Towards a Conceptual Framework for Object Oriented Software Metrics

Neville I. Churcher and Martin J. Shepperd

Department of Applied Computing & Electronics, Bournemouth University,
Talbot Campus, Fern Barrow, Poole, Dorset BH12 5BB
email: mshepper@bournemouth.ac.uk or neville@cosc.canterbury.ac.nz

ABSTRACT

The development of software metrics for object oriented (OO) languages is receiving increasing attention. We examine the reasons why this is a much more challenging problem than for conventional languages. It seems premature to develop and apply OO metrics while there remains uncertainty not only about the precise definitions of many fundamental quantities and their subsequent impact on derived metrics, but also a lack of qualitative understanding of the structure and behaviour of OO systems. We argue that establishing a standard terminology and data model will help provide a framework for both theoretical and empirical work and increase the chances of early success. One potential benefit is improvement of the ability to perform independent validation of models and metrics. We propose a data model and terminology and illustrate the importance of such definitions by examining the seemingly straightforward concept of the number of methods per class. We discuss the implications of ambiguities in definitions for a suite of metrics which has recently been proposed. Preliminary results from our analysis of industrial systems are presented.

1. *INTRODUCTION*

The idea that software systems, and the processes used to produce them, constitute an engineering discipline has been accepted for some two decades. This has led to attempts to assess and predict properties of software products and processes through the use of software metrics. 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 difficulty of comparing data obtained by different researchers or from different systems. Such problems are not specific to metrics and an overall improvement in standards throughout the wider discipline of software engineering is becoming more widely advocated [8].

While some difficulties are inevitable, given the scope and complexity of the subject, 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 standardisation of techniques and terminology. For example, the usefulness of the 'software science' family of metrics [11] was severely limited by the lack of precise definitions of counting rules [13, 15]. It is desirable to avoid repeating the mistakes made in the past as we move on to tackle the significantly more complex area of OO systems.

Over the past decade, OO analysis and design techniques, languages, databases and other systems have become prevalent. In particular, C++ is rapidly becoming the default language for software development (despite not being 'ideologically sound') and Smalltalk is also becoming common in a wider range of application domains. Conventional wisdom holds that OO models lead to designs which correspond better to the structure and semantics of the problem domain and that more maintainable implementations follow from the separation of interface and implementation. However, little is really known about the processes involved, and empirical evidence to support such beliefs is in short supply. In fact, there is reason to believe that maintenance of OO software is far from straightforward, poses a number of difficulties and requires considerable tool support [14, 16].

Object-oriented software systems differ from their conventional counterparts in a number of significant ways, presenting some major challenges, and little progress has been made to date. There is a growing need for techniques which can lead to some immediate advances, at least for common languages like C++ and Smalltalk. However, relatively

little data on the structure of OO systems is available and even qualitative models of the quality of designs are rare. Only a few OO metrics have been proposed and most studies to date are based on the set of 6 metrics of Chidamber and Kemerer (CK) [4, 5].

In the absence of sound models for OO metrics, one common approach is to simply apply 'length' and 'complexity' metrics which were developed for conventional languages, treating methods as corresponding to procedures. However, a major difference between OO and conventional language is the size of procedural components. There are many reasons for this, such as polymorphism eliminating the need for much multi-way selection code. Experience shows that methods are typically very short, consisting of only one or two executable statements, while procedures in conventional languages are likely to consist of tens of statements. For example, one recent study found over 50% of methods consisted of 2 or fewer C++ statements or 4 or fewer Smalltalk non-commentary lines of code [17]. This suggests that traditional metrics, such as cyclomatic complexity, are unlikely to be particularly useful in OO systems, apart from highlighting the most gross outliers. We should therefore be extremely cautious about trying to extend into the OO domain metrics and metric tools which were designed to work with conventional systems.

The field of software metrics for OO systems is currently in a state of flux and, apart from the CK metrics, few clear directions have emerged. Even relatively fundamental concepts are subject to a variety of different interpretations. We believe that it is important for a coherent approach to OO metrics to be developed. This requires a greater emphasis on empirical matters, not only to avoid reproducing the obstacles created by the *ad hoc* approach for conventional metrics, but also to enable progress to be made on the more challenging aspects of OO systems while responding to the pressure for immediate results. There are indications that the customary static analysis techniques may not be sufficient to capture fully the dependencies of OO systems [16] in which case a radically different approach to metrics would be required.

The remainder of this paper is organised as follows. In the next section we introduce a set of OO terms and concepts which are used in section 3 to discuss models of OO systems and metrics. In section 4 we discuss a number of factors which make the development of software metrics for OO systems a significantly more complex undertaking than for conventional systems. The importance of precise terminology is illustrated in section 5 by the example of counting methods in a class and in section 6 we present some preliminary results from a study of industrial C++ software. In section 7 we outline the potential impact of imprecision in mapping from the language-independent metrics proposed by Chidamber and Kemerer to specific programming languages.

2. *TERMINOLOGY*

There are many variations on the OO theme and a large number of OO terms and concepts may be found in the literature or in texts. The vocabulary, level of formality and mathematical sophistication appropriate to the communities involved with aspects of research, implementation and metrication differ and are subject to rapid change, making it difficult to standardise terminology. Nevertheless, we seek to establish an implementation-independent terminology which includes enough common concepts to form the basis for a description of OO systems and metrics and which can be extended as required. The value of an implicit underlying model derives from the need to specify explicitly only differences or extensions, thus providing a flexible and extensible framework for ongoing development.

The following have been selected for their generality and applicability to metrics.

system The entire software system under consideration. In the typical case of static analysis, the system may be described completely by source code or design documents. A system may include modules in different phases of development (e.g. design, coded) or in different languages (e.g. C, & C++). A system may include some non-OO components (e.g. C-like functions in C++).

component Any system entity whose properties may be measured. The most important components are typically classes, methods and state variables.

ACM SIGSOFT Software Engineering Notes vol 20 no 2 April 1995 Page 69

connection Instance of relationship between components. e.g. 'has method' or 'sends message to'.

class Compound structure encapsulating members.

object Instance of a class. Each object has access to the members of the corresponding class. Persistent objects exhibit state through the values of state variables.

member Procedure, or data item (type definition etc.) encapsulated within a class.

state variable Data item encapsulated by a class. Other names: data member, instance variable, attribute.

method A member which is a procedure or function. Methods may have return types and parameter lists. The body of a member may be supplied within the class definition (as for C++ `inlines`) or be provided elsewhere.

encapsulation Mechanism for specifying or restricting access to class members by other classes. This normally involves assigning access categories to each member. Those members visible outside the class may be referred to as the class interface.

access mode Specification of visibility of members and their behaviour under inheritance. Typical values are 'public' and 'private'.

operator Some languages, such as C++, allow special methods to be defined in order to overload the arithmetic and other operators. A given operator may be implemented by several methods even within the same class—perhaps in order to satisfy the constraints imposed by strong typing.

constructor A method, whose name is generally that of the host class, which creates and initialises instances of the class. Several different constructors, differing in their argument lists, may be implemented for a class to allow for different initialisation or dynamic storage allocation requirements.

function A procedure or function which is not a member of a class. For example, C++ allows calls to 'ordinary' C-like functions, either library functions or user defined. Other names: procedure, subprogram, module.

signature Unique identifier (typically a combination of name, scope, parameter list and return type) that identifies a 'real' method. Compilers produce signatures for internal use (e.g. 'mangled identifiers' in C++).

inheritance A class may be derived from another (base) class. Members of the base class are inherited by (available to) the derived class. Mechanisms involving access modes are used to limit access to inherited members and to restrict their subsequent inheritance by further derivation. Other names: generalization/specialization, subclass/superclass.

inheritance tree Graph in which nodes represent classes and edges represent base class/derived class dependencies. The graph may not be a tree if multiple inheritance is permitted. There will be a root, though this may be some system class. Other names: inheritance hierarchy, inheritance structure, inheritance graph.

parent, child A child class is derived directly from a base class i.e. there is an edge in the inheritance graph from the parent class to the child class.

ancestor, descendant A descendant class is derived from an ancestor class either directly (i.e. they are child & parent) or indirectly (i.e. there is a path in the inheritance graph from ancestor to descendant.

depth The inheritance graph has a root and the depth of a class is the length of the longest path to it from the root. In some languages, such as Smalltalk, all user-defined classes are (perhaps implicitly) derived from system classes such as 'object' or 'vanilla'. When discussing depth in the inheritance structure it is important to state whether such classes are included. Wilde et al. [17] report that this effect explains the apparent differences in depth for their Smalltalk and C++ data.

virtual method A virtual method has no implementation and is implemented by derived classes. Its presence enforces common functionality among derived classes.

abstract class Class designed to have no instances, but rather to allow derived classes to inherit common properties. Likely to have many virtual methods. Other names: generic.

scope Class to which the component 'belongs', usually the class in which it is declared. An artificial 'system' scope may be required for consistency. Other names: host class.

nested class Class defined within another class i.e. class whose scope is not 'system'.

override A class may contain methods, including those implementing operators, having the same identifiers as those inherited from ancestors. Such methods override those defined in ancestor classes, and will be the methods invoked in response to messages.

state space The set of members a class has. Includes inherited members.

message Classes, via their methods, send messages to request services from other classes, possibly including specific recipients and parameters. Ultimately, a method will respond to the message. This is the OO analogue of procedure calls in conventional languages. Features such as polymorphism make it difficult (sometimes impossible) to establish the correspondence between calling (message sending) and called (responding) methods by static analysis alone.

polymorphy An identifier may refer to instances of different classes (typically having a common ancestor) at run-time, allowing objects bound to this identifier to respond to the same set of messages in different ways.

class increment Those members defined explicitly in a class and not inherited from ancestor classes.

3. *Modelling Architecture*

In modelling the OO architecture it is appropriate to use the relational data model [7] and to describe more abstract concepts in terms of the entity-relationship conceptual data model [3] and its extensions (e.g. [1]). While it might well be more aesthetically pleasing to use an OO data model, there are a number of arguments in favour of the more standard approach, not the least of which is portability. The relational model is sufficiently standard that, despite its limitations, it provides a sound platform for development of portable models and tools. Furthermore, it has a straightforward mapping onto the flat files typically used by software tools and thus provides a lowest common denominator approach to communication.

Given the terms discussed in section 2 we can now begin to construct a simplified model of an OO architecture. We propose a model whose two major constituents are 'components' and 'connections'. Components may be of various kinds, such as classes, clusters or methods, but may have properties in common such as id, name and scope. Connections represent the various dependencies between components, such as method invocation (method/method) or inheritance (class/class). These represent the ways in which components 'use' each other by mechanisms such as procedure invocation or data access.

The following table shows some of the binary relationships between components which our model must be capable of representing, directly, or indirectly. While the precise details of the number of components and their interactions depends on the model or language under consideration, such a binary interaction model is capable of representing the fundamental structure of many systems.

	Class	Method	Variable	Message
Class	parent-of ancestor-of nested-within	defines-method inherits-method uses-method	defines-variable inherits-variable uses-variable	responds-to sends
Method	defined-in inherited-by visible-to	invokes-method overrides-method in-response-set-of	uses-variable has-parameter	responds-to
Variable	defined-in inherited-by used-by	used-by-method parameter-for-method	hides	is-parameter-of
Message	received-by sent-by	invokes-response	has-parameter	propagates

These may be separated into two major categories: those, such as 'parent-of', representing aspects of the inheritance structure of the system and those, such as 'invokes-method', representing the calling (message) structure.

We consider two approaches to forming data models of the OO architecture. In an explicit model, each component type is represented separately leading to a model consisting of relations such as the following:

system(ID, Name)
class(ID, Name, System, HostClass)
method(ID, Name, HostClass, AccessMode, ReturnType)
statevar(ID, Name, HostClass, AccessMode, Type)
parameter(ID, Name, Method, Type, Usage)
ancestry(ID, ParentClass, ChildClass)
invocation(ID, CallingClass, CallingMethod, CalledClass, CalledMethod)
stateaccess(ID, Method, Usage)

By contrast, an implicit model may exploit the common properties of the various components and connections, leading to a model featuring 'generic' components and connections, consisting of relations such as:

system(ID, Name)
component(ID, Name, HostClass, kind, access)
connection(ID, accessor_component, accessee_component, usage_type)
(NB: problems with attributes of connections e.g. access mode of has_member)

By using more sophisticated metadata models, such as those developed for CASE repositories [18], it is possible to handle properties of connections and connections of higher degree in an elegant manner. A more detailed discussion of the importance of data models in OO software metrics is beyond the scope of this paper.

We have employed such an implicit model in our work as it is more straightforward to extend the model to cope with new component and connection types.

It is possible to represent such a data model in a BNF form as shown below:

<system>	::=	<component>*
<component>	::=	<class> | <member> | <function> | <data_item>
<class>	::=	<identifier> <scope> <heritage> <member>* <connection>*
<member>	::=	<state_variable> | <method> | <nested_class> | ...
<method>	::=	<procedure_method> | <operator_method>
<connection>	::=	<structural_connection> | <invocation_connection>
<structural_connection>	::=	<inheritance_connection> | <attachment_connection> ...
<invocation_connection>	::=	<message> <respondent> <parameter_list>
<inheritance_connection>	::=	<parent_class> <child_class> <access_mode>
...	...	...

Metrics are then added into the model in a natural way. Fundamental measurements will involve properties of components and connections. Should more complex quantities be required, such as clusters of interactive objects, then these are simply introduced as new component types with appropriate new connections to denote membership.

<metric>	::=	<metric_ID> <metric_name> <metric_specification>
<measurement>	::=	<metric_ID> <property> <metric_value>
<property>	::=	<component_property> | <connection_property>
...	...	...

4. *CHARACTERISING OO SYSTEMS*

Conventional and OO systems differ in many ways. While the benefits of OO approaches are apparent, so too are the pitfalls for those who design, maintain and measure OO

software. Measuring apparently basic concepts such as coupling and cohesion poses considerable problems for software metrics in conventional systems. The corresponding concepts for OO systems are considerably more complex [2].

Results from recent studies indicate that methods tend to be small, both in terms of number of statements and in logical complexity [17] suggesting that the connectivity structure of the system may be more important than the content of individual methods. In OO systems this has contributions from 'architectural' structure, via mechanisms such as inheritance as well as from 'call-graph' structure. Investigations of the maintainability of OO software have raised several issues which are indicative of the major differences between OO and conventional software metrics [14, 16].

The definitions of program structures tend to be dispersed by features such as inheritance, making it difficult to comprehend fully local code artefacts. Dynamic (late) binding is one of the great advantages of OO languages, yet poses very real problems for programmers who must work with inherently static source code as is can be difficult—sometimes impossible—to correctly identify all relevant component interactions by static analysis alone. The resulting comprehension difficulties of both high level system structure and of component details must surely be of great relevance to the development of OO metrics.

There may be some cause for optimism in the fact that this structure is largely available early in the development process. Since the structure is available at design time and the implementation of methods leads to a fairly uniform distribution of properties of methods, it may well be possible to develop effective predictive metrics based on the assumption that the structure of the implementation reflects that of the design more closely than is the case for conventional languages.

A conventional system may be described in terms of a relatively small number of component and dependency (relationship) types. The major component type is the procedure (or function, module, subprogram), though components of larger granularity (e.g. scopes or modules) or smaller granularity (e.g. individual data structures or variables) are required for some purposes. A typical data model for conventional systems [9, 10] contains relations such as:

module(id, name, location, return_type, ...)
datum(id, name, scope, type, ...)
call(caller_id, callee_id, ...)
parameter(datum_id, module_id, read_write, ...)

Such a model is sufficient to provide an adequate description of a conventional system. The most significant feature is the N:M self-relationship 'module-calls-module'. This relationship enables many typical inter-module metrics (e.g. [12]) to be computed.

As is clear from the preceding discussion, a corresponding data model for OO systems is significantly richer and contains more such relationships. These include structural N:M self-relationships such as 'class-is-parent-of-class' and 'method-overrides-method' as well as those concerned with the calling hierarchy such as 'method-sends-message'. The number of component and dependency types relevant to OO systems is significantly greater. Some of these are given by [16].

In a conventional system the name used to invoke a module is the name of the module itself. By contrast, an object's methods causes procedures (methods) to be invoked by sending messages to other objects. The actual method invoked in response to a given message is not necessarily fixed uniquely until run-time. In some languages, such as C++, the syntax is that of a function call. However, this can be deceptive—a message will involve a message name, and may involve other parts of the signature directly (e.g. parameters) or indirectly (e.g. access mode or return type). Wherever the message cannot be directly mapped onto a unique signature by static analysis alone, e.g. because of polymorphism, there exists a potential difficulty for those involved in metrics and maintenance.

The distinction between methods and messages is particularly important when deciding the status of inherited methods. One possible interpretation is that inherited methods really are part of a class—another is that classes simply pass outstanding messages to their ancestors for possible resolution.

From a metrics viewpoint we are interested primarily in the calling method (message sender) and the actual method invoked. Where this may not be determined uniquely from available information we wish to know the set of possible respondents. Even where a message is sent to an object of a specific class, the method invoked may actually be one inherited from an ancestor class of the message recipient.

Methods are invoked in response to 'messages' sent by objects. Some methods may implement operators, if this is permitted by the language. We can think of each method as having a unique 'signature' which includes its name, host class (scope), return type, access mode and parameters. Our model must be capable of recording such signatures. A given class may implement several methods identical except for their parameter lists. Examples in C++ include constructors implementing differing initialisation behaviour and different implementations of the arithmetic operators permitting mixed-mode operations involving objects of a class and variables of the standard numeric types.

5. Example: Counting Methods

One of the most frequently reported OO metrics is the number of methods per class. This is an intuitively appealing, and apparently simple, direct measurement of the 'complexity' of a class. Many other metrics, such as the CK set, indirectly use the number of methods.

Clearly, there is no single 'right' way to count methods. However, it is important that precise counting rules be used, and stated, in order that empirical results be reproducible. Aspects of counting rules to be specified include, inheritance, access mode and treatment of operators. Should one count the methods that a class has or the messages to which it can respond? How do these differ?

We discuss aspects of method counting for C++ elsewhere [6].

Inheritance

In order to count methods, we must answer the fundamental question 'Does a method belong only to the class which defines it, or does it also belong to every class which inherits it directly or indirectly?' Questions such as this may seem trivial since the run-time system will ultimately resolve them. However, the implications for metrics may be significant.

One possibility is to restrict counting to the current class, ignoring inherited members. The motivation for this would be 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. In order 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).

At the other extreme, counting could include all methods defined in the current class, together with all inherited methods. This approach emphasises the importance of the state space, rather than the class increment, in understanding a class.

Between these extremes lie a number of other possibilities. For example, one could restrict counting to the current class and members inherited directly from parent(s). This approach would be based on the argument that the specialisation of parent classes is the most directly relevant to the behaviour of a child class.

In the most general case, a count of the number of methods involves the inclusion of members of the current class and those inherited from some number of generations of ancestors. We might then denote the number of methods for class c including d generations of ancestors by

$$m_d(c), (0 \le d \le \infty)$$

where $d = 0$ means only methods defined in the current class are included and $d = \infty$ means all ancestors are included.

Access Mode

Counting could be restricted to visible methods—those which appear in its interface. Since the clients of a class see only the members which constitute its interface, and not its implementation, then it is the interface which should form the basis for metrics—particularly those which 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?

Overriding

Where a class overrides an inherited method it is often possible, as 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? Both may need to be understood by maintainers even if only one is 'really' present.

Message Degeneracy

Similarly, where a class has more that one method which can respond to a given message (name), and the method to be invoked is determined by the parameter list or other data sufficient to determine a unique signature, then the number of possible respondents may be of interest. This situation commonly arises when considering constructors or operators.

Operators

For particular purposes it may, or may not, be appropriate to include methods which implement operators in method counts. Where they are included it may also be necessary to decide how degeneracy is to be handled.

Notation

It is desirable that the notation used for metrics be uncluttered and usable. However, as we have indicated, there are many possible interpretations even for such 'obvious' quantities as method count.

One approach is to explicitly state the assumptions implicit in such quantities and then proceed to use simple notation such as m for method count. However, the effects of factors such as those mentioned in the previous sections may well themselves be the subjects of measurements, in which case a more elaborate notation is required.

The following table includes some suggestions for suitable terms which might be used to indicate particular combinations of factors chosen:

Concept	*Candidate Terms*	*Note*
Inheritance	Gross, full, extended	All generations included.
	Nett, compact	Only the local increment included.
	n-extended	n generations included
Access Mode	restricted	Language-dependent restrictions on access
	unrestricted	Access mode ignored
Overriding	inclusive, distinguished	Both overriding and overridden method included
	exclusive, non-distinguished	Overridden methods excluded
Message Degeneracy	condensed	Treating degenerate members as one.
	expanded	Treating degenerate methods separately
Operators	augmented	Including operators.
	reduced	Excluding operators

A notation capable of denoting these choices might require the number of methods for class c to be written

$$m^{e}_{i,j}(c)_d$$

ACM SIGSOFT Software Engineering Notes vol 20 no 2 April 1995 Page 72

where *i* and *j* take values '+' or '-' and represent overriding and message degeneracy factors respectively, the superscript *o* indicates that operators are explicitly included, and *d* is as before.

6. *Sample Data*

We are currently analysing a number of industrial C++ systems. There are many different length metrics for conventional languages, based on various ways to count lines of code, but such metrics typically exhibit strong correlations both with each other and with other metrics such as cyclomatic complexity. In this section we present some preliminary results to demonstrate that the precise definitions of equally fundamental counts for OO languages do actually lead to distinguishable results.

While space allows us to present only a few results, we believe that such data helps in the formation of the qualitative understanding of the 'typical' structure of OO systems which is an essential prerequisite for future development of quantitative models.

The data is taken from a relatively small system called OMS, which is a library providing run-time support (persistence, transaction management etc.) for applications. It consists of 95 classes, and defines 998 methods and 20 functions.

Figure 1 shows the 95 classes arranged in decreasing order of the number of methods defined explicitly (including duplicates and operators), together with the corresponding number of available methods (i.e. defined or inherited). Given a sufficiently precise definition of the method counting rules used, the relationship between these two measures is determined. Clearly, their relationship is not a simple correlation and it would be quite wrong to substitute one for the other in the computation of indirect metrics based on method counts.

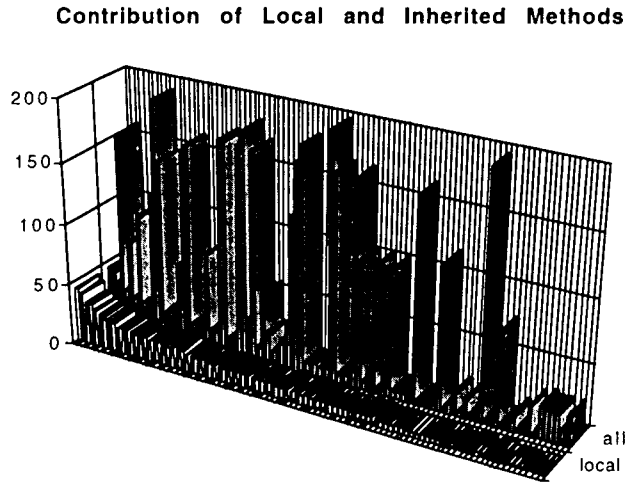

Figure 1. Contribution of inherited methods

Similar, though typically less pronounced, effects appear when the contributions from the other factors discussed above, such as duplicate method identifiers, are considered. For example, one class features one method identifier defined 24 times and another defined 20 times. Figure 2 shows the effects of distinguishing methods within a class by identifier or by signature. While some correlation is evident, the proportion of methods affected by this decision is high in some classes. Failure to specify precisely the counting rules used introduces unnecessary uncertainty into derived metrics and lowers the usefulness of the results to other researchers.

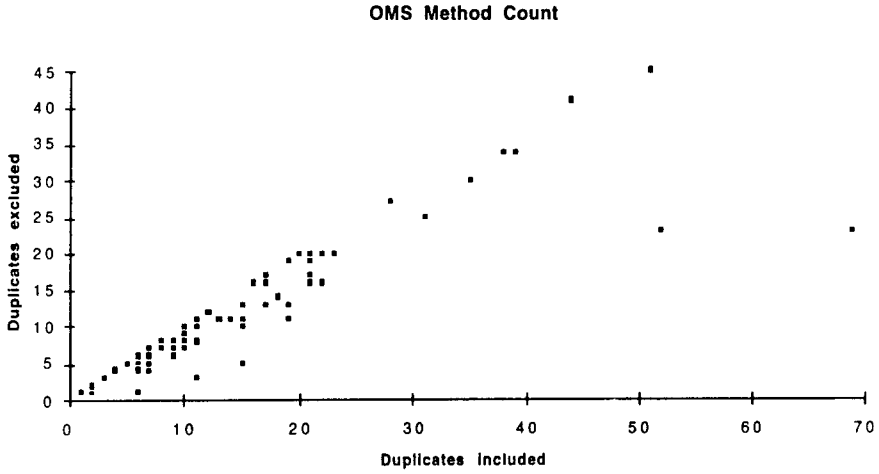

Figure 2. Effect of duplicate method identifiers

Two quantities commonly measured for each class in order to describe the inheritance structure of OO systems are the depth in the inheritance tree and the number of derived classes. These are the DIT and NOC metrics of CK. However, these are typically presented in histogram form, potentially aggregating out fine detail. Figures 3 and 4 below show the NOC and DIT distributions respectively for the OMS system. It is apparent that there are many top-level classes and that most classes have no children.

Figure 5 shows the distribution of NOC plotted separately for each DIT value. It is now clear that most of the classes for which NOC = 0 also have DIT = 0 or DIT =1. This is consistent with the nature of the OMS library which contains classes implementing several distinct service categories, many of which are implemented by single classes or by a class and its children. This is one of the first C++ products of a team of experienced C programmers and anecdotal evidence suggests that better use could perhaps been made of inheritance in designing the library. The 'bump' at DIT = 6 is visible in figure 4 but is more readily identified as a potential anomaly in figure 5. Another potential anomaly appears at NOC = 8, DIT = 4 in figure 5 but features such as this are easily washed out in histograms such as that of figure 4.

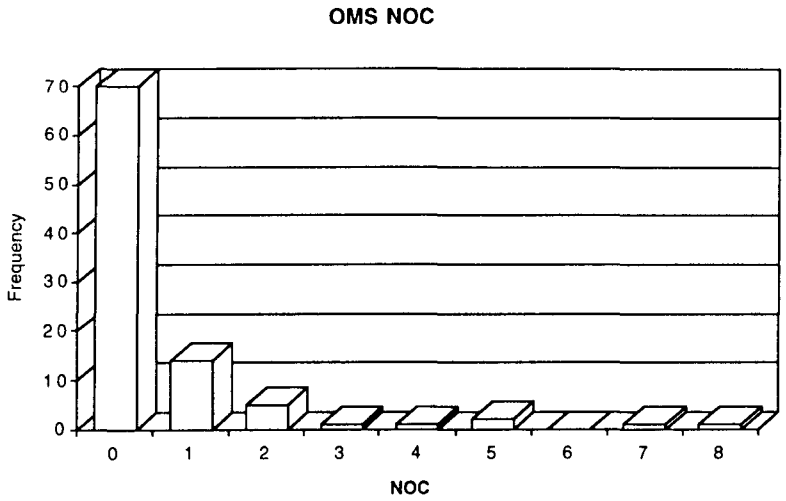

Figure 3. Distribution of NOC for the OMS library

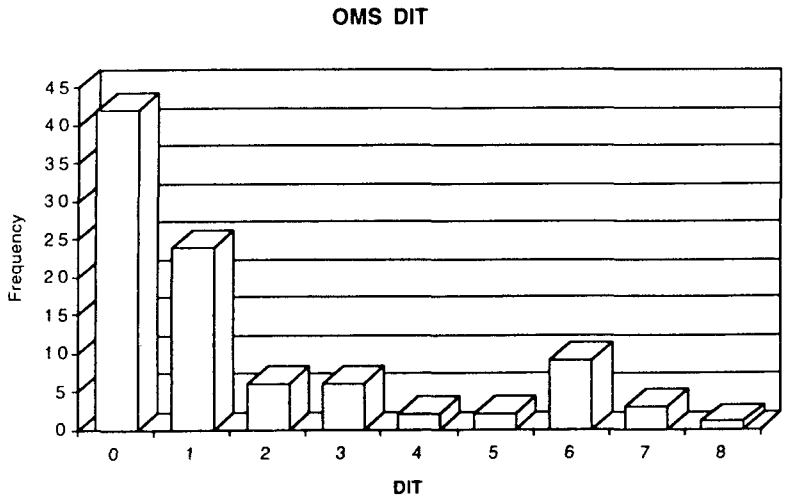

Figure 4. Distribution of DIT for the OMS library

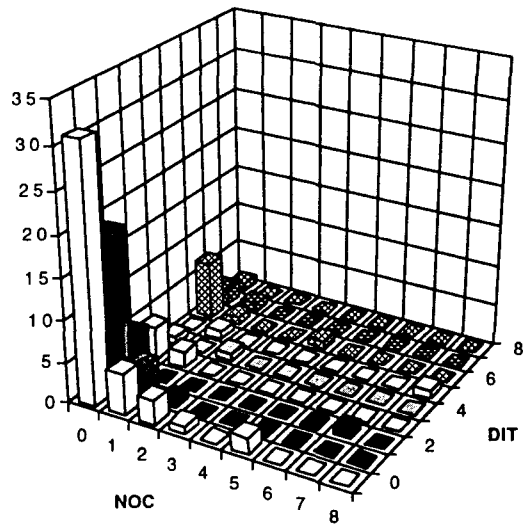

Figure 5. Variation of NOC with DIT

7. *Application to Existing Metrics*

Only a few OO metrics have been proposed and most studies to date are based on the set of 6 metrics of Chidamber and Kemerer (CK) [4, 5]. In this section we consider the potential effects of definitions and choices on the CK metrics. We do not intend any criticism of the CK metrics *per se*, but rather to highlight some of the issues facing those who wish to implement the CK or other OO metrics. For further discussion see [6].

In their papers [4, 5], CK state that "A method is an operation on an object that is defined as part of the declaration of a class". It is not immediately apparent whether or not this includes indirectly-defined inherited methods. However it is stated elsewhere that "...the combination of two object classes results in another class whose properties are the union of the properties of the component classes" suggesting that the 'duplication of inherited methods' interpretation has been made. However, the definitions of the inherited properties are located elsewhere, providing difficulties for maintainers that OO metrics should be able to measure.

Coupling is said to occur "...when methods declared in one class use methods or instance variables of the other class" subsequently it is noted that "this will include coupling due to inheritance" but this presumably refers to coupling between parent and child classes rather than indirect coupling between the class sending a message which is passed on by the recipient to one of its ancestors.

We now examine the 6 CK metrics and outline some of the ways in which they will be affected by such factors as choice of counting rules. In the remainder of this section C_i are classes and M_i are methods.

Weighted Methods Per Class (WMC)

This metric is defined as

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

where c_i is the complexity of M_i. The complexity may be estimated by using a conventional intra-module metric such as cyclomatic complexity. However, since the expected size of methods is low, it is reasonable to set $c_i = 1$ in which case WMC is simply the number of methods in the class.

Clearly, WMC is going to be influenced significantly by the choice of method counting rules as discussed in section 5.

Depth of Inheritance Tree (DIT)

DIT is defined as the depth of the inheritance tree. It will be sensitive to the inclusion of system classes in the ancestry of objects. In some languages, all user-defined classes are (perhaps implicitly) derived from system classes such as 'object' or 'vanilla'. When discussing depth in the inheritance structure it is necessary to state whether such classes are included. Wilde et al. [17] report that this effect explains the apparent differences for their Smalltalk and C++ data. We suggest using DIT = 0 to indicate top level user defined classes.

Since many languages permit multiple inheritance the inheritance graph is not necessarily a tree and is more likely to be a rooted DAG. The level of a class C_i is then the length of the longest path from the root to C_i. The level L is the highest level to be considered (e.g. only system classes appear above this level). A more applicable definition might then be DIT(C_i) = length of the longest path to C_i from any node at level L.

Number of Children (NOC)

NOC, the number of immediate subclasses in the inheritance structure, is delightfully straightforward to define.

Coupling Between Objects (CBO)

CBO is the number of other classes to which C_i is coupled. CK regard classes as coupled when "methods declared in one class use methods or instance variables defined by the other class".

The treatment of inherited methods will affect CBO. For example, it is not clear whether a call to an inherited method (i.e. message received by its host class) results in coupling to the host class, the class in which the method is defined or some combination of these possibilities. Our interpretation of CK is that only direct coupling is intended.

The development of more sensitive metrics is likely to require the separate consideration of different types of 'use'.

Response For a Class (RFC)

RFC is "the set of methods that can potentially be executed in response to a message received by an object of that class" or the number of members of the response set

$$\{M_i\} \bigcup_i \{R_i\}$$

where R_i is the set of methods called by M_i.

Clearly, RFC will be highly sensitive to the precise definition of number of methods and to other factors such as calls to functions which are not methods. CK do not include calls to such functions

Since long cascades of module invocations can result from a single initial invocation, CK propose that, for practical data capture reasons, only one level of nested calls should be considered. In general one may wish to use RFC_α ($\alpha = 1 \ldots \infty$). to achieve finer control of the metric.

Lack of Cohesion of Methods (LCOM)

LCOM is defined in terms of the sets I_i of instance variables used by method M_i. The I are considered pairwise, and LCOM is the number of pairs with null intersections - the number with non-null intersections.

Decisions to be made here include the inclusion of inherited state variables and the treatment of structured state variables—typically instances of other classes.

8. CONCLUSIONS

In this paper we have argued that software metrics for OO systems present significantly greater challenges than their conventional counterparts. Comparison of the data models of conventional and OO systems reveals that the latter is considerably richer, involving both a greater number of components and relationships between them. Management of this additional complexity will undoubtedly play a central role in the development of tools and models for OO metrics. We have presented a set of terms which cover the concepts fundamental to models of OO systems and metrics.

We have also argued that the 'micro' structure of models is also important. Our argument is illustrated by considering the particular example of a fundamental property of OO systems, the number of methods per class.

Until qualitative models of OO systems and their evolution are developed, it seems premature to proceed with the speculative development of specific metrics due to the absence of a satisfactory framework for their validation. We present some preliminary results from an empirical study of industrial C++ code. These demonstrate that ambiguities in the definitions of fundamental quantities can lead to apparent differences in empirical measurements. The effects of such differences are compounded when they are incorporated in indirect metrics. Our results also suggest that some insight into system features may be lost if they are swept up in aggregate metrics.

To summarise, we applaud the methodical approach of CK but have indicated how the points we raise might affect the empirical work using their proposed metrics suite.

REFERENCES

[1] Batini, C., S. Ceri, and S. Navathe, *Conceptual Database Design: an Entity-Relationship Approach*. Benjamin/Cummings: Redwood City, CA, 1992.

[2] Berard, E.V., *Object Coupling & Object Cohesion*, in *Essays on object-oriented software engineering*, Prentice-Hall: 1993.

[3] Chen, P.P.-S., 'The Entity Relationship Model - Towards a Unified View of Data', *ACM TODS*, 1(1), pp9-36, 1976.

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

[5] Chidamber, S.R. and C.F. Kemerer, 'A Metrics Suite for Object Oriented Design', *IEEE Trans. Softw. Eng.*, 20(6), pp476-493, 1994.

[6] Churcher, N.I. and M.J. Shepperd, Comment on "A Metrics Suite for Object Oriented Design". Technical Report No. OO-1/94, Bournemouth University, 1994.

[7] Codd, E.F., 'A Relational Model of Data for Large Shared Data Banks', *CACM*, 13, pp377-387, 1970.

[8] Fenton, N., S.L. Pfleeger, and R.L. Glass, 'Science and Substance: A Challenge to Software Engineers', *IEEE Software*, 11(4), pp86-95, 1994.

[9] Garner, S., *A Software Metrician's Workbench*. 1992, University of Canterbury:

[10] Garner, S. and N.I. Churcher. 'A Software Metrician's Workbench', in *Proc. 12th New Zealand Computer Conference*. Dunedin, N.Z.: NZCS, 1991.

[11] Halstead, M.H., *Elements of Software Science*. Elsevier North-Holland: 1977.

[12] Henry, S. and D. Kafura, 'Software structure metrics based on information flow', *IEEE Trans. Softw. Eng.*, 7(5), pp510-518, 1981.

[13] Lassez, J.-L., *et al.*, 'A Critical Examination of Software Science', *J. Systems & Software*, 2(2), pp105-112, 1981.

[14] Lejter, M., S. Meyers, and S.P. Reiss, 'Support for Maintaining Object-Oriented Programs', *IEEE Trans. Softw. Eng.*, 18(12), pp1045-1052, 1992.

[15] Shen, V.Y., S.D. Conte, and H.E. Dunsmore, 'Software Science Revisited: a critical analysis of the theory and its empirical support', *IEEE Trans. on Softw. Eng.*, 9(2), pp155-165, 1983.

[16] Wilde, N. and R. Huitt, 'Maintenance Support for Object-Oriented Programs', *IEEE Trans. Softw. Eng.*, 18(12), pp1038-1044, 1992.

[17] Wilde, N., P. Matthews, and R. Huitt, 'Maintaining Object-Oriented Software', *IEEE Software*, (1), pp75-80, 1993.

[18] Wilson, R.P., *RIPPLE: a metadata repository system.* 1992, University of Canterbury:

ACM SIGSOFT Software Engineering Notes vol 20 no 2 April 1995 Page 76

Models and Languages for Component Description and Reuse

Ben Whittle. ben@minster.york.ac.uk
Department of Computer Science,
University of York,
Heslington,
York, UK.

January 10, 1995

Abstract

This paper brings together the current research on reusable component models and component description languages for reuse. The paper contains a description and comparison of the 3C and REBOOT component models. The importance and further development of the 3C model is discussed. The component description language field is surveyed, and an introduction is given to the languages LIL, ACT TWO, Π, Meld, CDL, CIDER, LILEANNA, and RESOLVE. All of these languages are aimed at describing reusable components in the design stages of development. Criteria for examining component description languages are introduced and used as the basis of a comparison of the languages. The paper concludes with suggestions for the convergence of these developments, and suggestions for further work in this field.

1 Introduction

In order to develop a component for reuse and to subsequently reuse the component, the component, and the mechanisms used to describe its relationship to other components, must be well understood. A good description of a component, and its relationships helps us to understand, write, and reuse, that component. This paper looks at two approaches to describing reusable software components, Component Models, and Component Description Languages[1]. A Component Model is an abstract description for the components in a given domain. A Component Description Language, is a structured system of syntax and semantics, a language, with which to capture the essential attributes of components in a given domain. A Component Description Language for a given domain may be based on a Component Model for that domain.

[1]Acronyms will not be used for Component Model or Component Description Language to avoid confusion with the language name CDL used later in the paper.