A METRICS SUITE FOR OBJECT ORIENTED DESIGN

Shyam R. Chidamber
Chris F. Kemerer

December 1992

CISR WP No. 249
Sloan WP No. 3524-93

Center for Information Systems Research
Massachusetts Institute of Technology
Sloan School of Management
77 Massachusetts Avenue
Cambridge, Massachusetts, 02139

A METRICS SUITE FOR OBJECT ORIENTED DESIGN

Shyam R. Chidamber
Chris F. Kemerer

December 1992

CISR WP No. 249
Sloan WP No. 3524-93

Center for Information Systems Research
Sloan School of Management
Massachusetts Institute of Technology

A METRICS SUITE FOR OBJECT ORIENTED DESIGN

Shyam R. Chidamber
Chris F. Kemerer

Abstract

Given the central role that software development plays in the delivery and application of information technology, managers are increasingly focusing on process improvement in the software development area. This demand has spurred the provision of a number of new and/or improved approaches to software development, with perhaps the most prominent being object-orientation (OO). In addition, the focus on process improvement has increased the demand for software measures, or metrics with which to manage the process. The need for such metrics is particularly acute when an organization is adopting a new technology for which established practices have yet to be developed. This research addresses these needs through the development and implementation of a new suite of metrics for OO design. Metrics developed in previous research, while contributing to the field's understanding of software development processes, have generally been subject to serious criticisms, including the lack of a theoretical base. Following Wand and Weber, the theoretical base chosen for the metrics was the ontology of Bunge. Six design metrics are developed, and then analytically evaluated against Weyuker's proposed set of measurement principles. An automated data collection tool was then developed and implemented to collect an empirical sample of these metrics at two field sites in order to demonstrate their feasibility and suggest ways in which managers may use these metrics for process improvement.

TABLE OF CONTENTS

"A Metrics Suite For Object Oriented Design"

Introduction

It has been widely recognized that an important component of process improvement is the ability to measure the process. Given the central role that software development plays in the delivery and application of information technology, managers are increasingly focusing on process improvement in the software development area. This emphasis has had two effects. The first is that this demand has spurred the provision of a number of new and/or improved approaches to software development, with perhaps the most prominent being object-orientation (OO). Second, the focus on process improvement has increased the demand for software measures, or metrics with which to manage the process. The need for such metrics is particularly acute when an organization is adopting a new technology for which established practices have yet to be developed.

This research addresses these needs through the development and implementation of a new suite of metrics for OO design. Previous research on software metrics, while contributing to the field's understanding of software development processes, have generally been subject to one or more types of criticisms. These include: lacking a theoretical basis [26], lacking in desirable measurement properties [31], being insufficiently generalized or too implementation technology dependent [30], and being too labor-intensive to collect [15].

Following Wand and Weber, the theoretical base chosen for the OO design metrics was the ontology of Bunge [4, 5, 28]. Six design metrics were developed, and analytically evaluated against a previously proposed set of measurement principles. An automated data collection tool was then developed and implemented to collect an empirical sample of these metrics at two field sites in order to demonstrate their feasibility and to suggest ways in which managers may use these metrics for process improvement.

The key contributions of this paper are the development and empirical validation of a set of theoretically-grounded metrics of OO design. The rest of this paper is organized as follows. The next section presents a brief summary of the research problem followed by a section describing the theory underlying the approach taken. Then Weyuker's list of software metric evaluation criteria are presented, along with a brief description of the empirical data collection sites. The Results section presents the metrics, their analytical evaluation, the empirical data and a managerial interpretation of the data for each metric. Some concluding remarks are presented in the final section.

Research Problem

There are two general types of criticisms that can be applied to current software metrics. The first category are those theoretical criticisms that are leveled at conventional software metrics as they are applied to traditional, non-OO software design and development. Kearney, *et al.* criticized software complexity metrics as being without solid theoretical bases and lacking appropriate properties [14]. Vessey and Weber also commented on the general lack of theoretical rigor in the structured programming literature [26]. Both Prather and Weyuker proposed that traditional software complexity metrics do not possess appropriate mathematical properties, and consequently fail to display what might be termed normal predictable behavior [23, 31]. This suggests that software metrics need to be constructed with a stronger degree of theoretical and mathematical rigor.

The second category of criticisms is more specific to OO design and development. The OO approach centers around modeling the real world in terms of its objects, which is in contrast to older, more traditional approaches that emphasize a function-oriented view that separate data and procedures. Several theoretical discussions have speculated that OO approaches may even induce different problem-solving behavior and cognitive processing in the design process, e.g. [3, 16]. Given the fundamentally different notions inherent in these two views, it is not surprising to find that software metrics developed with traditional methods in mind do not readily lend themselves to OO notions such as classes, inheritance, encapsulation and message passing [12]. Therefore, given that current software metrics are subject to some general criticism and are easily seen as not supporting key OO concepts, it seems appropriate to develop a set, or suite of new metrics especially designed to measure unique aspects of the OO approach.

The shortcomings of existing metrics and the need for new metrics especially designed for OO have been suggested by a number of authors. Some initial proposals are set out by Morris, although they are not tested [20]. Lieberherr and his colleagues present a more formal attempt at defining the rules of correct object oriented programming style, building on concepts of coupling and cohesion that are used in traditional programming [18]. Moreau and Dominick suggest three metrics for OO graphical information systems, but do not provide formal, testable definitions [19]. Pfleeger also suggests the need for new measures, and uses simple counts of objects and methods to develop and test a cost estimation model for OO development [22]. Other authors, such as Chidamber and Kemerer, Sheetz, *et al.*, and Whitmire propose metrics, but do not offer any empirical data [8, 25, 32]. Despite the growing research interest in this area, no empirical metrics data from commercial object oriented applications has been published in the archival literature.

Given the extant software metrics literature, this paper has a three fold agenda: 1) to propose metrics that are constructed with a firm basis in theoretical concepts in measurement and the ontology of objects, and which incorporate the experiences of professional software developers; 2) evaluate the proposed metrics against established criteria for validity, and 3) present empirical data from commercial projects to demonstrate the feasibility of collecting these metrics and suggest ways in which these metrics may be used.

Theory Base for OOD Metrics

While there are many object oriented design (OOD) methodologies, one that reflects the essential features of OOD is presented in Booch (1991).[1] He outlines four major steps involved in the object-oriented design process:

1) *Identification of Classes and Objects* In this step, key abstractions in the problem space are identified and labeled as potential classes and objects.

2) *Identify the Semantics of Classes and Objects* In this step, the meaning of the classes and objects identified in the previous step is established, this includes definition of the life-cycles of each object from creation to destruction.

3) *Identify Relationships between classes (and objects)* In this step, class and object interactions, such as patterns of inheritance among classes and patterns of visibility among objects and classes (what classes and objects should be able to "see" each other) are identified.

4) *Implementation of Classes and Objects* In this step, detailed internal views are constructed, including definitions of methods and their various behaviors.

The metrics outlined in this paper will map to the first three (non-implementation) stages outlined by Booch, since they parsimoniously capture the essential features of OOD. Since the proposed metrics are specifically aimed at addressing the design phase and do not depend on code-based constructs, the potential benefits of this information can be substantially greater than metrics aimed at later phases in the life-cycle of an application. In addition, implementation-independent metrics will be applicable to a larger set of users, especially in the early stages of industry's adoption of OO before dominant implementation standards emerge.

Measurement theory base

An object oriented design can be conceptualized as a *relational system*, which is defined by Roberts as an ordered tuple consisting of a set of *elements*, a set of *relations* and a set of *binary operations*.

---

[1] For a comparison and critique of six different OO analysis and design methodologies see [10].

[24]. More specifically, an object oriented design, **D**, is conceptualized as a relational system consisting of object elements (classes and objects), empirical relations and binary operations that can be performed on the object elements. Notationally:

$\mathbf{D} \equiv (A, R_1 \ldots R_n, O_1 \ldots O_m)$

where

A is a set of object elements

$R_1 \ldots R_n$ are empirical relations on object elements A (e.g., bigger than, smaller than, etc.)

$O_1 \ldots O_m$ are binary operations (e.g., concatenation)

A useful way to understand empirical relations on a set of object elements is to consider the measurement of complexity. A designer generally has some intuitive ideas about the complexity of different object elements, as to which element is more complex than another or which ones are equally complex. For example, a designer intuitively understands that a class that has many methods is generally more complex, *ceteris paribus*, than one that has only a few methods. This intuitive idea is defined as a *viewpoint*. The notion of a viewpoint was originally introduced to describe evaluation measures for information retrieval systems and is applied here to capture designer views [7]. More recently, Fenton states that viewpoints characterize intuitive understanding and that viewpoints must be the logical starting point for the definition of metrics [9]. An empirical relation is identical to a viewpoint, and the two terms are distinguished here only for the sake of consistency with the measurement theory literature.

A viewpoint is a binary relation $\gtrsim$ defined on a set **P** (the set of all possible designs). For P, P', $P'' \in \mathbf{P}$, the following two axioms must hold:

$P \gtrsim P'$ or $P' \gtrsim P$ (*completeness*: P is more complex than P' or P' is more complex than P)

$P \gtrsim P'$, $P' \gtrsim P'' \Rightarrow P \gtrsim P''$ (*transitivity*: if P is more complex than P' and P' is more complex than P'', then P is more complex than P'')

i.e., a viewpoint must be of weak order [24].

To be able to measure something about a object design, the *empirical relational system* as defined above needs to be transformed to a *formal relational system*. Therefore, let a formal relational system **F** be defined as follows:

$\mathbf{F} \equiv (C, S_1 \ldots S_n, B_1 \ldots B_m)$

C is a set of elements (e.g., real numbers)

$S_1 ... S_n$ are *formal* relations on C (e.g., >, <, =)

$B_1 ... B_m$ are *binary* operations (e.g., +,-,*)

This required transformation is accomplished by a metric μ which maps an empirical system **D** to a formal system **F**. For every element $a \in D$, $\mu(a) \in F$. The example below involving a set of school children illustrates the mapping between an empirical relational system and a formal relational system [13]:

Empirical Relational System	Formal Relational System
Heights of school children	Real Numbers
Relations: Equal or taller than Child P is taller than Child P'	Relations: = or > 36 inch child > 30 inch child
Binary Operations: Concatenation: two children standing atop one another	Binary Operations: +: add the real numbers associated with the two children

The empirical relation "Child P is taller than Child P'" in the above example is transformed to the formal relation "36 inch child > 30 inch child", enabling the explicit understanding of the heights of school children.. The assumption in the argument for transformation of empirical relational systems to a formal empirical systems is that the "intelligence barrier" to understanding of the former is circumvented due to the transformation [17]. In the example of the school children the intelligence barrier is small, but the principle is that numerical representations produced by the transformation to formal systems help in understanding the empirical system better. While the exercise of transformation may seem laborious for the simple example above, it can prove to be valuable in understanding complexity of software where the complexity relationships are not visible [13]. Design of object oriented systems is a difficult undertaking in part due to the newness of the technology, and therefore formal metrics are proposed to aid designers and managers in managing complexity in OOD.

Definitions

The ontological principles proposed by Bunge in his "Treatise on Basic Philosophy" forms the basis of the concept of objects (1977). While Bunge did not provide specific ontological definitions for object oriented concepts, several recent researchers have employed his generalized concepts to the object oriented domain [27, 28]. Bunge's ontology has considerable appeal for OO researchers since it deals with the meaning and definition of representations of the world, which are precisely the goals of the object oriented approach [21]. Consistent with this ontology, objects

are defined independent of implementation considerations and encompass the notions of encapsulation, independence and inheritance. According to this ontology, the world is viewed as composed of things, referred to as *substantial individuals*, and concepts. The key notion is that substantial individuals possess *properties*. A property is a feature that a substantial individual possesses inherently. An observer can assign features to an individual, but these are attributes and not properties. All substantial individuals possess a finite set of properties: "there are no bare individuals except in our imagination" [4, p. 26]

Some of the attributes of an individual will reflect its properties. Indeed, properties are recognized only through attributes. A known property must have at least one attribute representing it. Properties do not exist on their own, but are "attached" to individuals. On the other hand, individuals are not simply bundles of properties. A substantial individual and its properties collectively constitute an *object* [27, 28].

An object can be represented in the following manner:

$X = \langle x, p(x) \rangle$ where x is the substantial individual and p(x) is the finite collection of its properties.

x can be considered to be the token or name by which the individual is represented in a system. In object oriented terminology, the instance variables[2] together with its methods[3] are the properties of the object [1]

Using these representations of objects, previous research has defined concepts like scope and similarity that are relevant to object oriented systems [4, 27]. Following this tradition, this paper defines two important software design concepts for objects, coupling and cohesion. Coupling refers to the degree of interdependence between parts of a design, while cohesion refers to the internal consistency within parts of the design. All other things being equal, good software design practice calls for minimizing coupling and maximizing cohesiveness.

Coupling. In ontological terms, two things are coupled if and only if at least one of them "acts upon" the other. X is said to act upon Y if the history of Y is affected by X, where history is defined as the chronologically ordered states that a thing traverses in time [4].

Let $X = \langle x, p(x) \rangle$ and $Y = \langle y, p(y) \rangle$ be two objects.

$$p(x) = \{ M_X \} \cup \{ I_X \}$$

---

[2] An instance variable is a repository for part of the state of the object.

[3] A method is an operation on an object that is defined as part of the declaration of the class.

$$p(y) = \{ M_Y \} \cup \{ I_Y \}$$

where $\{ M_i \}$ is the set of methods and $\{ I_i \}$ is the set of instance variables of object *i*.

Using the above definition of coupling, any action by $\{M_X\}$ on $\{M_Y\}$ or $\{I_Y\}$ constitutes coupling, as does any action by $\{M_Y\}$ on $\{M_X\}$ or $\{I_X\}$. When M_X calls M_Y, M_X alters the history of the usage of M_Y, similarly when M_X uses I_Y, it alters the access and usage history of I_Y. Therefore, any evidence of a method of one object using methods or instance variables of another object constitutes coupling.

Cohesion. Bunge defines *similarity* $\sigma()$ of two things to be the intersection of the sets of properties of the two things:

$$\sigma(X,Y) = p(x) \cap p(y)$$

Following this general principle of defining similarity in terms of sets, the degree of similarity of the methods within the object can be defined to be the intersection of the sets of instance variables that are used by the methods. This is an extension of Bunge's definition of similarity to similarity of methods. It should be clearly understood that instance variables are not properties of methods, but it is consistent with the notion that methods of an object are intimately connected to its instance variables.

$$\sigma(M_1,M_2...M_n) = \{ I_1 \} \cap \{ I_2 \} \cap \{ I_3 \} ... \{ I_n \}$$

where $\sigma()$ = degree of similarity of methods M_i and

$\{ I_i \}$ = set of instance variables used by method M_i.

The degree of similarity of methods relates both to the conventional notion of *cohesion* in software engineering, (i.e., keeping related things together) as well as encapsulation of objects, that is, the bundling of methods and instance variables in an object. The *degree of similarity* of methods can be defined to be a major aspect of object cohesiveness. The higher the degree of similarity of methods, the greater the cohesiveness of the methods and the higher the degree of encapsulation of the object.

Complexity of an object. Bunge defines complexity of an individual to be the "numerosity of its composition", implying that a complex individual has a large number of properties [4, p. 43]. Using this definition as a base, the complexity of an object can be defined to be the cardinality of its set of properties.

Complexity of $\langle x, p(x) \rangle = | p(x) |$, where $| p(x) |$ is the cardinality of $p(x)$.

Scope of Properties. The scope of a property P in J (a set of objects) is the subset G (P; J) of objects possessing the property [27].

$G(P; J) = \{ x \mid x \in J \text{ and } P \in p(x) \}$, where p(x) is the set of all properties of $x \in J$.

Class $C(p; J) = \cap_{\text{all } P} \{ G(P) \mid P \in g() \}$ where g() is a given set of properties.[4]

A class P with respect to a property set *g* is the set of all objects possessing all properties in *g*. In simple terms, a class is a set of objects that have common properties. The inheritance hierarchy, a directed acyclic graph will be described as a tree structure with classes as nodes, leaves and a root. In any design application, there can be many possible hierarchies of classes. Design choices on the hierarchy employed to represent the application are essentially choices about restricting or expanding the scope of properties of the objects in the application. Two design decisions which relate to the inheritance hierarchy can be defined. They are *depth of inheritance* of a class of objects and the *number of children* of the class.

Depth of Inheritance = height of the class in the inheritance tree

The height of a node of a tree refers to the length of the maximal path from the node to the root of the tree.

Number of Children = Number of immediate descendants of the class

Both these concepts relate to the notion of scope of properties, i.e., how far does the influence of a property extend? Depth of inheritance indicates the extent to which the class is influenced by the properties of its ancestors and number of children indicates the potential impact on descendants. The depth of inheritance and number of children collectively indicate the genealogy of a class.

Methods as measures of communication. In the object oriented approach, objects can communicate only through message passing.[5] A message can cause an object to "behave" in a particular manner by invoking a particular method. Methods can be viewed as definitions of responses to possible messages [1]. It is reasonable, therefore, to define a *response set* for a class of objects in the following manner:

Response set of a class of objects = {set of all methods that can be invoked in response to a message to an object of the class}

---

[4] In strict ontological terms, $\cap_{\text{all } P} \{ G(P) \mid P \in g() \}$ is the definition of a "kind", but is used here to define the concept of a class.

[5] While objects can communicate through more complex mechanisms like bulletin boards, the majority of OO designers employ message passing as the only mechanism for communicating between objects.

Note that this set will include methods outside the class as well, since methods within the class may call methods from other classes. The response set will be finite since the properties of a class are finite and there are a finite number of classes in a design.

Metrics Evaluation Criteria

Several researchers have recommended properties that software metrics should possess to increase their usefulness. For example, Basili and Reiter suggest that metrics should be sensitive to externally observable differences in the development environment and must also correspond to intuitive notions about the characteristic differences between the software artifacts being measured [2]. The majority of recommended properties are qualitative in nature and consequently, most proposals for metrics have tended to be informal in their evaluation of metrics.

More recently however, Weyuker developed a formal list of desiderata for software metrics and has evaluated a number of existing software metrics using these properties [31].[6] Two of Weyuker's nine properties are rudimentary in nature; one requires that there are only a finite number of cases having the same complexity metric and the other is that when the name of the measured object changes, the metric should remain unchanged. Both these properties are typically met by all metrics and will not be considered in this paper.[7]

Another property is that permutation of elements within the object being measured must change the metric value. The intent is to ensure that metric values change, for example, in case the nesting of if-then-else blocks change due to permutation of program statements. If this property were to be satisfied by an OOD complexity metric, values for a class will change when the order of declaration of methods or instance variables is changed, which seems at odds with OO design. And, in fact, Chemiavsky and Smith specifically suggest that this property is not appropriate for OOD metrics because "the rationales used may be applicable only to traditional programming" [6, p. 638]. Since this property seems to be based strictly in the traditional paradigm, it is not considered further. The remaining six properties are repeated below.[8]

---

[6]Weyuker's properties are not without criticism (e.g., [11, 6, 9, 33], and some of these critiques will be discussed below. However, her list remains the currently most widely used list of such criteria, and therefore is the list chosen for this evaluation.

[7]Since class libraries have a finite number of classes, no metric value can be repeated infinitely. And, unless the metric depends on the name of an object, the second property is also easily met.

[8]Readers familiar with Weyuker's work should note that the exclusion of these three properties makes the property numbers used here no longer consistent with the original property numbers. It should also be noted that Weyuker's definitions have been modified where necessary to use the term object rather than program.

Property 1: Non-coarseness

Given an object P and a metric μ another object Q can always be found such that: $\mu(P) \neq \mu(Q)$. This implies that not every object can have the same value for a metric, otherwise it has lost its value as a measurement.

Property 2: Non-uniqueness (notion of equivalence)

There can exist distinct objects P and Q such that $\mu(P) = \mu(Q)$. This implies that two objects can have the same metric value, i.e., the two objects are equally complex.

Property 3: Design Implementation not function is important

Given two object designs, P and Q, which provide the same functionality, does not imply that $\mu(P) = \mu(Q)$. The intuition behind Property 3 is that even though two object designs perform the same function, the details of the design matter in determining the object design's metric.

Property 4: Monotonicity

For all objects P and Q, the following must hold: $\mu(P) \leq \mu(P+Q)$ and $\mu(Q) \leq \mu(P+Q)$ where P + Q implies concatenation of P and Q.[9] This implies that the metric for the combination of two objects can never be less than the metric for either of the component objects.

Property 5: Non-equivalence of interaction

$\exists$ P, $\exists$ Q, $\exists$ R, such that

$\mu(P) = \mu(Q)$ does not imply that $\mu(P+R) = \mu(Q+R)$.

This suggests that interaction between P and R can be different than interaction between Q and R resulting in different complexity values for P+R and Q+R.

Property 6: Interaction increases complexity

$\exists$ P and $\exists$ Q such that:

$\mu(P) + \mu(Q) \leq \mu(P+Q)$

---

[9]Designers' empirical operations of combining two objects in order to achieve better representation are formally denoted here as concatenation and shown with a + sign. Concatenation results in a single joint state space of instance variables and methods instead of two separate state spaces, the only definite result of concatenation of two objects is the elimination of all prior messages between the two component objects.

The principle behind this property is that when two objects are combined, the interaction between objects will tend to increase the complexity metric value.

Assumptions. Some basic assumptions made regarding the distribution of methods and instance variables in the discussions for each of the metric properties.

Assumption 1:

Let X_i = The number of methods in a given class *i*.
Y_i = The number of methods called from a given method *i*.
Z_i = The number of instance variables used by a method *i*.
C_i = The number of couplings between a given class of objects *i* and all other classes.

X_i, Y_i, Z_i, C_i are discrete random variables each characterized by some general distribution function. Further, all the X_is are independent and identically distributed (i.i.d.). The same is true for all the Y_is, Z_is and C_is.

Assumption 2: In general, two classes can have a finite number of "identical" methods in the sense that a combination of the two classes into one class would result in one class's version of the identical methods becoming redundant.

Assumption 3: The inheritance tree is "full", i.e., there is a root, several intermediate nodes which have siblings, and leaves. The tree is not necessarily balanced, i.e., each node does not necessarily have the same number of children.

Empirical Data Collection

As defined earlier, a design encompasses the implicit ideas designers have about complexity. These viewpoints are the empirical relations R_1, R_2, ... R_n in the formal definition of the design **D**. The viewpoints that were used in constructing the metrics presented in this paper were gathered from extensive collaboration with a highly experienced team of software engineers from a software development organization. This organization has used OOD in more than four large projects over the past five years. Though the primary development language for all projects at this site was C++, the research aim was to propose metrics that were language independent. As a test of this, data were collected at two sites which used different languages.

The metrics proposed in this paper were collected using automated tools developed for this research at two different organizations which will be referred to here as Site A and Site B. Site A is a software company that uses OOD in their development work and has a collection of different C++ class libraries. Metrics data from 634 classes from two class libraries that are used in the

design of graphical user interfaces (GUI) were collected. These two class libraries are written in C++ and are typically used at Site A in conjunction with other C++ libraries and traditional C-language programs in the development of software sold to UNIX workstation users.

Site B is a semiconductor manufacturer and uses the Smalltalk programming language for developing flexible machine control and manufacturing systems. Metrics were collected on the class libraries used in the implementation of a computer aided manufacturing system for the production of VLSI circuits. Over 30 engineers worked on this application, after extensive training and experience with object orientation and the Smalltalk environment. Metrics data from 1459 classes from Site B were collected.

Results

Metric 1: Weighted Methods Per Class (WMC)

Definition. Consider a Class C_1, with methods $M_1,... M_n$. Let $c_1,... c_n$ be the complexity[10] of the methods. Then :

$$WMC = \sum_{i=1}^{n} c_i$$

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

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

Viewpoints.

• The number of methods and the complexity of methods involved is an indicator of how much time and effort is required to develop and maintain the class.

• The larger the number of methods in a class the greater the potential impact on children, since children will inherit all the methods defined in the class.

---

[10] Complexity is deliberately not defined more specifically here in order to allow for the most general application of this metric. It can be argued that developers approach the task of writing a method as they would a traditional program, and therefore some traditional static complexity metric, such as McCabe's Cyclomatic number, may be appropriate. This is left as an implementation decision, as the general applicability of any existing static complexity metric has not been generally agreed upon. The general nature of the WMC metric is presented as a strength, not a weakness as has been suggested elsewhere [12].

• Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse.

Analytical Evaluation of Weighted Methods Per Class (WMC)

From assumption 1, the number of methods in class P and another class Q are i.i.d., this implies that there is a finite probability that $\exists$ Q such that $\mu(P) \neq \mu(Q)$, therefore property 1 is satisfied. Similarly, there is a finite probability that $\exists$ a R such that $\mu(P) = \mu(R)$. Therefore property 2 is satisfied. The function of the object does not define the number of methods in a class. The choice the number of methods is a design implementation decision and independent of the functionality of the class, therefore Property 3 is satisfied. Let $\mu(P) = n_P$ and $\mu(Q) = n_Q$, then $\mu(P+Q) = n_P + n_Q$. Clearly, $\mu(P+Q) \geq \mu(P)$ and $\mu(P+Q) \geq \mu(Q)$, thereby satisfying Property 4. Now, let $\mu(P) = n$, $\mu(Q) = n$, and $\exists$ an object R such that it has a number of methods ∂ in common with Q (as per assumption 2) and β methods in common with P. Let $\mu(R) = r$.

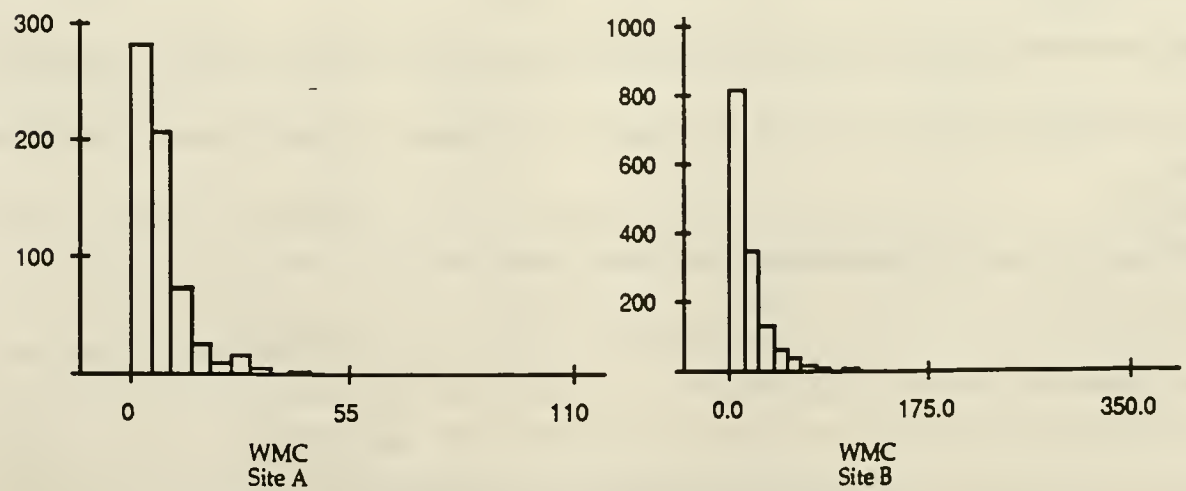
$$\mu(P+R) = n + r - \beta$$

$$\mu(Q+R) = n + r - \partial$$

therefore $\mu(P+Q) \neq \mu(Q+R)$ and Property 5 is satisfied. For any two objects P and Q, $\mu(P+Q) = n_P + n_Q - \partial$, where n_P is the number of methods in P, n_Q is number of methods in Q and P and Q have ∂ methods in common. Clearly, $n_P + n_Q - \partial \leq n_P + n_Q$ for all P and Q i.e., $\mu(P+Q) \leq \mu(P) + \mu(Q)$ for all P and Q. Therefore, Property 6 is not satisfied.

Empirical Data

The histograms and summary statistics from both sites are shown below:



Histograms for the WMC metric

Site	Metric	Mean	Median	Max	Min	StdDev	Skewness
A	WMC	7.69	5	106	0	10.15	4.38
B	WMC	16.39	10	346	0	21.61	4.96

Interpretation of Data. The most interesting aspect of the data is the similarity in the nature of the distribution of the metric values at Site A and B, despite differences in i) the nature of the application ii) the people involved in their design and iii) the languages (C++ and Smalltalk) used. This seems to suggest that most classes tend to have a small number of methods (0 to 10), while a few outliers declare a large number of them. This further suggests that a small number of classes may be responsible for a large number of the methods that execute in an application, and that if testing effort were concentrated on these outlier classes, a bulk of the dynamic behavior of these object oriented systems can be checked for customer acceptance.

Metric 2: Depth of Inheritance Tree (DIT)

Definition. Depth of inheritance of the class is the DIT metric for the class.

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

Viewpoints.

• The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior.

• Deeper trees constitute greater design complexity, since more methods and classes are involved.

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

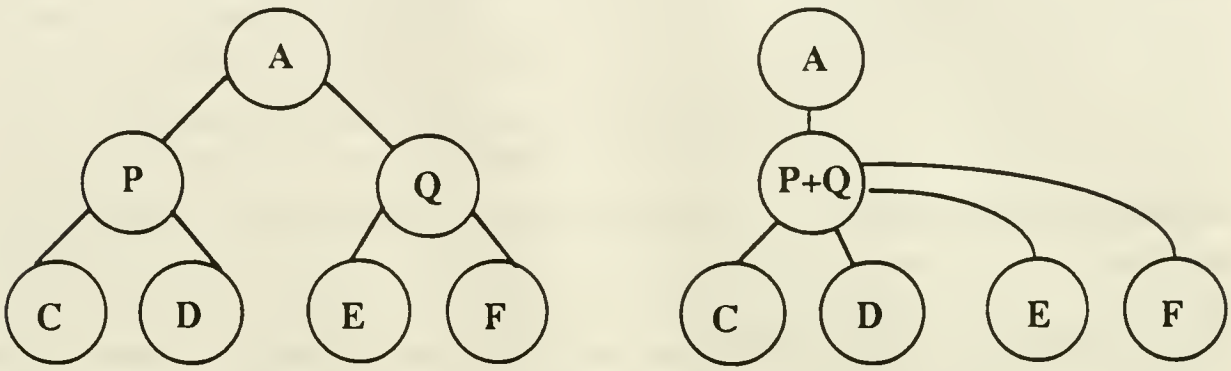
Analytical Evaluation of Depth of Inheritance Tree (DIT)

Per assumption 3, every tree has a root and leaves. The depth of inheritance of a leaf is always greater than that of the root. Therefore, property 1 is satisfied. Also, since every tree has at least some nodes with siblings (per assumption 3), there will always exist at least two objects with the same depth of inheritance, i.e., property 2 is satisfied. Design of an object involves choosing what

properties the object must inherit in order to perform its function. In other words, depth of inheritance is design implementation dependent, and Property 3 is satisfied.

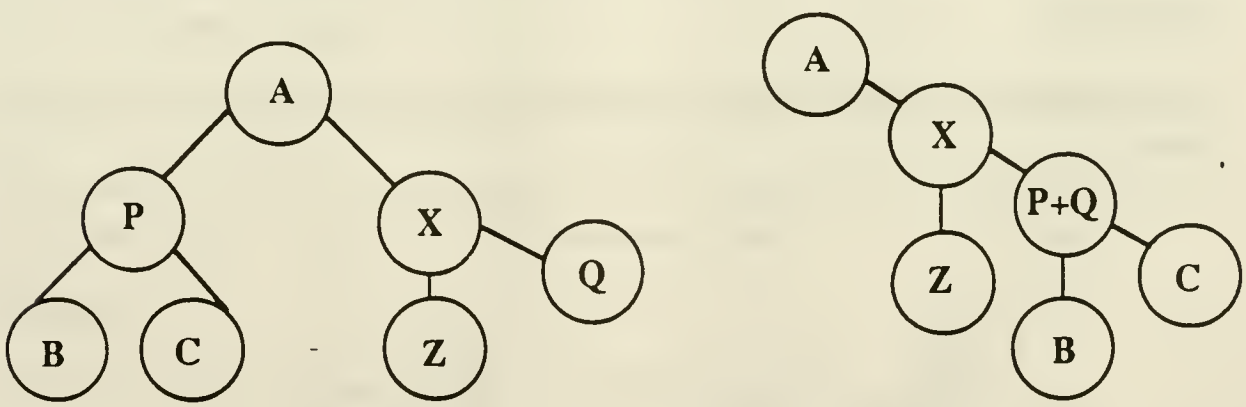
When any two objects P and Q are combined, there are three possible cases (in all cases, it is assumed that there is no multiple inheritance[11]) : i) P and Q are siblings ii) P and Q are neither children nor siblings of each other and iii) one is the child of the other.

Case i) P and Q are siblings



In this case, $\mu(P) = \mu(Q) = n$ and $\mu(P+Q) = n$, i.e. Property 4 is satisfied.

Case ii) P and Q are neither children nor siblings of each other.

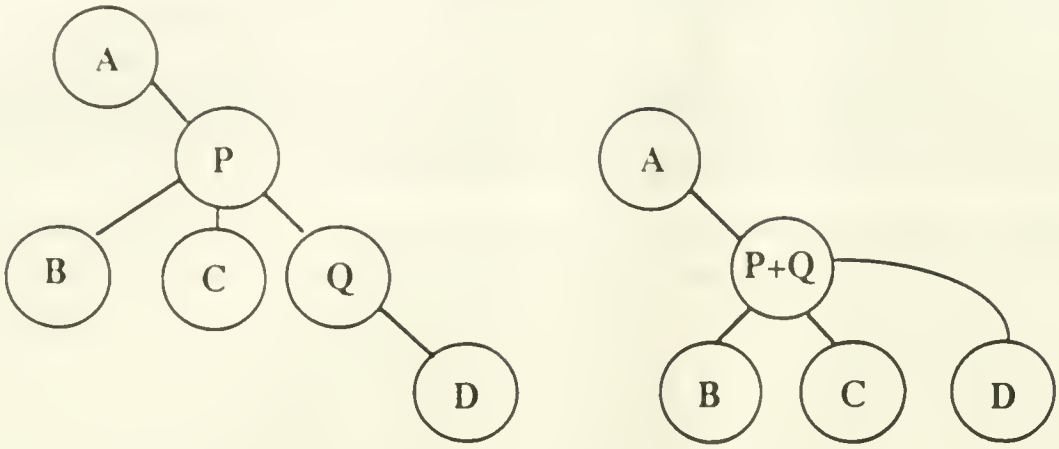


If P+Q moves to P's location in the tree, Q cannot inherit methods from X, however if P+Q moves to Q's location, P maintains its inheritance. Therefore, P+Q will be in Q's old location. In this case, $\mu(P) = x$, $\mu(Q) = y$ and $y > x$. $\mu(P+Q) = y$, i.e., $\mu(P+Q) > \mu(P)$ and $\mu(P+Q) = \mu(Q)$ and Property 4 is satisfied.

---

[11] It can be shown that Property 5 is satisfied even with multiple inheritance. $\mu(P)$ and $\mu(Q)$ will be n_P and n_Q respectively and $\mu(P+Q)$ will be equal to $Max(n_P + n_Q)$. Consequently $\mu(P+Q)$ will always be greater than or equal to $\mu(P)$ and $\mu(Q)$.

iii) when one is a child of the other:

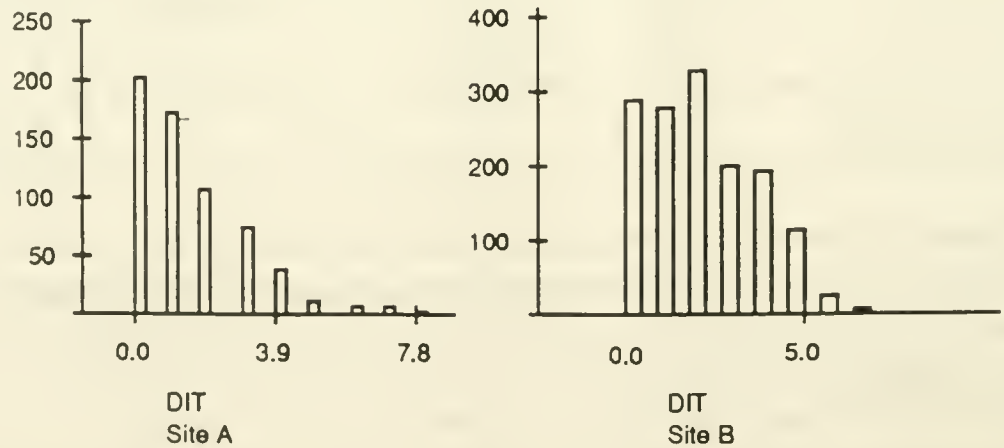


In this case, $\mu(P) = n$, $\mu(Q) = n + 1$, but $\mu(P+Q) = n$, i.e. $\mu(P+Q) < \mu(Q)$. Property 4 is not satisfied.

For all three cases, let P and Q' be siblings, i.e. $\mu(P) = \mu(Q') = n$, and let R be a child of P. Then $\mu(P+R) = n$ and $\mu(Q'+R) = n + 1$. i.e., $\mu(P+R)$ is not equal to $\mu(Q'+R)$. Property 5 is satisfied. For any two objects P and Q, $\mu(P+Q) = \mu(P)$ or $= \mu(Q)$. Therefore, $\mu(P+Q) \leq \mu(P) + \mu(Q)$, i.e., Property 6 is not satisfied.

Empirical Data

The histograms and summary statistics from both sites are shown below (all metric values are integers):



Histograms for the DIT metric

Site	Metric	Mean	Median	Max	Min	StdDev	Skewness
A	DIT	1.54	1	8	0	1.63	1.32
B	DIT	3.16	3	10	0	1.70	0.53

Interpretation of Data. Both Site A and B libraries have a low median value for the DIT metric. This suggests that most classes in an application tend to be close to the root in the inheritance hierarchy. By observing the DIT metric for classes in an application, a senior designer or manager can determine whether the design is "top heavy" (too many classes near the root) or "bottom heavy" (many classes are near the bottom of the hierarchy). At both Site A and Site B, the library appears to be top heavy, suggesting that designers may not be taking advantage of reuse of methods through inheritance.[12] Another interesting aspect is that the maximum value of DIT is rather small (10 or less). One possible explanation is that designers tend to keep the number of levels of abstraction to a small manageable number to facilitate comprehensibility of the overall architecture of the system. Designers may be forsaking reusability through inheritance for simplicity of understanding. This also illustrates one of the advantages of gathering metrics of design complexity in that a clearer picture of the conceptualization of software systems begins to emerge with special attention focused on design tradeoffs.

Metric 3: Number of children (NOC)

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

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

Viewpoints.

• Greater the number of children, greater the reuse, since inheritance promotes reuse.

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

[12] Of course, such occurences may also be a function of the application. It is interesting to note, however, that this phenomenon appears to be present in both data sets, which represent relatively different applications and implementation environments.

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

Analytical Evaluation of Number Of Children (NOC)

Let P and R be leaves, $\mu(P) = \mu(R) = 0$, let Q be the root $\mu(Q) > 0$. $\mu(P) \neq \mu(Q)$ therefore property 1 is satisfied. Since $\mu(R) = \mu(P)$, Property 2 is also satisfied. Design of a class involves decisions on the scope of the methods declared within the class, i.e., the sub-classing for the class. The number of sub-classes is therefore dependent upon the design implementation of the class. Therefore, Property 3 is satisfied.

Let P and Q be two classes with n_P and n_Q sub-classes respectively (i.e., $\mu(P) = n_P$ and $\mu(Q) = n_Q$). Combining P and Q, will yield a single class with $n_P + n_Q - \partial$ sub-classes, where ∂ is the number of children P and Q have in common. Clearly, ∂ is 0 if either n_P or n_Q is 0. Now, $n_P + n_Q - \partial \geq n_P$ and $n_P + n_Q - \partial \geq n_Q$. This can be written as: $\mu(P+Q) \geq \mu(P)$ and $\mu(P+Q) \geq \mu(Q)$ for all P and all Q. Therefore, Property 4 is satisfied. Let P and Q each have *n* children and R be a child of P which has *r* children. $\mu(P) = n = \mu(Q)$. The class obtained by combining P and R will have (n-1) + r children, whereas a class obtained by combining Q and R will have $n + r$ children, which means that $\mu(P+R) \neq \mu(Q+R)$. Therefore Property 5 is satisfied. Given any two classes P and Q with n_P and n_Q children respectively, the following relationship holds:

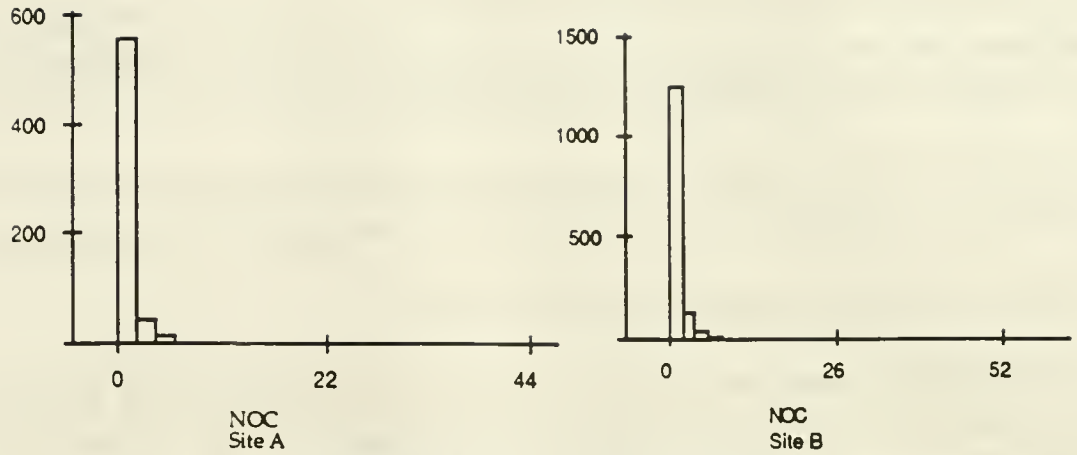
$$\mu(P) = n_P \text{ and } \mu(Q) = n_Q$$

$$\mu(P+Q) = n_P + n_Q - \partial$$

where ∂ is the number of common children. Therefore, $\mu(P+Q) \leq \mu(P) + \mu(Q)$ for all P and Q. Property 6 is not satisfied.

Empirical Data

The histograms and summary statistics from both sites are shown below:



Histograms for the NOC metric

Site	Metric	Mean	Median	Max	Min	StdDev	Skewness
A	NOC	0.67	0	42	0	2.61	9.11
B	NOC	0.66	0	50	0	2.41	12.55

Interpretation of Data. Like the WMC metric, an interesting aspect of the NOC data is the similarity in the nature of the distribution of the metric values at Site A and B. This seems to suggest that classes in general have few immediate children, only a very small number of outliers have many immediate sub-classes. This further suggests that designers may not be using inheritance of methods as a basis for designing classes as the data show that at least 50% of the classes in a library have no children. Considering the large sample sizes at both sites and their remarkable similarity, both the DIT and NOC data seem to strongly suggest that reuse through inheritance may not be being fully adopted in the design of class libraries, at least at these two sites.

Metric 4: Coupling between objects (CBO)

Definition. CBO for a class is a count of the number of couples with other classes.

Theoretical basis. CBO relates to the notion that an object is coupled to another object if one of them acts on the other, i.e., methods of one use methods or instance variables of another.

Viewpoints.

• Excessive coupling between objects is detrimental to modular design and prevents reuse. The more independent an object is, the easier it is to reuse it in another application.

• In order to improve modularity and promote encapsulation, inter-object couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design and therefore maintenance is more difficult.

• A measure of coupling is useful to determine how complex the testing of various parts of a design are likely to be. The higher the inter-object coupling, the more rigorous the testing needs to be.

Analytical Evaluation of Coupling Between Objects (CBO)

As per assumption 1, there exist objects P, Q and R such that $\mu(P) \neq \mu(Q)$ and $\mu(P) = \mu(R)$ thereby satisfying properties 1 and 2. Inter-object coupling occurs when methods of one object use methods or instance variables of another object, i.e., coupling depends on the manner in which methods are designed and not on the functionality provided by P. Therefore Property 3 is satisfied. Let P and Q be any two objects with $\mu(P) = n_P$ and $\mu(Q) = n_Q$. If P and Q are combined, the resulting object will have $n_P + n_Q - \partial$ couples, where ∂ is the number of couples reduced due to the combination. That is $\mu(P+Q) = n_P + n_Q - \partial$, where ∂ is some function of the methods of P and Q. Clearly, $n_P - \partial \geq 0$ and $n_Q - \partial \geq 0$ since the reduction in couples cannot be greater than the original number of couples. Therefore,

$n_P + n_Q - \partial \geq n_P$ for all P and Q and

$n_P + n_Q - \partial \geq n_Q$ for all P and Q

i.e., $\mu(P+Q) \geq \mu(P)$ and $\mu(P+Q) \geq \mu(Q)$ for all P and Q. Thus, Property 4 is satisfied. Let P and Q be two objects such that $\mu(P) = \mu(Q) = n$, and let R be another object with $\mu(R) = r$.

$\mu(P+Q) = n + r - \partial$, similarly

$\mu(Q+R) = n + r - ß$

Given that ∂ and ß are independent functions, they will not be equal, i.e., $\mu(P+R)$ is not equal to $\mu(Q+R)$, satisfying Property 5. For any two objects P and Q, $\mu(P+Q) = n_P + n_Q - \partial$.

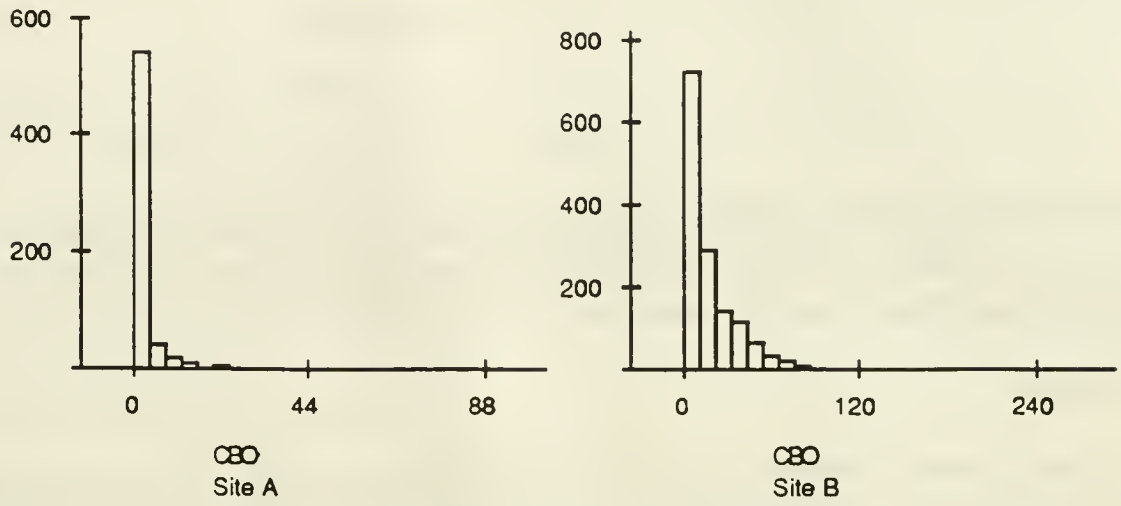
$\mu(P+Q) = \mu(P) + \mu(Q) - \partial$ which implies that

$\mu(P+Q) \leq \mu(P) + \mu(Q)$ for all P and Q.

Therefore Property 6 is not satisfied.

Empirical Data

The histograms and summary statistics from both sites are shown below:



Histograms for the CBO metric

Site	Metric	Mean	Median	Max	Min	StdDev	Skewness
A	CBO	2.01	0	84	0	5.56	7.25
B	CBO	16.90	9	234	0	22.21	3.04

Interpretation of Data. Both Site A and Site B class libraries have skewed distributions for CBO, but the Smalltalk application at Site B has higher mean and median values. One possible explanation is that contingency factors (type of application, language, people etc.) are responsible for the difference. However, that does not explain the similarity in the shape of the distribution. One interpretation that may account for both the similarity and the higher values for Site B is that coupling between objects is an increasing function of the number of classes in the application. The Site B application has 1459 classes compared to the 634 classes at Site A. It is possible that complexity due to increased coupling is a characteristic of large class libraries. This could be an argument for a more informed selection of the scale size (as measured by number of classes) in order to limit the deleterious effects of coupling. The low median values of coupling at both sites suggest that at least 50% of the classes are self-contained and do not refer to other classes. This is consistent with the accepted design principle of minimizing connections between various parts of a design. The CBO metric can be used by senior designers and project managers as a relative simple way to track whether the class hierarchy is losing its integrity and whether different parts of a large system are developing unnecessary interconnections.

Metric 5: Response For a Class (RFC)

Definition. RFC = | RS | where RS is the response set for the class.

Theoretical basis. The response set for the class can be expressed as:

$$RS = \{ M \} \cup_{all\ i} \{ R_i \}$$

where $\{ R_i \}$ = set of methods called by method *i* and $\{ M \}$ = set of all methods in the class

The response set is a set of methods available to the object and its cardinality is a measure of the attributes of an object. Since it specifically includes methods called from outside the object, it is also a measure of communication between objects.

Viewpoints.

• If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding required on the part of the tester.

• The larger the number of methods that can be invoked from an class, the greater the complexity of the class.

• A worst case value for possible responses will assist in appropriate allocation of testing time.

Analytical Evaluation of Response for a Class (RFC)

Let X_P = RFC for class P

X_Q = RFC for class Q.

X_P and X_Q are functions of the number of methods and the external coupling of P and Q respectively. It follows from assumption 1 (since functions of i.i.d. random variables are also i.i.d.) that X_P and X_Q are i.i.d.. Therefore, there is a finite probability that $\exists$ a Q such that $\mu(P) \neq \mu(Q)$ resulting in property 1 being satisfied. Also there is a finite probability that $\exists$ a Q such that $\mu(P) = \mu(Q)$, therefore property 2 is satisfied. Since the choice of methods is a design decision, Property 3 is satisfied. Let P and Q be two classes with RFC of P = n_P and RFC of Q = n_Q. If these two classes are combined to form one class, the response for that class will depend on whether P and Q have any common methods. Clearly , there are two possible cases: 1) when P and Q have no common methods and therefore the combined class P + Q will have a response set = $n_P + n_Q$. 2) In the second case, P and Q have methods in common, and the response set will

smaller than $n_P + n_Q$. Mathematically, $\mu(P + Q) = n_P + n_Q - \partial$, where ∂ is some function of the methods of P and Q. Clearly, $n_P + n_Q - \partial \geq n_P$ and $n_P + n_Q - \partial \geq n_Q$ for all possible P and Q. $\mu(P+Q) \geq \mu(P)$ and $\geq \mu(Q)$ for all P and Q. Therefore, Property 4 is satisfied.

Let P and Q be two objects such that $\mu(P) = \mu(Q) = n$, and let R be another object with $\mu(R) = r$.

$\mu(P+Q) = n + r - \partial$, similarly
$\mu(Q+R) = n + r - ß$

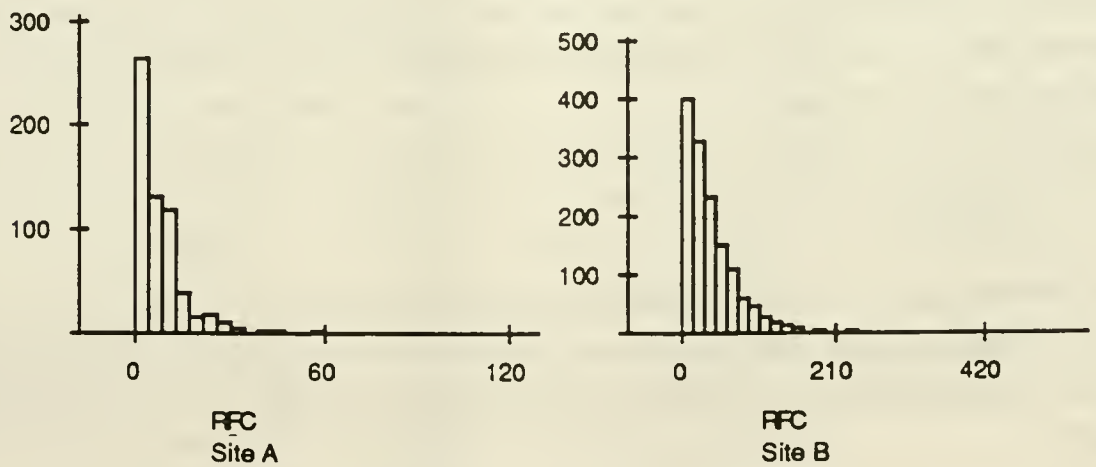
Given that ∂ and ß are independent functions, they will not be equal, i.e., $\mu(P+R)$ is not equal to $\mu(Q+R)$, satisfying Property 5. For any two objects P and Q, $\mu(P+Q) = n_P + n_Q - \partial$.

$\mu(P+Q) = \mu(P) + \mu(Q) - \partial$ which implies that
$\mu(P+Q) \leq \mu(P) + \mu(Q)$ for all P and Q.

Therefore Property 6 is not satisfied.

Empirical Data

The histograms and summary statistics from both sites are shown below:



Histograms for the RFC metric

Site	Metric	Mean	Median	Max	Min	StdDev	Skewness
A	RFC	10.00	6	120	0	13.78	3.70
B	RFC	42.35	29	422	3	45.24	2.95

Interpretation of Data. The data from both Site A and Site B suggest that most classes tend to able to invoke a small number of methods, while a few outliers maybe be most profligate in their

potential invocation of methods. This reinforces the argument that a small number of classes may be responsible for a large number of the methods that executed in an application, and that if testing effort were concentrated on these outlier classes, a bulk of the dynamic behavior of the object oriented systems can be checked for customer acceptance.

Another interesting aspect is the difference in values for RFC between Site A and B. Note that the mean, median and maximum values of RFC are higher than the RFC values at Site A. One possible hypothesis may relate to the differences in the memory management facilities provided by Smalltalk and C++. Site B has the benefit of automatic memory management provided by Smalltalk. C++ on the other hand, does not provide automatic garbage collection mechanisms for memory management. This could deter, at the design phase, the usage of message passing through method invocation, since this is memory intensive and requires periodic garbage collection. Another possibility is that complete adherence to object oriented principles in Smalltalk necessitates extensive method invocation, whereas C++'s incremental approach to object orientation gives designers alternatives to message passing through method invocation.

Metric 6: Lack of Cohesion in Methods (LCOM)

Definition. Consider a Class C_1 with n methods $M_1, M_2..., M_n$. Let $\{I_i\}$ = set of instance variables used by method M_i. There are n such sets $\{I_1\},... \{I_n\}$. Let $P = \{ (I_i,I_j) \mid I_i \cap I_j = 0 \}$ and $Q = \{ (I_i,I_j) \mid I_i \cap I_j \neq 0 \}$

$LCOM = |P| - |Q|$, if $|P| > |Q|$

$= 0$ otherwise

Example: Consider a class C with three methods M_1, M_2 and M_3. Let $\{I1\} = \{a,b,c,d,e\}$ and $\{I2\} = \{a,b,e\}$ and $\{I3\} = \{x,y,z\}$. $\{I1\} \cap \{I2\}$ is non-empty, but $\{I1\} \cap \{I3\}$ and $\{I2\} \cap \{I3\}$ are null sets and LCOM is the number of null intersections - number of non-empty intersections, which is 1.

Theoretical basis. This uses the notion of degree of similarity of methods. The degree of similarity for two methods in class C_1 is given by:

$$\sigma() = \{I_1\} \cap \{I_2\}$$

The larger the number of similar methods, the more cohesive the class, which is consistent with traditional notions of cohesion that measure the inter relatedness between portions of a program. If none of the methods of a class display any instance behavior, i.e. do not use any instance variables, they have no similarity and the LCOM value for the class will be zero. The LCOM value

provides a measure for the disparate nature of methods in the class. A smaller number of disjoint pairs (elements of set P) implies greater similarity of methods. LCOM is intimately tied to the instance variables and methods of an object, and therefore is a measure of the attributes of an object.

Viewpoints.

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

Analytical Evaluation of Lack Of Cohesion Of Methods (LCOM)

Let X_P = LCOM for class P
X_Q = LCOM for class Q.

X_P and X_Q are functions of the number of methods and the instance variables of P and Q respectively. It follows from assumption 1 (since functions of i.i.d. random variables are also i.i.d.) that X_P and X_Q are i.i.d.. Therefore, there is a finite probability that $\exists$ a Q such that $\mu(P) \neq \mu(Q)$ resulting in property 1 being satisfied. Also there is a finite probability that $\exists$ a Q such that $\mu(P) = \mu(Q)$, therefore property 2 is satisfied. Since the choice of methods and instance variables is a design decision, Property 3 is satisfied. Let P and Q be any two objects with $\mu(P) = n_P$ and $\mu(Q) = n_Q$. Combining these two objects can potentially reduce the number of disjoint sets. i.e., $\mu(P+Q) = n_P + n_Q - \partial$ where ∂ is the number of disjoint sets reduced due to the combination of P and Q. The reduction ∂ is some function of the particular sets of instance variables of the two objects P and Q. Now, $n_P \geq \partial$ and $n_Q \geq \partial$ since the reduction in sets obviously cannot be greater than the number of original sets. Therefore, the following result holds:

$n_P + n_Q - \partial \geq n_P$ for all P and Q and

$n_P + n_Q - \partial \geq n_Q$ for all P and Q.

Property 4 is satisfied.

Let P and Q be two objects such that $\mu(P) = \mu(Q) = n$, and let R be another object with $\mu(R) = r$.

$\mu(P+Q) = n + r - \partial$, similarly

$\mu(Q+R) = n + r - ß$

Given that ∂ and ß are independent functions, they will not be equal. i.e., $\mu(P+R) \neq \mu(Q+R)$, satisfying Property 5. For any two objects P and Q, $\mu(P+Q) = n_P + n_Q - \partial$. i.e.,

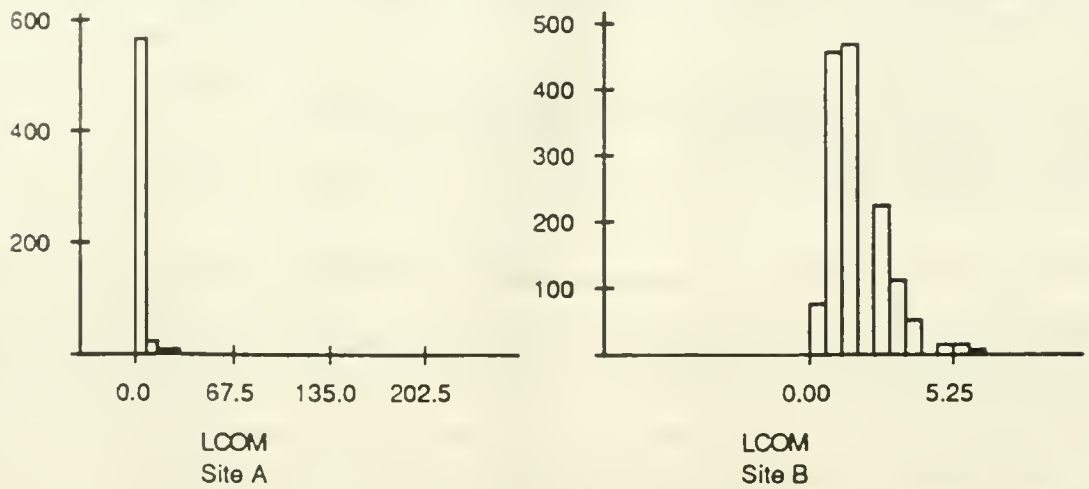
$\mu(P+Q) = \mu(P) + \mu(Q) - \partial$ which implies that

$\mu(P+Q) \leq \mu(P) + \mu(Q)$ for all P and Q.

Therefore Property 6 is not satisfied.

Empirical Data

The histograms and summary statistics from both sites are shown below:



Histograms for the LCOM metric

Site	Metric	Mean	Median	Max	Min	StdDev	Skewness
A	LCOM	4.03	0	200	0	16.94	7.31
B	LCOM	2.23	2	17	0	1.70	2.54

Interpretation of Data. At both sites, LCOM median values are extremely low, indicating that at least 50% of classes have cohesive methods. In other words, instance variables seem to be operated on by more than one method defined in the class. This is consistent with the principle of building methods around the essential data elements that define a class. The Site A application has a few outlier classes that have low cohesion, as evidenced by the high maximum value 200. In comparison, the Site B application has almost no outliers and smaller standard deviation, which is demonstrated by the difference in the shape of the two distributions.

A high LCOM value indicates disparateness in the functionality provided by the class. This metric can be used to identify classes that are attempting to achieve many different objectives, and consequently behave in less predictable ways than classes. Such classes could be more error prone and more difficult to test and could possibly be broken up into two or more classes that are more well defined in their behavior. The LCOM metric can be used by senior designers and project managers as a relatively simple way to track whether the cohesion principle is adhered to in the design of an application and advise if necessary, at an earlier phase in the design cycle.

The Metrics Suite and Booch OOD Steps

The six metrics attempt to capture the three non-implementation steps in Booch's definition of OOD. DIT and NOC directly relate to the layout of the class hierarchy, and WMC is an aspect of the complexity of the class. This suggests that WMC, DIT and NOC relate to the object definition step in OOD. WMC and RFC attempt to capture how the class is "behaves" when it gets messages. For example, if a class has a large WMC or RFC, it has many possible responses (since a potentially large number of methods can execute). The CBO and RFC metrics also attempt to capture the extent of communication between classes by counting the inter-object couples and methods external to a given class. The table below summarizes the six metrics in relation to the steps in Booch's definition of OOD.

Metric	Object Definition	Object Semantics	Relationships between Objects
WMC	✔	✔	
DIT	✔		
NOC	✔		
RFC		✔	✔
CBO			✔
LCOM		✔	

Table 1: Mapping of Metrics to Booch OOD Steps

Summary of Analytical Results

All the metrics satisfy the majority of the properties prescribed by Weyuker, with one strong exception, Property 6 (interaction increases complexity). Property 6 is not met by any of the metrics in this suite. Weyuker's rationale for Property 6 is to allow for the possibility of increased complexity due to interaction. The metrics proposed in this paper do not meet this requirement due to an assumption on how classes are combined. The presumption made in this analysis is that two objects may have similar methods or instance variables or children and, were these two objects to be combined, one set of the common elements may be discarded. If this possibility is not

considered, then all six metrics would satisfy Property 6. Zuse's analysis of Property 6 suggests that metrics that fail to meet this property cannot be used as a ratio scale, and therefore one could not use these metrics to, for example, demonstrate that one class is twice as complex as another [33].[13]

Failing to meet Property 6 implies that a complexity metric could *increase*, rather than reduce, if an object is divided into more objects. Interestingly, the experienced OO designers who participated in this study found that memory management and run-time detection of errors are both more difficult when there are a large number of objects to deal with. In other words, their viewpoint was that complexity can increase when objects are divided into more objects. Therefore, Property 6 may not be an essential feature for OO software design complexity metrics.

The only other violation of Weyuker's properties is for the single case of the DIT metric. The DIT metric fails to satisfy Property 4 (monotonicity) only in cases where two objects are in a parent-child relationship. This is because the distance from the root of a parent cannot become greater than one of its children. In all other cases, the DIT metric satisfies Property 4.[14]

Concluding Remarks

This research has developed and implemented a new set of software metrics for OO design. These metrics are based in measurement theory and also reflect the viewpoints of experienced OO software developers. In evaluating these metrics against a set of standard criteria, they are found to both (a) possess a number of desirable properties, and (b) suggest some ways in which the OO approach may differ in terms of desirable or necessary design features from more traditional approaches. Clearly, some future research designed to further investigate these apparent differences seems warranted.

In addition to the proposal and analytic test of theoretically grounded metrics, this paper has also presented empirical data on these metrics from actual commercial systems. The implementation independence of these metrics is demonstrated in part through data collection from both C++ and Smalltalk implementations, two of the most widely used object oriented environments. These data are used to demonstrate not only the feasibility of data collection, but also to suggest ways in which these metrics might be used by managers. In addition to the usual benefits obtained from valid measurements, OO design metrics should offer needed insights into whether developers are

---

[13] Property 6 is the only property that requires ratio scales.

[14] It is interesting to note that other authors have also observed difficulties in applying this particular property of Weyuker's. For example, see [11].

following OO principles in their designs. For example, in the results presented here it was shown that a significant percentage of classes were not benefiting from the potential reuse opportunities via inheritance. This use of metrics may be an especially critical one as organizations begin the process of migrating their staffs toward the adoption of OO principles.

Collectively, the suite provides senior designers and managers, who may not be completely familiar with the design details of an application, with an indication of the integrity of the design. They can use it as a vehicle to address the architectural and structural consistency of the entire application. By using the metrics suite they can identify areas of the application that may require more rigorous testing and areas that should be redesigned. Using the metrics in this manner, potential flaws and other leverage points in the design can be identified and dealt with earlier in the design-develop-test-maintenance cycle of an application. Yet another benefit of using these metrics is the added insight gained about trade-offs made by designers between conflicting requirements such as increased reuse (via more inheritance) and ease of testing (via a less complicated inheritance hierarchy). Since there are typically many possible OO designs for the same application, these metrics can help in selecting one that is most appropriate to the goals of the organization, such as reducing the cost of development, testing and maintenance over the life of the application.

This set of six proposed metrics is presented as the first empirically validated proposal for formal metrics for OOD. By bringing together the rigor of measurement theory, Bunge's ontology, Weyuker's evaluation criteria and empirical data from professional software developers working on commercial projects, this paper seeks to demonstrate the level of rigor required in the development of usable metrics for design of software systems. Of course, there is no reason to believe that the proposed metrics will be found to be comprehensive, and further work could result in additions, changes and possible deletions from this suite. However, they do provide coverage for all three of Booch's steps for OOD and, at a minimum, this metrics suite should lay the groundwork for a formal language to describe metrics for OOD. In addition, these metrics may also serve as a generalized solution for other researchers to rely on when seeking to develop specialized metrics for particular purposes or customized environments.

It is often noted that OO may hold some of the solutions to the software crisis. Further research in moving OO development management towards a strong theoretical base should provide a basis for significant future progress.

Bibliography

[1] Banerjee, J. *et al.*, "Data Model Issues for Object Oriented Applications", *ACM Transactions on Office Information Systems*, vol. 5, pp. 3-26, 1987.

[2] Basili, V. and Reiter, R., Evaluating Automatable Measures of Software Models, *IEEE Workshop on Quantitative Software Models*, 1979, Kiamesha, NY, pp. 107-116.

[3] Booch, G., *Object Oriented Design with Applications*, Redwood City, CA:Benjamin/Cummings, 1991.

[4] Bunge, M., *Treatise on Basic Philosophy : Ontology I : The Furniture of the World*, Boston:Riedel, 1977.

[5] Bunge, M., *Treatise on Basic Philosophy : Ontology II : The World of Systems*, Boston:Riedel, 1979.

[6] Chemiavsky, J.C. and Smith, C.H., "On Weyuker's Axioms for Software Complexity Measures", *IEEE Transactions on Software Engineering*, vol. 17, pp. 636-638, 1991.

[7] Chemiavsky, V. and Lakhuty, D.G., "On The Problem of Information System Evaluation", *Automatic Documentation and Mathematical Linguistics*, vol. 4, pp. 9-26, 1971.

[8] Chidamber, S.R. and Kemerer, C.F., Towards a Metrics Suite for Object Oriented Design, *Proc. of the 6th ACM Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1991, Phoenix, AZ, pp. 197-211.

[9] Fenton, N., "When a software measure is not a measure", *Software Engineering Journal*, vol. pp. 357-362, September 1992.

[10] Fichman, R. and Kemerer, C., "Object-Oriented and Conventional Analysis and Design Methodologies: Comparison and Critique", *IEEE Computer*, vol. 25, pp. 20-39, October 1992.

[11] Harrison, W., "Software Science and Weyuker's Fifth Property", University of Portland Computer Science Department Internal Report 1988.

[12] Kalakota, R. *et al.*, The Role of Complexity in Object-Oriented Systems Development, *26th Annual Hawaii International Conference on System Science*, 1993, Maui, Hawaii, pp. 759-768.

[13] Kaposi, A.A., Measurement Theory, in McDermid, J. ed., *Software Engineer's Reference Book*, Oxford: Butterworth-Heinemann Ltd., 1991.

[14] Kearney, J.K. *et al.*, "Software Complexity Measurement", *Communications of the ACM*, vol. 29, pp. 1044-1050, 1986.

[15] Kemerer, C.F., "Reliability of Function Points Measurement: A Field Experiment", *Communications of the ACM*, vol. 36, pp. 85-97, February 1993.

[16] Kim, J. and Lerch, J.F., "Cognitive Processes in Logical Design: Comparing Object-Oriented and Traditional Functional Decomposition Software Methodologies", Carnegie Mellon University Graduate School of Industrial Administration working paper 1991.

[17] Kriz, J., *Facts and Artifacts in Social Science: An Epistemological and methodological analysis of empirical social science techniques*, New York:McGraw Hill, 1988.

[18] Lieberherr, K. *et al.*, Object Oriented Programming : An Objective Sense of Style, *Third Annual ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1988, pp. 323-334.

[19] Moreau, D.R. and Dominick, W.D., "Object Oriented Graphical Information Systems: Research Plan and Evaluation Metrics", *Journal of Systems and Software*, vol. 10, pp. 23-28, 1989.

[20] Morris, K., *Metrics for Object Oriented Software Development*, M.I.T. Sloan School Masters thesis, 1988.

[21] Parsons, J. and Wand, Y., "Object-Oriented Systems Analysis: A Representational View", University of British Columbia Working Paper January 1993.

[22] Pfleeger, S.L. and Palmer, J.D., Software Estimation for Object-Oriented Systems, *1990 International Function Point Users Group Fall Conference*, 1990, San Antonio, Texas, pp. 181-196.

[23] Prather, R.E., "An Axiomatic Theory of Software Complexity Measures", *Computer Journal*, vol. 27, pp. 340-346, 1984.

[24] Roberts, F., *Encyclopedia of Mathematics and its Applications*, Addison Wesley Publishing Company, 1979.

[25] Sheetz, S.D. *et al.*, "Measuring Object-Oriented System Complexity", University of Colorado Working Paper 1992.

[26] Vessey, I. and Weber, R., "Research on Structured Programming: An Empiricist's Evaluation", *IEEE Transactions on Software Engineering*, vol. SE-10, pp. 394-407, 1984.

[27] Wand, Y., A Proposal for a Formal Model of Objects, in Kim, W. and Lochovsky, F. ed., *Object-Oriented Concepts, Databases and Applications*, Reading, MA: Addison-Wesley, 1989.

[28] Wand, Y. and Weber, R., An Ontological Evaluation of Systems Analysis and Design Methods, in Falkenberg, E.D. and Lindgreen, P. ed., *Information Systems Concepts: An In-depth Analysis*, Amsterdam: Elsevier Science Publishers, 1989.

[29] Wand, Y. and Weber, R., "An Ontological Model of an Information System", *IEEE Transactions on Software Engineering*, vol. 16, pp. 1282-1292, November 1990.

[30] Wand, Y. and Weber, R., Toward A Theory Of The Deep Structure Of Information Systems, *International Conference on Information Systems*, 1990, Copenhagen, Denmark, pp. 61-71.

[31] Weyuker, E., "Evaluating Software Complexity Measures", *IEEE Transactions on Software Engineering*, vol. 14, pp. 1357-1365, September 1988.

[32] Whitmire, S., Measuring Complexity in Object-Oriented Software, *Third International Conference on Applications of Software Measurement*, 1992, La Jolla, California.

[33] Zuse, H., "Properties of Software Measures", *Software Quality Journal*, vol. 1, pp. 225-260, December 1992.