented software. Within the last few years there has been a flood of newly proposed metrics [14] [7] [15]. Unfortunately, these new metrics have limited validation beyond regression analysis of observed behavior.

Validation of a software metric requires that the metric be shown to be a proper numerical characterization of the claimed dimension [1][9]. Zuse [23] [24] did an extensive validation and classification of conventional software metrics, using measurement theory. Neal [16] did a similar study for object oriented metrics. The current paper demsonstrates that many of the proposed metrics cannot be considered valid measures of the dimensions they are purported to measure.

The rest of this paper is organized as follows: in the next section, we describe measurement theory, describe Zuse's model [23] [24] [25] for validating metrics for structured software, and present an extension of the model for validating object-oriented software metrics is described. Following that the new model is applied to fifty (50) metrics which have been proposed for object oriented software.

2. The Validation Model

2.1 Measurement Theory

Fenton [9] described two meanings of validation. Validation in the narrow sense is the rigorous process of insuring that the measure properly represents the intended dimension of the software. Validation in the wide sense is the authentication of a prediction system using the dimensions of the software. Accurate prediction relies on careful measurement of the predictive dimensions. A model which accurately measures the dimensions is necessary but not sufficient for building an accurate prediction system [10]. The model, along with procedures for determining the parameters to feed the model, and procedures to elucidate the results all are necessary to build an accurate prediction system [9].

In the past, validation in the wide sense has been conducted without first carrying out validation in the narrow sense. In this paper the intent is to validate in the narrow sense the object-oriented software metrics that have appeared in the literature. This is a necessary step before these metrics can be used to predict such managerial concerns as cost, reliability, and productivity.

Measurement can be defined (conservatively) as counting concatenations (where the properties that hold for addition in the real number system hold for concatenation in the measurement system) of standard items that are approximately equal to the unit of measurement with respect to the property being measured.

Stevens [20] distinguished four main types of scale. In ascending order of measurement strength they are:

1. The *nominal* scale represents only differences among entities (e.g., the numbers on the jerseys of ball players).
2. The *ordinal* scale represents the order of entities with respect to some property (e.g., the ranking of ball players in importance to the team).
3. The *interval* scale represents intervals of a property (e.g., temperature in degrees Celsius).

4. The *ratio* scale represents ratios of a property (e.g., the team's won-loss record). [17] [19]

There are two fundamental problems in measurement theory: the *representation problem* and the *uniqueness problem*. The representation problem is to find sufficient conditions for the existence of a mapping from an observed system to a given mathematical system. More formally, given a particular empirical relational system R and a numerical relational system $\Re$, find sufficient conditions for the existence of a mapping from R into $\Re$. 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 [12] [17] [19].

Uniqueness theorems define the properties and valid processes of different measurement systems and tell us what type of scale results from the measurement system. Additionally, uniqueness theorems contribute to a theory of scales. According to this theory, the scale used dictates the meaningfulness of statements made about measures based on the scale [12] [17].

As an example of the representation problem, suppose in a group of three people we observe that Tom is the tallest of the three, Dick is the shortest of the three, and Harry is taller than Dick and shorter than Tom. Thus a *taller than* relationship among the three people has been empirically established [9]. 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 (.7Dick + .8Harry > 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 may 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 involving numerical scales are meaningful (such as the preceding) only if they are unique, i.e., only if their truth is maintained when the scale involved is replaced by another admissible scale.

Entities measured on the nominal scale are either equal to each other or not, i.e., entities may be said to belong to various groups. No other meaningful statements may be made comparing two entities measured on the nominal scale. When groups of entities are measured on the nominal scale, proportions are meaningful but many statistics should not be used. The mode is the only meaningful measure of centrality.

Based on the preceding discussion, it should be clear that any useful metric must be valid with respect to the ordinal scale, the interval scale or the ratio scale. The *strict ordinal scale* requires every entity to occupy a unique place in relation to the other entities, e.g., street numbers. The *weak ordinal scale* allows entities to measure equal to one another. Statements of the *greater than* type may be made about entities measured on the strict ordinal scale; statements of the *at least as great as* type may be made about entities measured on the weak ordinal scale. When groups of entities are measured on the ordinal scale, rank order statistics and non-parametric statistics can be used if the necessary probability distribution assumptions are present or non-critical to the statistical procedure. The median is the most powerful meaningful measure of centrality.

Use of the interval scale allows the differences between measurements to be meaningful, e.g., the measurement of calendar time. 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 the 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 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 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, viz., 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 [22]. 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 upon the observer's experience) should score better on the metric being used. Entities that appear similar should score roughly the same. Consistency is another important feature. The measurement should 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 should 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 to distinguish among them.

Recent work has questioned the applicability of the extensive structure (used to validate metrics on the interval and ratio scales) to object-oriented measures [3] [4] [5] [25]. In particular, it has been argued that the set theory *union* structure test can be used. The set theory union structure is a weaker test than the extensive structure test, i.e., the extensive structure dominates the set theory union structure. Then, if the extensive structure fails to produce a ratio scale, the set theory union structure will be used instead of the extensive structure, to test for a scale above the level of the ordinal scale [25]. We will call this the ratio´ scale.

The question really is: if a measure is ordinal but fails the extensive structure, is the measure strictly ordinal or would much valuable data be lost by not considering the measure as a higher order scale [3]? Parametric statistics have been shown to be more robust under the violation of the assumption of scale than nonparametric statistics [4]. Therefore, there are two reasons to extend the scale of a measure to a higher level. The metric may be more powerful than the ordinal scale will reflect and the parametric statistics that we wish to use to take advantage of this power are forgiving of miscategorization of scale.

2.2 Zuse's Model for Validating Structured Software Metrics

Zuse used the concept of flowgraphs to evaluate static metrics for structured code by defining *atomic modifications* to the flowgraphs to describe the properties of the metric. 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. The value of each metric increased, decreased or remained 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 after the modification is the *partial property* of the modification for the metric. Before a metric can be considered valid with respect to a certain scale, the atomic modifications must be defined to describe the changes that can affect the metric, the partial properties of the metric must be established, and then shown to satisfy common intuition.

Determining what the relevant atomic modifications are for each metric is itself a task based upon 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 presented, a theory is accepted as valid. The possibility of later refutal always is a reality.

2.3 Validating Object-Oriented Measures

Since not all of the metrics that will be validated are measures of source code, it will not be possible to use flowgraphs as the defining structure for all of the metrics. Structures must be defined for each metric depending on the atomic makeup of the metric. After the structure is defined, modifications must be designed such that all measured components of the structure are included in (at least) one of them. Then these modifications can be evaluated to determine the partial properties of the metric [24].

The validation procedure described here is an extension of Zuse's approach modified to be suitable for object-oriented software [16]. It uses the following seven steps:
1) Define the metric using a flowgraph or other appropriate structure.
2) Show an example of the calculation of the metric.
3) Define the atomic modifications used to describe the partial properties of the metric.
4) Assign the metric to the ordinal scale if (a) the measurer can accept the metric as a measure of the dimension in question, and (b) the axioms of the weak order hold based upon the atomic modifications defined in step (3).
5) Assign the metric to the ratio scale using additive operator for concatenation if the extensive structure validates it. When the extensive structure fails, consider the use of set theory to assign the metric to the ratio´ scale.
6) If the metric failed to be assigned to the ratio or ratio´ scales, determine if the mappings which result from the atomic modifications result in uniform differences between integer results. If so, the metric can be assigned to the interval scale.

3. Results of the Analysis

Because of the large number of metrics analyzed, details will be given only for one metric in each category.

3.1 Metrics which cannot be validated in the narrow sense

3.1.1.1 Subjective metrics

3.1.1 Attribute complexity (AC) [6]:

AC is the summation of subjective values assigned to each attribute in a class, i.e., $\Sigma R_{(i)}$, where $R_{(i)}$is the complexity value of attribute i. The value of each $R_{(i)}$ is assigned from Table 1, although no reasons were given for the values shown.

Attribute Rating	Assigned Value
Boolean or Integer	0
Char	1
Real	2
Array	3-4
Pointer	5
Record, Struct, or Object	6-9
file	10

Table 1. Chen and Lu's Argument/Attribute Value Table [6]

The order of the argument types probably comes from expert opinion. However, the values are arbitrary. Some arguments in Table 1 have multiple possible values. It is left to the measurer to determine the complexity value for a given record, struct, object or array. Such subjectivity will keep this metric from being a weak order [13].

Since the values in Table 1 are subjective, without any rules for assignment, it cannot be assured that the values would be consistent across measurers or even than the same measurer would arrive at the same score for a class if the class were to be measured multiple times. Without stringent, tested rules of value assignment this metric cannot be expected to produce the required monotony of measurement.

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. AC cannot be used as an ordinal scale because the axioms of the weak order do not hold. The subjectiveness of the weights from Table 1 makes it clear that without accompanying documentation on the assignment of weights, AC cannot be considered a valid measure.

Before a metric can be considered for the interval scale, it must meet the conditions for the ordinal scale. Before a metric can be considered for the ratio scale, it must meet the conditions for the ordinal scale. The AC metric of Chen and Lu does not meet the measurement theory properties of even the most rudimentary scale. Nor do the other three metrics in section 3.1.1, for the same reasons.

3.1.1.2 Operation argument complexity (OAC) [6]:

OAC is the summation of subjective values assigned to each argument in a class, i.e., $\Sigma P_{(i)}$, where $P_{(i)}$is the complexity value of argument i. The value of each $P_{(i)}$ is assigned from Table 1. No reasons are given for the values given in Table 1. The order of the argument types probably comes from expert opinion. However, the values are arbitrary. Some arguments, in Table 1, have multiple possible values. It is left to the measurer to determine the complexity value for a given record, struct, object or array.

3.1.1.3 Method complexity (MCX) [15]:

MCX is the summation of subjective values assigned to each message-send in a method, i.e., MCX = $\Sigma P_{(i)}$, where $P_{(i)}$is the complexity value of message-send i. The value of each $P_{(i)}$ is assigned from Table 2.

Message Type	Weight
API calls	5.0
Assignments	0.5
Binary expressions (Smalltalk) or arithmetic operators (C++)	2.0
Keyword messages (Smalltalk) or messages with parameters (C++)	3.0
Nested expressions	0.5
Parameters	0.3
Primitive calls	7.0
Temporary values	0.5
Unary expressions (Smalltalk) or messages without parameters (C++)	1.0

Table 2. Lorenz and Kidd's Message-Send Weight Table [15]

The values assigned to the message-sends are claimed by its authors to be meaningful in relation to each other. However, they go on to say that the values are not magic and could be adapted to fit any situation. Without experimental evidence to support the relative weights of each type of message-send this metric is not scientifically sound.

3.1.1.4 Operation complexity of a class (OCC) [6]:

OCC is the summation of subjective values assigned to each operation (method) in a class, i.e., $\Sigma O_{(i)}$, where $O_{(i)}$is the complexity value of method i. The value of each $O_{(i)}$ is assigned from Table 3.

Rating	Complexity Value
Null	0
Very low	1-10
Low	11-20
Nominal	21-40
High	41-60
Very high	61-80
Extra high	81-100

Table 3. Chen and Lu's Operations Complexity Value Table

No reasons are given for the values or the categories given in Table 3. However, its authors hint that they arrived at the table by imitating Boehm [2]. Boehm spent chapters explaining how he arrived at the tables he used. However, since Table 3 does not exactly match any table in [2] this does not help explain Table 3.

The problems with Boehm's model are well known. Consider that Boehm has spent quite some time refining his model and still the consensus is that the model must be calibrated for each company (or even unit) in order to provide consistent results. Without stringent, tested rules of measurement this metric cannot be expected to produce the required monotony of measurement.

3.1.2 Mixed attribute metrics

3.1.2.1 Specialization index (SIX) [15]:

According to Lorenz and Kidd [15], specialization through subclassing involves not only adding behavior through new methods, it also involves: 1) adding behavior to existing methods, 2) overloading methods with totally new behavior, and 3) effectively deleting existing methods by not using them.

Its authors claim to measure the quality of subclassing by the index:

$$\frac{\text{number of overloaded methods} * \text{class hierarchy nesting level}}{\text{total number of methods}}$$

Since SIX includes two very different components of object-oriented code (methods and class hierarchy nesting level), it is not clear that SIX fits even an ordinal scale. Who can say that a class with four methods, three of them overloaded at level two is equal in specialization through subclassing to a class with four methods, two of them overloaded at level three?

SIX is a valid ordinal metric when used to measure the quality of subclassing among subclasses in the same hierarchy nesting level. However, when subclasses are compared across hierarchy levels, the metric fails to measure the quality of subclassing at any scale.

3.1.2.2 Number of properties in a class (SIZE2) [14]:

According to its authors, SIZE2 is a measure of the size of a class. SIZE2 is a count of the number of attributes plus the number of (local) methods in a class. An attribute is defined as the data or variables that characterize the state of an object. SIZE2 ranges from 0 to N where N is a positive integer.

Since SIZE2 includes two very different components of object-oriented code, it is not clear that SIZE2 fits even an ordinal scale. Who can say whether a class with four attributes and zero methods is equal in size to a class with two attributes and two methods?

3.1.2.3 Class hierarchy (CH) [6]:

This metric is offered by its authors as a measure of complexity. CH is the summation of the depth of the class in the inheritance tree, the number of children of the class, the number of parents of the class, and the number of local or inherited methods available to the class. CH is a positive integer which ranges from 0 to N.

This metric is made up of four distinct parts. Each part is represented as a separate metric in this study. The first part is the metric DIT of Chidamber and Kemerer [7] (see section 3.1.3.1). DIT has been shown to be unacceptable as an ordinal scale. The second part is the metric NOC of Chidamber and Kemerer [7] (see section 3.2.1). NOC is indeterminate as a scale. Assignment of NOC to a scale is dependent on the domain knowledge of the user. The metric MUI of Lorenz and Kidd [15](see section 3.1.3.3) is the third part. MUI has been shown to be unacceptable as an ordinal scale. The metric NMI of Lorenz and Kidd [15] (see section 3.1.3.2) is the fourth part. NMI has been shown to be unacceptable as an ordinal scale.

The use in CH of three individual metrics which cannot be assigned to even the most rudimentary scale and one metric which is indeterminate because of the domain knowledge needed by the measurer causes CH to be unassignable to a scale. If the atomic modifications of the four primitive metrics upon which CH is based were applied to CH the results would remain the same as they were when they were applied to the primitive metrics. Thus, CH cannot be assigned to the ordinal scale.

3.1.3 Non-discriminate metrics

3.1.3.1 Depth of the inheritance tree (DIT) [7]; Class hierarchy nesting level (HNL) [15]:

DIT (HNL) has been proposed as an indicator of complexity. DIT (HNL) is a surrogate for the number of ancestor classes that could affect a class. According to Chidamber and Kemerer [7], 1) the deeper a class resides in the hierarchy the more methods it is likely to inherit and 2) the more methods inherited by a class, the more difficult it is to predict its behavior.

It seems reasonable that complexity increases (decreases) as classes are added to (deleted from) an existing hierarchy chart. However, let the new class be added as a subclass of B or C and the program's maximum DIT (HNL) does not increase. The measurer must decide if the complexity of the program increased.

DIT has other problems which would keep it from becoming other than a nominal scale. Chidamber and Kemerer found that DITs within a given project tend to cluster around one level and thus fail to discriminate one class from another.

3.1.3.2 Number of methods inherited by a subclass (NMI) [15]:

NMI is presented by its authors as a measure of the strength of subclassing by specialization. NMI is defined as the maximum count of methods that are inherited by a subclass. NMI ranges from 0 to N where N is a positive integer.

It can be shown that merging all sibling classes at the leaf node will result in a class that has the same NMI as each of the merged classes. Unless the measurer can accept that the two programs display equal strength of subclassing by specialization, NMI cannot be accepted as a measure on the ordinal scale.

3.1.3.3 Use of multiple inheritance (MUI) [15]:

MUI is presented as a measure of complexity. Lorenz and Kidd [15] do not tell us how to calculate MUI. Their assertion that multiple inheritance should be avoided could lead us to treat MUI as a dichotomous indicator which gets set on whenever multiple inheritance is present within a project. Of course it would also be possible to count the classes which have more than one superclass. Since a dichotomous indicator is looked at elsewhere (see RUS, section 3.3.1) and all dichotomous indicators function identically, the later definition for MUI will be assumed. MUI ranges from 0 to N where N is a positive integer.

Lorenz and Kidd admit that there have been times when allowing multiple inheritance would have allowed them to construct a "(seemingly) more elegant design." If the

assumption can be made that a more elegant design implies a design that is not more complex than the design chosen, then MUI cannot be accepted as an ordinal scale of complexity.

3.1.3.4 Global usage (GUS) [15]:

GUS is proposed as a measure of unnecessary coupling. This metric is a count of the total number of global variables in a system. This count includes system variables which are global to the entire system, class variables which are global to the instances of a class, and pool directories which are global to any classes which include them. GUS ranges from 0 to N, where N is a positive integer.

Adding a system global variable which potentially affects every class in the system adds as much to GUS as adding a class global variable which effects only the class within which it is declared. Since the system global variable is not counted once for each method which references it (or at least once for each class within the system) it is not weighted, in proportion to its potential impact, relative to a class global variable. Unless one can accept that each system global variable contributes equally with each class global variable to unnecessary coupling, GUS cannot be accepted as an ordinal scale.

3.1.3.5 Abstract data type coupling (ADC) [14]:

According to its authors, ADC is a measure of the coupling complexity caused by abstract data types. ADC is a count of the number of abstract data types defined in a class. Let ADT stand for an abstract data type and A be the set of ADTs. Then, ADC is the cardinality of $\{A\}$, e.g., ADC = $|A|$. The number of abstract data types may indicate the number of data structures dependent on the definitions of other classes. ADC varies from 0 to N, where N is a positive integer.

Li and Henry [14] talk about the type of ADT which is itself another class definition. This the normal use for ADTs. This type of ADT causes a particular type of coupling between the class containing the ADT and the class defined by the ADT because the calling class can access the properties of the ADT class. In order to get a measure for this type of ADT they count all ADTs. Since ADC is calculated solely from the ADTs within one class, the partial properties of the metric are described by adding (subtracting) an abstract data type to the class. As abstract data types are added (subtracted), ADC increases (decreases). ADC always increases (decreases) by the same value. However, all ADTs are not equally complex. A simple count does not seem comprehensive enough to measure the complexity of an ADT. Unless one can accept that each abstract data type contributes equally to the coupling complexity of a class, ADC cannot be accepted as an ordinal scale.

3.1.3.6 Number of class variables in a class (NCV) [15]:

NCV is presented by its authors as a possible indicator of poor design. Class variables are defined by its authors as localized globals that provide common objects to all

instances of a class. Class variables are used to initialize the customizable constant features of a class. According to Lorenz and Kidd [15], the number of these variables, relative to the number of instance variables, should be small. NCV is a count of the variables which provide customizable constant values and coordinate information across all instances. NCV ranges from 0 to N where N is a positive integer.

NCV cannot be considered a measure of poor design because class variables are sometimes necessary for the good design of a class, e.g., common constant values [15]. Although one would generally agree that the design of a class was poor if NCV were too large, i.e., if the workload were being accomplished by the class itself instead of the instances within the class, one could not necessarily agree that the design of a class was good if NCV were small. If a number of methods contain local variables that have the same constant value repeated in each method, NCV would be small but the design would not be optimal.

Lorenz and Kidd state that class variables are often used to provide customizable constant values that are used to affect all the instances' behavior. Perhaps NCV should be considered a measure of the customizability of a class. Since this metric counts the class variables within a class, the proper partial property description of the measure is the addition (deletion) of class variables to an existing class. One could agree that customizability increases (decreases) when class variables are added (deleted) from a class. In fact, NCV is viable under the extensive structure and may be used on the ratio scale as a measure of customizability.

The required customizability of a class varies greatly depending on the function of the class. So, although NCV can measure customizability, because of the variation in required customizability this measurement may not mean a class has either good or bad design. However, when NCV is large (the authors use three as a ceiling) the class should be examined for poor design.

3.3.3.7 Coupling between object classes (CBO) [7]: **Class coupling (CC)** [6]: **Class coupling (CCP1)** [15]:

CBO (CC, CCP1) has been proposed as an indicator of reusability. Object classes are said to couple whenever one of them uses a method or an instance variable of the other. CBO (CC, CCP1) is a count of the number of other classes with which a class shares methods or instance variables. The coupling between object classes ranges from 0 to *N*, where *N* is a positive integer.

The CBO (CC, CCP1) for a class is the count of other classes that contain methods and instance variables which the class being measured utilizes plus the number of classes which utilize methods and instance variables which reside in the class being measured. If a class is entirely self-contained, the CBO (CC, CCP1) for that class is zero.

The acceptance of CBO (CC, CCP1) as an ordinal scale is not clear. CBO (CC, CCP1) as a metric discriminates poorly. Two classes that couple to the same number of

outside classes may invoke a vastly different number of outside methods. While CBO (CC, CCP1) may be an indicator of *inter-object complexity*, it would not seem to be an ordinal measurement.

3.1.3.8 Number of methods overridden (overloaded) by a subclass (NMO) [15]:

NMO is presented by its authors as a possible indicator of poor design. NMO is the count of methods in subclasses which have the same name as methods in their superclasses. Lorenz and Kidd [15] argue that subclasses should extend their superclasses, i.e., be specializations of the superclass, rather than overloading the methods of the superclasses. Thus, they argue a large NMO indicates a design problem. NMO ranges from 0 to N where N is a positive integer.

Some object-oriented constructs are designed to define methods which are supposed to be overloaded by their subclasses. According to Lorenz and Kidd, the subclass methods which overload these superclass methods which are designed to be overloaded should not be counted in NMO. This restraint makes it very hard to authenticate the metric value. In fact, it may not be clear which constructs are designed to be overloaded. This ambiguity can cause misunderstanding among measurers and lead to anomalies in measurement. An anomaly occurs when the rules are so ambiguous that different measurers (or the same measurer measuring the same object multiple times) cannot reach values which differ only by an amount attributable to the measurement error of the system.

Because there are no clear rules to define constructs which are designed to be overloaded, two classes may be assigned the same large NMO when one has a design problem and the other does not. Thus, NMO does not differentiate classes with design problems from classes without design problems. Therefore, NMO cannot be accepted as an ordinal scale of complexity.

3.3.3.9 Lack of cohesion in methods (LCOM) [7]: Class cohesion (CCO) [15]:

LCOM (CCO) has been proposed as a measure of cohesiveness of methods within a class. LCOM (CCO) looks at the similarity between pairs of methods. Given all possible pairs of methods within a class, each pair will have no common instance variables (zero similarities) or will have some common instance variables (some similarities). LCOM (CCO) is the non-negative result of the count of pairs having zero similarities minus the count of pairs having some similarities. That is to say: given N = the count of pairs of methods with no similarities and S = the count of pairs of methods with some similarities, LCOM (CCO) = N - S if N > S otherwise LCOM (CCO) = 0.

Because of the seemingly arbitrary decision to disallow negative numbers, many classes group together at the bottom of the scale. This grouping causes many potentially very different classes to be assigned the same value of LCOM (CCO). Therefore, LCOM (CCO) does not constitute a homomorphism.

Because LCOM (CCO) does not constitute a homomorphism, LCOM (CCO) cannot be assigned to an ordinal scale.

3.1.4 Metrics that do not fit the weak order

3.1.4.1 Cohesion (COH) [6]:

COH is presented by its authors as a percentage of disjointedness based on the sets of method arguments in a class. The term arguments includes all objects passed to or returned from a method. Given N methods in a class, there are also N sets of arguments. Then, M is defined as the number of disjoint sets formed by the intersection of the N sets of arguments. M is calculated by placing the arguments of a method from the class in a set. Each other method is examined for arguments that are part of the set. When a matching argument is found, the arguments of the new method are added to the set. This procedure continues until no more methods exist which intersect with the set. The process starts over with the remaining methods and continues until all methods have been placed into a set. By definition, the resulting sets will be disjoint.

COH is defined as the number of disjoint sets divided by the total number of sets times 100, i.e., COH=(M/N)*100. M and N are positive integers which range from one to N, i.e., M≤N. COH is a positive real number which ranges from 100/N to 100.

Even a completely cohesive class (only one disjoint set results from the intersection of the sets of parameters of the class, i.e., M=1) is measured by this metric to have some degree of disjointedness. Two completely cohesive classes which have a different number of methods, i.e., a different N, will be assigned different values of COH. Thus, COH does not fulfill the axioms of the weak order. A measure cannot be used as an ordinal scale unless the axioms of the weak order hold, therefore, COH cannot be accepted as an ordinal scale.

3.1.4.2 Number of classes/methods thrown away (NCT) [15]:

NCT is presented by the authors as a measure of the maturity of the system design. NCT is a count of classes or methods which have been rewritten for a given project. According to Lorenz and Kidd [15] functionality in the form of processing speed, resilience and over-all quality can be realized by throwing away classes and methods which were not designed correctly the first (or subsequent) time. NCT ranges from 0 to *N*, where *N* is a positive integer.

Although one might be tempted to argue that small NCT indicates a good initial design, Lorenz and Kidd argue that developers do not get designs correct the first time and part of the design is learning about the domain and building it better the next time. Likewise, one might be tempted to argue that a large NCT indicates poor design or lousy programmers, but Lorenz and Kidd argue that true iterations on designs are essential to the development of elegant solutions. So, even though NCT can be assigned to an

ordinal scale, NCT does not seem to be an indication of the value of rewriting classes or methods. To truly be a measure, NCT should indicate the gain in quality from rewrite to rewrite.

3.2 Ambiguous metrics/measures

3.2.1 Number of children (NOC) [7]:

NOC has been proposed as an indicator of complexity because it is a surrogate for the number of classes that might inherit methods from a parent. NOC is the number of immediate subclasses subordinate to a class in the class hierarchy. According to Chidamber and Kemerer [7], 1) the greater the number of children, the greater the inheritance and 2) the more children a parent class has, the greater the potential for improper abstraction of the parent class.

The class hierarchy chart represents the inheritance tree (see Figure 1). Classes are represented by the nodes of the diagram. Inheritance is represented by the inbound arcs of the diagram. The number of immediate subclasses subordinate to a class in the class hierarchy ranges from 0 to *N*, where *N* is a positive integer.

The NOC of leaf nodes is zero, by definition.

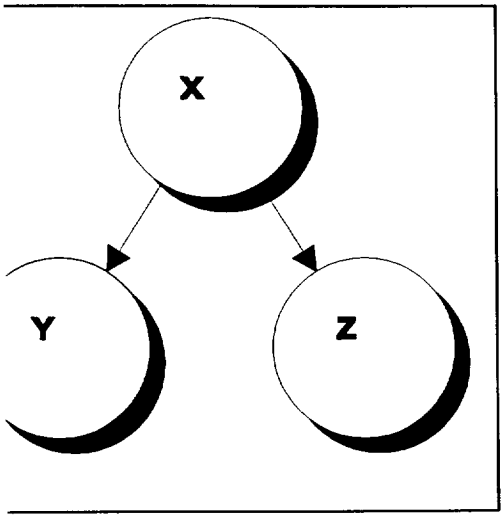
Consider Figure 1. Let a new class be added as a subclass of any existing class. It is clear that the NOC of that class increases. It seems reasonable that complexity increases (decreases) as classes are added to (deleted from) an existing hierarchy chart.

When complexity is defined as the ability of the developer to understand the solution, the ideal situation is to have enough information available without having information overload. If two sibling classes have limited information, the combination of them may simplify the solution. However, if either already is complex, their combination will be even more complex. The decision about the complexity of the combined class compared to the two individual classes cannot be made based solely on the metric NOC.

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. While the axioms of the weak order hold, the acceptance of NOC as an ordinal scale is not clear. It remains for the measurer to determine if NOC meets all of the partial properties of the atomic modifications.

Fig. 1 A Hierarchy Chart

3.2.2 Number of message-sends (NMS) [15]:



The metric is designed to measure the size of a method. According to its authors, NMS measures the size of the method in a rather unbiased way as it is unaffected by developer style. Its authors state that the messages should be segregated by type of message. They define three types of messages, viz., unary (without parameters), binary (with one parameter), and keyword (with more than one parameter). Its authors do not explain how the message type is to be integrated into the metric. Only one number is generated and only one threshold is given by its authors. Therefore, assume that all message types are to be treated the same. NMS is a positive integer and ranges from 0 to *N* where *N* is a count of the outgoing messages without regard to type.

NMS is presented as a rather unbiased measure of method size. Three types of messages are listed but not differentiated by the authors. If the measurer can accept that the method size is not influenced by the type of message, NMS may be used as a ratio scale.

3.3 Nominal measures

3.3.1 Reuse (RUS) [6]:

RUS is a dichotomous indicator: RUS is zero if the class is a completely new class and is not a superclass, i.e., the class was developed for the current project, is used only once in the project, and is not reused through inheritance; RUS is one if the class is fully or partially made up of reused parts or if it is reused by a subclass via inheritance.

No metric that can take on only two possible values is discriminate enough to be assigned to the ordinal scale. By definition a nominal scale assigns things to groups. RUS assigns classes to the group *reused* if it has the value 1 and assigns classes to the group *new* if it has the value 0. The metric RUS can be assigned to the nominal scale.

3.4 Ordinal measures

3.4.1 Operation coupling (OF) [6]: Class coupling (CCP2) [15]:

OF (CCP2) is the summation of the number of methods within the measured class which access other classes and the number of methods within the measured class which are accessed by other classes. The number of methods within the measured class which both access a method in another class and are accessed by the method in the other class are counted only once. Its authors call this last type of method co-operated methods but they give no example. If co-operated methods are included in the first two counts, the number of methods within the measured class which are co-operated with other classes has to be subtracted from the sum of the first two counts. If a class is entirely self-contained, the OF (CCP2) for that class is zero. The method coupling metric ranges from 0 to *N*, where *N* is a positive integer.

It is possible (or even probable) that the inter-object complexity of a class may well be related to the number of inter-object links identified with the class. However, based on the measurement theory validation of OF (CCP2), becaue OF (CCP2) cannot account for the methods in one class called by the other class when two classes are merged, OF (CCP2) is a valid measurement only on the ordinal scale.

3.4.2 Strings of message-sends (SMS) [15]:

SMS is presented by its authors as a measure of complexity even though they state that they are unsure of the usefulness of this metric. SMS is a count of the number of messages strung together each feeding the next with an intermediate result. This form of coding makes intelligent error recovery more difficult. SMS ranges from 1 to N, where N is a positive integer.

The complexity that SMS proposes to measure is that caused by the links among message calls inherent in strings of message calls. Instead of counting the message calls, this measure would better gauge complexity if the links were counted. Call this new measure strings of message links (SML). SML possesses a natural zero which is missing from the original measure. This new metric would fit the ratio scale but SMS does not.

3.5 Ratio measures

The measures in this section and the next are listed without comment. See Neal [16] for a full discussion. All of the measures in this section satisfy the requirements to be used on a ratio scale.

3.5.1 Lines of code (LOC) [15]: Number of statements (NOS) [15]: Number of semicolons in a class (SIZE1) [14]:

3.5.2 Message-passing coupling (MPC) [14]:

3.5.3 Number of local methods (NOM) [14]:

3.5.4 Number of public instance methods in a class (PIM) [15]:

3.5.5 Number of instance variables in a class (NIV) [15]:

3.5.6 Use of friend functions (FFU) [15]:

3.5.7 Number of abstract classes (NAC) [15]:

3.5.8 Number of times a class is reused (CRE) [15]:

3.5.9 Number of problem reports per class or contract (PRC) [15]:

3.6 Ratio' Measures

All of the measures in this section satisfy the requirements to be used on a ratio' scale.

3.6.1 Response for a class (RFC) [7]:

3.6.2 Number of instance methods in a class (NIM) [15]:

3.6.3 Number of methods added by a subclass (NMA) [15]:

3.6.4 Number of class methods in a class (NCM) [15]:

3.6.5 Weighted methods per class (WMC) [7]:

3.7 Averages

In order for an average to be validly used on an ordinal scale it is necessary that the parent metric be a measure that can be assigned to at least the interval scale. Arithmetic means of metrics measured on the interval scale are meaningful and will always result in an interval scale. Likewise, arithmetic means of metrics measured on the ratio scale are meaningful and will always result in a ratio scale [17].

3.7.1 Average method size (AMS) [15]:

AMS is really three metrics. AMS is the average method size as measured by dividing each of three previous metrics: number of message-sends (section 3.2.2), number of statements (section 3.5.1) and lines of code (section 3.5.1), by the total number of methods measured. It does not seem to matter whether one is measuring a system, a

sub-system, or a program. The metrics are calculated the same way regardless of the size of the project being measured.

The scale assigned to averages always depends on the scale assigned to the parent. Given that number of message sends is ambiguous, the average number of message sends also is ambiguous. The other two averages are measured on a ratio scale.

3.7.2 Average number of instance methods per class (AIM) [15]:

The parent (NIM) can be assigned to a ratio scale and the average (AIM) may also be assigned to a ratio scale.

3.7.3 Average number of instance variables per class (AIV) [15]:

The parent metric (NIV) can be assigned to a ratio scale and the average (AIV) may also be assigned to a ratio scale.

3.7.4 Average number of comment lines per method (CLM) [15]:

CLM is presented by its authors as one measure of documentation. CLM is defined as the number of explanatory comments divided by the number of methods. The number of explanatory comments is defined as the total number of comment lines minus comments due to coding convention or automatic insertion. CLM is a positive real number and ranges from 0 to N.

Explanatory comments are those lines placed in the code not as functional code but to explain the functionality of the method or function. Identifying comments, such as comments which merely report the date or the programmer's name or initials, should not be counted as explanatory comments.

When CLM is used as a measure of the documentation of a method, CLM can be used on a ratio scale.

3.7.5 Average number of commented methods (PCM) [15]:

PCM is presented by its authors as one measure of documentation. PCM is more accurately the proportion of methods in the class being measured which contain at least one explanatory comment line. PCM is the number of methods within a class which contain at least one explanatory comment line divided by the total number of methods in the class. This gives the developer an indicator of the completeness of a class' documentation where documentation is defined as the percentage of methods containing explanatory comment lines. PCM is a positive real number and ranges from 0 to 1.

PCM may not be assigned to the ordinal scale.

3.7.6 Average number of parameters per method (PPM) [15]:

PPM is presented by its authors as a measure of difficulty of use. PPM is defined as the total number of method parameters divided by the number of methods. More parameters place a higher burden on the user of the code. PPM is a positive real number and ranges from 0 to N.

Does adding (deleting) a method which requires zero parameters make it less (more) burdensome for the measurer to utilize the measured class? The other methods in the class which may require many parameters have not changed. If it can be accepted that adding (deleting) a parameterless method to (from) a class makes it less (more) burdensome, why not add many parameterless yet useless methods to every class and thus make them all less burdensome on the measurer? PPM cannot be assigned to the ordinal scale. It seems to us that the total number of parameters per class, or even the maximum number of parameters required for the use of any one method within a class would better define the burden on the measurer.

3.7.7 Percentage of function-oriented code (FOC) [15]:

FOC is presented by the authors as a measure of possible design problems. This metric is defined as the percentage of code written outside of objects, e.g., the C++ *main ()* routine and other non-member functions. The measurement of the code is left to us; the authors do not elaborate on a measurement procedure. The most obvious measure of code is LOC (see section 3.5.1). Although the name of the metric implies multiplication by 100, there is no apparent benefit in the conversion from proportion to percentage. Therefore, FOC as described in this section is a positive real number and ranges from zero to one.

LOC has been shown to be assignable to a ratio scale. Hence, $LOC_{nonobject\text{-}oriented\ code}$ can also be assignable to the ratio scale and the proportion FOC can also be assigned to a ratio scale. Functional (nonobject-oriented) code is required in some languages, e.g., the C++ *main ()* routine. However, functional code should be kept to a minimum. Therefore, FOC should be as small as possible.

4. Conclusion

Validity may depend upon the use to which a measure is to be applied. If one is looking for "red light" indicators that aspect of a software project may be wrong or that something may require extra attention to assure that nothing does go awry, then an ordinal scale measure may suffice. Finding outliers seems to be the state-of-the-art. However, to truly understand software and the development process, we must obtain a deeper understanding of software measurement.

At a minimum, measures should be validated before they are placed in use. Not every metric that has been proposed in the literature states the dimension that it purports to measure. Validation, however, requires that this dimension be identified. 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 philosopy 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] L. C. Briand, K. El Eman, and S. Morasca, "Theoretical and empirical validation of software product measures," *International Software Engineering Research Network Technical Report #ISERN-95-03*, 1995.

[4] L. C. Briand, K. El Eman, and S. Morasca, " On the application of Measurement Theory in Software Engineering," *International Software Engineering Research Network Technical Report #ISERN-95-04*, 1995.

[5] L. C. Briand, K. El Eman, and V. R. Basili, "Property-Based software engineering measurements," *IEEE Transactions on Software Engineering*, **22**(1), 1996.

[6] J-Y. Chen and J-F. Lu, "A new metric for object-oriented design," *Information and Software Technology*, 232-240, 1993.

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

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

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

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

[11] M. Hitz and B. Montazeri, "Chidamber and Kemerer's metric suite: A measurement theory perspective," *IEEE Transactions on Software Engineering*, **22**(4), April 1996.

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

[13] B. Kitchenham, S. Pfleeger, and N. Fenton, "Towards a Framework for Software Measurement Validation," *IEEE Transactions on Software Engineering*, **21**(12), December 1995.

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

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

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

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

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

[19] 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.

[20] Schneidewind, N. F., "Minimizing risk in apply metrics on multiple projects," *Proceedings: Third International Symposium on Software Reliability Engineering*, Oct. 7-10, 1992.

[21] S. S. Stevens, "On the theory of scales and measurement," *Science*, **103**, 677-680, 1946.

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

[23] H. Zuse, *Meßtheoretische Analyse von statischen Softwarekomplexitätsmaßen* (in German), unpublished dissertation, Technical University Berlin, 1985.

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

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

[26] H. 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.