• 1400/91308
COPIA 1

Design Quality Metrics for Object-Oriented Software Development

Alonso Peralta
Joan Serras
Olga Slavkova

Report LSI-95-4-R

Facultat d'Informàtica
de Barcelona - Biblioteca
- 2 FEB. 1995

Design Quality Metrics for Object-Oriented Software Development

Alonso Peralta   Joan Serras   Olga Slavkova

peralta@lsi.upc.es   serras@lsi.upc.es   slavkova@lsi.upc.es

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Pau Gargallo, 5
Barcelona (Spain)

Abstract

The availability of metrics for measuring software design quality and complexity are a great help in the development of such systems.

Application of such metrics in the initial stage of the software development process (specification, analysis and design) allows us to detect those software designs which are excessively complex or intricate at a time when correction costs are still low. In this way, we are able to avoid all the negative consequences that badly-designed systems may involve, such as, high maintenance cost, poor re-use, proneness to error, low portability to different environments, ...

Whereas there is a large number of studies and reviews on the measurement of systems which have been developed by means of procedural methods, the reverse is the case for systems designed through the object-oriented paradigm. In this paper we present a collection of measures which can be useful for measuring object-oriented systems and which may also help to extend the research so far carried out in this field.

1. Introduction

In this paper we shall make a description of a set of 31 measures which can be significant in order to measure the design quality of systems which have been developed by means of object-oriented tecnologies. It is our aim not only to make these measures directly applicable to the programme code but also that they should be applied in the analysis and design stages in object-oriented methodologies.

It is during these initial stages of the process that the application of design quality metrics is really effective, thus allowing the detection of excessively complex or intricate systems at a time when correction costs are still low. These metrics are applicable to the code itself as well, but at such stage of application the measures act as mere sources of information about design quality and complexity. Design correction during such advanced stages of the process would entail, in order to obtain satisfactory results, a great effort both in terms of time and human resources, and therefore great economic expense.

We shall approach this matter in three consecutive steps. The first one, wich is described in this paper, consists in defining a set of candidate measures. We shall do so by drawing upon three sources:

- the existing literature dealing with metrics for systems which have been developed by means of structured methods.
- the literature dealing with metrics for systems which have been developed by means of object-oriented methods.
- our own proposals, based on the experience gained in developing object-oriented software during the last years.

The second step consists in evaluating the candidate measures. In order to do so, we shall apply the proposed measures to a set of systems which have been developed through object-oriented technology. The results will be compared with the qualitative evaluation of the systems we have measured. The collection of systems we shall be using is a set created by the students of Object-Oriented Software Engineering postgraduate courses, which are run by the Computer Languages and Systems Department of the Polytechnic University of Catalonia |Peralta,

1994b|. The systems'qualitative evaluation is the one given by the different lecturers at the postgraduate courses. Although we are aware that this is a subjective evaluation, as it is owed to the lecturers, we believe that the measures' evaluation is valid, since it is supported by a relatively large number of projects and by five different lecturers. It is, at least, a valid evaluation for those metrics which are not affected by the programmes' life cycles.

As far as project development and evaluation are concerned, time is a key factor in order to determine the validity of the proposed measures. This is, in particular, the matter we shall be discussing in the third part of this object-oriented metrics project. For this third step we shall apply the given set of measures all through the process of creation (analysis, design and programming) and evolution (maintenance) of a number of industrials projects. In this way, a final and conclusive evaluation of the proposed measures will be obtained.

2. What is system quality and how it is measured

Before defining each particular measure, it is advisable to explain what we understand by system or programme quality and how it can be measured.

As it is stated in |Peralta, 1994a|, "the notion of software quality is, at least implicitly, present in all the people who are involved in software engineering tasks. We would all be able to express a long list of qualities which a programme should have: that it should be "correct" !!, it should be "efficient", it should be "reliable", it should be well "documented", it should be "intelligible" and "modifiable", etc. There may be other more remote qualities which, on the other hand, may be closer to management areas: that the system should be built through the least means possible (human, technical,...), that it should be "attractive from a commercial point of view", that it should be correctly "protected", etc.".

Therefore, we should aim to define clearly the factors that may influence software quality and try to weigh them.

According to Pressman |Pressman, 1992|, there are three basic factors to be used for capturing or expressing clearly software quality:

1) The basic standard for checking or verifying software quality lies in the specification of the requirements, both functional and operating ones. Any discrepancies between what the system does and what it is expected from it is a negative factor.
2) It is important to check not only end-product quality but also the building process. Any deviation from the set standards, from the documentation, the design, etc., the product may present is also a negative factor.
3) There are a number of quality factors which are not usually made explicit (they are part of the the manufacturer's or the customer's common sense or both's). The fact that these factors are not made explicit does not mean that they should not need to be taken into account.

We should also bear in mind that not all the factors which have an effect on software quality are easily (or at least objectively) measurable or detectable. Some (such as the number of mistakes spotted, the number of lines produced per unit of time and person) are easy to detect but some others (easy maintenance or use) are not. On the other hand, product quality factors are not the same from the customer's point of view, who buys the product, as from the user's, who will have to use it or as from the technician's, who will have to maintain and modify it.

In this respect, Teasley |Teasley, 1990| points out that for the customer the economic aspects (low cost,productivity increase, efficiency) and those aspects regarding the easiness of adaptation (flexibility) or security (reliability) will be the most important. In his/her turn, the user will value most those aspects concerned with the use of the product (performance, reliability, efficiency, user-friendliness, learning and memorizing facilities). The technician who has to maintain the programme will value most, on the other hand, error absence, documentation, code legibility, design qualities.

3. Software quality factors

McCall |McCall, 19xx|, as quoted by Pressman, classifies software quality factors into three large groups according to:

1. the product's performance
2. its capability to be modified
3. its portability to other environments

The first factor would correspond to the static part of software and the other two to its dynamic part. Let us now see how we can define the qualities of these three aspects.

3.1. Quality factors connected with the product's performance

- Correction: Degree of satisfactoriness of the functional requirements.
- Reliability: Degree of reliance on the correct functioning (for instance: probability of non-failure in a given environment during a certain period of time).
- Efficiency: Number of resources (time, computational resources, etc.) required by the system to perform its function.
- Integrity: Level of control on data access or functions on the part of non-authorized people.
- Usability: Effort that the user has to make in order to learn to use, handle, prepare the inputs or interpret the outputs of the programme.

3.2. Quality factors connected with the product's capability to be modified

- Maintenance: Effort that is necessary in order to detect and eliminate a mistake.
- Flexibility: Effort that is necessary to modify the programmme.
- Testability: Effort that is needed to test the programme.

3.3. Quality factors connected with the product's portability

- Portability: Effort that is needed in order to carry the program to a plattform (soft/hard) different from that for which it was created.
- Reusability: Capability of reusing the programme (or some parts of the programme) in other applications or for use in other programmes.
- Interoperation: Effort that is necessary to join one system to another.

4. Software design quality measures

Once we have gone through the factors that have an effect on software quality, let us proceed to see which properties can be measured.

- Inheritance: Inheritance hierarchies can be measured so that we shall be able to determine the system's complexity and reusability, due to one of the innovative characteristics introduced by object-oriented development.
- Coupling: We can also measure couplings or connections between objects. The more closely the objects of an application are connected, the worse and the more difficult its maintenance will be.
- Cohesion: Lack of internal cohesion is the result of a poor design. Generally speaking, it implies a lack of granularity in the decomposition of objects.
- Polymorphism: Polymorphism introduces a new degree of complexity in object-oriented applications. Polymorphism makes applications more difficult to be followed up and tested.
- Encapsulation.
- Information Hiding.
- Volume: The greater the volume, the greater the degree of complexity is. The greater the size of a thing is, the more difficult it is to control that thing.

Of course, not every measure is indicative of a quality factor. A factor may be determined by different measures and

one single measure can be used for quantifying more than one factor. As a matter of fact, other authors call these **measures** *middle qualities* or *means* and refer to what we have called **quality factors** as *end* or *final qualities*.

The most important aspect is that, by evaluating these measures and combining the results, we can obtain more objective evaluations of software quality factors.

5. Software quality aspects specially connected with object-oriented development

One of the main objectives of the object-oriented paradigm is class reuse. Any quality measure, as far as domain modelling or analysis, design and programming are concerned, must have this basic objective as a starting-point.

With regard to the defined classes, we must take into account:

- The number of classes. A model which has a great number of classes can make the system's global comprehension more difficult, even though it facilitates, on the otherhand, the reuse of its components.
- The nomenclature being used. If we bear in mind that the library of objects can contain thousands of components, it is clear from this the fundamental part a good nomenclature plays in relation to reusability aspects.
- The number of abstract classes and their relationships and connections. In general terms, a deep structure, rather than a flat structure, is to be preferred, together with many middle classes which will facilitate reuse. Vertical relationships allow us to capture and express the differences and the common aspects between objects, whereas flat structures provide very few attributes and services which can be reused and, besides, they tend to be redundant.
- The information contained in the classes. We should avoid an excess in the number of attributes. In any case, this criterium is of a relative nature; it is not the same to deal with a basic domain class as with a view or with a controller object. The number of services will also have

to be restricted. Coad/Yourdon |Coad 1991|, recommend a maximum number of seven visible services for the object to have. The couplings between objects will also have to be limited. As a general rule, we should favour simplicity of objects and their relationships.

In regard to services, a number of criteria can be defined:

- Coherent, small and consistent services. In general,we should try to make each service fulfil one single function. The volume of services should be small. Coad/Yourdon obtain, for their ObjectTool, an average volume of 5.5 lines of code per service. The services must be consistent between the different objects, as far as performance, nomenclature and signature aspects are concerned.
- The formalities or protocol observed for communication between objects must be as simple as possible. Code/Yourdon, for example, recommend to limit the number of parameters to three.

As regards the model's general organization, a number of criteria can also be suggested:

- We should favour the use of a consistent nomenclature between classes, as well as the definition of general services which are common to all classes. The use of naming standards for class attributes and services is an effective technique.
- The number of subsystems which can be detected among classes is also important. A system which has many subsystems and few isolated classes tends to be better for encapsuling the functionalities of an application.
- We should as well favour framework creation. A framework is a group of classes which provide skeletons for applications' families, either connected to a specific domain or to similar functionalities.

6. Notation

On defining metrics, the main worry has always been that of choosing a notation which should enable us to define the measures in an accurate way, thus preventing wrong interpretations from happening. The conditions this notation should observe are as follows:

- to represent univocally the systems of objects.
- to be independent of any of the many design methodologies which support the object-oriented paradigm.
- to be based on the basic principles of the paradigm: class encapsulation, information hiding, inheritance, polymorphism, object composition, message communication between objects,...

The fulfilment of these three requirements ensures that metrics' definition according to this notation will be correct and free of ambiguity.

We shall represent the system of objects named S as the group or set of classes named $C_1,\ldots,C_k$, which describe S:

$$S = \{ C_1,\ldots,C_k \}$$

where each of the system's classes is defined by giving its group of attributes and methods:

$$C_i = \{ A_1 ,\ldots,A_m , M_1 ,\ldots,M_n \}$$

In a system of objects, classes have basically three kinds of relations:

- Inheritance
- Aggregation
- Use

In order to represent the connections between objects we have opted for a matrix representation, which will allow us to state in a clear and unambiguous way which object is connected with which other object.

Inheritance

We shall define the inheritance matrix $H = (h_{ij})$ in such a way that if $h_{ij} = 1$, then class C_j will inherit from class C_i. If $h_{ij} = 0$, then C_j does not inherit from C_i. Due to the inherent characteristics of the inheritance relationship, matrix $H = (h_{ij})$ has the following properties:

- $\forall i \quad h_{ii} = 0$, a class cannot inherit from itself.
- $\forall i<j \quad h_{ij} = 0$, a class cannot inherit from one of its descendents. There can be no cyclic inheritance (ie.: the father cannot inherit from the son).

Aggregation

We shall define the aggregation matrix $A = (a_{ij})$ in such a way that if $a_{ij} = 1$, then class C_i aggregates to C_j. Since an object can aggregate objects of the same class to which it belongs, then a_{ij} can be other than zero (ie, 1). Moreover, in this case a_{ij} can have value 1 even though $i<j$, because the aggregation allows cycles in its structure (a part can contain the whole).

Coupling and use

In order to represent the use that an object can make of another object, we shall define the coupling matrix $C = (c_{ij})$ in such a way that if $c_{ij} = 1$, then class C_i uses class C_j. If $c_{ij} = 0$, then C_i does not use class C_j. Now let us consider that class A uses class B if A uses at least one method of B. We could also consider that two classes are coupled when they use each other reciprocally. Classes A and B are coupled if A uses B and B uses A. Therefore, C_i and C_j are coupled if $c_{ij} = c_{ji} = 1$.

7. Measures

The measures proposed in this paper fall under the following categories:

- measures related to inheritance hierarchy,
- measures for evaluating couplings between objects,
- measures for couplings between classes due to their inheritance relationship,

- measures regarding the flow complexity,
- measures for class internal coherence and
- measures related to the application's size or volume.

For each proposed measure we provide a detailed description, a definition (the mathematical expression of the description), a justification (the reason why we consider that it is worth evaluating this measure), observations (diverse comments) and finally the references from where we can draw complementary information about this measure.

7.1. Inheritance hierarchy

NOC Number Of Children

Description: It is the number of children a class has.

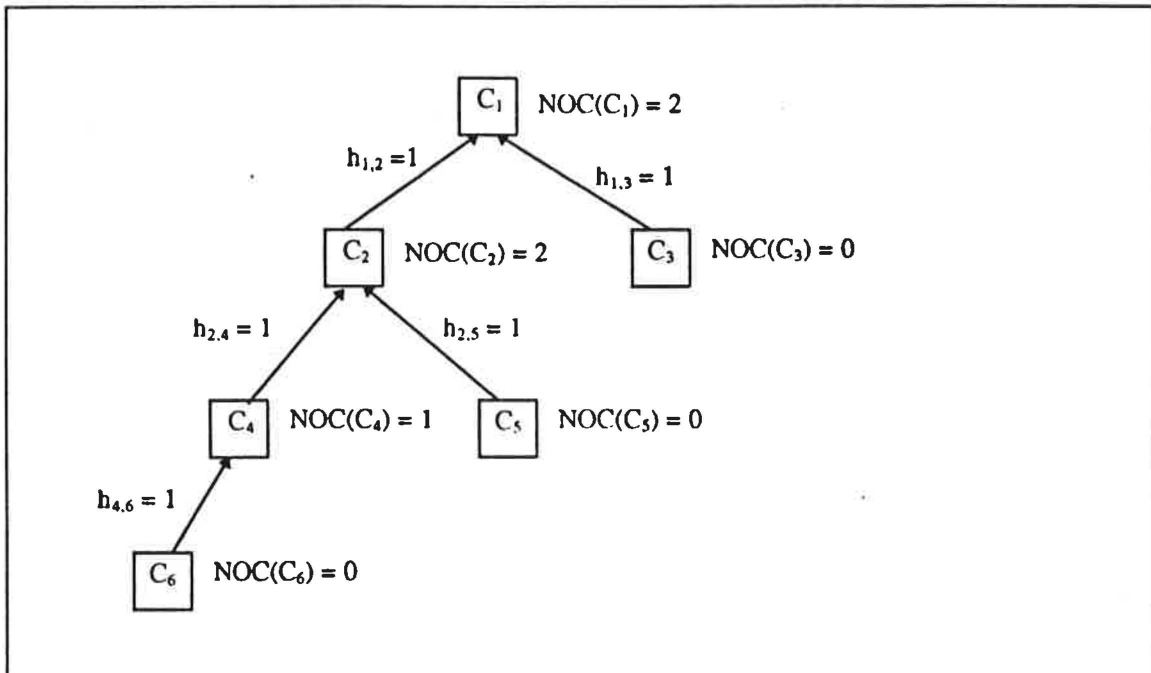
Definition: Assuming $S = \{C_1, \ldots, C_k\}$ is the system made up of classes $C_1, \ldots, C_k$, and $C_i = \{A_1, \ldots, A_m , M_1, \ldots, M_n\}$ is the class C_i whose responsibilities are the attributes $A_1, \ldots, A_m$ and the methods $M_1, \ldots, M_n$. Being $H = (h_{ij})$ the inheritance matrix between classes, then:

$$NOC(C_i) = \sum_j h_{ij}$$

Justification: It reflects in terms of number how many classes are directly affected via inheritance. It is a measure of the level of internal reuse. It also provides a quantitative value about the system's complexity.

Observations: The greater the Number Of Children (NOC) is, the more the parent's properties have been reused. On the other hand, if this index is very high, it could be indicative of a bad use of the generalization/especialization relation, that is to say, that there has been an inefficient especialization, instead of having used more suitable responsibilities from the parent or the subclasses.

Figure:



References: |Chidamber, 1991|

NOD Number Of Descendents

Description: It is the number of descendent classes a class has.

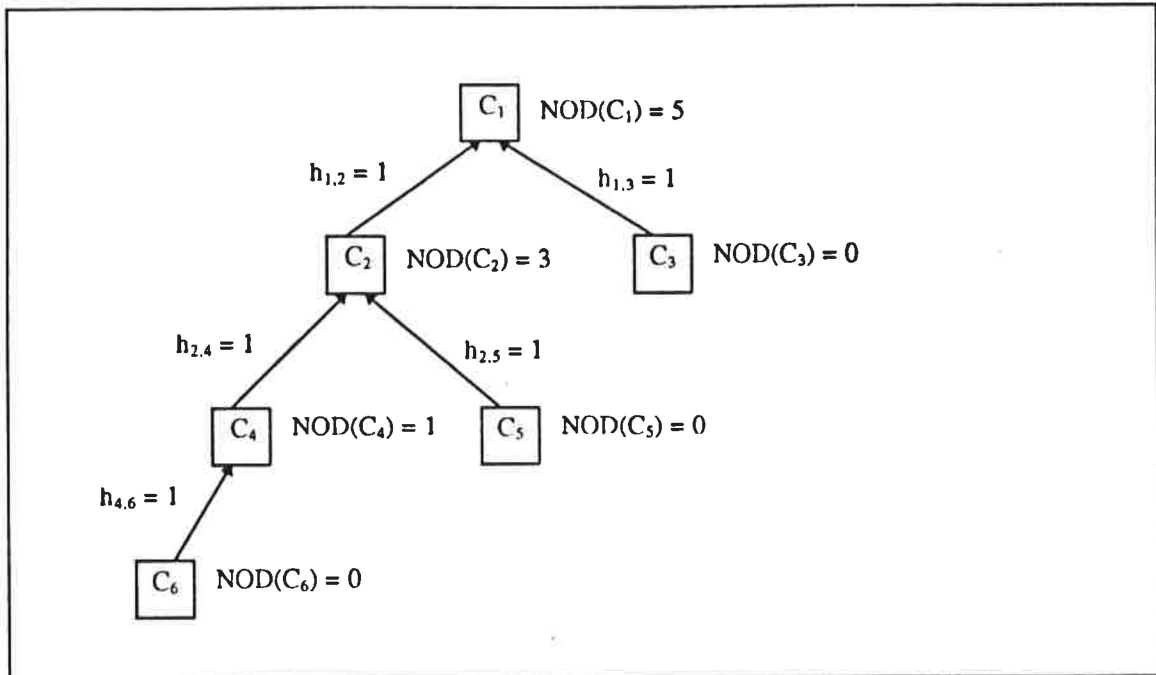
Definition: Assuming S = { $C_1, \ldots, C_k$ } is the class system earlier described, then:

$$NOD(C_i) = NOC(C_i) + \sum_j h_{ij}\ NOD(C_i)$$

Justification: It reflects in terms of number how many classes are affected via inheritance. It is another measure of the level of internal reuse and of the system's complexity.

Observations: Any modification carried out on a class affects both its children and any of its other descendents. Therefore, if a class has a high NOC index, then it is implied that any modification being carried out on the class affects a great number of classes and that the maintenance will be more delicate and complex.

Figure:



NOA Number Of Ascendents

Description: It is the number of ascendents a class has.

Definition: Let us assume that $S = \{ C_1, \ldots, C_k \}$ is the system earlier described.

If we define the NOP Number Of Parents as the number of parents a class has

$$NOP(C_i) = \sum_j h_{ji}$$

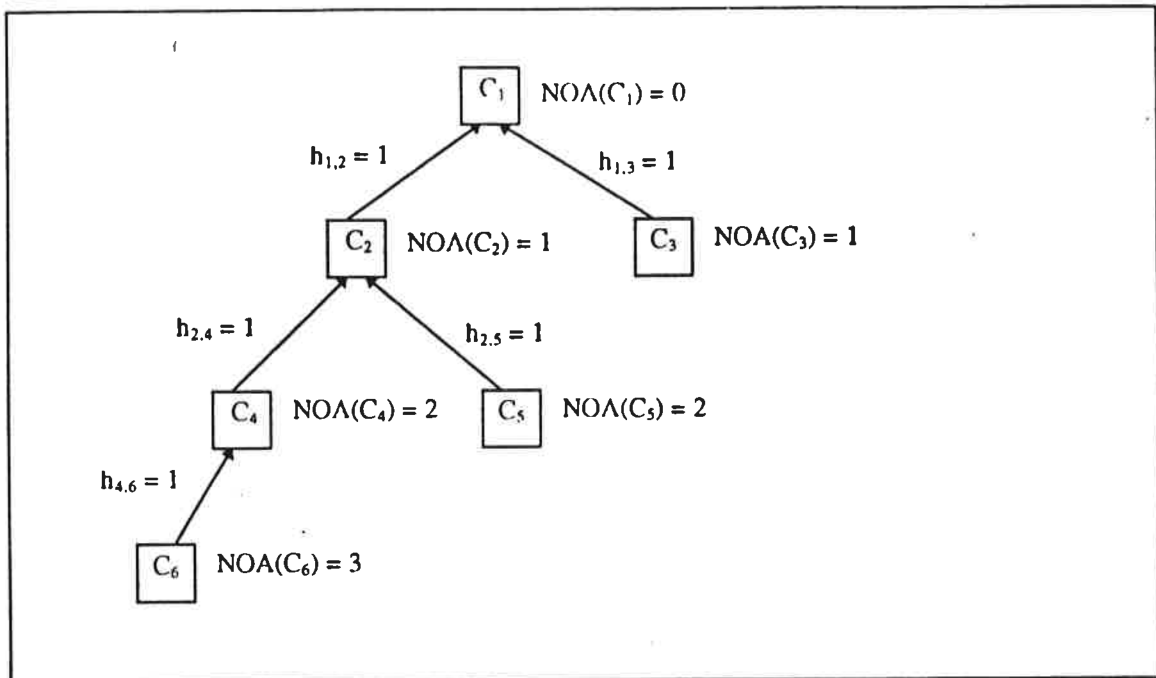
Then:

$$NOA(C_i) = NOP(C_i) + \sum_j h_{ji} \; NOA(C_i)$$

Justification: It reflects numerically the number of classes which affect this class via inheritance. Any modification carried out on any of its ascendents will affect the class itself.

Observations: NOA and DIT coincide in the case of single inheritance.

Figure:



References: NOA is equivalent to DIT as it is defined in |Chidamber, 1991|. In the case of single inheritance, NOA is equivalent to DIT as it is defined in |Li, 1993|.

DIT Depth Inheritance Tree

Description: It is the class's average depth within the inheritance hierarchy.

Definition: Let us assume that $S = \{ C_1, \ldots, C_k \}$ is the system earlier described. We shall define PLC Path Length between Classes as the length of the inheritance path which links two classes (notice that, in the case of multiple inheritance, there can be more than one inheritance path between two classes):

$$PLC_{ij}^{path=ab\ldots zy} = \left(h_{ia} + h_{ab} + \ldots + h_{yz} + h_{zj}\right) \cdot \left(h_{ia} \cdot h_{ab} \cdot \ldots \cdot h_{yz} \cdot h_{zj}\right)$$
$$= \left(\sum h_{ij}\right) \cdot \left(\prod h_{ij}\right)$$

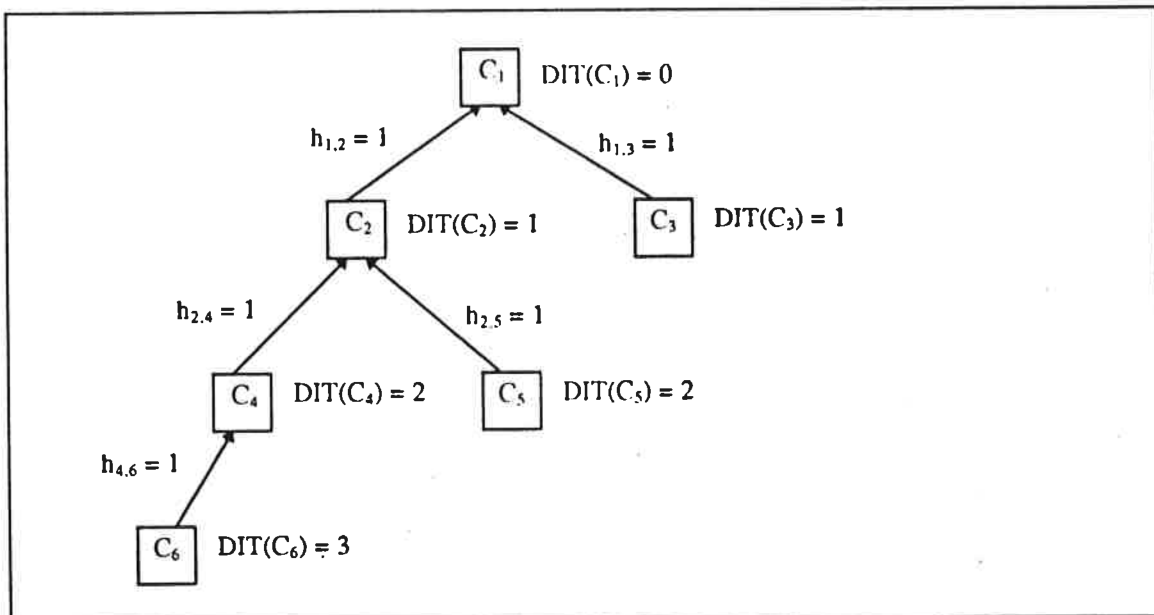
Then:

$$DIT(C_j) = \overline{LPC_{ij}^{path}} = \frac{\sum_{path} LPC_{ij}^{path}}{\sum_{path} 1}$$

Justification: It reflects numerically the number of classes which affect this class via inheritance. A deep level reflects a high index of reuse.

Observations: The DIT, as calculated in terms of the class's depth within the inheritance hierarchy, is valid when the hierarchy is a tree, in other words, when there is single inheritance. In the case of multiple inheritance, we are faced with ambiguity as regards to those classes which inherit from more than one branch of the inheritance hierarchy graph. In such cases, we must calculate the class's DIT interms of the average number of DIT's of each of the tree's branches.

Figure:



References: |Chidamber, 1991|, |Li, 1993|, DIT is HNL Hierarchy Nesting Level in |Lorenz, 1994|

7.2. Coupling

CBO1 Coupling Between Objects

Description: It is the number of classes a class is coupled with.

Definition: Assuming $S = \{ C_1, \ldots, C_k \}$ is the system made up of $C_1, \ldots, C_k$ and $C = (c_{ij})$ is the class coupling matrix, then:

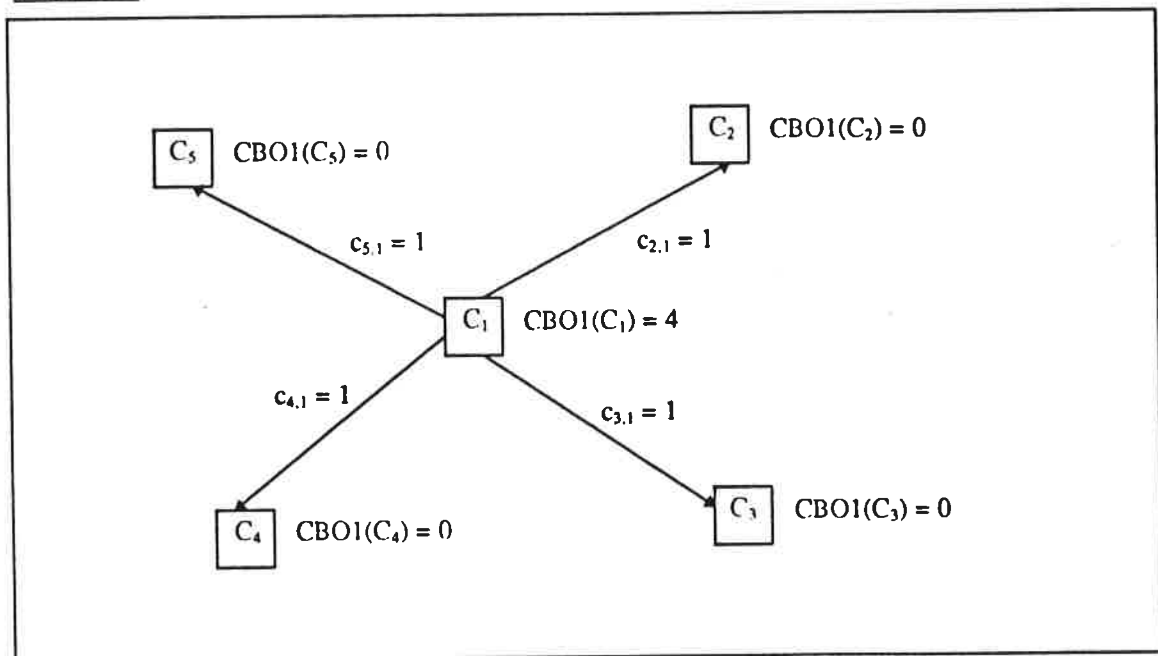
$$CBO1(C_i) = \sum_j c_{ji}$$

Justification: Coupling between classes is a direct consequence of the degree of dependence between classes. A system made up of classes which have a high index of couplings has components which restrict the system's reusability. High coupling index values indicate a high degree of dependence between classes and restrict the reusability of the system's classes and components.

Moreover, a high degree of dependence implies greater resistance to change, in detriment of modifiability and maintenance.

Observations: See Section ***6. Notation*** for a detailed definition of use and coupling between classes. Taking this measure as a starting-point, then when we refer to coupling we mean both the fact that a class uses another class and the fact that the two classes use each other reciprocally.

Figure:



References: |Li, 1993|, |Chidamber, 1991|, CCP Class Coupling in |Lorenz, 1994|

CBO1bis Coupling Between Objects

Description: It is the rate per one of the number of classes a class is coupled with, in relation to the total number of classes a class could be coupled with within the system.

Definition: Assuming S = { $C_1, ..., C_k$ } is the system earlier described, then:

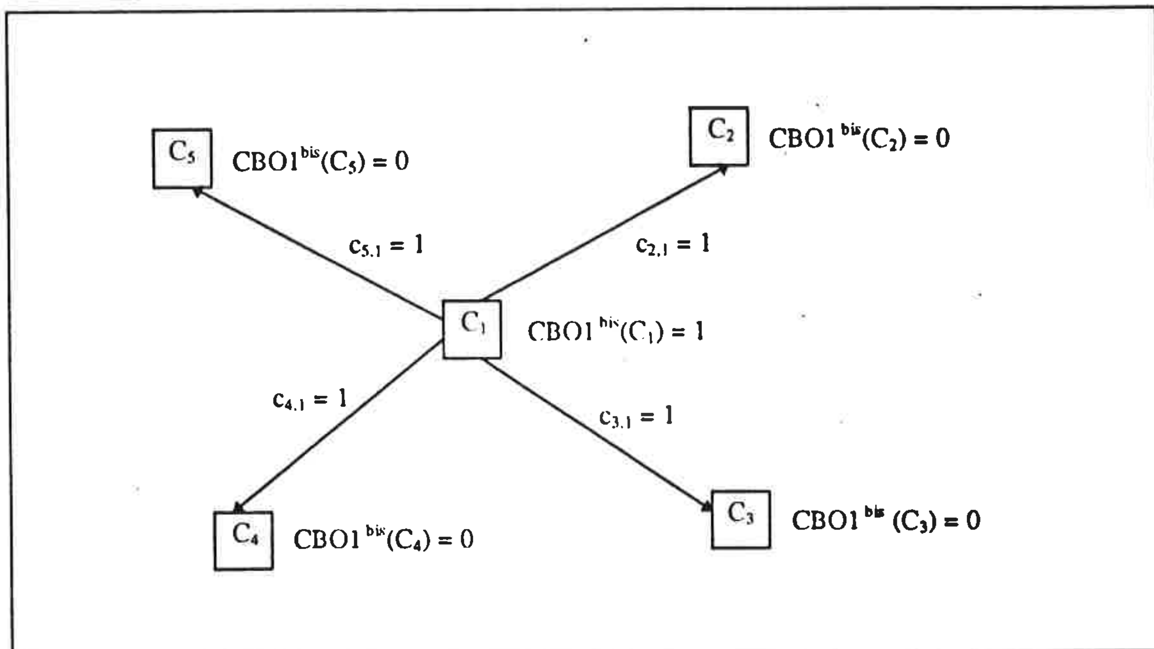
$$CBO1^{bis}(C_i) = \frac{\sum_j c_{ji}}{k-1}$$

where k is the total number of classes in the system.

Justification: For one single class, CBO1 tells us how many classes the class is coupled with. However, it does not say whether the coupling is small or large, as far as the total volume or size of the system is concerned. For this reason, we consider it necessary to normalize CBO1 in relation to the total number of classes.

Observations: CBO1 is an absolute measure. CBO1bis is CBO1's standardized version by the total number of classes. CBO1bis $\in |0,1|$. Assuming that a class is not coupled with it self ($\forall i$, $c_{ij} = 0$), we must bear this fact in mind when normalizing in relation to the total number of classes a class can be coupled with. Thus, it is necessary to normalize in relation to the total number of classes contained in the system, with the exception of one class (ie, the class itself in question).

Figure:



CBO1tris Coupling Between Objects

Description: It is the static complexity of the coupling graph.

Definition: The coupling graph is calculated from the class diagram, by associating one vertice of the graph to each class and by drawing, for each coupling between classes, an edge between the two corresponding vertices. Thus, the S system described in CBO1 can be represented by means of a graph consisting of $n = K$ vertices and $e = \sum_i \sum_j c_{ij}$ edges, where c_{ij} are the coeficients of the coupling matrix $C = (c_{ij})$.

Thus, the static complexity of the connected graph can be determined by the cyclomatic number:

$$CBO1^{tris} = v(G) = e - n + 2$$

which, being a closely connected graph, is equal to the maximum number of independent lineal paths.

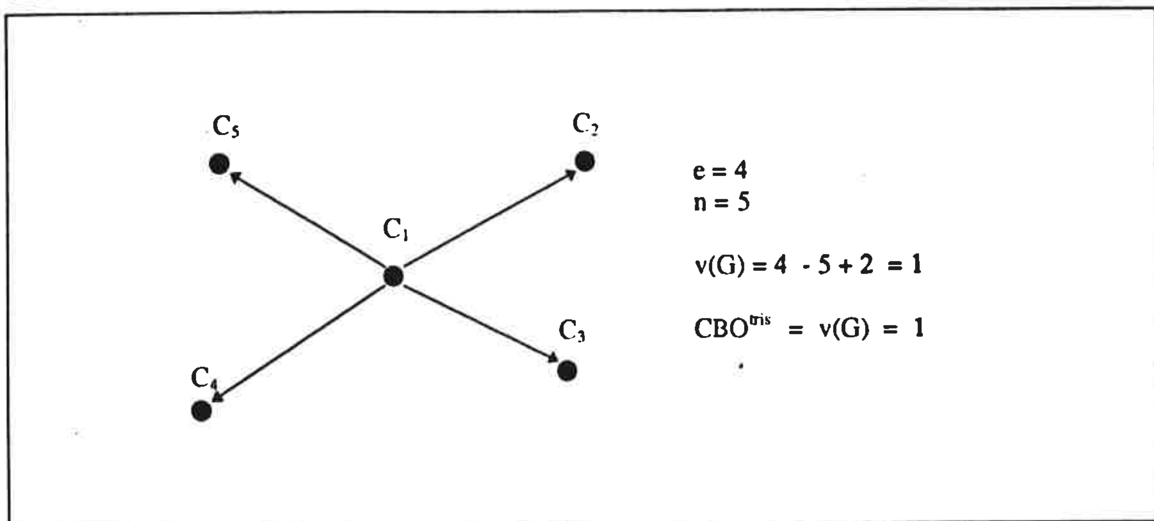
Justification: Since it is possible to associate a graph, the coupling graph, to the design of an object-oriented system in order to represent couplings between objects, then this is a valid option for evaluating coupling complexity through the traditional complexity measures of a graph, just in the same way as McCabe did with the flow graph for a module.

Observations: The higher the value of v(G) is, the more complex the system is. Assuming a fixed number of classes (vertices), then the highest v(G) is the coupling between classes.

Let us notice that the coupling graph is directed. If class A uses class B, we shall have an edge from A directed to B. If B uses A, we shall have another edge from B towards A, that is to say, A and B are coupled. Schematically it would be as follows:

Notice also that all the system's classes take part in the coupling graph, since there can be no class that is not being used by some other.

Figure:



References: |McCabe, 1976|, |Berge, 1973|

CBO2 Coupling Between Objects

Description: It is the number of methods used from the class couplings.

Definition: Assuming $S = \{ C_1, \ldots, C_k \}$ is the system made up of classes $C_1, \ldots, C_k$, in such a way that $C_i = \{A_1, \ldots, A_m, M_1, \ldots, M_n\}$ is the class named C_i which is responsible for attributes $A_1, \ldots, A_m$ and for methods $M_1, \ldots, M_n$, then we shall define $M = (M^{ij}_{rs})$ as the use matrix for the methods belonging to class C_i which are used by C_j. If $M^{ij}_{rs} = 1$, then method M_r belonging to class C_i is used by method M_s belonging to class C_j. Therefore:

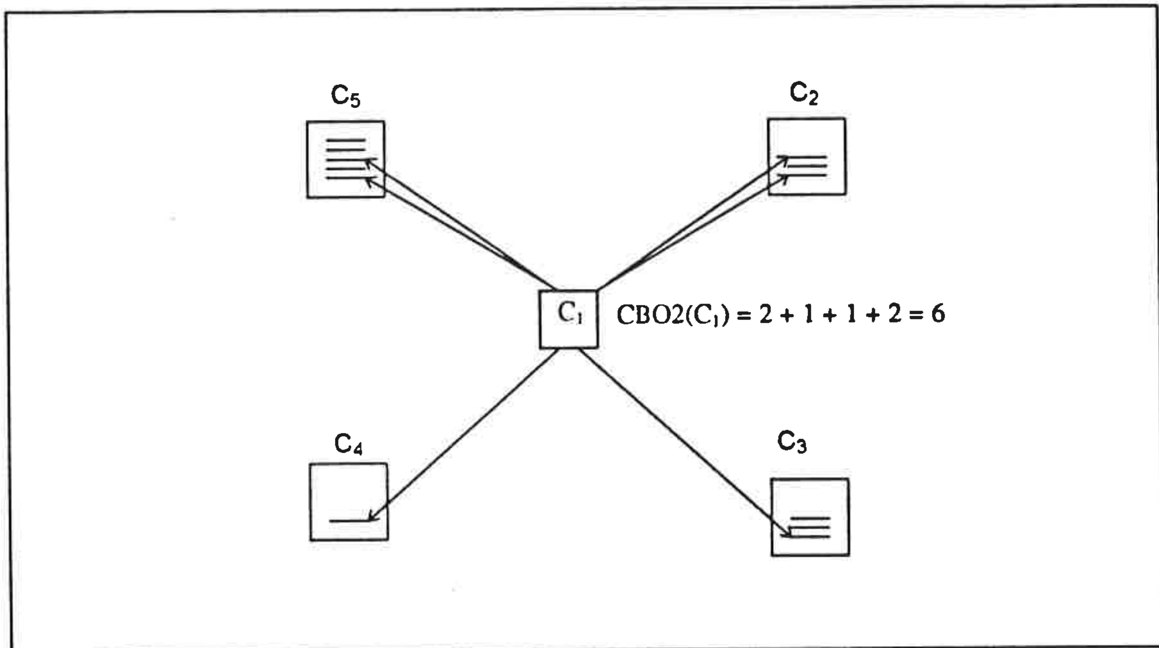
$$CBO2(C_j) = \sum_i \sum_{r,s} M^{ij}_{rs}$$

Justification: Class coupling is a direct consequence of the system's complexity.

This index is a sophisticated version of the CBO1 index. Through CBO1, we are able to know how many classes a class interacts with but we are not able to tell how strong this interaction is. It is not the same having a class coupled with another one which only uses one method from this class as having one which uses all the methods the class makes available to it.

Observations: As certain authors have stated, we believe it is necessary to go deeply into the evaluation of class couplings. For this reason, we have considered it necessary to examine class coupling at a more microscopical level, separating the class into methods in order to obtain more accurate coupling measures.

Figure:



CBO3 Coupling Between Objects

Description: It is the average number of methods used in relation to the number of class couplings.

Definition: Assuming S = { $C_1, \ldots, C_k$ } is the system defined in CBO2, then:

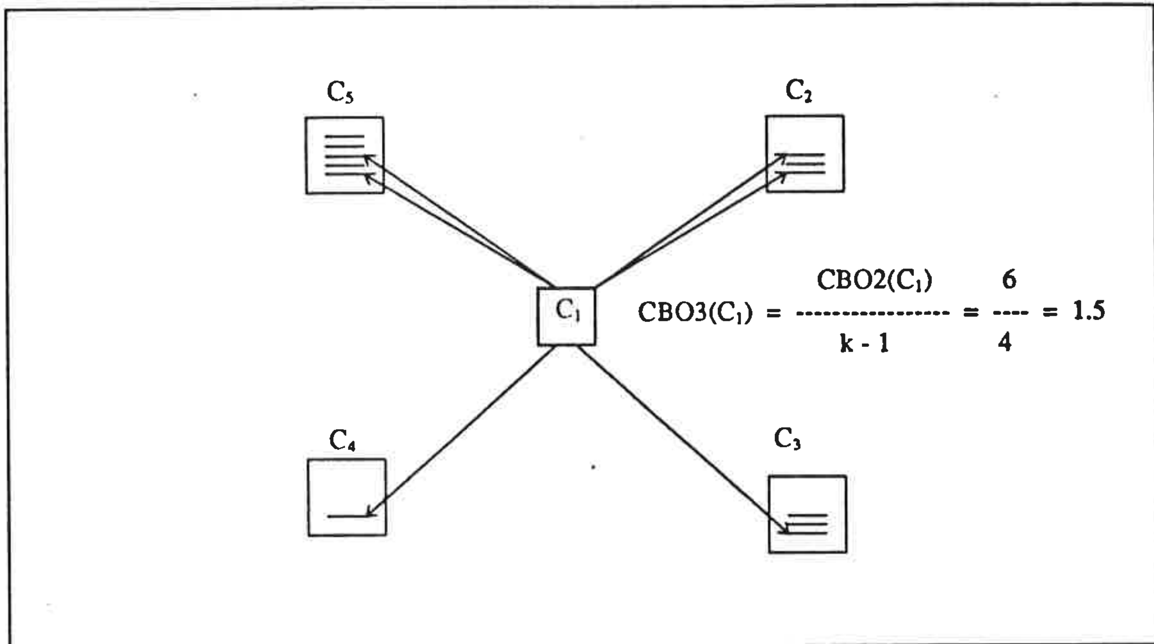
$$CBO3(C_i) = \frac{\sum_i \sum_{r,s} M^{ij}_{rs}}{k-1} = \frac{CBO2(C_i)}{k}$$

Justification: It is the average number of methods used in each class coupling. Thus, the justification and implications of this index are the same as those in CBO2.

Observations: This index is CBO2's normalized version in relation to the number of class couplings. Notice that its value can be higher than 1. If its value is 1, that means that the class in question uses an average number of methods from

each class it is coupled with. Therefore, in this case the coupling could be regarded as low. We shall normalize by k-1 for the same reasons as in CBO1bis.

Figure:



CBO4 Coupling Between Objects

Description: It is the rate per one of methods used in relation to the total number of methods there is access to.

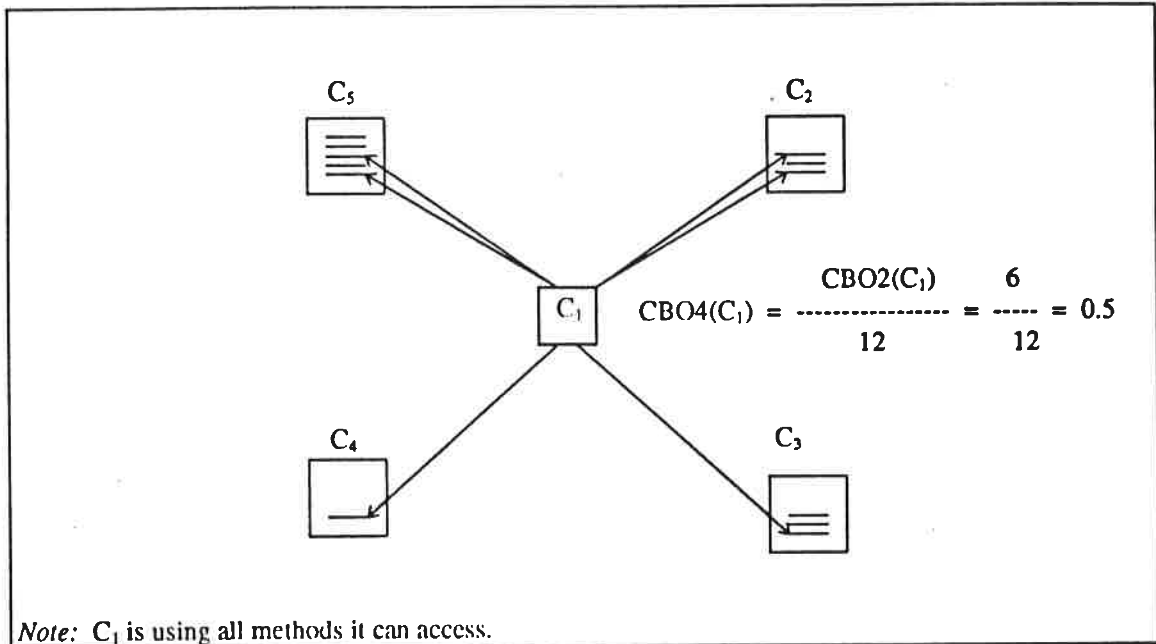
Definition: Assuming $S = \{ C_1, \ldots, C_k \}$ is the system defined in CBO2, then:

$$CBO4(C_j) = \frac{\sum_i \sum_{r,s} M_{rs}^{ij}}{\sum_i \sum_r 1}$$

Justification: It is CBO2 having been normalized in relation to the number of methods from class couplings C_j has access to. Therefore, the justification and implications of this index are the same as those in CBO2.

Observations: This index is between 0 and 1, that is, CBO4 ∈ |0,1|. If its value is 1, then it is implied that class C_j is using all the methods from the classes it is coupled with.

Figure:



Note: C_1 is using all methods it can access.

CBO5 Coupling Between Objects

Description: It is the normalized number of methods used from one class in relation to the total number of public methods from this class and averaged out by all the classes it is coupled with.

Definition: Assuming S = { $C_1, \ldots, C_k$ } is the system defined in CBO2, then:

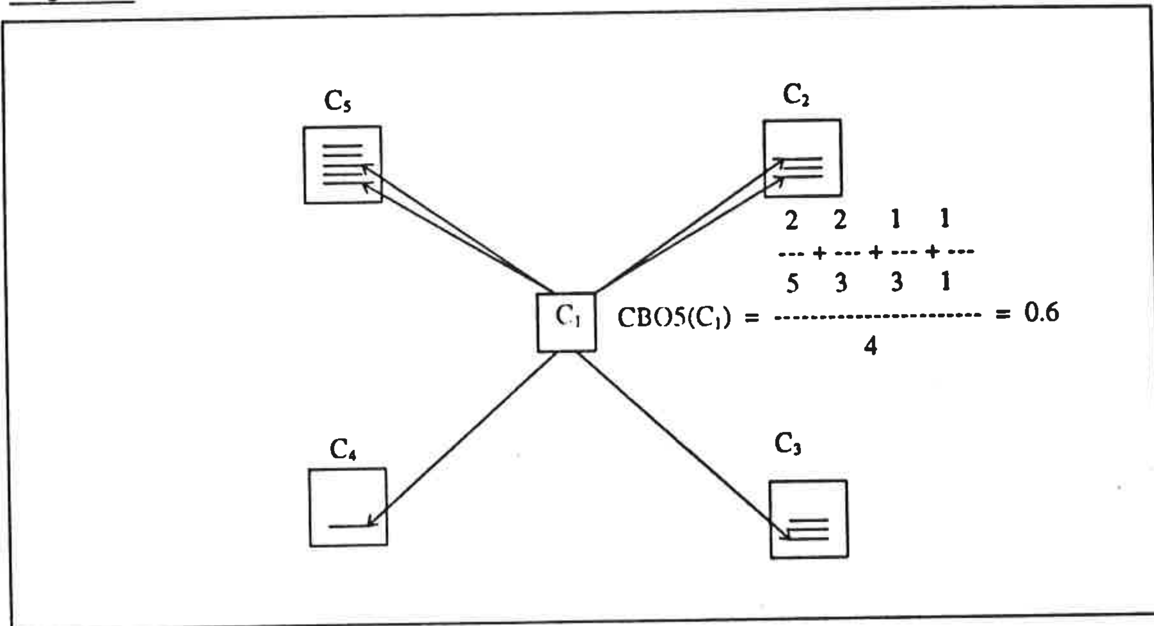
$$CBO5(C_j) = \frac{\dfrac{\sum_i \sum_{r,s} M_{rs}^{ij}}{\sum M_{rs}^{ij}}}{\sum_i 1}$$

Justification: CBO5 is the same as CBO2, differing in the fact that CBO5 has been normalized by the number of methods per class and averaged out by each class. Therefore, the justification provided for CBO2 is also valid for this measure.

Observations: This index is between 0 and 1, that is, $CBO5 \in [0,1]$. Even though this index has been normalized by the number of methods and classes, just in the same manner as CBO4 has,

the quantitative values of measure are different for each of the two indexes.

Figure:



CBO6.1 Coupling Between Objects

Description: It is the number of classes which used at least one method from one given class.

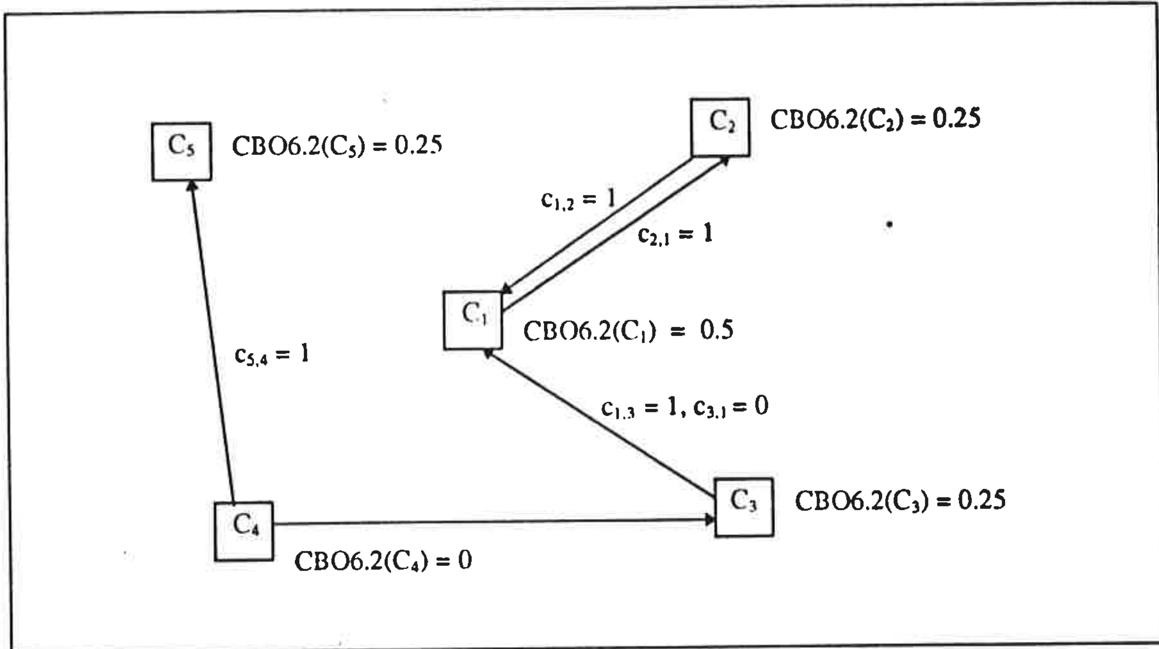
Definition: Assuming $S = \{ C_1, ..., C_k \}$ is the system made up of classes $C_1, ..., C_k$, and $C = (c_{ij})$ is the class coupling matrix, then:

$$CBO6.1(C_i) = \sum_j c_{ij}$$

Justification: If a class has this index high, this means that this class is being used by many other classes, thus being a class which is highly liable to modification or whose modification will affect many classes. A high index may also indicate a high degree of reusability.

Observations: So far we have been examining class coupling in relation to the classes used by a given class. Now we are shifting towards the use the system makes of the class under consideration. When we refer to the classes that use a class method, we are talking about either the classes that use this class or the classes that are coupled with it.

Figure:



7.3. Coupling and inheritance

ICBO1 Inheritance Coupling Between Object

Description: It is the number of added attributes.

Definition: Assuming $S = \{ C_1, ..., C_k \}$ is the system made up by classes $C_1, ..., C_k$, in such a way that $C_i = \{A_1, ..., A_m, M_1, ..., M_n\}$ is the C_i class whose responsibilities are the attributes $A_1, ..., A_m$ and the methods $M_1, ..., M_n$, whereby $A_1, ..., A_m$ are all the attributes belonging to C_i, including both the class's own attributes and the ones inherited from any of its ascendents. Thus:

$$ICBO1(C_i) = \sum_{\substack{added \\ atributes}} 1$$

Justification: It gives an idea of the level of reuse within an inheritance hierarchy and also of the level of class specialization.

Observations: If within a class few attributes are declared, then the ICBO1 index will be low, which means that basically all inherited attributes are being reused, and therefore, there

is a high degree of reuse. On the other hand, this also indicates that the class is very little specialized in relation to its parents and so the class will have a high degree of reuse, whereas if the class has a high ICBO1 index, it will be a highly specialized class and probably very specific for the application being developed and so, hardly reusable.

ICBO2 Inheritance Coupling Between Objects

Description: It is the number of added methods.

Definition: Let us assume $S = \{ C_1,...,C_k \}$ is the system earlier described and class $C_i = \{ A_1,...,A_m , M_1,...,M_n \}$ is responsible for attributes $A_1,...,A_m$ and methods $M_1,...,M_n$, whereby $M_1,...,M_n$ are all the methods belonging to C_i. All the methods are both all the methods inherited from any of the ascendents and also the methods which have been redefined or extended, plus the own new methods which have been declared and defined within class C_i itself. Then:

$$ICBO2(C_i) = \sum_{\substack{added \\ methods}} 1$$

Justification: As the class is responsible for the methods, as well as for the attributes, the justification for using this measure is supported by the justification provided in the previous section ICBO1.

Observations: The observations made in the previous case are also applicable to this section.

ICBO3 Inheritance Coupling Between Objects

Description: It is the number of redefined methods.

Definition: Let us assume that $S = \{ C_1,...,C_k \}$ is the system earlier described and class $C_i = \{A_1,...,A_m , M_1,...,M_n\}$ is responsible for attributes $A_1,...,A_m$ and methods $M_1,...,M_n$. We understand by redefined methods those methods which have been inherited and whose contents have either been modified or extended. Then:

$$ICBO3(C_i) = \sum_{\substack{redefined \\ methodss}} 1$$

Justification: This measure is a direct consequence, depending on the case, of the use that is made of polymorphism. All those methods which have been inherited and which are modified all through the inheritance hierarchy are capable of being used polymorphically. This condition is necessary but not sufficient. In order to be so, we should provide a detailed declaration about the methods so that a dynamic link is obtained.

Observations: The higher this index is, the more available polymorphism will be to the application, and thus, the better the design is bound to be, although the system's complexity and debugging will be at the same time increased. Despite this, maintenance will prove easier and simpler, in the sense that it will be easier to add new classes without any need to modify other components.

ICBO4 Inheritance Coupling Between Objects

Description: It is the rate per one of added attributes in relation to the total number of class attributes.

Definition: Assuming S = { $C_1, \ldots, C_k$ } is the system made up of classes $C_1, \ldots, C_k$, in such a way that C_i = { $A_1, \ldots, A_m$, $M_1, \ldots, M_n$ } is the C_i class which is responsible for attributes $A_1, \ldots, A_m$ and methods $M_1, \ldots, M_n$, and whereby $A_1, \ldots, A_m$ are all the attributes belonging to C_i, including both the class's own attributes and also those inherited from its ascendents, then:

$$ICBO4(C_i) = \frac{\sum_{\substack{added \\ atributes}} 1}{\sum_{j=1}^{m} 1} = \frac{ICBO1(C_i)}{m}$$

Justification: Since we are simply normalizing CBO1, then the justification provided in that section is also valid for this case.

Observations: ICBO4 is the same as ICBO1 having been normalized in relation to the total number of attributes. Thus, ICBO4 $\in$ |0,1|.

ICBO5 Inheritance Coupling Between Objects

Description: It is the rate per one of added methods in relation to the total number of class methods.

Definition: Let us assume $S = \{ C_1, \ldots, C_k \}$ is the system made up of classes $C_1, \ldots, C_k$, in such a way that $C_i = \{ A_1, \ldots, A_m, M_1, \ldots, M_n \}$ is the C_i class which is responsible for attributes $A_1, \ldots, A_m$ and for methods $M_1, \ldots, M_n$, whereby $M_1, \ldots, M_n$ are all the methods belonging to C_i. Now all the methods are both all those which have been inherited from any of the ascendents and also the redefined or extended methods, plus the own new methods which have been declared and defined within class C_i itself. Thus:

$$ICBO5(C_i) = \frac{\sum_{\substack{added\\methods}} 1}{\sum_{j=1}^{n} 1} = \frac{ICBO2(C_i)}{n}$$

Justification: Since we have simply normalized CBO2, the justification provided in that section is also valid for this case.

Observations: ICBO5 is ICBO2 having been normalized in relation to the total number of methods. $ICBO5 \in |0,1|$.

ICBO6 Inheritance Coupling Between Objects

Description: It is the rate per one of redefined methods in relation to the total number of class methods.

Definition: Assuming $S = \{C_1, \ldots, C_k\}$ is the system made up of classes $C_1, \ldots, C_k$, as it was described in previous sections, then:

$$ICBO6(C_i) = \frac{\sum_{\substack{redefined\\methods}} 1}{\sum_{j=1}^{n} 1} = \frac{ICBO3(C_i)}{n}$$

Justification: Since we have simply normalized CBO3, the justification provided in that section is also valid for this case.

Observations: ICBO6 is ICBO3 having been normalized in relation to the total number of methods. Therefore, ICB6 ∈ |0,1|.

7.4. Complexity

WMC Weighted Methods per Class

Description: It is the average sum of McCabe's static complexity indexes for all the class methods.

Definition: Let us assume that $S = \{C_1,...,C_k\}$ is the system made up of classes $C_1,...,C_k$, in such a way that $C_i = \{ A_1,...,A_m, M_1,...,M_n \}$ is the C_i class which its responsibilities are the attributes $A_1,...,A_m$ and the methods $M_1,...,M_n$. Let us also assume $c_1,...,c_n$ are the static complexity for methods $M_1,...,M_n$ respectivily. Then:

$$WMC(C_i) = \sum_{j=1}^{n} c_i$$

Justification: WMC is directly related to class complexity, since McCabe's static complexity indexes are a measure of the complexity of each class method.

Observations: The higher this index is, the more complex the class will be, and so the more difficult it will be to develop, maintain and test.

References: |Chidamber, 1991|, |McCabe, 1976|, |Berge, 1973|.

WMC1 Weighted Methods per Class

Description: It is the average sum of McCabe's static complexity indexes for all the class methods in relation to the total number of methods.

Definition: Let us assume that $S = \{ C_1,...,C_k \}$ is the system described in WMC. Therefore $c_1,...,c_n$ are the static complexity of methods $M_1,...,M_n$, thus:

$$WMC1(C_i) = \frac{\sum_{j=1}^{n} c_i}{\sum_{j=1}^{n} 1} = \frac{\sum_{j=1}^{n} c_i}{n}$$

Justification: Since we are dealing with an average sum, the most suitable justification is the one provided in the previous section for WMC.

Observations: This index is WMC having been divided by the number of methods belonging to the class.

References: |Chidamber, 1991|, |McCabe, 1976|, |Berge, 1973|

WMC2 Weighted Methods per Class

Description: It is the sum of McCabe's static complexity indexes for all the class methods, plus the methods which have been called by the class methods.

Definition: Assuming S = $\{C_1, \ldots, C_k\}$ is the system describe in WMC, then:

$$WMC2(C_j) = WMC(C_j) + \sum_i \sum_{r,s} M_{rs}^{ij} \, c_r^i$$

Justification: The criteria used for justifying WMC are also applicable to this case.

References: |McCabe, 1976|, |Berge, 1973|

WMC3 Weighted Methods per Class

Description: It is the sum of McCabe's static complexity indexes for all the class methods, plus the methods called by the class methods, and then being divided by the number of class methods.

Definition: Assuming S = $\{ C_1, \ldots, C_k \}$ is the system describe in WMC, then:

$$WMC3(C_j) = \frac{WMC2(C_j)}{n}$$

Justification: The criteria used for justifying WMC are as well applicable in this section.

Observations: WMC3 is WMC2 having been divided by the number of class methods.

References: |McCabe, 1976|, |Berge, 1973|

RFC Response For a Class

Description: It is the number of class methods plus the set or group of methods which have been called by the class methods.

Definition: Let us assume S = { $C_1,...,C_k$ } is the system made upof classes $C_1,...,C_k$, whereby C_i = { $A_1,...,A_m$, $M_1,...,M_n$ } is the C_i class whose responsibilities are the attributes $A_1,...,A_m$ and the methods $M_1,...,M_n$. We shall define the response set, named RS, as:

$RS = \{M_i\} \Upsilon_i \{R_i\}$ where $\{M_i\} = \{M_1,...,M_n\}$ are all the class methods and $\{R_i\}$ is the group of methods called by *Mi*.

Then:

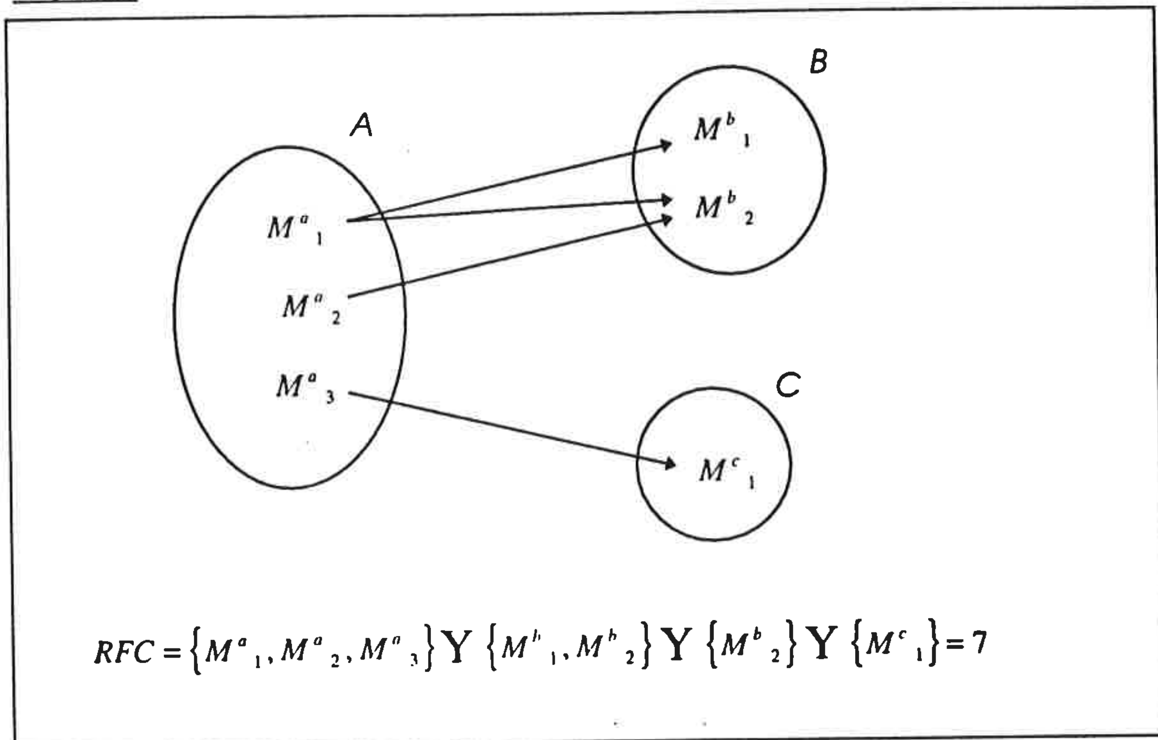
$$RFC = |RS|$$

where |x| denotes the module or number of elements contained in the response set.

Justification: The set of answers is the group of methods an object has access to. Therefore, it gives an idea of the size or volume of the object and, due to the fact that it also includes all the methods recalled from other classes,it is a measure of communication between objects, and, so a measure of coupling between objects.

Observations: The higher the value of this index is, the more expensive it will be to test, debug and maintain the class. A high index is also indicative of more communication between classes and, therefore, a higher degree of complexity.

Figure:



References: |Chidamber, 1991|

7.5. Coherence

LCOM Lack of Cohesion in Methods

Description: It is the number of disjointed sets created by the intersection of the sets of variables each class method uses.

Definition: Assuming S = { $C_1, \ldots, C_k$ } is the system made up ofclasses $C_1, \ldots, C_k$, so that C_i = { $A_1, \ldots, A_m$, $M_1, \ldots, M_n$ } is the C_i class which is responsible for attributes $A_1, \ldots, A_m$ and for methods $M_1, \ldots, M_n$, and assuming $\{I^i{}_k\}$ is the set of attributes being used by the M_i method which belongs to the C_i class, then we can define:

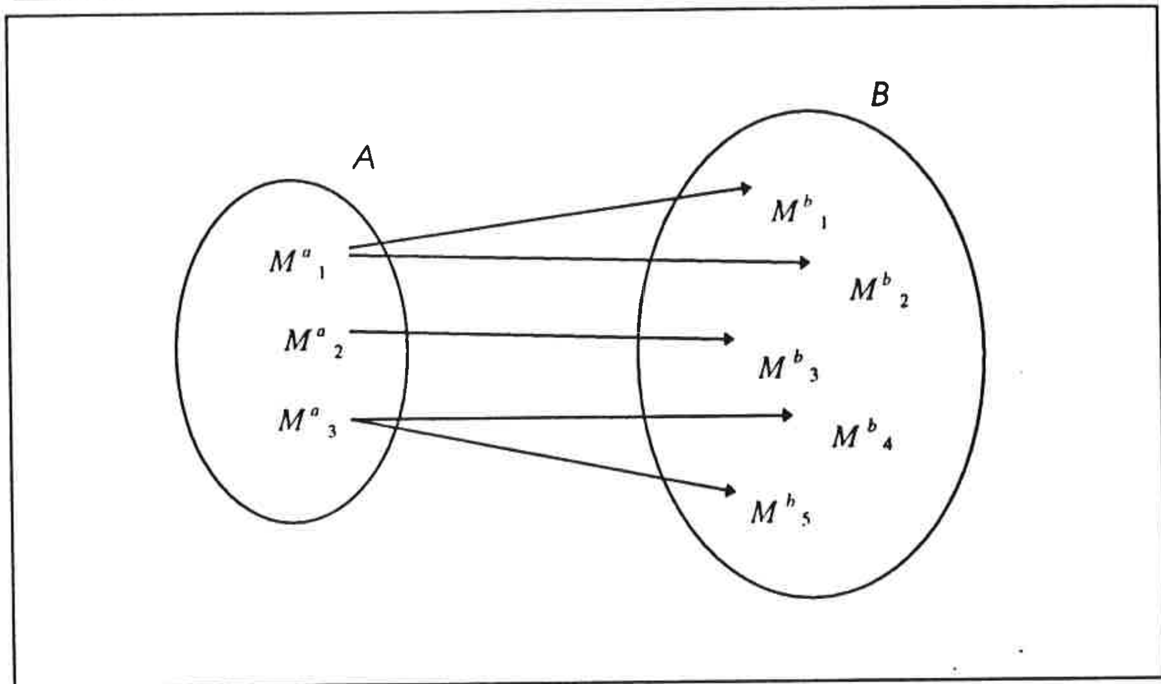
$$LCOM(C_i) = \sum_{\substack{disjoint \\ sets}} \left(\{I^i{}_k\} \cap \{I^i{}_l\} \right)$$

Justification: This is a design quality measure. If a class is lacking in internal cohesion (LCOM > 1), this probably means that the class is badly designed, as it is very likely that it undertakes independent responsibilities, which are capable of

being undertaken by as many other classes as the number of disjointed sets of responsibilities the class may contain.

Observations: The very best case of internal coherence in a class is that in which every method uses all of its attributes. In this case LCOM = 1 (Internal Cohesion).

Figure:



References: |Chidamber, 1991|

7.6. Volume

NOA Number Of Attributes

Description: It is the number of class attributes.

Definition: Assuming S = { $C_1,\dots,C_k$ } is the system made up of classes $C_1,\dots,C_k$, whereby C_i = { $A_1,\dots,A_m$, $M_1,\dots,M_n$ } is the C_i class which its attributes are $A_1,\dots,A_m$ then:

$$NOA(C_i) = \sum_{j=1}^{m} 1 = m$$

Justification: See section NOR further on.

NOM Number Of Methods

Description: It is the number of public class methods.

Definition: Assuming $S = \{C_1,...,C_k\}$ is the system made up of classes $C_1,...,C_k$, whereby $C_i = \{A_1,...,A_m , M_1,...,M_n\}$ is the C_i class which its methods are $M_1,...,M_n$, then:

$$NOM(C_i) = \sum_{j=1}^{n} 1 = n$$

Justification: See section NOR further on.

NOR Number of Responsibilities

Description: It is the number of responsibilities a class has, that is to say, the number of class attributes plus the number of class methods.

Definition: Assuming $S = \{ C_1,...,C_k \}$ is the system made up of classes $C_1,...,C_k$, so that $C_i = \{ A_1,...,A_m , M_1,...,M_n \}$ is the C_i class which is responsible for attributes $A_1,...,A_m$ and for methods $M_1,...,M_n$,. Then:

$$NOR(C_i) = NOA(C_i) + NOM(C_i)$$

Justification: The greater the number of attributes or methods a class has, the greater the number of responsibilities there will be for the class in question. Therefore, the class's volume will be larger and, if we associate size with complexity, then the more complex the class is bound to be. Thus, high NOA, NOM and NOR indexes imply that class development will be more expensive, and so will be its implementation and testing.

LCOM Lines Of Code per Method

Description: It is the number of lines of code a method has.

Definition: Let us assume $S = \{ C_1,...,C_k \}$ is the system made up of classes $C_1,...,C_k$, so that $C_i = \{ A_1,...,A_m , M_1,...,M_n \}$ is the C_i class which its methods are $M_1,...,M_n$. Assuming the M_i method, let us also assume that l_i is the number of lines the method has. Then:

$$LCOM(M_i) = l_i$$

Justification: It concerns the method's volume or size; therefore, the implications are the same as the ones described for NOR.

Observations: Notice that this measure is per method and not per class.

LOCC Lines Of Code per Class

Description: It is the average number of lines of code the class methods have.

Definition: Assuming $S = \{ C_1,...,C_k \}$ is the system made up of classes $C_1,...,C_k$, so that $C_i = \{ A_1,...,A_m , M_1,...,M_n \}$ is the C_i class which its methods are $M_1,...,M_n$,and assuming the M_i method, we shall assume l_i is the number of lines a method has. Then:

$$LOCC(C_i) = \frac{\sum_{j=1}^{n} LOCM(M_i)}{n} = \frac{\sum_{j=1}^{n} l_j}{n}$$

Justification: We calculate a class's size in relation to the number of lines of code its methods have. Then, if we follow the criteria applied for object-oriented development which says that, in adding a new class to the system, we must base our action on the notion of construction by difference, it is clear that methods' sizes have to be small.

Observations: This index must not be excessively high. If we bear in mind the values recommended by Coad/Yourdon, this index should be around 5.5 lines.

NPS Number of Parameters per Method

Description: It is the number of parameters per visible methods.

Definition: Let us assume $S = \{ C_1,...,C_k \}$ is the system previously described and class $C_i = \{ A_1,...,A_m , M_1,...,M_n \}$ its methods are $M_1,...,M_n$, whereby $M^i_j(p^i_{j1},...,p^i_{j1})$ is the M^i_j method

belonging to the C_i class which admits the parameters named $p^i_{j1},\ldots,p^i_{jl}$. Then:

$$NPS\left(M^i_j\right) = \sum_{r=1}^{l} 1 = l$$

Justification: When an object has methods which have many parameters, this implies that the client class needs to have comprehensive knowledge of the server class so as to be able to make good use of this latter class. If the method has many parameters, the internal algorithm will be more complex and, therefore, maintenance will be more difficult. An excess of parameters also goes against the criterium for reuse via black boxes.

Observations: This is a measure per method.

8. Conclusions and future research

In this paper we have introduced a collection of measures which we consider will be useful for quantifying the design quality of object-oriented systems. Obviously, even though there are available measures, we shall not be able to find a definitive answer about which is the best design for a given system; let us not forget that "not everything that counts can possibly be counted, and not everything that can be counted really counts" (Albert Einstein). Despite this, we believe that the set of measures which have been proposed in this paper is the first step towards a quantitative evaluation of systems' development, which will allow us to move on to a second stage where measures will be gathered and improved as we learn more about them.

Therefore, the next step is to evaluate the proposed measures, that is to say, to validate their utility by experimenting with them. It is in this second stage when we shall be able to draw the first conclusions about which measures can be actually useful for evaluating design quality.

These results will be obtained experimentally by applying the set of measures on to the group of systems developed by the students who have taken the postgraduate course and programme

of "Object-Oriented Software Engineering", which is run by the Computer Languages and Systems Department of the Polytechnic University of Catalonia. All of these systems have been developed through the use of object-oriented technology, and the results must allow us to break the vicious circle which revolves around the assertion that: we have no valid measures, for we do not collect data, and as we do not have data, then we do not collect measures.

Bibliography & References

|Lorenz, 1994| Lorenz, Mark and Jeff Kidd. "Object-Oriented Software Metrics. A Practical Guide", Prentice Hall ObjectOriented Series, 1994.

|Peralta, 1994a| Peralta, A.J., Rodriguez H. "Introducció a l'Enginyeria del Software: Programació Orientada a Objectes", Edicions UPC 1994.

|Peralta, 1994b| Peralta, A.J. "A Comparison of the Introduction of the Object-Oriented Paradigm in Undergraduate and Postgraduate Software Engineering", Proceedings of the TATTOO'95 Conference Leicester.

|Li, 1993| Li, Wei and Sallie Henry. "Maintenance Metrics For The Object Oriented Paradigm", Proceedings of First International Software Metrics Symposium, May 1993, pp. 52-60.

|Karunanithi, 1993| Karunanithi, Santhi and James M. Bieman. "Candidate Reuse Metrics For Object Oriented and Ada Software", Proceedings of First International Software Metrics Symposium, May 1993, pp. 120-128.

|Barnes, 1993| Barnes, G. Michael and Bradley R. Swim. "Inheriting Software Metrics", Journal of Object Oriented Programming JOOP, Nov-Dec 1993, pp. 27-34.

|Li, 1992| Li, Wei. "Applying Software Maintenance Metrics In The Object-Oriented Software Development Life Cycle", Ph.D. Dissertation, 1992.

|Roberts, 1992| Roberts, Teri. "Metrics For Object-Oriented Software Development", First OOPSLA Workshop on Metrics. Addendum to the Proceedings OF OOPSLA'92.

|Pressman, 1992| Pressman, Roger S. "Software Engineering (3rd edition)", McGraw Hill 1992.

|Chidamber, 1991| Chidamber, Shyam R. and Chris F. Kemerer. "Towards a Metrics Suite For Object-Oriented Design", Proceedings: OOPSLA'91, July 1991, pp. 197-211.

|Coad 1991| Coad, Peter and Edward Yourdon. "Object-Oriented Analysis" (Second Edition), Yourdon Press Computing Series, Prentice-Hall 1991.

|Teasley, 1990| Teasley Mynatt, Barbee. "Software Engineering with Student Project Guidance", Prentice-Hall 1990.

|Weyuker, 1988| Weyuker, E. "Evaluating Software Complexity Measures", IEEE Transactions on Software Engineering, Vol.14, No. 9, September 1988, pp. 1357-1365.

|Kearney, 1986| Joseph K. Kearney et al. "Software Complexity Measurement", Communications of ACM, Nov. 1986,Vol. 29, No. 11, pp. 1044-1050.

|Gong, 1985| Gong, H. and Schmidt, M. "A Complexity Measure Based on Selection and Nesting", Performance Evaluation Review, Vol. 13, No. 1, June 1985, pp. 14-19.

|McCabe, 1976| McCabe, Thomas J. "A Complexity Measure",IEEE Transactions on Software Engineering, Vol. SE-2, No. 4,December 1976, pp. 308-320.

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya

Recent Research Reports

LSI-94-45-R "On RNC approximate counting", Josep Díaz, María J. Serna, and Paul Spirakis.

LSI-94-46-R "Prototipatge semàntic d'un model conceptual deductiu" (written in Catalan), C. Farré and M.R. Sancho.

LSI-94-47-R "Generació i simplificació automàtica del Model d'Esdenivents Interns corresponent a un model conceptual deductiu" (written in Catalan), C. Farré and M.R. Sancho.

LSI-94-48-R "B-Skip trees, a data structure between skip lists and B-trees", Joaquim Gabarró and Xavier Messeguer.

LSI-94-49-R "A posteriori knowledge: from ambiguous knowledge or undefined information to knowledge", Matías Alvarado.

LSI-94-50-R "An approach to the control of completeness based on metaknowledge", Jordi Alvarez and Núria Castell.

LSI-94-51-R "On finding the number of graph automorphisms", Richard Chang, William Gasarch, and Jacobo Torán.

LSI-95-1-R "Octree simplification of polyhedral solids", Dolors Ayala and Pere Brunet.

LSI-95-2-R "A note on learning decision lists", Jorge Castro.

LSI-95-3-R "The complexity of searching implicit graphs", José L. Balcázar.

LSI-95-4-R "Design quality metrics for object-oriented software development", Alonso Peralta, Joan Serras, and Olga Slavkova.

Copies of reports can be ordered from:

Nuria Sánchez
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Pau Gargallo, 5
08028 Barcelona, Spain
secrelsi@lsi.upc.es