The 7th International Conference for Informatics and Information Technology (CIIT 2010)

INTERPRETATION OF OBJECT-ORINTED QUALITY MODELS

Daniela Boshnakoska
Software Developer at Innovaworks
Skopje, Macedonia
daniela.bosnakoska@gmail.com

KEYWORDS

object-oriented metrics, object-oriented quality models, MOOSE model, Li and Henry metrics, MOOD model, QMOOD model.

ABSTRACT

Although the object-oriented metrics are newer in the measurement theory, they have proven as useful predictors of good system design. Object-oriented metrics have been grouped to minimal sets to assess quality of the Object-oriented systems, known as object-oriented quality models. There are couples of proposed sets, which have been validated, empirically tested and applied on the real system.

I. INTRODUCTION

Object oriented quality models estimate object-oriented designs. They work by establishing relationships between desirable design attributes or predicting estimations about the quality of the components. The aim of quality models is to assess quality attributes such as maintainability. This can be gained by establishing relationships between quality characteristics and the metrics computed from object-oriented diagrams. If appropriately used, these models can provide significant reduction in costs of the overall implementation process and improvements in the quality of the final product. This work is focused on introduction and basic interpretation of the main quality models.

II. SOFTWARE MEASUREMENT

All industry branches use measurements in order to validate their improvements. Measurements are used extensively in most areas and productions to estimate costs, calibrate equipment, assess quality or monitor inventories. In software industry, as technology is changing per daily basis, decisions are even hard to make. Therefore guidelines are needed, so the managers and practitioners can make these decisions, plan and schedule activities or allocate recourses. Measuring software is one of the most valuable guidelines.

Software Measurement is a quantified attribute of a characteristic of the software product, or the software process. The main purpose of measures in the software industry is to improve the overall software process, assisting the planning activities, control of design, assessing the quality of the: design model, testing scenarios and coding.

Software metrics provide information so people can make informed decisions and intelligent choices. Software metric is a **measurement scale and method** to determine the value of an indicator of a certain software product. Software metric has been defined as a measure of some property of a piece of software or its specification. The range of the software metrics is wide, and they can be further categorized in meaningful categories. As part of their wide categorization, we can separate design and code metrics.

Design and Code metrics can be more classified according to the paradigm they adhere to. Such classification is to: procedural or object-oriented metrics. Object-oriented metrics reflect the impact of using the object oriented mechanisms such as inheritance, association, aggregation, polymorphism and message passing. They could be further categorized according the object oriented property they measure.

III. OBJECT-ORIENTED METRICS

The benefits of object oriented software development are now widely recognized. Object-oriented technology in software industry has created new challenges for monitoring, controlling and improving the ways to develop and maintain software. Object-oriented development requires different approach from traditional functional decomposition and data flow. Object-oriented programming claims faster development pace and higher quality of the software. Designing object-oriented system is focused on objects and designers use this approach because it is a faster development process and has high reusable features; increases design quality.

In object-oriented environment the metrics that will be used does not only depend on the code structure but also on several high-level features that easily can discriminate among different levels of complexity. Object-oriented metrics have been introduced as a way to measure the quality of the object-oriented design. These metrics focus on measurements that are applied to class and design characteristic. They help designers early in the development phase to make changes to reduce object complexity and improve the continuing capability of software. Such metrics can be used to identify the primary critical constructs of the design and to select metrics that evaluate those areas. Object-oriented metrics provide valuable information to the developers and managers.

There are at least three ways in which object-oriented metrics can be used: **quality estimation, risk management and estimating system testability**. Quality estimation means to build estimation model from the gathered historical data. In practice, quality estimation is gained either by estimating reliability (number of defects) or maintainability (change effort). Risk management is concerned in finding the potentially problematic classes as early as possible. Process and product metrics can help both managing activities (scheduling, costing, staffing and controlling) and engineering activities (analyzing, designing, coding, documenting and testing).

As previously stated, **the structure of the software design** is where metrics play an important role. Object-oriented metrics can evaluate the impact of object-oriented design on software quality characteristics such as defect density and rework.

Figure 1 Object-oriented characteristics and quality attributes

Data obtained from experiments shows that OO mechanisms such as inheritance, polymorphism, information hiding and coupling, can influence quality characteristics like reliability and maintainability. Inspecting the impact of OO design in quality attributes, infer that, in fact, the design alternatives may have a strong influence of reusing quality. The impact of other quality attributes such as efficiency, portability, usability and functionality must also be assessed. Many experimental validations have been introduced during the years performed on information systems to conclude that OO metrics can be used to identify fault prone classes.

One possible means to validate metrics is to conduct statistical analyses of the metrics and measures of system maintainability. Software maintenance is one of the most difficult and expensive tasks in the software development process. Metrics, especially those who measure the inner-connectivity of system components have been shown to have an impact on software system maintainability.

IV. Object-Oriented Quality Models

Object-oriented quality models estimate OO designs. They consist of four parts: Objectives, Metrics, Relationships and Thresholds. A significant number of OO metrics have been proposed: Briand, 1999, 2000; BritoeAbreu, 1994; Bucci, 1998; Chidamber, 1991, 1998; Fioravanti, 1998a, 1998c, 2001a, 2001b; Halstead, 1977; Henderson-Sellers, 1994; Kim, 1997; Li, 1993; Thomas, 1989.

A. MOOSE (C.K metric suite)

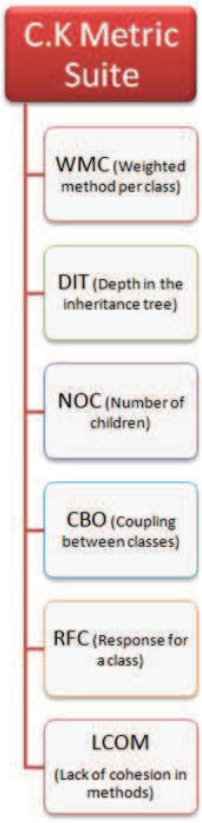

Figure 2 C.K Metric suite

Chidamber and Kemerer's metric suite for OO design is the deepest research in the OO metric investigation. This metric suite tells whether developers are following object oriented principles in their design. CK suite has been defined with the aim to assess the software development process as seen by expert object-oriented developers. CK metrics have developed significant interest and are currently the most well known suite of measurements for OO software [10].
C.K. metric suite consists of six metrics that assess different characteristics of the OOD:

WMC (Weighted Method per Class) assesses complexity of a class through aggregating complexity measure of its methods. When WMC is measured after the design phase and before implementation phase it has to be based on the methods' attribute complexity. If methods complexities are taken as equal, WMC can be taught as number of methods in a class and then considered as the measure of size and not complexity. It is difficult to implement WMC as complexity metric since not all of the methods are accessible within the class hierarchy due to inheritance. When we are indicating the number of methods with WMC we are giving prediction of the time and effort required in order to develop and maintain the class. Larger values for WMC mean we have greater number of methods that has greater impact of the children since they inherit the methods from the parent class. Classes with large number of methods are limited for reuse because they usually are application specific. WMC can also be used to estimate the usability and reusability of the class.
As the experiment s and statics show, low WMC indicates greater polymorphism in class, and high WMC indicates more complexity in the class.

DIT (Depth of the Inheritance Tree) assesses how deep a class in a class hierarchy is. The metric assess potential reuse of a class and its probable ease of maintenance. Classes that have smaller DIT indicate that are more general/abstract classes and have much potential for reuse. Classes that are deep into the hierarchy are difficult to maintain. They inherit greater number of methods which makes them more complex, hard to test and to predict their behavior. In deep inheritance trees more methods within a class are involved making the design more complex. The deeper a particular class is in the hierarchy, has greater potential reuse of inherited metrics. If the sub-classes asses the inherited properties from the super-class without using the methods defined in the super class, the encapsulation of the super class is violated. When calculating this value in languages that allow multiple inheritances, the longest path is usually taken. Large DIT is also related to understandability and testability.

NOC (Number of children) measures the number of classes associated with given class using the inheritance relationship. It could be used to assess the potential influence a class has on the overall design. Classes with many children are considered as bad design habit. NOC can be considered as measure on the impact of a class in the overall system design. Greater number of children indicates improper abstraction of the parent and can be considered as misuse of inheritance and sub- classing. On the other hand, the greater number of children, the greater reuse since inheritance is a form of reuse. The number of children gives an idea of the potential influence a class has on the overall design. Classes with large number of children require more testing. We expect that numerous children introduce more complexity into the class design.

CBO (Coupling between classes) points to the number of other classes to which a given class is coupled. CBO is measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends. One class is coupled to another if it uses its attributes or methods. CBO is beneficial in judging how complex the testing of various parts of the design is likely to be. Modular and encapsulated design should have low CBO, indicating that the class is more independent and easier to test or reuse. The more independent a class is the easier is to use it in another application. The larger number of couplings increases the sensitivity for change and maintenance. Strong couplings complicate the system since a class is harder to understand, change or correct. Systems designed with weakest possible coupling have reduced complexity and are easier to build and maintain. Measuring couplings can give basic insight how difficult testing is likely to be.

RFC (Response for a class) is defined as a count of the set of methods that can be potentially executed in response to a message received by an instance of the class. When calculating RFC all methods accessible within the class hierarchy are taken in consideration. Measures prove that RFC metric can be considered as complexity indicator, but can also represent the amount of communication with other classes. The larger the number of methods that can be invoked from a class the greater is its complexity. If a large number of methods can be invoked as response, testing and debugging the class becomes more complicated and it requires a greater level of understanding on the part of the tester, and it enlarges the testing time.As the results show and from the experiments provided, classes with large value for RFC indicates that the class is more complex and harder to maintain.

LCOM (Lack of cohesion in methods) is the difference between the number of methods whose similarity and the number of methods whose similarity is not zero. Similarity is the number of attributes used in common. Low value for LCOM indicates high cohesiveness, and vice versa. High LCOM means that the class should be split. Highly cohesive modules should stay alone because high cohesion indicates good class division. Lack of cohesion increases complexity, and complex development is more error prone. In order to improve the design, good practice is to subdivide low cohesion classes to increase the cohesiveness.

Studies on this metric suite have shown that they give insight beyond the traditional size metrics and that high value of the CK metrics correlate with: Lower productivity; High effort to reuse classes; High effort to design classes; Difficulties in implementing classes; Number of maintenance changes; Number of faulty classes; Faults; User reported problems.

B. Validation studies and examples

Chidamber and Kemerer's work is the most used in validation experiments and tests. **One of these experiments was**

performed as indication of fault-prone classes using statistical distributions and analysis using logistic regression. Logistic regression is a classification technique used in many experimental sciences based on maximum likelihood estimation. To evaluate whether the CK metrics are useful for predicting the probability of faulty classes, Basili and his colleagues designed and conducted empirical study at the x. The study involved students randomly grouped in 8 groups. Each team was responsible for developing medium sized management information system using the Waterfall management model. Testing phase was accomplished by an independent group of experienced software professionals. The analysis of six metrics have resulted that [7]:

WMC was shown to be somewhat important; for new and UI classes the results are much better. As expected the larger WMC, the larger the probability of fault detection.

DIT was shown to be very significant overall. As expected, defect detection is more probable with larger DIT values. Better results showed new and extensively modified classes.

RFC was shown to be very significant overall. Predictably, the larger the RFC, the larger is the probability of defect detection. Reasons are believed to be the same as WMC for new classes, and UI classes show a distribution which is significantly different from that of DB classes.

NOC appeared to be very significant, which is opposite to what was expected. The larger is the NOC; the lower is the probability of defect detection. This can be explained by the fact that most classes do not have more than one child and reuse classes have some larger value for NOC. Since we have observed that reuse was a significant factor in fault density, this explains why large NOC classes are less fault-prone.

LCOM was shown to be insignificant in all classes. This comes out directly from its definition where LCOM is 0 when the number of class pairs sharing variable instances is larger than that of the ones not sharing any instances.

CBO is significant and more particularly so for UI classes. No satisfactory explanation could be found for differences between UI and DB classes.

It is important that various metrics have different units. Most importantly, besides NOC, all metrics appear to have a very stable impact across various categories of classes. It has been also shown that code metrics appear to be somewhat poorer indicator of class fault proneness and OO metrics are better. They have shown that Chidamber and Kemerers OO metric suite seem to be better predictors than the best set of traditional code metrics provided on our data set.

This validation study provides positive confirmation of the value of CK metrics. The authors however caution that several factors may limit the generality of the results. These factors include: small project sizes, limited conceptual complexity, and because the testing suite was performed on student projects.

In 1997 Chidamber, Kemerer and Darcy applied the metric suite on three financial applications and assessed the usefulness of the metrics from a managerial perspective [16]. The systems were developed by single company and used by financial traders to assist in buying, selling, recording and analysis of various financial instruments. At first the researchers noted small values for DIT and NOC indicating that developers were not taking advantage of the inheritance reuse features in OO design. It was also noted that WMC, RFC and CBO were highly correlated. This finding was opposite from the previous framework were Basili stated that all six metrics was found to be relatively independent. The main objective from this research was to explore the CK suite to managerial variables such as productivity, reusability and design effort. In this consideration, productivity was calculated as size divided by the number of hours required. The interpretation of the gathered statistics is that high values of CBO and LCOM were associated with lower productivity, higher effort to make class reusable and greater design effort. This finding is significant because it reflects the strength of the underlying concepts of coupling and cohesion.

In 1999, Rosenberg, Stapko and Gallo regarding to the metrics used at the NASA Software Assurance Technology Center (SATC) **recommended the CK metric suite extended with 3 traditional metrics** adopted for an object oriented environment (Cyclomatic Complexity, Size measuring the lines of code and Comment Percentage) [11]. These authors also used the metrics to point classes with potential problems. Measuring the results, they are giving guidelines that can help developers to improve quality of the programs.

There is a trade off with many of the metrics, such as with DIT high values that can indicate maintainability complexity, but is also an indicator of increased reuse. High value for NOC will increase the testing effort but will also accompany increased the extent of reuse efficiency. As their summary conclusion at that time, is that there are no clear interpretation guidelines for these metrics although there are guidelines based on common sense and experience.

C. Evaluation on C.K. metric suite

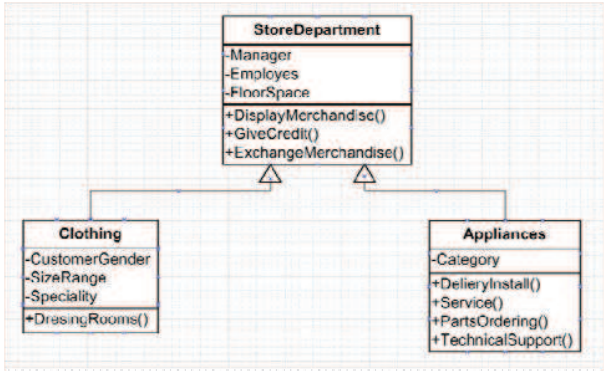

Figure 3 Example 1 for evaluating C.K Metric Suite

Figure 4 Example 2 for evaluating C.K Metric Suite

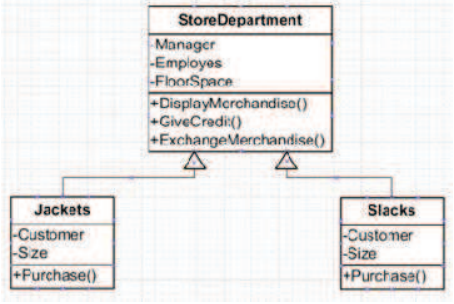

Figure 5 Example 3 for evaluating C.K Metric Suite

WMC - for Clothing department = 1
for Appliance department = 4 (Figure 1)
Used basic WMC calculation: count of methods implemented within a class.
RFC - for Store Department = 8
(Figure 1)
Used: number of methods that can be invoked in response to a message. There are 3 messages that can be invoked by itself, one by Clothing and 4 by Appliances class.
LCOM - High lack of cohesion
(Figure 2)
There are few common methods among the objects. Auto needs PartsOrdering() but not FragranceDemo(). On the other hand, Cosmetics need FragranceDemo() and not PartsOrdering(). Such design implies that further abstraction is required, introducing child classes.
CBO - High coupling (Figure 3)
Jackets and Slacks have the same attributes and methods. These means that changes in the Purchase method has to be done in multiple places.
DIT - for Store Department = 0
for Clothing = 1 (Figure 1)
Store Department is the root class; on the other hand, Clothing has one ancestor.
NOC - for Store Department = 2
for Clothing = 0 (Figure 1)
Store Department has two subclasses, and Clothing is the leaf node in the tree structure.

Although is the most used and well known model, there are improvements suggested for this set so it can be complete and can cover situations that were not taken into consideration, when the set was proposed [8]. Some misunderstandings exist when we try to evaluate the set in certain situations:

WMC – was defined as measure of complexity of methods within a class. What the author missed because of generalization purposes and the ability to use this metric in multiple contexts, is to give well formed definition of complexity.

NOC – initially this metric was set to measure **direct** successors and present measure of the level of reuse, possibility of improper abstraction and the level of testing a class needs [5].

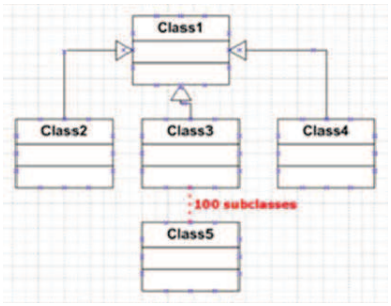

Figure 6 Example 4 for evaluating C.K Metric Suite

In the given scenario, NOC will result with measure of 3 suggesting limited reuse and less testing for the Class 1. As we can observe, Class 3 has 100 subclasses. As long as Class 3 is properly tested, NOC shows good indication that Class 1 needs less testing. However, in this case NOC is not quite correct in the case to suggest low level reuse of Class1. Class1 is reused by Class2, Class3 and Class4, as well as all their subclasses, including 100 subclasses of Class3.

D. Li and Henry Metrics

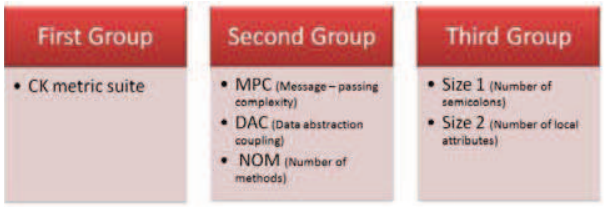

Figure 7 Li and Henry Metric Set

Li and Henry's framework has three sets of metrics. The first group is consisted of CK metrics; the second group contains message-passing coupling and method counting, data abstraction coupling and number of methods; while the third group includes Size 1 and Size 2 metrics.

Shot description of the other metrics besides CK suite is given below:

MPC (Message – Passing Complexity) –number of send statements defined in a class. This is the number of procedure calls originating from a method in the class and going to other classes.

DAC (Data Abstraction Coupling) –number of abstract data types defined in a class. OO introduces abstract data types such as instance variables along with the use of their inherited data types. All the relationship usually known as aggregation relationships are also counted with this metric.

NOM (Number of Methods) –number of local methods in a class.

Size 1 –number of semicolons in a class. It can be considered as a sort of LOC for C-style programming languages.

Size 2 –number of locally defined attributes and methods for a class.

MPC deals with cohesion aspects measuring indirectly the amount of classes needed by the class under examination. It is also related to maintenance aspects, since the change in one of the target classes can influence its behavior. DAC measures aggregation and encapsulation of data in the class. NOM is related to class complexity by counting its functionalities. Size 1 can be considered as modified version of LOC, while Size 2 adds the local attributes to the NOM measure, taking into account the class state by its attributes.

E. Lorenz and Kidd object-oriented Metrics

Lorenz and Kidd introduced eleven metrics. Their metrics are applicable to class diagram and are focused on size, inheritance, internal and external measurements. These metrics can be further classified into four categories.

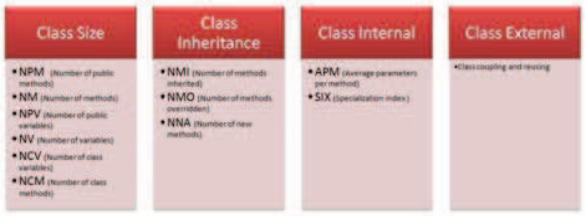

Figure 8 Lorenz and Kidd Metric Suite

Class size metrics – size metrics for the object oriented class. They count attributes and operations for an individual class.

NPM (Number of public methods) counts the number of public methods in a class. It is used as estimation of the amount of work needed to development a class.

NM (Number of methods) counts total number of methods including public, private and protected methods. Indicate classes that have too much functionality.

NPV (Number of public variables per class) counts the number of public variables in a class. If one class has more public variables than another, might imply that the class has more relationships with other objects and is likely to be a key class.

NV (Number of variables per class) counts total number of variables including public, private and protected variables.

NCV (Number of class variables) counts the total number of class variables.

NCM (Number of class methods) counts total number of class methods.

Class inheritance metrics – inheritance based metrics. They are focused on the method in which operations are reused through the class hierarchy.

NMI (Number of methods inherited) measures the number of methods inherited by a subclass.

NMO (Number of methods overridden) Large number of overridden methods indicates design problem. It is suggested that a subclass should be specialization of its super classes, resulting in unique names and operations .

NNA (Number of new methods) Counts newly added methods. A method is defined as added in a subclass if there is no method of the same name in any of its super classes.

Class internal metrics – internal metrics are focused on cohesion and code oriented issue.

APM (Average parameters per method) is the total number of parameters in a class divided by the total number of methods. According to Lorenz and Kidd, value for this metric should not exceed 0.7

SIX (Specialization index) is calculated as (NMO*DIT)/NM and measures to what extend the subclasses redefine the behavior of their super classes

Class external metrics – observe coupling and reuse.

F. MOOD

MOOD (Metrics for Object-Oriented Design) suite was proposed by Fernando Brito e Abreu and Rogerio Capurca and is empirically validated. Newer versions are known as MOOD2 and MOODKIT. In 2003 appeared formal method for representing MOOD2 metrics using OCL (Object Constraint Language).

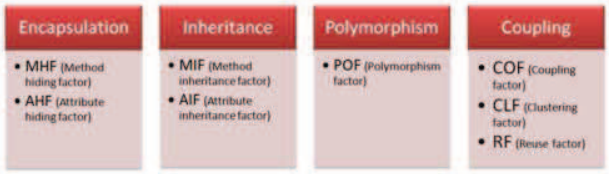

Figure 9 MOOD Metric suite

Original MOOD metric suite consisted of 6 metrics with values ranging from 0 to 1. This model refers to a basic structural mechanism of the object-oriented paradigm, such as encapsulation (MHF, AHF), inheritance (MIF, AIF), polymorphism (POF), and message passing (COF) [6]. MOOD metrics are calculated for two main features: methods and attributes. Methods are used to perform operations. Attributes are used to represent the status of each object in the system. Each feature (method or attribute) is either visible or hidden from a given class.

Encapsulation **-** Method Hiding Factor and Attribute Hiding Factor were proposed as measure of encapsulation. They represent the average amount of hiding between all classes in the system.

MHF (Method Hiding Factor) of a class diagram represents the percentage of invisibilities of methods. MHF is the sum of the invisibilities of all methods defined in all classes. The invisibility of a method is the percentage of the total class from which the method is hidden. MHF is computed by dividing the number of all visible methods in all classes by the number of all methods in all classes. The number of visible methods is a measure of class functionality. High MHF values means there are a lot of private methods which indicates very little functionality and insufficient abstraction. Low MHF values means that many of the methods are public indicating they are not properly protected.

AHF (Attribute Hiding Factor) represents the percentage of attribute invisibilities. AHF is the sum of the invisibility of all attributes defined in all classes. The invisibility of an attribute is the percentage of the total classes from which the attribute is hidden. It is calculated by dividing the number of visible attributes by the number of all attributes in the diagram. High AHF values mean that many of the attributes are private and low AHF values mean that most of the attributes are public.

Inheritance **–** inherited features in a class are those which are not overridden in that class. Method Inheritance Factor (MIF) and Attribute Inheritance Factor (AIF) are proposed to measure inheritance. High values for these metrics indicates either unnecessary inheritance of too wide member scopes. Low values indicate lack of inheritance.

MIF (Method Inheritance factor) represents the percentage of effective inheritance of methods. MIF is the sum of inherited methods in all classes of the system. MIF is calculated by dividing the number of all inherited methods in all classes by the sum of all methods available of all classes. The degree to which the class architecture of an object oriented system makes use of inheritance for both methods and attributes. Very low MIF values mean that the class lack inheritance or there are no methods pointing to lazy classes "bed smell".

AIF (Attribute Inheritance Factor) is the percentage of effective inheritance of attributes. AIF is the sum of inherited attributes in all classes of the system. It is calculate by dividing the number of all inherited attributes in all classes by the sum of all attributes available of all classes. AIF provides an indication of the impact of inheritance in the object oriented software. Very low AIF values indicate lack of inheritance or that the class has no attributes.

Polymorphism **–** is an important characteristic of an object oriented paradigm. Polymorphism measures the degree of overriding in the system. We can intuitively expect that polymorphism can be used as reasonable extent to keep the code clear, but that excessively polymorphic code may be too complex to understand.

POF or PF (Polymorphism Factor) represents the actual number of possible different polymorphic situations with respect to the maximum number of possible distinct polymorphic situations. POF is calculated by dividing the total number of overridden methods in all classes by the result of multiplying of new methods times the number of descendants for all classes. If a project have 0 % POF, it indicates the project that uses no polymorphism, and 100% POF indicates that all methods are overridden in all derived classes.

Coupling **–** shows the relationship between modules. A class is coupled to another class if it calls methods of another class. Coupled systems are complex, non-maintainable and have reduced potential of reusing. High COF values should be avoided.

COF (Coupling Factor) represents the percentage of couplings between classes. It is calculated by dividing the number of associations between all classes by the number of classes squared minus the number of classes. It is reasonable to conclude that as "the COF value" increases, the complexity of object oriented design will also increase, and as a result the understandability, maintainability and the potential for reuse may suffer. 0% COF indicates that classes are no coupled, and 100% indicates that all classes are coupled with all other classes. COF is supposed to have low values, and increasing COF values should be taken seriously. COF is similar to CBO because they both use the coupling factor. The main difference is that in COF all variables access are counted whereas CBO metric does not count variables.

CLF (Clustering Factor) represents the percentage of actual number of standalone class hierarchies (clusters) with respect to the maximum possible number of coupling in the class clusters. CLF is computed by dividing the number of class clusters in the by the number of classes in a class diagram.

RF (Reuse Factor) represents the percentage of classes that are specializations of previously defined classes. The parent classes may be external to the class diagram, or internal from super classes. It cannot be calculated solely from the class diagram because of external libraries that are used.

MOOD 2 metrics – is a latter addition from the author of MOOD model, that introduces new metrics such as OHEF/AHEF (Operation/Attribute Hiding Factor that measures the goodness of scope settings on class operation); IIF (Internal Inheritance Factor that measures the amount of internal inheritance in the system); PPF (Parametric Polymorphism Factor that is the percentage of the classes that are parameterized – parameterized class is generic class) and other metrics.

It has been observed that majority of the MOOD Metrics are fundamentally flawed because they either fail to meet the MOOD team's own criteria or are founded on an imprecise or inaccurate.

G. Misunderstandings in the evaluation on MOOD metric suite

Encapsulation (MHF / AHF) – the number of private methods does not tell us anything about the degree of information hiding in the component. It may tell us that particular methods have been broken down into smaller methods to avoid duplication or for clarity of understanding. In the following example [5] both classes have equal "information – hiding" levels:

```
Class A                          Class B
{                                {
  private int x;                   private int x;
  public int m0()                  public int m0()
  {                                {
    do_1;                            m1();
    do_2;                            m2();
    do_3;                            m3();
    return x;                        return x;
  }                                }
}                                  private void m1(){do_1;}
                                   private void m2(){do_2;}
                                   private void m3(){do_3;}
                                 }
```

Figure 10 Example 1 for evaluating MOOD Metric Suite

In class A all of the behavior is contained in the body of A whereas in class B it has been separated in three smaller methods. As can be concluded a count of the private methods is not particularly useful metric, and certainly does not contribute anything to our knowledge of a component's encapsulation level.

Inheritance (MIF / AIF)

Considering the following hierarchical structure:

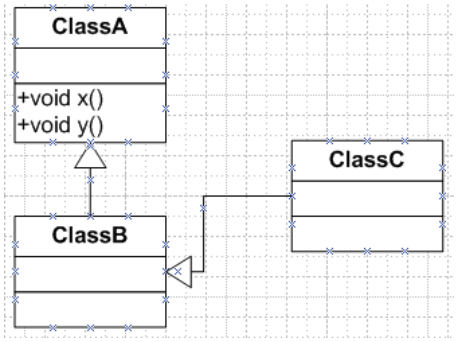

Figure 11 Example 2 for evaluating MOOD Metric Suite

B and C inherit the two methods defined in A and have no further methods. This is the maximum possible method inheritance in the system and intuitively is seams the MIF should be 100%, but in fact is 66%:

Mi(A)=0 Mi(B)=2 Mi(C)=2

Ma(A)=2 Ma(B)=2 Ma(C)=2

Mi – inherited methods

Ma – available methods

If component C in the above example had a new method added it **should not change the MIF value**, as it is consistent with our intuitive understanding of method inheritance. With calculation we found that MIF has changed since Ma(C)=3, so MIF=4/7=57%.

Polymorphism (POF) – Systems often extend frameworks. When measuring such a system it should be only the components that belong to the system to be measured and the ones outside the boundaries should not be considered. In such

cases the denominator for POF may be less than the numerator, resulting in value greater than 1, which is contrary that the fact that POF values range from 0 to 1 (0% to 100%). Such situation can be shown with the following example:

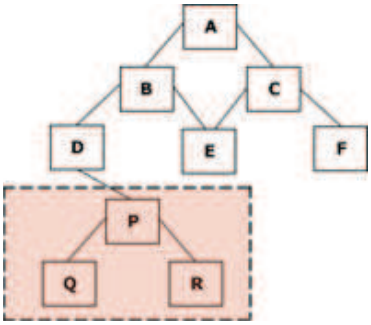

Figure 12 Example for evaluating MOOD Metric Suite

P – overrides 1 method, adds 2 new methods
Q – overrides 2 methods, adds 2 new methods
R – overrides 2 methods, adds 2 new methods
Mo(P)=1 Mo(Q)=2 Mo(R)=2
(Mn(P)=2 * DC(P)=2)=4
(Mn(Q)=2 * DC(Q)=0)=0
(Mn(R)=2 * DC(R)=0)=0

Mo – overridden methods
Mn – new methods
DC- descendants

Therefore, POF for the system is (1+2+2) / (4+0+0) = 5/4 > 1. These can a typical situation in languages that are shipped with large component libraries.

Coupling (COF) – There are two types of relationships: inheritance and when one component uses the other component as instance variable (client supplier relationship). There are situations where mixture of both types can be found. As an example we can take Component and Container in java.awt library. Component is the super component of all graphical components and Container is one of its subcomponents. Thus the two are in inheritance relationship. However, each component also uses an attribute of the other component type. The question that the MOOD team does not adequately answer is whether a client supplier relationship under these conditions is counted. There is no "correct" way of dealing with these situations in term of the COF metric.

H. QMOOD

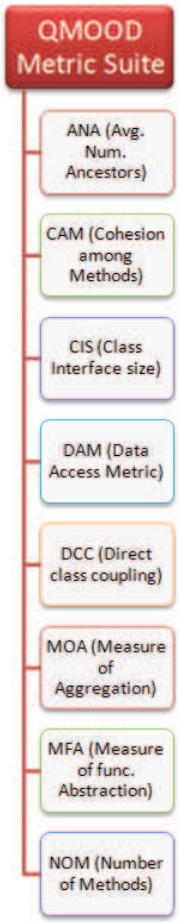

Figure 13 QMOOD Metric Suite

QMOOD (Quality Model for Object-Oriented Design) is a comprehensive quality model that establishes a clearly defined and empirically validated model to assess object-oriented design quality attributes such as understandability and reusability, and relates them through mathematical formulas, with structural object-oriented design properties such as encapsulation and coupling [6]. The QMOOD model consists of six equations that establish relationship between object-oriented design quality attributes (reusability, flexibility, understandability, functionality, extendibility and effectiveness) and eleven properties. For example reusability is function of the coupling measure, cohesion measure, messaging measure and the design size.

ANA (Average Number of Ancestors) – is the average value of DIT measure for all classes in the system. It is connected with the inheritance as OO attribute.

CAM (Cohesion among Methods) – is measure of cohesion. This is based on similarity of method signatures in a class.

CIS (Class Inheritance size) – counts the public methods in a class. This metric is connected with coupling as OO attribute.

DAM (Data Access Metric) – is the ratio of private and protected attributes to the total number of attributes declared in a class. Refers to information hiding as OO attribute.

DCC (Direct class coupling) – counts the classes that accept instances of a given class a parameter plus classes including attributes of given class type. Refers to coupling as OO attribute.

MOA (Measure of aggregation) – the percentage of data declaration in the system whose types are of user defined classes, as opposed to those of system defined classes such as integers, real numbers etc. It is connected to class OO attribute.

MFA (Measure of functional abstraction) – analogue to MIF metric defined in the MOOD suite. Connected to information hiding as OO attribute.

NOM (Number of Methods) – counts the methods in a class. It is the same as WMC when methods counted are considered with equal unity.

In the literature there are other proposed metric suits and metrics.

Chen proposed: (Class Coupling Metric), OXM (Operating complexity Metric), OACM (Operating Argument Complexity Metric), ACM (Attribute Complexity Metric), OCM (Operating Coupling Metric), CM (Cohesion Metric), CHM (Class Hierarchy of Method) and RM (Reuse Metric). Metric 1 to 3 is subjective in nature; metrics 4 to 7 involves count of features and 8 are Boolean. To validate these metrics the authors conduct an experiment involving 6 "experts" whose subjective class scores are regressed against the eight metrics.

Thomas and Jacobson Class Complexity (1989) – deals with aspects related to local attributes and methods and is a weighted sum of all local and inherited methods and attributes. On this basis, different metrics for measuring size of complexity of the class can be created.

Henderson-Sellers Class Complexity (1991) – is an extension of Thomas and Jacobson and adds a component related to the inherited methods. It can also generate several other different metrics.

There is no individual research of which of these metrics is significant in prediction.

V. Notes on Quality Models

The work of Chidamber and Kemerer has been the basis in defining and validating quality models. Lorenz and Kidd metrics are criticized for not being a part of quality model, however they have the advantages of being well defined, easy to collect and could be computed in the early phases. MOOD model is very well defined, through mathematical formulas and OCL statements, empirically validated, and provides thresholds that could be used for judgments. QMOOD has similar properties as MOOD but distinguishes itself by providing mathematical formulas that links design quality attributes with design metrics.

The impact of quality models has been widely used and empirically validated. Different and sometime opposite results has been introduced in the literature. Among them are the following:

Demeyer and Ducasse tested the object-oriented metrics and their impact when frameworks are developed [15]. They found that size and inheritance metrics (gathered from multiple sets) are not reliable in framework development environment. Although, these metrics were found as important when they provide results between different versions and in this situation they can be considered as stability indication. Bruntink and Deursen used the object-oriented metrics for creating a model for system testability [18]. They used the metric set defined by Binder which is based on the C.K. metric suite. The conclusion was that there is an important connection between class level metrics and testability metrics (variations of LOC and NOTC – Number of test cases). The research is complete and gives detailed situations how specific metrics affect the testability of a system.

VI. Conclusion

The concerns about metrics and quality suites is because there is large number of proposed measures, many of them are similar; there is large number of external attributes of interest; lack of reliable and complete data sets; it is still difficulty in integrating quality prediction models in realistic decision processes. Despite the difficulties, there is a huge amount of reported studies that draw important conclusions about the usefulness of the metrics.

Although the Object-oriented metrics are newer in the measurement theory, they have proven as useful predictors of good system design. Object-oriented metrics have been grouped to minimal sets to assess quality of the Object-oriented systems. There are couples of proposed sets, which have been validated, empirically tested and applied on the real systems. The common thing about these suits is that they cover the same basic predictions of fault-proneness, effort, and basic quality attributes. Implementing the positive side of prediction and applying the guidelines, the system can improve its design, lower complexity and become easy to use.

References

[1] Linda M. Laird, M. Carol Brennan (2006) Software Measurement and Estimation: A Practical Approach

[2] Stephen H. Kan (2002) - Metrics and Models in Software Quality Engineering, Second Edition
[3] Christof Ebert, Reiner Dumke, Manfred Bundschuh, Andreas Schmietendorf (2004) – Best Practices in Software Mesurement
[4] Khaled El Emam – A Primer on Object-Oriented Measurement
[5] Abreu - Metric Suite Evaluation
http://www.comp.nus.edu.sg/~bimlesh/oometrics/Findings/Comments%20on%20metrics.pdf
[6] Mohamed El-Wakil, Ali El-Batawisi, Mokhtar Boshra, Ali Fahmy – Object-Oriented Design Quality Models: A Survey and Comparison
http://homepages.wmich.edu/~m5elwakil/INFOS04_ElWakil.pdf
[7] Fernando Brito e Abreu, Walcélio Melo - Evaluating the Impact of Object-Oriented Design on Software Quality
[8] Mark Schroeder – A Practical Guide to Object-Oriented Metrics
http://www.cin.ufpe.br/~inspector/relacionados/metricsbymark.pdf
[9] Muktamyee Sarker - An overview of Object Oriented Design Metrics
http://www.cs.umu.se/education/examina/Rapporter/MuktamyeSarker.pdf
[10] Seyyed Mohsen Jamali - Object Oriented Metrics
http://ce.sharif.edu/~m_jamali/resources/ObjectOrientedMetrics.pdf
[11] Linda H. Rosenberg - Applying and Interpreting Object Oriented Metrics http://www.literateprogramming.com/ooapply.pdf
[12] Victor R. Basili, Lionel Briand, Walcélio L. Melo – A Validation of Object-Oriented Design Metrics as Quality Indicators
http://www.cs.umd.edu/~basili/publications/journals/J62.pdf
[13] Wei Li, Sallie Henry – Object-Oriented Metrics Which Predict Maintainability http://eprints.cs.vt.edu/archive/00000347/01/TR-93-05.pdf
[14] Cem Kaner, Walter P. Bond - What Do They Measure and How Do We Know? http://www.kaner.com/pdfs/metrics2004.pdf
[15] Serge Demeyer, Stephane Ducasse - Metrics, Do They Really Help?
http://scg.unibe.ch/archive/papers/Deme99aDemeyerDucasseLMO99.pdf
[16] Shyam R. Chidamber, David P. Darcy, and Chris F. Kemerer - Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis
http://www.pitt.edu/~ckemerer/CK%20research%20papers/ManagerialUseMetrics_ShidamberDarcyKemerer98.pdf
[17] S Kanmani, V Sankaranarayanan, P Thambidurai - Evaluation of Object Oriented Metrics http://www.ieindia.org/pdf/86/pcn5fl4.pdf
[18] Magiel Bruntink, Arie van Deursen - Predicting Class Testability using Object-Oriented Metrics
http://www.st.ewi.tudelft.nl/~arie/papers/testability/scam04.pdf