An Overview of Object-Oriented Design Metrics

Daniel Rodriguez Rachel Harrison

RUCS/2001/TR/???/A
March 2001

Keywords: Empirical Software Engineering, metrics, object-oriented design
Research Group: ASE
Grant Awarding Body: Computer Science Department
Departmental Grant Number:

THE UNIVERSITY OF READING
DEPARTMENT OF COMPUTER SCIENCE

Abstract

It is stated that object-oriented technology approach to software construction needs a specific set of metrics. This paper examines object-oriented design metrics of common use as a means of assessing of quality characteristics of objects-oriented systems. The set of metrics described are the ones defined by Abreu, Chidamber and Kemerer, and a subset of Lorenz and Kidd as well as other traditional ones if extensive use. Each metric is described based on a template composed of name, definition, intent, interpretation guidelines, problems, and possible improvements are discussed.

Table of contents

1. Introduction

The use of metrics is in order to manage, predict and improve the quality of software product is increasing popularity. There is a large different kind of metrics that need to be used in projects (estimating, tracking…) but this paper focuses on Object-Oriented (OO) design metrics. As use of the OO paradigm has become widely used, the advent of specific set of metrics for these systems has also gained popularity in the last years. As the use of the OO paradigm does not get quality by itself (there are examples of applications that are not robust, maintainable or reusable, etc.), the goal using OO metrics is to assess the systems in order to produce high-quality results.

Some traditional metrics or modifications of them can be useful within OO paradigm, especially in the method level. However, in order to quantify the OO features (encapsulation, information hiding, inheritance, polymorphism, and message-passing), authors have proposed new sets of metrics. In this paper, metrics defined by Abreu [Abreu, 1996], Chidamber and Kemerer (C&K) [Chidamber and Kemerer, 1994], a subset of Lorenz and Kidd (L&K) [Lorenz and Kidd, 1994] and a few other relevant metrics are reviewed and explained. These sets of metrics are explained according to a classification defined by Archer [Archer and Stinson, 1995] to cover all the possible features and granularity of the OO systems. Each metric is described based on a template considering important features of each metric such as, how are it computed, some published thresholds, and assesses its appropriateness and usefulness. Thresholds and interpretation guidelines are given as a mean to spot outliners. Classes or methods that differ substantially from average values may indicate future problems, ill-conceived abstractions or bad implementations, which are good candidates for inspection or rework. Some empirical studies have been performed but there is still more empirical work needed. [Lorenz and Kidd, 1994] have suggested some thresholds based on some IBM projects in C++ and Smalltalk. Although some work about empirical validation has been performed, much more work is necessary.

Section 2 of this paper considers the validity of metrics. Those are properties that metrics should verify to be valid both theoretically and empirically. In section 3, a literature survey is presented in a categorization according to the granularity of their intended measure. Finally, section 4 presents some conclusions and future work.

2. Validity of Metrics

A set of metrics must make clear what aspects of quality they are trying to measure and who they are directed at because there are different points of view about what means quality (developers, managers, users). First of all, metrics should clarify what attributes of the software that are going to be measured, how we should measure those attributes [Basili et al, 1995][Kitcheham et al, 1995][Fenton, 1994] so they are meaningful and related to the product. Second, metrics as the mapping between entities of the real world and numbers, should be validated theoretical and empirically. In addition, the chosen set of metrics should be based on a properly defined quality model, incorporating ideas from many of the current standard quality (from fixed hierarchical models (Boehm, McCall's FCM) to more flexible approaches such as the Goal-Question-Metric –GQM- [Basili and Rombach, 1988] or Kitchenham's QMS subsystem [Kitchenham et al, 1986]). The GQM and the Kitchenham's model help to construct a quality a requirements model and according to quality factors

(*usability*, *integrity*, *efficiency*, *maintainability*, *reusability*, *interoperability*, *…)*. These flexible models would help to clarify what quality aspects are considered and why.

[Kitchenham et al, 1995] describe a list of features, which must hold for a metric to be valid. For *direct metrics*, which are the ones that involves no other attribute or entity (length, duration of testing process, number of defects…), those properties are:

1. For an attribute to be measurable, it must allow different entities to be distinguished from one another.
2. A valid metric must obey the representation condition.
3. Each unit contributing to a valid metric is equivalent.
4. Different entities can have the same attribute value.

For *indirect metrics*, i.e. when direct metrics are combined (ex., programmer productivity = LOC/persons month of effort, etc), these feature are:

1. The metric should be based on an explicitly defined model of the relationship between certain attributes.
2. The model must be dimensionally consistent.
3. The metric must not exhibit any unexpected discontinuities.
4. The metric must use units and scale types correctly.

The *representation condition*, as described by [Fenton and Pfleeger, 1997], asserts that a measurement mapping M must map entities into numbers and empirical relations into numerical relations in such a way that the empirical relations preserve and are preserved by the numerical relations. For example, if A is taller than B if and only if M (A)> M(B).

Empirical methods provide corroborating evidence of validity. Using statistical and experimental techniques assesses the usefulness and relevance of metrics [Basili et al, 1995] [Schneidewind, 1992].

3. Object-Oriented Design Metrics

A representative set of metrics are explained according to a classification defined by Archer [Archer and Stinson, 1995] to broad all the possible features and granularity of OO.

3.1. System metrics level

There are system metrics that can be derived from class metrics with statistics, as relative measures, identifying systems that deviate from the norm. Unusual trends or characteristics of the system under construction can be spotted and corrected.

The MOOD (Metrics for Object oriented Design) set of metrics of Abreu and [Abreu and Melo, 1996] operate at System level. They refers to a basic structural mechanism of the OO paradigm as *encapsulation* (MHF and AHF), *inheritance* (MIF and AIF), *polymorphishm* (PF) and *message-passing* (COF). The set of metrics are:

Method Hiding Factor (MHF). [Abreu and Melo, 1996]

Definition MHF is defined as the ratio of the sum of the invisibilities of all methods defined in all classes to the total number of methods defined in the system

under consideration.
The invisibility of a method is the percentage of the total classes from which this method is not visible.
In other words, MHF is the ratio of hidden methods –protected or private methods- to total methods.

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

where $M_d(C_i)$ is the number of methods declared in a class, and

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

and:

$$is\_visible\left(M_{mi}, C_j\right) = \begin{cases} 1\, iff\ \ j \neq i \wedge C_j \text{ may call } M_{mi} \\ 0 \qquad \text{otherwise} \end{cases}$$

That is, for all classes C_1,… ,C_n, a method counts as 0 if it can be used by, another class, and 1 if it cannot.
TC = Total number of classes in the system under consideration.
In languages such as C++, where there is the concept of *protected* method, the method is counted as a fraction between 0 and 1:

$$V\left(M_{mi}\right) = \frac{DC(C_i)}{TC-1}$$

Intent — MHF is proposed to measure the encapsulation (the relative amount of information hidden).

Guides and comments — [Abreu and Melo, 1996] have found that as MHF increases, the defect density and the effort spent to fix is expected to decrease.
Inherited methods are not considered.

Attribute Hiding Factor (AHF). [Abreu and Melo, 1996]

Definition — AHF is defined as the ratio of the sum of the invisibilities of all attributes defined in all classes to the total number of attributes defined in the system under consideration.
In other words, AHF is the ratio of hidden attributes –protected or private- to total attributes.

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

where:

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

and:

$$is\_visible\left(A_{mi}, C_j\right) = \begin{cases} 1\, iff\ \ j \neq i \wedge C_j \text{ may call } M_{mi} \\ 0 \qquad \text{otherwise} \end{cases}$$

TC = Total number of classes in the system under consideration.

Intent — AHF is also proposed to measure the encapsulation (amount of information hidden).

Guides and comments: Ideally this metric should be always 100%. Systems as a rule should try to hide nearly all instance data. Design guidelines suggest that public attributes should not be used because are generally considered to violate the rules of OO encapsulation since they expose the object's implementation. In benefit of performance, some times is avoided the use of the use of accessors and modifiers methods (*getters* and *setters* methods).

Comments

Method Inheritance Factor (MIF). [Abreu and Melo, 1996]

Definition: MIF is defined as the ratio of the sum of the inherited methods in all classes of the system under consideration to the total number of available methods (locally defined plus inherited) for all classes.
MIF measure directly the number of inherited methods as a proportion of the total number of methods.

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

where:

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

and:
$M_d(C_i)$ = the number of methods declare in a class
$M_a(C_i)$ = the number of methods that can be invoked in association with *Ci*
$M_i(C_i)$ = the number of methods inherited (and not overriden in *Ci*).
TC = Total number of classes in the system under consideration.

Intent: Abreu proposes that MIF is a measure of inheritance, and consequently as a means of measure the level of reuse.
It is also a aid to the assessment of testing needed.

Guides and comments: The use of inheritance is seen as a trade-off (see inheritance metrics below). Empirical studies related to inheritance are [Daly et al, 1996] and [Harrison, 199x].

Attribute Inheritance Factor (AIF). [Abreu and Melo, 1996]

Definition: AIF is defined as the ratio of the sum of inherited attributes in all classes of the system under consideration to the total number of available attributes (locally defined plus inherited) for all classes.
It is defined in an analogous fashion to MIF. AIF measure directly the number of inherited attributes as a proportion of the total number of attributes.

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

where:

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

And:
$A_d(C_i)$ = the number of attributes declared in a class
$A_a(C_i)$ = the number of attributes that can be invoked in association with C_i
$A_i(C_i)$ = the number of attributes inherited (and not overriden in C_i).
TC = Total number of classes in the system under consideration.

Intent	AIF as a way to express the level of reuse in a system.
Guides and comments	Too much reuse trough inheritance makes worst the understandability and testability.

Polymorphism Factor (PF). [Abreu and Melo, 1996]

Definition	PF is defined as the ratio of the actual number of possible different polymorphic situation for class C_i to the maximum number of possible distinct polymorphic situations for class C_i. PF is the number of methods that redefine inherited methods, divided by the maximum number of possible distinct polymorphic situations. $$PF = \frac{\sum_{i=1}^{TC} M_0(C_i)}{\sum_{i=1}^{TC}[M_n(C_i) \times DC(C_i)]}$$ where: $M_d(C_i) = M_n(C_i) + M_o(C_i)$ and: $M_n(C_i)$ = the number of new methods $M_o(C_i)$ = the number of overriding methods $DC(C_i)$ = the descendants count TC = Total number of classes in the system under consideration
Intent	A measure of polymorphism potential.
Guides and comments	Polymorphism arises from inheritance. Abreu claims that in some cases, overriding methods reduce complexity, so increasing undestandability and maintainability. It is an indirect measure of the dynamic binding in a system. This metric does not satisfy the all properties of Kitchenham because in a system without inheritance, the value of PF will be undefined. Therefore, it exhibits a discontinuity [Harrison et al, 1998].

Coupling Factor (CF). [Abreu and Melo, 1996]

Definition	CF is defined as the ratio of the maximum possible number of couplings in the system to the actual number of couplings not imputable to inheritance. That is, this metrics counts the number of inter-class communications. $$CF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} is\_client(C_i, C_j)\right]}{TC^2 - TC}$$ where: $$is\_client = \begin{cases} 1 iff\ C_c \Rightarrow C_s \wedge C_c \neq C_s \\ 0 \quad otherwise \end{cases}$$ The client-supplier relationship ($C_c \Rightarrow C_s$) represents means that the client class, C_c, contains at least one non-inheritance reference to a feature of the supplier class, C_s. TC = Total number of classes in the system under consideration.
Intent	Coupling is viewed as a measure of increasing complexity, reducing both encapsulation and potential reuse and limiting understandability and maintainability.

Guides and comments	CF can be an indirect measure of the attributes to which it was said to related: complexity, lack of encapsulation, lack of reuse potential, lack of understandability, lack of maintainability. [Abreu and Melo, 1996] have found a positive correlation. As a coupling among classes increases, the defect density and lack of maintainability increase.

3.2. Coupling Level

Coupling is the use of methods or attributes defined in a class that are used by another class. Classes interact with other classes to form a subsystem/system and this interaction can indicate the complexity of the design. Representative metrics of this set are:

Coupling Between Objects (CBO). [Chidamber and Kemerer, 1994]

Definition	CBO for a class is a count of the number of other classes to which it is coupled. Coupling between two classes is said to occur when one class uses methods or variables of another class. COB is measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends.
Intent	C&K suggest that it's an indication of the effort needed for maintenance and testing.
Guides and comments	C&K view the more coupling in class, the more difficult to reuse. Classes should be as independent from other classes as possible in order to promoting reuse. Classes always have interdependencies, but if they are large, to understand the design can be a nightmare and therefore, the reuse can be more expensive than rewrite. Classes with excessive coupling makes more difficult the maintenance and the more rigorous the testing needs to be done. Strong coupling complicates a system since a class is harder to understand, change or correct by itself if it is interrelated with other classes. Systems with low-coupling reduces complexity, improves modularity and promotes encapsulation.

3.3. Inheritance Level

Inheritance relationship is viewed as a trade-off. It difficulties the understability and maintainability of systems making hard to change the interface of a superclass. Cartwright and Shepperd [Cartwright and Shepperd, 1996] report almost no use of inheritance, as measured by the *Depth of Inheritance Tree (DIT)* and the *Number Of Children (NOC)* metrics, at their UK telecommunications site. In [Chidamber et al, 1998] have also reported a conscience lack in the use of inheritance in their analysis of an European bank because Systems with the high inheritance levels are usually built on frameworks where most classes are inherited from an existing hierarchy.

The use of *composition* can be a worth alternative to inheritance. Composition provides an approach that yields easier to change code and deserves to take it into account when an aim is

reuse code.

The use of *multiple inheritance* generally is not recommended due to name collisions and lack of understanbility in languages that allow method to be inherited from multiple superclasses such as C++. Languages such as Java or Smalltalk only support single inheritance.

Representative metrics of this set are:

Depth of Inheritance Tree (DIT). [Chidamber and Kemerer, 1994]

Definition	DIT measures the maximum level of the inheritance hierarchy of a class; the root of the inheritance tree inherits from no class and is at level zero of the inheritance tree. Direct count of the levels of the levels in an inheritance hierarchy.
Intent	According to C&K, DIT is intended to measure the class complexity, design complexity and the potential reuse because the more deeper is a class, the greater number of methods is likely to inherit.
Guides and comments	Large inheritance depths indicate complex objects that may be difficult to test and reuse. Small inheritance depths can indicate functional code that does not take advantage of the inheritance mechanism. L&K suggest a threshold of 6 levels for individual classes to indicate excessive inheritance in Smalltalk and C++ projects. [Cartwright and Shepperd, 1996] found a positive correlation between the DIT metric and number of user-reported problems, casting doubt on the effective use of inheritance. The use of frameworks affects the DIT frameworks include several levels of inheritance and often serve as the base for application objects. In languages such as Java or Smalltalk objects always inherit from the Object class, which adds one to DIT. A problem with this metric is that DIT is not clearly defined, as there are different features of inheritance. For example, As in DIT, this metric is not clearly defined, as they are measuring different features of inheritance. DIT can not be a valid measure of reuse due it can easily happen that a high value of DIT reuse the same or less number of methods than a shallow, wide hierarchy. A support metric for DIT is the number of method inherited (NMI).

Number of Children (NOC). [Chidamber and Kemerer, 1994]

Definition	NOC is the number of immediate subclasses subordinate to a class in the hierarchy. NOC counts the number of subclasses belonging to a class.
Intent	According to C&K, NOC indicate the level of reuse, the likelihood of improper abstraction and as possible indication of the level of testing required.
Guides and comments	Although a greater the number of children represents a greater reuse, it has the following drawbacks: A greater the likelihood of improper abstraction (misuse of sub-classing). More difficult to modify because it affects all its children have a heavy

dependence on the base class.
More testing is required.
It is an indicator of the potential influence a class can have on the design of the system. If the design has a high dependence on reuse through inheritance, and may be better to split off functionality into several classes.

Specialisation Index per Class (SIX). [Lorenz and Kidd, 1994]

Definition

The specialisation index measures the extent to which subclasses override (replace) behaviour of their superclasses.

$$Specialization\ index = \frac{\text{Number of overriden methods} \cdot \text{Class hierarchy nesting level}}{\text{Total number of methods}}$$

This formulation weights more heavily overrides that occur farther down the inheritance tree since these classes should be more specialised and less likely to replace base behaviour.

Intent

SIX provides a measure of the quality of sub-classing.

Guides and comments

SIX may indicate that there are overridden methods in excess, so the abstraction may not have been appropriate, since if much behaviour needs to be replaced. A subclass should extend the superclass behaviour with new methods rather than replacing or deleting behaviour through overrides.
Subclassing by specialisation means to create new classes that are a superset. It is characterised by low number of method overrides, decreasing number of added method and few or no deleted methods.
L&K suggest a value of 15% to help identifying classes for which remedial action might be appropriate, that is, the superclass may not have much in common with the subclass.
The more father down in a class hierarchy, the more specialised the subclass should be.
Using frameworks, some methods are meant to be overridden and should not be included in the count.

3.4. Class Level

These metrics identify characteristics within the class, highlighting different aspects of the class abstractions and help identify where remedial action may be taken. Representative metrics of this set are:

Response For a Class (RFC). [Chidamber and Kemerer, 1994]

Definition

RFC is the count of the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class. This includes all methods accessible within the class hierarchy.
RFC counts the occurrences of calls to other classes from a particular class.
$RFC = |RS|$ where RS is the response set for the class.

The response set for the class can be expressed as:
$RS = \{M\} \bigcup_i \{R_i\}$

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

the class.

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

Intent — According to C&K, RFC is a measure of the complexity of a class through the number of methods and the amount of communication with other classes.

Guides and comments — RFC is an indicator of the effort of testing and debugging. The larger RFC, the greater the complexity because a large number of methods can be invoked in response to a message. Therefore, more allocation of testing time is required since it requires a greater level of understanding.
There is ambiguity in definition of this metric forcing the user to interpret [Harrison et al, 1996].

Weighted Methods per Class (WMC). [Chidamber and Kemerer, 1994]

Definition — WMC measures the complexity of an individual class. If all methods are considered equally complex, then WMC is simply the number of methods defined in each class.

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

where a Class C_1 has M_1, …M_n, methods with c_1, …, c_n complexity respectively.

Intent — C&K suggests that WMC is intended to measure the complexity of a class. Therefore, it is an indicator of the effort needed to develop and maintain a class.

Guides and comments — Classes with large number of methods:
Require more time and effort to develop and maintain the class because it will impact on subclasses inheriting all the methods.
Are likely to be more application specific, limiting the possibility of reuse.
L&K suggest a threshold of 20 and 40 instance methods, depending on if they are UI (User Interface) classes or not.
In [Harrison et al, 1997] highlight some problems with this metric:
WMC is supposed to measure complexity but not definition of complexity is given.
WMC as an indicator of the effort to develop a class, since a class containing a single large method may take as long to develop as a class containing a large number of small methods.
Therefore, it should be considered simply as a measure of class size.

Lack of Cohesion in Methods (LCOM). *[Chidamber and Kemerer, 1994]*

Definition — LCOM measures the extent to which methods reference the classes instance data.
Consider a class C_1 con n methods M_1, M_2, …, M_n. Let $\{I_j\}$ = set of instance variables used by method M_i.

There are *n* such sets $\{I_1\},...\{I_n\}$. Let $P = \{(I_i, I_j) \mid I_i \cap I_j = \varnothing\}$, and $Q = \{ (I_i, I_j) \mid I_i \cap I_j \_ \varnothing\}$. If all n sets $\{I_1\},..., \{I_n\}$ are $\varnothing$ then let $P = \varnothing$.

$$LCOM = \begin{cases} |P| - |Q| & \text{if } |P| > |Q| \\ 0 & \text{otherwise} \end{cases}$$

Example:
Consider a class C con 3 methods (M_1, M_2, M_3). Let $\{I_1\}$ = {a, b, c, d, e}, $\{I_2\}$ = {a, b, e}, and $\{I_3\}$ = {x, y, z}. $\{I_1\} \cap \{I_2\}$ is nonempty but $\{I_1\} \cap \{I_3\}$ and $\{I_2\} \cap \{I_3\}$ are null set. LCOM is the number of number of null intersections – number of nonempty intersection), which in this case is 1.

Intent

LCOM is a quality measure for the cohesiveness of a class by measuring the number of common attributes used by different methods.

Guides and comments

LCOM measure indicates the fitness of each class abstraction.
A high value of LCOM implies lack of cohesion, namely, low similarity and the class may be a composition of unrelated objects. High cohesiveness of methods within a class is desirable, since classes cannot be splited and promotes the *encapsulation*.
Low cohesiveness increases complexity, thereby increasing the likelihood of errors during the development process.
[Henderson-Sellers, 94] arises two problems with this metric. First, two classes can have a LCOM=0 while one has more common variables than other. And second, there are not guidelines about interpreting the values.

Therefore, Henderson-Sellers has suggested a new LCOM measure:
Consider a set of methods $\{M_I\}$ $(I=1,\ldots,m)$ assessing a set of attributes $\{A_j\}$ $(j=1,\ldots,a)$ and the number of methods that access each attribute to be $\mu(A_j)$, then define *LCOM** as:

$$LCOM^* = \frac{\left(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)\right) - m}{1-m}$$

If all methods access all attributes, then $\Sigma\ \mu(A_j) = ma$, so that LCOM* = 0, which indicates perfect cohesion.
If each method access only 1 attribute then $\Sigma\ \mu(A_j) = a$ and LCOM* = 1, which indicates a lack of cohesion.
A value near zero for this metric indicates very high cohesion, where most methods refer to most instance variables.

3.5. Method Level

Attributes and methods occur at the finest level of detail. These characteristics are known, and techniques to analyse them are common but methods have the additional complexity of calling objects other than the object that contains them.

In OO systems, traditional metrics such as *Lines Of Code (LOC)* and *cyclomatic complexity* [McCabe, 1976] are usually applied to the methods. Methods are usually developed much like procedures are in structured programming. However, they are less important in OO than in functional programs due to: short methods and less conditional statements (ex., if , case

statements) and the complexity tend to be in the interaction of methods rather than within one method.
The cyclomatic complexity metric has not been used extensively in with OO systems. This is because the complexity found in a highly OO system will tend to be in the interaction of methods rather than within one method.

Two traditional metrics are only briefly commented because are extensively used at method level: cyclomatic complexity and size.

Lines of Code per method (LOC). [Lorenz and Kidd, 1994]

Definition	LOC is the number of physical lines of active code (executable lines) that are in a of code within the method. Size can be measured in a variety of ways. These include counting all physical lines of code, the number of statements, etc.
Intent	Size of a method is used to evaluate the ease of understandability, reusability, and maintainability of the code by developers and maintainers.
Guides and comments	Thresholds for evaluating the size depend on the programming language and the complexity of the method. In OO systems, LOC should be low. L&K suggest a threshold of 24 LOC for C++ methods and 8 for Smalltalk, being the outliner candidates for splitting into smaller methods. LOC for the same functionality varies with the programming language used and the coding style/standard. It is not a recommended metric to use in OO systems but it is easy to measure and collect.

Number of Messages Send (NOM). [Lorenz and Kidd, 1994]

Definition	NOM measures the number of messages sent in a method, segregated by type of message. The types include: Unary, Messages with no arguments Binary, messages with one argument, separated by a special selector name. (concadenation and math functions). Keyword, messages with one or more arguments.
Intent	NOM quantifies the size of the method in a relative unbiased way.
Guides and comments	L&K suggest a threshold of 9. Languages as C++ can call to non-OO parts of the systems, that should not be counted in the number of message sends.

4. Conclusion

A set of OO design metrics has been described with some interpretation guides as a way to assess systems in order to produce of a robust, high-quality result. Each metric was described considering important features such as, how to use it, interpretation guidelines, published thresholds whenever is possible, and assesses its appropriateness and usefulness.

Some problems and suggestions are also commented such some ambiguities in some of the definitions, usefulness for its intention and even its validity. In the way forward, lessons learned will help to overcome those problems. More work about empirical validation is necessary using proven statistical and experimental techniques in order to improve their interpretation. More clear interpretation guidelines for these metrics based on common sense and experience are necessary.

5. References

[Abreu and Melo, 1996] F. Brito e Abreu, Walcelio Melo. Evaluating the impact of Object-Oriented Design on Software Quality. Proceedings of 3rd International Software Metrics Symp., Berlin,1996

[Archer and Stinson, 1995] Clark Archer, Michael Stinson. Object-Oriented Software Measures. Technical Report CMU/SEI-95-TR-002,April 1995

[Basili and Rombach, 1988] V. R. Basili and H. Dieter Rombach, The TAME Project: Towards Improvement-Oriented Software Environments, IEEE Transactions on Software Eng., Vol. 14, No. 6, June 1988.

[Basili et al, 1995] V. R. Basili, L. Briand, and W. Melo, “A Validation of OO Design Metrics as Quality Indicators,” Technical Report CS-TR-3443, 1995.

[Cartwright and Shepperd, 1996] Cartwright, M. and Shepperd, M., "An Empirical Investigation of Object Oriented Software in Industry", Dept. of Computing, Talbot Campus, Bournemouth University Technical Report TR 96/01, 1996.

[Chidamber and Kemerer, 1994] S. R. Chidamber and C. F. Kemerer, “A metric suite for object oriented design,” IEEE Transactions on Software Engineering, pp. 476–493, 1994.

[Chidamber et al, 1998] Chidamber, S. R., D. P. Darcy, C.F. Kemerer. Managerial use of object oriented software metrics: an exploratory analysis. IEEE Transactions on Software Engineering, 24(8), pp629-639, Aug 1998

[Daly et al, 1992] J. Daly, A Brooks, J. Miller, M. Roper, and Murray Wood. Evaluating inheritance Depth on the Maintainability of Object-Oriented Software. Empirical SE 1(2) Feb 96

[Fenton and Pfleeger, 1997] Norman E. Fenton, Shari Lawrence Pfleeger. Software Metrics. A rigorous & Practical Approach. 2nd Edition. ITP, International Thomson Computer Press, 1997

[Fenton, 1994] N. E. Fenton, “Software measurement: a necessary scientific basis,” IEEE Transactions on Software Engineering, vol. 20, no. 3, pp. 199–206, 1994.

[Harrison et al, 1997] Harrison, SJ Counsell, R Nithi, An Overview of Object-Oriented Design Metrics, Proc. of the conference on Software Technology and Engineering Practice (STEP), IEEE Press, pp. 230-237, Jul 1997, ISBN 08186 78402.

[Harrison et al, 1998] R. Harrison, S. Counsell and R Nithi. An evaluation of the MOOD Set of Object-Oriented Software Metrics. IEEE Transaction on Software Engineering, Vol. 24, No. 6,June 1998

[Henderson-Sellers, 1996] Henderson-Sellers, Brian. Object-Oriented Metrics Measures of Complexity. Upper Saddle River, NJ: Prentice Hall, 1996.

[Kitchenham et al, 1986] B. A. Kitchenham, J. D. Walker, and I. Domville, “Test specification and quality management - design of a QMS sub-systemfor quality requirements specification”, Project Deliverable A27, Alvey Project SE/031, Nov 1986.

[Kitchenham et al, 1995] B. Kitchenham, S. L. Pleeger, and N. Fenton. Towards a Framework for Software Measurement Validation. IEEE Transactions on Software Engineering, Vol. 21, No. 12, pp 929-944,December 1995

[Lorenz and Kidd, 1994] Mark Lorenz and Jeff Kidd. Object Oriented Metrics. Englewood, NJ: Prentice Hall, 1994.

[McCabe, 1976] McCabe, T. J. "A Complexity Measure," IEEE Transactions on Software Engineering, 2(4), 308-320, 1976.

[Schneidewind, 1992] N. F. Schneidewind, “Methodology for validating software metrics,” IEEE Transactions on Software Engineering, vol. 18(5), pp. 410–422, 1992.