A Data Model for Object-Oriented Design Metrics

Joe Raymond Abounader David Alex Lamb

October 1, 1997
External Technical Report
ISSN-0836-0227-
1997-409

Department of Computing and Information Science
Queen's University
Kingston, Ontario, Canada K7L 3N6

Version 1.2
Document prepared October 1, 1997

$Id: metr.tex,v 1.10 1997/10/01 16:22:55 dalamb Exp $

Contents

List of Figures

List of Tables

1 Introduction

A common approach to improving the run-time performance of a software system is to measure various run-time properties of the components of the system to find "bottlenecks" that account for the majority of the time cost of the system. By analogy, a promising approach to reducing life-cycle costs of software development has been to measure properties of software artifacts to find "complexity" bottlenecks that account for the majority of the difficulty of developing or maintaining the software.

With object-oriented systems, the structure (and thus some of the complexity) of much of the code should reflect similar structure in analysis and design artifacts. Thus, it should in principle become possible to more accurately predict development and maintenance costs for object-oriented systems. Many object-oriented metrics have been proposed, but there is as yet no consensus on which are best, and most have not been well-validated.

Our contribution is to provide a data model for a database of design information from which most of the proposed OO metrics can be computed. Combined with suitable tools to extract basic design information from designs (or, perhaps, from code), such a database would make it easier to compare metrics and validate a large number of metrics from the same basic data.

To prepare for presenting the data model, we discuss the differences between traditional and OO metrics, present a list of the most known sets of OO metrics that have been derived so far, and show some of the various guidelines along which OO metrics are classified.

2 Traditional versus Object-Oriented Metrics

The OO paradigm for software development differs from the traditional procedural paradigm, which suggests that OO metrics should differ from their traditional counterparts. However, there has been no universal agreement as to the significance of the difference(s) between the metrics, with opinions ranging from a total rejection of the traditional metrics to more positive, yet cautious approaches to making use of them within the OO methodologies.

Opponents of the use of traditional metrics within the OO paradigm argue that such metrics were originally designed to go along procedural methodologies and languages, and therefore fail to capture such concepts as inheritance and polymorphism which are unique to the OO paradigm[AKM94, LHKS95, LH93]. Henderson-Sellers[HS91] notes that "the traditional and OO paradigms

differ in that the traditional paradigm requires more effort during the coding and maintenance phases than its OO counterpart", and that "the OO methodologies put more emphasis on the earlier stages of analysis and design", thus implying that a new set of metrics is needed to reflect those differences. Moreau and Dominick[MD89] point out that "many existing metrics that have been utilized within conventional programming environments are inappropriate for evaluating object-oriented systems in certain circumstances". They mention the traditional lines of code (LOC) metric as an example that would be "a very poor indicator of developmental complexity [a measurement pertaining to OO] within object-oriented systems since only a small part of the code is likely to be unique to an object [due to inheritance-related reuse of code]". Other traditional metrics such as Halstead's "Software Science" metrics[Hal77] and McCabe's "cyclomatic complexity"[McC76] are judged as needing to be "recalibrated to OO systems to be effective"[LK94].

On the other side of the debate, some researchers and practitioners think that there is still hope for the set of traditional metrics in the world of object-orientation, especially since those metrics have already been defined, well-tested and calibrated. Tegarden[TSM92] argues that traditional metrics "are well understood by researchers and practitioners". He further presents the results form experiments he conducted on four software systems[TSM92], using the traditional metrics of source lines of code, Halstead's "software science" metrics[Hal77], and McCabe's "cyclomatic complexity"[McC76]; he notes that the use of inheritance and/or polymorphism should decrease the complexity of an OO system, and the results from the experiments show that this is captured by the traditional metrics. However, they conclude that "additional metrics are required to measure all aspects of OO systems". A similar experiment was performed by Coppick and Cheatham[CC92] who also applied Halstead's[Hal77] and McCabe's[McC76] metrics to objects and found that the results were "intuitively reasonable". Finally, Binkley and Schach[BS96] observe that "there is a tendency for some developers to reinvent the wheel as they unnecessarily redefine well-known metrics in purely object-oriented terms ..."; they list Chidamber and Kemerer[CK94] as some of the example researchers who have "reinvented the wheel" in their definition of new metrics; for instance, the lack of cohesion metric (LCOM) was (re-)defined specifically for OO systems, while "it has been shown that the cohesion of a class can be expressed in terms of classical cohesion".

Our perspective on this issue is that it does not matter, from our narrow focus. We propose to develop a design database from which many different

metrics can be computed; in so far as both traditional and new metrics can be computed from the same information, we can represent all of them without prejudice. We expect others to validate and compare the metrics; our data model might serve as a useful adjunct to a major comparison among metrics.

3 An Overview of Existing Object-Oriented Metrics

In this section, we show some of the most widely cited OO metrics, together with their pros and cons (derived from both the literature and from our own reflections).

3.1 Moreau and Dominick

Moreau and Dominick[MD89] were some the earliest researchers whose work we surveyed who defined metrics for the OO paradigm[1].i Only three metrics were derived:

1. Message vocabulary size (*MVS*): The number of different types of message sent by a particular object; it is related to the number of functions that the programmer must be familiar with to comprehend the object.

2. Inheritance complexity (*IC*): "Compound" and "multilevel" inheritance contribute to the complexity of building, understanding, and maintaining an object. They believe that size of the inheritance tree is a "simple" approximation to such a metric.

3. Message domain size (*MDS*): The number of distinct procedures within the object that manipulate its state, thus the number of "types of messages to which an object will respond".

The three defined metrics need clarifications, such as what exactly is meant by "sending messages", and how the metrics are to be computed. The metrics were not tested nor validated, a fact also acknowledged by the authors themselves. We can, however, draw some parallels between these metrics and the three OO software quality abstractions of coupling, inheritance complexity,

[1]In fact, we obtained a copy of an earlier work by Morris [Mor89], in which he defined metrics for OO environments, just before printing; thus his work could unfortunately not be included in this report.

and cohesion. Hence, this research could be considered as the basis for an expanded and more sensitive collection of metrics for the OO paradigm.

3.2 Chidamber and Kemerer

The Chidamber and Kemerer[CK94] (C & K) metrics suite is the most cited set of metrics we have found, and also the most criticized. The original suite was derived in 1991 in [CK91], and the popularity of the paper prompted a newer version in 1994. There are six metrics in the suite, all of them being design metrics:

1. Weighted Methods per Class (WMC). Consider a class C1 with methods $M_1, \ldots, M_n$ that are defined in the class. Let $c_1, \ldots, c_n$ be the complexity of the methods, then

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

   If all method complexities are equal to unity, then WMC = n, or the number of methods. The authors add that the complexity metric to be used here was deliberately not specified "to allow for the most general application of the metric".

   It is thought that in OO systems, methods are, in general, small enough so that the complexity of each could be considered as equal to unity. Henderson-Sellers[HS96] notes that if c_i is taken as equal to V(G), the cyclomatic complexity[McC76], then for class i, $WMC = \sum_{j=1}^{m} V_{ij}(G)$, where m is the number of methods in class i; but if $c_i = 1$, then $WMC \equiv NOM$ ($number\ of\ methods$).

2. Depth of Inheritance Tree (DIT). Depth of inheritance of a class is its depth in the inheritance tree; if multiple inheritance in involved, then the depth of the class is the length of the maximum path from the node representing the class to the root of the tree.[2]

3. Number of Children (NOC). Number of immediate subclasses subordinated to a class in the class hierarchy.

[2]We assume that the depth of the root class is 0, since the authors show a 0 as the minimum DIT value in one of the experiments; also, in the interpretation of DIT, it is said that "DIT is a measure of how many ancestor classes can potentially affect this class".

4. Coupling Between Object classes (CBO). CBO for a class is a count of the number of other classes to which it is coupled, where coupling is defined as "any evidence of a method of one object using methods or instance variables of another object".

5. Response For Class (RFC). RFC = |RS|, the size of the Response Set of a class, defined as the set of methods in the class together with the set of methods called by the class's methods[3].

6. Lack of COheshion in Methods (LCOM). LCOM is a count of the number of method pairs whose similarity is zero, minus the count of method pairs whose similarity is not zero. where similarity of a pair of methods is the number of joint instance variables[4] used by both methods.

   e.g.: Consider a class C with 3 methods M_1, M_2, and M_3. Let $I_1 = \{a, b, c, d, e\}$, $I_2 = \{a, b, e\}$, and $I_3 = \{x, y, z\}$, where "I_i" is the set of instance variables used by method "M_i". Here, we have two disjoint sets: $I_1 \cap I_2$ $(= \{a, b, e\})$ and I_3. We have one pair of methods who share at least one instance variable (I_1 and I_2). So LCOM = 2 -1 = 1.

As pointed out previously, the C & K metrics suite is one of the most criticized, perhaps due to its popularity.

Churcher and Shepperd[CS95a, CS95b] point out that definitions of some of the basic direct counts are imprecise, which could have an impact on the defined metrics. Their main concern lies with the number of methods in a class count, used directly in the computation of WMC and indirectly in LCOM. Due to the various possibilities in counting the methods, the result could vary dramatically, leading to confusion. The various possibilities result from the decision on whether to count inherited methods as belonging to the class, whether to count methods with the same name (but different signatures), whether operators should be considered in the count, and so on. An example of a C++ class shows that the number of methods could vary from 12 to 37, depending on which combination of counting rules is followed. The conclusion is that "it is vitally important to precisely specify the mapping form a language-independent set of metrics to specific programming languages".

In a reply to these remarks, Chidamber and Kemerer[CK95] clarify their position by stating that "the methods that require additional design effort and are defined in the class should be counted, and those that do not should not".

[3]Note the similarity to Moreau and Dominick's third and first metrics, respectively.

[4]Joint instance variables between two methods are those variables referred to by both methods.

A more rigorous criticism comes from Hitz and Montazeri[HM96a], who agree with the remarks made by Churcher and Shepperd[CS95a, CS95b], but focus on CBO and LCOM. They argue that CBO is not a sensitive enough measure of coupling, since it considers all couples to be of equal strength. There are, however, different factors which should discriminate between couples of classes. For instance, access to instance variables should constitute stronger coupling than pure message passing, as does message passing with a wide parameter interface vs. one with a slim interface. They show that LCOM exhibits an anomaly, namely that the same value from the metric is computed for different classes that intuitively appear to have different cohesion levels. Hitz and Montazeri propose a graph-theoretic formulation of the LCOM metric in order to correct the anomaly and to differentiate among ties in cases where LCOM = 1.

Henderson-Sellers[HS96] also studies the LCOM measure, and finds that while "a large value suggests poor cohesion, a zero value does not necessarily indicate good cohesion"; thus the LCOM measure is not sensitive enough for cases of high cohesion. A counter example is used, based on the example presented by C & K[CK94] (refer to the example in the definition of LCOM). If a new method M_4, using the set of variables $I_4 = \{x, y, z, d\}$, is added to the class, the LCOM measure will be 0; this suggests a very high cohesion in the class, while this would not intuitively be the case, with M_1 and M_2 representing a cohesive pair of methods, and M_3 and M_4 another. On another note, he adds that RFC and CBO are not "orthogonal".

In the WMC metric, a weight for each method in a class is to be computed. This is generally thought to be the complexity of the method; however, Chidamber and Kemerer did not specify how to come up with the weight, as pointed out by Kalakota et al.[KRW93]

On the other hand, Li and Henry[LH93] conducted their own empirical experiments, and showed that by using a combination of five of the six C & K metrics,[5] along with some newly defined metrics (see Section 3.3), it is possible to predict the maintenance effort required for a software system.

In another study, Basili et al.[BBM96] show that five of the six C & K metrics were useful in predicting class fault-proneness during the high and low level design phases of the life cycle[6]. The metrics were found to be "statically

[5]CBO was omitted.

[6]The exception sixth metric was LCOM, which showed no significant relationship with fault-proneness.

independent" and did not capture a great deal of redundant information[7]. They conclude that the C & K metrics proved to be better predictors than "the best set of the traditional metrics", which are only available at the latter phases of the software life cycle.

The C & K metrics were validated by using six[8] of Weyuker's[Wey88] nine axioms, and were found to be generally compliant with most of the properties. None of the metrics was found to comply to either of properties 7 and 9 of Weyuker. Property 7 says that if there exist program bodies (objects according to C & K) P and Q such that Q is formed by permuting the statements in P, then $\mu(P) \neq \mu(Q)$, where μ is a complexity metric; i.e. permutation affects complexity. According to C & K, failure to meet this property suggests that permutation might not be significant in OO systems[9]. Property 9 says that $\exists$ program bodies P and Q such that $\mu(P) + \mu(Q) < \mu(P + Q)$, or that interaction between program bodies (objects) increases complexity. Failure to meet this property suggests that it is probably not applicable to OO systems, where interaction might in fact decrease complexity by rendering classes closer to the abstractions they are supposed to portray.

3.3 Li and Henry

Li and Henry[LHKS95, LH93] present ten metrics in their system; they include five of the six metrics defined by C & K, namely DIT, NOC, RFC, LCOM, and WMC. In addition they define five more metrics of their own.

In addition to coupling through inheritance, where one class derives from another class (C & K's DIT or NOC could be used here), there are two coupling metrics:

- Message-Passing Coupling (MPC) measures the complexity of message passing among classes. MPC is the (static) number of send statements defined in a class, where a send statement is a message sent out from a method in a class to a method in another class. Although messages are passed among objects, the types of messages passed are defined in classes. Therefore, message passing is calculated at the class level instead of the object level.

- Data Abstraction Coupling (DAC). A class can be viewed as an implementation of an ADT. A variable declared within a class may have a

[7]contrast with Henderson-Sellers remark on RFC and CBO above

[8]Seven of the nine properties were considered in [CK91].

[9]Property 7 was dropped in [CK94].

type of ADT which is another class definition, and hence a particular type of coupling is created between the two classes. DAC for a class is its number of instances of ADTs, or the number of its variables (data members) having an ADT type.

The third new metric measures the class increment interface, or the number of methods locally defined in a class.

- The Number Of Methods (NOM) = number of local methods.

Two size metrics are also defined. They are:

- The **number of semicolons (SIZE1)** in a class is a LOC traditional metric.
- Number of properties (SIZE2) is the number of attributes plus the number of local methods.

To validate the metrics, the authors present the experiments they conducted on two software systems on which the metrics were applied. In addition, the maintenance effort made during a period of three years is calculated for all classes of the two systems and a multiple regression model is used in order to derive the metrics capabilities.

The authors finally use the experiment results to derive the following conclusions:

- "There is a strong relationship between metrics and maintenance effort in object-oriented systems".
- "Maintenance effort can be predicted from combinations of metrics collected from source code".

Besides the previously cited comments on the C & K metrics used by Li and Henry, the following remarks are noteworthy:

- In SIZE1, the authors use the number of semicolons in a class, which is language-dependent, and also not derivable until the source code is available.
- The DIT metric is used as a measure of complexity, where the larger the value of DIT, the more complex the system is supposed to be. This point criticized by Hitz and Montazeri[HM95] where it is induced that trying to minimize DIT (in order to decrease complexity) leads to the guideline "do not use inheritance at all", while inheritance is one the major advantages of the OO paradigm.

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

Table 1: Operation complexity value
(from Chen and Lu[CL93])

3.4 Chen and Lu

Chen and Lu[CL93] present a new set of metrics for OO design. Most of the metrics measure the complexity of classes.

Operation complexity (*OpCom*) of a class. The definition for operation complexity is:

$$\sum O(i)$$

where O(i) is operation i's complex value, and is evaluated from Table 1. [10] Summing up the O(i) in for each operation i in the class gives this metric value.

Operation argument complexity (*OAC*): Defined as

$$\sum P(i)$$

where P(i) is the value of each argument i in each operation in the class; it is evaluated from Table 2.

Attribute complexity (*AC*) metric: defined as $\sum R(i)$ where R(i) is the value of each attribute in the class, and is also evaluated from Table 2. Summing up all R(i) in the class gives this metric value.

Operation coupling (*OpCpl*) metric: Measures the coupling between operations in the class and operations in other classes. It is defined as the sum of:

[10] The authors explain that this table is similar to one derived from Boehm[Boe81], but did not elaborate further as to which table is exactly referred to. We found that table 24-8 in [Boe81] is probably the closest to table 1. We still find, however, that this is a subjective rating and relies heavily on expert judgment.

Type	Value
Boolean or integer	1
Char	1
Real	2
Array	3-4
Pointer	5
Record, Struct, or Object	6-9
File	10

Table 2: Argument/attribute value
(from Chen and Lu[CL93])

- The number of operations which access other classes.
- The number of operations which are accessed by other classes.
- The number of operations which are co-operated with other classes. A co-operated operation is one which accesses some other class' operations and vice versa[11].

Class coupling (*ClCpl*) metric: Measures the coupling between a class and other classes. It is the sum of:

- The number of accesses to other classes.
- The number of accesses by other classes.
- The number of co-operated classes. A co-operated class is one which accesses some other class and vice versa.

The authors say that the difference between the latter two metrics lies in their "different viewpoints".

Cohesion (*Coh*) metric: Consider a class with N operations: F(1), F(2), ..., F(N), with N sets of arguments I(1),I(2), ... ,I(N); M is the number of disjoint sets of arguments formed by the intersection of these N sets. The cohesion metric is defined as $\frac{M}{N}$*100% In the example of Figure 1, we have a class of four operations (methods) $F(1), ..., F(4)$, with I(1)={a,b,c}; I(2)={a,b,d,e}; I(3)={a,b,e}; and I(4)={a,b,c,d}; there is only one disjoint set (this is the minimum case), so M = 1, and N = 4. The cohesion metric here = 25%.

[11]We deduced that this might be the intersection of the two sets of operations formed in the previous two points.

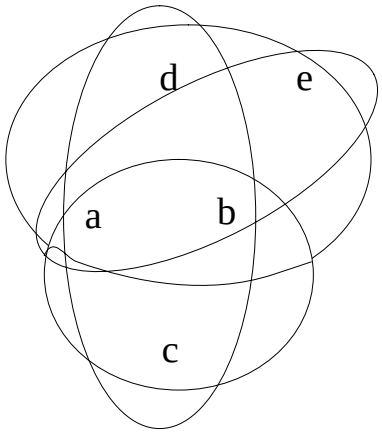

Figure 1: Class with high cohesion

The lower the value is, the higher the cohesion. The authors differentiate their cohesion metric from LCOM of C & K[CK94] by pointing out that in the latter, the intersection of "instance variables" is taken into account, while "arguments" are used in the former, and "instance variables used may be unavailable in the design phase".

Class hierarchy (CH) metric: Defined as the sum of the following:

- The depth of the class in the inheritance tree.
- The number of sub-classes of the class.
- The number of direct super classes of the class.
- The number of local or inherited operations available to the class.

It is claimed that the deeper a class is in the hierarchy and the more a class has children, the more complex that class is likely to be.

Reuse (*Re*) metric: It measures whether a class is a reused one. A value of 1 is given to the class if it is reused from the current or from a previous project, and 0 otherwise.

The metrics were validated by applying them to two small projects, and using the judgment of a number of expert designers as to the complexity of the designs. A statistical regression model is then used in an attempt to derive a correlation between the metrics and the experts' judgments (and hence complexity).

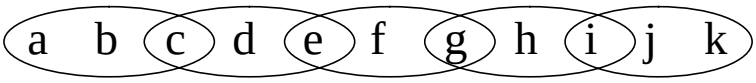

Figure 2: Class with low cohesion

In [HS96], Henderson-Sellers explicitly cites Table 1 and argues that "metrics with subjective weightings in which not only are Likert scales used, but the mapping from that scale to a numerical scale is itself fuzzy, have no scientific validity, and should be avoided if at all possible".

A very similar criticism of subjective ratings is given by Brito e Abreu in [BeA92], where it is argued that "subjectivity makes metrics comparisons throughout software industry an impossible mission". Also, "subjective ratings (e.g. "Very Low", "Low", "Average", "High", "Very High") are copious in the metrics literature".

Moreover, we have the following remarks on the above set of metrics:

One of the metrics is called "cohesion metric", which is a bit misleading, since a lower value reflects more cohesiveness. The metric should rather represent the "lack" of cohesion of a class.

There is some ambiguity surrounding the difference between the "operation coupling" and "class coupling" metrics. The latter metric uses the number of accesses from a class to another; but it is not specified what constitutes a class access. We deduce that the class coupling metric simply takes into account whether a class accesses another class (through message passing), regardless of how many messages are sent from that class to the other; thus the value of the class coupling metric is either 0 or 1 when taken between two classes. However, the number of messages does matter in the operation coupling metric.

There is an apparent flaw in the cohesion metric, which seems to encourage having a large number of arguments in each operation over having a small number of arguments. In the example used in the definition, there are 4 sets of arguments, and only one disjoint set, resulting in a value of 25% (see Figure 1). However, by looking at Figure 2, there are 11 arguments in 5 different operations in the class; but there is only one disjoint set, so the cohesion metric yields a value of 20%, which reflects more cohesion than in the previous case. However, a quick look at the two figures shows that the class in the first example intuitively exhibits more cohesion than the one in the second. Moreover, when there is only one method in the class, then the metric would yield 100%, the worst possible cohesion, while in fact the opposite is true.

In dealing with C & K's LCOM metric, Hitz and Montazeri[HM96a] propose a solution to the case where only one disjoint argument set is present. They proposed a modification for LCOM which would discriminate between cases where there is only one disjoint set. The same problem also occurs here.

We propose the modification of the cohesion metric so that if there is only one disjoint set, then the metric would yield the same value (e.g. zero), and then use the formula from Hitz and Montazeri[HM96a] for discriminating between such cases.

Finally, the class hierarchy metric seems to discourage any sort of inheritance, since the complexity increases whenever the inheritance increases. Therefore, a more careful and balanced approach should be given so that a reasonable use of inheritance would not be punished, as would be expected in any OO system.

3.5 Brito e Abreu

Brito e Abreu[BeA92, BeAM96] derived a set of six metrics known as the MOOD (Metrics for Object Oriented Design) metrics. In the following, C stands for class, M for method, A for attribute, and TC for the total number of classes in the system being measured.

Method Hiding Factor (MHF):

$MHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{M_d(C_i)} (1 - V(M_{mi}))}{\sum_{i=1}^{TC} M_d(C_i)}$

where $V(M_{mi}) = \frac{\sum_{j=1}^{TC} is\_visible(M_{mi}, C_j)}{TC-1}$

and $is\_visible(M_{mi}, C_j) = \begin{cases} 1 & iff \begin{cases} j \neq i \\ C_j \ may\ call\ M_{mi} \end{cases} \\ 0 & otherwise \end{cases}$

The MHF numerator is the sum of the invisibilities of all methods defined (M_d) in all classes. The invisibility of a method is the percentage of the total classes from which this method is not visible. The MHF denominator is the total number of methods defined in the system under consideration.

Attribute Hiding Factor (AHF):

$$AHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{A_d(C_i)} (1 - V(A_{mi}))}{\sum_{i=1}^{TC} A_d(C_i)}$$

where $V(A_{mi}) = \frac{\sum_{j=1}^{TC} is\_visible(A_{mi}, C_j)}{TC-1}$ and

$$is\_visible(A_{mi}, C_j) = \begin{cases} 1 & iff \begin{cases} j \neq i \\ C_j \ may\ call\ A_{mi} \end{cases} \\ 0 & otherwise \end{cases}$$

The AHF numerator is the sum of the invisibilities of all attributes defined (A_d) in all classes. The invisibility of an attribute is the percentage of the total classes from which this attribute is not visible. The AHF denominator is the total number of attributes defined in the system under consideration.

Method Inheritance Factor (MIF):

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

where $M_a(C_i) = M_d(C_i) + M_i(C_i)$

The MIF denominator is the sum of inherited methods (M_i) in all classes of the system under consideration. The MIF denominator is the total number of available methods (M_a) (locally defined plus inherited) for all classes.

Attribute Inheritance Factor (AIF):

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

Where $A_a(C_i) = A_d(C_i) + A_i(C_i)$

The AIF numerator is the sum of inherited attributes (A_i) in all classes of the system under consideration. The AIF denominator is the total number of available attributes (A_a) (locally defined plus inherited) for all classes.

Polymorphism Factor (POF):

$$POF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) \times DC(C_i)]}$$

where $M_d(C_i) = M_n(C_i) + M_o(C_i)$ and M_n = new methods; M_o = overriding methods; DC = descendants count.

The POF numerator represents the actual number of possible different polymorphic situations. Indeed, a given message sent to class C_i can be bound, statically or dynamically, to a named method implementation. The latter can have as many shapes (morphos) as the number of times this same method is overridden (in C_i's descendants). The POF denominator represents the

maximum number of possible distinct polymorphic situations for class C_i. This would be the case where all new methods defined in C_i would be overridden in all of their derived classes.

Coupling Factor (COF):

$$COF = \frac{\sum_{i=1}^{TC} [\sum_{j=1}^{TC} is\_client(C_i, C_j)]}{TC^2 - TC}$$

where

$$is\_client(C_c, C_s) = \left\{ \begin{array}{ll} 1 & iff\ C_c \Longrightarrow C_s \wedge C_c \neq C_s \\ 0 & otherwise \end{array} \right\}$$

The COF denominator stands for the maximum possible number of couplings in a system with TC classes. The client-supplier relation (represented by $C_c \Longrightarrow C_s$) means that C_c (client class) contains at least one non-inheritance reference to a feature (method or attribute) of class C_s (supplier class). The COF numerator then represents the actual number of couplings not imputable to inheritance.

The metrics were applied to eight projects representing eight variations of the design for the same requirements document[BeAM96]. A correlation is established between the metrics and defect density, failure density, and normalized rework. The results show that most of the metrics were good predictors of the three quality measures.

3.6 Abbott, Korson, and McGregor

Abbott et al.[AKM94] propose metrics for measuring the number and strength of the object "interactions permitted" by an object oriented design. They distinguish between two types of complexity applicable to class definitions: Interaction level and interface size. The complexity referred to is the cognitive complexity.

The proposed metric is said to be derivable directly from design information and is able to predict experts' preferences of design alternatives.

Classes in OO designs exhibit various levels of complexity along 2 dimensions:

1. Interaction level (IL): The degree to which an object is prone to interaction with other objects by providing opportunities for such interactions.

2. Interface size (IS): The degree to which classes provide means for information to flow in and out of their encapsulation.

Interface size measures the surface complexity of a class, while the interaction level measures the opportunities for interaction between its surface and its interior.

The quantum for interaction level is called a "permitted interaction", and is an instance of two objects being permitted to interact by the design. The quantum for interface size is called an "interface item", and is an occurrence of a parameter or return value in a method's signature. The method itself counts as an interface item, while data members do not. The interaction level of a class is the sum of the interaction levels of its methods. The interaction level of a design is the sum of the interaction levels of its classes. The interface size of a class is the sum of the interface sizes of its methods. The interface size of a design is the sum of the interface sizes of its classes.

Only one metric is singled out; it is termed the "permitted interaction" metric and measures the interaction level of a design.

Three approaches are considered in deriving the permitted interactions metric:

- The first assumes that each permitted interaction has equal weight, so the number of interactions determines the complexity. The problems with this approach are that it ignores the interface sizes of the interacting objects and that it excessively encourages grouping data together so that multiple data members and/or procedures count as one instead of many.

  The example in Figure 3 illustrates this fact; on the lefthand side, we have a method with 3 arguments, 3 data members, and a return value; there are 12 possible interactions, 9 between the arguments and the data members, and 3 between the data members and the return value. However, if the arguments and the data types are packed into one composite data type, such as shown on the righthand side in Figure 3, then the number of permitted interactions is reduced to 2, one between the composite argument and the composite data member, and one between the composite data member and the return value.

- The second approach takes into consideration a measure of the strength of interaction between the objects. The strength of interaction between an input object A and an output object B is defined as:

  $$\text{strength} = (\text{interface size of A}) * (\text{interface size of B}).$$

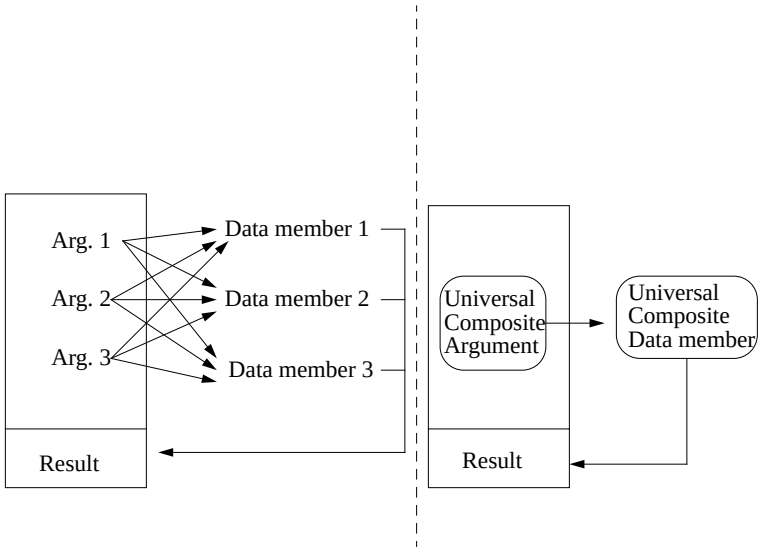

Figure 3: Transformation to minimize the number of interactions (from Abbott et al.[AKM94])

This approach also has its drawbacks: It encourages the use of elementary classes by other classes (in order to reduce the strength of interaction), rather than classes representing cohesive groupings of data.

In the example of Figure 4, we have a method whose purpose is to determine whether two dates occur in the same quarter of the year. On the left, we have one argument representing a compact date object, 3 data members, and a return value. By assigning 3 as a weight to the date object and 1 to each of the data members, the strength of interaction between the argument and each of the data members is $3 \times 1 = 3$, and that between each of the data members and the return value is $1 \times 1 = 1$, for a total of 12 units. However, since only the month and year are needed to determine the quarter, we can have the method take two arguments, year and month, each having a weight of 1, thus reducing the strength of interaction down to 9. But the first choice of arguments is clearly preferred in OO systems because it represents better encapsulation.

- The third approach merges the first two on the basis that they "fail in complementary ways". The interaction level is defined as:

  interaction level = K1 * (value based on number of interactions) +

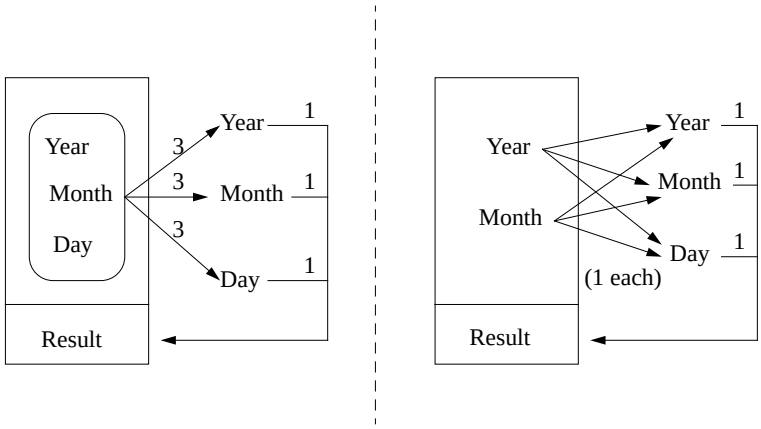

Figure 4: Transformation to minimize the strength of interactions (from Abbott et al.[AKM94])

K2 * (value based on strength of interactions); where K1, and K2 are tentatively set to 1.0 each.

The rationale behind this formula is that it represents a trade-off between two points of view, the first saying that only the number of interactions is important, while the second claims that only the strength of interactions is important.

In a similar argument, the interface size metric is given as:

interface size = K3 * (value based on number of interface items) + K4 * (value based on size of interface items); K3 and K4 are also tentatively set to $\frac{1}{8}$ and $\frac{1}{4}$, respectively.

The authors propose setting values for elementary classes/types (integers, characters ...), such as the value of 1.0; the computation of more complex types is built starting from the elementary values that compose it or constitute is parts.

The metrics were applied to 9 different sets of 2 or 3 design alternatives each, for a total of 20 design cases, with a number of expert designers giving their independent preferences between the various alternatives. The metrics and the experts' preferences agreed in 16 of the 20 cases.

3.7 Hitz and Montazeri

Hitz and Montazeri[HM96b] discuss the concept of coupling in OO systems in detail. Two levels of coupling are identified: Object level coupling (OLC), and class level coupling (CLC). A framework for measuring OO coupling is set up. It emphasizes the distinction between OLC and CLC.

Class level coupling is defined as the coupling that results from state dependencies between classes during the development life-cycle, where the state of a class refers to the class definition and the program code of its methods (a version of the class implementation). In what follows, CC denotes a dependent client class, and SC is the server class being changed.

The following factors contribute to the strength of CLC:

- Stability of SC: If SC is considered stable, no changes are likely to occur, so CC will not incur any dependent changes. If SC is unstable, then 2 sub-cases are considered: Only SC's implementation is subject to change, while the interface is stable or, SC's interface is modified. The second sub-case is clearly more harmful.

- Type of access to SC: CC may either restrict its access to the interface of SC, following its protocol, or may refer to at least one instance variable defined in SC[12]. The latter case is considered a breach of encapsulation and results in higher coupling values.

- Scope of access to SC within CC: this refers to where in the program area of CC SC may potentially be referenced. The larger that program area, the more potential changes to CC are incurred following a change in SC.

Table 3 shows the metric values for CLC, given on an ordinal scale.

Class level coupling is given weights according to stability, access type, and scope of access as outlined in the framework. Access to a stable server class represents the least coupling value, while in an unstable server class case, access to the interface is better than access to implementation. Also, a restricted scope of access gives inferior coupling values than access to a server class from potentially a large program area inside the client class.

In object-level coupling, the authors differentiate between two types of objects: native and non-native. An object M is called native to another object A if and only if:

[12]The original definition uses "instance variable 'of' SC", which we judged to be ambiguous.

SC native to CC?		SC stable?		
		yes	no	
		Access to		
		interface	interface	implementation
yes	CC is genuine aggregate of SC	1	3	5
	Local variable of type SC used within method of CC	1	2	4
	SC is superclass of CC	1	3	5
no	SC is class of shared instance variable of CC	1	3	5
	SC is type of parameter of method of CC	1	2	4
	CC accesses global variable of SC	1	3	5

Table 3: Class Level Coupling

- M is a genuine aggregate of M, i.e. M is a sub-object of A that can only exist as part of A or,
- M is represented by a local variable of one of A's methods, so M can only exist during the activation period of that method or,
- M is a sub-object of A inherited from one of A's super classes.

All other objects are non-native. The state of a native object represents part of the owner object's state; thus, sending a message to a native object does not contribute to object level coupling, since such a message to one of the object's own components will only affect its own history. On the other hand, such a message might constitute class level coupling. "Any evidence of a method of one object using methods or instance variables of a non-native object constitutes OLC".

The following factors affect the strength of OLC between objects X and O:

- Type of access to X by O (O restricts its access to the interface, or refers to at least one instance variable of X).
- Scope of X (X may be a parameter to one of O's methods, a non-native part of O, or a global object).
- Complexity of interface (number of arguments).

	Access to	
	interface	implementation
SC is class of shared instance variable of CC	II	V
SC is type of parameter of method of CC	I	IV
CC accesses global variable of SC	III	VI

Table 4: Object Level Coupling
Adapted from Hitz and Montazeri[HM96b]

Table 4 shows the metric values for OLC; the authors use Roman numerals as a reminder that OLC values can't be added to CLC values.

In object-level coupling, the values are given along the access type and scope of access, so that, for example, a case where SC is a type of parameter of a method of CC and CC has access to the interface of SC represents less coupling than a case where CC accesses a global variable of class SC and has access to the implementation of SC.

The values from the two tables of metrics belong to ordinal scales and cannot be summed; thus the two scales are incomparable. This explains why the values in the two tables were given as Arabic and Roman numerals, respectively. The interface complexity factor was not considered directly in the metrics.

Two supplementary coupling-related attributes are also defined:

Change Dependency Between Classes (CDBC), which determines the number of methods of a client class CC to be considered when a change in a server class SC occurs. This depends on the scope of usage of SC inside CC. The degree of CDBC is given as :

$$A = \sum_{accesses\ i\ to\ implementation} \alpha_i + (1-k) \sum_{accesses\ i\ to\ interface} \alpha_i$$

where for class CC with n methods, k is a factor corresponding to the stability of SC ($0 \leq k \leq 1$), and α is deduced from table 5.

CDBC(CC,SC)=min(n,A).

Locality of Data (LD), which is defined as the ratio of the amount of data local to a class to the total amount of data used by that class. Such a measure relates to the quality of abstraction embodied by a class, where classes with

	α = number of methods of CC potentially affected by a change
SC is not used by CC at all	0
SC is the class of an instance variable of CC	n
Local variables of type SC are used within j methods of CC	j
SC is a superclass of CC	n
SC is the type of parameter of j methods of CC	j
CC accesses a global variable of class SC	n

Table 5: Relationship types between CC and SC and their corresponding contribution α to change dependency.

high data locality are more self-sufficient than those with low data locality. A definition for C++ is given as :

$$LD = \frac{\sum_{i=1}^{n} |L_i|}{\sum_{i=1}^{n} |T_i|}$$

where M_i $(1 \leq i \leq n)$ are the methods of the class, L_i is the set of local variables accessed by M_i, and T_i is the set of all variables used in M_i.

The conclusion is that no one metric can capture complexity, but rather, complexity measures should be looked at from several dimensions, as the proposed metrics show. This fact is also portrayed in the inability to sum up the values in the tables.

The authors did note state whether the metrics were validated or tested.

3.8 Miscellaneous Metrics

Many other metrics were defined for the OO paradigm. In this section, we briefly list some of metrics we came across in the literature.

Yap and Henderson-Sellers[YHS93] define two measures related to class reuse:

The reuse ratio (U) = (number of superclasses)/(total number of classes). This indicates the the level of reuse of superclasses through the creation of new classes.

The specialization ratio (S) = (number of subclasses)/(number of superclasses). It measures the degree to which a superclass was successful in representing the abstraction required. A large value indicates a high degree of reuse by subclassing.

Rising and Calliss[RC94] suggest a metric for information hiding at the module level. They note that a module which hides more than one design decision is poorer in **information hiding (IH)** than one hiding a single design decision.

IH(Module)=(0|1)+sum of all extraneous entities;

A 0 is assigned if one design decision is encapsulated, and 1 otherwise. The extraneous entities are "those not required for a single design decision".

"For each module m in a system S, E_m is the set of all extraneous entities. For each j in E_m, C_j is the number of client modules referencing j. U_m is the use of each module by the system, $U_m = \sum \{C_j | j \in E_m\}$"

The information hiding metric for system S would be:

$$IH(S) = Median\{IH(m) + U_m | m \in S\}$$

Note: The authors did not clearly specify a method for counting extraneous entities.

Lorenz and Kidd[LK94] defined many OO design metrics, but did not validate nor thoroughly test them. The metrics are listed below, along with the level at which they are taken:

Method Size: Number of message sends, number of statements, lines of code, average method size.

Method Internals: Method complexity, strings of message sends[13].

Class size: Number of public instance methods (NPM) per class, number of instance methods per class, average number of instance methods per class, number of instance variables per class, average number of instance variables per class, number of class methods per class, number of class variables per class.

Class inheritance: Class hierarchy level, **number of abstract classes (*NAC*)**, use of multiple inheritance.

Method inheritance: Number of methods overridden (NOV) by a subclass, **number of methods inherited (*NIM*)** by a subclass, number of methods added in a subclass, specialization index.

Class internals: Class cohesion, global usage, average number of parameters per method, use of friend functions, percentage of function-oriented code,

[13]Messages can be "strung" together in Smalltalk.

average number of comment lines per method, average number of commented methods, number of problem reports per class or contract.

Class externals: Class coupling, number of times a class is reused, number of classes/methods thrown away.

Bellin, Tyagi, and Tyler[BTT] classify object oriented metrics into three groups (see Section 4). The proposed metrics along with the groups are:

Group A: Number of methods, number of classes, number of messages, number of receiving classes (servers), number of sender classes (actors), number of agent classes, number of global variables in each class, number of levels in the class hierarchy, number of leaves in the class hierarchy tree, ratio between depth and breadth, ratio of method/class, ratio of private/public methods, ratio of abstract/instantiated classes, ratio of lines of code/method, ratio of lines of code/comment.

Group B: Number of classes reused, percent of reused classes modified.

Group C: Coupling, cohesion, sufficiency, completeness, primitiveness.

The proposed metrics were neither validated nor tested as to the time of writing of the paper. However, the authors promised to apply and validate the metrics in future work.

Sheetz, Tegarden, and Monarchi[STM91] define another extensive set of metrics which they classify in four levels (see Section 4). The metrics are :

Variable level metrics : **Variable fan-in (*vfi*)**, **variable fan-out (*vfo*)**, **variable polymorphism (*vp*)**, **variable fan-down (*vfd*)**.

Method level metrics : Method input parameters (*mip*), **method parameters returned (*mpr*)**, **object variables accessed (*ova*)**, **parameters returned to the method (*prm*)**, **method parameters passed (*mpp*)**, method fan-in (*mfi*), method fan-out (*mfo*), method polymorphism (*mp*), method fan-down (*mfd*).

Object level metrics : **Object input parameters (*oip*)**, **object parameters returned (*opr*)**, **parameters returned to the object (*pro*)**, **object parameters passed (*opp*)**, number of local variables (*olv*), **number of inherited variables (*oiv*)**, **number of local methods (*olm*)**, number of inherited methods (*oim*), object fan-in (*ofi*), object fan-out (*ofo*), **object fan-down (*ofd*)**, **object fan-up (*ofu*)**, **object to root depth (*ord*)**, **object to leaf depth (*old*)**, **object polymorphism (*op*)**.

Application level metrics Number of classes (NCL), application concrete classes (acc), application abstract classes (aac), maximum depth of the object hierarchy (amd), maximum breadth of the object hierarchy (amb).

In addition, a number of other metrics were defined using formulae based on the previous metrics, e.g. the number of input/output variables for method i is defined as $miov_i = mip_i + mpr_i + ova_i + prm_i + mpp_i$, which is a measure of the amount of information flow associated with the method.

3.9 Summary

Table 6 summarizes some of the various metrics surveyed previously. The metrics derived at the code level were omitted. Also, only a representative subset of the extensive sets of metrics were used.

4 Classifications of Object-Oriented Metrics

Several researchers classify the OO metrics along different dimensions in an attempt to organize the metrics collection. The classifications are mainly aimed at easing the collection of metrics by helping the users know which metrics are found at which level of resolution[HS94].

4.1 Henderson-Sellers

In[HS94], Henderson-Sellers considers the various perspectives of an OO system in classifying the metrics. The perspectives are: inside a class, external at the class level, system level (ignoring relationships), systems level relationships (excluding inheritance), and inheritance coupling.

Inside a class: Size and complexity measures are to be found at this level; examples are McCabe's cyclomatic complexity[McC76], Chidamber and Kemerer's[CK94] weighted methods per class (WMC), and Li and Henry's[LH93] number of methods and number of attributes counts ...

External at the class level: This level concerns the interface of classes; the metrics here can be viewed as measuring the services offered by a class.

System level (ignoring relationships): Measures from the two previous levels are accumulated at this level. An example is the total number of classes in the system. Statistical computations can also be derived here, such as the mean/standard deviation of the number of methods in a system ...

Acronym	Name	Origin	Defined
aac	application abstract classes	Sheetz, Tegarden, and Monarchi[STM91]	3.8 (p.24)
acc	application concrete classes	Sheetz, Tegarden, and Monarchi[STM91]	3.8 (p.24)
AHF	Attribute Hiding Factor	Brito e Abreu[BeA92, BeAM96]	3.5 (p.13)
AIF	Attribute Inheritance Factor	Brito e Abreu[BeA92, BeAM96]	3.5 (p.13)
amb	maximum breadth of the object hierarchy	Sheetz, Tegarden, and Monarchi[STM91]	3.8 (p.24)
amd	maximum depth of the object hierarchy	Sheetz, Tegarden, and Monarchi[STM91]	3.8 (p.24)
CBO	Coupling Between Object classes	Chidamber and Kemerer[CK94]	3.2 (p.4)
CDBC	Change Dependency Between Classes	Hitz and Montazeri[HM96b]	3.7 (p.19)
CH	Class hierarchy	Chen and Lu[CL93]	3.4 (p.9)
CLC	class level coupling	Hitz and Montazeri[HM96b]	3.7 (p.19)
ClCpl	Class coupling	Chen and Lu[CL93]	3.4 (p.9)
COF	Coupling Factor	Brito e Abreu[BeA92, BeAM96]	3.5 (p.13)
Coh	Cohesion	Chen and Lu[CL93]	3.4 (p.9)
DAC	Data Abstraction Coupling	Li and Henry[LHKS95, LH93]	3.3 (p.7)
DIT	Depth of Inheritance Tree	Chidamber and Kemerer[CK94]	3.2 (p.4)
IC	Inheritance complexity	Moreau and Dominick[MD89]	3.1 (p.3)
IL	Interaction level	Abbott et al.[AKM94]	3.6 (p.15)
IS	Interface size	Abbott et al.[AKM94]	3.6 (p.15)
LCOM	Lack of COheshion in Methods	Chidamber and Kemerer[CK94]	3.2 (p.4)
LD	Locality of Data	Hitz and Montazeri[HM96b]	3.7 (p.19)
MDS	Message domain size	Moreau and Dominick[MD89]	3.1 (p.3)
mfd	method fan-down	Sheetz, Tegarden, and Monarchi[STM91]	3.8 (p.24)
mfi	method fan-in	Sheetz, Tegarden, and Monarchi[STM91]	3.8 (p.24)
mfo	method fan-out	Sheetz, Tegarden, and Monarchi[STM91]	3.8 (p.24)
MHF	Method Hiding Factor	Brito e Abreu[BeA92, BeAM96]	3.5 (p.13)
MIF	Method Inheritance Factor	Brito e Abreu[BeA92, BeAM96]	3.5 (p.13)
mip	Method input parameters	Sheetz, Tegarden, and Monarchi[STM91]	3.8 (p.24)
mp	method polymorphism	Sheetz, Tegarden, and Monarchi[STM91]	3.8 (p.24)
MPC	Message-Passing Coupling	Li and Henry[LHKS95, LH93]	3.3 (p.7)
MVS	Message vocabulary size	Moreau and Dominick[MD89]	3.1 (p.3)
NCL	Number of classes	Sheetz, Tegarden, and Monarchi[STM91]	3.8 (p.24)
NOC	Number of Children	Chidamber and Kemerer[CK94]	3.2 (p.4)
NOM	Number Of Methods	Li and Henry[LHKS95, LH93]	3.3 (p.7)
NOV	Number of methods overridden	Lorenz and Kidd[LK94]	3.8 (p.23)
NPM	Number of public instance methods	Lorenz and Kidd[LK94]	3.8 (p.23)
ofi	object fan-in	Sheetz, Tegarden, and Monarchi[STM91]	3.8 (p.24)
ofo	object fan-out	Sheetz, Tegarden, and Monarchi[STM91]	3.8 (p.24)
oim	number of inherited methods	Sheetz, Tegarden, and Monarchi[STM91]	3.8 (p.24)
OLC	Object level coupling	Hitz and Montazeri[HM96b]	3.7 (p.19)
olv	number of local variables	Sheetz, Tegarden, and Monarchi[STM91]	3.8 (p.24)
OpCpl	Operation coupling	Chen and Lu[CL93]	3.4 (p.9)
POF	Polymorphism Factor	Brito e Abreu[BeA92, BeAM96]	3.5 (p.13)
RFC	Response For Class	Chidamber and Kemerer[CK94]	3.2 (p.4)
S	specialization ratio	Hitz and Montazeri[HM96b]	3.7 (p.19)
SIZE2	Number of properties	Li and Henry[LHKS95, LH93]	3.3 (p.7)
U	reuse ratio	Hitz and Montazeri[HM96b]	3.7 (p.19)
WMC	Weighted Methods per Class	Chidamber and Kemerer[CK94]	3.2 (p.4)

Table 6: Metrics Surveyed

System level relationships (excluding inheritance): Coupling is the main measure here. The coupling between objects (CBO)[CK94], message passing coupling (MPC), and data abstraction coupling (DAC)[LH93] are examples of metrics found at this level.

Inheritance coupling: Measures of the inheritance hierarchy of a system and the resulting complexity are classified in this level. The main inheritance-related metrics used by Henderson-Sellers here are the depth of inheritance tree (DIT) and the number of children (NOC), both of Chidamber and Kemerer[CK94].

4.2 Sheetz et al.

Sheetz et al.[STM91] define four levels along which metrics are classified. These are: **variable level, method level, object level, and application level**. They define their own set of metrics and list them within the four classification levels (see Section 3.8). All the metrics measure the complexity of software.

4.3 Bellin et al.

Yet another approach to classifying metrics comes from Bellin[BTT]; three groups of metrics are presented; these are called group A, group B, and group C.

Group A consists of "statistical aspects of OO design captured via a static analysis of the source code". It contains such metrics as "number of methods", "number of classes", "number of levels in the class hierarchy tree" ...

Group B deals with code reuse, and contains the metrics "number of classes reused", and "percent of reused classes modified".

Group C contains subjective measures which involve the human factor, such as coupling and cohesion, and deal with the quality of an abstraction in an OO system.

4.4 Brito e Abreu and Carapuca

The final classification framework for metrics we present in this section is that of Brito e Abreu and Carapuca[BeAC94]. The classification framework is called TAPROOT (taxonomy precis for object-oriented metrics). The metrics are classified along two "independent vectors", **category** and **granularity**. The authors reveal that the categories were derived after a sample of 128 references was reviewed in order to find a "common denominator" in the extensive metrics literature. The categories are: **Design, size, complexity, reuse,**

	Method	Class	System
Design	MD	CD	SD
Size	MS	CS	SS
Complexity	MC	CC	SC
Reuse	MR	CR	SR
Productivity	MP	CP	SP
Quality	MQ	CQ	SQ

Table 7: TAPROOT Classification Framework

productivity, and **quality**. The second dimension (or vector), granularity, further refines the categories by considering metrics in each category at the **method, class,** and **system levels**. This is somewhat similar to the classifications given in[STM91] (and, to a certain extent, in[HS94]); however, when the six category levels are considered, a combination of eighteen independent levels is obtained. Table 7 shows the different combinations of category and granularity. The following examples are of metrics from the design granularity level, as we are mainly interested in design metrics.

- Method design (MD) metrics: percentage of used instance variables, comments density ...
- Class design (CD) metrics: lack of cohesion in methods (LCOM), depth of inheritance, class response[CK94].
- System design (SD): Average number of methods, average number of instance variables ...

5 A Data Model

Our contribution is to develop a data model for a database of design information from which most of the proposed OO metrics can be computed. Figures 8 and 9 summarize the entities and relationships in the data model. Combined with suitable tools for extracting basic design information from designs (or, perhaps, from code), such a database would make it easier to compare metrics and validate a large number of metrics from the same basic data. Researchers with access to development and maintenance cost information might then be able to compare the validity and utility of most proposed metrics. Table 10

E1	class	E2	method
E3	public method	E4	private method
E5	attribute	E6	public attribute
E7	private attribute	E8	method arguments
E9	return value	E10	global variable
E11	abstract class	E12	concrete class

Table 8: Summary of Entities

R1	class sends message to class/method	R2	class uses attribute of class
R3	class inherits from class	R4	class inherits method
R5	class inherits attribute	R6	class is a child/descendant of class
R7	class uses instance of other class	R8	class is parent/ancestor to class
R9	method returns value	R10	method calls method
R11	class/method uses attribute	R12	method overrides method
R13	argument interacts with attribute	R14	class is type of argument to method
R15	class accesses interface of class	R16	class accesses implementation of class
R17	class accesses global variable of class		

Table 9: Summary of Relationships

summarizes what parts of the data model are used by the metrics we survey.[14] The abbreviations refer to specific metrics covered in our survey. The final "count" entry reports how many metrics use each element of the data model.

We plan to validate the data model by extracting a design database for each of several object-oriented systems, and computing all the desired metrics from the data model. We expect that some metrics will be computable from "early" design information, but some will require detailed design and even code-level information.

6 Conclusions

As can be inferred from some proposed sets, there seems to be a rush towards defining as many metrics as possible, sometimes disregarding their usefulness. One possible reason for this is that many of the proposed metrics are easily derivable.

Binkley and Schach[BS96] note that "a metric that is easy to compute is

[14]We have omitted metrics that require code-level properties.

	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	E11	E12	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17
MVS	*												*																
IC	*														*			*		*								
MDS	*												*																
WMC	*	*																											
DIT	*														*														
NOC	*																	*		*									
CBO	*												*						*										
RFC	*	*																				*							
LCOM		*			*																		*						
MPC	*	*																				*							
DAC	*																		*										
NOM	*	*																											
SIZE2		*			*																								
IL	*	*						*	*												*				*				
IS	*	*						*	*																				
CLC	*	*								*									*	*						*	*	*	*
OLC	*	*								*									*							*	*	*	*
CDBC	*	*								*			*																
LD	*	*			*	*	*																*						
NCL	*																												
aac	*										*																		
acc	*											*																	
amd/amb	*														*			*		*									
olv	*				*	*	*																						
oim	*	*														*													
ofi/ofo	*												*																
mip		*						*																					
mfi		*											*																
mfo		*											*																
mp	*	*																											
mfd	*	*																		*									
U	*												*							*									
S	*																	*		*									
NPM	*		*																										
NOV	*																							*					
MHF	*	*	*	*									*																
AHF	*				*	*	*							*															
MIF	*	*														*													
AIF	*				*												*												
POF	*	*																*						*					
COF	*												*																
OpCpl	*	*											*																
CH	*	*													*	*		*		*									
ClCpl	*												*																
Coh	*	*			*																		*						
count	40	24	2	1	7	3	3	3	2	3	1	1	12	1	4	3	1	6	4	8	1	2	3	3	1	2	2	2	2

Table 10: Data Model Elements Used by Each Metric

sometimes preferred over a valid metric". Counts such as the number of classes in a system do not necessarily contribute to the complexity of a system, and therefore are of limited use[HS96, KRW93]. Also, most of the defined metrics were not validated or based on measurement theory, nor were they extensively tested and applied to show their intuitive usefulness. Many researchers applied their defined metrics to small-scale experiments, and then hastily jump to conclude that the metrics are valid and significant. Henderson-Sellers[HS96] points out that most metrics were validated by a single experiment, which leads to the observation that "for every positive validation there is a negative validation". Moreover, even if some metrics were proven to be valid in some contexts, this would not mean that the metrics become applicable in other contexts[BMB94]. Kalakota et. al [KRW93] argue that "it is difficult to generalize any conclusions drawn from a given software project to all software projects", and Henderson-Sellers[HS96] mentions that "local validity does not imply global validity". As a possible remedy to the local validity problem, Brito e Abreu[BeA92] proposes to collect and analyze metrics "throughout time in as many different projects as possible". Hitz and Montazeri[HM95] refer to the fact that some researchers pick up the easily derivable numbers and then try to correlate them with external product attributes in an attempt to create metrics, and this leads to "attributes which do not necessarily have a causal relationship with the external variables considered".

Another problem lies in the attempt to combine several different metrics into one single measure of some quality aspect. Such metrics lack sensitivity[Bie96], and could become misleading, as one combined metric fails to capture all the aspects of a quality abstraction.

We also note that while it has been said that OO metrics have the advantage of being applicable early in the life-cycle, which generally means the analysis and design phase, we have noticed that many metrics sets cannot be derived until the source code is available. For instance, listing metrics such as "percentage of function-oriented code, average number of comment lines per method, average number of commented methods"[LK94] as being "design metrics" seems inappropriate and misleading; the same applies to the metrics "ratio of lines of code/method", and "ratio of lines of code/comment"[BTT].

This last point is also reflected in the tendency of some researchers to derive their design metrics from source code, probably due to the ease in the development of the tools; for instance, Stiglic et. al[SHR95] developed a tool which derives such metrics as the C & K[CK94] set – all of which being design metrics – out of C++ source code.

We are of the opinion that since the focus of OO metrics is on earlier phases of the development life cycle, at least theoretically, then the metrics collection methods/tools should also attempt to collect the metrics data from the earlier phases, specifically the design phase. The problem here is that the OO community has not yet settled on a standard notation to be used in design documents, so any data collection tool would have to apply only to particular contexts.[15] Also, metrics derived from source code are usually expected to show better accuracy than those drawn from design documents; so a tradeoff exists between getting early estimates of the quality of software products at the expense of having accurate measures of the quality. Nevertheless, we think that OO software metricians should derive their metrics and design their collection tools while keeping in mind the importance of computing the metrics on the earlier phases of the life cycle.

All this is not to say that the OO metrics community is in disarray; it is simply at the start of its evolution. We encourage researchers to keep deriving metrics and testing them, even if many would eventually turn out to be non useful. It is after a long period of trials and errors that we expect a "good" set of OO metrics to be available.

References

[AKM94] D. H. Abbott, T. D. Korson, and J.D. McGregor. A proposed design complexity for object-oriented development. Technical report, Clemson University, April 1994. TR 94-105.

[BBM96] V. Basili, L. Briand, and W. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10), October 1996.

[BeA92] F. Brito e Abreu. Object-oriented software design metrics. proc. of the OOPSLA' 92 workshop on OO metrics, 1992.

[BeAC94] F. Brito e Abreu and Carapua. Candidate metrics for object-oriented software within a taxonomy framework. *Journal of Systems and Software*, 26(1):87–96, July 1994.

[15] Perhaps the Unified Modeling Language (UML) might alleviate this problem.

[BeAM96] F. Brito e Abreu and W. Melo. Evaluating the impact of object-oriented design on software quality. In *Proc. METRICS' 96*, Berlin, Germany, March 1996. IEEE.

[Bie96] J. M. Bieman. Metric development for object-oriented software. In A. Melton, editor, *Software Measurement: Understanding Software Engineering*. Thomson Computer Press, Andover, Hampshire, UK, 1996. proceedings of the software metrics workshop held in conjunction with the 1992 ACM Computer Science Conference in Kansas City, Missouri.

[BMB94] L. Briand, S. Morasca, and V. R. Basili. Defining and validating high-level design metrics. Technical report, University of Maryland, 1994. UMIACS-TR-94-75.

[Boe81] Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[BS96] A. B. Binkley and R. S. Schach. Impediments of the effective use of metrics withing the oo paradigm. proc. of the OOPSLA' 96 workshop on OO metrics, 1996.

[BTT] D. Bellin, M. Tyagi, and M. Tyler. Object-oriented metrics : An overview. web site.

[CC92] J.C. Coppick and Cheatham. Software metrics for object-oriented systems. In J.P. Agrawal, V. Kumar, and V. Wallentine, editors, *Proceedings of the 20th Annual Computer Science Conference*, pages 317–322, Kansas, Missouri, March 1992. ACM, New York.

[CK91] S.R. Chidamber and C.F. Kemerer. Towards a metrics suite for object oriented design. In A. Paepcke, editor, *Proc. Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'91)*, pages 197–211, Phoenix, AZ, October 1991. ACM, New York.

[CK94] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[CK95] S.R. Chidamber and C.F. Kemerer. Authors' reply to : 'comments on : A metrics suite for object oriented design'. *IEEE Transactions on Software Engineering*, 21(3):265, March 1995.

[CL93] J.-Y. Chen and J.F. Lu. A new metric for object-oriented design. *Information and Software Technology*, 35(4):232–240, April 1993.

[CS95a] N.I. Churcher and M.J. Shepperd. Comments on : 'a metrics suite for object oriented design'. *IEEE Transactions on Software Engineering*, 21(3):263–265, March 1995.

[CS95b] N.I. Churcher and M.J. Shepperd. Towards a conceptual framework for object-oriented software metrics. Technical report, Dept of Applied Computing and Electronics, Bournemouth University, UK, 1995. internal report.

[Hal77] M. H. Halstead. *Elements of Software Science*. Elsevier North Holland, New York, 1977.

[HM95] M. Hitz and B. Montazeri. Measuring product attributes of object-oriented systems. In W. Schfer and P. Botella, editors, *Proc. ESEC '95 (5th European Software Engineering Conference)*, pages 124–136. Springer Verlag, September 1995.

[HM96a] M. Hitz and B. Montazeri. Chidamber and kemerer's metrics suite : A measurement theory perspective. *IEEE Transactions on Software Engineering*, 22, 1996.

[HM96b] M. Hitz and B. Montazeri. Measuring coupling in object-oriented systems. *Object Currents*, 1(4), 1996.

[HS91] B. Henderson-Sellers. Some metrics for object-oriented software engineering. In J. Potter, M. Tokoro, and B. Meyer, editors, *TOOLS 6: Technology of Object-Oriented Languages and Systems*, pages 131–139, Englewood Cliffs, NJ, 1991. Prentice Hall TOOLS Conference Series.

[HS94] B. Henderson-Sellers. Identifying internal and external characteristics of classes likely to be useful as structural complexity metrics. In D. Patel, Y. Sun, and S. Patel, editors, *Proc. 1994 International Conference on Object Oriented Information Systems OOIS'94*, pages 227–230. Springer-Verlag, London, December 1994.

[HS96] B. Henderson-Sellers. *Software Metrics*. Prentice Hall, Hemel Hempstead, UK, 1996.

[KRW93] R. Kalakota, S. Rathnam, and A. B. Whinston. The role of complexity in object-oriented systems development. In J.F. Nunamaker, Jr. and R.H. Sprague, editors, *Proceedings of the Twenty-Sixth Hawaii International Conference on System Sciences*, pages 759–768. IEEE Computer Society Press, January 1993.

[LH93] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of systems and software*, 23(2):111–122, 1993.

[LHKS95] W. Li, S. Henry, D. Kafura, and R. Schulman. Measuring object-oriented design. *Journal of Object Oriented Programming*, pages 48–55, July-August 1995.

[LK94] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice Hall, Englewood Cliffs, N.J., 1994.

[McC76] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[MD89] D.R. Moreau and W.D. Dominick. Object-oriented graphical information systems : Research plan and evaluation metrics. *Journal of Systems and Software*, 10:23–28, 1989.

[Mor89] Keneth L. Morris. Metrics for object-oriented software development environments. Master's thesis, MIT Sloan School of Management, May 1989.

[RC94] L.S. Rising and F.W. Calliss. Measuring coupling in object-oriented systems. *Journal of Systems and Software*, 26(4):211–220, 1994.

[SHR95] B. Stiglic, M. Hericko, and I. Rozman. How to evaluate object-oriented software development. *ACM SIGPLAN Notices*, 30(5):3–10, May 1995.

[STM91] S.D. Sheetz, D.P. Tegarden, and D.E. Monarchi. Measuring object-oriented system complexity. In *Proceedings of the First Workshop on Information Technologies and Systems WITS'91*, pages 285–307, December 1991.

[TSM92] D.P. Tegarden, S.D. Sheetz, and D.E. Monarchi. The effectiveness of traditional software metrics for object-oriented systems. In J.F. Nunamaker, Jr. and R.H. Sprague, editors, *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, pages 359–368. IEEE Computer Society Press, January 1992.

[Wey88] E. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, September 1988.

[YHS93] L. M. Yap and B. Henderson-Sellers. Consistency considerations of object-oriented class libraries. Technical report, University of New South Wales, Sydney, Australia, 1993. research report no 93/3.