A Critical Analysis of Current OO Design Metrics

Tobias Mayer & Tracy Hall
Centre for Systems and Software Engineering (CSSE)
South Bank University,
London, UK

Abstract

Chidamber and Kemerer (C&K) outlined some initial proposals for language-independent OO design metrics in [10]. This suite is expanded on in [11] and the metrics were tested on systems developed in C++ and Smalltalk? .

The six metrics making up the C&K suite can be criticised for a number of reasons. This does not make them bad metrics; on the contrary the C&K work represents one of the most thorough treatments of the subject at the current time. However, the authors explicitly state,

> *"…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".*

This analysis will serve to make other researchers and practitioners aware of some of the problems that may arise from using these measures. As a by-product, the axioms of E. Weyuker [29] come under scrutiny in terms of their applicability to object-orientated metrics.

1. Introduction

Since the emergence of object-orientation as a prominent approach to software development, various academics and software practitioners have proposed metrics specific to the OO paradigm.

Probably the best known of these is the suite proposed by Chidamber and Kemerer [10, 11]. C&K attempt to establish a firm foundation for their metrics by adopting the ontology of Bunge [7, 8] as a theoretical basis, by evaluating each of the metrics using a set of axioms proposed by E. Weyuker [29] and by empirically testing the metrics on professional systems developed in C++ and Smalltalk? . However, despite C&K's attempts to produce theoretically sound metrics the suite has come under some criticism from the academic community.

These criticisms focus on the lack of a clear terminology and language binding [12] and a number of inadequacies in the meaningfulness of the metrics and in the method of evaluation used [15].

Despite the criticisms, and with little further empirical or theoretical evaluation, the C&K metrics have been incorporated into a number of software measurement tools and look set to become industry standards. Like McCabe's Cyclomatic Complexity metric [20] (also criticised for being based on a poor theoretical foundation [24]) the C&K metrics appear to have an intuitive appeal to software engineers.

Such intuitive appeal is a necessary criterion, but is by no means sufficient to warrant the use of a metric. We use a set of criteria, drawn from the work of Fenton and Pfleeger [13] and from our own research [18, 19a, 19b], to explore and analyse the C&K metrics. Our criteria, expressed as a set of requirements, state that any new object-orientated design metric should:

1. be language-independent
2. measure a single attribute
3. be defined in OO terms
4. be firmly rooted in measurement theory [5, 13]
5. be collectable by static analysis
6. be automatable
7. have intuitive appeal to software engineers
8. be genuinely useful to project and business managers

Our analysis indicates that the C&K metrics do not satisfy all of these requirements; their use as quality indicators is thus in doubt.

The remainder of this paper is structured as follows: Section Two briefly outlines the foundations on which the metrics suite was built; Section Three addresses issues relating to the use of Weyuker's axioms as a validation criterion, in particular the issues of complexity and class combination are considered. Section Four is a general criticism concerning the concepts of coupling and cohesion as applied to object-orientated metrics. Section Five is a detailed analysis of each of the six metrics; Section Six offers an example of inaccurate data interpretation and Section Seven offers some concluding remarks.

2. Foundations

2.1 Theoretical Foundation

The concept of objects used by C&K is founded on the ontological principles proposed by M. Bunge in his "Treatise on Basic Philosophy" [7, 8]. This follows from the work of other researchers, including Y. Wand, e.g. [27] and R. Weber, e.g. [28], who have found Bunge's generalised principles to be ideally suited to the OO paradigm.

2.2 Theoretical Validation

Each of the metrics is validated against a sub-set of Weyuker's axioms for software complexity measures. The authors acknowledge that this rule set has received a number of criticisms, e.g. [13, 9] but they chose to use it because it *"is a widely known formal analytical approach"*.

2.3 Empirical Validation

To ensure the usefulness of the metrics C&K incorporate the experiences of professional software developers, which are expressed as 'viewpoints' for each metric. These viewpoints are considered when interpreting the results produced by the empirical tests.

3. The Use of Weyuker's Axioms as a Validation Criterion

3.1 Measuring Complexity

There is a widespread misconception in the software measurement field that any design metric is a complexity metric. By validating their metrics against Weyuker's axioms - which are designed specifically for evaluating software complexity measures - C&K are implicitly stating that all the metrics in the suite are complexity metrics. However, C&K do not define complexity and it appears to have a different definition for each metric.

Intuitively, complexity (by any definition) is thought of as undesirable in a program. In measuring the complexity of an algorithm a high complexity value is clearly undesirable and it would be a goal of the designer or programmer to simplify the algorithm or split it into smaller units, each with a lower complexity value. In keeping with this intuitive view C&K organise their metrics such that a low value implies low complexity, and *visa-versa*. But, a low 'complexity' value for some of these metrics is not necessarily desirable, for instance it would not be preferable for all classes - or even most classes - in a system to have a DIT (depth of inheritance) value of 0 (i.e. be root classes). It can be argued that classes with a high DIT value, whether it is desirable or not, are indeed more 'complex' as it is harder to understand and maintain a class that inherits data and behaviour from other classes. This same argument however does not hold for the NOC (number of children) metric. It is again undesirable for all classes in a system to have a NOC value of 0, but regardless of the number of children a class has its 'complexity' remains constant. There is no way of knowing how many children a class has by examining the class itself. The NOC value will change as more child classes are added, but there is no internal change to the class itself. Furthermore, in different systems the same (re-used) class will have different values for NOC. It is illogical and unintuitive that the complexity of a thing can change if the thing itself remains the same.

A design that makes good use of the inheritance mechanism, although perhaps appearing more complex on the surface (due to the high metric values), would

probably be less complex than a design where the majority of classes were root classes that had few or no children. In the latter case it is likely that there would be code duplication, a greater need for multi-way case statements and generally larger, non-unified classes.

3.2 Combination of Classes

The fourth, fifth and sixth of Weyuker's properties are concerned with the combination of two classes. To evaluate the DIT metric against property four, 'Monotonicity' (i.e. 'The metric for two classes in combination can never be less than the metric for either of the component classes') C&K state three possible relationships between two classes, P and Q.

i) P and Q are siblings,
ii) Q is a descendant of P
iii) P and Q are neither siblings nor descendants of each other.

In case ii) the DIT metric fails to satisfy property four. However our concern is not whether or not the property is satisfied, rather it is with the whole concept of 'class combination'.

The second relationship (above) can be further classified as:

a) Q is an *immediate* descendant of P
b) Q is a *non-immediate* descendant of P

C&K do not give adequate consideration to this latter relationship, which may explain why the problems of combination we describe below were overlooked.

3.2.1 Combining two classes in a non-descendant, non-sibling relationship

In order to combine the two classes P and Q in case iii) C&K change the hierarchical structure from a single inheritance structure to a multiple inheritance structure. Putting aside the fact that many OO languages do not support multiple inheritance, this re-structuring seems unacceptable as it alters the entire design pattern and adds to the overall complexity of the system. In order to avoid it, other combinations need to be made to the structure (see figures 1-3, below).

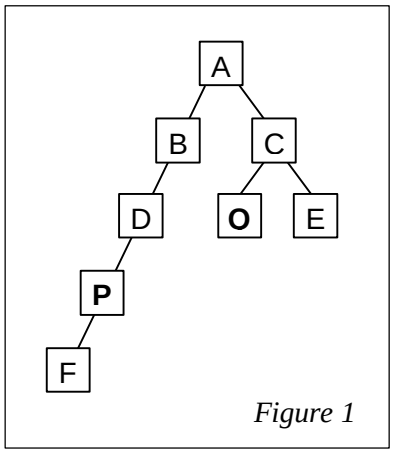

Figure 1

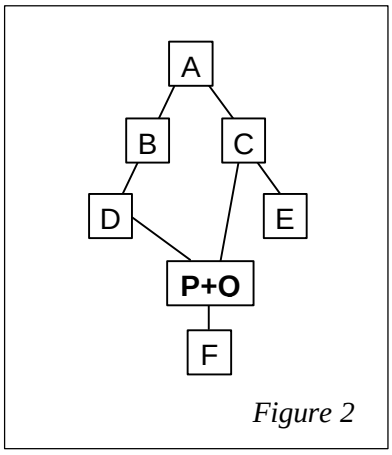

Figure 2

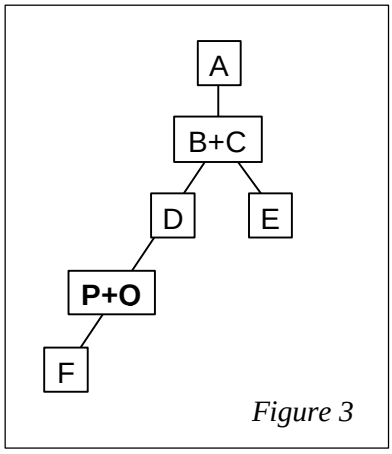

Figure 3

Figure 1 shows a hierarchy with classes P and Q in a non-sibling/non-descendant relationship.
Figure 2 shows the C&K method of combining P and Q by changing the structure to one of multiple inheritance. The new class (P+Q) now has two parent classes: D and C.
Figure 3 shows the combination of P and Q whilst retaining the single inheritance structure. Note that classes B and C also need to be combined to achieve this.

3.2.2 Combining Two Classes in a Non-immediate Descendant Relationship

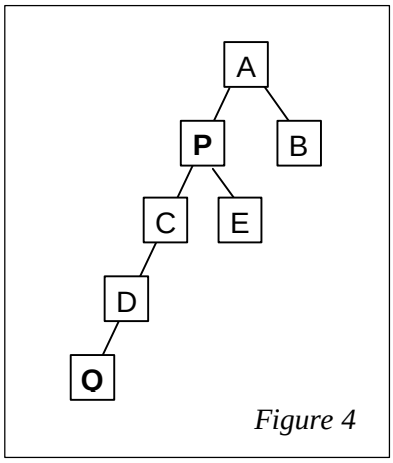

Figure 4

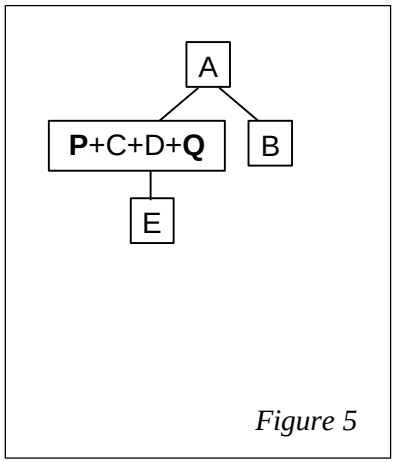

Figure 5

Figure 4 shows P and Q in a (non-immediate) 'descendant-of' relationship.

Figure 5 shows the only way of combining P and Q without either class (or any other class in the hierarchy) losing its inherited properties. Moving Q up to P's position would result in Q losing the properties inherited from C and D. Moving P down to Q's position would result in C and D losing the properties inherited from P. (C&K offer no alternative way of combining two classes in this kind of relationship.)

3.2.3 Conditions for the Combination of Two Classes in a Hierarchy

Two conditions for the combination of two classes then emerge.

1. No class in the hierarchy should lose its inherited properties as a result of the combination.
2. A single inheritance structure should remain as such after the combination.

The above examples illustrate how combination in terms of classes has very different implications than it does for traditional (i.e. function-orientated design) measures. It may be that combination is not a valid operation between some classes, e.g. in the examples given above where it is necessary to combine more than the two required classes into one class or to make additional combinations within the hierarchy. If so, this renders Weyuker's fourth, fifth and sixth properties invalid and raises doubts as to the suitability of using properties designed to evaluate function-orientated metrics on class/object-orientated metrics.

3.3 Invalid Validation

The use of Weyuker's properties as a method of validation produces further inconsistencies. Weyuker's sixth property, 'Interaction Increases Complexity states that when two components, *A* and *B*, are combined the resulting complexity of the new component *AB* can be greater than the sum of the two separate components. The intention of this property is to ensure that dividing the problem space into a set of smaller, more manageable problem spaces can reduce the overall complexity of the system. This view is supported by the fourth property, 'Monotonicity', which states that the metric for two classes in combination can never be *less* than the metric for either of the component classes. It is cause for concern that none of the C&K metrics satisfies property six, and two of them, DIT and LCOM, fail to satisfy property four. The implication of this is that class designs become *less* complex the more you combine the classes, resulting - ultimately - in a design with just one class. In terms of OO design this is clearly ludicrous (although in their summary of the analytical results C&K present an argument against this, but fail to take the implication to its logical conclusion).

Rather than concluding that the metrics themselves are inadequate it seems more likely that either the rule set is inappropriate for validating object-orientated metrics (the conclusion reached in section 3.2) or that the C&K metrics are not measuring complexity - by the definition(s) intended by Weyuker.

3.4 A Note of Caution

Until the term 'complexity' is more precisely defined it seems sensible to avoid it as a 'catch-all' attribute to measure, and define each individual measure in terms of what it is *actually* measuring. For NOC, (as an example) this would be 'influence on system' where a value of 0 would imply 'no influence' and higher values would imply increasingly more influence. As for Weyuker's axioms, until the criticisms (referenced above) are addressed - and possibly even after that - they are probably best left in the realm for which they were devised, namely function-orientated design, and even then only used as a supplement to the validation principles of measurement theory.

4. A General Criticism of Object-Orientated Metrics

An important criticism, not only of the C&K metrics but of many of the OO measures that have been proposed, is that of the terminology used and the underlying thinking that accompanies it. The term complexity is commonly used without any clear definition being given of its meaning in relation to OO systems. The same is true of other concepts/terms used in the language of traditional software measurement. Two such terms that play an important role in conventional software design and measurement are coupling and cohesion.

Coupling and cohesion are, in a sense, artificial attributes, devised for structured programming due to the absence of clear relationships between discrete components on the one hand, and the absence of rules relating to the connection of parts contained in a single component on the other hand. OOP has a number of design fundamentals, or mechanisms not present in structured programming that renders these attributes redundant. However, because of their historical importance, academics and practitioners interested in devising new measures for the OO paradigm have attempted to translate the meanings of coupling and cohesion to suit the new measures. We believe this to be a mistake and one that has caused OO measures to be imprecise, overly complicated and inadequate. On one hand it indicates that many metricians are not recognizing the unique aspects of OO design but are assuming it to be simply an extension of structured programming techniques which can then be measured using conventional measures such as LOC or McCabe's Cyclomatic Complexity[1]. On the other hand it shows lack of understanding of an important principle of measurement: that we should start with an entity and identify the attribute of interest, not start with an attribute and try to make it apply to the entity.

---

[1] For example, C&K suggest that developers approach the task of writing a method as they would a traditional program and, that being the case, methods can be measured using conventional complexity measures.

5. The Six Metrics

5.1 WMC Weighted Methods per Class

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

where $c_1 \ldots c_n$ is the complexity of methods $M_1 \ldots M_n$ of a given class. If all method complexities are equal to 1 then WMC = *n*, the number of methods.

WMC breaks a fundamental rule of measurement theory: that a measure should be concerned with a single attribute [13]. If all method complexities equal one then WMC supplies a count of the methods. Once weighting, in the form of a complexity value, is added this count is lost. A class with one method that has a complexity value of ten would give the same WMC value as another class with ten methods, each having a complexity value of one. The viewpoints for WMC are largely concerned with method count, so the addition of a complexity value seems an unnecessary complication. It is worth noting that the McCabe measure of the same name is defined as a count of all methods defined in a class, in which case the word 'weighted' is superfluous.

5.2 DIT Depth of Inheritance Tree

DIT = ?C?-1

where C = set of all classes in the linear branch from *c* to the root.
In the case of multiple inheritance DIT = max(?C1?…?Cn?) -1.

In a single inheritance structure DIT for a class *c*, is equivalent to the count of all classes *c* inherits from.

Two of the viewpoints for this metric refer to increased complexity due to more methods being inherited and more classes being involved, the third is concerned with potential reuse of inherited methods. DIT is a measure of how many ancestor classes affect the measured class so it creates an inconsistency when the definition of this metric in a multiple inheritance structure is given as DIT = $max(?C_1? \ldots ?C_n?)$ -1, in other words the longest path from class *c* to the root class. This would not indicate the number of classes involved or the number of methods inherited, e.g. a single class may have a DIT value of two but have half a dozen (or more) ancestor classes. Excessive use of multiple inheritance is generally discouraged in OO design so an inheritance measure should be able to detect this anomaly.

5.3 NOC Number of Children

NOC = ?*C*?
where *C* = set of all classes which immediately extend a class *c*.

The NOC metric is concerned with potential reuse through inheritance but in the same way that DIT fails to give the full picture of number of ancestor classes the NOC metric fails to count all the descendants of a class. Only counting immediate subclasses may give a distorted view of the system, as the following example illustrates:

Example 5.3a

A hierarchy is structured thus: Class A is the root class, class B extends A, class C extends B and classes D, E, F, G and H extend C. Both A and B have a NOC value of one, but a total of seven classes inherit A's properties and a total of six inherit B's properties. As the hierarchy grows bigger so, potentially, does the discrepancy.

The definition of NOC does not satisfy viewpoint 1 (reuse) or viewpoint 3 (influence on the design) [11, p.485], both of which support the idea of counting all of a class's descendants.

Both DIT and NOC are useful measures but are not sufficient to fully assess the quality of a hierarchy or to satisfy the respective viewpoints. Additional measures are required to count all of a class's ancestors (i.e. in a multiple inheritance system) and all of its descendants.

5.4 CBO Coupling Between Object Classes

CBO = ?*C*? where *C* = set of all classes to which a class *c* is coupled.

where 'coupled' is defined as: two classes are considered coupled when a method of one references a method or field of the other. CBO is a count of all such references for a single class.

C&K state that

"…two objects are coupled when methods declared in one class use methods or instance variables of the other class."

Coupling is therefore a property of pairs of classes, but in terms of CBO it is defined for a single class as

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

This does not clearly indicate the direction of the coupling, whether suppliers or clients (or both) should be counted. The viewpoints [11, p.486] however, indicate that CBO is a count of suppliers. C&K further state that their definition of coupling also applies to

"coupling due to inheritance",

but do not make it clear if all ancestors are automatically coupled to the measured class, simply because they are ancestors, or if the measured class has to explicitly access a field or method in an ancestor class for it to count. In general, the

definition for this metric is ambiguous, which makes its application impossible for anyone other than the original proponents.

Hitz and Montazeri criticise the CBO metric in [15] for failing to distinguish between different coupling strengths, claiming that the empirical relation system established by C&K for the attribute of coupling is insufficient[2]. A similar criticism is given by Binkley and Schach in [6]. Both papers suggest that a single couple should contribute more or less to an overall coupling metric, depending on circumstances such as the number of parameters passed, what is being accessed (field or method) and whether the class is an ancestor or not.

As mentioned in Section 4, the term 'coupling' is perhaps not an appropriate one to use in describing an OO design, where the relationships between components (which are classes) are more clearly defined than in structured programming.

5.5 RFC Response For a Class

RFC = ? RS? where RS = {M} ? all i {R i}

and where {M} = set of all methods of a class and {R i} = set of all methods recursively called by method i.

RFC aims to count (for a class) all methods that can potentially be executed in response to a message received by an object of that class. Due to the practical considerations of collecting this data C&K recommend that only one level of nesting should be considered i.e. should only count the methods declared in a class plus those methods called from within the class's methods. This gives a distorted picture as a single method call often has deeply nested 'call-back'. If C&K intend the metric to be a measure of the methods in a class plus the methods called then the definition should be re-defined to reflect this.

As all methods must be defined within a class (RFC ignores functions such as 'printf' in C++) a count of all supplier classes (i.e. direct and indirect suppliers) would perform a similar function and would satisfy the viewpoints [11, p.487] for this metric. It would also be an easier measure to collect data for.

5.6 LCOM Lack of Cohesion in Methods

if ? P? > ? Q? LCOM = ? P? ? ? Q? ,

else LCOM = 0

P = set of all method pairs of class c with zero similarity,

Q = set of all method pairs of class c with non-zero similarity,

where similarity of a pair of methods is defined as the number of instance variables that are referenced by both methods.

LCOM is defined 'in reverse' to be in keeping with C&K's concept of a low value being less complex and *visa-versa*. Hitz & Montazeri comprehensively cover the

---

[2] Hitz and Montazeri's own framework for class and object coupling is described in [14].

weaknesses of this metric in [14, 15], a critique with which we fully concur, and propose a new formulation for LCOM, restated here in non-mathematical terms.

LCOM for a single class is a count of 'method clusters', where any method in a cluster must access at least one instance variable (directly, or via an access method) that is also accessed by at least one other method in the cluster. The number of clusters, c, in a class will be 1<=c<=m, where m = the number of methods in a class.

We suggest that the name (which is confusing anyway) is now a misnomer, and what the Hitz and Montazeri version of LCOM is actually measuring is an important aspect of encapsulation, namely the *unity* of a class. The importance of focusing on OO-specific attributes, rather than those attributes that have been found to be useful for function-orientated systems was touched on in section 4. The theme will be developed in a later paper.

6. Interpretation of Data

In the interpretation of the data taken for the NOC metric C&K comment on the high number of classes (73% at one site and 68% at the other) having NOC=0 and suggest this indicates poor use of inheritance. What does not seem to be taken into consideration is that this percentage will almost always be large for any design that does make use of inheritance. For instance, a structure where every class (except the leaf-node classes) has two children will result in >50% of classes where NOC=0 and a structure where every class (except the leaf-node classes) has three children will result in >66% of classes having no children. This percentage grows the more children the childbearing classes have. In general, the formula for calculating % of classes where NOC=0 for regular inheritance trees is:

$$c^n \,/\, \sum_{i=0}^{n} c^i$$

where c = number of children per class and n = depth of nesting; e.g. for a hierarchy with a maximum depth of 4 where all classes have 2 children:

$2^4 / (2^0 + 2^1 + 2^2 + 2^3 + 2^4) = 16/31 = 0.516$ or 51.6%

The times a design will have less than 50% of classes with NOC=0 will be in cases such as figure 6 which does not indicate particularly effective use of inheritance.

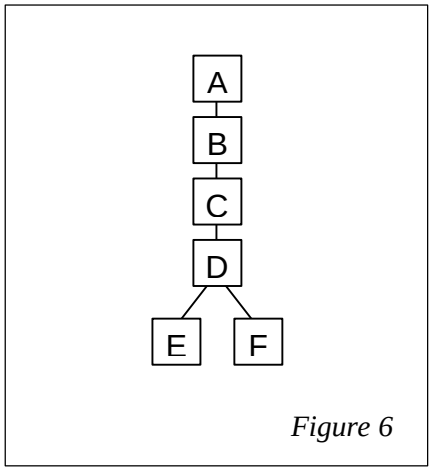

Figure 6

This example illustrates the caution that needs to be observed when interpreting the data produced by a metric, particularly with a new (i.e. object-orientated) metric that we are not familiar with. Inaccurate interpretation can result in the wrong actions being taken to 'fix' the problem. This in turn results in wasted person-hours and, of course, costs for the organisation involved.

7. Concluding Remarks

In general the C&K metrics suffer from unclear definitions and a failure to capture OO-specific attributes. The attributes of data-hiding, polymorphism and abstraction are not measured at all and the attributes of inheritance and encapsulation are only partially measured. The focus on coupling, cohesion and complexity restricts the usefulness of these measures and the use of Weyuker's axioms limits their validity.

As a by-product of this analysis Weyuker's axioms are shown to be inadequate to validate object-orientated metrics.

An equally detailed analysis [19a] has been carried out on the metrics proposed by the MOOD project [1, 2, 3, 4]. A more general critique of current OO metrics [19b] also includes the metrics of Lorenz/Lorenz & Kidd [16, 17], McCabe [21, 22], and other individual object-orientated metrics and metrics frameworks, e.g. [14, 23, 25, 26]. Our long term plan is to utilise, and build on, the best of the existing work in order to propose a set of basic, language-independent design measures that are theoretically sound as well as being acceptable, understandable and useful to all sections of the software engineering community.

References

[1] F. B. e Abreau and R. Carapuça, "Candidate metrics for object-orientated software within a taxonomy framework", *Proc. AQUIS'93 Conference,* Venice, Italy, Oct. 1993

[2] F. B. e Abreau and R. Carapuça, "Object-orientated software engineering: measuring and controlling the development process", *Proc. 4th Int. Conf. On Software Quality,* McLean, VA, USA, Oct. 1994

[3] F. B. e Abreau, M. Goulão and R. Esteves, "Toward the design quality evaluation of object-orientated software systems", *Proc.* 5th *Int. Conf. On Software Quality, 1995*

[4] F. B. e Abreau and W. Melo, "Evaluating the impact of object-orientated design on software quality", *Proc. 3rd International Software Metrics Symposium (METRICS'96),* IEEE, Berlin, Germany, Mar. 1996

[5] A. L. Baker, J. M. Bieman, N. Fenton, D. A. Gustafson, A. Melton and R. Whitty, "A philosophy for software measurement", *J. Systems Software, 1990; 12: pp277-281*

[6] A. B.Binkley and S. R. Schach, "Impediments to the effective use of metrics within the object-orientated paradigm", in *Proc. OOPSLA'96,* UK, 1996

[7] M. Bunge, *Treatise on basic philosophy: Ontology I: The furniture of the world.* Boston: Riedel, 1977

[8] M. Bunge, *Treatise on basic philosophy: Ontology II: The world of systems.* Boston: Riedel, 1979

[9] J. C. Cherniavsky and C.H. Smith, "On Weyuker's axioms for software complexity metrics", *IEEE Transactions on Software Engineering,* 17(6), 1991, pp. 636-638.

[10] S. R. Chidamber and C. F. Kemerer, "Towards a metrics suite for object-orientated design*"*, *Proc. Sixth OOPSLA Conference*, 1991, pp. 197-211.

[11] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object-orientated design*"*, *IEEE Transactions on Software Engineering,* 20(6), 1994, pp. 476-493.

[12] N. I. Churcher and M. J. Shepperd, "Comments on ' A metrics suite for object orientated design'", *IEEE Transactions on Software Engineering,* 21(3), 1995, pp. 263-265.

[13] N. Fenton & S Pfleeger, "Software Metrics: A rigorous and practical approach", Thompson, 1996

[14] M. Hitz and B. Montazeri, "Measuring coupling and cohesion in object orientated systems", *Proc. International Symposium for Applied Corporate Computing (ISACC '95), Monterrey, Mexico,* Oct. 25-27, 1995.

[15] M. Hitz and B. Montazeri, "Chidamber and Kemerer's metrics suite: A measurement perspective", *IEEE Transactions on Software Engineering,* 22(4), 1996, pp. 267-271.

[16] M. Lorenz and J. Kidd, "Object-orientated software metrics", Prentice Hall Object Orientated Series, Englewood Cliffs, NJ 07632, 1994.

[17] M. Lorenz, "Object-Oriented Software Metrics", listed at: http://www.hatteras.com/metr_des, 1998.

[18] T. G. Mayer, "A fresh look at object-orientated design metrics", *Procs. UKSMA 10th Anniversary Conference*, London, Oct.1998

[19a] T. G. Mayer and T. Hall, "Measuring OO systems: a critical analysis of the MOOD metrics", *unpublished*, South Bank University, February 1999

[19b] T. G. Mayer and T. Hall, " Object-orientated design measurement: a critique", *unpublished*, South Bank University, March 1999

[20] Thomas J. McCabe, "A complexity measure", IEEE Transactions on Software Engineering, SE-2(4), 1976, pp. 308-320.

[21] T. J. McCabe, L. A. Dreyer, A. J. Dunn and A. H. Watson, "Testing an object-orientated application", The Journal, October 1994, pp. 21-27.

[22] McCabe & Associates, "McCabe Object-Oriented Software Metrics", listed at: http//:www.mccabe.com/features/complex, 1998

[23] M. J. Shepperd and N. I. Churcher, "Towards a conceptual framework for object orientated software metrics", Bournemouth University, 1994

[24] M. J. Shepperd, "A critique of cyclomatic complexity as a software metric", *Software Engineering Journal*, March 1988, pp. 30-36.

[25] D. P. Tegarden and S. D. Sheetz, "Object-orientated system complexity: an integrated model of structure and perceptions", *Proc. OOPSLA '92 Workshop on Metrics for OO Software Development*, 1992

[26] D. P. Tegarden, S. D. Sheetz and D. E. Monarchi, "A software complexity model of object-orientated systems", *University of Denver*, 1992

[27] Y. Wand, "A proposal for a formal model of objects", in *Object-Orientated Concepts, Databases and Applications*, W. Kim and F. Lochovsky, Eds. Reading, MA: Addison-Wesley, 1989

[28] R. Weber and Y. Zhang, "An ontological evaluation of Niam's grammar for conceptual schema diagrams", in Proc. Twelfth International Conference on Information Systems, New York, 1991, pp 75-82

[29] E. Weyuker, "Evaluating software complexity measures", *IEEE Transactions on Software Engineering*, vol. 14, 1983, pp. 1357-1365