Object Oriented Measurement

Diego Chaparro González
dchaparro@acm.org
Student number: **59881P**

17th January 2003

Abstract

This document examines the state of art in software products measurement, with focus in the object oriented approach, which has become high popular because of his benefits: quick development, reusability, complexity management, etc., all of this, characteristics that increase directly the quality of software products.

Contents

1 Introduction

It is clear that software measurement is necessary for the software development process to be successful. The main goals of the software measurement are:

- Evaluate the software systems.
- Improve quality of software systems.
- Identify and correct problems early.
- Defend and justify decisions.

One side of the concept of software engineering is the idea that the software should be under control. As De Marco said: ”You cannot control what you cannot measure”[De Marco, 1982]. Fenton and Pfleeger added: ”You cannot predict what you cannot measure” [Fenton and Pfleeger, 1997].

Most of the measure strategies has as the main goal to evaluate the different characteristics of software quality, such as reliability, ease of use, maintainability, robustness, ...

2 Metrics

2.1 Measurement history

The earliest software measure is the Measure LOC (Lines Of Code). The basis of the Measure LOC is that program length can be used as a predictor of program characteristics such as reliability and ease of maintenance.

Then in the 70's, they tried to set the relation between the programs size and the needed effort to write and maintain them (COCOMO).

In the 80's they studied more about complexity metrics, module and structured design, cost estimation and time used to develop the software projects. By this time the norm ISO 9000 appeared.

In the 90's they tried to center in the classification, study and validity of the software metrics. COCOMO model was reviewed (COCOMO II) to show the big changes in the development techniques during the last ten years.

2.2 Metrics properties

2.2.1 Factors in software quality

Quality factors are usually attributes in a high level of abstraction, as reliability, ease of use, ease of maintenance.

The McCall quality software model [McCall et al., 1977] defines three aspects or important characteristics of a product: operational characteristics, of modification and transaction, and each of them are broken down in quality factors of high level. These are broken down in quality criterion. And these are associated to an attributes set named quality metrics.

- **Operational characteristics**

  Reliability:

    Correctness: a program is correct when it make his work without errors.

    Strength: ability of a program to answer to non predicted situations.

  Efficiency: to offer the maximum of performance qualities with the minimum resources.

  Ease of use: needed effort to learn to control a program

  Integrity: control the access to software or data by non authorized people.

- **Modification characteristics**
  - Ease of maintenance: the needed effort to modify a program.
  - Expandable: possibility of add new functionality of an easy way.
  - Ease of testing: the needed effort to test a program in a way that this testing assures that the program works well.
- **Transition characteristics**
  - Portability: possibility of execution in different platforms without changes.
  - Re-usability: property of software components. This components can be used in another tasks from which they were built.
  - Interoperability: the systems should be able to inter-operate with another software components.

There are two kind of attributes:

- Internal attributes: those which can be measured with the own product independently of his behaviour.
- External attributes: those which can only be measured when it relates with the environment.

The measurement of internal and external attributes is considered as the basis to improve the quality of software products.

2.2.2 Measurement and metrics properties

Measurement is "the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules"[Fenton and Pfleeger, 1997]. An entity may be a person, a developed program or the development process. An attribute of entity may be the weight of a person or the length of the development process. The goal of the metrics use is evaluate systems to get high quality, strength and ease of maintenance.

Metrics can be applicable to one or more steps in the life cycle of software development: requirement specification, analysis, design, implementation, ...

There are two kind of metrics:

- Direct metrics: those which only need an attribute to measure it.
- Indirect metrics: those which need more than one attribute to measure it. It need a combination of some metrics.

2.2.3 Metrics validation

A measure is valid if it accurately characterizes the real world behaviour it claims to measure [Fenton and Pfleeger, 1997].

As [Kitchenham et al., 1995] says, in order for a measure to be valid, both of the following conditions must hold:

- the measure must not violate any necessary properties of its elements.
- each model used in the measurement process must be valid.

And to decide whether a measurement is valid we need to confirm [Kitchenham et al., 1995]:

- attribute validity, whether the attribute in which we are interested is actually exhibited by the entity we are measuring. Attribute validity must be considered both for directly measurable attributes and for indirectly measurable attributes that are derived from other attributes.
- unit validity, whether the measurement unit being used is an appropriate means of measuring the attribute.
- instrument validity, whether any model underlying a measuring instrument is valid and the measuring instrument is properly calibrated.
- protocol validity, whether an acceptable measurement protocol is adopted.

3 Object oriented approach

One of the problems of any software project is to be able to manage the concepts, flow of control, and data. The solution to this problem may be the object oriented approach, which groups both data and procedures into and entity called and object. These objects allow a programmer, designer, and analyst to examine larger cognitive blocks of a project, and thus clarify the programming process.

3.1 Elements of the Object Oriented Approach

The main elements in the object oriented approach are:

- Identity. Data is organized in entities called objects.
- Objects. Each object has his own identity which distinguish from another one even if both have the same values in their attributes.
- Classification. Objects with the same attributes and behaviour are grouped in classes.
- Class. A class is an abstraction which describes the important properties of a set of objects for an application.
- Polymorphism. The ability of two or more objects to interpret a message differently at execution, depending upon the superclass of the calling object.
- Inheritance. A relationship among classes, wherein one class shares the structure or methods defined in one other class or in more than one other class.
- Abstraction. The essential characteristics of an object that distinguish it from all other kinds of object.
- Encapsulation. The process of bundling together the elements of an abstraction that constitute its structure and behaviour.
- Information hiding. The process of hiding the structure of an object and the implementation details of its methods. An object has a public interface and a private representation; these two elements are kept distinct.
- Specialization. Get subclasses from a superclass.

4 Object oriented metrics taxonomy

Software Engineering introduce the measures in each step in a life cycle of a software project, independently of the used model: waterfall, spiral, ...

Then, the metrics can be viewed from a three-dimensional approach, with the next dimensions:

- Software attribute to measure (complexity, re-usability, ...)
- Step in the life cycle in which is done the measure (design, analysis, ...)
- Granularity level in which the measure is taken (system level, program level, class level, ...)

Metrics cannot be applied to any software attribute that want to be measured indiscriminately. The typical case of a mistaken measure is to measure the lines of code (LOC) as a complexity program measure, when this is valid as a measure of the size program, not as complexity program measure.

In another way, all the metrics don't have to be taken in the implementation stage, although part of them are taken from this step. It should be desirable to get them in the earlier design step.

[Chindamber and Kemerer, 1994] proposed six metrics for object oriented design:

- Metric 1: Weighted Methods Per Class (WMC)
- Metric 2: Depth of Inheritance Tree (DIT)
- Metric 3: Number Of Children (NOC)
- Metric 4: Coupling Between Objects (CBO)
- Metric 5: Response For a Class (RFC)
- Metric 6: Lack of Cohesion in Methods (LCOM)

The appearance of the Unified Modelling Language (UML) [Booch et al., 1999] as a standard of modelling object oriented information systems have provided a great contribution toward building quality object oriented systems. In [Genero et al., 2000] propose a set of metrics in order to assess the complexity of UML class diagrams from the relations in UML, such as association, aggregation, ...

[Briand et al., 1999] made a study about the coupling in object oriented systems:

- There are different types of coupling among classes, methods, attributes.
- Classes and methods can be coupled more or less strongly, depending on:
  - The type of connections between them
  - The frequency of connections between them
- A distinction can be made between import and export coupling.
- Both direct and indirect coupling may be relevant.
- The server class can be stable or unstable.
- The effect of inheritance on coupling has to be considered.

The study of [Abreu and Melo, 1996] is based in a suit of metrics called MOOD, with the next motivations:

- coverage of the basic structural mechanisms of the object oriented paradigm as encapsulation, inheritance, polymorphism and message-passing.
- formal definition to avoid subjectivity of measurement and thus allow replicability
- size independence to allow inter-project comparison, thus fostering cumulative knowledge
- language independence to broad the applicability of this metric set by allowing comparison of heterogeneous system implementations.

In the next sections a taxonomy of metrics will be presented, grouped with the main characteristic of the object oriented approach that is wanted to measure, showing in each of them the granularity level and the step in the life cycle in which it is measured.

4.1 Coupling

It is viewed as a measure of the complexity increment, reducing the encapsulation and his possible reutilization; limiting the ease of understanding and system maintenance.

4.1.1 Coupling between objects (CBO)

[Chidamber and Kemerer, 1994]

- Definition: CBO of a class is the number of classes to which a particular class is coupled, without have inheritance relations with it. A class is coupled with another class if either one accesses a method or attribute of the other.
- Valuation: if an object is more independent, more easily is to reuse it in another application. When the coupling is reduced, the complexity is reduced, the modularity is improved and the encapsulation is promoted. A coupling method is useful to know the complexity of the necessary testing of the different parts of the design.
- Granularity level: class, program, system.
- Life cycle: design.

4.1.2 Coupling factor (COF)

[Abreu and Melo, 1996]

- Definition: is the proportion between the real number of coupling and the maximum possible number of coupling in the system. It shows the communication between classes.
- Valuation: COF may be an indirect measure of the attributes in which is related: complexity, lack of encapsulation, lack of re-usability, ease of understanding and bit ease of maintenance. With the coupling increment between classes, density of faults is increased and ease of maintenance is reduced.
- Granularity level: class, program, system.
- Life cycle: design.
- Related with: complexity, encapsulation, re-usability.

4.2 Cohesion

4.2.1 Lack of Cohesion in Methods (LCOM)

[Chindamber and kemerer, 1994]

- Definition: number of local method groups which don't access to common attributes.
- Valuation: it shows the quality of the abstraction done in the class. It uses the concept: rate of similarity methods. If there is not common attributes, the similarity rate is zero. A low cohesion increase the complexity and the ease to make faults during the development process. These classes would probably can be divided in two or more subclasses increasing the cohesion of the final classes. It is desirable a high cohesion in the methods inside a class, because it cannot be divided promoting encapsulation.
- Granularity level: class.
- Life cycle: design, implementation.

4.3 Complexity

4.3.1 Response For a Class (RFC)

[Chidamber and Kemerer, 1994]

- Definition: RFC 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.
- Valuation: is a measure of the class complexity with the number of methods and the communications with another, because it includes methods called from outside the class. When the RFC is larger, more complexity has the system, because it is possible to invoke more methods as answers to a message, with larger understanding, which involve more time and effort of testing and debugging.
- Granularity level: class, program, system.
- Life cycle: design.
- Related with: coupling.

4.3.2 Weighted methods per Class (WMC)

[Chidamber and Kemerer, 1994]

- Definition: consider a class C_1, with methods M_1, ..., M_n and c_1, ..., c_n the complexity of the methods. WMC is defined as the sum of the complexity of each method in a class. If all methods are considered the same complexity, then $c_1 = 1$ and $WMC = n$ (number of methods).

- Valuation: describe the algorithm complexity of a class in terms of the complexity of all his methods. It is linked with the quality of the complexity method definition (c_i). The authors simplify it, assigning 1 to each method, then it is a counter of methods number in a class.

- Granularity level: class, program, system.

- Life cycle: design

4.3.3 Average (v(G))

[McCabe, 1976]

- Definition: is the ciclomatic complexity average of the methods in the class. The ciclomatic complexity v(G) of a method is the number of independent ways in the method. In structured methods, v(G)= number of decision nodes plus one.

- Valuation: is a measure of structural complexity. The measure in more complex methods is the maximum ciclomatic complexity (Maximum v(G)). It is useful to know the difficult of testing and maintenance of a program.

- Granularity level: class, program, system.

- Life cycle: implementation.

4.4 Encapsulation

4.4.1 Method Hiding Factor (MHF)

[Abreu and Melo, 1996]

- Definition: is the proportion between the invisibility degree of methods in all classes and the total number of defined methods in the system. Method invisibility degree is the percentage over the total number of classes from where a method is not visible. It is to say, MHF is the proportion between the protected and private methods and the total number of methods.

- Valuation: MHF is proposed as a measured encapsulation, relative amount of hiding information. When MHF is increased, density of faults and the needed effort to correct them should decrease. Inheritance methods are not considered.
- Granularity level: class.
- Life cycle: implementation.

4.4.2 Attribute Hiding Factor (AHF)

[Abreu and Melo, 1996]

- Definition: is the proportion between the invisibility degree of attributes in all classes and the total number of defined attributes in the system. The invisibility degree of an attribute is the percentage over the total number of classes from where an attribute is not visible. It is to say, AHF is the proportion between the protected and private attributes and the total number of attributes.
- Valuation: Ideally this metric value should always be 100%, trying to hide all the attributes. Public attributes should not be used, because it violates the encapsulation principles. To improve the performance, sometimes it is avoided the use of methods that access or modify attributes directly.
- Granularity level: class.
- Life cycle: implementation.

4.5 Inheritance

The inheritance use has to be shown as a compromise between the ease of re-usability that it provides and the ease of understanding and maintenance.

- High inheritance levels show complex objects, which can be difficult to testing and reusing.
- Low inheritance levels could show that the code is written in a functional way, without use the inheritance property provided by the object oriented programming.

4.5.1 Method Inheritance Factor (MIF)

[Abreu and Melo, 1996]

- Definition: is the proportion between the sum of all inherited methods in all classes and the total number of methods (locally defined plus inherited methods) in all classes.
- Valuation: is an indicator of the re-usability level. It is proposed as a help to evaluate the amount of resources needed to testing.
- Granularity level: class, program, system.
- Life cycle: design, implementation.
- Related with: re-usability.

4.5.2 Attribute Inheritance Factor (AIF)

[Abre and Melo, 1996]

- Definition: is the proportion between the number of inheritance attributes and the total number of attributes.
- Valuation: is an indicator of the re-usability level.
- Granularity level: class, program, system.
- Life cycle: design, implementation.
- Related with: re-usability.

4.5.3 Specialization Index per Class (SIX)

[Lorenz and Kidd, 1994]

- Definition: shows how the subclasses redefine the behaviour of the super-classes.
- Valuation: when frameworks are used, some methods has to be redefined: these methods has not to be taken in account when this metric is calculated. SIX is proposed as a inheritance quality measure. SIX can shows when there are too much redefined methods, in a way that some abstractions have to change their behaviour. In general, a subclass should extend the behaviour of the superclass with new methods instead of modify the existing methods, redefining them.

- Granularity level: class.
- Life cycle: design, implementation.

4.5.4 Depth of Inheritance (DIT)

[Chidamber and Kemerer, 1994]

- Definition: DIT measure the maximum level in inheritance hierarchy. It is the count of the levels in the inheritance hierarchy. In level zero there is the root class.
- Valuation: DIT is proposed as a complexity class measure, design complexity and potential re-usability, because when a class is more depth in the hierarchy, higher is the probability of inherit more methods. Systems built from frameworks often show high inheritance levels, because the classes are built from an existing hierarchy. In languages as Java or Smalltalk, classes always inherit from Object class, which plus one to DIT.
- Granularity level: class.
- Life cycle: design.

4.6 Polymorphism

4.6.1 Polymorphism Factor (POF)

[Abreu and Melo, 1996]

- Definition: is the proportion between the real number of possible polymorph situation for a class C_i and the maximum number of possible polymorph situations in C_i. It is to say, it is the number of inherited redefined methods divided by the maximum number of possible polymorph situations.
- Valuation: POF is a polymorphism measure and it is an indirect measure of dynamic association in a system. Polymorphism is due to inheritance. In some cases, overloading methods reduces the complexity, and then, it is increased the ease of maintenance and understanding of the system.
- Granularity level: class.
- Life cycle: design, implementation.
- Related with: inheritance.

4.7 Re-usability

4.7.1 Number of Children (NOC)

[Chidamber and Kemerer, 1994]

- Definition: NOC is the number of subordinate subclasses in a class hierarchy, it is to say, the amount of subclasses that belong to a class.
- Valuation: it is an indicator of re-usability level, the possibility of create mistaken abstractions, and it is and indicator of required testing. A high number of children require more testing in the method of this class, and a high difficulty to modify a class, because it influences in all the children of this class. Classes in a high level of the hierarchy should have more subclasses than the classes in a lower level in the hierarchy. NOC may be an indicator of the mistaken use of inheritance. It is a potential indicator of the influence than a class can have over the system design. If the system depends a lot of re-usability by inheritance, maybe would be better to divide the functionality in some classes.
- Granularity level: class.
- Life cycle: design.

4.8 Size

4.8.1 Lines of Code per method (LOC)

[Lorenz and Kidd, 1994]

- Definition: LOC is the number of active lines of code (executable lines) in a method.
- Valuation: the method size is used to evaluate the understanding ease, re-usability capacity and maintenance ease of code It depends on the programming language and the methods complexity. In object oriented systems the lines of code number should be low. It is not a recommended metric to use in object oriented systems, but it is easy to get and use.
- Granularity level: method, class, program, system.
- Life cycle: implementation.

4.8.2 Number of Messages Send (NOM)

[Lorenz and Kidd, 1994]

- Definition: NOM measure the number of sent messages in a method, classified by the message type. Types:
  - Unary: messages without arguments.
  - Binary: messages with one argument which belongs to a special type.
  - Password: messages with one or more arguments.
- Valuation: NOM quantifies the method size of a slant way. A high value can show functional style and/or poor responsibility assignment.
- Granularity level: method.
- Life cycle: implementation.

5 Other metrics

5.1 Testing coverage metrics

It is oriented to white-box testing. There are some types:

- Sentences set coverage: is the ratio of executed sentences set by testing set to total number of program sentences. A sentences set is a piece of code which is executed sequentially, without branches.
- Branches coverage: is the ratio of the number of branches in the testing set to total number of program branches.
- Ways coverage: any set of sentences defined by the control flow is a possible way.
- Data flow coverage: is the ratio of pairs variables definition-use and data structures in the testing set to total number of pairs definition-use of the program.

Testing has to be designed with the goal of find the maximum number of faults with the minimum amount of effort and time.

But in object oriented, traditional coverage is not enough, because when a method is inherited by a class is necessary an additional test.

Polymorphism and encapsulation are main characteristics of object oriented design, and coverage metrics have to consider these characteristics in the software testing. [IPL 1999] shows an extension to traditional structural coverage: object oriented context coverage, divided in three types:

5.1.1 Inheritance Context Coverage (ICC)

[IPL, 1999]

- Definition: it considers the methods execution in the base class context and in all derived classes context.
- Valuation: it helps to measure if the polymorphed calls in the system have been tested correctly. [Harrold, 1999] proposes the execution of hierarchical integration testing (HIT), that as the first step execute all methods in the context of a particular class. This proposal has effect in all classes, of such way that redefinitions of a method in a derived class (overload method) are tested with the same detail level that the original base class.

- Granularity level: method, class.
- Life cycle: implementation.

5.1.2 State-based Context Coverage (SCC)

[IPL, 1999]

- Definition: consider the method execution in the class context which are described as state machines, which behaviour depends on particular state in each moment.
- Valuation: it helps to measure if have been done coverage testing for each possible state in a class. A difficulty in implementation used to be the non availability of actual class state, and usually the code should be changed in order to the class returns the possible states.
- Granularity level: class.
- Life cycle: implementation.

5.1.3 Multi-threaded Context Coverage (UCC)

[IPL, 1999]

- Definition: consider the method execution in the multi-threaded context, maintaining coverage information for each thread.
- Valuation: if allows to apply the context coverage approach in other cases in which traditional coverage is not appropriate.
- Granularity level: class.
- Life cycle: implementation.

6 Conclusion

The metrics proposed by different authors are well-known: Chidamber and Kemerer, Brito and Abreu, Lorenz and Kidd, etc. In this paper has been reviewed the most representative metrics of this authors and others, from the perspective of the object oriented approach, more than the analysis of the metrics set of a single author.

Special interest is placed in coverage metrics. Special characteristics in object oriented as polymorphism and encapsulation suggest new focus in the test cases design to get the 100% of method coverage.

7 References

Abreu and Melo, 1996 Brito e Abreu F. and Melo, W., *Evaluating the impact of object oriented design on software quality*, Proceedings of 3rd international software metrics symp., 1996.

Booch et al., 1999 Booch, G., Rumbaugh J. and Jacobson, I., *El lenguaje Unificado de Modelado: Guía de usuario.* Addison Wesley, 1999.

Briand et al., 1999 Briand, L.C., Daly, J.W. and Wüst, J.K., *A unified framework for Coupling Measurement in Object-Oriented Systems*, IEEE transactions on software engineering, 25(1), 1999.

Chidamber and Kemerer, 1994 Chidamber, S.R. and Kemerer, C.F., *A metric suite for object oriented design*, IEEE transactions on software engineering, 20(6), 1994.

De Marco, 1982 De Marco, T., *Controlling software projects*, Yourdon Press Prentice-Hall, 1982.

Fenton and Pfleeger, 1997 Fenton, N. E. and Pfleeger, S.L., *Software metrics. A rigorous and practical approach*, PWS Pub., 1997.

Genero et al., 2000 Genero, M., Manso, M^a.E., Piattini, M. and García, F.J., *Early metrics for object oriented information systems.* In Proceedings of the 6th International Conference on Object Oriented Information System (OOIS-2000), 2000.

Harrold, 1999 Harrold, M.J. and McGregor, *Incremental Testing of Object-Oriented Class Structures*, 1999.

IPL, 1999 IPL Information Processing Ltd., *Advanced Coverage Metrics for Object-Oriented Software, 1999*

Kitchenham et al., 1995 Kitchenham, B. Pfleeger, S. and Fenton, N. *Towards a Framework for Software Measurement Validation*, IEEE Transactions on Software Engineering, 21(12), December 1995

Lorenz and Kidd, 1994 Lorenz, M. and Kidd, J., *Object oriented metrics*, Prentice Hall, 1994.

McCabe, 1976 McCabe, T.J., *A Complexity Measure*, IEEE Transactions on Software Engineering, Vol. 5, 1976.

McCall et al., 1977 McCall, J.A., Richards, P.K. y Walters, G.F. *Factors in software quality.* US Rome Air Development Center Reports NTIS AD/A-049 014, 015. 055 USA Air Force, 1977.