A METRICS SUITE FOR OBJECT ORIENTED DESIGN

Shyam R. Chidamber

Chris F. Kemerer

M.I.T. Sloan School of Management
E53-315
30 Wadsworth Street, Cambridge, MA 02142
(617)-253-2791 (office)
(617)-258-7579 (fax)
CKEMERER@SLOAN.MIT.EDU

Revised December 1993

This research was supported in part by the M.I.T. Center for Information Systems Research (CISR), and the cooperation of two industrial organizations who supplied the data. Helpful comments were received on earlier drafts from N. Fenton, Y. Wand, R. Weber and S. Zweben, plus four anonymous reviewers.

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.

“A Metrics Suite For Object Oriented Design”

Shyam R. Chidamber

Chris F. Kemerer

Index Terms

CR Categories and Subject Descriptors: D.2.8 [Software Engineering]: Metrics; **D.2.9 [Software Engineering]**: Management; **F.2.3 [Analysis of Algorithms and Problem Complexity]**: Tradeoffs among Complexity Measures; **K.6.3 [Management of Computing and Information Systems]**: Software Management

General Terms: Class, Complexity, Design, Management, Measurement, Metrics, Object Orientation, Performance.

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 [41], lacking in desirable measurement properties [47] being insufficiently generalized or too implementation technology dependent [45], and being too labor-intensive to collect [22].

Following Wand and Weber, the theoretical base chosen for the OO design metrics was the ontology of Bunge [5, 6, 43]. 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 is 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 [21]. Vessey and Weber also commented on the general lack of theoretical rigor in the structured programming literature [41]. 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 [34, 47]. This suggests that software metrics need to be constructed with a stronger degree of theoretical and mathematical rigor.

2

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 separates 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. [4, 23. 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 [49. 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. Tegarden *et al.* and Bilow have called for theoretical rigor in the design of OO metrics [40 [The challenge is therefore to propose metrics that are firmly rooted in theory and relevant to practitioners in organizations. Some initial proposals for such metrics are set out by Morris, although they are not tested [31]. 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 [28. Likewise Coplien suggests a number of rules of thumb for OO programming in C++ [12. Moreau and Dominick suggest three metrics for OO graphical information systems, but do not provide formal, testable definitions [30. 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 [33]. Lake and Cook prescribe metrics for measurement of inheritance in C++ environments, and have gathered data from an experimental system using an automated tool [25. Other authors, such as Chidamber and Kemerer, Sheetz, *et al.*, and Whitmire propose metrics, but do not offer any empirical data [10, 38, 48]. More recently, Rajaraman and Lyu [35] and Li and Henry [27 test the metrics proposed in [10 and measured them for applications developed by university students. However, despite the active interest in this area, no empirical metrics data from commercial object oriented applications have 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 illustrate the characteristics of these metrics on real applications, 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 by Booch [4][1]. He outlines four major steps involved in the object-oriented design process:

[20] For a comparison and critique of six different OO analysis and design methodologies see [15.

3

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.

Whether the design methodology chosen is Booch's OOD or any of the several other methodologies, design of classes is consistently declared to be central to the OO paradigm. As deChampeaux *et al*. suggest, class design is the highest priority in OOD, and since it deals with the functional requirements of the system, it must occur before systems design (mapping objects to processors, processes) and program design (reconciling of functionality using the target languages, tools etc.) [13]. Given the importance of class design, the metrics outlined in this paper specifically are designed to measure the complexity in the design of classes[2]. The limitation of this approach is that possible dynamic behavior of a system is not captured. Since the proposed metrics are aimed at assessing the design of an object oriented system rather than its specific implementation, 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 design 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*. [36]. 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. By starting with these definitions, the mathematical role of metrics as a mapping (or transformation) can be formally outlined. Notationally:

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

where

A is a set of object-elements

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

$O_1 ... O_m$ are binary operations on elements of **A** (e.g., combination)

[2] These are therefore static metrics, and they can be gathered prior to program execution.

4

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 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 [9]. More recently, Fenton states that viewpoints characterize intuitive understanding and that viewpoints must be the logical starting point for the definition of metrics [14. 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 $.^3$ defined on a set **P** (the set of all possible designs). For P, P', P'' $\in$ **P**, the following two axioms must hold:

P $.^3$ P' or P' $.^3$ P (*completeness*: P is more complex than P' or P' is more complex than P)

P$.^3$ P', P'$.^3$ P'' => P$.^3$ 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 [36].

To be able to measure something about an 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:

F $\equiv$ (**C**, $S_1 ... S_n$, $B_1 ... B_m$)

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

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

$B_1 ... B_m$ are *binary* operations on elements of **C** (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**, μ(a) $\in$ **F**. It must be noted here that μ preserves and does not alter the implicit notion underlying the empirical relations. The example below involving a set of school children illustrates the mapping between an empirical relational system and a formal relational system20:

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: Combination: two children standing atop one another	Binary Operations: +: add the real numbers associated with the two children

5

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 [24. 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 better understanding the empirical system. 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 or not well understood [20]. Design of object oriented systems is a difficult undertaking in part due to the newness of the technology, and the consequent lack of formal metrics 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. 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 [42, 44, 46]. 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 [32]. 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; as Bunge notes, "there are no bare individuals except in our imagination" [5.

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* [42, 43. Therefore, an object is not simply a bundle of methods, but a representation of the application domain that includes the methods and instance variables that a designer assigns to that object. Another benefit of this stream of research is that it provides a formal mathematical approach to dealing specifically with the key ideas of object orientation.

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 object is represented in a system. In object oriented terminology, the instance variables[3] together with its methods[4] are the properties of the object [1.

[3] An instance variable stores a unique value in each instance of a class.

6

Using these representations of objects, previous research has defined concepts like scope and similarity that are relevant to object oriented systems [5, 42]. Following this tradition, this paper defines in the paragraphs below two important software design concepts for object classes, coupling and cohesion. Intuitively, 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. It should be noted that these definitions are derived from the ontology of objects as opposed to other sources that have been graph-theory (e.g. McCabe [29]), information content (e.g. Halstead [17]) or structural attributes (e.g. Card and Agresti [7]). For further details on the appropriateness of the ontological approach the reader is referred to the comprehensive treatment of the subject in [42, [32] and [39].

Coupling. In ontological terms, "two objects 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" [41, p 547].

Let X = <x, p(x)> and Y = <y, p(y)> be two objects.

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

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

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. Since objects of the same class have the same properties, two classes are coupled when methods declared in one class use methods or instance variables of the other class[5].

Cohesion. Bunge defines *similarity* $\sigma()$ of two things to be the intersection of the sets of properties of the two things [5, p 87]:

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

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.

$$\sigma(M_1,M_2) = \{ I_1 \} \cap \{ I_2 \}$$

---

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

[5] Note that this will include coupling due to inheritance.

7

where $\sigma(M_1,M_2)$ = degree of similarity of methods M_1 and M_2

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

Example: Let $\{I_1\} = \{a,b,c,d,e\}$ and $\{I_2\} = \{a,b,e\}$. $\{I1\} \cap \{I2\}$ is non-empty, and $\sigma(M_1,M_2) = \{a,b,e\}$.

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, that is, the bundling of methods and instance variables in an object class. The *degree of similarity* of methods can be viewed as a major aspect of object class cohesiveness. If an object class has different methods performing different operations on the same set of instance variables, the class is cohesive. This view of cohesion is centered on data that is encapsulated within an object and on how methods interact with data. It is proposed for object orientation as an alternative to other previous approaches, such as generalization-specialization cohesion or service cohesion as defined by Coad and Yourdon [11].

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 [5]. Using this definition as a base, the complexity of an object class can be defined to be the cardinality of its set of properties.

Complexity of <x, p(x)> = | p(x) |, where | p(x) | is the cardinality of p(x).

Scope of Properties. In simple terms, a class is a set of objects that have common properties (i.e. methods and instance variables). A designer develops an abstraction of the application domain by arranging the classes in a hierarchy. The inheritance hierarchy is a directed acyclic graph that can be described as a tree structure with classes as nodes, leaves and a root. In any application, there can be many possible choices for the class hierarchy. Design choices on the hierarchy employed to represent the application are essentially choices about restricting or expanding the scope of properties of the classes of 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 = depth of the class in the inheritance tree

The depth 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 ontological notion of scope of properties[6], 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.

[6] For formal mathematical definitions of scope of properties, see [44].

8

Methods as measures of communication. In the object oriented approach, objects communicate primarily through message passing[7]. 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}

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. During the implementation and maintenance phases of systems development, the response set may change, since new object instantiations may create different communication links.

Combination of object classes. As Booch observes, class design is an iterative process involving sub-classing (creating new classes based on existing ones), factoring (splitting existing classes into smaller ones) and composition (or combination) that unites existing classes into one. The notion of sub-classing is well understood in OO design, but the semantics of combination are less clear. However, Bunge's ontology provides a basis for defining the combination of object classes. From the principle of additive aggregation of two (or more) things, the combination of two object classes results in another class whose properties are the union of the properties of the component classes.

Let X = <x, p(x)> and Y = <y, p(y)> be two object classes, then X+Y is defined as <z, p(z)> where z is the token with which X+Y is represented and p(z) is given by:

$$p(z) = p(x) \cup p(y)$$

For example, if a class *foo_a* has properties (i.e. methods and instance variables) *a,b,c,d* and class *foo_b* has properties *a,l,m,n* then *foo_a + foo_b* has properties *a,b,c,d,l,m,n*.. If *foo_a* and *foo_b* both have identical properties *a,b,c,d* , then *foo_a + foo_b* will also have the same properties *a,b,c,d* .

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

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

[7]While objects can communicate through more complex mechanisms like bulletin boards, a majority of OO designers employ message passing as the primary mechanism for communicating between objects [4.

9

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.

Consistent with the desire to move metrics research into a more rigorous footing, it is desirable to have a formal set of criteria with which to evaluate proposed metrics. More recently, Weyuker has developed a formal list of desiderata for software metrics and has evaluated a number of existing software metrics using these properties [47]. These desiderata include notions of monotonicity, interaction, non-coarseness, non-uniqueness and permutation.

Weyuker's properties are not without criticism. Fenton suggests that Weyuker's properties are not predicated on a single consistent view of complexity [14]. Zuse criticizes Weyuker on the grounds that her properties are not consistent with the principles of scaling [50]. Cherniavsky and Smith suggest that Weyuker's properties should be used carefully since the properties may only give necessary, but not sufficient conditions for good complexity metrics [8].

However, as Gustafson and Prasad suggest, formal analytical approaches subsume most of the earlier, less well-defined and informal properties and provide a language for evaluation of metrics [16]. Her list, while currently still subject to debate and refinement, is a widely known formal analytical approach, and is therefore chosen for this analysis. Finally, in the course of the analysis presented below further suggestions are offered on the relative appropriateness of these axioms for object oriented development.

Of Weyuker's nine properties, three will be dealt with only briefly here. Weyuker's second property, "granularity", only requires that there be a finite number of cases having the same metric value. Since the universe of discourse deals with at most a finite set of applications, each of which has a finite number of classes, this property will be met by any metric measured at the class level. The "renaming property" (Property 8) requires that when the name of the measured entity changes[8], the metric should remain unchanged. As all metrics proposed in this paper are measured at the class level and, as none of them depend on the names of the class or the methods and instance variables, they also satisfy this property. Since both these properties are met, they will not be discussed further.

Weyuker's seventh property requires that permutation of elements within the item being measured can change the metric value. The intent is to ensure that metric values change due to permutation of program statements. This property is meaningful in traditional program design, where the ordering of if-then-else blocks could alter the program logic (and consequent complexity). In OOD, a class is an abstraction of the problem space, and the order of statements within the class definition has no impact on eventual execution or use. For example, changing the order in which methods are declared does not affect the order in which they are executed, since methods are triggered by the receipt of different messages from other objects. In fact, Cherniavsky and Smith specifically suggest that this property is not appropriate for

[8] Note, this property deals only with the name of the entity, and not the names associated with any of the internals of the entity.

10

OOD metrics because "the rationales used may be applicable only to traditional programming" [8, p. 638]. Therefore, this property is not considered further. The remaining six properties are repeated below[9].

Property 1: Non-coarseness

Given a class P and a metric μ another class Q can always be found such that: μ(P) _ μ(Q). This implies that not every class 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 classes P and Q such that μ(P) = μ(Q). This implies that two classes can have the same metric value, i.e., the two classes are equally complex.

Property 3: Design details are important

Given two class designs, P and Q, which provide the same functionality, does not imply that μ(P) = μ(Q). The specifics of the class must influence the metric value. The intuition behind Property 3 is that even though two class designs perform the same function, the details of the design matter in determining the metric for the class.

Property 4: Monotonicity

For all classes P and Q, the following must hold: μ(P) ² μ(P+Q) and μ(Q) ² μ(P+Q) where P + Q implies combination of P and Q[10]. This implies that the metric for the combination of two classes can never be less than the metric for either of the component classes.

Property 5: Non-equivalence of interaction

∃ P, ∃ Q, ∃ R, such that

μ(P) = μ(Q) does not imply that μ(P+R) = μ(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

∃ P and ∃ Q such that:

μ(P) + μ(Q) < μ(P+Q)

---

[9]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 classes rather than programs.

[10]It should be noted that P+Q is the combination of two classes, whereas μ(P) + μ(Q) is the addition of the metric value of P and the metric value of Q.

11

The principle behind this property is that when two classes are combined, the interaction between classes can 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. This suggests that the number of methods, variables and couplings follow a statistical distribution that is not apparent to an observer of the system. Further, the observer cannot predict the variables, methods etc. of one class based on the knowledge of the variables, methods and couplings of another class in the system.

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. For example, a class "foo_one" has a method "draw" that is responsible for drawing an icon on a screen; another class "foo_two" also has a "draw" method. Now a designer decides to have a single class "foo" and combines the two classes. Instead of having two different "draw" methods the designer can decide to just have one "draw" method (albeit modified to reflect the new abstraction).

Assumption 3: The inheritance tree is "full", i.e., there is a root, intermediate nodes and leaves. This assumption merely states that an application does not consist *only* of stand-alone classes; there is some use of subclassing[11].

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, later data were collected at two new sites which used different languages[12].

---

[11] Based on the data from sites A and B, this appears to be a reasonable assumption.

[12] The metrics were gathered from code, since no other design artifacts were available at either site.

12

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 vendor that uses OOD in their development work and has a collection of different C++ class libraries. Metrics data from 634 classes from two C++ class libraries that are used in the design of graphical user interfaces (GUI) were collected. Both these libraries were used in different product applications for rapid prototyping and development of windows, icons and mouse-based interfaces. Reuse across different applications was one of the primary design objectives of these libraries. These typically were 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 $M_1,... M_n$ *that are defined in the class.* Let $c_1,... c_n$ be the complexity[13] 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 object classes and complexity is determined by the cardinality of its set of properties. The number of methods is, therefore, a measure of class definition as well as being attributes of a class, since attributes correspond to properties[14].

[13] 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 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. Any complexity metric used in this manner should have the properties of an interval scale to allow for summation. The general nature of the WMC metric is presented as a strength, not a weakness of this metric as has been suggested elsewhere [19].

[14] Note that this is one interpretation of Bunge's definition, since it does not include the number of instance variables in the definition of the metric. This is done for two reasons: 1) Developers expressed the view that methods are more time consuming to design than instance variables, and adding the instance variables to the definition will increase the noise in the relationship between this metric and design effort. 2) By restricting the metric to methods, the process of adding static complexity weights to methods will not detract from the comprehensibility of the metric.

13

Viewpoints

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

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

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

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 non-zero probability that $\exists$ Q such that $\mu(P) \neq \mu(Q)$, therefore property 1 is satisfied. Similarly, there is a non-zero probability that $\exists$ R such that $\mu(P) = \mu(R)$. Therefore property 2 is satisfied. The function of the class does not define the number of methods in a class. The choice of the number of methods is a design 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 $\mu(P+Q) = n_P+n_Q-\_$, where _ is the number of common methods between P and Q. Clearly, the maximum value of _ is $\min(n_P, n_Q)$. Therefore, $\mu(P+Q) \geq n_P+n_Q-\min(n_P, n_Q)$. It follows that $\mu(P+Q) \geq \mu(P)$ and $\mu(P+Q) \geq \mu(Q)$, thereby satisfying Property 4. Now, let $\mu(P) = n$, $\mu(Q) = n$, and $\exists$ a class R such that it has a number of methods _ in common with Q (as per assumption 2) and β methods in common with P, where $\_ \neq \beta$. Let $\mu(R) = r$.

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

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

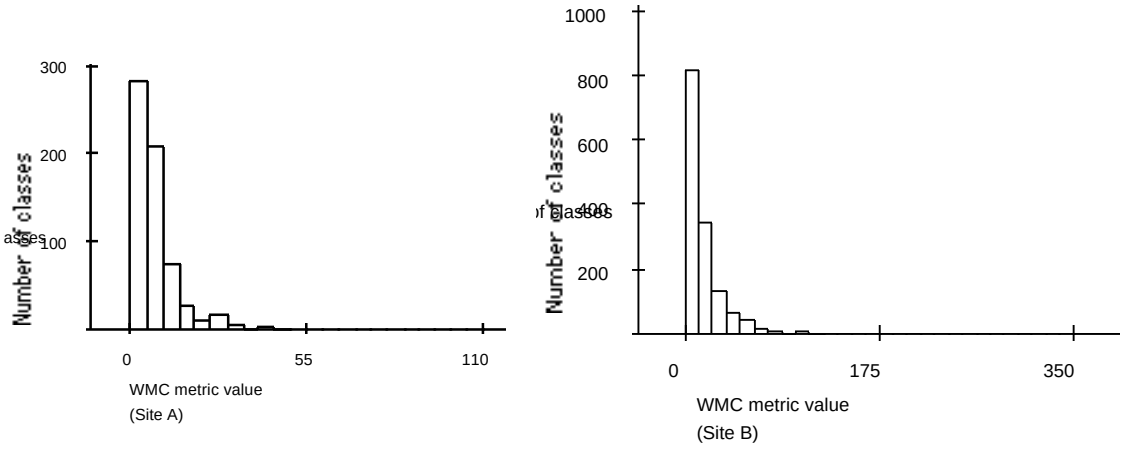
therefore $\mu(P+R) \neq \mu(Q+R)$ and Property 5 is satisfied. For any two classes P and Q, $n_P + n_Q - \_ \leq n_P + n_Q$ i.e., $\mu(P+Q) \leq \mu(P) + \mu(Q)$ for any P and Q. Therefore, Property 6 is not satisfied[15].

[15] The implications of not satisfying Property 6 is discussed in the Summary section.

14

Empirical Data

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



Histograms for the WMC metric

Site	Metric	Median	Max	Min
A	WMC	5	106	0
B	WMC	10	346	0

Summary Statistics for the WMC metric

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. Most classes in an application appear to be relatively simple in their construction, providing specific abstraction and functionality.

Examining the outlier classes at Site A revealed some interesting observations. The class with the maximum number of methods (106) had no children and was at the root of the hierarchy, whereas another outlier class with 87 methods had 14 sub-classes and a total number of 43 descendants. In the first case, the class's methods have no reuse within the application and, unless this is a generalized class that is reused across applications, the effort expended in developing this class will be a one-shot investment. However, the class with 87 methods has significant reuse potential within the application making increased attention to testing the methods in this class worthwhile, since the methods can have widespread use within the system.

Metric 2: Depth of Inheritance Tree (DIT)

Definition. Depth of inheritance of the class is the DIT metric for the class. In cases involving multiple inheritance, the DIT will be the maximum length from the node to the root of the tree.

15

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

Viewpoints.

• The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior[16].

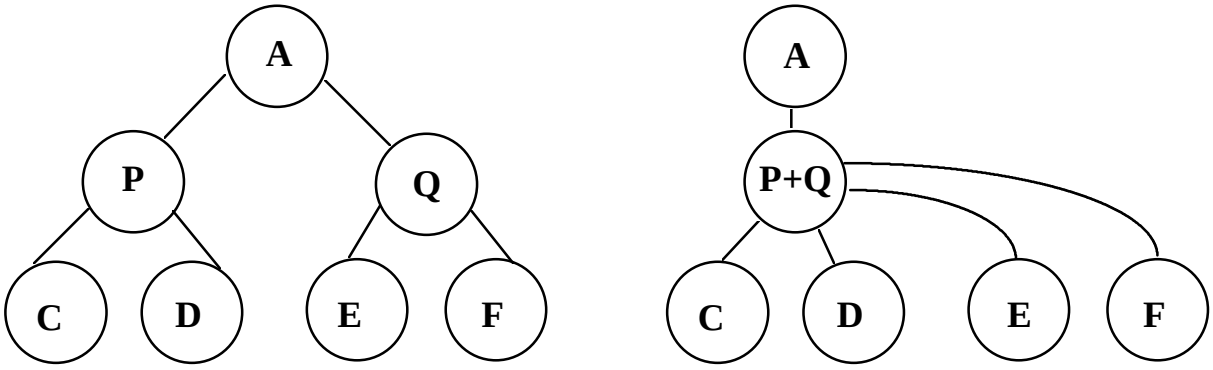
• Deeper trees constitute greater design complexity, since more methods and classes are involved.

• 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, the inheritance hierarchy 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 classes with the same depth of inheritance, i.e., property 2 is satisfied. Design of a class involves choosing what properties the class 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 classes P and Q are combined, there are three possible cases[17] : 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



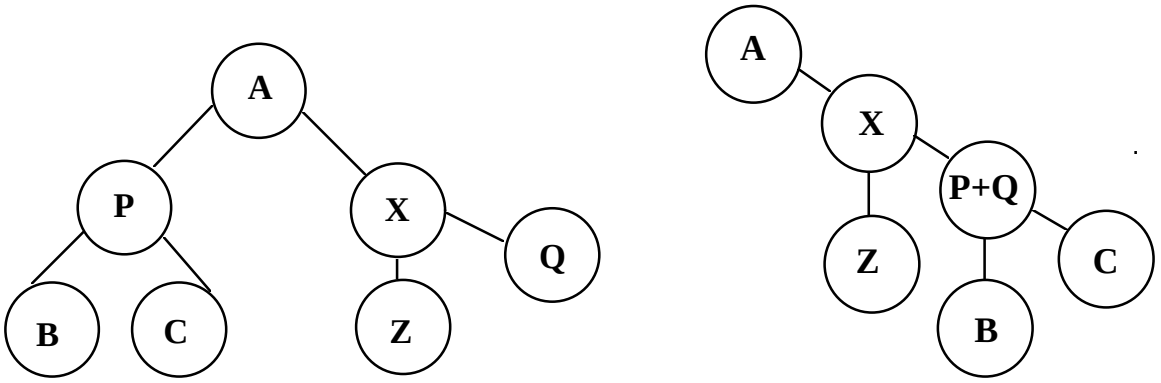
In this case, $\mu(P) = \mu(Q) = n$ and $\mu(P+Q) = n$, i.e. Property 4 is satisfied.

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

[16] Interestingly, this has been independently observed by other researchers [26].

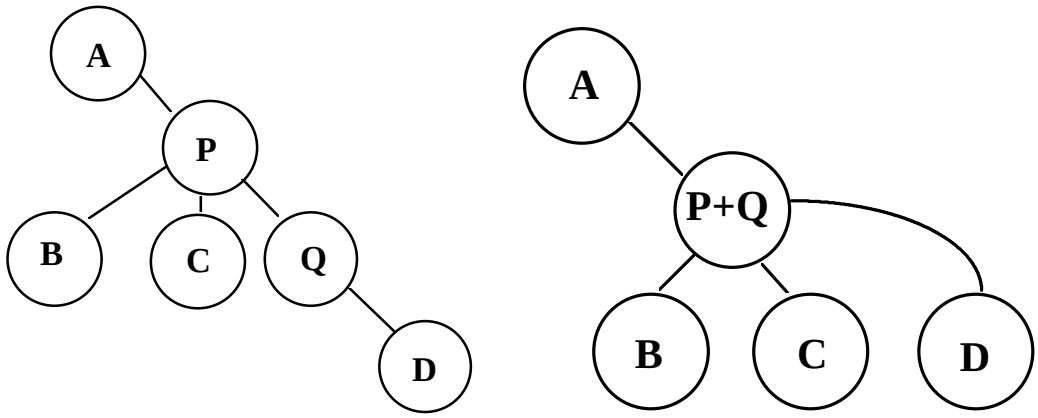
[17] A fourth case would involve multiple inheritance, and it can be shown that Property 4 is satisfied in this case also. Suppose A has two subclasses P and X, Q is a subclass of X and also a subclass of B. $\mu(P) = 1$ and $\mu(Q) = 2$. The combined class P+Q will be a subclass of X and B, and $\mu(P+Q) = 2$. In general, $\mu(P)$ and $\mu(Q)$ will be n_P and n_Q respectively and $\mu(P+Q)$ will be equal to $\text{Max}(n_P, n_Q)$. Consequently $\mu(P+Q)$ will always be greater than or equal to $\mu(P)$ and $\mu(Q)$.

16



If P+Q is located as the immediate ancestor to B and C (P's location) in the tree, the combined class cannot inherit methods from X, however if P+Q is located as an immediate child of X (Q's location), the combined class can still inherit methods from all the ancestors of P and Q. Therefore, P+Q will be in located Q's location[18]. In this case, μ(P) = x, μ(Q) = y and y > x. μ(P+Q) = y, i.e., μ(P+Q) > μ(P) and μ(P+Q) = μ (Q) and Property 4 is satisfied.

iii) when one is a child of the other[19]:



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

Let P and Q' be siblings, i.e. μ(P) = μ(Q')= n, and let R be a child of P. Then μ(P+R) = n and μ(Q'+R) = n + 1. i.e., μ(P+R) is not equal to μ(Q'+R). Therefore, Property 5 is satisfied. For any two classes P and Q, μ(P+ Q) = μ(P) or = μ(Q). Therefore, μ(P+Q) ² μ(P) + μ(Q), i.e., Property 6 is not satisfied.

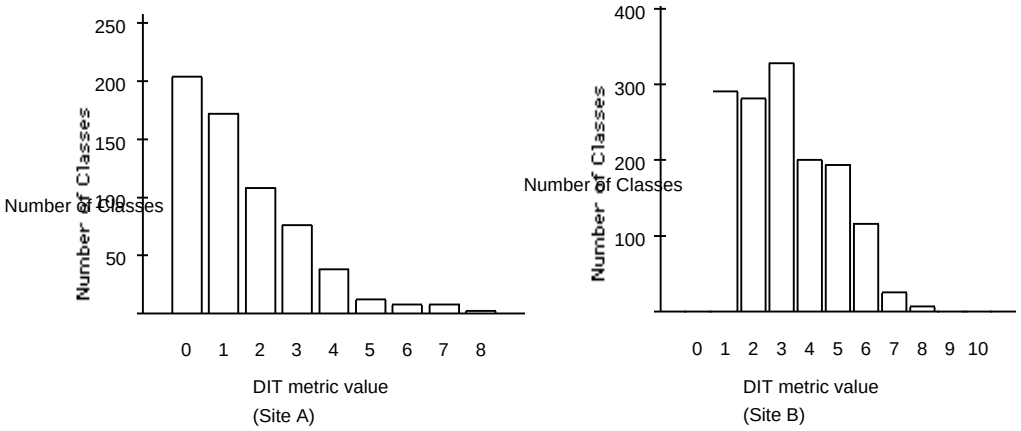
Empirical Data

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

[18] If there are several intermediate classes between P and the common ancestor of P and Q, the combined class will still be located as an immediate child of X and also inherit (via multiple inheritance) from P's immediate ancestors.

[19] This case is also representative of the situation where Q is a descendent, but not an immediate child of P.

17



Histograms for the DIT metric

Site	Metric	Median	Max	Min
A	DIT	1	8	0
B	DIT	3	10	0

Summary Statistics for the DIT metric

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[20]. Note that the Smalltalk application has a higher depth of inheritance due, in part, to the library of reusable classes that are a part of the language. For example, all classes are sub-classes of the class "object". 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 manageable number in order 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. Examining the class at Site A with a DIT value of 8 revealed that it was a case of increasingly specialized abstractions of a graphical concept of control panels. The class itself had only 4 methods and only local variables, but objects of this specialized class had a total 132 methods available through inheritance. Designing this class would have been a relatively simple task, but the testing could become

[20]Of course, such occurrences 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.

18

more complicated due to the high inheritance[21]. Resources between design and testing could be adjusted accordingly to reflect this.

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 is a form of reuse.

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

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

Analytical Evaluation of Number Of Children (NOC)

Let P and R be leaves, μ(P) = μ(R) = 0, let Q be the root μ(Q) > 0. μ(P) _ μ(Q) therefore property 1 is satisfied. Since μ(R) = μ(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., μ(P) = n_P and μ(Q) = n_Q). Combining P and Q[22], will yield a single class with $n_P + n_Q$ - _ 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. If Q is a sub-class of P, then P+Q will have $n_P + n_Q$ - 1 sub-classes . Therefore, in general the number of sub-classes of P+Q is $n_P + n_Q$ - β, where β = 1 or δ. Now, $n_P + n_Q$ -β ³ n_P and $n_P + n_Q$ -β ³ n_Q. This can be written as: μ(P+Q) ³ μ(P) and μ(P+Q) ³ μ(Q) for all P and all Q. Therefore, Property 4 is satisfied[23]. Let P and Q each have *n* children and R be a child of P which has *r* children. μ(P) = *n* = μ(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 μ(P+R) _ μ(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:

[21]Testers have frequently experienced that finding which method is executing (and from where) is a time consuming task [26].

[22] The combination of two classes will result in the combined class located in the inheritance hierarchy at the position of the class with the greater depth of inheritance.

[23] In cases where a class is both a parent and a grandparent of another class, this property will be violated. However, most OO environments will disallow this type of hierarchy.

19

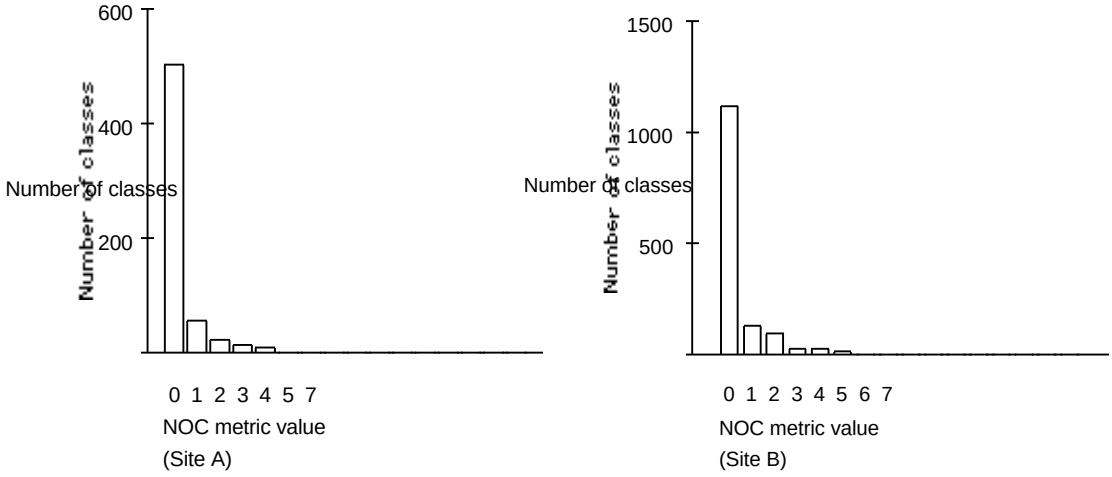
$\mu(P) = n_P$ and $\mu(Q) = n_Q$

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

where _ is the number of common children. Therefore, $\mu(P+Q) \geq \mu(P) + \mu(Q)$ 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	Median	Max	Min
A	NOC	0	42	0
B	NOC	0	50	0

Summary Statistics for the NOC metric

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 and that 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 from the histograms show that a majority of the classes (73% at Site A and 68% at Site B) 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. One explanation for the small NOC count could be that the design practice followed at the two sites dictated the use of shallow inheritance hierarchies[24]. A different explanation could be a lack of communication between different class designers and therefore that reuse opportunities are not being realized. Whatever the reason, the metric values and their distribution provide designers and managers with an opportunity to examine whether their particular design philosophy is being adhered to in the application. An examination of the class with 42 sub-classes at Site A was a GUI-command class for which all possible commands were separate sub-

[24] Some C++ designers at this site systematically avoid sub-classing in order to maximize operational performance.

20

classes. Further, none of these sub-classes had any sub-classes of their own. Systematic use of the NOC metric could have helped to restructure the class hierarchy to exploit common characteristic of different commands (e.g. text commands, mouse commands etc.).

Metric 4: Coupling between object classes (CBO)

Definition. CBO for a class is a count of the number of other classes to which it is coupled.

Theoretical basis. CBO relates to the notion that an object is coupled to another object if one of them acts on the other, i.e., methods of one use methods or instance variables of another. As stated earlier, since objects of the same class have the same properties, two classes are coupled when methods declared in one class use methods or instance variables defined by the other class.

Viewpoints.

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

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

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

Analytical Evaluation of Coupling Between Objects (CBO)

As per assumption 1, there exist classes P, Q and R such that μ(P) _ μ(Q) and μ(P) = μ(R) thereby satisfying properties 1 and 2. Inter-class coupling occurs when methods of one class use methods or instance variables of another class, 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 classes with $\mu(P) = n_P$ and $\mu(Q) = n_Q$. If P and Q are combined, the resulting class will have $n_P + n_Q - \_$ couples, where _ is the number of couples reduced due to the combination. That is $\mu(P+Q) = n_P + n_Q - \_$, where _ is some function of the methods of P and Q. Clearly, $n_P - \_ \geq 0$ and $n_Q - \_ \geq 0$ since the reduction in couples cannot be greater than the original number of couples. Therefore,

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

$n_P + n_Q - \_ \geq n_Q$ 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 classes such that $\mu(P) = \mu(Q) = n$, and let R be another class with $\mu(R) = r$.

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

$\mu(Q+R) = n + r - ß$

21

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

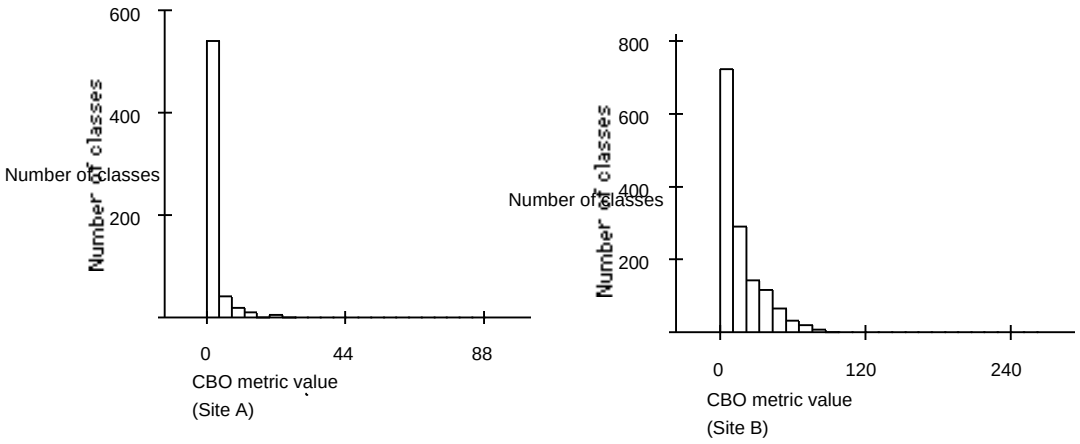
$\mu(P+Q) = \mu(P) + \mu(Q) - \_$ which implies that

$\mu(P+Q) \geq \mu(P) + \mu(Q)$ 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	Median	Max	Min
A	CBO	0	84	0
B	CBO	9	234	0

Summary Statistics for the CBO metric

Interpretation of Data. Both Site A and Site B class libraries have skewed distributions for CBO, but the Smalltalk application at Site B has relatively high median values. One possible explanation is that contingency factors (e.g., type of application) are responsible for the difference. A more likely reason is the difference between the Smalltalk and C++ languages[25]. Smalltalk requires virtually every interaction between run-time entities be done through message passing, while C++ does not. In Smalltalk, simple scalar variables (integers, reals, and characters) and control flow constructs like *if*, *while*, *repeat* statements are objects. Each of these invocations is performed via message passing which will be counted as an interaction in the CBO metric. Simple scalars will not be defined as C++ classes, and certainly control flow entities

[25] We are indebted to an anonymous referee who provided the following explanation.

22

are not objects in C++. Thus, CBO values are likely to be smaller in C++ applications. 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 classes 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 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 (including super-classes). Since a fair number of classes at both sites have no parents or no children, the limited use of inheritance may be also responsible for the small CBO values. Examination of the outliers at Site B revealed that classes responsible for managing interfaces have high CBO values. These classes tended to act as the connection point for two or more sub-systems within the same application. At Site A, the class with the highest CBO value was also the class with the highest NOC value, further suggesting the need to re-evaluate that portion of the 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 in inappropriate places.

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:

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

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

The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class[26]. The cardinality of this set is a measure of the attributes of objects in the class. Since it specifically includes methods called from outside the class, it is also a measure of the potential communication between the class and other classes.

Viewpoints.

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

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

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

[26] It should be noted that membership to the response set is defined only up to the first level of nesting of method calls due to the practical considerations involved in collection of the metric.

23

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 non-zero probability that $\exists$ a Q such that $\mu(P) \neq \mu(Q)$ resulting in property 1 being satisfied. Also there is a non-zero 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 three possible cases: 1) when P and Q have no common methods nor do their methods use any of the same methods, and therefore the combined class P + Q will have a response set = $n_P + n_Q$. 2) when P and Q have methods in common, and the response set will smaller than $n_P + n_Q$. 3) when P and Q have no methods in common but some of methods used by methods of P and Q are the same, the response set will be smaller than $n_P + n_Q$. For both cases 2 and 3, $\mu(P + Q) = n_P + n_Q - \_$, where _ is some function of the methods of P and Q. Clearly, $n_P + n_Q - \_ \geq n_P$ and $n_P + n_Q - \_ \geq n_Q$ 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 classes such that $\mu(P) = \mu(Q) = n$, and let R be another class with $\mu(R) = r$.

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

Given that _ and ß are independent functions, they will not necessarily be equal, i.e., $\mu(P+R)$ is not necessarily equal to $\mu(Q+R)$, satisfying Property 5. For any two classes P and Q,

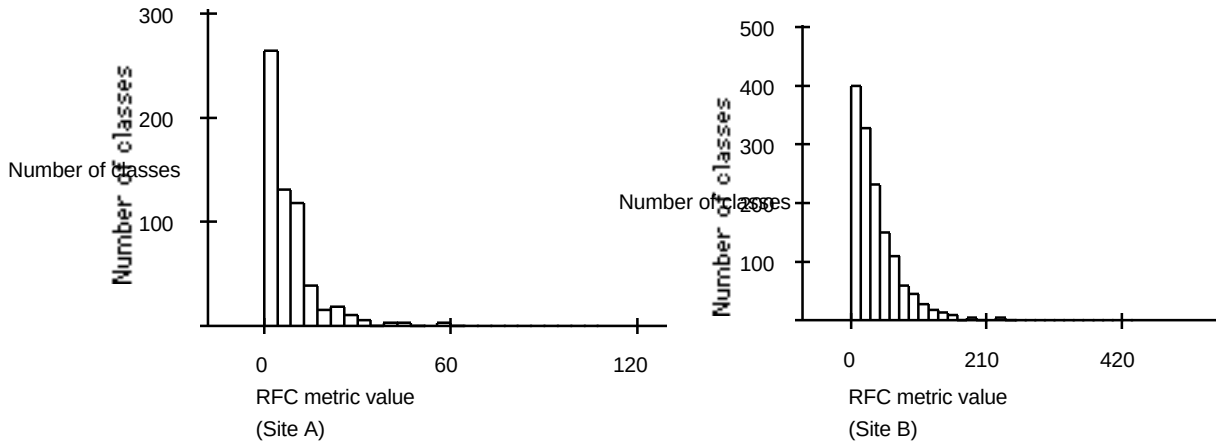
$\mu(P+Q) = \mu(P) + \mu(Q) - \_$ which implies that
$\mu(P+Q) \leq \mu(P) + \mu(Q)$ for all P and Q.

Therefore Property 6 is not satisfied.

Empirical Data

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

24



Histograms for the RFC metric

Site	Metric	Median	Max	Min
A	RFC	6	120	0
B	RFC	29	422	3

Summary Statistics for the RFC metric

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, either because they contain many methods (this appears to be the case at Site A) or that they call many methods. By using high RFC valued classes as structural drivers, high test coverage can be achieved during system test.

Another interesting aspect is the difference in values for RFC between Site A and B. Note that the median and maximum values of RFC at Site B are higher than the RFC values at Site A. As in the case of the CBO metric, this may relate to the complete adherence to object oriented principles in Smalltalk which necessitates extensive method invocation, whereas C++'s incremental approach to object orientation gives designers alternatives to message passing through method invocation[27]. Not surprisingly, at Site B high RFC value classes performed interface functions within the application. Since there are a number of classes that are stand-alone (i.e. no parents, no children, no coupling) the RFC values also tend to be low. Again, the metrics collectively and individually provide managers and designers a basis for examining the design of class hierarchies.

[27] RFC does not count calls to X-library functions and I/O functions like printf, scanf that are present in C++ applications. Similar functionality is obtained through interface classes in Smalltalk that are counted in the RFC calculations.

25

Metric 6: Lack of Cohesion in Methods (LCOM)

Definition. Consider a Class C_1 with *n* methods M_1, M_2..., M_n. Let $\{I_j\}$ = set of instance variables used by method M_i. There are *n* such sets $\{I_1\}$,... $\{I_n\}$. Let P = { (I_i, I_j) | $I_i \cap I_j = \varnothing$ } and Q = { (I_i, I_j) | $I_i \cap I_j$ _ $\varnothing$ }. If all n sets $\{I_1\}$,... $\{I_n\}$ are $\varnothing$ then let P = $\varnothing$.

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

= 0 otherwise[28]

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. LCOM is the (number of null intersections - number of non-empty intersections), which in this case is 1.

Theoretical basis. This uses the notion of degree of similarity of methods. The degree of similarity for two methods M_1 and M_2 in class C_1 is given by:

$\sigma() = \{I_1\} \cap \{I_2\}$ where $\{I_1\}$ and $\{I_2\}$ are the sets of instance variables used by M_1 and M_2

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

Viewpoints.

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

Analytical Evaluation of Lack Of Cohesion Of Methods (LCOM)

Let X_P = LCOM for class P

X_Q = LCOM for class Q.

[28] Note that the LCOM metric for a class where |P| = |Q| will be zero. This does not imply maximal cohesiveness, since within the set of classes with LCOM = 0, some may be more cohesive than others.

26

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 non-zero probability that $\exists$ a Q such that $\mu(P) \neq \mu(Q)$ resulting in property 1 being satisfied. Also there is a non-zero 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.

Suppose class P has 3 methods M_1, M_2, M_3, and M_2 and M_3 use common instance variables, while M_1 has no common instance variables with M_2 and M_3. The LCOM for P will be 1. Now, let another class Q have 3 methods, all of which use common instance variables. The LCOM for Q will be 0. When P and Q are combined, if the instance variables of Q are the same as the variables used by M_2 and M_3, the LCOM for P+Q will become 0, since the number of non-empty intersections will exceed the number of empty intersections. This implies that $\mu(P+Q) > \mu(Q)$, which violates Property 4. Therefore, LCOM does not satisfy Property 4[29].

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

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

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

Given that α and β are independent functions, they will not necessarily be equal. i.e., $\mu(P+R) \neq \mu(Q+R)$, satisfying Property 5. For any two classes P and Q, $\mu(P+Q) = n_P + n_Q - \alpha$. i.e.,

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

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

Therefore Property 6 is not satisfied[30].

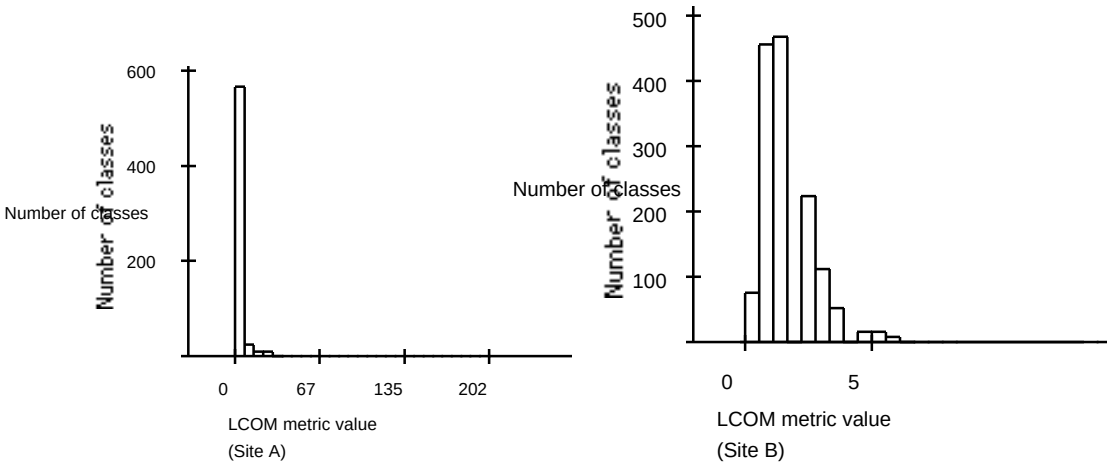
[29] We are indebted to the associate editor for providing this example.

[30] It can be shown for some cases that the number of disjoint sets will increase (i.e. δ will be negative) when two classes are combined. Under these circumstances, property 6 *will* be satisfied.

27

Empirical Data

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



Histograms for the LCOM metric

Site	Metric	Median	Max	Min
A	LCOM	0	200	0
B	LCOM	2	17	0

Summary Statistics for the LCOM metric

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, 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 are likely to behave in less predictable ways than classes that have lower LCOM values. Such classes could be more error prone and more difficult to test and could possibly be disaggregated 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 changes, if necessary, at an earlier phase in the design cycle.

28

Summary

The Metrics Suite and Booch OOD Steps

The six metrics are designed to measure the three non-implementation steps in Booch's definition of OOD. Each metric is one among several that can be defined using Bunge's ontological principles. For example, the cardinality of the set of properties of an object (which will include both methods and instance variables could be defined as a metric. But inclusion in the proposed suite is influenced by three additional criteria: 1) ability to meet analytical properties 2) intuitive appeal to practitioners and managers in organizations and 3) ease of automated collection. Other comprehensive approaches may prove equally useful.

Reading down the columns of the table below, WMC, DIT and NOC relate to the first step (identification of classes) in OOD since WMC is an aspect of the complexity of the class and both DIT and NOC directly relate to the layout of the class hierarchy. WMC and RFC capture how objects of a class may "behave" when they get 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 LCOM metric relates to the packaging of data and methods within a class definition provides a measure of the cohesiveness of a class. Thus WMC, RFC and LCOM relate to the second step (the semantics of classes) in OOD. A benefit of having a suite of metrics is that there is the potential for multiple measures of the same underlying construct[31]. The RFC and CBO metrics also capture the extent of communication between classes by counting the inter-class couples and methods external to a given class, providing a measure of the third step (the relationships between classes) in OOD.

Metric	Identification	Semantics	Relationships
WMC	✓	✓	
DIT	✓		
NOC	✓		
RFC		✓	✓
CBO			✓
LCOM		✓	

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. Failing to meet Property 6 implies that a complexity metric could *increase*, rather than reduce, if a class is divided into more classes. Interestingly, the experienced OO designers who participated in this study found that memory management and run-time detection of errors are both more

[31] Another outcome of multiple measures is the statistical correlation between some metrics. For example, the RFC and WMC metric were highly correlated (Spearman rank correlation of 0.9) at both sites, while the NOC and LCOM had low correlation (less than 0.1). The median value of inter-metric correlations was 0.22 at Site A and 0.16 at Site B.

29

difficult when there are a large number of classes to deal with. In other words, their viewpoint was that complexity can increase when classes are divided into more classes. Therefore, satisfying Property 6 may not be an essential feature for OO software design complexity metrics. From a measurement theoretic standpoint, a metric that meets property 6, cannot be an interval or a ratio scale. This means that such a metric cannot be used to make judgments like "class A is twice as complex as class B", which limits its appeal. Thus, not satisfying property 6 is beneficial, rather than detrimental to widespread usage of the metrics.

The only other violation of Weyuker's properties is in the case of the DIT and LCOM metrics. The DIT metric fails to satisfy Property 4 (monotonicity) only in cases where two classes are in a parent-descendent relationship. This is because the distance from the root of a parent cannot become greater than one of its descendants. In all other cases, the DIT metric satisfies Property 4[32]. Also, under certain conditions of class combination, the LCOM metric can fail to satisfy this property as well.

Summary of Managerial Results

The data from two different commercial projects and subsequent discussions with the designers at those sites lead to several interesting observations that may be useful to managers of OOD projects. Designers may tend to keep the inheritance hierarchies shallow, forsaking reusability through inheritance for simplicity of understanding. This potentially reduces the extent of method reuse within an application. However, even in shallow class hierarchies it is possible to extract reuse benefits, as evidenced by the class with 87 methods at Site A that had a total of 43 descendants. This suggests that managers need to proactively manage reuse opportunities and that this metrics suite can aid this process.

Another demonstrable use of these metrics is in uncovering possible design flaws or violations of design philosophy. As the example of the command class with 42 children at Site A demonstrates, the metrics help to point out instances where sub-classing has been misused. This is borne out by the experience of the designers interviewed at one of the data sites where excessive declaration of subclasses was common among engineers new to the OO paradigm. These metrics can be used to allocate testing resources. As the example of the interface classes at Site B (with high CBO and RFC values) demonstrates, concentrating test efforts on these classes may have been a more efficient utilization of resources.

Using several of the metrics together can help managers and senior designers, who may be unable to review design materials for the entire application, to exercise some measure of architectural control over the evolution of an OO application. They could be by means of the WMC, DIT and NOC metrics to check whether the application is getting "top heavy" (i.e. too many classes at the root level declaring many methods) or using the RFC and CBO metrics check whether there are interconnections between various parts of the application that are unwarranted. The metrics values are likely to change as a project proceeds from design to implementation. If the system has been well architected, the class hierarchy will be stable, and the WMC, NOC, DIT metrics will reflect this. However, during implementation, new class coupling and communication may develop, affecting the CBO and RFC metric values. If implementation requires changes in the class

[32]It is interesting to note that other authors have also observed difficulties in applying this particular property of Weyuker's. For example, see [18.

30

definitions itself, the WMC and LCOM metrics will also change. Tracking these metrics through the life of the project, will provide managers with information to monitor OO systems evolution. As maintenance of the architectural integrity of an application becomes an important managerial responsibility, and this metrics suite could be used as a tool to meet this challenge.

Future Directions

The proposed OOD metrics have already begun to be used in a few leading edge organizations. Sharble and Cohen report on how these metrics were used by Boeing Computer Services to evaluate different OO methodologies [37]. Two implementations of an example system, one using responsibility based methodology and another using data driven methodology were analyzed using these six metrics. Based on this analysis, Sharble and Cohen recommended the responsibility based design methodology for use in the organization. This suggests an active interest in the practitioner community to use well-constructed metrics as a basis for managerial decision-making.

Another application of these metrics is in studying differences between different OO languages and environments. As the RFC and DIT data suggest, there are differences across the two sites that may be due to the features of the two target languages. However, despite the large number of classes examined (634 at Site A and 1459 at Site B), only two sites were used in this study, and therefore no claims are offered as to any systematic differences between the C++ and Smalltalk environments. This is suggested as a future avenue where OO metrics can help establish a preliminary benchmarking of languages and environments.

The most obvious extension of this research is to analyze the degree to which these metrics correlate with managerial performance indicators, such as design, test and maintenance effort, quality and system performance. The metrics proposed in this paper were used recently by Li and Henry who found that they explain additional variance in maintenance effort beyond that explained by traditional size metrics [27]. Another interesting study would be to follow a commercial application from conception to deployment and gather metrics at various intermediate stages of the project. This would provide insight into how application complexity evolves and how it can be managed through the use of metrics. These are highly promising avenues for research in the immediate future.

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, 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

31

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 following OO principles in their designs. 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 are candidates for redesign. 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. In general the idea is to use measurement to improve the process of software development.

This set of six proposed metrics is presented as the first empirically validated proposal for formal metrics for OOD. By bringing together the formalism 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 help to 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] Bilow, S.C., "Applying Graph-Theoretic Analysis Models to Object Oriented System Models", OOPSLA 92 Workshop on Metrics for Object Oriented Software Engineering Position Paper 1992.

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

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

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

[7] Card, D.N. and Agresti, W.W., "Measuring Software Design Complexity", *Journal of Systems and Software*, vol. 8, pp. 185-197, 1988.

[8] Cherniavsky, 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.

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

[10] 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.

[11] Coad, P. and Yourdon, E., *Object-Oriented Design*, Englewood Cliffs, NJ: Prentice-Hall, 1991.

[12] Coplien, J., "Looking Over One's Shoulder at a C++ Program", AT&T Bell Laboratories Technical Memorandum January 1993.

[13] deChampeaux, D. *et al.*, The Process of Object Oriented Design, *The Seventh Annual Conference on Object Oriented Programming Systems, Languages and Applications.*, 1992, Vancouver, Canada, pp. 45-62.

[14] Fenton, N.E., *Software Metrics, A rigorous approach*, New York: Chapman & Hall, 1991.

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

[16] Gustafson, D.A. and Prasad, B., Properties of Software Measures, in Denvir, T. *et al.* (ed.), *Formal Aspects of Measurement*, Springer-Verlag, 1991.

[17] Halstead, M., *Elements of Software Science*, New York, NY: Elsevier North-Holland, 1977.

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

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

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

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

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

[23] 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.

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

[25] Lake, A. and Cook, C., "A Software Complexity Metric for C++", Oregon State University Technical report 92-60-03, 1992.

[26] Lejter, M. *et al.*, "Support for Maintaining Object-Oriented Programs", *IEEE Transactions on Software Engineering*, vol. 18, pp. 1045-1052, 1992.

[27] Li, W. and Henry, S., Maintenance Metrics for the Object Oriented Paradigm, *First International Software Metrics Symposium*, 1993, Baltimore, MD, pp. 52-60.

[28] 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.

[29] McCabe, T.J., "A Complexity Measure", *IEEE Transactions on Software Engineering*, vol. SE-2, pp. 308-320, 1976.

[30] 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.

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

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

[33] 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.

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

[35] Rajaraman, C. and Lyu, M.R., "Some Coupling Measures for C++ Programs", *TOOLS USA 92*, vol. pp. 225-234, 1992.

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

[37] Sharble, R.C. and Cohen, S.S., "The Object Oriented Brewery: A Comparison of Two Object-Oriented Development Methods", *ACM SIGSOFT Software Engineering Notes*, vol. 18, pp. 60-73, 1993.

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

[39] Tagaki, K. and Wand, Y., An Object-Oriented Information Systems Model Based on Ontology, in F. Van Assche, B.M., C. Rolland (ed.), *Object Oriented Approach in Information Systems*, New York: Elsevier Science Publishers B.V. (North Holland), 1991.

[40] Tegarden, D.P. *et al.*, Effectiveness of Traditional Software Metrics for Object Oriented Systems, *25th Annual Conference on Systems Science*, 1992, Maui, Hawaii.

[41] 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.

[42] 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.

[43] 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.

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

[45] 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.

[46] Weber, R. and Zhang, Y., An Ontological Evaluation of Niam's Grammar for Conceptual Schema Diagrams, *Proceedings of the Twelfth International Conference on Information Systems*, 1991, New York, pp. 75-82.

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

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

[49] Wilde, N. and Huitt, R., "Maintenance Support for Object-Oriented Programs", *IEEE Transactions on Software Engineering*, vol. 18, pp. 1038-1044, 1992.

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