# Correspondence

## Comments on "A Metrics Suite for Object Oriented Design"

Neville I. Churcher and Martin J. Shepperd

*Abstract*—**A suite of object oriented software metrics has recently been proposed. While the authors have taken care to ensure their metrics have a sound measurement theoretical basis, we argue that is premature to begin applying such metrics while there remains uncertainty about the precise definitions of many of the quantities to be observed and their impact upon subsequent indirect metrics. In particular, we show some of the ambiguities associated with the seemingly simple concept of the number of methods per class. The usefulness of the proposed metrics, and others, would be greatly enhanced if clearer guidance concerning their application to specific languages were to be provided. Such empirical considerations are as important as the theoretical issues raised by the authors.**

*Index Terms*—**CR categories and subject descriptors: D.2.8 [software engineering]: metrics; D.2.9 [software engineering]: management; F.2.3 [analysis of algorithms and problem complexity]: tradeoffs among complexity measures; K.6.3 [management of computing and information systems]: software management. General terms: Class, complexity, design, management, metrics, object orientation.**

### I. Introduction

In the above paper,[1] Chidamber and Kemerer (CK) proposed a set of six software metrics for Object Oriented (OO) software systems and presented data collected from a field study [3]. Earlier work by CK [2] is, deservedly, among the most frequently cited in the emerging field of OO software metrics and empirical studies beginning to appear [6].

We strongly support CK's attempts to develop theoretically sound metrics appropriate to OO software rather than simply seek ways to apply existing conventional software metrics. However, we are concerned that it may be premature to proceed with complex indirect metrics until a number of fundamental issues have been clarified.

Software metrics have been used for some two decades to assess or predict properties of conventional software systems, but success has been limited by factors such as the lack of sound models, the difficulty of conducting controlled, repeatable experiments in commercial or military contexts, and the inability to compare data obtained by different researchers or from different systems. The development and acceptance of software metrics as a discipline has been hampered by the indifferent quality of some experimental work and a lack of standardization of techniques and terminology. An example is the "software science" family of metrics [4], where the absence of precise definitions of counting rules severely limited the value of results [5], [7].

OO software differs from its conventional counterparts in a number of significant ways, and familiar concepts, such as coupling between components, are much less straightforward to define [1]. The greater number of system components and their interactions present software metricians with some major challenges, and little real progress has been made to date.

In our own work on OO metrics, we have found the lack of standard terminology and practices to be a major obstacle to ready comprehension and comparison of the work of others. The varied nature of OO languages means that great care must be taken in comparing results. We believe that the ad hoc approach that has characterized conventional software metrics work must be avoided to enable progress to be made on the more challenging problem of developing OO metrics. This suggests that empirical and theoretical considerations should receive equal weighting in the development of new metrics.

We illustrate our concerns by considering a fundamental property, the number of methods in a class, and demonstrate that this is open to a variety of interpretations. We give a specific example from a study of a C++ system that we are currently undertaking. The number of methods per class is required directly for the computation of the weighted methods per class (WMC) metric and indirectly for others [3].

Composite or indirect metrics will be compromised if there is uncertainty in the definitions of the primitive counts, such as number of methods, from which they are constructed. While there is no ideal counting strategy, it is important to make explicit the technique actually employed, as misleading results may be obtained if like is not compared with like. Now is the time to address such issues, before it becomes too late to apply to OO metrics the lessons we should have learned from the history of conventional metrics.

### II. Example: Number of Methods

Consider the following C++ class definition, taken from an industrial system we are currently analyzing. Although some of the points raised in this section are specific to C++, similar issues arise in other OO programming languages.

```
class BitSet: public Object {
protected:
   u_long m;

   BitSet (const BitSet &);
   BitSet (u_long i, double);
public:
   MetaDef (BitSet);

   BitSet ();
   BitSet (int i1);
   BitSet (int i1, int i2);
   BitSet (int i1, int i2, int i3);
   BitSet (int i1, int i2, int i3, int i4);
   BitSet (int i1, int i2, int i3, int i4,
           int i5);
   BitSet (int i1, int i2, int i3, int i4,
           int i5, int i6);
   BitSet (int i1, int i2, int i3, int i4,
           int i5, int i6,int i7);
```

Manuscript received July 1994. Recommended by S. H. Zweben.

The authors are with the Department of Applied Computing and Electronics, Bournemouth University, Talbot Campus, Fern Barrow, Pole, Dorset BH12 5BB, England (e-mail: mshepper@bournemouth.ac.uk or neville@cosc.canterbury.ac.nz).

IEEE Log Number 9409036.

[1] S. R. Chidamber and C. F. Kemerer, *IEEE Trans. Software Eng.*, vol. 20, pp. 476–493, June 1994.

```
    BitSet (BitSet *n);

    BitSet operator~();
    BitSet operator-(BitSet n);
    bool operator>(BitSet n);
    bool operator<(BitSet n);
    bool operator>=(BitSet n);
    bool operator<=(BitSet n);
    bool operator==(BitSet n);
    bool operator!=(BitSet n);
    BitSet operator&(BitSet n);
    BitSet operator^(BitSet n);
    BitSet operator|(BitSet n);
    void operator-=(BitSet n);
    void operator&=(BitSet n);
    void operator^=(BitSet n);
    void operator|=(BitSet n);

    int AsMask ();
    bool Includes (int i);
    bool Contains (int i);
    unsigned long Hash ();
    bool IsEmpty ();
    bool IsEqual (Object*);

    OStream& PrintOn (OStream&);
    IStream& ReadFrom (IStream&);
    int Size();
    int Capacity ();
};
```

This class defines an apparent total of 37 methods including 11 constructors and 15 operators. In fact, `MetaDef` is a parameterized preprocessor macro that defines several additional members. Consequently, different results may be obtained depending on whether or not data are gathered before (as in this example) or after preprocessing.

Each method has a unique signature, determined by combining its identifier, parameters, and return type. When developing metrics concerned with the services offered by an object (i.e., the messages to which it may respond) it is likely that only distinct method names will be considered, while metrics aimed at predicting maintenance effort are more likely to include all the constructors separately by counting signatures. In the `BitSet` class, this choice could change the number of methods from 37 to 27. This phenomenon is by no means limited to constructors.

The treatment of operators raises further issues. Since the methods that implement operators are seldom invoked explicitly, service-oriented metrics may omit them when counting methods. This would lower the method count by 15 for the `BitSet` class. Methods corresponding to operators may also be defined multiple times with the same name but different signatures—perhaps to allow mixed-mode expressions.

Several other factors may influence the number of methods counted, and the treatment of inherited methods is perhaps the most crucial. To count methods, we must answer the fundamental question, "Does a method belong only to the class that defines it explicitly or does it also belong to every class that inherits it directly or indirectly?"

One possibility is to restrict counting to the current class, ignoring inherited members, as we have done so far in this section. The motivation for this is that inherited members have already been counted in the classes where they are defined, so the class increment is the best measure of its functionality—what it does reflects its reason for existing. To understand what a class does, the most important source of information is its own operations. If a class cannot respond to a message (i.e., it lacks a corresponding method of its own) then it will pass the message on to its parent(s), and this fact should be captured by metrics concerned with the call structure.

At the other extreme, counting could incorporate methods defined in the current class, together with all inherited methods. This approach emphasizes the importance of the state space, rather than the class increment, in understanding a class. If inherited methods are included then features of derived classes may be obscured. The only parent class of `BitSet` is the top-level class `Object`, which defines a further 67 methods. However, the results of our study show a number of classes that define only a handful of methods themselves while inheriting over 300 methods. If inherited members are regarded as belonging to each generation of derived classes, then such numbers would become enormous.

If we identify $n$ binary choices to be made in order to specify the counting strategy, then there are $2^n$ possible ways of counting methods for each class. The two such choices we have identified (unique names and operator inclusion) lead to the four values shown in the following table.

| | | |
|---|---|---|
| $M_{11}$ | all defined methods, operators included | 37 |
| $M_{01}$ | unique names, operators included | 27 |
| $M_{10}$ | all defined methods, operators excluded | 22 |
| $M_{00}$ | unique names, operators excluded | 12 |

Allowing different strategies for each level of inheritance further increases the number of potential counting strategies.

Inheritance raises further issues requiring resolution. Where a class overrides an inherited method it is possible, in C++, to access either method by including sufficient information about the scope of the target method in messages. If a class can respond to a message requesting the invocation of an ancestor's method then we may wish to record the fact via appropriate metrics. Does the class actually "have" both methods?

Counting might be restricted to visible methods (i.e., those that appear in the class interface) whether inherited or not. The rationale for this approach would be that since the clients of a class see only the members that constitute its interface, and not its implementation, then it is the interface that should form the basis for metrics—particularly those that aim to assess coupling.

A complication is that a class may present several different interfaces. For example, a derived class may have access to a greater number of its parents' methods than other classes and a C++ "friend" class may subvert the normal access rules. Apparently bizarre situations may arise. For example, a C++ class may be regarded as "having" the private members of its parent class, but it cannot access them via its own methods. Should such methods be counted?

It is clear that there are many different ways to count methods, just as there are many ways to count lines of code. Similar flexibility exists for the definitions of other fundamental properties. There is no "correct" definition; rather, the precise form appropriate to particular circumstances must be specified. This is particularly important where the quantities measured are then used to derive indirect metrics.

## III. Conclusion

A number of factors potentially affect the data values collected from OO systems, their interpretation, and subsequent usefulness. We have shown that there are competing interpretations of the precise meaning of a key primitive count in the CK suite of metrics. We do not doubt that CK have resolved, to their own satisfaction, many of

the points we raise, and it may well be that the information we seek has become a casualty of the pressure on space in academic journals. Nevertheless, it is vitally important to precisely specify the mapping from a language-independent set of metrics to specific programming languages and sets of observations. Such precision is particularly important since the source code used in published work is generally not publicly available. The usefulness of the proposed metrics (and others) will be limited until their application to specific languages is clearly specified. Failure to resolve such issues in the near future may impede the development and validation of effective OO software metrics. We believe that this is as important as the establishment of a sound theoretical basis for the metrics.

## REFERENCES


[1] E. V. Berard, "Object coupling & object cohesion," in *Essays on Object-Oriented Software Engineering*. Englewood Cliffs, NJ: Prentice-Hall, 1993.

[2] S. R. Chidamber and C. F. Kemerer, "Toward a metric suite for object oriented design," in *Proc. OOPLSA '91*, ACM, 1991.

[3] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Software Eng.*, vol. 20, pp. 476–493, 1994.

[4] M. H. Halstead, *Elements of Software Science*. Amsterdam, The Netherlands: Elsevier North-Holland, 1977.

[5] J.-L. Lassez *et al.*, "A critical examination of software science," *J. Syst. Software*, vol. 2, no. 2, pp. 105–112, 1981.

[6] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *J. Syst. Software*, vol. 23, pp. 111–122, 1993.

[7] V. Y. Shen, S. D. Conte, and H. E. Dunsmore, "Software science revisited: A critical analysis of the theory and its empirical support," *IEEE Trans. Software Eng.*, vol. 9, pp. 155–165, 1983.


## *Authors' Reply*[2]

Shyam Chidamber and Chris F. Kemerer

The main thesis of the Churcher and Shepperd comment is that it is important that the software metrics be clearly defined in order that other researchers can replicate the results, a point that we, of course, completely agree with. Therefore, we view this response to their comment not as a rebuttal, but merely as an opportunity to provide additional detail and insight to our thinking about how we developed the metrics for the readership of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING who may have interest in object-oriented metrics.

Churcher and Shepperd focus on the calculation of the number of methods per class, and suggest approximately half a dozen binary questions about what methods should either be included or not in the count. They then note that a potentially large number of different possible answers can be generated by combining all the possible combinations of answers to these questions

In our view, the entire list of questions can be answered by reference to a simple principle that methods which required additional design effort and are defined in the class should be counted, and those that do not should not. Therefore, we would count all the distinct methods and operators in the class even when some share a name identifier or even when they are not interface methods, and we would count these prior to preprocessing, consistent with the role of WMC as a design metric. We would not count indirect methods available through ancestors, friends, (a C++ construct), or any inherited methods, as these are defined outside the class. (See the portion of our definition of WMC on p. 482 of the original article, as follows: "...Consider a Class $C_1$ with methods $M_1 \cdots, M_N$ that are defined in the class. Let...") We have provided other metrics in the suite which directly relate to notions of complexity arising from inheritance.

We find this principle to be relatively straightforward in its application and consistent with designers' intuitions about the complexity of a class. Additional evidence for this may be found in the fact that our value for the Churcher and Shepperd example is 37 methods, which is the same value that they determine before they begin discussing some hypothetical options.

We are encouraged that Churcher and Shepperd are exploring the use of some of our metrics in their own research, and greatly appreciate their effort in considering alternative formulations for object-oriented metrics. In fact, we would like to conclude by emphasizing a point made in our original article: while we propose a particular suite of six metrics, there is no reason to believe that these six will ultimately be found to be comprehensive. Further work by ourselves or others may result in additions, changes, or even possible deletions from this suite, particularly as at the current time the suite has been subject to only limited empirical observation. We welcome questions and comments from the software engineering community that will enable the further explication and refinement of our metrics suite.

[2]Manuscript received July 1994. Recommended by S. H. Zweben.

The authors are with the Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA 02142 USA.

IEEE Log Number 9409035.

## Correction to "A Practical Approach to Programming with Assertions"

D. S. Rosenblum

In the above paper,[1] a printing error resulted in the incorrect publishing of line 7 of Fig. 3 on p. 22.

The incorrect line read:

```
&& all (int i=0; i < in size-1; i=i+1) S[i] <= S[i+1] // S is ordered.
```

The correct line should read:

```
&& all (int i=0; i < in size-1; i=i+1) S[i] < S[i+1] // S is ordered.
```

Manuscript received February 6, 1995.

The author is with AT&T Bell Laboratories, Murray Hill, NJ 07974 USA (e-mail: dsr@research.att.com).

IEEE Log Number 9410122.

[1]D. S. Rosenblum, *IEEE Trans. Software Eng.*, vol. 21, pp. 19–31, Jan. 1995.