Revised version: Originally published in Proceedings of
"4th Int. Conf. on Software Quality", McLean, VA, USA, 3-5 October 1994

Object-Oriented Software Engineering: Measuring and Controlling the Development Process

Fernando Brito e Abreu (INESC/ISEG)
Rogério Carapuça (INESC/IST)

INESC, Rua Alves Redol, 9, Apartado 13069, 1000 Lisboa, PORTUGAL
(phone: +351-1-3100226 / fax: +351-1-525843 / email: fba@inesc.pt)

ABSTRACT

Although the benefits of Object-Orientation are manifold and it is, for certain, one of the mainstays for software production in the future, it will only achieve widespread practical acceptance when the management aspects of the software development process using this technology are carefully addressed. Here, software metrics play an important role allowing, among other things, better planning, the assessment of improvements, the reduction of unpredictability, early identification of potential problems and productivity evaluation. This paper proposes a set of metrics suitable for evaluating the use of the main abstractions of the Object-Oriented paradigm such as *inheritance, encapsulation, information hiding or polymorphism* and the consequent emphasis on *reuse* that, together, are believed to be responsible for the increase in software quality and development productivity. Those metrics are aimed at helping to establish comparisons throughout the practitioners' community and setting design recommendations that may eventually become organization standards. Some desirable properties for such a metrics set are also presented. Future lines of research are envisaged.

1. INTRODUCTION

The continuing increase in both the amount and complexity of the information to be handled by organizations, and the need to develop cheaper, more reliable, easily alterable and reusable application systems, required to deal with that information in an efficient and user-friendly way, was the ground where Object-Oriented (OO) technology has bloomed in recent years. However, object-orientation is not a panacea for successful system development as noted in [Jacobson92]. The shift from craftsmanship to industrialism must come on a more fundamental level that also includes the organization of the complete development process. Object-Oriented Software Engineering (OOSE) has, accordingly, received an inflated attention. Its main objective is to make the OO software development process an engineering activity, either by adapting "traditional" software engineering techniques, or by proposing its own. OOSE deals with technical and managerial issues, the former having received much more attention in the past few years.

Several problem areas in OOSE are, among others, the adoption of the OO technology itself (i.e. paradigm shift), the lack of adequate life-cycle models that support reusability, the ability to assess the quality of the development process and resulting products, and the capability of evaluating the productivity of development teams. These last two issues are fundamental in order for managers to control, steer and follow up software development efforts.

Quality and productivity are indeed the two most important input parameters for controlling any industrial process. Successful control requires some means of measurement. The need for software metrics is now fully recognized by the software engineering community and included in standards like the [ISO9000-3]. The reasons for using metrics in software development are mainly independent of the adopted paradigm, although the latter deeply influences the set of metrics to choose, as its concepts and corresponding abstractions are either disjoint or implemented differently [Abreu93].

This paper aims at achieving some advances towards a solution to the problem of assessing quality and productivity on OO systems and is organized as follows: the next section introduces the specific goals of the current research work in quantitative methods applied to Object-Orientation from which this paper originated. Section 3 presents a set of criteria for choosing a suitable metrics set. Section 4, the core of this paper, includes the detailed proposal of a metrics set named MOOD, for guiding and assessing OO design quality and potential productivity gains. The possible shapes of expected design recommendations based on the MOOD set are introduced in section 5. Some complementary research topics that deserve further effort are identified in section 6. Some concluding remarks are presented in section 7.

2. QUALITY AND OBJECT-ORIENTATION

Quality of software systems can be characterized by the presence of a certain number of **external**[1] **attributes**

[1] - Perceptible to purchasers, subcontractors and end users.

like *functionality, reliability, usability, efficiency, maintainability* and *portability* [ISO9126] which can be further detailed. However, due to the evident difficulty and cost[2] of evaluating those attributes, most efforts on the software quality field have focused on defining and evaluating suitable development processes. It is believed that a defined and controlled process will lead to the production of quality software products and with fewer costs. Let us call this approach "**outside-in**". Object-Orientation came to strengthen the complementary "**inside-out**" approach where the quality of internal structure is supposed to be the key for ensuring that (external) quality and increased productivity are achieved.

Object-Orientation is well suited for what is called "seamless development", which basically stands for using the same formalism and corresponding notation throughout the life-cycle, by means of stepwise refinement. The traditional barriers between analysis and design and particularly between design and coding, characterized by formalism shifts with corresponding translation rules, are bound to diminish. Therefore, analysis and design play an even more important role than ever. Coding, for instance, can be considered just as a "fill-in-the-design-blanks" activity. Better internal quality is due to new abstractions brought by this paradigm such as classes, methods, inheritance, polymorphism, encapsulation or messages and a corresponding increased emphasis on reuse. However, the use of those abstractions can be varied, depending mainly on the designer ability, so we can expect rather different quality products to emerge, as well as different productivity gains.

The aim of the research going on in our team is two-fold. First, we want to be able to identify quality OO designs by means of quantitative evaluation (i.e. using metrics) of the use of the paradigm abstractions that are supposed to be responsible for internal quality. Second, we want to express some of the external quality attributes and productivity advances as a function of those metrics. Our first step was to develop some metrics for OO designs, from a set of criteria presented in next section, that among other things, is expected to allow doing comparative evaluation throughout the OO community, and eventually help training new OO software practitioners by setting design standards that traduce best practice.

3. CRITERIA FOR DESIRED METRICS

The choice of a set of metrics exposes the pitfalls of measuring too much and becoming overwhelmed by a large amount of unmanageable numeric data, or measuring too little and not gaining sufficient insight into the desired objective. After surpassing this problem by deciding to adopt just a few (but not too few) metrics, we need to set some evaluation criteria based on the goals we want to achieve. Without them it is relatively easy to fall in the "YAM"[3] trap or become swamped by the myriad of those proposed in the available literature. A better perspective on this problem can be obtained in [Zuse91]. Next we will derive a set of criteria to help define the MOOD set.

Different people at different times or places should yield the same values when measuring the same systems. Subjectivity makes metrics comparisons throughout software industry an impossible mission. Subjective ratings (e.g. "Very Low", "Low", "Average", "High", "Very High") are copious in the metrics literature. That is undoubtedly one of the reasons that leads to metrics suspicion among software practitioners and the computer science community in general. One road to achieve this objectivity is:

Criterion 1: metrics determination should be formally defined

For being useful, metrics must be collected and analyzed throughout time in as many different projects as possible in order to establish comparisons and derive conclusions. However, those projects will surely vary in size. If metrics other than the ones specifically designed to measure size also depend on it, no cumulative knowledge will be achieved. So:

Criterion 2: non-size metrics should be system size independent

Metrics are supposed to represent some product or process attribute. Thus we are faced with the issue of units of measurement. Subjective or "artificial" units inevitably yield to misunderstandings. Remember, for instance, the discussions around different interpretations of LOC (lines of code) [Albrecht83] and Function Points [Symons91][Dreger89]. Then:

Criterion 3: metrics should be dimensionless or expressed in some consistent unit system

The cost of recovering from effects caused by an error increases exponentially with elapsed project progress, since its commitment. Metrics, particularly design ones, are aimed at exposing the defects provoked by those errors and buried in the design. We must be able to collect metrics as soon as a first design is available, if we want to identify the possible flaws, before too much effort is built on top of them. Therefore:

[2] - In industries other than software, quality assessment is often done by evaluating samples of massively produced products, thus allowing to dilute the corresponding effort (i.e. scale economy benefit).

[3] - Yet Another Metric...

Criterion 4: metrics should be obtainable early in the life-cycle

Real software systems are usually built by a team of people. Often is possible to break down the specification in almost independent modules or subsystems. Each team member or small group of members can be responsible for each of those subsystems. Then, we need metrics applicable not only to the whole system under consideration, but also to each one of its modules or subsystems, thus allowing to pin-point "ill-designed" ones. So:

Criterion 5: metrics should be down-scaleable

Metrics collection is a repetitive task, therefore tedious and boring for human beings. The worst is that it takes a lot of time and money! Provided that criterion 1 is met, and that designs are also formally defined, it is possible to build some kind of syntactic analyzer that extracts from them the needed information for computing the metrics. The effort to build such a tool is considerable but it is worth while. Then:

Criterion 6: metrics should be easily computable

Many specification and programming languages (either graphical or textual) that support the OO paradigm abstractions are available in the marketplace. Each of them has its own constructs that allow for implementation of those abstractions in more or less detail. Again, the requirement of a common base of understanding for the metrics analysis process, leads us to the need of avoiding the syntactic level. Tools such as those mentioned in the previous paragraph can guarantee this independence. Therefore:

Criterion 7: metrics should be language independent

Several authors have suggested sets of metrics for the OO paradigm; see for instance [Bieman92, Campanai94, Chidamber91, Karunanithi93, Yousfi92]. However, most of the proposed metrics do not fulfill all the above criteria, mainly 1, 2, 3 and 7.

4. THE MOOD METRICS SET

4.1 Introduction

The MOOD (Metrics for Object Oriented Design) set is a collection of metrics that were designed with the above defined criteria in mind. The set includes the following metrics:

- *Method Inheritance Factor*
- *Attribute Inheritance Factor*
- *Coupling Factor*
- *Clustering Factor*
- *Polymorphism Factor*
- *Method Hiding Factor*
- *Attribute Hiding Factor*
- *Reuse Factor*

Considering that metrics are intended to quantify the presence or absence of a certain property or attribute, we can view them as ***probabilities***. They would then range from **0** (total absence), to **1** (maximum possible presence). This perspective was also used in the ESPRIT REBOOT project [Stalhane92]. This kind of interpretation allows the application of statistical theory to software metrics. For instance, statistically independent metrics can be combined (e.g. multiplied) so that the result can still be interpreted as a probability.

The MOOD metrics definitions are based on a group of formally defined functions and on set theory and simple mathematics. This fulfills criteria 1 and 6. All MOOD metrics are expressed as quotients where the numerator is the actual use (in the design under consideration) of a given OO abstraction and the denominator is the maximum achievable possible value for the same abstraction use. As a result, all metrics are size independent and dimensionless and criteria 2 and 3 are met. The MOOD metrics meet criterion 4 because they are applicable as soon as a preliminary system design is available. The proposed metric set can be applied to any existent *Class Cluster* (as defined in section 4.4), or any combination of them, and so we can say that criterion 5 is also fulfilled. No reference is made to specific language constructs, that is, MOOD metrics refer to OO abstractions, and not to its implementations. Criterion 7 is therefore also accomplished.

4.2 Encapsulation and Information Hiding

Plenty of OO analysis and design methods, as well as programming languages exist, but often different terminologies are used to refer to the same abstraction with the inevitable entanglement of non-experts. The basic OO concepts are introduced along with metrics definitions in order to help this paper to be as self-explanatory as possible.

Objects are an encapsulation of information and behavior relative to some entity of the application domain under consideration (sometimes referred as the UoD - Universe of Discourse). In real systems many objects with similar information (data) and behavior (functionality) can be found. The ***class*** abstraction captures this reality and can be viewed as an abstract data type. The class definition includes at least two types of ***features: attributes*** (also called ***variables***, ***fields*** or ***data members***), which stand for the stored information and ***methods*** (also called ***operations, function members, tasks*** or ***routines***), which represent the behavior. All objects created belong to a certain class, so they are often referred as ***class***

instances. Within each class all objects share the same methods. The object instance variables (i.e. local data) are defined in the corresponding class.

Many features defined in a class are designed for implementing certain functionalities that are only relative to the class itself. Those features should be hidden from client programmers[4], not for protection sake but for helping them to cope with complexity, as internal implementation details do not (or should not) bring a better perspective on how to use the services of the class. Another very important advantage of this ***information hiding*** mechanism is that the use of a class is independent of its implementation, thus allowing to alter it without "side-effects". As a conclusion, all information about a class should be private to the class, unless it is specifically declared public. The public part of a class is called its ***interface*** and should only be the "tip of the iceberg". The public methods represent the ***services*** that the supplier class is able to render to client classes. The hidden part is called the ***implementation***. The number of methods defined in class C_i is given by:

$$M_d(C_i) = M_v(C_i) + M_h(C_i)$$

where:

- $M_v(C_i)$ - number of visible methods (interface) in class C_i
- $M_h(C_i)$ - number of hidden methods (implementation) in class C_i

Then we define the ***Method Hiding Factor***:

$$MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)}$$

Conversely, the number of attributes defined in class C_i is given by:

$$A_d(C_i) = A_v(C_i) + A_h(C_i)$$

where:

$A_v(C_i)$ - number of visible attributes in class C_i

$A_h(C_i)$ - number of hidden attributes in class C_i

Then we define the ***Attribute Hiding Factor***:

$$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}$$

[4]- Programmers in a team using the class, other than its creator.

4.3 Inheritance

Inheritance is a mechanism for expressing similarity among classes. Semantically, it allows the portrayal of generalization and specialization. As far as design is concerned, it allows for the simplification of the definition of inheriting classes. When a class inherits from another, that means it can use its methods and attributes, unless they are redefined locally. A class C_d that inherits directly or indirectly from a class C_a is said to be a ***descendent***[5] of class C_a which, conversely, is called an ***ancestor***[6] of class C_d. Now, being more restrictive, a class C_c that inherits directly from a class C_p is said to be a ***child*** of class C_p which, conversely, is called a ***parent*** of class C_c. Inheritance can be single or multiple, depending on the number of parents. The inheritance relation will be represented here by an arrow, for instance as in $C_c \rightarrow C_p$ or $C_d \rightarrow C_a$.

The composition of several inheritance relations defines a directed acyclic graph, often called an ***inheritance hierarchy tree***. A ***base class*** is the root of such an inheritance hierarchy (i.e. the one that does not inherit; who has no ancestors). Formally we can define the function:

$$is\_base(C_b) = \begin{cases} 1 & iff \quad \forall j \in [1, TC], \neg\exists C_j : C_b \rightarrow C_j \\ 0 & otherwise \end{cases}$$

where *TC* is the *total number of classes* in the system under consideration.

A ***leaf class*** is a class that has no descendants. Formally, we can define the function:

$$is\_leaf(C_l) = \begin{cases} 1 & iff \quad \forall j \in [1, TC], \neg\exists C_j : C_j \rightarrow C_l \\ 0 & otherwise \end{cases}$$

In a class hierarchy, each class can have features either inherited from its ancestors or the locally defined. These latter features can be either new or a redefined version of inherited ones (overriding situation), or even only declared but not implemented there[7]. Some approaches even allow to drop inherited features.

For defining the MOOD set of metrics we will need some basic ***class metrics***. They will be introduced next, by means of functions where the argument is the class under consideration and the returned value is the value for the corresponding metric. Let C_i be any class of the system under consideration. We define:

[5] - Sometimes also called subclass.

[6] - Also referred as superclass.

[7] - The implementation is left to its descendants. These kind of features are usually called "deferred".

- ***Children Count,*** $CC(C_i)$ - number of children of C_i (note: if $CC(C_i)$ = 0 then C_i is a leaf class)
- ***Descendants Count,*** $DC(C_i)$ - number of descendants of C_i
- ***Parents Count,*** $PC(C_i)$ - number of parents of C_i (notes: if $PC(C_i)$=0 then C_i is a base class; if $PC(C_i)$>1 we have multiple inheritance).
- ***Ancestors Count,*** $AC(C_i)$ - number of ancestors of C_i.
- $M_d(C_i)$ - number of Methods defined in C_i
- $M_n(C_i)$ - number of New[8] Methods defined in C_i
- $M_i(C_i)$ - number of Methods Inherited in C_i (not overridden)
- $M_o(C_i)$ - number of Overridden Methods in C_i (inherited methods that are redefined)
- $M_a(C_i)$ - number of Available[9] Methods in C_i

The following relations hold, and can be easily transposed to attributes instead of methods:

$$M_d(C_i) = M_n(C_i) + M_o(C_i)$$

$$M_a(C_i) = M_d(C_i) + M_i(C_i)$$
$$= M_n(C_i) + \sum_{j=1}^{PC(C_i)} M_a(C_j)$$
where $C_i \rightarrow C_j$

This last expression is recursive up the inheritance chain, till a base class is found. The only situation in which it is not valid is when methods inherited from different parents (when $PC(C_i) > 1$) have the same designation[10].

Some ***system metrics*** are now defined, based on the above ones. The demonstration of some expressions is not included here because they are either trivial or beyond the scope of this paper.

- ***Total number of Methods Defined,***

$$TM_d = TM_n + TM_o = \sum_{k=1}^{TC} M_d(C_k)$$

- ***Total number of New Methods defined,***

$$TM_n = \sum_{k=1}^{TC} M_n(C_k)$$

- ***Total number of Methods Overridden,***

$$TM_o = \sum_{k=1}^{TC} M_o(C_k)$$

- ***Total number of Methods Inherited,***

$$TM_i = \sum_{k=1}^{TC} M_i(C_k) = \sum_{k=1}^{TC} \left[M_n(C_k) * DC(C_k) - M_o(C_k) \right]$$
$$= \sum_{k=1}^{TC} \left[\left[M_d(C_k) - M_o(C_k) \right] * DC(C_k) - M_o(C_k) \right]$$

- ***Total number of Methods Available,***

$$TM_a = TM_d + TM_i = \sum_{k=1}^{TC} M_a(C_k)$$
$$= \sum_{k=1}^{TC} \left[\left[M_d(C_k) - M_o(C_k) \right] * \left[1 + DC(C_k) \right] \right]$$
$$= \sum_{k=1}^{TC} \left[M_n(C_k) * \left[1 + DC(C_k) \right] \right]$$

Note: iff there are no overriding situations, that is,

$$\forall i \in [1, TC] \quad M_d(C_i) = M_n(C_i)$$

the two expressions above become:

$$TM_i = \sum_{k=1}^{TC} M_i(C_k) = \sum_{k=1}^{TC} \left[M_d(C_k) * DC(C_k) \right]$$
$$= \sum_{k=1}^{TC} \left[M_n(C_k) * DC(C_k) \right]$$

$$TM_a = TM_d + TM_i = \sum_{k=1}^{TC} M_a(C_k)$$
$$= \sum_{k=1}^{TC} \left[M_n(C_k) * \left[1 + DC(C_k) \right] \right]$$
$$= \sum_{k=1}^{TC} \left[M_d(C_k) * \left[1 + DC(C_k) \right] \right]$$

- ***Total Length of Inheritance Chain***, (total number of inheritance relations),

$$TLIC = \sum_{i=1}^{TC} PC(C_i) = \sum_{i=1}^{TC} CC(C_i)$$

- ***Total Number of Inheritance Paths,*** **(**total number of inheritance paths from a base class to a leaf class),

$$TNIP =$$

$$TLIC - TC - \sum_{i=1}^{TC} \left[is\_base(C_i) + is\_leaf(C_i) \right]$$

We then define the ***Method Inheritance Factor*** as:

$$MIF = \frac{TM_i}{TM_a}$$

Note: MIF=0 means that there is no effective inheritance (i.e. there are no inheritance hierarchies or all inherited methods are overridden).

Similarly, we can also define the ***Attribute Inheritance Factor*** given by:

---

8 - "New" denotes those methods that are not overriding inherited ones.

9 - "Available" stands for those methods that can be invoked in association with the class under consideration (i.e. those defined in that class as well as the inherited ones).

10 - This situation is generally referred as "name clashing".

$$AIF = \frac{TA_i}{TA_a}$$

where TA_i and TA_a have definitions similar to TM_i and TM_a.

4.4 Coupling and Clustering

A class C_c is called a ***client*** of class C_s, and C_s a ***supplier***[11] of class C_c, whenever C_c contains at least one reference to a feature (method or attribute) of class C_s. We will represent this client-supplier relation by $C_c \Rightarrow C_s$.

Some of these client-supplier relations can be viewed as communications between class instances. These communications should be made explicit for the sake of understandability. Some approaches use the designations ***message, event*** or ***stimulus,*** to refer to the call that an instance of the client class does to a supplier class method. It is desirable that classes communicate with as few others as possible and even then, that they exchange as little information as possible [Meyer88].

Supplier class references are not only made by means of messages. A supplier class type reference can be made inside client classes in situations such as:

- a public or private global attribute (some languages only allow global attributes to be private)
- a public or private method argument or local attribute

A bigger number of client-supplier relations increase complexity, reduces encapsulation and potential reuse, and limits understandability and maintainability.

Every class in a certain system is a potential supplier of all other classes and vice-versa. Thus, the maximum value of client-supplier relations[12] is given by $TC^2 - TC$. If we consider the following function:

$$is\_client(C_c, C_s) = \begin{cases} 1 & iff \quad C_c \Rightarrow C_s \wedge C_c \neq C_s \\ 0 & otherwise \end{cases}$$

then, the ***Coupling Factor*** is given by:

$$COF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} is\_client(C_i, C_j)\right]}{TC^2 - TC}$$

The "web" defined by client-supplier and inheritance relations is, in real systems, a set of disjoined graphs where nodes represent classes and edges represent the relations. Each of these graphs is a potential cluster for reuse because they do not need to drag along anything else. We shall name them ***Class Clusters*** and the shape of each one will be of a single or several intercommunicating inheritance hierarchy trees. For a total number of Class Clusters, TCC, we define the ***Clustering Factor*****:**

$$CLF = \frac{TCC}{TC}$$

4.5 Polymorphism

Polymorphism is a greek originated word that means "many forms". When applied to Object-Orientation, it stands for the possibility of sending a message without knowing which will be the form (class) of the object that will bind that message to one of its interface methods. All the potential receiving classes belong to the same inheritance hierarchy tree. Binding can be static (at compilation time) or dynamic[13] (at run time).

Messages can be bound to instances of a certain class or to instances of one of its descendants and not the other way around. Consider for instance that the class ***ball*** is specialized by classes ***tennis_ball***, ***golf_ball***, ***soccer_ball*** and ***rugby_ball***. If by sending the message ***ball.new*** (call of constructor method), we get a ***tennis_ball*** or a ***golf_ball***, everything is fine. However, if the message ***soccer_ball.new*** is sent, getting some instance of class ***ball*** is not acceptable as it might consist, for example, of a ***rugby_ball***.

If there is no overriding, a message to a class or to one of its descendants will be bound to the same method (i.e. no polymorphism). Conversely, we will obtain the possible maximum polymorphism potential if all methods are overridden in all classes (except the base ones, of course). In fact, if a method M in class C_i, is overridden in all descendants of class C_i, then a message bound to M can have $DC(C_i)$ possible addressees other than the implementation of M in class C_i. Those correspond to the same amount of different implementations of M in C_i descendants (polymorphic situations). Extending this reasoning to whole methods in the whole system, the *maximum number of possible different polymorphic situations* is given by:

$$\sum_{i=1}^{TC}\left[M_d(C_i) * DC(C_i)\right]$$

For a given system, however, the total number of *possible different polymorphic situations* is:

$$\sum_{i=1}^{TC}\left[\sum_{j=1}^{DC(C_i)} M_o(C_j)\right]$$

11 - Note that a class can be client (and therefore supplier) of itself.

12 - Here we have not considered the "harmless" reflexive client-supplier situations, that is, a class being supplier of itself.

13 - Also called late or delayed binding.

Note: the inner sum refers to the descendants of C_i

Finally we define the ***Polymorphism Factor*:**

$$PF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{DC(C_i)} M_o(C_j)\right]}{\sum_{i=1}^{TC}\left[M_d(C_i) * DC(C_i)\right]}$$

4.6 Reuse

Reuse, enforced by the OO paradigm abstractions, is expected to produce big impacts on development productivity and quality. It apparently[14] saves a lot of development time, thus reducing system costs or allowing savings to be spent on building more functionality, quality assurance or other activities. Reusable components are usually more carefully designed than ordinary program code. Besides, its repeated use brings out quickly any flaws in its design or implementation. That is why those components tend to be of better quality, therefore embedding that quality in systems who incorporate them.

Reuse in OO software development can mainly take two shapes: reuse of ***library components*** and reuse by means of ***inheritance***. Then, we can consider three types of classes in a given system: (i) those base classes built totally new, (ii) those extracted from a library and finally (iii) those that reuse existing classes by means of inheritance. As stated in the inheritance section above, inherited class definitions are usually specialized either by extending its features or by redefining ("overriding") them. This specialization effort, together with that of building classes "from scratch" corresponds to the effective "new" part of the system under consideration. We want to quantify exactly the other part, that is, the one corresponding to used library classes and the fraction of all others that may be imputed to inheritance. For this fraction calculation, we will only consider methods, as those are much more expensive to build and maintain than attributes. Now we can define the ***Reuse Factor*** as:

$$RF = \frac{\sum_{i=1}^{TC} in\_library(C_i)}{TC} + \frac{MIF * \sum_{i=1}^{TC}\left[1 - in\_library(C_i)\right]}{TC}$$

where

$$in\_library(C_i) = \begin{cases} 1 & iif \quad C_i \in L \\ 0 & otherwise \end{cases}$$

L = {classes in the reusable class library}

5. DESIGN HEURISTICS

By thorough interpretation of data taken from real projects, we believe that we will be able to compute design heuristics. Those can exhibit three shapes: *recommended lower limit* (LL), *recommended interval* (INT) and *recommended upper limit* (UL). Table 1 shows which shape applies for each of the MOOD metrics. The appropriateness of each limit (including the interval ones) is expected to increase as our metrics collection and analysis process proceed.

MOOD METRIC	LL	INT	UL
Method Inheritance Factor		x	
Attribute Inheritance Factor		x	
Coupling Factor			x
Clustering Factor	x		
Polymorphism Factor		x	
Method Hiding Factor	x		
Attribute Hiding Factor	x		
Reuse Factor	x		

Table 1: Shape of design heuristics based on MOOD

According to this framework, expected recommendations will be of the kind:

- *"Keep the Method Inheritance Factor between 0.25 and 0.37"*
- *"Coupling Factor should be below 0.52"*
- *"Good Reuse Factors are those above 0.43"*

Values mentioned above are irrelevant. We expect to disclose some realistic ones in a following paper.

6. FUTURE WORK

We are presently developing a tool for supporting the collection, storage and analysis of the MOOD metrics set. The core of this tool (metrics definition dictionary, metrics storage, human-machine interface) is language independent. Specific implementation language stubs will parse the specification code and determine the base measure function values. The C++ stub is under construction and we plan to develop an Eiffel [Meyer92] one in the near future. When the tool becomes fully operational, we will proceed to an extensive evaluation of available systems and try to derive and refine the values for the limits mentioned in the design recommendations.

The study of correlation between MOOD metrics and quality attributes as those mentioned in [ISO9126] will be

[14] - We can not downplay the effort of searching, and eventually adapting, reusable components from the class library, as well as the effort of building, validating and maintaining it.

one of next steps. We will also investigate the statistical independence of each MOOD metric towards each of the other ones.

We think that the MOOD metrics (except the Reuse Factor) can be combined to obtain a generic OO software system complexity metric. That is one of our future challenges. We will start by evaluating the MOOD metrics against a set of *desiderata* for software complexity metrics defined in [Weyuker88].

A concurrent effort for developing a resource estimation model named MOORED (Model for Object Oriented Resource Estimation Determination) is under way, and some cross-fertilization is expected in the field of complexity and productivity evaluation.

7. CONCLUSIONS

The adoption of the Object-Oriented paradigm is expected to help produce better and cheaper software. The main concepts of this paradigm, namely, *inheritance, encapsulation, information hiding or polymorphism*, are the keys to foster reuse and achieve easier maintainability. However, the use of constructs that support those concepts can be more or less intensive, mainly depending on the designer ability. Advances in quality and productivity need to be correlated with the use of those constructs. Therefore, we need to evaluate them quantitatively to guide OO design. The availability of these metrics should allow comparison of different systems or different implementations of the same system, thus helping to derive some design heuristics that could/should be included in design tools. Those heuristics would at least be a valuable help to new staff members.

"Blind" choice (or creation) is dangerous, so a set of common requirements for metrics and corresponding rationale was introduced, which includes the need for formal definition, language independence, dimensionlessness, ease of calculation and early obtainability. A suitable metrics set named MOOD was then proposed. We believe that these metrics can help in setting OO design standards at the organization level, helping OO practitioners to guide their development process and, hopefully, leaving them in a cheerful MOOD...

REFERENCES

[Abreu93] Abreu F.B., "Metrics for Object Oriented Software Development", *Proceedings of 3rd International Conference on Software Quality*, ASQC, Lake Tahoe, USA, October 1993.

[Albrecht83] Albrecht A.J. and Gaffney J.E., "Software Function, Source Lines of Code and Development Effort Prediction", *IEEE TSE*, vol.9, n.6, pp.639-648, November 1983.

[Bieman92] Bieman J., "Deriving Measures of Software Reuse in Object Oriented Systems", *Proceedings of The BCS-FACS Workshop on Formal Aspects of Measurement*, Springer-Verlag, 1992.

[Campanai94] Campanai M. and Nesi P., "Supporting O-O Design with Metrics", *Proceedings of TOOLS Europe'94*, France, 1994.

[Chidamber91] Chidamber S. and Kemerer C., "Towards a Metrics Suite for Object Oriented Design", *Proceedings of OOPSLA'91*, pp.197-211, 1991.

[Dreger89] Dreger J.B., *Function Point Analysis*, Prentice-Hall, Englewood Cliffs, NJ,1989

[ISO9000-3] ISO/IEC 9000 Part 3, *Guidelines for the Application of ISO 9001 to the Development, Supply and Maintenance of Software*, 1991.

[ISO9126] ISO/IEC 9126, *Information Technology - Software Product Evaluation - Quality Characteristics and Guidelines for their use*, 1991.

[Jacobson92] Jacobson I., Christerson M., Jonsson P. and Övergaard G., *Object-Oriented Software Engineering - A Use Case Driven Approach*, ACM Press / Addison-Wesley, 1992.

[Karunanithi93] Karunanithi S. and Bieman J., "Candidate Reuse Metrics For Object Oriented and Ada Software," *Proceedings of IEEE International Software Metrics Symposium*, pp.120-128, May 1993.

[Meyer88] Meyer B., *Object-oriented Software Construction*, Prentice Hall International, 1988.

[Meyer92] Meyer B., *Eiffel: The Language*, Prentice Hall International, 1992.

[Stalhane92] Stalhane T. and Coscolluela A., "Final Report on Metrics", *Deliverable D1.4.B1*, ESPRIT Project 5327 (REBOOT), February 1992.

[Symons91] Symons C.R., *Software Sizing and Estimating - Mk II Function Point Analysis*, John Wiley & Sons, 1991.

[Weyuker88] Weyuker E., "Evaluating Software Complexity Metrics", *IEEE TSE*, vol.14, n.9, pp.1357-1365, September 1988.

[Yousfi92] Yousfi N., "Measuring Internal Attributes of Object-Oriented Software Products", *Proceedings of the 5th Int. Conference on Software Engineering & its Applications*, Toulouse, 1992.

[Zuse91] Zuse H., *Software Complexity: Measures and Methods*, Walter de Gruyer, 1991.