

Early measures for UML class diagrams

Marcela Genero, Mario Piattini and Coral Calero

*ALARCOS Research Group
Department of Computer Science
University of Castilla - La Mancha
Ronda de Calatrava N° 5
13071 - Ciudad Real –
SPAIN*

mgenero@inf-cr.uclm.es

mpiattin@inf-cr.uclm.es

ccalero@inf-cr.uclm.es

ABSTRACT: Building software models before implementing them has become widely accepted in the software industry. Object models, graphically represented by class diagrams, lay the foundation for all later design work. So, their quality can have a significant impact on the quality of the software which is ultimately implemented, and an even greater impact if we take into account the size and complexity of current software systems. It is widely recognised that the production of better software requires the improvement of early development phases and the artifacts they produce. In this paper, we will introduce and analyse a set of an existent object oriented metrics that can be applied for assessing class diagrams complexity at the initial phases of the object oriented development life cycle. We also define our own proposal for new ones.

KEY WORDS: object oriented metrics, object oriented software quality, class diagram complexity

RESUME: La construction de modèles d'un logiciel bien avant sa mise en oeuvre est une pratique maintenant largement acceptée dans l'industrie du logiciel. Les modèles d'objets, représentés graphiquement par des diagrammes de classes, servent de fondations aux autres modèles. La qualité de ces diagrammes a donc un impact important sur la qualité du logiciel qui sera implémenté. Cet impact est même très important si on prends en compte la taille et la complexité des systèmes courants. Il est bien connu que la production d'un système de qualité nécessite un grand soin dans les premières phases du développement. Dans cet article nous introduisons et analysons un ensemble de métriques pour les objets permettant d'évaluer la complexité des diagrammes de classes lors de l'étape initiale d'un développement orienté-objet. Nous définissons également nos propres métriques pour compléter cet ensemble.

MOTS-CLES: Métrique orienté-objet, qualité du logiciel orienté-objet, diagramme de classe, complexité.

1. Introduction

The importance of models has been evident in all engineering disciplines for some time. This trend is also accepted in the software industry, where building software models before implementing them has become widely accepted.

Software models are used for several purposes [RUM 99]:

- To capture and state precisely requirements and domain knowledge so that all stakeholders may understand and agree on them
- To think about the system design
- To think about design decisions in a mutable form separate from the requirements
- To generate usable work products
- To organise, find, filter, retrieve, examine, and edit information about large systems
- To explore multiple solutions economically
- To master complex systems

The quality of object-oriented (OO) software systems depends heavily on the accuracy of the requirements specification. Therefore a major effort should focus on improving the models produced in the early phases of the software development life cycle.

Object models graphically represented by class diagrams form the basis of requirements specification and lay the foundation for all later design work. Therefore, their quality can have a significant impact on the quality of the system, which is ultimately implemented, and even greater impacts if we take into account the size and complexity of current software systems. Improving the quality of class diagrams will therefore be a major step forward in improving the quality of software development.

Quality is a multidimensional concept, composed of different characteristics such as functionality, reliability, usability, efficiency, maintainability and portability [ISO 99]. However, the definition of the different characteristics that compose the concept of “quality” is not enough on its own to ensure quality in practice as people will generally make different interpretations of the same concept. According to Total Quality Management (TQM) literature, measurable criteria for assessing quality are necessary to avoid “arguments of style” [ZUL 92].

Metrics provide a valuable insight into specific ways of enhancing software quality. They are used not only for understanding, controlling, and improving development but also for determining the best ways to help practitioners and researchers [PFL 97].

As is widely recognised, maintenance is the most important problem of software development, ranging between 60 and 90 percent of life cycle costs [CAR 90; PIG 97]. Consequently, we consider it is very important to tackle in this work the quality characteristic, maintainability.

Maintainability can be evaluated through several quality sub-characteristics [ISO 99] such as analysability, changeability, testability, stability and maintainability compliance. The first three sub-characteristics are in turn influenced by complexity [LI 87]. For many years researchers have sought to characterise general notions of “complexity” by a real single number. However, as is quoted by Fenton [FEN 94] a general complexity metric is “*the impossible holy grail*”. Henderson-Sellers in [HEN 96] distinguishes three types of complexity: computational, psychological and representational, and for psychological complexity he considers three components: problem complexity, human cognitive factors and product complexity. In particular, for class diagrams we want to measure model complexity by collecting data on several product internal metrics [TIA 99].

Although, within the field of software measurement a plethora of metrics have been proposed for measuring OO software products most of them are related to products obtained from design and implementation phases [CHI 94; LOR 94; BRIT 96; HEN 96; ZUS 98]. De Champeaux [DEC 97] has proposed only a few metrics in relation to class diagrams at the analysis phase. And Genero et al. [GEN 99] has proposed a set of metrics for OMT [RUM 91] class diagrams.

The goal of this paper is to present a state of the art in metrics that can be applied to measure class diagram complexity obtained in the initial phases of the development life cycle (section 2). We also propose a set of new ones (section 3). We will focus this paper on The Unified Modelling Language (UML) [OMG 99], which has emerged as a standard general-purpose visual modelling language. This is used to specify, visualise, construct, and document the artifacts of an OO software system. UML is intended to unify past experience about OO modelling techniques and to incorporate the best current software practices into a standard approach.

We have to be aware that a modelling language such as UML, can only give us a syntax and semantics to work with, but it cannot tell us whether a “good” model has been produced. Naturally, even when the language is mastered, there is no guarantee that the models produced will be good. Therefore, it is necessary to assess their quality. Marchesi’s work [MAR 98], to our knowledge, is the only work proposed to measure UML class diagrams at the analysis phase. But it disregards some measurable UML elements.

Finally, section 4 summarises the paper, draws on our conclusions, and presents future trends in metrics for object modelling.

2. State of the art in metrics for class diagrams

In this section we will summarise a set of OO product internal metrics extracted from existing literature. We only consider those metrics that may be applied to measure class diagram complexity, built at the initial stages of the development life cycle. For each metric we present its definition, its goal and some useful comments about it. Finally, in section 2.5 we will present an analysis of the different proposals.

As our idea is to collect measurement data at the beginning of the software development life cycle, we focus our metrics on class diagrams. For our purpose we consider that a class diagram has the following elements:

- Packages
- Classes
- Each class has attributes and operations
- Operations only have their signature, i.e. the definition of parameters
- Relationships: Association, Aggregation, Generalisation and Dependencies

We call these elements “measurable” only because our metrics are related to them. We think that these elements are always available in a class diagram in the initial phases of the development life cycle.

2.1 Chidamber and Kemerer's metrics

Chidamber and Kemerer [CHI 94] proposed a set of six OO design metrics. They are well-known in the field of OO design metrics but have not been widely accepted. These metrics can be used in the design phase, and are defined at class level. We only list those which can be applied to an UML class diagram constituted of the elements listed above.

– *Depth Inheritance Tree*

DEFINITION. The Depth of inheritance of a class is the DIT metric for a class. In cases involving multiple inheritance, the DIT will be the maximum length from the node to the root of the tree.

GOAL. DIT is a measure of how many ancestor classes can potentially affect this class. This metric was proposed as a measure of class complexity, design complexity and potential reuse. It is based on the idea that the deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit.

COMMENTS. Basili et al. [BAS 96] have put this metric under empirical validation, concluding that the larger the DIT, the larger the probability of fault detection. Briand et al. [BRI 96] have classified this metric as a length measure.

Zuse [ZUS 98] has demonstrated that this metric is above the ordinal scale, which, he argues, has relevance because with ordinal numbers little can be done.

– *Number of Children*

DEFINITION. The Number of Children (NOC) is the number of immediate subclasses subordinated to a class in the class hierarchy.

GOAL This is a measure of how many subclasses are going to inherit the methods of the parent class.

COMMENTS. Although a greater number of children indicate a greater code reuse, it has some problems: a) The greater likelihood of improper abstraction of the parent class and misuse of subclassing, b) Greater difficulty for modifying a class, because it affects all of the dependants of this class. It may require more testing of the methods in the class. Basili et al. [BAS 96], have put this metric under empirical validation, observing that the larger the NOC, the lower the probability of fault detection. This surprising trend can not be generalised, more experimentation is needed. Briand et al. [BRI 96] have classified this metric as a length measure. Zuse [ZUS 98] has demonstrated that this metric is above the ordinal scale, which, he argues, has relevance because with ordinal numbers little can be done.

2.2 Lorenz and Kidd 's metrics

Lorenz and Kidd [LOR 94] proposed a group of metrics called “design metrics”, which deal with the static characteristics of software design. Lorenz and Kidd have categorised their metrics thus:

- Class size metrics, which deal with quantifying an individual class.
- Class Inheritance metrics, which look at the quality of the classes' use of inheritance.
- Class Internals, which look at general characteristics of classes.

We only list those which can be applied to an UML class diagram constituted by the elements listed above.

2.2.1 Class size metrics

– *Number of Public Instance Methods in a Class*

DEFINITION. Number of Public Instance Methods in a Class (PIM) is defined as the total number of public instance methods in a class. Public methods are those that are available as services to other classes.

GOAL. This metric was defined by Lorenz and Kidd [LOR 94] as a measure of the class size. The number of public instance methods in a class is a good measure of the amount of responsibility in the class.

COMMENTS. Lorenz and Kidd [LOR 94] suggest using this metric to help in estimating the amount of work needed to develop a class.

– *Number of Instance Methods in a Class*

DEFINITION. The Number of Instance Methods in a Class (NIM) counts all the public, protected, and private methods defined for class' instances.

GOAL. This metric was defined by Lorenz and Kidd [LOR 94] as a measure of the class size. The number of methods in a class relates to the amount of collaboration being used.

COMMENTS. Larger classes may be trying to do too much of the work themselves instead of putting the responsibilities where they belong. They are more complex and harder to maintain. Smaller classes tend to be more reusable, since they provide one set of cohesive services instead of a mixed set of capabilities. Lorenz and Kidd [LOR 94] suggest examining patterns of instance variable usage to see if there are useful ways to split the class.

– *Number of Instance Variables in a Class*

DEFINITION. The Number of Instance Variables in a Class (NIV) is defined as the total number of instance variables in a class. Instance variables include private and protected variables available to the instances.

GOAL. This metric was defined by Lorenz and Kidd [LOR 94] as a measure of the class size.

COMMENTS. The fact that a class has more instance variables indicates that the class has more relationships to other objects in the system. Lorenz and Kidd [LOR 94] suggest that classes are more reusable when they have fewer instance variables.

– *Number of Class Methods in a Class*

DEFINITION. The Number of Class Methods in a Class (NCM) is defined as the total number of class methods in a class. A class method is a method that is global to its instances.

GOAL. The number of methods available to the class and not its instances affects the size of a class.

COMMENTS. This number should generally be relatively small compared to the number of instance methods. The number of class methods can indicate the amount of commonality being handled for all instances.

– *Number of Class Variables in a Class*

DEFINITION. The Number of Class Variables in a Class (NCV) is defined as the total number of class variables in a class.

GOAL. The number of variables available to the class and not its instances affects the size of a class.

COMMENTS. Class variables are often used to provide customisable constant values that are used to affect the behaviour of all the instances. The average number of class variables should be low. In general there should be fewer class variables than instance variables.

2.2.2 Class inheritance metrics

– *Number of Methods Overridden*

DEFINITION. The Number of Methods Overridden metric (NMO) is defined as the total number of methods overridden by a subclass. A subclass is allowed to define a method of the same name as a method in one of its superclasses. This is called overriding the method.

GOAL. This metric looks at the quality of the classes' use of inheritance. It examines superclass-subclass inheritance relationships

COMMENTS. A large number of overridden methods indicate a design problem.

– *Number of Methods Inherited*

DEFINITION. The Number of Methods Inherited metric (NMI) is defined as the total number of method inherited by a subclass.

GOAL. This metric looks at the quality of the classes' use of inheritance. It examines superclass-subclass inheritance relationships

COMMENTS. The number of methods inherited from superclasses indicates the strength of subclassing by specialisation.

– *Number of Methods Added*

DEFINITION. The Number of Methods Added (NMA) is defined as the total number of methods defined in a subclass.

GOAL. This metric looks at the quality of the classes' use of inheritance. It examines superclass-subclass inheritance relationships

COMMENTS. Subclasses should define new methods, extending the behaviour of the superclasses. A class with no methods is certainly questionable. The number of new methods should usually decrease as you move down through the layers of the hierarchy.

– *Specialisation Index*

DEFINITION. The Specialisation Index (SIX) for each class is defined thus:

$$\frac{\text{NumberOfOverriddenMethods} * \text{HierarchyNestingLevel}}{\text{TotalNumberOfMethods}}$$

GOAL. This metric looks at the quality of the classes' use of inheritance. The specialisation index measures to what extent subclasses redefine the behaviour of their superclasses.

COMMENTS. Lorenz and Kidd [LOR 94] have commented that this weighted calculation has done a good job on identifying classes worth looking at for their placement in the inheritance hierarchy and for design problems.

– *Average Parameters Per Method*

DEFINITION. The Average Parameters per Method (APPM) is defined thus:

$$\frac{\text{TotalOfMethodsParameters}}{\text{TotalNumberOfMethods}}$$

GOAL. This metric looks at the design of the class' internals.

COMMENTS. Lorenz and Kidd [LOR 94] suggest that parameters require more effort from clients, and high and low numbers of parameters imply a style of design. They also suggest an upper threshold of 0.7 parameters per method.

2.3 Brito e Abreu and Melo's metrics

Brito e Abreu and Melo [BRIT 96] proposed the MOOD (Metrics for Object Oriented Design) set of metrics. These metrics allow the measurement of the main mechanisms of the OO paradigm, such as, encapsulation, inheritance, polymorphism and message passing. MOOD metrics can be used in the design phase, and are defined at system level. Hereafter we describe only those that can be applied to the elements of a class diagram listed below.

2.3.1 Metrics at system level

– *Method Hiding Factor*

DEFINITION. The Method Hiding Factor (MHF) is defined as a quotient between the sum of the invisibilities of all methods defined in all of the classes and the total number of methods defined in the system under consideration. The invisibility of a method is the percentage of the total classes from which this method is not visible.

$$MHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{M_d(C_i)} (1 - V(M_{mi}))}{\sum_{i=1}^{TC} M_d(C_i)} \quad V(M_{mi}) = \frac{\sum_{j=1}^{TC} is_visible(M_{mi}, C_j)}{TC - 1}$$

$$is_visible(M_{mi}, C_j) = \begin{cases} 1 & \Leftrightarrow j \neq i \wedge C_j \text{ may call } M_{mi} \\ 0 & \text{otherwise} \end{cases}$$

Where: TC=total number of classes in the system under consideration, $M_d(C_i)=M_v(C_i)+M_h(C_i)$ =methods defined in C_i , $M_v(C_i)$ =visible methods in class C_i (public methods), $M_h(C_i)$ =hidden methods in class C_i (private and protected methods).

GOAL. MHF is defined as a measure of the information hiding concept that is supported by the encapsulation mechanism.

COMMENTS. The number of visible methods is a measure of the class functionality. Increasing the overall functionality will then reduce MHF. Brito e Abreu and Melo [BRI 96] have demonstrated empirically that when the value of MHF increases, the density of defects and the effort required to correct them would have to decrease. This metric does not take into account inherited methods. Harrison et al. [HAR 98] have shown that this metric is a valid measure within the context of the theoretical framework proposed in [KIT 95].

– *Attribute Hiding Factor*

DEFINITION. The Attribute Hiding Factor (AHF) is defined as a quotient between the sum of the invisibilities of all attributes defined in all of the classes and the total number of attributes defined in the system under consideration. The invisibility of an attribute is the percentage of total classes from which this attribute is not visible.

$$AHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{A_d(C_i)} (1 - V(A_{mi}))}{\sum_{i=1}^{TC} A_d(C_i)} \quad V(A_{mi}) = \frac{\sum_{j=1}^{TC} is_visible(A_{mi}, C_j)}{TC - 1}$$

$$is_visible(A_{mi}, C_j) = \begin{cases} 1 & \Leftrightarrow J \neq i \wedge C_j \text{ may reference } A_{mi} \\ 0 & \text{otherwise} \end{cases}$$

Where: $A_d(C_i) = A_v(C_i) + A_h(C_i)$ = attributes defined in C_i , $A_v(C_i)$ = visible attributes in class C_i , $A_h(C_i)$ = hidden attributes in class C_i (public attributes), $M_h(C_i)$ = hidden attributes in class C_i (private and protected attributes),.

GOAL. AHF is defined as a measure of the information hiding concept that is supported by the encapsulation mechanism.

COMMENTS. Ideally the value of this metric would be 100%, all attributes would be hidden and only accessed by the corresponding class methods. Harrison et al. [HAR 98] have shown that this metric is a valid measure within the context of the theoretical framework proposed in [KIT 95].

– *Method Inheritance Factor*

DEFINITION. The Method Inheritance Factor (MIF) is defined as a quotient between the sum of inherited methods in all classes of the system under consideration and the total number of available methods (locally defined plus inherited) for all classes.

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

Where: $M_a(C_i) = M_d(C_i) + M_i(C_i)$ = available methods in C_i (those that can be invoked in association with C_i), $M_d(C_i) = M_n(C_i) + M_o(C_i)$ = methods defined in class C_i (those declared in C_i), $M_n(C_i)$ = new methods in class C_i (those declared within C_i that do not override inherited ones), $M_o(C_i)$ = overriding methods in

class C_i (those declared within C_i that override (redefine) inherited ones, $M_i(C_i)$ =inherited methods in class C_i (those inherited (and not overridden) in C_i)

GOAL. MIF is defined as a measure of inheritance, and therefore a measure of the level of reuse.

COMMENTS. Brito e Abreu and Melo [BRIT 96] have empirically demonstrated that when the value of MHF increase, the density of defects and the effort required to correct them would have to decrease. Harrison et al. in [HAR 98] have shown that this metric is a valid measure within the context of the theoretical framework proposed in [KIT 95].

– *Attribute Inheritance Factor*

DEFINITION. The Attribute Inheritance Factor (AIF) is defined as a quotient between the sum of inherited attributes in all classes of the system under consideration and the total number of available attributes (locally defined plus inherited) for all classes.

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

Where: $A_i(C_i)$ = $A_d(C_i)$ + $A_r(C_i)$ = attributes available in C_i (those that can be manipulated in association with C_i), $A_d(C_i)$ = $A_n(C_i)$ + $A_o(C_i)$ =attributes defined in class C_i (those declared in C_i), $A_n(C_i)$ =new attributes in class C_i (those declared within C_i that do not override inherited ones), $A_o(C_i)$ =overriding attributes in class C_i (those declared within C_i that override (redefine) inherited ones), $A_r(C_i)$ = attributes inherited in class C_i (those inherited (and not overridden) in C_i)

GOAL. Like MIF, AIF is defined as a measure of inheritance, and therefore a measure of the level of reuse.

COMMENTS. At first sight we might be tempted to think that inheritance should be used extensively. However, the excessive reuse trough inheritance make the system more difficult to understand and maintain. Harrison et al. [HAR 98] have shown that this metric is a valid measure within the context of the theoretical framework proposed in [KIT 95].

– *Polymorphism Factor*

DEFINITION. The Polymorphism Factor (PF) is defined as the quotient between the actual number of different possible polymorphic situations, and the maximum number of possible distinctive polymorphic situations for class C_i .

$$PF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) \times DC(C_i)]}$$

Where: $M_o(C_i)$ =overriding methods in class C_i , $M_n(C_i)$ =new methods in class C_i , $DC(C_i)$ =number of descendants of class C_i

GOAL. PF is a measure of the potential polymorphism.

COMMENTS. Polymorphism arises from inheritance and Brito e Abreu and Melo [BRIT 96] suggest that in some cases overriding methods could contribute to reduce complexity and therefore to make the system more understandable and easier to maintain. Harrison et al. [HAR 98] have shown that this metric is a valid measure within the context of the theoretical framework proposed in [KIT 95].

2.4 Marchesi's metrics

If Marchesi [MAR 98] proposed a set of metrics to measure UML class diagrams at the analysis phase, which is our objective in this work, she did not take into account some UML measurable elements, such as associations, aggregations and dependencies.

In this proposal an UML class diagram at the analysis phase includes:

- Classes and packages
- Simple inheritance hierarchies
- Dependencies among classes: every relationship between classes except inheritance.
- Single classes defined in term of their responsibilities, attributes and methods.

These metrics are divided into three categories: those related to single classes, those related to packages and those related to the system as a whole.

The main variables and constants used in class diagram metrics are shown in table 1:

Variables and constants	Description
C	Array whose elements are classes of the system
$N_C = \dim(C)$	Total number of classes
P	Array whose elements are the packages of the system
$N_P = \dim(P)$	Total number of packages
G	Array whose elements are the classes which are roots of inheritance hierarchies of the system
$N_G = \dim(G)$	Total number of root classes
$B^{(i)}$	Array whose elements are all super-classes of class C_i (at all levels)
$b^{(i)}$	Array whose elements are the indexes of all super-classes of class C_i . Consequently: $B^{(i)} = \{C_k \mid k \in b^{(i)}\}$
$R^{(k)}$	Array with responsibilities of class C_k
$NR_k = \dim(R^{(k)})$	Number of responsibilities of class C_k
NA_k	Number of abstract responsibilities of class C_k
$S^{(k)}$	Array with immediate subclasses of class C_k
$NS_k = \dim(S^{(k)})$	Number of immediate subclasses of class C_k
$D^{(k)}$	Array with dependencies starting from class C_k
$ND_k = \dim(D^{(k)})$	Number of dependencies starting from class C_k
$[P]_{NC \times NP}$	Class-package matrix: the element p_{ik} is one if class C_i belongs to package P_k . Each row of $[P]$ has one and only one element equal to one; all others are zero
$[D]_{NC \times NC}$	Dependency matrix: element d_{ik} is the number of dependencies between class C_i (client) and class C_k (server)

Table 1. Variables and constants used in Marchesi's metrics

2.4.1 Metrics for single classes

– CL1 metric

DEFINITION. CL1 metric is the weighted number of responsibilities of a class, inherited or not. It is defined thus:

$$CL1 = NC_i + K_a NA_i + K_r \sum_{h \in b^{(i)}} NC_h$$

Where NC_i is the number of concrete responsibilities of class C_i , NA_i is the number of abstract responsibilities of class C_i , $b^{(i)}$ is an array whose elements are the indexes of all superclasses of class C_i .

Responsibilities can be abstract, if they are specified in subclasses, or concrete, if they are detailed in the class they are defined.

GOAL. This measure is defined as a measure of the class complexity.

COMMENTS. This metric has not been validated either empirically or theoretically. Marchesi [MAR 98] planned validation as a future work.

– *CL2 metric*

DEFINITION. CL2 metric is the weighted number of dependencies of a class. Specific and inherited dependencies are differently weighted. It is defined thus:

$$CL2 = \sum_{k=1}^{N_C} (d_{ik})^{K_d} + K_e \sum_{j \in b^{(i)}} (d_{jk})^{K_d}$$

Where the exponent $K_d < 1$, N_C is the total number of classes, $[D]_{N_C \times N_C}$ is the dependency matrix and an element d_k is the number of dependencies between class C_i (client) and class C_k (server), $b^{(i)}$ is an array whose elements are the indexes of all superclasses of class C_i .

A dependency between a class C_i (client class) and a class C_k (server class), indicate indicates that the class C_i will use one or more service offered by the C_k .

GOAL. This measure is defined as a measure of the class complexity.

COMMENTS. This metric has not been validated either empirically or theoretically. Marchesi [MAR 98] planned validation as a future work.

2.4.2 Metrics for packages

– *PK1 metric*

DEFINITION. PK1 metric is related to the number of dependencies among classes belonging to a given package, P_k , and classes belonging to other packages. PK1 refers to dependencies whose clients are classes of P_k and whose servers are outside P_k . It is defined thus:

$$PK1 = \sum_{i/p_{ik}=1} \left(\sum_{h/p_{hk} \neq 1} d_{ih} \right)$$

Where $[P]_{NC \times NP}$ is the class-package matrix and an element p_{ik} is one if class C_i belongs to package P_k . Each row of $[P]$ has one and only one element equal to one; all others are zero.

GOAL. PK1 measures the extent of usage of classes of other packages by classes of package P_k . This metric is aimed to measure inter-package coupling.

COMMENTS. Marchesi [MAR 98] has considered that every relationship between classes, except inheritance, could be classified as a dependency (or collaboration) between a client class depending on a server class. Marchesi [MAR 98] suggests that one of the main quality criteria at the OO analysis level is the minimisation of inter-package coupling. So the value of this metric should not be high.

COMMENTS. This metric has not been validated either empirically or theoretically. Marchesi [MAR 98] planned validation as a future work.

– *PK2 metric*

DEFINITION. PK2 metric refers to the dependencies on server classes belonging to P_k . It is defined thus:

$$PK2 = \sum_{i/p_{ik} \neq 1} \left(\sum_{h/p_{hk}=1} d_{ih} \right)$$

Where $[P]_{NC \times NP}$ is the class-package matrix and an element p_{ik} is one if class C_i belongs to package P_k . Each row of $[P]$ has one and only one element equal to one; all others are zero.

GOAL. PK2 metrics is related to degree of reuse of the classes within a package. This metric is aimed at measuring inter-package coupling.

COMMENTS. Marchesi [MAR 98] has considered that every relationship between classes, except inheritance, could be classified as a dependency (or collaboration) between a client class depending on a server class. Marchesi [MAR 98] suggests that one of the main quality criteria at the OO analysis level is the minimisation of inter-package coupling. So the value of this metric should not be high. This metric has not been validated either empirically or theoretically. Marchesi [MAR 98] planned validation as a future work.

– *PK3 metric*

DEFINITION. PK3 metric is the average value of PK1 metric. It is defined thus:

$$PK3 = \frac{1}{N_p} \sum_{k=1}^{N_p} \left[\sum_{i/p_{ik}=1} \left(\sum_{h/p_{hk} \neq 1} d_{ih} \right) \right]$$

Where N_p is the total number of packages, $[P]_{N_C \times N_P}$ is the class-package matrix and the element p_{ik} is one if class C_i belongs to package P_k . Each row of $[P]$ has one and only one element equal to one; all others are zero.

GOAL. This metric is an estimate of overall coupling among packages.

COMMENTS. Marchesi [MAR 98] suggests that one of the main quality criteria at the OO analysis level is the minimisation of inter-package coupling. So the value of this metric should not be high. This metric has not been validated either empirically or theoretically. Marchesi [MAR 98] planned validation as a future work.

2.4.3 Metrics for systems

– *OA1 metric*

DEFINITION. OA1 is the overall number of classes, N_C

GOAL. This metric measures the global complexity of the whole class diagram.

COMMENTS. It is only a proposal, but Marchesi [MAR 98] didn't explain the grounds for this proposal. This metric has not been validated either empirically or theoretically. Marchesi [Mar 98] planned validation as a future work.

– *OA2 metric*

DEFINITION. OA2 is the overall number of inheritance hierarchies, N_G

GOAL. This metric measures the global complexity of the whole class diagram, due to generalisation of relationships.

COMMENTS. It is only a proposal, but Marchesi [MAR 98] didn't explain the grounds for this proposal. This metric has not been validated either empirically or theoretically. Marchesi [MAR 98] planned validation as a future work.

– *OA3 metric*

DEFINITION. OA3 is the average weighted number of classes. Let us define as $PR^{(i)}$ the value of metric CL1 for class C_i . Its average in all classes of the system is:

$$OA3 = \langle PR^{(i)} \rangle = \frac{1}{N_C} \sum_{i=1}^{N_C} PR^{(i)}$$

GOAL. This metric measures the global complexity of the whole class diagram.

COMMENTS. It is only a proposal, but Marchesi [MAR 98] didn't explain the grounds for this proposal. This metric has not been validated either empirically or theoretically. Marchesi [MAR 98] planned validation as a future work.

– *OA4 metric*

DEFINITION. OA4 is the standard deviation of the weighted number of classes. Let us define as $PR^{(i)}$ the value of metric CL1 for class C_i . The standard deviation of $PR^{(i)}$ is:

$$OA4 = \sqrt{\frac{1}{N_C} \sum_{i=1}^{N_C} (PR^{(i)} - \langle PR^{(i)} \rangle)^2}$$

GOAL. This metric measures the global complexity of the whole class diagram.

COMMENTS. It is only a proposal, but Marchesi [MAR 98] didn't explain the grounds for this proposal. This metric has not been validated either empirically or theoretically. Marchesi [MAR 98] planned validation as a future work.

– *OA5 metric*

DEFINITION. OA5 is the average of the number of direct dependencies of classes. The average of ND_i on all the classes of the system is:

$$OA5 = \langle ND_i \rangle = \frac{1}{N_C} \sum_{i=1}^{N_C} ND_i$$

GOAL. This metric measures the global complexity of the whole class diagram.

COMMENTS. It is only a proposal, but Marchesi in [MAR 98] didn't explain the grounds for this proposal. This metric has not been validated either empirically or theoretically. Marchesi in [MAR 98] planned validation as a future work.

– *OA6 metric*

DEFINITION. OA6 is the standard deviation of the number of direct dependencies of classes. The standard deviation of ND_i is:

$$OA6 = \sqrt{\frac{1}{N_C} \sum_{i=1}^{N_C} (ND_i - \langle ND_i \rangle)^2}$$

GOAL. This metric measures the global complexity of the whole class diagram.

COMMENTS. It is only a proposal, but Marchesi [MAR 98] didn't explain the grounds for this proposal. This metric has not been validated either empirically or theoretically. Marchesi [MAR 98] planned validation as a future work.

– *OA7 metric*

DEFINITION. This metric is the percentage of inherited responsibilities with respect to their total number. Let us define AR_k as the total number of inherited responsibilities of class C_k , excluding those concretely specified or redefined in class C_k , and with XR_k the total number of responsibilities of class C_k , both inherited or not:

$$AR_k = \sum_{h \in b^{(k)}, \text{excluding} \\ \text{responsibilities} \\ \text{redefined in } C_k}$$

Then:

$$XR_k = NR_k + \sum_{h \in b^{(k)}} NR_h \qquad OA7 = \frac{\sum_{k=1}^{N_C} AR_k}{\sum_{k=1}^{N_C} XR_k}$$

GOAL. This metric measures the global complexity of the whole class diagram.

COMMENTS. It is only a proposal, but Marchesi [MAR 98] didn't explain the grounds for this proposal. This metric has not been validated either empirically or theoretically. Marchesi [MAR 98] planned validation as a future work.

2.5 Analysis of the existent metrics

After showing some proposals of metrics that can be applied to measure class diagram complexity at the analysis phase, we have concluded that most of them lack metrics referring to the complexity introduced by relationships, such as, associations, aggregations and dependencies. This fact can be deduced analysing tables 2 and 3. Also, it is evident that generalisation has been the area most widely addressed in the field of OO metrics.

By studying tables 2 and 3, we can deduce that most of the proposed OO metrics deals with class complexity, without paying special attention to package complexity or system complexity.

Scope →	Classes					
	Attributes	Methods	Relationships			
			Gen	Agg	Assoc	Dep
Proposals ↓						
Lorenz and Kidd [LOR 94]	X	X	X			
Chidamber and Kemerer [CHI 94]		X	X			
Brito e Abreu and Melo [BRI 96]						
Marchesi [MAR 98]		X	X		X	

Table 2. Comparison on class-scope metrics

Scope →	Packages					
	Attributes	Methods	Relationships			
			Gen	Agg	Assoc	Dep
Proposals ↓						
Lorenz and Kidd [LOR 94]						
Chidamber and Kemerer [CHI 94]						
Brito e Abreu and Melo [BRI 96]	X	X	X			

Marchesi [MAR 98] ^(*)	X	X	X
----------------------------------	---	---	---

^(*) Marchesi in [MAR 98] considered all relationships (except generalisations) as dependencies, without distinguishing between them.

Table 3. *Comparison of package-scope metrics*

It is also important to highlight that not all of the metrics presented in this section have been validated not only theoretically but also empirically.

3. Our proposal: Metrics related to relationships

In this section we will define metrics related to UML relationships, focussing above all on aggregations, associations, and dependencies. We don't take into account generalisation, because as you can see in tables 2 and 3, this kind of relationship has been addressed in most of the existent proposals.

We will consider intra-package relationships (within a package) but we could easily extend them to inter-package relationships. The allowed relationships between packages are dependency and generalisations [ERI 98].

3.1. Association metrics

Some proposals of metrics that have been defined to measure conceptual data models, like the Entity/Relationship models [GEN 00b; GEN 00c], could be tailored to measure UML class diagrams.

3.1.1 Class-scope metrics

– *Number of Associations of a Class*

DEFINITION. The Number of Associations of a Class metric (NAC) is defined as the total number of associations that a class has in a class diagram.

GOAL. The complexity of a class depends on the number of associations it has with other classes. This metric could identify which classes are the main ones in a class model.

COMMENTS. NAC is a size measure according to Briand et al.'s framework [BRI 96]. We will also define metrics that can be applied to measure the overall complexity of packages due to association relationships.

3.1.2 Packages-scope metrics

– *Number of Associations in a Package*

DEFINITION. The Number of Associations in a Package metric (NAP) is defined as the total number of associations within a package.

GOAL. The size of the package depends on the number of associations it has. This metric could identify which packages are the biggest and perhaps which are candidate for splitting.

COMMENTS. NAP is a size measure according to Briand et al.'s framework [BRI 96].

– *Number of Associations vs. Classes in a Package*

DEFINITION. The Number of Associations vs. Classes in a Package metric (NAVCP) is defined as the ratio between the number of associations in a package (NAP) divided by the number of classes in the package.

GOAL. This metric refines the previous one. The more association per class the package has, the more complex it will be, and the more difficult to understand and maintain.

3.2 Aggregation metrics

UML supports two different ways of representing the aggregation concept: aggregation as a special kind of binary association and the aggregation tree notation. Henderson-Sellers [HEN 97] has criticised how UML deals with aggregation. He made a different proposal concerning aggregation relationships in conceptual design, richer than the UML's. In spite of this, as our focus is UML class diagrams we tackle UML's aggregation.

3.2.1 Class-scope metrics

– *Height of Aggregation.*

DEFINITION. The height of a class within an aggregation hierarchy (HAgg) is defined as the maximal path from the class to the leaves.

GOAL. This metric measures the class complexity due to aggregation relationship (whole-part relationship).

COMMENTS. We suggest that the higher the class is in an aggregation hierarchy, the greater its complexity and therefore, the greater the cost of implementation and maintenance results. This metric has been theoretically validated [GEN 00a]

as a length measure, following the formal measurement framework proposed by Briand et al. [BRI 96].

– *Number of Direct Parts*

DEFINITION. The Number of Direct Parts metric (NODP) is defined as the total number of “direct part” classes which compose a composite class.

GOAL. This metric measures the class complexity due to aggregation relationship (whole-part relationship).

COMMENTS. We suggest that the greater the number of “part” classes of a “whole” class, the greater the likelihood of improper abstraction of the “whole” class, which may be a misuse of aggregation. The number of “part” classes gives an idea of the potential influence a class has on the design. If a “whole” class has a large number of “part” classes, it may require more implementation and maintenance time. This metric has been theoretically validated [GEN 00a] as a size measure, following the formal measurement framework proposed by Briand et al. [BRI 96].

– *Number of Parts*

DEFINITION. The Number of Parts metric (NP) is defined as the number of “part” classes (direct and indirect) of a “whole” class. Making an analogy with generalisation hierarchies, it will be the number of descendants of a class.

GOAL. This metric measures class complexity due to aggregation relationship (whole-part relationship).

COMMENTS. We suggest that the greater the number of “part” classes of the subtree that has as a root a “whole” class, the greater the design complexity of such a class. This may also require a greater cost of implementation and maintenance. This metric has been theoretically validated [GEN 00a] as a size measure, following the formal measurement framework proposed by Briand et al. [BRI 96].

– *Number of Wholes*

DEFINITION. The Number of Wholes metric (NW) is defined as the number of “whole” classes (direct or indirect) of a “part” class. Making an analogy with generalisation hierarchies, it will be the number of predecessors of the class being measured.

GOAL. This metric measures the class complexity due to aggregation relationship (whole-part relationship).

COMMENTS. The greater the number of “whole” which form a “part” class, the greater the likelihood of misuse of aggregation and greater design complexity when adding new classes or modifying the structure of the aggregation hierarchy. This metric has been theoretically validated [GEN 00a] as a size

measure, following the formal measurement framework proposed by Briand et al. [BRI 96].

– *Multiple Aggregation*

DEFINITION. The Multiple Aggregation metric (MAgg) is defined as the number of direct “whole” classes that have a class in an aggregation hierarchy.

GOAL. This metric measures class complexity due to multiple aggregation.

COMMENTS. A higher number of extra “whole” classes of a class, may indicate greater use of multiple aggregation, greater design complexity and therefore, a greater cost of implementation and maintenance. This metric has been theoretically validated [GEN 00a] as a size measure, following the formal measurement framework proposed by Briand et al. [BRI 96].

3.2.2 Packages-scope metrics

– *Number of Aggregation Relationships*

DEFINITION. The Number of Aggregation Relationships metric (NAggR) is defined as the number of aggregation relationships within a package.

GOAL. We propose this new metric to measure the package complexity due to aggregation relationship (whole-part relationship).

COMMENTS. A higher number of aggregation relationships constitutes a greater design complexity. Consequently, they may require greater cost in their implementation and maintenance. This metric has been theoretically validated [GEN 00a] as a size measure, following the formal measurement framework proposed by Briand et al. in [BRI 96].

3.3 Dependency metrics

Firstly in this section we will define metrics that can be applied to measure the complexity of each class due to dependency relationships. They help reveal the degree to which inter-class dependencies exist. Ideally, classes should be independent, making them easy to maintain and reuse.

3.3.1 Class-scope metrics

– *Number of Dependencies In*

DEFINITION. The Number of Dependencies In metric (NDepIn) is defined as the number of classes that depend on a given class.

GOAL. We propose this new metric to measure the class complexity due to dependency relationships.

COMMENTS. The greater the number of classes that depend on a given class, the greater the inter-class dependency and therefore the greater the design complexity of such a class. The inter-class dependency is also called export coupling [BRI 99a], which if misused could be a potential source of design complexity.

– *Number of Dependencies Out*

DEFINITION. The Number of Dependencies Out metric (NDepOut) is defined as the number of classes on which a given class depends.

GOAL. We propose this new metric to measure class complexity due to dependency relationships.

COMMENTS. The greater the number of classes on which a given class depends, the greater the inter-class dependency and therefore the greater the design complexity of such a class. This inter-class dependency is also called import coupling [BRI 99], which if misused could be a potential source of design complexity. It is better to minimise NDepOut value, since, higher values represent a situation in which many dependencies are spreading across the class diagram.

3.3.2 Packages-scope metrics

NDepIn and NDepOut metrics are defined at class level, but they can easily be extended to measure the dependencies inter-packages.

4. Conclusions

Due to the growing complexity of software systems, continuous attention to and assessment of object models are necessary in order to produce quality software systems. The fact that UML has emerged is a great step forward in object modelling. Even so this does not guarantee the quality of the models produced throughout the software life cycle. It is therefore necessary to have metrics in order to evaluate their quality from the first steps in the software development process.

In this paper we have presented a state of the art in OO metrics that can measure the complexity of UML class diagrams obtained in the initial phases of the OO development life cycle. Analysing several proposals, we deduce that there is a gap in OO metrics related to relationships, such as association, aggregation and dependency. We therefore propose new metrics, to cover this necessity. Our metrics are defined, at different levels of granularity: class and package. They should be useful for:

- Software Designers, because they will detect anomalies in the class diagram design, earlier in the software development life cycle, and they will take appropriate decisions before too much work is spent based on them. Also, they will be able to choose between alternative class diagram design
- Project managers, because they will be able to estimate maintenance costs

We want to highlight that our proposal cannot be considered as a final proposal. Instead, it is a starting point and we require feedback to improve it.

The proposed metrics use concepts and elements of UML, although they could be easily adapted to other modelling languages like OML [FIR 97], TROLL [JUN 91] and OASIS [PAS 95].

As with other aspects of Software Engineering, proposing techniques and metrics is not enough, it is also necessary to put them under theoretical and empirical validation, in order to assure their utility. Validation is critical to the success of software measurement [KIT 95; FEN 97; SCH 92; BAS 99].

In relation to empirical validation, we are carrying out some experimentation not only with controlled experiments but also with “real” cases taken from some companies, with the goal of assessing these metrics as predictors of maintenance efforts, and therefore, determining whether they can be used as early quality indicators. We have also put some of our proposed metrics under theoretical validation [GEN 00a] following Briand et al.’s framework [BRI 96]. But we are aware that more empirical and theoretical validation is needed in order to consider the proposed metrics as a final proposal.

In future work, we will focus our research towards measuring other quality factors like those proposed in the [ISO 99], which not only tackle class diagrams, but also evaluate other UML diagrams, such as use-case diagrams, state diagrams, etc. In our knowledge, little work has been done towards measuring dynamic and functional models [DER 95; POE 99; POE 00]. As is quoted in [BRIT 99] this is an area which lacks further investigation.

We are building a metric tool, called MANTICA, for collecting, analysing and visualising metric values, with the goal of obtaining threshold values that can help software designers from early phases of the OO development life cycle.

Acknowledgements

This research is part of the MANTICA project, partially supported by CICYT and the European Union (1FD97-0168).

References

[BAS 96] BASILI V., BRIAND L., MELO W., “A Validation of Object-Oriented Design Metrics as Quality Indicators”, *IEEE Transactions of Software Engineering*, vol. 22 no. 10, 1996, p. 751-761.

- [BAS 99] BASILI, V., SHULL F., LANUBILE F., “Building Knowledge Through Families of Experiments”, *IEEE Transactions on Software Engineering*, vol. 25 no. 4, 1999, p. 435-437.
- [BRI 96] BRIAND L., MORASCA S., BASILI V., “Property-Based Software Engineering Measurement”, *IEEE Transactions on Software Engineering*, vol. 22 no. 6, 1996, p. 68-86.
- [BRI 99] BRIAND L., MORASCA S., BASILI V., “Defining and Validating measures for Object-Based High-Level design”, *IEEE Software Engineering*, vol. 25 no. 5, 1999, p. 722-743.
- [BRIT 96] BRITO E ABREU F., MELO W., “Evaluating the Impact of Object-Oriented Design on Software Quality”, *Proceedings of 3rd International Metric Symposium*, 1996.
- [BRIT 99] BRITO E ABREU F., ZUSE H., SAHRAOUI H., MELO W., “Quantitative Approaches in Object-Oriented Software Engineering”, *Object-Oriented technology: ECOOP '99 Workshop Reader*, Lecture Notes in Computer Science 1743, Springer-Verlag, 1999, p. 326-337.
- [CAR 90] CARD D., GLASS R., “*Measuring Software Design Quality*”, Englewood Cliffs. USA, 1990.
- [CHI 94] Chidamber S., Kemerer C., “A Metrics Suite for Object Oriented Design”, *IEEE Transactions on Software Engineering*, vol. 20 no. 6, 1994, p. 476-493.
- [DEC 97] DE CHAMPEAUX D., *Object Oriented Development Process and Metric*, Prentice Hall, 1997.
- [DER 95] Derr K. *Applying OMT*, SIGS Books, New York, 1995.
- [ERI 98] ERIKSSON H, PENKER M., *UML Toolkit*, John Wiley & Sons, Inc., 1998.
- [FEN 94] FENTON, N., “Software Measurement: A Necessary Scientific Basis”, *IEEE Transactions on Software Engineering*, vol. 20 no. 3, 1994, p. 199-206.
- [FEN 97] FENTON N., PFLEEGER S., *Software Metrics: A Rigorous Approach*. 2nd. edition. London, Chapman & Hall, 1997.
- [FIR 97] FIRESMITH D., HENDERSON-SELLERS B., GRAHAM I., *OPEN Modeling Language (OML) Reference Manual*, New York: SIGS, 1997.
- [GEN 99] GENERO M., MANSO M^a E., PIATTINI M., GARCÍA, F., “Assessing the Quality and the Complexity of OMT Models”, *2nd European Software measurement Conference - FESMA 9.*. Amsterdam, The Netherlands, 1999, p. 99-109.
- [GEN 00a] GENERO, M., PIATTINI, M. AND CALERO, C., “Métricas para jerarquías de agregación en diagramas de clases UML”, *Memorias del Jornadas Iberoamericanas de Ingeniería de Requisitos y ambientes de Software, IDEAS'2000*, Cancún, México, 5-7 Abril, 2000, p. 373-384.
- [GEN 00b] GENERO M., PIATTINI M., CALERO C. “Formalization of Metrics for Conceptual Data Models”, *UKAIS 2000*, Cardiff, 26-28 April 2000, McGraw Hill International (UK) Limited, p. 99-100.
- [GEN 00c] GENERO M., PIATTINI M., CALERO C. SERRANO M., “Measures to get better quality databases”, *ICEIS 2000*, Stafford, 4-7 July 2000, p. 49-54.
- [HAR 99] HARRISON R., COUNSELL S., NITHI, R., “An Evaluation of the MOOD set of Object-Oriented Software Metrics”, *IEEE Transactions on Software Engineering*, vol. 24 no. 6, 1999, p. 491-496.

- [HEN 96] HENDERSON-SELLERS B., *Object-oriented Metrics - Measures of complexity*, Prentice-Hall, Upper Saddle River, New Jersey, 1996.
- [HEN 97] HENDERSON-SELLERS B., "OPEN Relationships-Compositions and Containments", *Journal of Object-Oriented Programming*, SIGS Publications, vol. 10 no. 7, p. 51-55.
- ISO/IEC 9126-1, *Information technology- Software product quality – Part 1: Quality model*, 1999.
- [JUN 91] JUNGCLAUS R., et al., "Introduction to TROLL-A Language to Object-Oriented Specification on Information Systems", *Proceedings of IS-CORE Workshop WS '91*, 1991, p. 97-128.
- [KIT 95] KITCHENHAM B., PFLEGEER S., HENTON N., "Towards a Framework for Software Measurement Validation", *IEEE Transactions of Software Engineering*, vol. 21 no. 12, 1995, p. 929-943.
- [LI 87] LI H., CHENG W., "An empirical study of software metrics", *IEEE Transactions on Software Engineering*, vol. 13 no. 6, 1987, p. 679-708.
- [LOR 94] LORENZ M., KIDD J., *Object-Oriented Software Metrics: A Practical Guide*. Prentice Hall, Englewood Cliffs, New Jersey, 1994.
- [MAR 98] MARCHESI M., "OOA Metrics for the Unified Modeling Language", *Proceedings of the 2nd Euromicro Conference on Software maintenance an reengineering*, 1998, p. 67-73.
- [OMG 99] OBJECT MANAGEMENT GROUP, "UML Revision Task Force", *OMG Unified Modeling Language Specification, v. 1.3*. document ad/99-06-08., 1999.
- [PAS 95] PASTOR O., RAMOS I., *OASIS 2.1.1.: A Class Definition Language to Model Information Systems Using an Object-Oriented Approach*, 3rd ed., Technical Report. Politechnical University of Valencia, 1995
- [PFL 97] PFLEEGER S., "Assessing Software Measurement", *IEEE Software*, March/April 1997, p. 477-482.
- [PIG 97] PIGOSKI T., *Practical Software Maintenance*, Wiley Computer Publishing, New York, USA, 1997.
- [POE 99] POELS G. "On the use of a Segmentally Additive Proximity Structure to Measre Object Class Life Cycle Complexity", *Software Measurement: Current Trends in Research and Practice*, Deutscher Universitäts Verlag, 1999, p. 61-79.
- [POE 00] POELS G. "On the Measurement of Event-Based Object-Oriented Conceptual Models", *4th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, June 13 2000, Cannes, France.
- [RUM 91] RUMBAUGH J., BLAHA M., PREMERLANI W., EDDY F., LORENSEN W., *Object-Oriented Modeling and Design*, Prentice Hall, USA, 1991.
- [RUM 99] RUMBAUGH J. JACOBSON I., BOOCH G., *The Unified Modeling Language Reference Manual*, Addison Wesley Longman, Inc., 1999.
- [SCH 92] SCHNEIDEWIND N., "Methodology For Validating Software Metrics", *IEEE Transactions of Software Engineering*, vol. 18 no. 5, 1992, p. 410-422.
- [TIA 99] TIAN J., "Taxonomy and Selection of Quality Measurements and Models", *Proceedings of SEKE '99, The 11th International Conference on Software Engineering & Knowledge Engineering*, June 16-19 1999, p. 71-75.

[ZUL 92] ZULTNER R., “The Deming Way: Total Quality Management for Software”, *Proceedings of Total Quality Management for Software Conference*, April, Washington, DC, April 1992.

[ZUS 98] ZUSE H., *A Framework of Software Measurement*, Berlin, Walter de Gruyter.

Marcela Genero. *She is Assistant Professor at the Department of Computer Science at the University of Castilla-La Mancha, Ciudad Real, Spain. She received her MSc degree in Computer Science in the Department of Computer Science of the University of South, Argentine in 1989. Actually, she is a PhD student at the University of Castilla-La Mancha, Ciudad Real, Spain. Her research interests are: advanced databases design, software metrics, object oriented metrics, conceptual data models quality, database quality.*

Mario Piattini. *He is MSc and PhD in Computer Science by the Politechnical University of Madrid. Certified Information System Auditor by ISACA (Information System Audit and Control Association). Associate Professor at the Department of Computer Science at the University of Castilla-La Mancha, in Ciudad Real, Spain. Author of several books and papers on databases, software engineering and information systems. He leads the ALARCOS research group of the Department of Computer Science at the University of Castilla-La Mancha, in Ciudad Real, Spain. His research interests are: advanced database design, database quality, software metrics, object oriented metrics, software maintenance.*

Coral Calero. *She is Assistant Professor at the Department of Computer Science in the University of Castilla-La Mancha, Ciudad Real, Spain. She received her MS degree Computer Science in the Department of Computer Science of the University of Seville, Seville, Spain, in 1996. Actually, she is a PhD student at the University of Castilla-La Mancha, Ciudad Real, Spain. Her research interests are: database quality, metrics for advanced databases, formal verification and empirical validation of software metrics.*